Date: June 2011

DDS for Lightweight CCM

Version 1.0

OMG Document Number: formal/2011-06-01
Standard document URL: http://www.omg.org/spec/dds4ccm/1.0
Associated File(s)*: http://www.omg.org/spec/dds4ccm/20100501
http://www.omg.org/spec/dds4ccm/20100501/ccd_dds.idl
http://www.omg.org/spec/dds4ccm/20100501/ccm_dds_dlrl.idl
http://www.omg.org/spec/dds4ccm/20100501/DDS_DefaultQoS.xml
http://www.omg.org/spec/dds4ccm/20100501/DDS_QoSProfile.xsd
http://www.omg.org/spec/dds4ccm/20100501/DDSforCCM-meta.uml

* original file: dtc/2010-05-09

OMG’s Issue Reporting Procedure

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

Table of Contents

Preface

OMG

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

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

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

OMG Specifications

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

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

Specifications within the Catalog are organized by the following categories:

Business Modeling Specifications

Business Strategy, Business Rules and Business Process Management specifications

Middleware Specifications

- CORBA/IIOP
- Minimum CORBA
- CORBA Component Model (CCM)
- Data Distribution Service (DDS)

Specialized CORBA Specifications

Includes CORBA/e and Realtime and Embedded Systems

Language Mappings

- IDL / Language Mapping
- Other Language Mapping specifications

Modeling and Metadata Specifications

- UML®, MOF, XMI, and CWM specifications
- UML Profiles

Modernization Specifications

KDM

Platform Independent Model (PIM), Platform Specific Model (PSM), and Interface Specifications

- OMG Domain
- CORBAServices
- CORBAFacilities
- OMG Embedded Intelligence
- OMG Security

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

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

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

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

1 Scope

CCM (including lightweight CCM[1]) offers as main features

- to make explicit connections between components, and
- to offer a nice architectural pattern to keep separated the business code from the non-functional properties.

This specification deals with the first point, i.e., the supported interactions between components.

In the initial version of CCM the only supported interactions between components were

- synchronous method invocation, and
- events, with no possibility to adjust the behavior of these (e.g., via QoS).

A recent extension has added the support for streams. This specification deals with support for DDS interactions. However, rather than specifying an ad-hoc solution for that support, the specification is made of two parts:

1. A Generic Interaction Support allowing to define new interactions in CCM. This support is made of two constructs:
    - a new port type (namely extended port) to capture as a whole a set of basic interactions that need to be kept consistent (a trivial example is e.g., how to provide message passing with flow control), and
    - abstractions in between components (namely connectors) to support new interaction mechanisms. Those extensions are complementary - extended ports being the declarative part (attached to a component definition), while connectors can be seen as their operative part. It should be noted however that both (extended ports and connectors) can be used in isolation, even if maximum benefit results from their combination. Chapter 7 contains this part of the specification.
2. The specialization of those constructs to define DDS support. This results in the specification of a set of DDS extended ports and connectors. This definition is itself divided in two parts:
    - extended ports and connectors for DDS/DCPS, and
    - extended ports and connectors for DDS/DLRL. Chapters 8 (for DCPS) and 9 (for DLRL) contain this part of the specification.

2 Conformance

The conformance criteria of an implementation with respect to this specification is stated through the support for the following extensions:

- ***A CCM framework claiming conformance with the "Generic Interaction Support***" part of this specification shall support extended ports and connectors:
    - Extensions of IDL3 to support **porttype**, **mirrorport,** and **port** declarations
    - Extension of IDL3 to support parameterized interfaces (template)
    - Extension of D&C PSM for CCM to describe extended ports

1. In the remaining document, CCM will implicitly refer also to Lightweight CCM.

    - Extension of IDL3 to support **connector** declaration
    - Extension of D&C PSM for CCM to deploy and configure **connector** fragments

- ***A CCM framework claiming conformance with this "DDS for Lightweight CCM" specification*** shall, in addition, support DDS-DCPS normative ports and connectors and their configuration.

An optional compliance point for this "DDS for Lightweight CCM" specification is the support for DLRL ports and connectors and their configuration.

3 Normative References

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

- [CORBA] Common Object Request Broker Architecture: Core Specification, OMG, V3.1, part 1, part 2, and part 3 (formal/08-01-05; formal/08-01-07; formal/08-01-06).
- [UML CCM] UML Profile for CORBA & CORBA Components, v1.0 (formal/08-04-07)
- [CCM] CORBA Component Model Specification, v4.0 (formal/06-04-01); CORBA Component Model, v4.0 XML (formal/07-02-02); CORBA Component Model, v4.0 IDL (formal/07-02-01);
- [IDL] Draft CORBA Core 3.0 consisting of CORBA Core 2.6 + Core and Interop RTF 12/2000 Changes + Components FTF Changes (only the changed chapters are in this document) (ptc/02-01-14)
- [QOS4CCM] Quality of Service for CORBA Components (ptc/07-08-14)
- [D&C] Deployment and Configuration of Component-based Distributed Applications, OMG, V4.0 (formal/06-04-02).
- [DDS] Data Distribution Service for Real-time Systems Specification, OMG, V1.2, (formal/07-07-01).
- [XMLSchema] XML Schema,W3C Recommendation, 28 October 2004. Latest version at http://www.w3.org/TR/xmlschema-1/ and http://www.w3.org/TR/xml-schema-2/.

4 Terms and Definitions

In the scope of this specification, the following terms and definitions apply.

- **Connector** – Interaction entity between components. A connector is seen at design level as a connection between components and is composed of several fragments (artifacts) at execution level, to realize the interaction.
- **Extended Port** – Consists of zero or more provided as well as zero or more required interfaces, i.e., closely resembling the UML2 specification of a port.
- **Fragment** – Artifact, part of the connector implementation. A fragment corresponds to one executor that can be deployed onto an execution node, co-localized with one component for which it supports the interaction provided by the connector.

5 Symbols (and abbreviated terms)

The followings acronyms are intensely used in the following specification:

• CCM	CORBA Component Model
• CIF	Component Implementation Framework
• CORBA	Common Object Request Broker Architecture
• DCPS	Data-Centric Publish-Subscribe (part of DDS)
• DDS	Data Distribution Service
• DLRL	Data Local Reconstruction Layer (part of DDS)
• IDL	Interface Definition Language
• UML	Unified Modeling Language
• XML	eXtensible Mark-up Language

6 Additional Information

6.1 Changes to Adopted OMG Specifications

This specification does not impact the existing CCM specification [CCM] on the following items:

- Component Model
- OMG CIDL Syntax and Semantics
- CCM Implementation Framework
- The Container Programming Model
- Integrating with Enterprise Java Bean
- Interface Repository MetaModel
- CIF Metamodel
- Lightweight CCM profile

6.2 Extensions

Nevertheless, for a CCM implementation conformant to this specification, extensions to [CCM] are provided for:

- Component Model level to support new keywords **porttype, port, mirrorport,** and **connector**.
- CIF MetaModel defined in [UML CCM] with the addition of **ExtendedPortType, ExtendedPortDef, ConnectorDef.**

- D&C PSM for CCM where 2 classes are added for the support of connectors: **ConnectorPackageDescription** and **ConnectorImplementationDescriptor.**

6.3 Changes

The D&C PSM for CCM defined in [D&C] is modified to integrate:

- New CCM port kinds (**ExtendedPort** and **ExtendedMirrorPort**) in the class **CCMComponentPortKind**.
- A **templateParam** attribute in the class **ComponentPortDescription**

6.4 Acknowledgements

The following companies submitted this specification:

- Mercury Computer Systems, Inc.
- PrismTech Group Ltd
- Real-Time Innovations, Inc.
- Thales

The following company supported this specification:

- Commissariat à l'Energie Atomique (CEA)

7 Generic Interaction Support

The Generic Interaction Support (GIS) includes the definition of *extended ports* and *connectors*. Extended ports can be used at component level to specify the programming contracts that the components need to fulfill in order to interact with other components. Connectors are the entities that can be connected to components via these extended ports, in order to actually realize the interactions.

These extensions fall within the scope of adapting CCM model to specialized application domains, in particular embedded and real-time systems. The lightweight CCM specification has de fined a profile to meet embedded equipments. QoS for CCM [QOS4CCM] allows providing non-functional services to components and by this mean allows the use of real-time services plugged into the container. This Generic Interaction Support complements these adaptations with the ability to provide interactions or communication patterns (control of flow, synchronous, asynchronous, shared memory …) very specific to real-time software.

As for non-functional services, connectors can be platform dependent because they deal with specific communication buses (1553, UDP, TCP, direct calls…) or specific semantics (management of buffers, threads, mutex… inside the fragment). For this reason, they are rather intended to be provided by CCM framework providers or platform providers.

7.1 Simple Generic Interaction Support

7.1.1 Overview

The GIS relies on two constructs: extended ports and connectors. Extensions to IDL3[1] are provided to allow defining and using those constructs.

IDL3+ declarations can be easily translated in plain IDL3. The following figure presents the steps of component definition. Only the first step is new and will be detailed in the following sections.

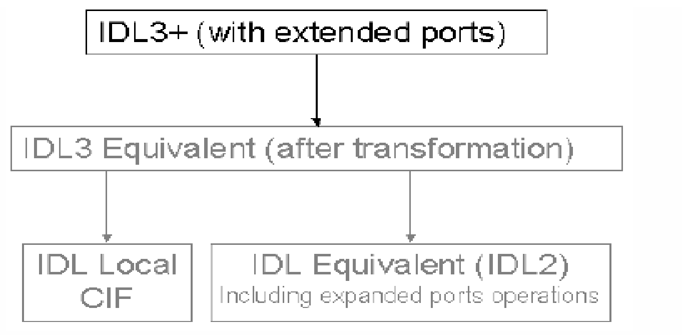

IDL3+ (with extended ports)
IDL3 Equivalent (after transformation)
IDL Local CIF
IDL Equivalent (IDL2)
Including expanded ports operations

Figure 7.1 - IDL3+ Transformation

As resulting IDL3 is exactly as before, the rest of the transformation is kept unchanged.

The transformation from IDL3+ to equivalent IDL3 shall be done by a tool part of a CCM framework implementing the current specification.

1. IDL3 plus its extensions is called IDL3+ in this specification.

7.1.2 Extended Ports

An extended port is the means to represent the programming contract that the components need to deal with, in order to interact according to the corresponding interaction pattern. A programming contract can always be expressed by means of interfaces to call and/or interfaces to be called. Extended ports can be thus subsumed in a group of provided/required interfaces, which can be used/provided. In other words, extended ports are just groups of single CCM ports (facet/**provides** and receptacle/**uses**)[2] and attributes to configure them if needed.

7.1.2.1 IDL3+ Representation

A new keyword **porttype** has been added to IDL3[3] to allow defining extended ports. An extended port definition consists of a list of basic ports (**uses** and/or **provides**) and attributes.

A second new keyword **port** allows setting a previously defined extended port to a component. The following is an example of such definitions.

```
//--------------
// IDL3+
//--------------

interface Data_Pusher {
    void push(in Data dat);
    };

interface FlowControl {
    void suspend ();
    void resume();
    readonly attribute nb_waiting;
    };

// Extended port definition
porttype Data_ControlledConsumer {
    provides Data_Pusher            consumer;
    uses FlowControl                control;
    };

// Component declaration with that port
component C1 {
    port Data_ControlledConsumer p;
    };
```

In the original CCM, existing port kinds are seen as groups of matching basic ports (provided/required interfaces, or events sinks/sources). Similarly, it is needed to define inverses of extended ports (i.e., the ones that will "match" them). To avoid duplicated definitions, the keyword **mirrorport** has been introduced for that purpose. A **mirrorport** results in exactly the same number of simple ports as the **port** of the same **porttype**, except that all the **uses** are turned into **provides** and vice-versa.

2. The receptacles correspond to the interfaces that the components will call and the facets, the ones that they will provide to be called.
3. In this section and the following, the new syntax is just introduced. Formal definition of the new grammar is in Section 7.3, "IDL3+ Grammar," on page 13.

7.1.2.2 Translation from Extended Ports to Basic Ports

The extensions provided to IDL3 with **porttype**, **port**, and **mirrorport** keywords can be directly mapped to usual IDL3 constructs (basic port declarations).

The rules for this transformation are as follows:

- A **provides** in a **port** becomes a **provides** in the equivalent IDL3 declaration of the component.
- A **uses** in a **port** becomes a **uses** in the equivalent IDL3 declaration of the component.
- A **provides** in a **mirrorport** becomes a **uses** in the equivalent IDL3 declaration of the component.
- A **uses** in a **mirrorport** becomes a **provides** in the equivalent IDL3 declaration of the component.
- The name of the basic port is the concatenation of the extended port name and the related basic port name of the **porttype**, separated by '_'.
- An **attribute** in a port or a **mirrorport** becomes an **attribute** in the equivalent IDL3 declaration of the component.
- The name of the generated attribute is the concatenation of the extended port name and the related attribute name of the **porttype**, separated by '_'.

Applying these rules, the previous example will result in the following IDL3 declaration:

```
// Resulting IDL3 component definition
component C1 {
    provides Data_Pusher          p_consumer;
    uses FlowControl              p_control;
    };
```

7.1.3 Connectors

Connectors are used to specify an interaction mechanism between components. Connectors can have ports in the same way as components. They can be composed of simple ports (CCM **provides** and **uses**) or extended ports[4].

The following figure shows a connector as it can be represented at design level:

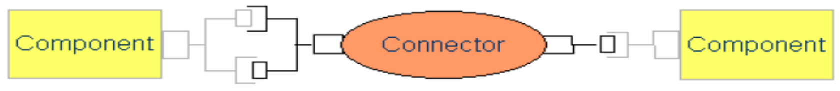


Figure 7.2 - Logical View of a Connector

The connector will concretely be composed of several parts (called fragments) that will consist of executors, each in charge of realizing a part of the interaction. Each fragment will be co-localized to the component using them.

By default, for each port, a fragment (an executor) is produced. If several ports are always co-localized because it corresponds to the semantic of the connector, their behavior can be provided by the same fragment. This is an implementation choice for the connector developer.

4. As generally components will be given extended ports by means of keyword **port**, it is very likely that connectors will use **mirrorport** instead.

The following figure shows the connector with its fragments at execution time:

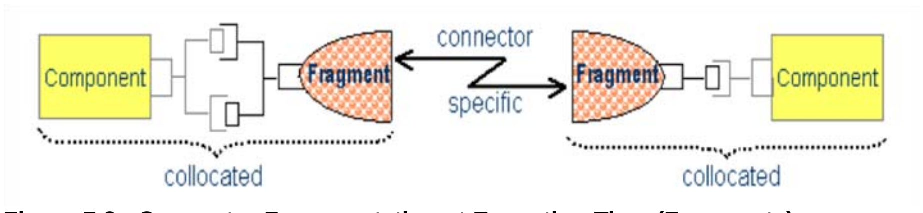


Figure 7.3 - Connector Representation at Execution Time (Fragments)

The communication mechanism between the fragments is connector specific and will be addressed only for DDS support in this specification.

The connector concept brings another way of seeing CCM: connectors are used to provide interaction (in particular communication support) between components, and are realized via fragments collocated with the concerned components. This contrasts with the classical approach, which entails CORBA servants for facets typically provided by code generation and encapsulating the component executors. An implementation compliant with the present connector specification is not required to provide CORBA servants and CORBA object references for the component facets.

7.1.3.1 IDL3+ Representation

The new IDL3 keyword **connector** allows defining connectors. A connector definition is very similar to a simplified component's one as a connector is just meant to gather ports (simple or extended). It thus cannot include the **support** keyword.

The following is an example of a connector definition:

```
connector Data_Cnx {
    mirrorport Data_ControlledConsumer        cc;
    provides        Data_Pusher                p;
    };
```

7.1.3.2 Connector Attributes

A connector can declare attributes in the same way as components. Attributes are declared at connector definition level and are reflected in each fragment at realization level. For instance in a DDS connector, the topic can be seen as an attribute and the value of the topic is reflected on each fragment that composes the connector: each fragment of the connector will work on the same topic.

7.1.3.3 Connector Inheritance

A connector can inherit from another connector. It means that the new connector is composed of all the ports and attributes of the inherited connector in addition to all the ones that are locally defined.

The syntax used to declare a connector inheritance is similar to the one used to declare a component inheritance.

7.1.3.4 Composite Connectors

A connector (type) can have multiple implementations. As is the case for components, such an implementation may be an assembly of other components. For example, an implementation of a local FIFO queue can be provided by a monolithic implementation, but if this FIFO should enable distribution, an alternative implementation needs to provide multiple fragments co-localized with the components using them. These fragments can be considered as sub-components within an assembly (parts within UML composite structures), i.e., an implementation of a connector with multiple fragments is an assembly implementation. There is no restriction on the level of assembly implementations, for instance a fragment might itself be realized by an assembly implementation. The advantage of assembly implementations is twofold:

1. They enable expressing the fragmented implementation of connectors by concepts already existing in CCM.
2. Assembly implementations enable the composition of connectors, which facilitates the development of new connectors.

Consider the example of a remote FIFO. One possible implementation is a FIFO on the consumer's site and a remote access. The structure of such a remote FIFO implementation is shown in Figure 7.4. It is composed of two fragments called respectively SocketClient and FIFO_Socket_f_pull.

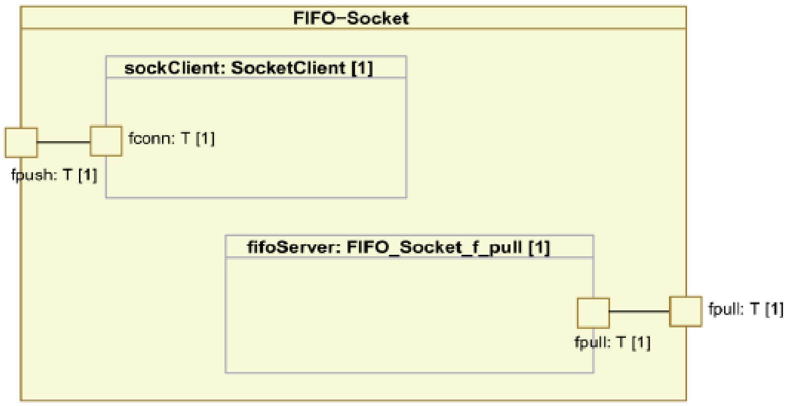

FIFO–Socket
sockClient: SocketClient [1]
fconn: T [1]
fpush: T [1]
fifoServer: FIFO_Socket_f_pull [1]
fpull: T [1]
fpull: T [1]

Figure 7.4 - Example of a Distributed FIFO Implementation

Figure 7.5 shows the detailed implementation of the second fragment (FIFO_Socket_f_pull) which is itself an assembly of 2 fragments: SocketServer and ConnFIFO.

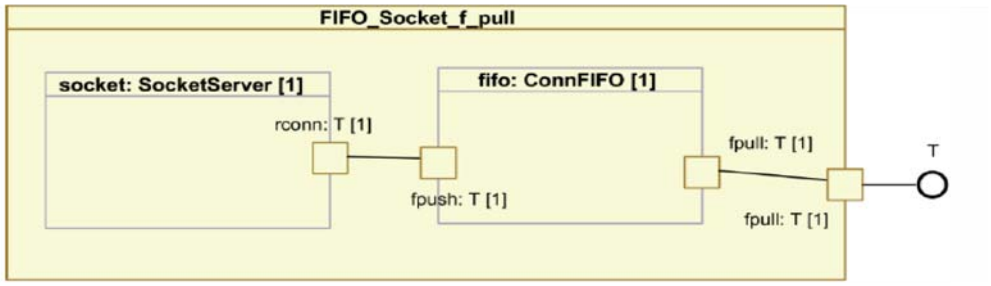


Figure 7.5 - Assembly Implementation of a Connector Fragment

It is thus possible to create new connector implementations by re-assembling existing connectors or fragment implementations. In case of the example, the socket could be replaced by another transport mechanism, for instance an inter process communication.

7.1.3.5 Translation to IDL3

The mapping of connector definitions to standard IDL3 is trivial: The connector definitions are removed, but the information is used to provide type information at the assembly level. The information shall be described at assembly level to check whether the binding of two ports from a component and a connector, respectively binds identical provided and used types or vice versa.

In a CCM framework providing the support for connector extension, the connector definition in “IDL3+” can be used to generate partly the fragment executors where the connector implementation will be realized (see Section 7.4, “Programming Model for Connectors,” on page 19 for more details on connectors realization).

7.2 Generic Interaction Support with Templates

As extended ports and connectors are meant to capture interaction logics, their main benefit is obtained if they can be parameterized by types. In the above example, the port Data_ControlledConsumer and the connector Data_Cnx are only valid for manipulating elements of specific type Data**.** It would be very useful to define generic port type and connector that would provide similar interaction logics to any type of data.

For that purpose, an extension allowing parameterizing definitions of interfaces, ports, and connectors has been added to IDL3+. Parameterized definitions can easily be resolved at IDL3+ compilation time.

7.2.1 Template Support Overview

The template support is aiming at integrating smoothly in the current IDL specification. It follows therefore the current rules that apply to existing predefined templates (such as sequences): the syntax is very lightweight and anonymous types are not allowed.

The template support allows defining all sensible parameterized interaction support.

Almost any sensible parameterized interaction support will associate at least one port type (itself comprising at least one parameterized interface) with one connector. *The component*s that integrate the concrete port (resulting from the instantiation of the port type with a given parameter type) *and the related concrete connector* that provides the mirror port of the same port type *need eventually to use/provide exactly the same interface instantiation*.

If the port type on one hand and the connector on the other hand were placed in separate template definitions, this constraint would not be achievable due to anonymous types not being allowed. There is therefore a necessity to offer the means to group several identically parameterized definitions in the same template scope. Modules are the only IDL grouping constructs. *Therefore the template support is introduced at the module level.*

Note that as parameterizing a module will result in de facto parameterization of all the embedded constructs, this support offers a lot of possibilities despite its limited impact on the IDL grammar.

7.2.2 Template Modules

Using template modules is a two-step process:

1. The template module is declared.
2. Its instantiation results in a concrete module that is usable as any module.

In addition, a template module can be referenced inside another similarly parameterized module, in order to reuse the related definitions.

7.2.2.1 Template Module Declaration

A template module is declared in adding to its declaration a list of comma-separated formal parameters embedded between angular brackets (**<** and **>**).

Formal parameters associate a type constraint and the formal parameter name. At instantiation time each formal parameter will be substituted by a concrete value. Only the concrete values that comply with their formal parameter type constraints will be accepted.

Type constraints can be:

- **typename**, meaning that any type will be acceptable.
- Some more restricted type designators:
    - **interface**, meaning all interfaces
    - **valuetype**, meaning all valuetype types
    - **eventtype**, meaning all eventtypes
    - **struct**, meaning all struct types
    - **union**, meaning all union types
    - **sequence**, meaning all sequence types
    - **enum**, meaning all enum types
- A **const** primitive type, meaning that any constant of the required primitive type will be acceptable.
- A sequence specification, with the constraints that its formal parameters must appear previously in the formal parameters list of the module. In this case, the passed parameter should be a sequence complying with the sequence specification[5].

The following is a refactoring of the previous example, which has been generalized in order to be usable with any data type.

```
interface FlowControl {
    void suspend ();
    void resume();
    readonly attribute nb_waiting;
    };

module Flow <typename T> {
    interface Pusher {
        void push(in T dat);
        };

    // Extended port definition
    porttype ControlledConsumer {
        provides Pusher              consumer;
```

5. This disposal is useful to pass, without duplication, an existing sequence type. Actually, the removal of anonymous types from IDL leads to each similar sequence instantiation be a different type. In case the interaction support needs to manipulate sequence<T> (T being the formal parameter of the template), then there is no means to use the same sequence as the rest of the application but to pass it as a formal parameter.

```
        uses FlowControl                 control;
        };

    // Connector
    connector Cnx {
        mirrorport  ControlledConsumer        cc;
        provides Pusher                       p;
        };
    };
```

Note that all constructs that are not T-dependent (here the FlowControl interface) have been put outside the template module to avoid useless duplications. Note also that T-dependency of a construct may be direct, because the formal type is used in the definition (here the Pusher interface) or indirect when the definition makes use of a T-dependent construct (here the ControlledConsumer port type or the Cnx connector).

7.2.2.2 Template Module Instantiation

Once defined a template module has to be explicitly instantiated before being used. Instantiation consists in providing actual values to any formal parameters and a name to the resulting concrete module.

This is done by declaring the concrete module with a new form of the module declaration that inserts, between the keyword module and the module name, the template module instantiation with all values for formal parameters enclosed in angular brackets.

When the module is instantiated, all the embedded constructs are de facto instantiated with the proper parameters values.

The following is an example of instantiating the previously defined template module with the data type Data and using the port type in a component.

```
// module instantiation
module Flow<Data> Data_Flow;

// component declaration
component C2 {
    port Data_Flow::ControlledConsumer p;
    };
```

Applying the IDL3+ to IDL3 translation rules will give the following result:

```
// Resulting IDL3 component definition
component C2 {
    provides Data_Flow::Pusher        p_consumer;
    uses FlowControl                  p_control;
    };
```

7.3 IDL3+ Grammar

The following description of IDL grammar extensions uses the same syntax notation that is used to describe OMG IDL in CORBA Core, IDL Syntax and Semantics clause. For reference, the following table lists the symbols used in this format and their meaning.

Table 7.1 - IDL EBNF Notation

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Nonterminal
"text"	Literal
*	The preceding syntactic unit can be repeated zero or more times
+	The preceding syntactic unit can be repeated one or more times
{}	The enclosed syntactic units are grouped as a single syntactic unit
[]	The enclosed syntactic unit is optional—may occur zero or one time

7.3.1 Summary of IDL Grammar Extensions

The following table gathers all the new grammar rules supporting this specification. Those rules aim at completing the existing IDL grammar ("OMG IDL Syntax and Semantics" [IDL]).

The items that are in ***italics-blue*** are already described in the existing IDL grammar. When they appear here in the right part of a rule, they are considered as terminals. When they appear in the left part of a rule, they are extended by this specification.

Table 7.2 - IDL3+ Grammar Extensions

<definition> **:**	**:=** ***<type_dcl>*** **";"**
	| ***<const_dcl>*** **";"**
	| ***<except_dcl>*** **";"**
	| ***<interface>*** **";"**
	| ***<module>*** **";"**
	| ***<value>*** **";"**
	| ***<type_id_dcl>*** **";"**
	| ***<type_prefix_dcl>*** **";"**
	| ***<event>*** **";"**
	| ***<component>*** **";"**
	| ***<home_dcl>*** **";"**
	| <porttype_dcl> ";"
	| <connector> ";"
	| <template_module> ";"
	| <template_module_inst> ";"
<porttype_dcl> :	**:= "porttype"** ***<identifier>*** **"{" <port_export>+ "}"**
<port_export>:	**:=** ***<provides_dcl>*** **";"**
	| ***<uses_dcl>*** **";"**
	| ***<attr_dcl>*** **";"**
<port_dcl> :	**:= {"port" | "mirrorport" }** ***<scoped_name>*** **<identifier>**

```
<component_export>:             := <provides_dcl> ";"
                                | <uses_dcl> ";"
                                | <emits_dcl> ";"
                                | <publishes_dcl> ";"
                                | <consumes_dcl> ";"
                                | <port_dcl> ";"
                                | <attr_dcl> ";"
<connector> :                   := <connector_header> "{" <connector_export>* "}"
    <connector_header> :        := "connector" <identifier> [ <connector_inherit_spec> ]
<connector_inherit_spec> :      := ":" <scoped_name>
    <connector_export> :        := <provides_dcl> ";"
                                | <uses_dcl> ";"
                                | <port_dcl> ";"
                                | <attr_dcl> ";"
<template_module> :             := "module" <identifier> "<" <formal_parameters> ">"   "{" <tpl_definition>* "}"
<formal_parameters>:            := <formal_parameter> {"," <formal_parameter>}*<formal_parameter>::=
<formal_parameter_type> <identifier>
<formal_parameter_type>:        := "typename"
                                | "interface"| "valuetype"|  "eventtype"
                                | "struct"| "union"| "exception"| "enum"| "sequence"
                                | "const" <const_type>
                                | <sequence_type>
<tpl_definition> :              :=  <type_dcl> ";"
                                | <const_dcl> ";"
                                | <except_dcl> ";"
                                | <interface> ";"
                                | <fixed_module> ";"
                                | <value> ";"
                                | <type_id_dcl> ";"
                                | <type_prefix_dcl> ";"
                                | <event> ";"
                                | <component> ";"
                                | <home_dcl> ";"
                                | <porttype_dcl> ";"
                                | <connector> ";"
                                | <template_module_ref> ";"
        <fixed_module>:         := "module" <identifier>    "{" <fixed_definition>* "}"
        <fixed_definition>:     := <type_dcl> ";"
                                | <const_dcl> ";"
                                | <except_dcl> ";"
                                | <interface> ";"
                                | <fixed_module> ";"
                                | <value> ";"
                                | <type_id_dcl> ";"
                                | <type_prefix_dcl> ";"
                                | <event> ";"
                                | <component> ";"
                                | <home_dcl> ";"
                                | <porttype_dcl> ";"
                                | <connector> ";"
        <template_module_inst>:         := "module" <scoped_name> "<" <actual_parameters> ">" <identifier> ";"
        <actual_parameters>:            := <actual_parameter>{"," <actual_parameter>}*
```

```
<actual_parameter>:            := <type_spec>
                               | <const_exp>
<template_module_ref>:         := "alias" <scoped_name> "<" <formal_parameter_names> ">"  <identifier>
<formal_parameter_names>::= <identifier> {"," <identifier>}*
```

Those rules are detailed in the following sections.

7.3.2 New First-Level Constructs

The first rule extends the existing **<definition>** with the new first-level constructs that can be used natively or inside a module, namely:

- port type declarations,
- connector declarations,
- template module declarations, and
- template module instantiations.

Those new constructs are detailed in the following sections.

```
<definition> :    :=  <type_dcl> ";"
                  | <const_dcl> ";"
                  | <except_dcl> ";"
                  | <interface> ";"
                  | <module> ";"
                  | <value> ";"
                  | <type_id_dcl> ";"
                  | <type_prefix_dcl> ";"
                  | <event> ";"
                  | <component> ";"
                  | <home_dcl> ";"
                  | <porttype_dcl> ";"
                  | <connector> ";"
                  | <template_module> ";"
                  | <template_module_inst> ";"
```

7.3.3 IDL Extensions for Extended Ports

7.3.3.1 Port Type Declarations

The following rules allow port type declarations:

```
<porttype_dcl> :      := "porttype" <identifier> "{" <port_export>+  "}"
<port_export>:        :=
<provides_dcl> ";"
                      | <uses_dcl> ";"
                      | <attr_dcl> ";"
```

A port type declaration is made of:

- the **porttype** keyword,
- an identifier for the port type name,

- the list of provided and/or used basic ports and attributes that constitutes the extended port.

7.3.3.2 Extended Port Declarations

The following rules allow port declarations:

```
<port_dcl> :        := {"port" | "mirrorport" } <scoped_name> <identifier>
<component_export>::= <provides_dcl> ";"
                    | <uses_dcl> ";"
                    | <emits_dcl> ";"
                    | <publishes_dcl> ";"
                    | <consumes_dcl> ";"
                    | <port_dcl> ";"
                    | <attr_dcl> ";"
```

An extended port declaration comprises:

- the **port** or **mirrorport** keyword,
- the name of a previously defined port type, and
- the identifier for the port.

The existing **<component_export>** is modified so that such a port declaration can be used to add an extended port to a component.

7.3.4 IDL Extensions for Connectors

The following rules allow connector declarations:

```
<connector> :                := <connector_header> "{" <connector_export>* "}"
<connector_header> :         := "connector"
<identifier> [ <connector_inherit_spec> ]
<connector_inherit_spec> :   := ":"
<scoped_name>
<connector_export> :         := <provides_dcl> ";"
                             | <uses_dcl> ";"
                             | <port_dcl> ";"
                             | <attr_dcl> ";"
```

A connector is defined by its header and its body.

A connector header comprises:

- the keyword **connector**,
- an identifier for the connector, and
- an optional inheritance specification, consisting of a colon and a single scoped name that must denote a previously-defined connector.

A connector body may comprise:

- facet declarations,
- receptacle declarations,

- extended port declarations,
- attribute declarations.

7.3.5 IDL Extensions for Template Modules

7.3.5.1 Template Module Declarations

The following rules allow template module declarations:

```
<template_module> :          := “module” <identifier> “<” <formal_parameters> “>”   “{“ <tpl_definition>+ “}”
<formal_parameters>:         := <formal_parameter> {“,” <formal_parameter>}*
<formal_parameter>:          := <formal_parameter_type>
<identifier>
<formal_parameter_type>:     := “typename”
                             | “interface”| “valuetype”|  “eventtype”
                             | “struct”| “union”| “exception”| “enum”| “sequence”
                             | ”const” <const_type>
                             | <sequence_type>
<tpl_definition> :           :=  <type_dcl> “;”
                             | <const_dcl> “;”
                             | <except_dcl> “;”
                             | <interface> “;”
                             | <fixed_module> “;”
                             | <value> “;”
                             | <type_id_dcl> “;”
                             | <type_prefix_dcl> “;”
                             | <event> “;”
                             | <component> “;”
                             | <home_dcl> “;”
                             | <porttype_dcl> “;”
                             | <connector> “;”
                             | <template_module_ref> “;”
<fixed_module>:              := “module”
<identifier>    “{“ <fixed_definition>+“}”
<fixed_definition>:          := <type_dcl> “;”
                             | <const_dcl> “;”
                             | <except_dcl> “;”
                             | <interface> “;”
                             | <fixed_module> “;”
                             | <value> “;”
                             | <type_id_dcl> “;”
                             | <type_prefix_dcl> “;”
                             | <event> “;”
                             | <component> “;”
                             | <home_dcl> “;”
                             | <porttype_dcl> “;”
                             | <connector> “;”
```

A template module specification comprises:

- The **module** keyword.
- An identifier for the module name.

- The specification of the template parameters between angular brackets, each of those template parameters consisting of:
  - a type classifier, which can be:
    - **typename** to indicate that any valid type can be passed as parameter;
    - **interface**, **valuetype**, **eventtype**, **struct**, **union**, **exception**, **enum**, **sequence** to indicate that a more restricted type must be passed as parameter;
    - a constant type to indicate that a constant of that type must be passed as parameter;
    - a sequence type declaration to indicate that a compliant sequence type must be passed as parameter. (The formal parameters of that sequence must appear previously in the module list of formal parameters.)
  - an identifier for the formal parameter.
- The module body that may contain declarations for port types and/or connectors, other template module references, as well as all that previously made a classical module body (that last part is named **<fixed_module>** in the grammar)[6].

A template module cannot be re-opened (as opposed to a classical one).

7.3.5.2 Template Module Instantiations

The following rules allow template module instantiations:

<template_module_inst>: **:= "module"** ***<scoped_name>*** **"<" <actual_parameters> ">"** ***<identifier>*** **";"**
<actual_parameters>: **:= <actual_parameter>{"," <actual_parameter>}***
<actual_parameter>: **:=**
<type_spec>
| ***<const_exp>***

A module template instantiation consists of providing values to the template parameters and a name to the resulting module. Once instantiated, the module is exactly as a classical module.

The provided values must fit with the parameter specification as described in the previous section. In particular, if the template parameter is of type "sequence type declaration," then an instantiated compliant sequence must be passed.

7.3.5.3 References to a Template Module

The following rules allow referencing template modules:

<template_module_ref>: **:= "alias"** ***<scoped_name>*** **"<" <formal_parameter_names> ">"** ***<identifier>***
<formal_parameter_names>::= ***<identifier>*** **{","** ***<identifier>*****}***

An alias directive allows reference to an existing template module inside a template module definition.

This directive allows providing an alias name (which can be identical to the template module name) and the list of formal parameters to be used for the referenced module instantiation. Note that that list must be a subset of the formal parameters of the embedding module.

When the embedding module will be instantiated, then the referenced module will be instantiated in the scope of the embedding one (i.e., as a submodule).

6. Note that this implies that a template module cannot contain another template module.

7.3.6 Summary of New IDL Keywords

The following table gathers all new keywords introduced by this specification.

Table 7.3 - New IDL Keywords

alias	**connector**	**mirrorport**	**port**	**porttype**	**typename**

As all IDL keywords, they are now reserved and thus may not be used otherwise, unless escaped with a leading underscore.

7.4 Programming Model for Connectors

This section presents the rules a connector implementer has to follow. This is the counterpart of the CCM component model interfaces for connectors and connector ports. As presented in Section 7.4.6, "Connector Implementation Interfaces," on page 21, connectors' implementations consist in the collaboration of several objects, named fragments, or connector executors. They realize the implementation of the connector ports and are collocated with the components logically connected to the connector. The programming model is oriented towards the provision of objects corresponding to the ports of the connector under consideration. It is therefore composed of an API for fragments programming and fragments bootstrapping.

The following are the interfaces necessary to implement a connector's fragment:

```
module Components {
    interface CCMObject :Navigation,
                Receptacles,
                Events{
        CCMHome get_ccm_home();
        void configuration_complete() raises ( InvalidConfiguration );
        void remove () raises ( RemoveFailure );
        };

    interface KeylessCCMHome {
        CCMObject create_component() raises ( CreateFailure );
        };

    interface CCMHome {
        void remove_component() raises ( RemoveFailure );
        };

    interface HomeConfiguration : CCMHome {
        void set_configuration_values( in ConfigValues config );
        };
};
```

This presents the interfaces that need to be implemented by a connector provider. A **Components::CCMObject** interface has to be implemented for each identified fragment of the connector.

This set of interfaces is a subset of the component model coming from the Lightweight CCM specification. All the previous methods declared in interfaces have to be defined in the fragment implementation to conform to all D&C deployment tools.

In a fragment's implementation, some of these interfaces could be left empty, others are mandatory, among them: **Navigation**, **Receptacles**, and **KeylessCCMHome**.

Note that the **Events** CCM interface is never used in connectors' executors. The reason is that, as the component and its connector's fragment are collocated, they only interact via synchronous calls (a potential asynchronous nature of the actual interaction between components would be provided by connector's fragments themselves).

7.4.1 Interface CCMObject

Given a **porttype**, the fragment inheriting **CCMObject** has to implement all necessary operations (**provide_facet, connect, disconnect**…) inherited from **Components::Navigation** and **Components:: Receptacles** interfaces in accordance with the **porttype**.

7.4.2 configuration_complete

This operation, similarly to components, will be called by the **Application::start** operation [D&C]. This operation is necessary to activate the handshake between connector fragments at deployment time, after the configuration of all components and connector fragments.

7.4.2.1 get_ccm_home

This operation, similarly to components, returns a **CCMHome** reference to the home that manages this component.

7.4.2.2 remove

This operation, similarly to components, is used to delete a fragment. Application failures during remove raise the **RemoveFailure** exception.

7.4.3 Interface KeylessCCMHome

This interface merely implements a bootstrapping facility to create connector fragment instances. The same interface is used by the components. As for components, an entry point allowing the container to create a connector's home instance is defined and is of type:

```
extern "C" { Components::HomeExecutorBase_ptr  (*)(); }   // in C++

Components_HomeExecutorBase*  (*)();                      // in C
```

7.4.3.1 create_component

This operation is called to create the connector fragment during deployment.

7.4.4 Interface HomeConfiguration

7.4.4.1 set_configuration_values

As for components, this operation establishes an attribute configuration for the target fragment object, as an instance of **Components::ConfigValues**. Factory operations on the home of fragment will apply this configurator to newly-created instances.

7.4.5 Equivalent IDL (w.r.t Equivalent IDL section in CCM)

The connector extension does not need to specify equivalent IDL interfaces deduced from ports since only generic operations inherited from **Navigation** and **Receptacles** are mandatory in the lightweight CCM profile, which is addressed by this specification.

If necessary for a connector, the rules to obtain equivalent interfaces are the same as for a component.

7.4.6 Connector Implementation Interfaces

This section explains how connector fragments can be implemented.

The CCM provides a standardized Component Implementation Framework (CIF) defining the programming model for constructing component implementations.

The connector implementation (implementation of several fragments) is specific to the semantic it defines; it can be dependant of the underlying platform and is connector provider specific. For that reason, there is no need to standardize a counter part of the CIF for connectors.

As explained before, the implementation of a fragment inherits the **Components::CCMObject** interface and shall implement the specified operations of **Navigation**, **Receptacles**. This is mandatory to provide a connector that can be deployed and configured with lwCCM deployment framework (compliant to D&C specification [D&C]). This implementation corresponds to a classical implementation of IDL interfaces using the standard language mapping.

As for components, the skeleton of connector fragments can be partly generated taking into account the transformation rules defined in the connector definition. This is fully the responsibility of the connector framework provider.

7.5 Connector Deployment and Configuration

This section introduces all the modifications to the OMG Deployment and Configuration (D&C) specification considered as necessary in order to deal with the packaging and deployment of connectors. The extensions are to be added in the PSM for CCM part of D&C reference in the following specification [CCM].

Remark: this section and the following are based on the D&C specification [D&C]; all conventions defined in this specification are applicable:

- In particular, standard attributes (e.g., label) have the semantics defined in the D&C specification.
- All classes that are not explicitly defined in this document are taken from the specification.
- In the UML diagrams, when no multiplicity is indicated on an association end, the multiplicity is one.

Note that extended ports and connectors (considered as CCM extensions) defined in the previous sections, as an extension of IDL3 have no impact on the D&C PIM; it will only impact the PSM for CCM level [CCM].

7.5.1 Integration of Connectors in D&C

As said before, the objective of this specification is to provide new interaction modes for component-based applications. To achieve this goal, it shall not add complexity for the assembly of components. For this reason, the connectors in a component-based application design shall be seen as an interaction element that links two components and not as a new functional entity that will imply multiplication of connections at assembly level. Nevertheless it implies some modifications to the D&C Component Data Model at assembly level where connections will include connector information.

On the contrary, at Execution Data Model level, since the deployment plan aims to be [automatically] produced at planning phase by tools, and since it is a *flattened assembly*, the connector defined in the connection elements of the assembly will appear as artifacts that have to be deployed by the deployment tools. This implies that the fragment instances (artifacts) are described in the deployment plan with their configuration values; and that connections between components and their related fragment are basic connections (facet / receptacles).

7.5.2 Component Data Model

A connector is an entity very similar to a component. It is packaged, deployed, and owns implementation(s), as well as interfaces, etc. Therefore, it would not have been relevant to define a completely new data model for connectors.

7.5.2.1 Connector Description

Connectors may be packaged in the same way components are, thus most of the elements defined in the component data model are relevant in the case of connectors. However, component packages and connector packages shall be distinguishable; therefore, a **ConnectorPackageDescription** class is defined.

Like a component package, a connector package owns descriptive information (interface description) and one or more implementation(s).

As far as the interface description is concerned, no differences exist between components and connectors, thus the **ComponentInterfaceDescription** class is used for connectors as well and is extended at PSM level to integrate extended port specificities.

In the following, all diagrams of the component data model impacted by the above statements are displayed.

The following figure displays the additions[7] that are to be made to the Component Data Model at PSM for CCM level. Actually, two classes are added: **ConnectorPackageDescription**[8] and **ConnectorImplementationDescription**.

7. Note that this diagram displays only the two classes that have to be added, along with their relations to already existing classes. All the classes originally defined in the specification are, even if not represented here, left intact, as well as their relations.
8. The association between `ComponentUsageDescription` and `ConnectorPackageConfiguration` is in mutual exclusion with those defined in the initial component data model between `ComponentUsageDescription` and `ComponentPackageDescription`.

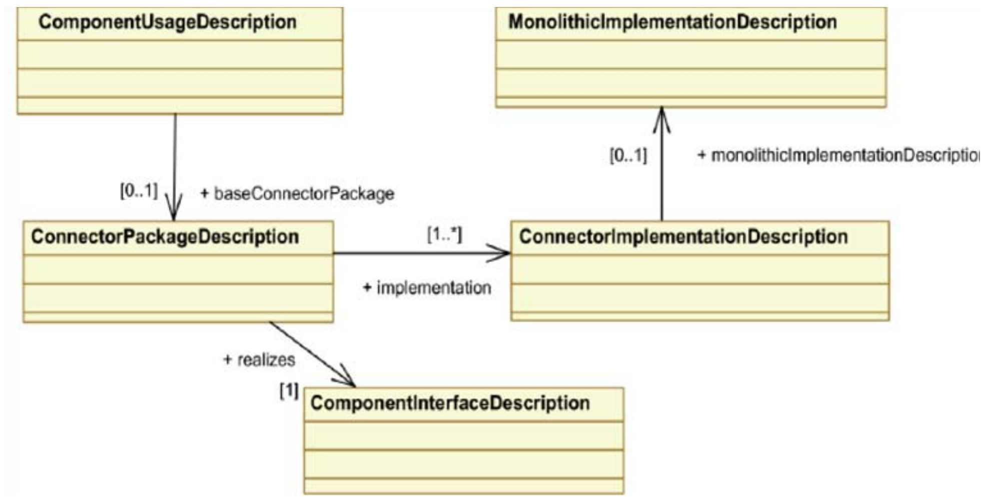

ComponentUsageDescription
MonolithicImplementationDescription
[0..1]
+ monolithicImplementationDescriptio
[0..1]
+ baseConnectorPackage
ConnectorPackageDescription
[1..*]
ConnectorImplementationDescription
+ implementation
+ realizes
[1]
ComponentInterfaceDescription

Figure 7.6 - Revised component data model overview

The two following figures (Figure 7.7 and Figure 7.8) give a detailed description of ConnectorPackageDescription and ConnectorImplementationDescription classes.

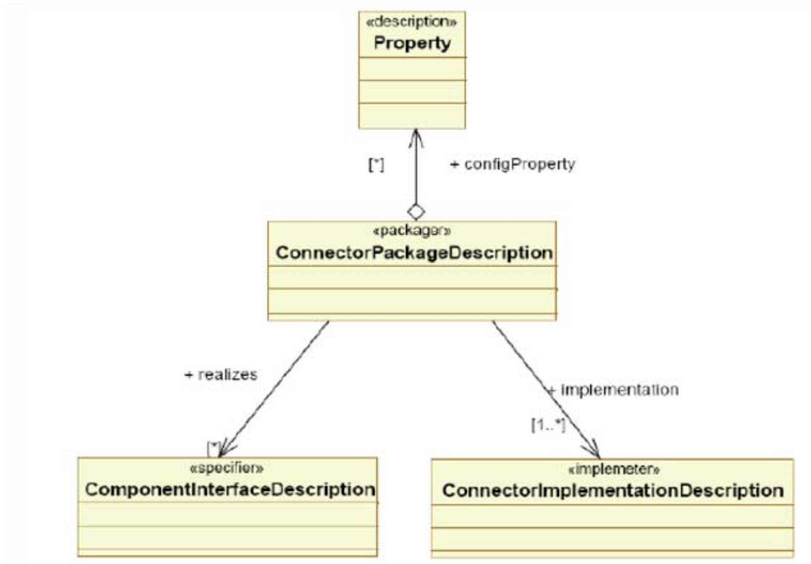


Figure 7.7 - ConnectorPackageDescription Class

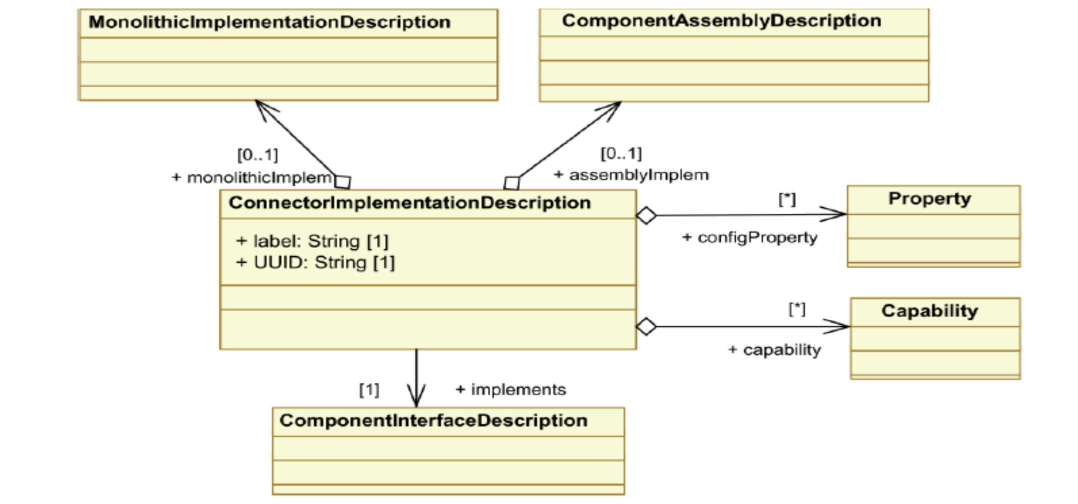


Figure 7.8 - ConnectorImplementationDescription Class

7.5.2.2 ConnectorPackageDescription

A ConnectorPackageDescription describes multiple alternative implementations of the same connector. It references the interface description for the connector and contains a number of configuration properties to configure the running connector (which may override implementation-defined properties and which may be overridden by a PackageConfiguration). These configuration properties enable the packager to define default values for a connector's properties regardless of which implementation for that component is chosen at deployment (planning) time.

7.5.2.3 ConnectorImplementationDescription

A ConnectorImplementationDescription describes a specific implementation of a connector. This implementation can be only monolithic. The ConnectorImplementationDescription may contain configuration properties that are used to configure each connector fragments instance ("default values"). Implementations may be tagged with user-defined capabilities. Administrators can then select among implementations using selection requirements in a PackageConfiguration.

The ComponentInterfaceDescription class is used to describe components and connectors. This description contains information on the ports of components and connectors.

ComponentPortDescription class shall be extended to support the extended ports. As explained in previous sections, extended ports are defined at least by their specific types (specificType member of ComponentPortDescription) but they can also be parameterized by several template parameters. The class is therefore extended with a templateParam member. The kind of port shall also support extended ports and inverse ports. The CCMComponentPortKind enumeration is extended with two values: ExtendedPort, ExtendedMirrorPort.

7.5.2.4 ComponentInterfaceDescription

The added ComponentPortDescription::templateParam (String [0..*]) contains all the template parameter types needed to parameterize the port (if extended). This member is null if the port is simple or if it is an extended port without template. If templateParam contains values, the kind attribute shall be ExtendedPort or ExtendedMirrorPort.

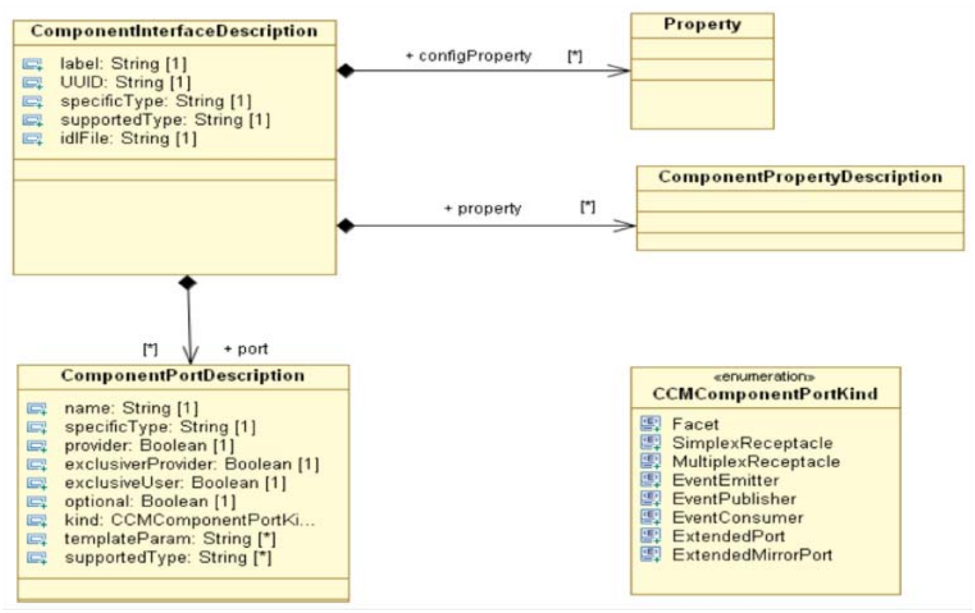


Figure 7.9 - Support for Extended Ports

The connector description will be part of .ccd files.

7.5.2.5 Component Assembly with Connectors

At D&C assembly level, using a connector shall result in a set of connections between components and shall not appear as a new component instance in the assembly.

In the D&C specification, the ComponentAssemblyDescription element contains information about subcomponent instances (SubcomponentInstantiationDescription), connections among ports (AssemblyConnectionDescription), and about the mapping of the assembly's properties (i.e., of the component that the assembly is implementing) to properties of its subcomponents.

Connectors at assembly level are considered as particular connections. It means that the AssemblyConnectionDescription needs to be extended to support connector descriptions. At PSM for CCM level, the following extensions are specified:

- The AssemblyConnectionDescription can be realized by a connector. Therefore, this class provides a direct association with ConnectorPackageDescription. The principle is similar to SubComponentInstantiationDescription that (by inheritance of ComponentUsageDescription) references ComponentPackageDescription itself referencing the connector definition (ComponentInterfaceDescription).

The association is 0..1. If the cardinality is 0 the connection is a basic CCM connection (facet → receptacle and events). If it is 1, the connection is implemented by a connector.

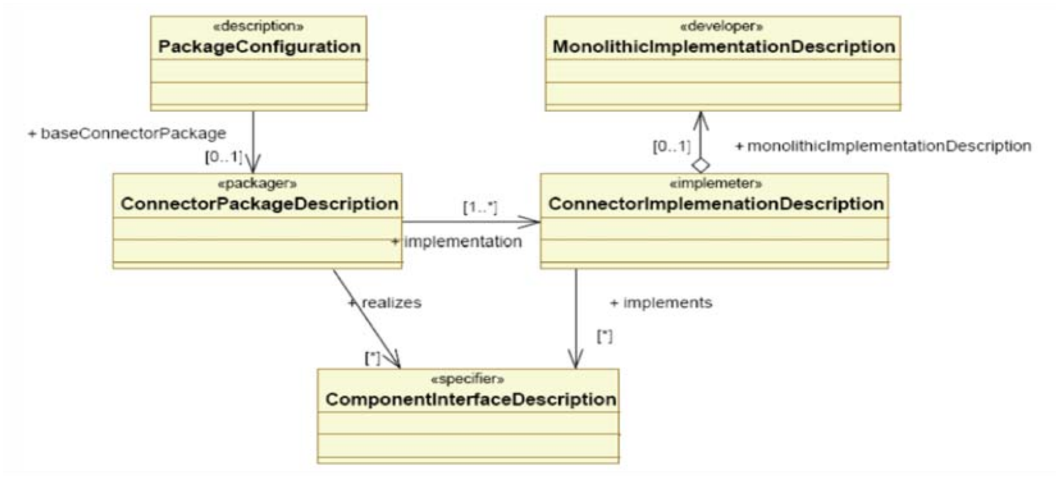

«description»
PackageConfiguration
«developer»
MonolithicImplementationDescription
+ baseConnectorPackage
[0..1]
[0..1]
+ monolithicImplementationDescription
«packager»
ConnectorPackageDescription
[1..*]
+ implementation
«implemeter»
ConnectorImplemenationDescription
+ realizes
+ implements
[*]
[*]
«specifier»
ComponentInterfaceDescription

Figure 7.10 - AssemblyConnectionDescription Extension

7.5.3 Execution Data Model

At the Execution Data Model level, the deployment plan is produced from the assembly description and corresponds to the assembly fully flattened. All executable artifacts are part of the deployment plan.

Each connector fragment is described as artifacts (ArtefactDeploymentDescription), implementations (MonolithicDeploymentDescription), instances (InstanceDeploymentDescriptions). By definition a connector implementation is the result of its fragment implementations. Each fragment can be deployed on a different target, that’s why at the execution model level fragments are manipulated, while at Component Data Model connectors are manipulated.

The transformation from assembly level (designed in a modeling tool) to the resulting deployment plan can be easily generated since all parameterized types are resolved when the assembly tool connects components with a connector. The resulting simple ports of components and connector fragments will be the endpoints to connect at deployment time.

This way of proceeding implies a very small impact on the existing deployment frameworks since they will deal with the same entities (artifacts, implementations, instances, and connections). Nevertheless few extensions are necessary to allow the instantiation of connector fragments and their configuration.

7.5.3.1 Compliance with Entry Points

This section refers to Section 10.6.1 of D&C [D&C] regarding the CCM entry points.

If the instance to be deployed is a connector, then the name of the execution parameter shall be “home factory.”

The parameter is of type String, and its name is the name of an entry point that has no parameters and that returns a pointer of type Connectors::HomeExecutorComponents:: HomeExecutorBase.

Thanks to this object, the deployment tool will call the create_component() operation on the KeylessCCMHome to instantiate a connector fragment.

7.5.4 Connector Configuration

The configuration of port type at assembly level produces the needed configuration values at deployment time.

All fragments of a given connector are in relation and have to be configured consistently. In some cases, this could require them to share configuration information that cannot be set statically. This dynamic initialization, if required, is connector implementation-specific and thus not specified. However it has to be completed by the end of the 'configuration_complete' phase of CCM deployment.

To configure the fragments the Components::HomeConfiguration IDL interface could be used. The method set_configuration_values is called in order to set the needed ConfigValues for the connector.

If two fragments need to exchange some configuration data (e.g., CORBA reference), the naming service could be used.

The configuration data are specified in the Component Deployment Plan file. Following is an example that shows how to configure fragments at deployment plan level.

```
<!-- ********************************************** -->
<!-- *************** INSTANCES ******************** -->
<!-- ********************************************** -->
<!-- Instance for fragment_instance_1 -->
<instance id="fragment_instance_1">
      <name>fragment_instance_1</name>
      <node>node1</node>
      <implementation ref="fragment_impl_1"/>
      <configProperty>
            <name>mcast_addr</name>
            <type>string</type>
            <value>224.1.1.1</value>
      </configProperty>
      <configProperty>
            <name>mcast_port</name>
            <type>unsigned short</type>
            <value>31337</value>
      </configProperty>
      <configProperty>
            <name>msg_size</name>
            <type>unsigned long</type>
            <value>50</value>
      </configProperty>
</instance>
```

7.5.5 CCM Meta-model Extension to support Generic Interactions

In this section, the basic concepts of the component model are summarized, based on the CCM meta-model [UML_CCM]. Central to it is the notion of Component definition (ComponentDef). It corresponds to the specification of a new component type, providing, using, and supporting possibly several interfaces, as well as consuming, emitting, or publishing event types. For configuration issues, attributes can be used as part of component definitions.

This part is based on the specification [UML_CCM] and extends it with new meta classes.

As an extension, the specification introduces the ExtendedPortDef as well as ExtendedPortType in the meta-model in order to allow definition of custom types of ports, the primary motivation being the reification at component level of interactions, which will be supported by the Connector concept.

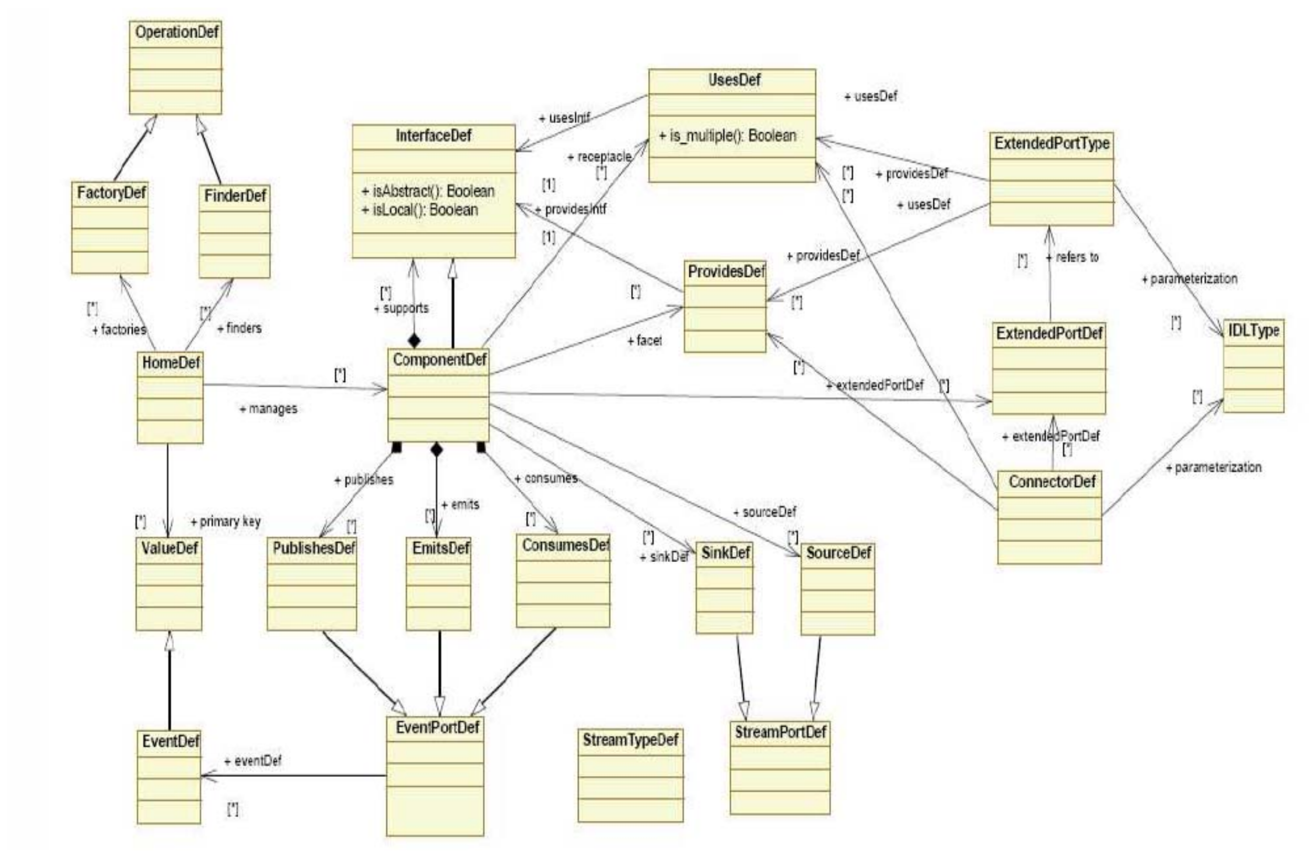


Figure 7.11 - Component Meta-Model - With Extended Ports and Connectors

ExtendedPortTypes are aggregation of zero or several provisions or needs of interfaces.

A matching relation for ExtendedPortType is defined as follows: two such types are compatible when they present one by one compatible UsesDef and ProvidesDef.

Finally, ConnectorDef is a new construct of the Component meta-model allowing modeling of connectors.

All those extensions are represented in Figure 7.11.

8 DDS-DCPS Application

This chapter instantiates the Generic Interaction Support described in the previous chapter, in order to define ports and connectors for DDS-DCPS. This chapter assumes an *a-priori* knowledge of DDS specification, at least of the DCPS part.

8.1 Introduction

8.1.1 Rationale for DDS Extended Ports and Connectors Definition

DDS is a very versatile middleware. It allows accommodating almost any conceivable flavor of data-centric publish/subscribe communication and therefore presents a very rich API and a very complete set of underlying behaviors and QoS policies. The counterpart of this richness is a certain complexity that may lead to errors or malfunctions due to mistaken uses.

Therefore, the purpose of "DDS for lightweight CCM" should be twofold:

1. Easing the deployment of applications made of components interacting through DDS by placing DDS configuration in the general component scheme (where configuration is carefully kept separated from the pure application code).
2. Providing to the components'author an easier access to DDS, by defining ready-to-use ports that would hide as much as possible DDS complexity.

However, ease of use should not come with too many restrictions that would compromise usefulness. In addition, as DDS is very versatile, defining a single couple of write and read ports that could accommodate simply all potential DDS usages seems unrealistic.

The process used to identify relevant DDS ports and connectors has been as follows:

- A large variety of DDS use patterns have been analyzed, then for each pattern, the roles1 have been identified and characterized in terms of:
  - associated DDS entities,
  - related QoS settings, and
  - programming contracts.
- All the identified programming contracts have been then analyzed and grouped to define DDS ports (each resulting programming contract corresponds to one DDS port).
- The most common DDS use patterns have been then identified as connectors, with their related DDS ports, their underlying DDS entities and associated QoS settings.

Even if these principles are general enough to be applicable to DCPS and DLRL uses of DDS, their actual realization results in extended ports and connectors that are specific to DCPS or DLRL.

8.1.2 From Connector-Oriented Modeling to Connectionless Deployment

It should be well understood that, even if at modeling levels DDS-enabled components are said ‘connected’ to a DDS-connector through their DDS-ports, that does not mean at all that they are physically connected (DDS is connectionless by nature). The following picture illustrates this change of paradigm from components connected to a DDS pattern at modeling time (in green) to components interacting via DDS through DDS ports to fulfill this DDS pattern at execution time (in yellow).

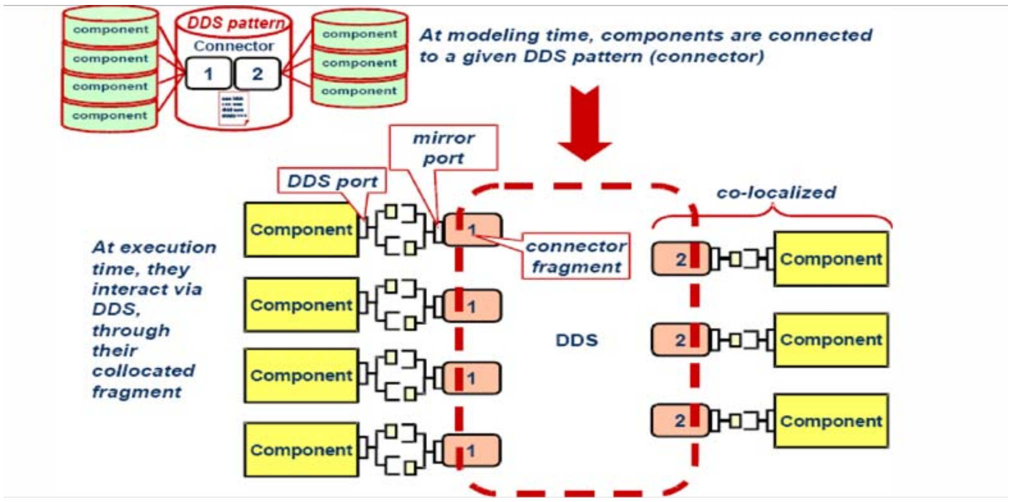


Figure 8.1 - From Modeling to Actual Deployment

8.2 DDS-DCPS Extended Ports

8.2.1 Design Rules

8.2.1.1 Parameterization

DDS-DCPS ports and connectors will be grouped in a module, itself parameterized by the data type and a sequence type of that data type.

- Grouping the definitions for port types and connectors in the same module allows that they share the same concrete interface when eventually instantiated.
- Passing that second parameter may seem redundant but it is the only way to allow sharing the sequence definition with the rest of the application.

To avoid useless duplications when instantiated, this template module will only contain the constructs that depend on the data-type. It will be included in a more general module that will also contain all the constructs that do not depend on the data-type.

Note: The following ports selected to be normative as fitting most DDS use patterns, are all parameterized by only one data type. However, as the Generic Interaction support allows defining new port types, nothing prevents users to define more specific ports that would be parameterized by several data types.

8.2.1.2 Basic Ports Definition

DDS-DCPS ports, as extended ports, will be made of several basic ports (uses and/or provides) with their defined interfaces.

The rationale to group operations as a single interface (thus one basic port), or on the contrary, to split them in different interfaces (thus several basic ports) is as follows:

- Different interaction directions (i.e., whether the component is a caller or a callee) result in different interfaces.
- Each interface is focused on a precise area of functionality (such as data access, status access...).

All those interfaces could be then considered as building blocks for DDS-DCPS extended ports.

8.2.1.3 Interface Design

For simplicity reasons, it has been chosen not only to keep the strictly needed operations, but also to simplify their parameters as much as possible, in particular:

- Information that comes with the read data samples have been simplified to what is most commonly used.
- Data access parameters, when they are likely to be shared by all the access of a given port (e.g., a query for read) are expressed by means of basic port interface attributes. Those attributes can be seen configurations for the ports.

Errors are reported by means of exceptions.

Sequences to be returned (of data and of accompanying information) are designed as ‘inout’ parameters, even if the actual information flow is only ‘out.’ This disposal allows for implementation of smarter memory management.

8.2.1.4 Simplicity versus Richness Trade-off

The goal of this specification is not to prevent the advanced user from making use of advanced DDS features, if needed. In return, complicating the mainstream port interfaces should be avoided. This is the reason why each DDS port contains an extra basic port to access directly to the more scoped underlying DDS entity (e.g., the DataWriter if it is a port for writing). If needed, all the involved DDS entities can be retrieved by this starting point.

Note: The DDS-DCPS ports are of large potential usage; however, as the Generic Interaction support allows defining new port types, nothing prevents users from defining their own DDS ports to fulfill more specific use patterns.

8.2.2 Normative DDS-DCPS Ports

This section lists the normative DDS extended ports. It starts with the list of interfaces for basic ports and then assembles them to make the DDS ports. All those constructs are included in the Typed template sub-module of the CCM_DDS module, as follows:

```
module CCM_DDS {
    // Non-typed definitions
    ...
    module Typed <typename T, sequence<T> TSeq> {
        // Typed definitions
        …
        };
    };
```

The following sections list extracts from the template module CCM_DDS::Typed<typename T, sequence<T> Tseq>.

The whole consolidated IDL is listed in 54IDL3+ of DDS-DCPS Ports and Connectors.

This IDL file is named “ccm_dds.idl.”

8.2.2.1 DDS-DCPS Basic Port Interfaces

8.2.2.1.1 Data Access - Publishing Side

Two interfaces allow writing DDS data.

1. A Writer allows publication of data on a given topic without paying any attention to the instance lifecycle; therefore, it just allows writing values of the related data type.

2. An Updater allows publication of data on a given topic when you do care about instance lifecycle. It allows creating, updating, and deleting instances of the related data type. It can be configured to actually check the lifecycle globally or just locally.

The following IDL declarations of those related interfaces are followed by explanations when needed.

InstanceHandleManager

```
local interface InstanceHandleManager {
    DDS::InstanceHandle_t register_instance (in T datum)
        raises (InternalError);
    void unregister_instance (in T datum, in DDS::InstanceHandle_t instance_handle)
        raises (InternalError);
    };
```

This abstract interface gathers the two operations that allow manipulating DDS instance handles and will serve as a basis for the Writer or the Updater interfaces.

- register_instance asks DDS to register an instance, which results in allocating it a local instance handle. The targeted instance is indicated by the key value in the passed data (datum).

- unregister_instance asks DDS to unregister the instance, indicated by the passed instance_handle and the key values of the passed data (datum) and thus to release the instance handle.

Both operations are very similar to the DDS operations and are just passed to the DDS DataReader in support for the relater DDS port (see the DDS documentation for more details). Any DDS error will be reported through an InternalError exception.

Interface Writer

```
local interface Writer : InstanceHandleManager {
    void write_one (in T datum, in DDS::InstanceHandle_t instance_handle)
        raises (InternalError);
    void write_many (in TSeq data)
        raises (InternalError);
    attribute boolean is_coherent_write;// FALSE by default
    };
```

Behavior of a Writer is as follows:

- write_one allows publishing one instance value. The targeted instance is designated by the passed instance handle (instance_handle) if not DDS::HANDLE_NIL or else by the key values in the passed data (datum). If a valid handle is passed, it must be in accordance with the key values of the passed data; otherwise, an InternalError exception is raised with the returned DDS error code. More generally, any DDS error when publishing the data will be reported by an InternalError exception.
- write_many allows publishing a batch of instance values is a single operation. Resulting DDS orders are stopped at the first error (and the index of the erroneous instance value is reported in the raised InternalError exception).

If the attribute is_coherent_write is TRUE, the resulting successful write DDS orders are placed between a DDS begin_coherent_updates and an end_coherent_updates.

Interface Updater

```
local interface Updater : InstanceHandleManager {
    void create_one (in T datum)
        raises (AlreadyCreated,
            InternalError);
    void update_one (in T datum, in DDS::InstanceHandle_t instance_handle)
        raises (NonExistent,
            InternalError);
    void delete_one (in T datum, in DDS::InstanceHandle_t instance_handle)
        raises (NonExistent,
            InternalError);

    void create_many (in TSeq data)
        raises (AlreadyCreated,
            InternalError);
    void update_many (in TSeq data)
        raises (NonExistent,
            InternalError);
    void delete_many (in TSeq data)
        raises (NonExistent,
            InternalError);

    readonly attribute boolean is_global_scope;// FALSE by default
    attribute boolean is_coherent_write;// FALSE by default
    };
```

Behavior of an Updater is as follows:

- create_one (resp. update_one, delete_one) allows creating (resp. updating, deleting) one instance. For create_one this instance is designated by the key value in datum. For the two others, it is designated by the passed instance handle (instance_handle) if not DDS::HANDLE_NIL or else by the key value in the passed instance data (datum). If a valid handle is passed, it must be in accordance with the key value of the passed instance data; otherwise, an InternalError exception is raised with the returned DDS error code. More generally, any DDS error when publishing the data will be reported by an InternalError exception.
- create_many (resp. update_many, delete_many) allows creating (resp. updating, deleting) several instances in a single call. Resulting DDS orders are stopped at the first error (and the index of the erroneous instance value is reported in the raised InternalError exception).

If the attribute is_coherent_write is TRUE, the resulting successful write or dispose DDS orders are placed between a DDS begin_coherent_updates and an end_coherent_updates.

- create_one and create_many operations check that the targeted instances are not existing prior to the call. This check is performed locally to the component if the attribute is_global_scope is FALSE or globally to the data space if is_global_scope is TRUE. In any case, this check is performed before any attempt ordering DDS to write and is applied to all the submitted instances. All the erroneous instances are reported in the AlreadyCreated exception (by means of their index in the submitted sequence).
- update_one and update_many operations check that the targeted instances are existing prior to the call. This check is performed locally to the component if the attribute is_global_scope is FALSE, or globally to the data space if is_global_scope is TRUE. In any case, this check is performed before any attempt ordering DDS to write and is applied to all the submitted instances. All the erroneous instances are reported in the NonExistent exception (by means of their index in the submitted sequence).
- delete_one and delete_many operations check that the targeted instances are existing prior to the call. This check is performed locally to the component if the attribute is_global_scope is FALSE, or globally to the data space if is_global_scope is TRUE. In any case, this check is performed before any attempt ordering DDS to dispose and is applied to all the submitted instances. All the erroneous instances are reported in the NonExistent exception (by means of their index in the submitted sequence).

Note: Global checks may require an attempt to get the instance under the scene and cannot be a full guarantee as a write or a dispose from another participant may always occur between the check and the actual write or dispose. Therefore this setting should be restricted to architectures where a single writer is involved.

Note: In case of a single operation (create_one, update_one, or delete_one) failing on the life cycle check, the sequence parameter of the exception (AlreadyExisting or NonExistent) will contain 0.

8.2.2.1.2 Data Access - Subscribing Side

Preamble: for all the following operations, read means implicitly “with no wait” and get means implicitly “with wait.”

Several interfaces allow retrieving data values from DDS data readers:

- A Reader allows reading one or several instance values on a given topic according to a given criterion, with no wait.

In addition, the following interfaces allow getting fresh values from a given topic:

- A Getter allows getting them in pull mode. It may block to get the proper information.
- A Listener allows getting them in push mode, regardless of the instance status.
- A StateListener allows getting them in push mode when the instance status is a concern: different operations will be triggered according to the instance status.

The following IDL declarations for those interfaces and related types, are followed by explanations when needed:

Related Types

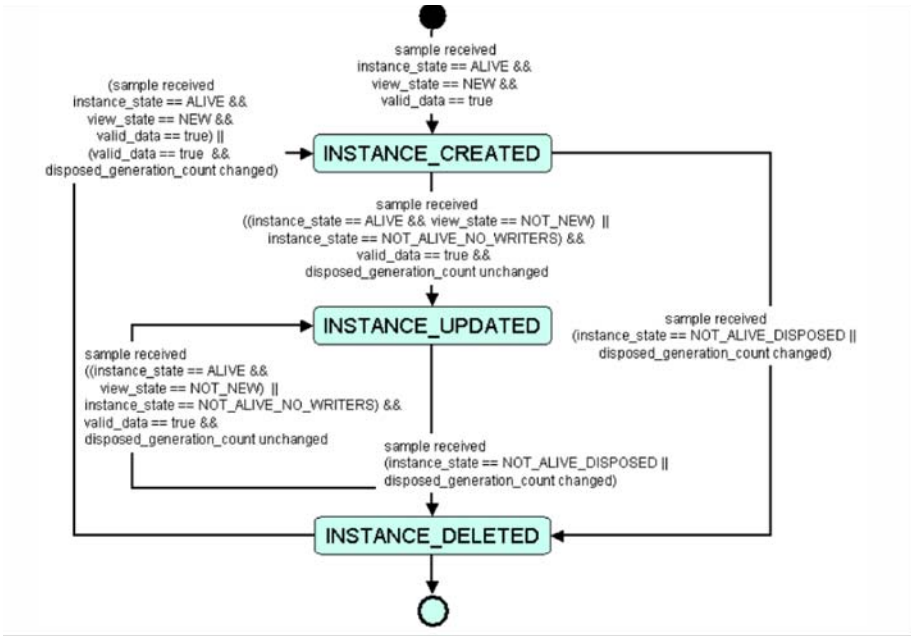
```
enum AccessStatus {
    FRESH_INFO,
    ALREADY_SEEN
    };

enum InstanceStatus {
    INSTANCE_CREATED,
    INSTANCE_FILTERED_IN,
    INSTANCE_UPDATED,
    INSTANCE_FILTERED_OUT,
    INSTANCE_DELETED
    };

struct ReadInfo {DDS::InstanceHandle_t      instance_handle;
    DDS::Time_t                             source_timestamp;
    AccessStatus                            access_status;
    InstanceStatus                          instance_status;
    };

typedef sequence<ReadInfo> ReadInfoSeq;
```

ReadInfo is the simplified version of DDS SampleInfo. Each read or gotten piece of data is accompanied with a ReadInfo that specifies:

- The DDS instance_handle
- The DDS source_timestamp
- Whether the value has already been seen or not by the component (access_status)
- The instance status (instance_status) at the time of the sample. This status can be:
    - INSTANCE_CREATED if this is the first time that the component sees that instance (the instance is then existing for the component).
    - INSTANCE_FILTERED_IN if an existing instance reenters the filter after having been filtered out.
    - INSTANCE_ UPDATED if an existing instance is modified and stays within the filter.
    - INSTANCE_FILTERED_OUT if an existing instance just stopped passing the filter.
    - INSTANCE_DELETED if the instance just stopped existing.

The instance_status is therefore a combination of several fields in the original DDS SampleInfo. Unfortunately, in the current DDS, the fact that a data is filtered out is not reported. However as this is likely to change soon, the two statuses INSTANCE_FILTERED_IN and INSTANCE_FILTERED_OUT have been added for provision. As long as this feature is not available in DDS, a compliant implementation of this specification is not required to deliver those two statuses.

The following figure shows how the three other values can be computed based on DDS returned information.


sample received
instance_state == ALIVE &&
view_state == NEW &&
valid_data == true
(sample received
instance_state == ALIVE &&
view_state == NEW &&
valid_data == true) ||
(valid_data == true &&
disposed_generation_count changed)
INSTANCE_CREATED
sample received
((instance_state == ALIVE && view_state == NOT_NEW) ||
instance_state == NOT_ALIVE_NO_WRITERS) &&
valid_data == true &&
disposed_generation_count unchanged
INSTANCE_UPDATED
sample received
(instance_state == NOT_ALIVE_DISPOSED ||
disposed_generation_count changed)
sample received
((instance_state == ALIVE &&
view_state == NOT_NEW) ||
instance_state == NOT_ALIVE_NO_WRITERS) &&
valid_data == true &&
disposed_generation_count unchanged
sample received
(instance_state == NOT_ALIVE_DISPOSED ||
disposed_generation_count changed)
INSTANCE_DELETED
Figure 8.2 - ReadInfo::instance_status State Chart

Note: Except if the **instance_status** is **INSTANCE_DELETED**, the associated data value is valid (other cases where **DDS::SampleInfo::valid_data** would be **FALSE** should be managed by the connector fragment and shouldn't be passed to the component).

Note: When several values are returned, they may be different samples of the same or of different instances. They will always be ordered by instances (i.e., all the samples of the first instance, followed by all the samples of the second one…).

```
struct QueryFilter {
    string              expression;
    DDS::StringSeq      parameters
    };
```

QueryFilter gathers in a single structure a query expression and its related parameters. The **QueryFilter** attribute placed on the **Reader** interface acts as a filter for all the read operations made through a port where such a **Reader** is attached. An empty string **expression** means no query.

This query expression and its related parameters are for DDS use and must comply with DDS rules (see the DDS specification for more details). Any attempt to set the attribute with values that are not accepted by DDS will result in an **InternalError** exception.

Interface Reader

```
 local interface Reader  {
void read_last (inout TSeq data, inout ReadInfoSeq infos)
      raises (InternalError);
void read_all (inout TSeq data, inout ReadInfoSeq infos)
      raises (InternalError);
void read_one_last (inout T datum, out ReadInfo info,
              in DDS::InstanceHandle_t instance_handle)
      raises (NonExistent,
              InternalError);
void read_one_all (in T datum, inout TSeq data, inout ReadInfoSeq infos,
              in DDS::InstanceHandle_t instance_handle)
      raises (NonExistent,
              InternalError);
attribute QueryFilter query
      setraises (InternalError);
};
```

Behavior of a **Reader** is as follows:

- Underlying DDS read operations will be performed with the following DDS access parameters:
    - **SampleStateMask**: **READ** or **NO_READ**
    - **ViewStateMask**: **NEW** or **NOT_NEW**
    - **InstanceStateMask**: **ALIVE**
    - Through the query as specified in the **query** ("" as **expression** means no query).
- **read_last** returns the last sample of all instances. In case of no data, the resulting data will be a void sequence. Any other DDS error when reading the data will be reported by an **InternalError** exception.
- **read_all** returns all samples of all instances. In case of no data, the resulting data will be a void sequence. Any other DDS error when reading the data will be reported by an **InternalError** exception.
- **read_one_last** returns the last sample of a given instance. The targeted instance is designated by the passed instance handle (**instance_handle**) if not **DDS::HANDLE_NIL** or else by the key value in the passed data (**datum**). If a valid handle is passed, it must be in accordance with the key value of the passed data; otherwise, an **InternalError** exception is raised with the returned DDS error code. More generally, any DDS error when reading the data will be reported by an **InternalError** exception.
    - In case the instance does not exist (no data are registered for that instance in DDS), the exception **NonExistent** is raised.
    - In case of a keyless topic, the last value in the topic will be returned as DDS considers all values in such a topic as samples of one unique instance.
- **read_one_all** returns all the samples of a given instance The targeted instance is designated by the passed instance handle (**instance_handle**) if not **DDS::HANDLE_NIL** or else by the key value in the passed data (**datum**). If a valid handle is passed, it must be in accordance with the key value of the passed data; otherwise, an **InternalError** exception is raised with the returned DDS error code. More generally, any DDS error when reading the data will be reported by an **InternalError** exception.

- In case the instance does not exist (no data are registered for that instance in DDS), the exception **NonExistent** is raised.
- In case of a keyless topic, all values will be returned as DDS considers all values in such a topic as samples of one unique instance.

Note: This interface is the basis for a passive data reader (i.e., a component that just looks at the data as they are). It is also very useful for the reactive data getters (i.e., components that need to react to new data, whether they choose to get them in pull mode or be notified in push mode) in their initialization phase. This is the reason why all the DDS ports on the subscribing side will embed a **Reader** basic port.

Interface Getter

```
local interface Getter {
 boolean get_one (out T datum, out ReadInfo info)
        raises (InternalError);
 boolean get_many (inout TSeq data, inout ReadInfoSeq infos)
        raises (InternalError);
 attribute DDS::Duration_t        time_out;
 attribute DataNumber_t           max_delivered_data;       // default 0 (no limit)
 };
```

Behavior of a **Getter** is as follows:

- **Get** operations are meant to provide information that has not been previously communicated to the participant. They may wait until fresh information is available and are performed with the following parameters:
    - **SampleStateMask**: **NO_READ**
    - **ViewStateMask**: **NEW** or **NOT_NEW**
    - **InstanceStateMask**: **ALIVE** or **NOT_ALIVE**
    - Through the query (if any) of the **Reader** associated to the port
    - Within the time limit specified in **time_out**.
- They all return a **boolean** as result indicating whether actual data are provided (**TRUE**) or if the time-out occurred (**FALSE**).
- **get_one** returns the next sample to be gotten.
- **get_many** returns all the available samples within the limits set by the attribute **max_delivered_data**. In case there are too many available samples, only the first **max_delivered_data** is returned, the others remaining available for a subsequent call. The default value for that attribute is **UNLIMITED** (0).

Interface Listener

```
local interface Listener {
   void on_one_data (in T datum, in ReadInfo info);
   void on_many_data (in TSeq data, in ReadInfoSeq infos);
   };
```

Behavior of a **Listener** is as follows:

- The semantics of **on_one_data** is similar to the one of **Getter::get_one**, except that it is in push mode instead of pull mode.
- The semantics of **on_many_data** is similar to the one of **Getter::get_many**, except that it is in push mode instead of pull mode.
- The operations are called according to the listener **mode** as set in the associated **DataListenerControl** (see 8.2.2.1.3, 'Data Listener Control'). The mode can be:
  - **NOT_ENABLED**: none of these operations are called.
  - **ONE_BY_ONE**: the data are delivered one sample at a time through the on_one_data_operation.
  - **MANY_BY_MANY**: the data are delivered, though the **on_many_data** operation, by groups of samples, according to the **max_delivered_data** limit set in the associated **DataListenerControl**.
- Query filter (if any) will be found in the associated **Reader**.

Interface StateListener

```
local interface StateListener {
void on_creation (in T datum, in ReadInfo info);
void on_one_update (in T datum, in ReadInfo info);
void on_many_updates (in TSeq data, in ReadInfoSeq infos);
void on_deletion (in T datum, in ReadInfo info);
};
```

Behavior of a **StateListener** is as follows:

- No operation is called if the **mode** of the associated **StateListenerControl** is **NOT_ENABLED**.
- **on_creation** is triggered if the instance is considered as new in the component scope. Note that in case there is a filter in the **Reader** associated to the port and the attribute **is_filter_interpreted** of the listener control is **TRUE**, this gathers also the case when the instance is filtered in.
- **on_deletion** is triggered if the instance is no more existing. Note that in case there is a filter in the **Reader** associated to the port and the attribute **is_filter_interpreted** of the listener control is **TRUE**, this gathers also the case when the instance is filtered out. The only fields valid in the provided **datum** parameter are the ones that make the key.
- **on_one_update** is triggered if neither **on_creation** nor **on_deletion** apply and the mode of the associated listener control is **ONE_BY_ONE.**
- **on_many_updates** is triggered if neither **on_creation** nor **on_deletion** apply and the mode of the associated listener control is **MANY_BY_MANY.** The number of returned samples is within the limits of the attribute **max_delivered_data** of the associated listener control.
- Query filter (if any) will be found in the associated **Reader**.

8.2.2.1.3 Data Listener Control

The following interface allows controlling the data listener attached to the port to which they are attached. There are two data listener controls:

1. **DataListenerControl**, which embed the basic controlling behavior for any kind of data listeners.
2. **StateListenerControl**, which is a specialization of the former, which add extra features for a **StateListener**.

Interface DataListenerControl

```
enum ListenerMode {
    NOT_ENABLED,
    ONE_BY_ONE,
    MANY_BY_MANY
    };

local interface DataListenerControl {
    attribute ListenerMode    mode;                  // default NOT_ENABLED
    attribute DataNumber_t    max_delivered_data;    // default 0 (no limit)
    };
```

The two attributes of a **DataListenerControl** allows controlling the associated data listener, as follows:

- If the **mode** is **NOT_ENABLED**, the associated listener's operations are not triggered. This is the default setting as it allows the component to perform its initialization phase (likely using the associated **Reader**) before receiving any data notifications.
- If the **mode** is **ONE_BY_ONE**, the unitary operations (i.e., **on_one_data** or **on_one_update**) of the associated listener are triggered.
- If the **mode** is **MANY_BY_MANY**, the grouped operations (i.e., **on_many_data** or **on_many_updates**) of the associated listener are triggered. These operations are called with as many relevant samples as available, possibly limited by the value of **max_delivered_data**. The default value for that attribute is **UNLIMITED** (0).

StateListenerControl

```
local interface StateListenerControl : DataListenerControl {
    attribute boolean      is_filter_interpreted;      // default FALSE
    };
```

This listener control, specific to control a **StateListener**, extends the former **DataListenerControl** with the attribute **is_filter_interpreted**.

- If **TRUE**, the associated listener should consider an instance entering in (resp. going out) the filter (if any) of the related **Reader**, as an instance creation (resp. deletion) and thus trigger the operation **on_creation** (resp. **on_deletion**).
- If **FALSE**, those events should be considered as normal instance updates and thus lead to triggering **on_one_update** or **on_many_updates**, depending on the **mode**.

Note: DDS is not currently reporting that an instance has been filtered out. This behavior has been thus added for provision. A compliant implementation of this specification is not required to support it as long as DDS does not report when instances are filtered out.

8.2.2.1.4 Content Filter Management

In addition to plain topics, DDS provides content-filtered topics for content-based subscriptions. Such a topic has to be created in relation with a classical one and given a filter expression. All data provided by this topic must pass the filter expression. Apart from that characteristic, content-filter topics and classical ones can be used the same way.

The following attribute allows declaring a filter to the port that will be used for DDS content-filtered subscriptions, in case it is given a value at configuration time.

Attribute Filter

```
attribute QueryFilter filter
    setraises (NonChangeable);
```

While the filter expression is immutable and can be thus considered as a structural configuration attribute of a given port, its parameters can be modified dynamically.

The following interface allows changing those parameters.

Interface ContentFilterSetting

```
local interface ContentFilterSetting {
    void set_filter_parameters (in DDS::StringSeq parameters)
        raises (InternalError);
    };
```

8.2.2.1.5 Status Access

DDS is communicating errors or warnings by means of statuses. Some of those statuses are relevant for the component author (e.g., sample lost), others are meaningful system wide (e.g., incompatible QoS) while others carry information that are needed for functioning (e.g., data on readers).

- The first are made available through a **PortStatusListener**; as those statuses may only concern a DDS data reader, a **PortStatusListener** is meaningful only on a DDS port related to subscribing.
- The second are made available through a **ConnectorStatusListener**
- The last are kept for internal implementation of connectors fragments and therefore not reported.

Interface portStatusListener

```
local interface PortStatusListener {// status that are relevant to the component
    void on_requested_deadline_missed(
        in DDS::DataReader the_reader,
        in DDS::RequestedDeadlineMissedStatus status);
    void on_sample_lost(
        in DDS::DataReader the_reader,
        in DDS::SampleLostStatus status);
    };
```

Interface ConnectorStatusListener

```
local interface ConnectorStatusListener {// status that are relevant system-wide
    void on_inconsistent_topic(
        in DDS::Topic the_topic,
        in DDS::InconsistentTopicStatus status);
    void on_requested_incompatible_qos(
        in DDS::DataReader the_reader,
        in DDS::RequestedIncompatibleQosStatus status);
    void on_sample_rejected(
        in DDS::DataReader the_reader,
        in DDS::SampleRejectedStatus status);
    void on_offered_deadline_missed(
        in DDS::DataWriter the_writer,
        in DDS::OfferedDeadlineMissedStatus status);
    void on_offered_incompatible_qos(
        in DDS::DataWriter the_writer,
        in DDS::OfferedIncompatibleQosStatus status);
    void on_unexpected_status (
        in DDS::Entity the_entity,
        in DDS::StatusKind status_kind);
    };
```

All the operations of those two listeners mimic exactly the related DDS operations, with exactly the same operation name and parameters.

In addition a last operation is added on **ConnectorStatusListener** to report unexpected statuses (**on_unexpected_status**). The two parameters are then the reporting DDS Entity and the DDS status kind.

8.2.2.2 DDS-DCPS Extended Ports

All the interfaces presented in the previous section, can be considered as building blocks to be assembled to form the extended ports. The following are defined:

```
porttype DDS_Write {
    uses Writer                     data;
    uses DDS::DataWriter            dds_entity;
    };

porttype DDS_Update {
    uses Updater                    data;
    uses DDS::DataWriter            dds_entity;
    };

porttype DDS_Read {
    uses Reader                     data;
    attribute QueryFilter           filter
        setraises(NonChangeable);
    uses ContentFilterSetting       filter_config;
    uses DDS::DataReader            dds_entity;
    provides PortStatusListener     status;
    };

porttype DDS_Get {
    uses Reader                     data;
    uses Getter                     fresh_data;
    attribute QueryFilter           filter
        setraises(NonChangeable);
    uses ContentFilterSetting       filter_config;
    uses DDS::DataReader            dds_entity;
    provides PortStatusListener     status;
    };
```

```
porttype DDS_Listen {
    uses Reader                     data;
    uses DataListenerControl        data_control;
    provides Listener               data_listener;
    attribute QueryFilter           filter
        setraises(NonChangeable);
    uses ContentFilterSetting       filter_config;
    uses DDS::DataReader            dds_entity;
    provides PortStatusListener     status;
    };

porttype DDS_StateListen {
    uses Reader                     data;
    uses StateListenerControl       data_control;
    provides StateListener          data_listener;
    attribute QueryFilter           filter
        setraises(NonChangeable);
    uses ContentFilterSetting       filter_config;
    uses DDS::DataReader            dds_entity;
    provides PortStatusListener     status;
    };
```

All DDS ports combine at least a basic port to access data with a basic port to access underlying DDS entity. **DDS_Get**, **DDS_Listen**, and **DDS_StateListen** split the data access functionality in two ports; the first (**Reader**) is there to set the read criterion and provide operations for the initialization phase, while the second (**Getter**, **Listener**, or **StateListener**) is rather intended to be used in the application processing loop. All the ports intended for the subscribing side comprise also a configuration attribute (**filter**) to set the content filter, a basic port to change the parameters of the filter expression (**filter_config**), and a port to be notified of the relevant statuses (**status**).

8.3 DDS-DCPS Connectors

DDS-DCPS connectors are intended to gather the connector fragments for all possible roles in a given DDS use pattern.

They come with several DDS-DCPS supported ports (which are expressed in the connector as mirror ports), each of them corresponding to a given role within this pattern as well as with related DDS entities and QoS setting.

As DDS-DCPS ports, DDS-DCPS connectors are parameterized by a data type. As they are very similar to components (from the D&C standpoint), they have configuration properties that allow specifying all the elements that are needed to properly instantiate them, namely:

- The name of the DDS Topic that is associated to the data type.
- The list of fields making up the key for that Topic.
- The DDS Domain Id.
- The QoS settings that are to be applied to the underlying DDS entities (how these settings are expressed is explained in Section 8.4, “Configuration and QoS Support,” on page 45).

Having all this information gathered at the connector-level (rather than split in each DDS participants) gives the ability to better master system consistency.

In addition, they provide a port to report configuration errors (e.g., to be used i.e., by a supervision service).

8.3.1 Base Connectors

DDS_Base connector uses a **ConnectorStatusListener** port for reporting configuration errors and contains attributes to store the Domain identifier and the QoS profile (see 8.4.2, 'DDS QoS Policies in XML' for more details on QoS profile). The QoS profile could be given either as a file URL or as the XML string itself.

Any attempt to change those attributes once the configuration is complete will raise a **NonChangeable** exception.

All DDS connectors should inherit from that base.

```
connector DDS_Base {
    uses ConnectorStatusListener     error_listener;
    attribute DDS:DomainId_t         domain_id
        setraises (NonChangeable);
    attribute string                 qos_profile          // File URL or XML string
        setraises (NonChangeable);
    };
```

DDS_TopicBase extends the **DDS_Base** with the name of one topic and its key description. **DDS_TopicBase** should be the base for all mono-topic connectors.

```
connector DDS_TopicBase : DDS_Base {
    attribute string                 topic_name
        setraises (NonChangeable);
     attribute DDS::StringSeq        key_fields
        setraises (NonChangeable);
    };
```

As the attributes of **DDS_Base**, the attributes of **DDS_TopicBase** are also non changeable once configured. Any attempt to change them once the configuration is complete will raise a **NonChangeable** exception.

8.3.2 Pattern State Transfer

This pattern corresponds to participants that publish the state of data they manage (role **observable**), associated with other participants that subscribe to get the information (role **observer**). All those roles relate to the connector's topic.

Observers can be of various kinds:

- **passive_observer** are just reading the state when they want.
- **pull_observer** are getting the state changes.
- **push_observer** are being notified with the state changes.
- **push_state_observer** are being notified with the state changes with different operations depending on the instance status.

The connector definition is as follows:

```
connector DDS_State : DDS_TopicBase {
    mirrorport DDS_Update           observable;
    mirrorport DDS_Read             passive_observer;
    mirrorport DDS_Get              pull_observer;
    mirrorport DDS_Listen           push_observer;
    mirrorport DDS_StateListen      push_state_observer;
    };
```

Typically, with this pattern, **HISTORY QoS** should be set to **KEEP_LAST**.

8.3.3 Pattern Event Transfer

This pattern corresponds to participants sending events over DDS (role **supplier**), while others consume them (role **consumer**). All those roles relate to the connector's topic.

Consumers can be of various kinds:

- **pull_consumer** are getting the events.
- **push_consumer** are being notified with the events.

The connector definition is as follows:

```
connector DDS_Event : DDS_TopicBase {
    mirrorport DDS_Write            supplier;
    mirrorport DDS_Get              pull_consumer;
    mirrorport DDS_Listen           push_consumer;
    };
```

Typically, with this pattern, **HISTORY QoS** should be set to **KEEP_ALL**.

8.4 Configuration and QoS Support

8.4.1 DCPS Entities

When the connector fragments are deployed, they must create under the scene the DDS entities that are needed to get the wanted interaction.

As they are defined, the DDS ports are related to one data type and should therefore be attached to one **DataReader** and/or **DataWriter**, which are entirely dedicated to their port.

The allocation rule for the **Subscriber**, **Publisher**, and **DomainParticipant** is less straightforward as they may be allocated to the port or to the component (meaning that they will be shared by the ports of that component) or to the container (meaning that they will be shared by the components running in that container). Consequently, even if the QoS requirements are expressed on a port basis, components and containers can be given DDS entities that can be used by the infrastructure for servicing embedded ports if they meet the port requirements.

8.4.2 DDS QoS Policies in XML

To ease the consistent management of DDS QoS settings, this specification defines *QoS profiles*. A QoS profile takes the form of an XML string and can gather *QoS*[1] for several DDS entities that form a whole.

The following sections explain how to build QoS Profiles in XML. The XML Schema as well as a QoS Profile with all default values QoS policies, as specified in [DDS], are in Annex C and Annex D respectively.

8.4.2.1 XML File Syntax

The XML configuration file must follow these syntax rules:

- The syntax is XML and the character encoding is UTF-8.
- Opening tags are enclosed in **<>**; closing tags are enclosed in **</>**.
- A value is a UTF-8 encoded string. Legal values are alphanumeric characters. All leading and trailing spaces are removed from the string before it is processed.
  For example, "**<tag>   value   </tag>**" is the same as "**<tag>value</tag>**."
- All values are case-sensitive unless otherwise stated.
- Comments are enclosed as follows: **<!-- comment -->**.
- The root tag of the configuration file must be **<dds>** and end with **</dds>**.

The primitive types for tag values are specified in the following table:

Table 8.4 QoS Profile: Supported Tag Values

Type	Format	Notes
Boolean	**yes**, **1**, **true** or **BOOLEAN_TRUE**:	Not case-sensitive
	no, **0**, **false** or **BOOLEAN_FALSE**:	
Enum	A string. Legal values are the ones defined for QoS Policies in the DCPS IDL of DDS specification [DDS]	Must be specified as a string. (Do not use numeric values.)
Long	**-2147483648** to **2147483647** or **0x80000000** to **0x7fffffff** or **LENGTH_UNLIMITED**	A 32-bit signed integer
UnsignedLong	**0** to **4294967296** or **0** to **0xffffffff**	A 32-bit unsigned integer

8.4.2.2 Entity QoS

To configure the QoS for a DDS Entity using XML, the following tags have to be used:

- <participant_qos>
- <publisher_qos>
- <subscriber_qos>
- <topic_qos>
- <datawriter_qos>
- <datareader_qos>

1. A QoS is the set of QoS policies for a given DDS entity (DataReader, DataWriter...)

Each QoS is identified by a name. The QoS can inherit its values from other QoSs described in the XML file. For example:

```
<datawriter_qos name="DerivedWriterQos" base_name="BaseWriterQos">
    <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
</datawriter_qos>
```

In the above example, the writer QoS named '**DerivedWriterQos**' inherits the values from the writer QoS **'BaseWriterQos.'** The **HistoryQosPolicy** kind is set to **KEEP_ALL_HISTORY_QOS**.

Each XML tag with an associated name can be uniquely identified by its fully qualified name in C++ style. The writer, reader, and topic QoSs can also contain an attribute called **topic_filter** that will be used to associate a set of topics to a specific QoS when that QoS is part of a DDS profile. See Section 8.4.2.3.2, "Topic Filters," on page 50.

8.4.2.2.1 QoS Policies

The fields in a **QosPolicy** are described in XML using a 1-to-1 mapping with the equivalent IDL representation in the DDS specification [DDS]. For example, the **Reliability QosPolicy** is represented with the following structures:

```
struct Duration_t {
    long sec;
    unsigned long nanosec;
    };

struct ReliabilityQosPolicy {
    ReliabilityQosPolicyKind kind;
    Duration_t max_blocking_time;
    };
```

The equivalent representation in XML is as follows:

```
<reliability>
    <kind></kind>
    <max_blocking_time>
        <sec></sec>
        <nanosec></nanosec>
    </max_blocking_time>
</reliability>
```

8.4.2.2.2 Sequences

In general, the sequences contained in the QoS policies are described with the following XML format:

```
<a_sequence_member_name>
    <element>...</element>
    <element>...</element>
    …
</a_sequence_member_name>
```

Each element of the sequence is enclosed in an **<element>** tag., as shown in the following example:

```
property>
    <value>
        <element>
            <name>my name</name>
            <value>my value</value>
        </element>
        <element>
            <name>my name2</name>
            <value>my value2</value>
        </element>
    </value>
</property>
```

A sequence without elements represents a sequence of length 0. For example:

```
<a_sequence_member_name/>
```

As a special case, sequences of octets are represented with a single XML tag enclosing a sequence of decimal / hexadecimal values between 0..255 separated with commas. For example:

```
<user_data>
    <value>100,200,0,0,0,223</value>
</user_data>
<topic_data>
    <value>0xff,0x00,0x8e,0xEE,0x78</value>
</topic_data>
```

8.4.2.2.3 Arrays

In general, the arrays contained in the QoS policies are described with the following XML format:

```
<an_array_member_name>
    <element>...</element>
    <element>...</element>
    ...
</an_array_member_name>
```

Each element of the array is enclosed in an **<element>** tag.

As a special case, arrays of octets are represented with a single XML tag enclosing an array of decimal/hexadecimal values between 0..255 separated with commas. For example:

```
<datareader_qos>
    ...
    <user_data>
        <value>100,200,0,0,0,223</value>
    </user_data>
</datareader_qos>
```

8.4.2.2.4 Enumeration Values

Enumeration values are represented using their IDL string representation. For example:

```
<history>
    <kind>KEEP_ALL_HISTORY_QOS</kind>
</history>
```

8.4.2.2.5 Time Values (Durations)

Following values can be used for fields that require seconds or nanoseconds:

- DURATION_INFINITE_SEC
- DURATION_ZERO_SEC
- DURATION_INFINITE_NSEC
- DURATION_ZERO_NSEC

The following example shows the use of time values:

```
<deadline>
    <period>
        <sec>DURATION_INFINITE_SEC</sec>
        <nanosec>DURATION_INFINITE_NSEC</nanosec>
    </period>
</deadline>
```

8.4.2.3 QoS Profiles

A QoS profile groups a set of related QoS, usually one per entity. For example:

```
<qos_profile name="StrictReliableCommunicationProfile">
    <datawriter_qos>
        <history>
            <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
    </datawriter_qos>
    <datareader_qos>
        <history>
            <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
    </datareader_qos>
</qos_profile>
```

8.4.2.3.1 QoS-Profile Inheritance

A QoS Profile can inherit its values from other QoS Profiles described in the XML file using the tag **base_name**. For example:

```
<qos_profile name="MyProfile" base_name="BaseProfile">
    ...
</qos_profile>
```

A QoS profile cannot inherit from other QoS profiles if the last one has not been parsed before.

8.4.2.3.2 Topic Filters

A QoS profile may contain several writer, reader, and topic QoSs that can be selected based on the evaluation of a filter expression on the topic name.

The filter expression is specified as an attribute in the XML QoS definition thanks to a **topic_filter** tag. For example:

```
<qos_profile name="StrictReliableCommunicationProfile">
    <datawriter_qos topic_filter="A*">
        <history>
            <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
    </datawriter_qos>
    <datawriter_qos topic_filter="B*">
        <history>
            <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
        <resource_limits>
            <max_samples>128</max_samples>
            <max_samples_per_instance>128</max_samples_per_instance>
            <initial_samples>128</initial_samples>
            <max_instances>1</max_instances>
            <initial_instances>1</initial_instances>
        </resource_limits>
    </datawriter_qos>
    …
</qos_profile>
```

If **topic_filter** is not specified, the filter '*' will be assumed. The QoSs with an explicit **topic_filter** attribute definition will be evaluated in order; they have precedence over a QoS without a **topic_filter** expression.

8.4.2.3.3 QoS Profiles with a Single QoS

The definition of an individual QoS is a shortcut for defining a QoS profile with a single QoS. For example:

```
<datawriter_qos name="KeepAllWriter">
    <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
</datawriter_qos>
```

is equivalent to the following:

```
<qos_profile name="KeepAllWriter">
    <writer_qos>
        <history>
            <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
    </writer_qos>
</qos_profile>
```

8.4.3 Use of QoS Profiles

A QoS Profile shall be attached as a configuration attribute to a DDS connector. This profile should contain all values for initializing DDS Entities that are required by the connector.

In case of the connector involves several topics (which is not the case with the normative DDS-DCPS extended ports and connectors), then the **topic_filter** feature of the QoS Profile may be used to properly allocate values to entities.

A QoS Profile could also be attached to a DDS-capable component (i.e., a component that has at least one DDS port) to define component's default **DomainParticipant**, **Subscriber**, and/or **Publisher**. These default entities should be used preferably if their setting is compatible with the QoS requested in the connector's profile. If they are not compatible, specific entities dedicated to the 'non-compatible' port will be created. In this component profile, any **topic_qos**, **datareader_qos**, or **datawriter_qos** is simply ignored.

In addition, a similar QoS Profile could be attached to a DDS-capable container (i.e., a container hosting DDS-capable components to define container's defaults that should be used in priority, if suitable).

8.4.4 Other Configuration – Threading Policy

As opposed to the DDS QoS policies that need to be managed system-wide, the threading policy is local to the component using a DDS port. The threading policy could be set at several levels:

- port (for all its facets)
- component (for all the facets of its ports)
- container (for all the facets of its components' ports)

When a facet is activated, the threadpool attached to the port:

- if there is no port's policy, the component's threadpool is used.
- if there is no component's one, the container's threadpool is used.
- if there is no container's policy, then the default is applied.

9 DDS-DLRL Application

This chapter instantiates the Generic Interaction Support described in Chapter 7, in order to define ports and connectors for DDS-DLRL. This chapter assumes an *a-priori* knowledge of DDS specification (particularly the DLRL part).

The rationale for providing support to DLRL flavor of CCM in CCM is very similar to the one that drives the DCPS support, namely simplify the use and enforce separation of concerns.

The DLRL principles have been to ease as much as possible the publication and reception of data by providing ability to define plain application objects whose some data members are mapped to DDS topics. Then plain object manipulation (creation, update, deletion) is automatically translated under the scene by the DLRL layer in DCPS publications, while similarly DCPS receptions are automatically turned in updating objects. This interface is very developer-friendly and can hardly be simplified.

In return, according to CCM principles, the setting of the DLRL infrastructure, namely the creation of the Cache and of the Object Homes, their registration as well as the adjustment if needed of the DCPS entities QoS (all this making up the DLRL configuration) can be put apart from the application code.

The design principles to identify DLRL ports and connectors is identical to DCPS application, in that:

- Ports will capture programming contracts for components.
- Connectors will be the support for system-wide configuration.

9.1 Design Principles

9.1.1 Scope of DLRL Extended Ports

In DLRL, the natural entry point to deal with objects of a given type is the related **ObjectHome** and all objects of a given **Cache** are very related and need to be managed consistently.

Consequently, a DLRL extended port should be created to give access to all objects of a given Cache. That extended port will contain one **receptacle** for each **ObjectHome** and another **receptacle** for the **Cache** functional operations (i.e., excluding all the operations that are related to configuration that will be for the only use of the **Connector** implementation).

9.1.2 Scope of DLRL Connectors

A connector is the natural support to gather all the DLRL extended ports that are related to the same set of topics in order to master their configuration system-wide.

As potentially a DLRL object model (consistent set of DLRL classes and their relations) is specific to one participant, it could be as many DLRL extended ports as participants sharing the same set of DCPS topics. However, nothing prevents deploying several components using the same DLRL object model (therefore using the same extended port definition).

9.2 DDS-DLRL Extended Ports

Due to its essential variable composition, it is not possible to define one normative DLRL extended port. In return, the definition of their basic ports as well as the extended port composition rule are normative.

9.2.1 DLRL Basic Ports

9.2.1.1 Cache Operation

This interface is intended to type the **receptacle** dedicated to using the **Cache** once initialized by the infrastructure. It therefore contains only the operative subset of the **DDS::Cache** functions and attributes.

All the retained functions mimic exactly the **DDS::Cache** ones, and therefore request the same parameters and return the same result. Similarly, all the retained attributes are identical to the **DDS::Cache** ones.

```
local interface CacheOperation {
    // Cache kind
    // ----------
    readonly attribute DDS::CacheUsage          cache_usage;

    // Other Cache attributes
    // ---------------------
    readonly attribute DDS::ObjectRootSeq       objects;
    readonly attribute boolean                  updates_enabled;
    readonly attribute DDS::ObjectHomeSeq       homes;
    readonly attribute DDS::CacheAccessSeq      sub_accesses;
    readonly attribute DDS::CacheListenerSeq    listeners;

    // Cache update
    // ------------
    void DDS::refresh( )
        raises (DDS::DCPSError);

    // Listener management
    // -------------------
    void attach_listener (in DDS::CacheListener listener);
    void detach_listener (in DDS::CacheListener listener);

    // Updates management
    // ------------------
    void enable_updates ();
    void disable_updates ();

    // CacheAccess Management
    // ---------------------
    DDS::CacheAccess create_access (in DDS::CacheUsage purpose)
        raises (DDS::PreconditionNotMet);
    void delete_access (in DDS::CacheAccess access)
        raises (DDS::PreconditionNotMet);
    };
```

9.2.1.2 DLRL Class (ObjectHome)

For each DLRL object type to be part of the application, the DLRL extended port should comprise a **receptacle** of type the related home inheriting from **DDS::ObjectHome**. That class should have been generated by the DDS-DLRL product tooling.

All accesses to the DLRL objects of this type will be manageable through this entry point.

9.2.2 DLRL Extended Ports Composition Rule

DLRL extended ports are as many as applications. A DLRL extended port should be made of:

- A **CacheOperation** receptacle
- As many **DDS:ObjectHome**-derived receptacles as DLRL object types that will be used by the component using that DLRL port (those types having been generated by the DDS-DLRL product tooling).

Following is an example of such a declaration:

```
porttype MyDlrlPort_1 {
    uses CCM_DDS::CacheOperationcache;
    uses FooHome        foo_home;           // entry point for Foo objects
    uses BarHome        bar_home            // entry point for Bar objects
    };
```

Based on this information, the related connector fragment will, under the scene:

- Create the cache according to the specified **CacheOperation::cache_usage**.
- Instantiate and register the specified **ObjectHome** (that will create the DCPS entities according to the DLRL → DCPS mapping).
- Apply the QoS profile to modify underlying DCPS entities (if specified in the connector).
- Enable the infrastructure so that DLRL objects can be created and used the DLRL way.

9.3 DDS-DLRL Connectors

As a DLRL connector aims at gathering as many mirror ports as there are different object models in the system sharing the related topics, its composition is essentially variable and application-dependent, and a unique standard DLRL connector cannot be defined. A DLRL connector should inherit from the connector **DDS_Base,** to be given a **ConnectorStatusListener** port, a domain id and a QoS profile attribute, and add as many mirror ports as there exist DLRL extended ports to share the related set of topics.

Following is an example of such a declaration:

```
connector MyDlrlConnector : CCM_DDS::DDS_Base {
    mirrorport MyDlrlPort_1 p1;
    mirrorport MyDlrlPort_2 p2;
    mirrorport MyDlrlPort_3 p3;
    };
```

9.4 Configuration and QoS Support

9.4.1 DDS Entities

As a DLRL port corresponds to one **Cache**, it must be given its own **Publisher** and/or **Subscriber** (depending on the cache usage). In addition, it will get as many **DataReaders** and/or **DataWriters** as there are topics used by the DLRL objects.

9.4.2 Use of QoS Profiles

Configuring DLRL ports can be achieved exactly with the same philosophy as for DCPS ports, with the same definition for a QoS Profile (see sections Section 8.4.2, “DDS QoS Policies in XML,” on page 45 and Section 8.4.3, “Use of QoS Profiles,” on page 51), except that, as the QoS Profile attached to the DLRL connector should contain values for all the topics involved, the **topic_filter** feature of the QoS Profile is to be used in case there is a need to specify different QoS values for different topics.

Annex A
IDL3+ of DDS-DCPS Ports and Connectors

(normative)

```
#include "dds_rtf2_dcps.idl"

module CCM_DDS {

// ==============================================================================
// Non-typed part
//      (here are placed all the constructs that are not dependent on the data type)
// ==============================================================================
// --------------------------
// Enums, structs and Typedefs
// --------------------------
typedef unsigned long               DataNumber_t;           // count or index of data
typedef sequence<DataNumber_t>      DataNumberSeq;

const DataNumber_t UNLIMITED = 0;

enum AccessStatus {
        FRESH_INFO,
        ALREADY_SEEN
        };

enum InstanceStatus {// at sample time, as perceived by the component
        INSTANCE_CREATED,
        INSTANCE_FILTERED_IN,
        INSTANCE_UPDATED,
        INSTANCE_FILTERED_OUT,
        INSTANCE_DELETED
        };

struct ReadInfo {
        DDS::InstanceHandle_t   instance_handle;
        DDS::Time_t             source_timestamp;
        AccessStatus            access_status;
        InstanceStatus          instance_status;
        };
typedef sequence<ReadInfo> ReadInfoSeq;

struct QueryFilter {
        string                  expression;
        DDS::StringSeq          parameters;
        };

// Data Listener control
// --------------------
enum ListenerMode {
        NOT_ENABLED,
        ONE_BY_ONE,
        MANY_BY_MANY
        };

// ----------
// Exceptions
// ----------
exception AlreadyCreated {
        DataNumberSeq indexes;             // of the erroneous
        };

exception NonExistent{
        DataNumberSeq indexes;             // of the erroneous
        };

exception InternalError{
        DDS::ReturnCode_t error_code;  // DDS codes that are relevant:
                                           // ERROR (1);
                                           // UNSUPPORTED (2);
                                           // BAD_PARAMETER (3)
                                           // PRECONDITION_NOT_MET (4)
                                           // OUT_OF_RESOURCE (5)
        DataNumber_t index;                // of the erroneous
        };

exception NonChangeable {};

// ----------
// Interfaces
// ----------

// Listener Control
// ----------------
local interface DataListenerControl {
        attribute ListenerMode             mode;                       // default NOT_ENABLED
        attribute DataNumber_t             max_delivered_data;         // default 0 (no limit)
        };

local interface StateListenerControl : DataListenerControl {
        attribute boolean                  is_filter_interpreted;      // default FALSE
        };

// Content Filter Parameters Setting
// ---------------------------------
local interface ContentFilterSetting {
        void set_filter_parameters (in DDS::StringSeq parameters)
                raises (InternalError);
        };

// Status Access
// -------------
local interface PortStatusListener {// status that are relevant to the component
        void on_requested_deadline_missed(
                in DDS::DataReader                                     the_reader,
                in DDS::RequestedDeadlineMissedStatus                  status);
        void on_sample_lost(
                in DDS::DataReader                          the_reader,
                in DDS::SampleLostStatus                    status);
        };

local interface ConnectorStatusListener { // status that are relevant system-wide
        void on_inconsistent_topic(
                in DDS::Topic                               the_topic,
                in DDS::InconsistentTopicStatus             status);
        void on_requested_incompatible_qos(
                in DDS::DataReader                          the_reader,
                in DDS::RequestedIncompatibleQosStatus      status);
        void on_sample_rejected(
                in DDS::DataReader                          the_reader,
                in DDS::SampleRejectedStatus                status);
        void on_offered_deadline_missed(
                in DDS::DataWriter                          The_writer,
                in DDS::OfferedDeadlineMissedStatus         status);
        void on_offered_incompatible_qos(
                in DDS::DataWriter                          the_writer,
                in DDS::OfferedIncompatibleQosStatus        status);
        void on_unexpected_status (
                in DDS::Entity                              the_entity,
                in DDS::StatusKind                          status_kind);
        };

// ---------------
// Connector bases
// ---------------
connector DDS_Base {
        uses ConnectorStatusListener                error_listener;
        attribute DDS::DomainId_t                   domain_id
                setraises (NonChangeable);
        attribute string                            qos_profile     // File URL or XML string
                setraises (NonChangeable);
        };

connector DDS_TopicBase : DDS_Base {
        attribute string                            topic_name
                setraises (NonChangeable);
        attribute DDS::StringSeq                    key_fields
                setraises (NonChangeable);
        };

// ================================================================================
// Typed sub-part
//      (here are placed all the construct that are depending on the data type
//       either directly or indirectly)
// ================================================================================

module Typed <typename T, sequence<T> TSeq> {
// Gathers all the constructs that are dependent on the data type (T),
// either directly -- interfaces making use of T or TSeq,
// or indirectly -- porttypes using or providing those intefaces.
// TSeq is passed as a second parameter to avoid creating a new sequence type.
        // ------------------------------------
        // Interfaces to be 'used' or 'provided'
        // ------------------------------------

        // Data access - publishing side
        // ----------------------------

        // -- InstanceHandle Manager
        local interface InstanceHandleManager {
                DDS::InstanceHandle_t register_instance (in T datum)
                            raises (InternalError);
                void unregister_instance (in T datum, in DDS::InstanceHandle_t instance_handle)
                            raises (InternalError);
                };

        // -- Writer: when the instance lifecycle is not a concern
        local interface Writer : InstanceHandleManager {
                void write_one (in T datum, in DDS::InstanceHandle_t instance_handle)
                            raises (InternalError);
                void write_many (in TSeq data)
                            raises (InternalError);
                attribute boolean is_coherent_write;// FALSE by default
                // behavior
                // ---------
                // - the handle is exactly managed as by DDS (cf. DDS spec for more details)
                // - attempt to write_many is stopped at the first error
                // - if is_coherent_write, DDS write orders issued by a write_many
                //          are placed between begin/end coherent updates (even if an error occurs)
                };

        // -- Updater: when the instance lifecycle is a concern
        local interface Updater : InstanceHandleManager {
                void create_one (in T datum, in DDS::InstanceHandle_t instance_handle)
                            raises (AlreadyCreated,
                                        InternalError);
                void update_one (in T datum, in DDS::InstanceHandle_t instance_handle)
                            raises (NonExistent,
                                        InternalError);
                void delete_one (in T datum,in DDS::InstanceHandle_t instance_handle)
                            raises (NonExistent,
                                        InternalError);

                void create_many (in TSeq data)
                            raises (AlreadyCreated,
                                        InternalError);
                void update_many (in TSeq data)
                            raises (NonExistent,
                                        InternalError);
                void delete_many (in TSeq data)
                            raises (NonExistent,
                                        InternalError);

                readonly attribute boolean is_global_scope;         // FALSE by default
                attribute boolean is_coherent_write;                // FALSE by default
```

```
        // behavior
        // --------
        // - the handle is exactly managed as by DDS (cf. DDS spec for more details)
        // - exceptions AlreadyCreated or NonExistent are raised at least if a local
        //          conflict exists; in addition if is_global_scope is true, the test on
        //          existence attempts to take into account the instances created outside
        //                  - note: this check requires to previously attempt to read (not free)
        //                  - note: this check is not 100% guaranteed as a creation or a deletion
        //                    may occur in the short time between the check and the DDS order
        // - For *-many operations:
        //                  - global check is performed before actual write or dispose
        //                    (in case of error, all the erroneous instances are reported
        //                    in the exception)
        //                  - attempt to DDS write or dispose is stopped at the first error
        //                  - if is_coherent_write, DDS orders resulting from a *_many operation
        //                    are placed between begin/end coherent updates (even if an error
        //                    occurs)
        };

// Data access - subscribing side
// ------------------------------

// -- Reader: to simply access to the available data (no wait)
local interface Reader {
        void read_last (inout TSeq data, inout ReadInfoSeq infos)
                raises (InternalError);
        void read_all (inout TSeq data, inout ReadInfoSeq infos)
                raises (InternalError);
        void read_one_last (inout T datum, out ReadInfo info,
                        in DDS::InstanceHandle_t instance_handle)
                raises (NonExistent,
                        InternalError);
        void read_one_all (in T datum, inout TSeq data, inout ReadInfoSeq infos,
                        in DDS::InstanceHandle_t instance_handle)
                raises (NonExistent,
                        InternalError);
        attribute QueryFilter query
                setraises (InternalError);
        // behavior
        // --------
        // - read operations are performed with the following parameters
        //                  - READ or NO_READ
        //                  - NEW or NOT_NEW
        //                  - ALIVE
        //                  - through the query as specified ("" expression means no query)
        // - data returned:
        //                  - read_last returns for each living instance, its last sample
        //                  - read_all returns all the samples of all instances
        //                    ordered by instance first and then by sample
        //                  - read_one_last returns the last sample of the given instance
        //                  - read_one_all returns all the samples for the given instance
        //          - read_one operations use the instance_handle the same way
        //            the Writer or Updater *_one operations do
        };
```

```
// -- Getter: to get new data (and wait for)
local interface Getter {
        boolean get_one (out T datum, out ReadInfo info)
                raises (InternalError);
        boolean get_many (inout TSeq data, inout ReadInfoSeq infos)
                raises (InternalError);
        attribute DDS::Duration_ttime_out;
        attribute DataNumber_t max_delivered_data;      // default 0 (no limit)
        // behavior
        // --------
        // - get operations are performed with the following parameters
        //                      - NO_READ
        //                      - NEW or NOT_NEW
        //                      - ALIVE or NOT_ALIVE
        //                      - through the query as specified in the associated Reader
        //                      - within the time limit specified in time_out
        // - all operations returns TRUE if data are provided
        //              or FALSE if time-out occurred
        // - data returned:
        //                      - get_one returns each read sample one by one
        //              - get_many returns all available samples within the
        //                       max_delivered_data limit
        };

// -- Listener: similar to a Getter but in push mode
local interface Listener {
        void on_one_data (in T datum, in ReadInfo info);
        void on_many_data (in TSeq data, in ReadInfoSeq infos);
        // behavior
        // --------
        // - on_one_data() trigered is the mode of the associated listener control
        //              is ONE_BY_ONE (then similar to a get_one(), except that in push mode
        //              instead of pull mode)
        // - on_many_data() triggered if the listener mode is MANY_BY_MANY (then
        //              similar to get_many() but in push mode)
        // - query filter (if any) in the associated Reader
        };

// -- StateListener: listener to be notified based on the instance lifecycle
local interface StateListener {
        void on_creation (in T datum, in ReadInfo info);
        void on_one_update (in T datum, in ReadInfo info);
        void on_many_updates (in TSeq data, in ReadInfoSeq infos);
        void on_deletion (in T datum, in ReadInfo info);
        // behavior
        // --------
        // - no operations are trigerred if the mode of the associated listener
        //              control is NOT_ENABLED
        // - on_creation() is triggered if the instance is considered as new in the
        //              component scope; note that in case there is a filter and the attribute
        //              is_filter_interpreted of the listener control is TRUE, this gathers also
        //              the case when the instance is filtered-in.
        // - on_deletion() is triggered if the instance is no more existing; note
        //              that in case there is a filter  and the attribute
```

```
	//		is_filter_interpreted of the listener control is TRUE, this gathers
	//		also the case when the instance is filtered-out
	// - on_one_update() is trigrered if neither on_creation() nor on_deletion()
	//		are triggered and the mode of the associated listener control is
	//		ONE_BY_ONE
	// - on_many_updates()is triggered if neither on_creation() nor on_deletion()
	//		are triggered and the mode of the associated listener control is
	//		MANY_BY_MANY; the number of returned samples is within the limits of
	//		max_delivered_data attribute of the associated listener control.
	// - query filter (if any) in the associated Reader
	};


// ---------
// DDS Ports
// ---------

porttype DDS_Write {
	uses Writer				data;
	uses DDS::DataWriter			dds_entity;
	};

porttype DDS_Update {
	uses Updater				data;
	uses DDS::DataWriter			dds_entity;
	};

porttype DDS_Read {
	uses Reader				data;
	attribute QueryFilter			filter
		setraises(NonChangeable);
	uses ContentFilterSetting		filter_config;
	uses DDS::DataReader			dds_entity;
	provides PortStatusListener		status;
	};

porttype DDS_Get {
	uses Reader				data;
	uses Getter				fresh_data;
	attribute QueryFilter			filter
		setraises(NonChangeable);
	uses ContentFilterSetting		filter_config;
	uses DDS::DataReader			dds_entity;
	provides PortStatusListener		status;
	};

porttype DDS_Listen {
	uses Reader				data;
	uses DataListenerControl		data_control;
	provides Listener			data_listener;
	attribute QueryFilter			filter
		setraises(NonChangeable);
	uses ContentFilterSetting		filter_config;
	uses DDS::DataReader			dds_entity;
	provides PortStatusListener		status;
```

```
        };

porttype DDS_StateListen {
        uses Reader                     data;
        uses StateListenerControl       data_control;
        provides StateListener          data_listener;
        attribute QueryFilter           filter
                setraises(NonChangeable);
        uses ContentFilterSetting       filter_config;
        uses DDS::DataReader            dds_entity;
        provides PortStatusListener     status;
        };

// ----------------------------
// Connectors
// (Correspond to DDS patterns)
// ----------------------------

connector DDS_State : DDS_TopicBase {
        mirrorport DDS_Update           observable;
        mirrorport DDS_Read             passive_observer;
        mirrorport DDS_Get              pull_observer;
        mirrorport DDS_Listen           push_observer;
        mirrorport DDS_StateListen      push_state_observer;
        };

connector DDS_Event : DDS_TopicBase {
        mirrorport DDS_Write            supplier;
        mirrorport DDS_Get              pull_consumer;
        mirrorport DDS_Listen           push_consumer;
        };
};

};
```

Annex B
IDL for DDS-DLRL Ports and Connectors

(normative)

```
    #include "dds_rtf2_dlrl.idl"

    module CCM_DDS {

    local interface CacheOperation {
    // Cache kind
    // ----------
    readonly attribute DDS::CacheUsage cache_usage;

    // Other Cache attributes
    // ---------------------
    readonly attribute DDS::ObjectRootSeq                objects;
      readonly attribute boolean   updates_enabled;
      readonly attribute DDS::ObjectHomeSeq              homes;
      readonly attribute DDS::CacheAccessSeq             sub_accesses;
      readonly attribute DDS::CacheListenerSeq           listeners;

    // Cache update
    // ------------
    void refresh( )
          raises (DDS::DCPSError);

    // Listener management
    // -------------------
    void attach_listener (in DDS::CacheListener listener);
    void detach_listener (in DDS::CacheListener listener);

    // Updates management
    // ------------------
    void enable_updates ();
    void disable_updates ();

    // CacheAccess Management
    // ----------------------
    DDS::CacheAccess create_access (in DDS::CacheUsage purpose)
          raises (DDS::PreconditionNotMet);
    void delete_access (in DDS::CacheAccess access)
          raises (DDS::PreconditionNotMet);
    };

};
```

Annex C
XML Schema for QoS Profiles

(normative)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://www.omg.org/dds/" xmlns:dds="http:/
/www.omg.org/dds/" targetNamespace="http://www.omg.org/dds/" elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <!-- definition of simple types -->
  <xs:simpleType name="elementName">
    <xs:restriction base="xs:string">
      <xs:pattern value="([a-zA-Z0-9 ])+"></xs:pattern>
      <!-- <xs:pattern value="^((::)?([a-zA-Z0-9])+(::([a-zA-Z0-9])+)*)$"/> -->
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="topicNameFilter">
    <xs:restriction base="xs:string">
      <xs:pattern value="([a-zA-Z0-9])+"></xs:pattern>
      <!-- <xs:pattern value="^((::)?([a-zA-Z0-9])+(::([a-zA-Z0-9])+)*)$"/> -->
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="destinationOrderKind">
    <xs:restriction base="xs:string">
      <xs:enumeration value="BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS"></xs:enumeration>
      <xs:enumeration value="BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS"></xs:enumeration>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="durabilityKind">
    <xs:restriction base="xs:string">
      <xs:enumeration value="VOLATILE_DURABILITY_QOS"></xs:enumeration>
      <xs:enumeration value="TRANSIENT_LOCAL_DURABILITY_QOS"></xs:enumeration>
      <xs:enumeration value="TRANSIENT_DURABILITY_QOS"></xs:enumeration>
      <xs:enumeration value="PERSISTENT_DURABILITY_QOS"></xs:enumeration>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="historyKind">
    <xs:restriction base="xs:string">
      <xs:enumeration value="KEEP_LAST_HISTORY_QOS"></xs:enumeration>
      <xs:enumeration value="KEEP_ALL_HISTORY_QOS"></xs:enumeration>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="livelinessKind">
    <xs:restriction base="xs:string">
      <xs:enumeration value="AUTOMATIC_LIVELINESS_QOS"></xs:enumeration>
      <xs:enumeration value="MANUAL_BY_PARTICIPANT_LIVELINESS_QOS"></xs:enumeration>
      <xs:enumeration value="MANUAL_BY_TOPIC_LIVELINESS_QOS"></xs:enumeration>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="presentationAccessScopeKind">
    <xs:restriction base="xs:string">
      <xs:enumeration value="INSTANCE_PRESENTATION_QOS"></xs:enumeration>
```

```
            <xs:enumeration value="TOPIC_PRESENTATION_QOS"></xs:enumeration>
            <xs:enumeration value="GROUP_PRESENTATION_QOS"></xs:enumeration>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="reliabilityKind">
        <xs:restriction base="xs:string">
            <xs:enumeration value="BEST_EFFORT_RELIABILITY_QOS"></xs:enumeration>
            <xs:enumeration value="RELIABLE_RELIABILITY_QOS"></xs:enumeration>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="ownershipKind">
        <xs:restriction base="xs:string">
            <xs:enumeration value="SHARED_OWNERSHIP_QOS"></xs:enumeration>
            <xs:enumeration value="EXCLUSIVE_OWNERSHIP_QOS"></xs:enumeration>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="nonNegativeInteger_UNLIMITED">
        <xs:restriction base="xs:string">
            <xs:pattern value="(LENGTH_UNLIMITED|([0-9])*)?"></xs:pattern>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="nonNegativeInteger_Duration_SEC">
        <xs:restriction base="xs:string">
            <xs:pattern value="(DURATION_INFINITY|DURATION_INFINITE_SEC|([0-9])*)?"></xs:pattern>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="nonNegativeInteger_Duration_NSEC">
        <xs:restriction base="xs:string">
            <xs:pattern value="(DURATION_INFINITY|DURATION_INFINITE_NSEC|([0-9])*)?"></xs:pattern>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="positiveInteger_UNLIMITED">
        <xs:restriction base="xs:string">
            <xs:pattern value="(LENGTH_UNLIMITED|[1-9]([0-9])*)?"></xs:pattern>
        </xs:restriction>
    </xs:simpleType>
    <!-- definition of named types -->
    <xs:complexType name="duration">
        <xs:all>
            <xs:element name="sec" type="dds:nonNegativeInteger_Duration_SEC" minOccurs="0"></xs:element>
            <xs:element name="nanosec" type="dds:nonNegativeInteger_Duration_NSEC" minOccurs="0"></
xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="stringSeq">
        <xs:sequence>
            <xs:element name="element" type="xs:string" minOccurs="0" maxOccurs="unbounded"></xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="deadlineQosPolicy">
        <xs:all>
            <xs:element name="period" type="dds:duration" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="destinationOrderQosPolicy">
```

```
        <xs:all>
            <xs:element name="kind" type="dds:destinationOrderKind" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="durabilityQosPolicy">
        <xs:all>
            <xs:element name="kind" type="dds:durabilityKind" default="VOLATILE_DURABILITY_QOS"
minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="durabilityServiceQosPolicy">
        <xs:all>
            <xs:element name="service_cleanup_delay" type="dds:duration" minOccurs="0"></xs:element>
            <xs:element name="history_kind" type="dds:historyKind" default="KEEP_LAST_HISTORY_QOS"
minOccurs="0"></xs:element>
            <xs:element name="history_depth" type="xs:positiveInteger" minOccurs="0"></xs:element>
            <xs:element name="max_samples" type="dds:positiveInteger_UNLIMITED" minOccurs="0"></xs:element>
            <xs:element name="max_instances" type="dds:positiveInteger_UNLIMITED" minOccurs="0"></xs:element>
            <xs:element name="max_samples_per_instance" type="dds:positiveInteger_UNLIMITED"
minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="entityFactoryQosPolicy">
        <xs:all>
            <xs:element name="autoenable_created_entities" type="xs:boolean" default="true" minOccurs="0"></
xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="groupDataQosPolicy">
        <xs:all>
            <xs:element name="value" type="xs:base64Binary" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="historyQosPolicy">
        <xs:all>
            <xs:element name="kind" type="dds:historyKind" default="KEEP_LAST_HISTORY_QOS" minOccurs="0"></
xs:element>
            <xs:element name="depth" type="xs:positiveInteger" default="1" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="latencyBudgetQosPolicy">
        <xs:all>
            <xs:element name="duration" type="dds:duration" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="lifespanQosPolicy">
        <xs:all>
            <xs:element name="duration" type="dds:duration" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="livelinessQosPolicy">
        <xs:all>
            <xs:element name="kind" type="dds:livelinessKind" default="AUTOMATIC_LIVELINESS_QOS"
minOccurs="0"></xs:element>
            <xs:element name="lease_duration" type="dds:duration" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="ownershipQosPolicy">
        <xs:all>
            <xs:element name="kind" type="dds:ownershipKind" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="ownershipStrengthQosPolicy">
        <xs:all>
            <xs:element name="value" type="xs:nonNegativeInteger" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="partitionQosPolicy">
        <xs:all>
            <xs:element name="name" type="dds:stringSeq" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="presentationQosPolicy">
        <xs:all>
            <xs:element name="access_scope" type="dds:presentationAccessScopeKind"
default="INSTANCE_PRESENTATION_QOS" minOccurs="0"></xs:element>
            <xs:element name="coherent_access" type="xs:boolean" default="false" minOccurs="0"></xs:element>
            <xs:element name="ordered_access" type="xs:boolean" default="false" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="readerDataLifecycleQosPolicy">
        <xs:all>
            <xs:element name="autopurge_nowriter_samples_delay" type="dds:duration" minOccurs="0"></
xs:element>
            <xs:element name="autopurge_disposed_samples_delay" type="dds:duration" minOccurs="0"></
xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="reliabilityQosPolicy">
        <xs:all>
            <xs:element name="kind" type="dds:reliabilityKind" minOccurs="0"></xs:element>
            <xs:element name="max_blocking_time" type="dds:duration" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="resourceLimitsQosPolicy">
        <xs:all>
            <xs:element name="max_samples" type="dds:positiveInteger_UNLIMITED" minOccurs="0"></xs:element>
            <xs:element name="max_instances" type="dds:positiveInteger_UNLIMITED" minOccurs="0"></xs:element>
            <xs:element name="max_samples_per_instance" type="dds:positiveInteger_UNLIMITED"
minOccurs="0"></xs:element>
            <xs:element name="initial_samples" type="xs:positiveInteger" minOccurs="0"></xs:element>
            <xs:element name="initial_instances" type="xs:positiveInteger" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="timeBasedFilterQosPolicy">
        <xs:all>
            <xs:element name="minimum_separation" type="dds:duration" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="topicDataQosPolicy">
```

```
        <xs:all>
            <xs:element name="value" type="xs:base64Binary" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="transportPriorityQosPolicy">
        <xs:all>
            <xs:element name="value" type="xs:nonNegativeInteger" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <!-- userDataQosPolicy uses base64Binary encoding:
        * Allowed characters are all letters: a-z, A-Z,  digits: 0-9, the characters: '+' '/' '=' and ' '
              +,/.=,the plus sign (+), the slash (/), the equals sign (=), and XML whitespace characters.
        * The number of nonwhitespace characters must be divisible by four.
        * Equals signs, which are used as padding, can only appear at the end of the value,
          and there can be zero, one, or two of them.
        * If there are two equals signs, they must be preceded by one of the following characters:
          A, Q, g, w.
       * If there is only one equals sign, it must be preceded by one of the following characters: A, E, I, M, Q, U, Y, c, g,
k, o, s, w, 0, 4, 8.
    -->
    <xs:complexType name="userDataQosPolicy">
        <xs:all>
            <xs:element name="value" type="xs:base64Binary" minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>
    <xs:complexType name="writerDataLifecycleQosPolicy">
        <xs:all>
            <xs:element name="autodispose_unregistered_instances" type="xs:boolean" default="true"
minOccurs="0"></xs:element>
        </xs:all>
    </xs:complexType>

    <xs:complexType name="domainparticipantQos">
        <xs:all>
            <xs:element name="user_data" type="dds:userDataQosPolicy" minOccurs="0"></xs:element>
            <xs:element name="entity_factory" type="dds:entityFactoryQosPolicy" minOccurs="0"></xs:element>
        </xs:all>
        <xs:attribute name="name" type="dds:elementName"></xs:attribute>
        <xs:attribute name="base_name" type="dds:elementName"></xs:attribute>
        <xs:attribute name="topic_filter" type="dds:topicNameFilter"></xs:attribute>
    </xs:complexType>
    <xs:complexType name="publisherQos">
        <xs:all>
            <xs:element name="presentation" type="dds:presentationQosPolicy" minOccurs="0"></xs:element>
            <xs:element name="partition" type="dds:partitionQosPolicy" minOccurs="0"></xs:element>
            <xs:element name="group_data" type="dds:groupDataQosPolicy" minOccurs="0"></xs:element>
            <xs:element name="entity_factory" type="dds:entityFactoryQosPolicy" minOccurs="0"></xs:element>
        </xs:all>
        <xs:attribute name="name" type="dds:elementName"></xs:attribute>
        <xs:attribute name="base_name" type="dds:elementName"></xs:attribute>
        <xs:attribute name="topic_filter" type="dds:topicNameFilter"></xs:attribute>
    </xs:complexType>
    <xs:complexType name="subscriberQos">
        <xs:all>
            <xs:element name="presentation" type="dds:presentationQosPolicy" minOccurs="0"></xs:element>
```

```
      <xs:element name="partition" type="dds:partitionQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="group_data" type="dds:groupDataQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="entity_factory" type="dds:entityFactoryQosPolicy" minOccurs="0"></xs:element>
    </xs:all>
    <xs:attribute name="name" type="dds:elementName"></xs:attribute>
    <xs:attribute name="base_name" type="dds:elementName"></xs:attribute>
    <xs:attribute name="topic_filter" type="dds:topicNameFilter"></xs:attribute>
  </xs:complexType>
  <xs:complexType name="topicQos">
    <xs:all>
      <xs:element name="topic_data" type="dds:topicDataQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="durability" type="dds:durabilityQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="durability_service" type="dds:durabilityServiceQosPolicy" minOccurs="0"></
xs:element>
      <xs:element name="deadline" type="dds:deadlineQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="latency_budget" type="dds:latencyBudgetQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="liveliness" type="dds:livelinessQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="reliability" type="dds:reliabilityQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="destination_order" type="dds:destinationOrderQosPolicy" minOccurs="0"></
xs:element>
      <xs:element name="history" type="dds:historyQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="resource_limits" type="dds:resourceLimitsQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="transport_priority" type="dds:transportPriorityQosPolicy" minOccurs="0"></
xs:element>
      <xs:element name="lifespan" type="dds:lifespanQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="ownership" type="dds:ownershipQosPolicy" minOccurs="0"></xs:element>
    </xs:all>
    <xs:attribute name="name" type="dds:elementName"></xs:attribute>
    <xs:attribute name="base_name" type="dds:elementName"></xs:attribute>
    <xs:attribute name="topic_filter" type="dds:topicNameFilter"></xs:attribute>
  </xs:complexType>
  <xs:complexType name="datareaderQos">
    <xs:all>
      <xs:element name="durability" type="dds:durabilityQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="deadline" type="dds:deadlineQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="latency_budget" type="dds:latencyBudgetQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="liveliness" type="dds:livelinessQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="reliability" type="dds:reliabilityQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="destination_order" type="dds:destinationOrderQosPolicy" minOccurs="0"></
xs:element>
      <xs:element name="history" type="dds:historyQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="resource_limits" type="dds:resourceLimitsQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="user_data" type="dds:userDataQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="ownership" type="dds:ownershipQosPolicy" minOccurs="0"></xs:element>
      <xs:element name="time_based_filter" type="dds:timeBasedFilterQosPolicy" minOccurs="0"></
xs:element>
      <xs:element name="reader_data_lifecycle" type="dds:readerDataLifecycleQosPolicy" minOccurs="0"></
xs:element>
    </xs:all>
    <xs:attribute name="name" type="dds:elementName"></xs:attribute>
    <xs:attribute name="base_name" type="dds:elementName"></xs:attribute>
    <xs:attribute name="topic_filter" type="dds:topicNameFilter"></xs:attribute>
  </xs:complexType>
  <xs:complexType name="datawriterQos">
    <xs:all>
```

```
         <xs:element name="durability" type="dds:durabilityQosPolicy" minOccurs="0"></xs:element>
         <xs:element name="durability_service" type="dds:durabilityServiceQosPolicy" minOccurs="0"></
xs:element>
         <xs:element name="deadline" type="dds:deadlineQosPolicy" minOccurs="0"></xs:element>
         <xs:element name="latency_budget" type="dds:latencyBudgetQosPolicy" minOccurs="0"></xs:element>
         <xs:element name="liveliness" type="dds:livelinessQosPolicy" minOccurs="0"></xs:element>
         <xs:element name="reliability" type="dds:reliabilityQosPolicy" minOccurs="0"></xs:element>
         <xs:element name="destination_order" type="dds:destinationOrderQosPolicy" minOccurs="0"></
xs:element>
         <xs:element name="history" type="dds:historyQosPolicy" minOccurs="0"></xs:element>
         <xs:element name="resource_limits" type="dds:resourceLimitsQosPolicy" minOccurs="0"></xs:element>
         <xs:element name="transport_priority" type="dds:transportPriorityQosPolicy" minOccurs="0"></
xs:element>
         <xs:element name="lifespan" type="dds:lifespanQosPolicy" minOccurs="0"></xs:element>
         <xs:element name="user_data" type="dds:userDataQosPolicy" minOccurs="0"></xs:element>
         <xs:element name="ownership" type="dds:ownershipQosPolicy" minOccurs="0"></xs:element>
         <xs:element name="ownership_strength" type="dds:ownershipStrengthQosPolicy" minOccurs="0"></
xs:element>
         <xs:element name="writer_data_lifecycle" type="dds:writerDataLifecycleQosPolicy" minOccurs="0"></
xs:element>
      </xs:all>
      <xs:attribute name="name" type="dds:elementName"></xs:attribute>
      <xs:attribute name="base_name" type="dds:elementName"></xs:attribute>
      <xs:attribute name="topic_filter" type="dds:topicNameFilter"></xs:attribute>
   </xs:complexType>

   <xs:complexType name="domainparticipantQosProfile">
      <xs:complexContent>
         <xs:restriction base="dds:domainparticipantQos">
            <xs:attribute name="name" type="dds:elementName" use="required"></xs:attribute>
         </xs:restriction>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="topicQosProfile">
      <xs:complexContent>
         <xs:restriction base="dds:topicQos">
            <xs:attribute name="name" type="dds:elementName" use="required"></xs:attribute>
         </xs:restriction>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="publisherQosProfile">
      <xs:complexContent>
         <xs:restriction base="dds:publisherQos">
            <xs:attribute name="name" type="dds:elementName" use="required"></xs:attribute>
         </xs:restriction>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="subscriberQosProfile">
      <xs:complexContent>
         <xs:restriction base="dds:subscriberQos">
            <xs:attribute name="name" type="dds:elementName" use="required"></xs:attribute>
         </xs:restriction>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="datawriterQosProfile">
```

```
            <xs:complexContent>
               <xs:restriction base="dds:datawriterQos">
                  <xs:attribute name="name" type="dds:elementName" use="required"></xs:attribute>
               </xs:restriction>
            </xs:complexContent>
         </xs:complexType>
         <xs:complexType name="datareaderQosProfile">
            <xs:complexContent>
               <xs:restriction base="dds:datareaderQos">
                  <xs:attribute name="name" type="dds:elementName" use="required"></xs:attribute>
               </xs:restriction>
            </xs:complexContent>
         </xs:complexType>

         <xs:complexType name="qosProfile">
            <xs:sequence>
               <xs:choice maxOccurs="unbounded">
                  <xs:element name="datareader_qos" type="dds:datareaderQos" minOccurs="0"
      maxOccurs="unbounded"></xs:element>
                  <xs:element name="datawriter_qos" type="dds:datawriterQos" minOccurs="0"
      maxOccurs="unbounded"></xs:element>
                  <xs:element name="topic_qos" type="dds:topicQos" minOccurs="0" maxOccurs="unbounded"></
      xs:element>
                  <xs:element name="domainparticipant_qos" type="dds:domainparticipantQos" minOccurs="0"
      maxOccurs="unbounded"></xs:element>
                  <xs:element name="publisher_qos" type="dds:publisherQos" minOccurs="0"
      maxOccurs="unbounded"></xs:element>
                  <xs:element name="subscriber_qos" type="dds:subscriberQos" minOccurs="0"
      maxOccurs="unbounded"></xs:element>
               </xs:choice>
            </xs:sequence>
            <xs:attribute name="name" type="dds:elementName" use="required"></xs:attribute>
            <xs:attribute name="base_name" type="dds:elementName"></xs:attribute>
         </xs:complexType>

   </xs:schema>
```

Annex D
Default QoS Profile

(non normative)

The following file content is a XML QoS Profile with all default values as specified in DDS.

```
<!--
Data Distribution Service QoS Profile – Default Values
-->
<dds xmlns="http://www.omg.org/dds/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="file://DDS_QoSProfile.xsd">
<qos_profile name=" DDS DefaultQosProfile">
    <datareader_qos>
        <durability>
              <kind>VOLATILE_DURABILITY_QOS</kind>
        </durability>
        <deadline>
            <period>
                <sec>DURATION_INFINITE_SEC</sec>
                <nanosec>DURATION_INFINITE_NSEC</nanosec>
            </period>
        </deadline>
        <latency_budget>
            <duration>
                <sec>0</sec>
                <nanosec>0</nanosec>
            </duration>
        </latency_budget>
        <liveliness>
            <kind>AUTOMATIC_LIVELINESS_QOS</kind>
            <lease_duration>
                <sec>DURATION_INFINITE_SEC</sec>
                <nanosec>DURATION_INFINITE_NSEC</nanosec>
            </lease_duration>
        </liveliness>
        <reliability>
            <kind>BEST_EFFORT_RELIABILITY_QOS</kind>
            <max_blocking_time>
                <sec>0</sec>
                <nanosec>100000000</nanosec>
            </max_blocking_time>
        </reliability>
        <destination_order>
             <kind>BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</kind>
        </destination_order>
        <history>
            <kind>KEEP_LAST_HISTORY_QOS</kind>
            <depth>1</depth>
        </history>
        <resource_limits>
```

```
            <max_samples>LENGTH_UNLIMITED</max_samples>
            <max_instances>LENGTH_UNLIMITED</max_instances>
            <max_samples_per_instance>LENGTH_UNLIMITED</max_samples_per_instance>
        </resource_limits>
        <user_data>
            <value></value>
        </user_data>
        <ownership>
            <kind>SHARED_OWNERSHIP_QOS</kind>
        </ownership>
        <time_based_filter>
            <minimum_separation>
                <sec>0</sec>
                <nanosec>0</nanosec>
            </minimum_separation>
        </time_based_filter>
        <reader_data_lifecycle>
            <autopurge_nowriter_samples_delay>
                <sec>DURATION_INFINITE_SEC</sec>
                <nanosec>DURATION_INFINITE_NSEC</nanosec>
            </autopurge_nowriter_samples_delay>
            <autopurge_disposed_samples_delay>
                <sec>DURATION_INFINITE_SEC</sec>
                <nanosec>DURATION_INFINITE_NSEC</nanosec>
            </autopurge_disposed_samples_delay>
        </reader_data_lifecycle>
    </datareader_qos>
    <datawriter_qos>
        <durability>
            <kind>VOLATILE_DURABILITY_QOS</kind>
        </durability>
        <durability_service>
            <service_cleanup_delay>
                <sec>0</sec>
                <nanosec>0</nanosec>
            </service_cleanup_delay>
            <history_kind>KEEP_LAST_HISTORY_QOS</history_kind>
            <history_depth>1</history_depth>
            <max_samples>LENGTH_UNLIMITED</max_samples>
            <max_instances>LENGTH_UNLIMITED</max_instances>
            <max_samples_per_instance>LENGTH_UNLIMITED</max_samples_per_instance>
        </durability_service>
        <deadline>
            <period>
                <sec>DURATION_INFINITE_SEC</sec>
                <nanosec>DURATION_INFINITE_NSEC</nanosec>
            </period>
        </deadline>
        <latency_budget>
            <duration>
                <sec>0</sec>
                <nanosec>0</nanosec>
            </duration>
        </latency_budget>
        <liveliness>
```

```
            <kind>AUTOMATIC_LIVELINESS_QOS</kind>
            <lease_duration>
                <sec>DURATION_INFINITE_SEC</sec>
                <nanosec>DURATION_INFINITE_NSEC</nanosec>
            </lease_duration>
        </liveliness>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
            <max_blocking_time>
                <sec>0</sec>
                <nanosec>100000000</nanosec>
            </max_blocking_time>
       </reliability>
       <destination_order>
           <kind>BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</kind>
        </destination_order>
        <history>
            <kind>KEEP_LAST_HISTORY_QOS</kind>
            <depth>1</depth>
        </history>
        <resource_limits>
            <max_samples>LENGTH_UNLIMITED</max_samples>
            <max_instances>LENGTH_UNLIMITED</max_instances>
            <max_samples_per_instance>LENGTH_UNLIMITED</max_samples_per_instance>
        </resource_limits>
        <transport_priority>
            <value>0</value>
        </transport_priority>
        <lifespan>
            <duration>
                <sec>DURATION_INFINITE_SEC</sec>
                <nanosec>DURATION_INFINITE_NSEC</nanosec>
            </duration>
        </lifespan>
        <user_data>
            <value></value>
       </user_data>
       <ownership>
           <kind>SHARED_OWNERSHIP_QOS</kind>
        </ownership>
        <ownership_strength>
            <value>0</value>
        </ownership_strength>
        <writer_data_lifecycle>
            <autodispose_unregistered_instances>true</autodispose_unregistered_instances>
        </writer_data_lifecycle>
    </datawriter_qos>
    <domainparticipant_qos>
        <user_data>
            <value></value>
        </user_data>
        <entity_factory>
            <autoenable_created_entities>true</autoenable_created_entities>
        </entity_factory>
    </domainparticipant_qos>
```

```
<subscriber_qos>
    <presentation>
        <access_scope>INSTANCE_PRESENTATION_QOS</access_scope>
        <coherent_access>false</coherent_access>
        <ordered_access>false</ordered_access>
    </presentation>
    <partition>
        <name></name>
    </partition>
    <group_data>
        <value></value>
    </group_data>
    <entity_factory>
        <autoenable_created_entities>true</autoenable_created_entities>
    </entity_factory>
</subscriber_qos>
<publisher_qos>
    <presentation>
        <access_scope>INSTANCE_PRESENTATION_QOS</access_scope>
        <coherent_access>false</coherent_access>
        <ordered_access>false</ordered_access>
    </presentation>
    <partition>
        <name></name>
    </partition>
    <group_data>
        <value></value>
    </group_data>
    <entity_factory>
        <autoenable_created_entities>true</autoenable_created_entities>
    </entity_factory>
</publisher_qos>
<topic_qos>
    <topic_data>
        <value></value>
    </topic_data>
    <durability>
        <kind>VOLATILE_DURABILITY_QOS</kind>
    </durability>
    <durability_service>
        <service_cleanup_delay>
            <sec>0</sec>
            <nanosec>0</nanosec>
        </service_cleanup_delay>
        <history_kind>KEEP_LAST_HISTORY_QOS</history_kind>
        <history_depth>1</history_depth>
        <max_samples>LENGTH_UNLIMITED</max_samples>
        <max_instances>LENGTH_UNLIMITED</max_instances>
        <max_samples_per_instance>LENGTH_UNLIMITED</max_samples_per_instance>
    </durability_service>
    <deadline>
        <period>
            <sec>DURATION_INFINITE_SEC</sec>
            <nanosec>DURATION_INFINITE_NSEC</nanosec>
        </period>
```

```
            </deadline>
            <latency_budget>
                <duration>
                    <sec>0</sec>
                    <nanosec>0</nanosec>
                </duration>
            </latency_budget>
            <liveliness>
                <kind>AUTOMATIC_LIVELINESS_QOS</kind>
                <lease_duration>
                    <sec>DURATION_INFINITE_SEC</sec>
                    <nanosec>DURATION_INFINITE_NSEC</nanosec>
                </lease_duration>
            </liveliness>
            <reliability>
                <kind>BEST_EFFORT_RELIABILITY_QOS</kind>
                <max_blocking_time>
                    <sec>0</sec>
                    <nanosec>100000000</nanosec>
                    </max_blocking_time>
            </reliability>
            <destination_order>
                <kind>BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</kind>
            </destination_order>
            <history>
                <kind>KEEP_LAST_HISTORY_QOS</kind>
                <depth>1</depth>
            </history>
            <resource_limits>
                <max_samples>LENGTH_UNLIMITED</max_samples>
                <max_instances>LENGTH_UNLIMITED</max_instances>
                <max_samples_per_instance>LENGTH_UNLIMITED</max_samples_per_instance>
            </resource_limits>
            <transport_priority>
                <value>0</value>
            </transport_priority>
            <lifespan>
                <duration>
                    <sec>DURATION_INFINITE_SEC</sec>
                    <nanosec>DURATION_INFINITE_NSEC</nanosec>
                </duration>
            </lifespan>
            <ownership>
                <kind>SHARED_OWNERSHIP_QOS</kind>
            </ownership>
        </topic_qos>
    </qos_profile>

</dds>
```

Annex E
QoS Policies for the DDS Patterns

(non normative)

The following tables summarizes the DDS QoS policies that are relevant for the two DDS patterns that have been selected (State Transfer Pattern as defined in 8.3.2, ’Pattern State Transfer’ and Event Transfer Pattern as defined in 8.3.3, ’Pattern Event Transfer ’).

In those tables the color code is as follows:

🟨	Qos is not defined for that DDS entity or entity is not relevant for that role
🟩	Default value changeable by the designer
🟩	Value changeable by the designer
🟧	Default value required by the pattern (invariant)
🟥	Value required by the pattern (invariant)

Pattern	State				
Role	Observer / State Pattern				
Entity	Topic	Data Reader	Data Writer	Subscriber	Publisher
QoS					
Deadline	infinite	infinite			
Destination order	BY_SOURCE_TIMESTAMP	BY_SOURCE_TIMESTAMP			
Durability	TRANSIENT_LOCAL TRANSIENT	TRANSIENT_LOCAL TRANSIENT			
Durability service					
Entity factory				autoenabled_created_entities=TRUE	
History	KEEP_LAST depth=1	KEEP_LAST depth=1			
Latency budget	0	0			
Lifespan	infinite				
Liveness	AUTOMATIC lease_duration=infinite	AUTOMATIC lease_duration=infinite			
Ownership	SHARED	SHARED			
Partition				""	
Presentation				INSTANCE coherent_access=FALSE ordered_access=TRUE	
Reader data lifecycle		autopurge_nowriter_samples_delay=infinite autopurge_disposed_samples_delay=infinite			
Reliability	RELIABLE	RELIABLE			
Resource limits	max_samples=LENGTH_UNLIMITED max_instances=LENGTH_UNLIMITED max_samples_per_instance=LENGTH_UNLIMITED	max_samples=LENGTH_UNLIMITED max_instances=LENGTH_UNLIMITED max_samples_per_instance=LENGTH_UNLIMITED			
Time based filter		minimum_separation=0			
Transport priority	0				

Pattern	State				
Role	Observable / State Pattern				
Entity	Topic	Data Reader	Data Writer	Subscriber	Publisher
QoS					
Deadline	infinite		infinite		
Destination order	BY_SOURCE_TIMESTAMP		BY_SOURCE_TIMESTAMP		
Durability	TRANSIENT_LOCAL TRANSIENT		TRANSIENT_LOCAL TRANSIENT		
Durability service	service_cleanup_delay=0 history_kind=KEEP_LAST history_depth=1 max_*=LENGTH_UNLIMTED		service_cleanup_delay=0 history_kind=KEEP_LAST history_depth=1 max_*=LENGTH_UNLIMTED		
Entity factory					autoenabled_created_entities=TRUE
History	KEEP_LAST depth=1		KEEP_LAST depth=1		
Latency budget	0		0		
Lifespan	infinite	infinite	infinite		
Liveness	AUTOMATIC lease_duration=infinite	AUTOMATIC lease_duration=infinite	AUTOMATIC lease_duration=infinite		
Ownership	SHARED		SHARED		
Partition					""
Presentation					INSTANCE coherent_access=FALSE ordered_access=TRUE
Reader data lifecycle					
Reliability	RELIABLE		RELIABLE		
Resource limits	max_samples=LENGTH_UNLIMTED max_instances=LENGTH_UNLIMTED max_samples_per_instance=LENGTH_UNLIMTED		max_samples=LENGTH_UNLIMTED max_instances=LENGTH_UNLIMTED max_samples_per_instance=LENGTH_UNLIMTED		
Time based filter					
Transport priority	0		0		

Role	Supplier / Event Pattern				
Entity	Topic	Data Reader	Data Writer	Subscriber	Publisher
Deadline	infinite		infinite		
Destination order	BY_SOURCE_TIMESTAMP		BY_SOURCE_TIMESTAMP		
Durability	VOLATILE		VOLATILE		
Durability service					
Entity factory					autoenabled_created_entities=TRUE
History	KEEP_ALL		KEEP_ALL		
Latency budget	0	0	0		
Lifespan	infinite	infinite	infinite		
Liveness	AUTOMATIC lease_duration=infinite		AUTOMATIC lease_duration=infinite		
Ownership	SHARED		SHARED		
Partition					""
Presentation					INSTANCE coherent_access=FALSE ordered_access=TRUE
Reader data lifecycle					
Reliability	BEST_EFFORT		BEST_EFFORT		
Resource limits	max_samples=LENGTH_UNLIMITED max_instances=LENGTH_UNLIMITED max_samples_per_instance=LENGTH_UNLIMITED	max_samples=LENGTH_UNLIMITED max_instances=LENGTH_UNLIMITED max_samples_per_instance=LENGTH_UNLIMITED	max_samples=LENGTH_UNLIMITED max_instances=LENGTH_UNLIMITED max_samples_per_instance=LENGTH_UNLIMITED		
Time based filter					
Transport priority	0		0		
Writer data lifecycle			autodispose unregistered_instance=FALSE		

Role	Consumer / Event Pattern				
Entity	Topic	Data Reader	Data Writer	Subscriber	Publisher
Deadline	infinite	infinite			
Destination order	BY_SOURCE_TIMESTAMP	BY_SOURCE_TIMESTAMP			
Durability	VOLATILE	VOLATILE			
Durability service					
Entity factory				autoenabled_created_entities=TRUE	
History	KEEP_ALL	KEEP_ALL			
Latency budget	0	0			
Lifespan	infinite				
Liveness	AUTOMATIC lease_duration=infinite	AUTOMATIC lease_duration=infinite			
Ownership	SHARED	SHARED			
Partition				""	
Presentation				INSTANCE coherent_access=FALSE ordered_access=TRUE	
Reader data lifecycle		autopurge_nowriter_samples_delay=infinite autopurge_disposed_samples_delay=infinite			
Reliability	BEST_EFFORT	BEST_EFFORT			
Resource limits	max_samples=LENGTH_UNLIMITED max_instances=LENGTH_UNLIMITED max_samples_per_instance=LENGTH_UNLIMITED	max_samples=LENGTH_UNLIMITED max_instances=LENGTH_UNLIMITED max_samples_per_instance=LENGTH_UNLIMITED			
Time based filter		minimum_separation=0			
Transport priority	0				
Writer data lifecycle					