Date: February 2009

DDS for Lightweight CCM

Version Beta 1

OMG Document Number: ptc/2009-02-02

Standard document URL: http://www.omg.org/spec/dds4ccm/1.0

Associated File(s)*:
http://www.omg.org/spec/dds4ccm/20081201
http: www.omg.org/spec/dds4ccm/20081202

Original file(s): DDS for Lightweight machine-readable files (mars/2008-12-11)

dds4ccm.dlrl.idl
dds4ccm.idl.3+
DDS_DefaultQos.xml
DDS_QosProfile.xsd
DDSforCCM-meta.uml

This OMG document replaces the submission document (mars/2008-12-10, Alpha 1). It is an OMG Adopted Beta Specification and is currently in the finalization phase.

Comments on the content of this document are welcome, and should be directed to issues@omg.org by August 31, 2009. You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*.

The FTF Recommendation and Report for this specification will be published on December 18, 2009. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

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.)

Table of Contents

1 Scope....1

2 Conformance....1

3 Normative References....2

4 Terms and Definitions....2

5 Symbols (and abbreviated terms)....2

6 Additional Information....3

- 6.1 Changes to Adopted OMG Specifications....3
  - 6.1.1 Extensions....3
  - 6.1.2 Changes....3
- 6.2 Acknowledgments....4

7 Generic Interaction Support....5

- 7.1 Extended Ports....5
  - 7.1.1 Extended Port Definition....5
    - 7.1.1.1 Fixed Extended Ports....5
    - 7.1.1.2 Parameterized Extended Ports....6
  - 7.1.2 Component Model Extension for Extended Ports....8
    - 7.1.2.1 Component Port Declaration....8
    - 7.1.2.2 Transformation from IDL3+ to Plain IDL....9
    - 7.1.2.3 Equivalent IDL (w.r.t Equivalent IDL section in CCM)....10
  - 7.1.3 IDL Grammar 11
    - 7.1.3.1 Template Interfaces....11
    - 7.1.3.2 Port and Port Types....12
- 7.2 Connector Extensions....13
  - 7.2.1 Connector Definition....13
    - 7.2.1.1 Fixed Connectors....14
    - 7.2.1.2 Parameterized Connectors....15
    - 7.2.1.3 Connector Attributes....15
    - 7.2.1.4 Connector Inheritance....16
    - 7.2.1.5 Composite Connectors....16
  - 7.2.2 IDL Grammar for Connectors....17
  - 7.2.3 Programming Model for Connectors....18
    - 7.2.3.1 Interface CCMObject....18
    - 7.2.3.2 configuration_complete....18
    - 7.2.3.3 Interface KeylessCCMHome....19
    - 7.2.3.4 Interface HomeConfiguration....19
    - 7.2.3.5 Equivalent IDL (w.r.t Equivalent IDL section in CCM)....19
    - 7.2.3.6 Connector Implementation Interfaces....19
  - 7.2.4 Connector Deployment and Configuration....20
    - 7.2.4.1 Integration of Connectors in D&C....20
    - 7.2.4.2 Component Data Model....20
    - 7.2.4.3 Execution Data Model....24
    - 7.2.4.4 Connector Configuration....25
    - 7.2.4.5 CCM Meta-model Extension to support Generic Interactions....25

8 DDS-DCPS Application....27

8.1 Introduction....27
8.1.1 Rationale for DDS Extended Ports and Connectors Definition....27
8.1.2 From Connector-Oriented Modeling to Connectionless Deployment....27
8.2 DDS-DCPS Extended Ports....28
8.2.1 Design Rules....28
8.2.1.1 Parameterization....28
8.2.1.2 Basic Ports Definition....28
8.2.1.3 Interface Design....28
8.2.1.4 Simplicity versus Richness Trade-off....29
8.2.2 Normative DDS-DCPS Ports....29
8.2.2.1 DDS-DCPS Basic Port Interfaces....29
8.2.2.2 DDS-DCPS Extended Ports....35
8.3 DDS-DCPS Connectors....36
8.3.1 Base Connectors....37
8.3.2 Pattern State Transfer....37
8.3.3 Pattern Event Transfer....38
8.4 Configuration and QoS Support....38
8.4.1 DCPS Entities....38
8.4.2 DDS QoS Policies in XML....38
8.4.2.1 XML File Syntax....39
8.4.2.2 Entity QoS....39
8.4.2.3 QoS Profiles....42
8.4.3 Use of QoS Profiles....43
8.4.4 Other Configuration – Threading Policy....44
9 DDS-DLRL Application....45
9.1 Design Principles....45
9.1.1 Scope of DLRL Extended Ports....45
9.1.2 Scope of DLRL Connectors....45
9.2 DDS-DLRL Extended Ports....45
9.2.1 DLRL Basic Ports....46
9.2.1.1 Cache Operation....46
9.2.1.2 DLRL Class (ObjectHome)....46
9.2.2 DLRL Extended Ports Composition Rule....46
9.3 DDS-DLRL Connectors....47
9.4 Configuration and QoS Support....47
9.4.1 DDS Entities....47
9.4.2 Use of QoS Profiles....47
Annex A: IDL3+ of DDS-DCPS Ports and Connectors....48
Annex B: IDL for DDS-DLRL Ports and Connectors....55
Annex C: XML Schema for QoS Profiles....56
Annex D: Default QoS Profile....63
Annex E: QoS Policies for the DDS Patterns....68

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 Specifications Catalog 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 Specifications
- Minimum CORBA
- CORBA Component Model (CCM) Specification
- Data Distribution Service (DDS) Specifications

Specialized CORBA Specifications

- Includes CORBA/e and Realtime and Embedded Systems

Language Mappings

- IDL / Language Mapping Specifications
- 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 Specifications
- CORBAservices Specifications
- CORBAfacilities Specifications
- OMG Embedded Intelligence Specifications
- OMG Security Specifications

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

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

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

Typographical Conventions

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

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

Arial - 9 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

Arial – 9 pt: Examples

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

1 Scope

CCM (including lightweight CCM[1]) offers as main features i) to make explicit connections between components and ii) 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 i) synchronous method invocation and ii) 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:

- ***Firstly, a Generic Interaction Support*** allowing to define new interactions in CCM. This support is made of two constructs: i) 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 ii) 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.
  Section 7 contains this part of the specification.

- ***Secondly, 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: i) extended ports and connectors for DDS/DCPS and ii) extended ports and connectors for DDS/DLRL.
  Sections 8 (for DCPS) and 9 (for DLRL) contain this part of the specification.

2 Conformance

The conformance criteria of an implementation w.r.t this specification is stated through the support for the following extensions:

1. ***A CCM framework claiming conformance with the "Generic Interaction Support***" part of this specification shall support extended ports and connectors:

   a) Extensions of IDL3 to support **porttype**, **mirrorport** and **port** declarations

   b) Extension of IDL3 to support parameterized interfaces (template)

   c) Extension of D&C PSM for CCM to describe extended ports

   d) Extension of IDL3 to support **connector** declaration

   e) Extension of D&C PSM for CCM to deploy and configure **connector** fragments

2. ***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.

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

[1] In the remaining document, CCM will implicitly refer also to lightweight CCM .

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 5formal/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 Modelling Language
• XML	eXtensible Mark-up Language

6 Additional Information

6.1 Changes to Adopted OMG Specifications

The proposed submission 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

6.1.1 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 **ExtendePortType** , **ExtendedPortDef** , **ConnectorDef**
- D&C PSM for CCM where 2 classes are added for the support of connectors: **ConnectorPackageDescription** and **ConnectorImplementationDescriptor**

6.1.2 Changes

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

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

6.2 Acknowledgments

The following companies submitted this specification:

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

The following company supported this specification:

- Commissariat à l'Energie Atomique (CEA)

7 Generic Interaction Support

The proposed Generic Interaction Support 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 defined 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 dependant 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 Extended Ports

7.1.1 Extended Port Definition

Extended ports allow grouping a set of single CCM ports (facet/**provides** and receptacle/**uses**), to define a particular semantic. The extended ports are declared in IDL3+ (extended IDL3 with new keywords). A new keyword is introduced to define extended ports (**porttype**) and to declare an extended port at component level (**port**).

Extended ports can be fixed or parameterized.

7.1.1.1 Fixed Extended Ports

Extended ports can be defined as a list of fixed provided and used IDL interfaces.

The following is an example of such a definition:

```
//--------------
// IDL3+
//--------------
// fixed interfaces
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;
        };
```

7.1.1.2 Parameterized Extended Ports

7.1.1.2.1 Definition

As extended ports are meant to capture interaction logics, their main benefit is obtained if they can be parameterized by types. In the above example, the **DataControlledConsumer** is only valid for consuming elements of specific type **Data.** It could be very useful to define a generic port type **ControlledConsumer** that would be usable for any type of sent data.

A parameterized definition to ports and interfaces (close to C++ templates, but simplified) is therefore added. This definition can easily be resolved at IDL compilation time.

The following is an example of such a definition::

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

// parameterized interface
interface Pusher <typename T>{
        void push(in T dat);
        };

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

// extended port definition
porttype ControlledConsumer <typename T> {
        provides Pusher <T> consumer;
        uses FlowControl control;
        };
```

The keyword **typename** allows to instantiate the template with any kind of IDL element. Instead of **typename**, the template can be forced to more specific IDL elements such as: **struct**, **eventtype**, **primitive**, **fixed**, **sequence**, **interface**, **valuetype**.

7.1.1.2.2 Transformation to IDL interfaces

When a port type is defined (whether it is fixed or parameterized), it can be declared as a component port or a connector port using new keywords **port** and **mirrorport** that will be specified later on. When a port type is based on a parameterized interface, this later needs to be instantiated to obtain the plain IDL interface.

Plain interface definitions will be derived from the parameterized ones, by:

- Applying a simple naming convention to define the interface name: the implied resulting interface will be named by concatenating the names of all the parameters, in their declarative order, separated with '_', followed by the name of the template interface, with '_' as separator;
- Replacing the **typename** identifiers by the actual type names in the type declarations;
- Replacing the **typename** identifiers placed in the operation names by the actual type names. In this operation the character **$** features a concatenation operator in the parameterized definition. This allows the definition of generic port types with operation names or interface names depending on parameters. (see example 4)

The following shows the result of such a transformation::

```
//--------------------------------
// declaration of an extended port
//--------------------------------
…
            port ControlledConsumer<Data> p;
…

//---------------
// Equivalent IDL
//--------------

// Implied interface definition
interface Data_Pusher {
            void push (in Data dat);
            };
```

The following is another example (with name construction):

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

// Parameterized interface
interface EventsPusher <typename T> {
            typedef sequence<T> T$Seq;                  // construction of a type name
            void push (in T$Seq events);
            void push_$T (in T event);             // construction of an operation name
            };

//--------------------------------
// declaration of the extended port
//--------------------------------
…
            port EventsPusher<Data> p;
…

//---------------
// Equivalent IDL
//---------------

// Implied interface declaration
interface Data_EventsPusher {
            typedef sequence<Data> DataSeq;
            void push (in DataSeq events);
            void push_Data (in Data event);
            };
```

The following table shows the way to define a port type:

Table 1: New IDL3 Keyword to Define a Port

porttype *<port_type>* ["<" **typename | struct | eventtype | primitive | fixed | sequence | interface | valuetype** template_id+">"] {…};	The port type `port_type` is an extended port and can be parameterized. **template_id** is replaced by the type used at instantiation.

The following table shows the way to define parameterized interfaces (template):

Table 2: Syntax to Define a Parameterized Interface

interface <interface_type< ["<" typename | struct | eventtype | primitive | fixed | sequence | interface | valuetype template_id+">"] {…};	The interface interface_type is a parameterized interface. template_id is replaced by the type used at instantiation

7.1.2 Component Model Extension for Extended Ports

This section deals with the way to define component types using port types

7.1.2.1 Component Port Declaration

The proposed submission introduces a new kind of port at component level, namely an extended port, by means of the new IDL3 keyword **port** as shown in the following table:

Table 3: New IDL3 Keyword to Declare an Extended Port

port *<porttype>* ["<"param_type+">"] *port_name*	The port `port_name` is an extended port of type `porttype` `[<param _type+>]`

The new **port** declaration can be parameterized with one or more type elements to instantiate a parameterized extended port.

In the original CCM, existing port kinds are seen as basic ports (provided/required interfaces, or events sinks/sources). An extended port can be subsumed in a group of provided/required interfaces, which can be used / provided.

To have a similar semantic as basic CCM ports (**uses** and **provides**) for extended ports, it is necessary to introduce a counter part to **port** keyword to differentiate if the component (or connector) uses or provides the extended port.

To avoid duplicated definitions, the keyword **mirrorport** has been introduced to define inverse of extended ports (as shown in the following table). A **mirrorport** results in exactly the same number of simple ports as the port of the same type, except that all the **uses** are turned into **provides** and vice-versa.

Table 4: New IDL3 Keyword to Declare an Inversed Extended Port

mirrorport *<porttype>* ["<"param_type+">"] *port_name*	The port `port_name` is an inverse port of type `porttype` `[<param _type+>]`

Even if the extended ports could be used also directly between components, they are likely to be of primary use for connectors as introduced in the next section. Connectors can also only use simple basic CCM ports if sufficient. If a component has to be connected to a connector, it has to define basic or extended ports that correspond to those of the connector.

In the following example, the component C1 makes use of the fixed extended port, while C2 makes use of the parameterized one.

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

component C1 {
        port Data_ControlledConsumer      p;          // use of a fixed extended port
        };

```

```
    component C2 {
            port ControlledConsumer<Data> p; // use of a parameterized extended port
            };
};
```

7.1.2.2 Transformation from IDL3+ to Plain IDL

7.1.2.2.1 Overview of the Transformation Process

As mentioned above, a set of new keywords has been specified to define extended ports and to declare them in a component or connector definition. The resulting declaration with new keywords is called **IDL3+** and has to be transformed in the equivalent IDL3 standard definition.

The following figure presents the steps of component definitions. Only the first step is new and is detailed in the following section:

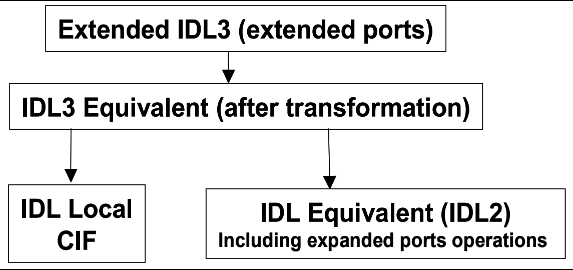
Extended IDL3 (extended ports)
IDL3 Equivalent (after transformation)
IDL Local CIF
IDL Equivalent (IDL2)
Including expanded ports operations

Figure 1: IDL3 Transformation

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

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

7.1.2.2.2 Translation from Extended Ports to Simple 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 transformations 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 an **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 '_'.
- If needed, the interface used in the basic port declaration is instantiated from its parameterized version as explained in section 7.1.1.2.

The translation for the previous example is as follows:

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

// Parameterized interface
interface Pusher <typename T>{
		void push(in T dat);
		};
interface FlowControl {
		void suspend();
		void resume();
		readonly attribute nb_waiting;
		};

// Extended port definition
porttype ControlledConsumer <typename T> {
		provides Pusher <T> consumer;
		uses FlowControl control;
		};

// Component definition
component C2 {
		port ControlledConsumer<Data> p; // use of a parameterized extended port
		};

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

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

// Component definition
component C2 {
		provides Data_Pusher	p_consumer;
		uses FlowControl p_control;
		};
```

As the resulting IDL3 is as of now, after this transformation, the CIF (Component Implementation Framework) remains unchanged, and nothing is new from the component developer's viewpoint.

At run time, each resulting **provides** will need to be connected to a similar **uses** and each resulting **uses** connected to a similar **provides**. Most of the times, even if it is not mandatory, those last **uses** and **provides** will themselves be grouped in an extended port, which will be exactly the inverse of the first one. This enforces the need to introduce the **mirrorport** keyword.

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

This specification does not impact the current CCM specification on the equivalent interfaces that the component developer can access. The transformation rules for components are the same. The equivalent IDL interfaces are fully deduced from the equivalent IDL3 after extended port transformation. At this stage, the component is defined with standard equivalent CCM IDL3 and can be mapped to equivalent IDL in a standard manner (cf. [CCM]).

7.1.3 IDL 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 5: 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

7.1.3.1 Template Interfaces

The definition of template interface is as follows:

Table 6: IDL Grammar Extensions for Template Interfaces

```
(1)  <type_classifier> ::= "typename"  | "struct" | "eventtype" | "primitive" | "fixed" | "sequence" |
   "interface" | "valuetype"

(2)  <template_interface> ::= "interface" <identifier> "<" {<type_classifier>} <identifier> {","
   {<type_classifier>} <identifier>}* ">" "{" <export>* "}" [“:” <interface_name> "<" <identifier> {","
   <identifier>}* ">"{ “,” <interface_name> "<" <identifier> {","  <identifier>}* ">"}* ]
```

7.1.3.1.1 Type Classifier Definition

The definition of type classifier used to define parameter types for template interfaces or port types is as follows:

```
(1)  <type_classifier> ::= "typename"  | "struct" | "eventtype" | "primitive" | "fixed" | "sequence" |
   "interface" | "valuetype"
```

7.1.3.1.2 Template Interface Definition

A template interface specification comprises:

- the **interface** keyword,
- an identifier for the interface type name,
- the specification of the template parameters,
- the interface definition,
- optionally, the inheritance from an other template interface.

(2) <template_interface> ::= "interface" <identifier> "<" {<type_classifier>} <identifier> {"," {<type_classifier>} <identifier>}* ">" "{" <export>* "}" [":" <interface_name> "<" <identifier> {"," <identifier>}* ">"{ "," <interface_name> "<" <identifier> {"," <identifier>}* ">"}*]

See section "Interface Declaration" of "OMG IDL Syntax and Semantics" [IDL] for the specification of <identifier> and <export>.

7.1.3.2 Port and Port Types

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.

The extensions to IDL3 grammar consist in the following productions:

Table 7: IDL Grammar Extensions for Ports and Port Types

(3) <porttype_dcl> ::= "porttype" <identifier> { "<" {<type_classifier>} <identifier> {"," {<type_classifier>} <identifier>} * ">" } "{" <port_export>+ "}"
(4) <port_export> ::= <extended_provides_dcl> ";" | <extended_uses_dcl> ";"
(5) <extended_provides_dcl> ::= <provides_dcl> | "provides" <generic_template_spec> <identifier>
(6) <extended_uses_dcl> ::= <uses_dcl> | "uses" {"multiple"} <generic_template_spec> <identifier>
(7) <template_type_spec> ::= <sequence_type> | <string_type> | <wide_string_type>| <fixed_point_type> | <generic_template_spec>
(8) <generic_template_spec> ::= <scoped_name> "<" <simple_type_spec> {"," <simple_type_spec>}* ">"
(9) <component_export> ::= <provides_dcl> ";" | <uses_dcl> ";" | <emits_dcl> ";" | <publishes_dcl> ";" | <consumes_dcl> ";" | <attr_dcl> ";" | <extended_port_dcl> ";"
(10) <extended_port_dcl> ::= "port" <generic_template_spec> <identifier> | "port" <scoped_name> <identifier> | "mirrorport" <generic_template_spec> <identifier> | "mirrorport" <scoped_name> <identifier>

7.1.3.2.1 Generic Template Specification

The definition of generic templates for interfaces satisfies the following syntax:

(7) <template_type_spec> ::= <sequence_type> | <string_type> | <wide_string_type>| <fixed_point_type> | <generic_template_spec>

(8) <generic_template_spec> ::= <scoped_name> "<" <simple_type_spec> {"," <simple_type_spec>}* ">"

See section "Type Declaration" of "OMG IDL Syntax and Semantics" [IDL] for the specification of <sequence_type>, <string_type>, <wide_string_type>, <fixed_point_type>, <simple_type_spec>.

The <scoped_name> in <generic_template_spec> must be previously defined and introduced either by:

- An interface declaration (<interface_dcl> – see Section "Interface Declaration" of "OMG IDL Syntax and Semantics" [IDL]),
- A value declaration (<value_dcl>, <value_box_dcl> or <abstract_value_dcl> – see Section "Value Declaration")
- A type declaration (<type_dcl> – see Section "Type Declaration").

Note that exceptions are not considered types in this context.

7.1.3.2.2 Port type definition

A port type definition comprises:

- the keyword **porttype**,
- an identifier for the port type name,
- the template specification if necessary,
- the declaration of basic or parameterized **provides** and **uses** ports

The syntax for port type definition is as follows:

(3) <porttype_dcl> ::= "porttype" <identifier> { "<" {<type_classifier>} <identifier> {"," {<type_classifier>} <identifier>} * ">" } "{" <port_export>+ "}"

(4) <port_export> ::= <extended_provides_dcl> ";" | <extended_uses_dcl> ";"

(5) <extended_provides_dcl> ::= <provides_dcl> | "provides" <generic_template_spec> <identifier>

(6) <extended_uses_dcl> ::= <uses_dcl> | "uses" {"multiple"} <generic_template_spec> <identifier>

See section "Type classifier Definition" of this document for the specification of <type_classifier definition>.

See section "Generic Template Specification" of this document for the specification of <generic_template_spec>

See section "Component Body" of "OMG IDL Syntax and Semantics" [IDL] for the definition of <provides_dcl>, <uses_dcl>

7.1.3.2.3 Extended port Declaration

An extended port declaration comprises:

- the **port** or **mirrorport** keyword,
- the name of a previously defined port type (simple or parameterized)
- the identifier for the name of the port provided by the component or the connector.

the syntax for extended port type declaration is as follows:

(10) <extended_port_dcl> ::= "port" <generic_template_spec> <identifier> | "port" <scoped_name> <identifier> | "mirrorport" <generic_template_spec> <identifier> | "mirrorport" <scoped_name> <identifier>

7.1.3.2.4 Component Declaration Extension

This syntax complements the defined syntax of section "Component Body" of "OMG IDL Syntax and Semantics" [IDL] for the definition of components, by adding the declaration of extended ports.

(9) <component_export> ::= <provides_dcl> ";" | <uses_dcl> ";" | <emits_dcl> ";" | <publishes_dcl> ";" | <consumes_dcl> ";" | <attr_dcl> ";" | <extended_port_dcl> ";"

7.2 Connector Extensions

This section presents the extensions to CCM component model required to support flexible interactions mechanisms through connectors.

7.2.1 Connector Definition

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.

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

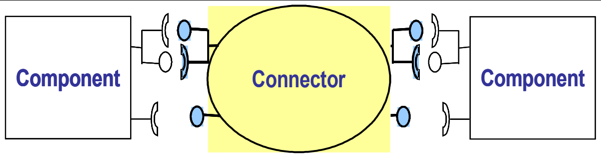

Figure 2: Logical View of a Connector

The connector is declared using a new IDL3 keyword: **connector**. This keyword is used as the **component** one to define the connector. As for **porttype**, connectors can be fixed or parameterized.

The connector definition cannot include the support keyword because a connector is just meant to gather ports (simple or extended)

The connector defined in IDL3 will concretely be composed of several parts (called *fragments*) that will consist of executors, each in charge of realizing a part of the interaction. 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, several ports behavior can be part of the same fragment. Each fragment will be co-localized to the component using them. This is an implementation choice for the connector developer.

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

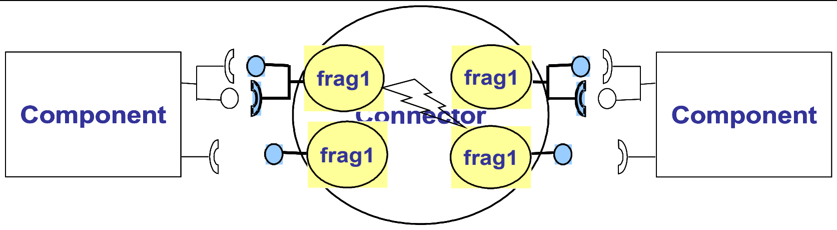

Figure 3: Connector Representation at Execution Time (Fragments)

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.

The mapping of connector definition 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 next section on connector programming).

7.2.1.1 Fixed Connectors

Like components, a connector can declare used and provided ports and interfaces.

The following is an example of connector definition:

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

7.2.1.2 Parameterized Connectors

7.2.1.2.1 Definition

The above example shows the declaration of a connector with fixed port and extended port; the port types can only be **Data_ControlledConsumer** and **Data_Pusher.** It could be useful to define a generic connector that could use parameterized extended port types.

One interest of connectors is to allow the definition of generic interaction modes following a particular semantic that could be adapted easily (by code generation) to concrete user defined interfaces. This is why the notion of template is also used for connectors.

The following is an example with parameterized ports::

```
// Connector definition
connector Cnx <typename T> {
        port ControlledConsumer<T> cc;
        mirrorport Pusher<T> p;
        };
```

The keyword **typename** allows to instantiate the template with any kind of IDL element. Instead of **typename**, the template can be forced to more specific IDL elements such as: **struct**, **eventtype**, **primitive**, **fixed**, **sequence**, **interface**, **valuetype**.

7.2.1.2.2 Transformation to Concrete Connector

Plain connector definition will be derived from the parameterized ones, by:

- Applying the transformation to the contained ports.
- Replacing the **typename** identifiers by the actual type names in the type declarations;

The following table shows the way to declare a connector:

Table 8: New IDL3 Keyword to Define a Connector

connector <*connector_type*> ["<" **typename | struct | eventtype | primitive | fixed | sequence | interface | valuetype** template_id+">"] {…};	The connector `connector_type` is a connector and can be parameterized. **template_id** is replaced by the type used at instantiation.

The following example shows how a connector can be instantiated:

```
// Connector instantiation

typedef Cnx<Data> MyCnx;
```

This example when evaluated, will instantiate a connector Cnx with the interfaces Data. It means that this connector declaration, after transformation will be similar to the one declared in the fixed connector example.

7.2.1.3 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.

Table 9: Declaration of Attributes for Connectors

connector *<connector_type>* {
[readonly]attribute <attr_type> <attr_name>;
};
	The connector `connector_type` exposes attributes named `attr_name`, of type `attr_type` and that can be *`readonly`*.

7.2.1.4 Connector Inheritance

Conceptually, a connector can inherit from an other connector. It means that the new connector is composed of all the ports and attributes of the inherited connector. The syntax to inherit from a connector is similar to the component inheritance:

Table 10: Declaration of Inheritance for Connectors

connector *<connector_type>* : <base_name> {…};	The connector `connector_type` inherits from the connector of type `base_name`

At realization level, the connector to build corresponds to a connector exposing all the inherited ports and attributes, plus its own ports and attributes. All the ports have to be implemented in the fragments composing the connector.

7.2.1.5 Composite Connectors

A connector (type) can have multiple implementations. As it 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: first, they enable to express the fragmented implementation of connectors by concepts already existing in CCM. Second, assembly implementations enable the composition of connectors, which facilitates the development of new connectors.

Consider the example of remote a 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 4. It is composed of two fragments called respectively SocketClient and FIFO_Socket_f_pull.

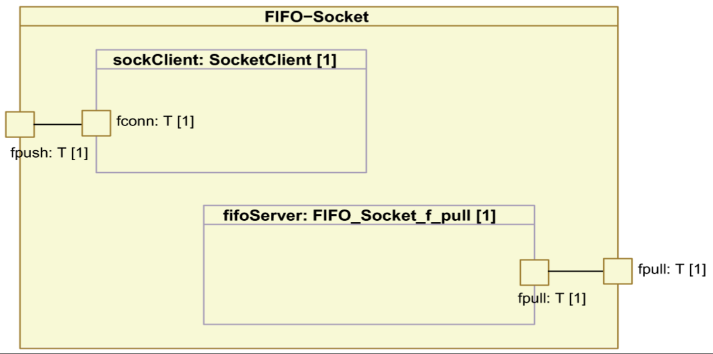
FIFO–Socket
sockClient: SocketClient [1]
fconn: T [1]
fpush: T [1]
fifoServer: FIFO_Socket_f_pull [1]
fpull: T [1]
fpull: T [1]

Figure 4: Example of a Distributed FIFO Implementation

Figure 5 shows the detailed implementation of the second fragment (FIFO_Socket_f_pull) which is itself an assembly of 2 fragments: SocketServer and ConnFIFO.

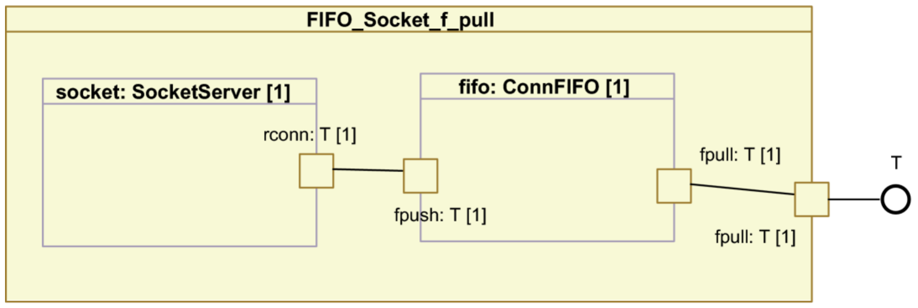

Figure 5: Assembly Implementation of a Connector Fragment

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.

7.2.2 IDL Grammar for Connectors

A connector is defined by its header and its body.

A connector header comprises:

- the keyword **connector**,
- an identifier for the name of the connector type,
- an optional classifier with its identifier if the connector is a parameterized one (generic connectors)
- an optional <connector_inheritance_spec>, consisting of a colon and a single <scoped_name> that must denote a previously-defined connector type.

A connector body comprises:

- facet declarations,
- receptacle declarations
- extended port declarations.

The following description of IDL grammar extensions uses the same syntax notation as 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 11: IDL Grammar Extensions for Connectors

(11) <connector_inheritance_spec> ::= ":" <scoped_name>
(12) <connector_header> ::= "connector" <identifier> ["<" <type_classifier> <identifier> ["," <type_classifier> <identifier> ">"]] [<connector_inheritance_spec>]
(13) <connector_body> ::= "{" <connector_export> "}"
(14) <connector_export> := <provides_dcl> ";" | <uses_dcl> ";"| <extended_port_dcl> ";" | <attr_dcl> ";"
(15) <connector> ::= <connector_header> <connector_body>

See section 7.1.3.1.1 of this document for the specification of <type_classifier>.

See section "Attribute Declaration" of "OMG IDL Syntax and Semantics" [IDL] for the definition of <attr _dcl>.

7.2.3 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 Figure 3, 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 proposed 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, in order 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 **Events** CCM interface is never used in the connector's executor. The reason is that the connector' fragment and the component itself, are only interacting via synchronous calls as they are collocated. The actual interaction semantics between components is carried by connector' fragments themselves.

7.2.3.1 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**.

7.2.3.2 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.

7.2.3.2.1 get_ccm_home

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

7.2.3.2.2 remove

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

7.2.3.3 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
```

7.2.3.3.1 create_component

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

7.2.3.4 Interface HomeConfiguration

7.2.3.4.1 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.

7.2.3.5 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.

7.2.3.6 Connector Implementation Interfaces

This section explains how can be implemented connector fragments.

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 implements 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.

7.2.4 Connector Deployment and Configuration

This section introduces all the modifications to the OMG 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].

7.2.4.1 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 2 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).

7.2.4.2 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.

7.2.4.2.1 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 exists 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[2] that are to be made to the Component Data Model at PSM for CCM level. Actually, two classes are added: **ConnectorPackage-Description**[3] and **ConnectorImplementationDescription**.

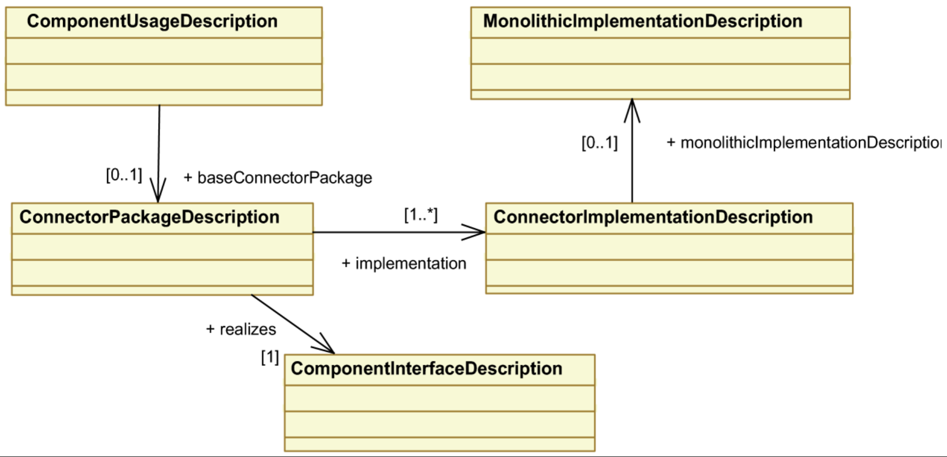
ComponentUsageDescription
MonolithicImplementationDescription
[0..1]
+ monolithicImplementationDescriptio
[0..1]
+ baseConnectorPackage
ConnectorPackageDescription
[1..*]
ConnectorImplementationDescription
+ implementation
+ realizes
[1]
ComponentInterfaceDescription

Figure 6: Revised component data model overview

The two following figures give a detailed description of **ConnectorPackageDescription** and **ConnectorImplementationDescription** classes.

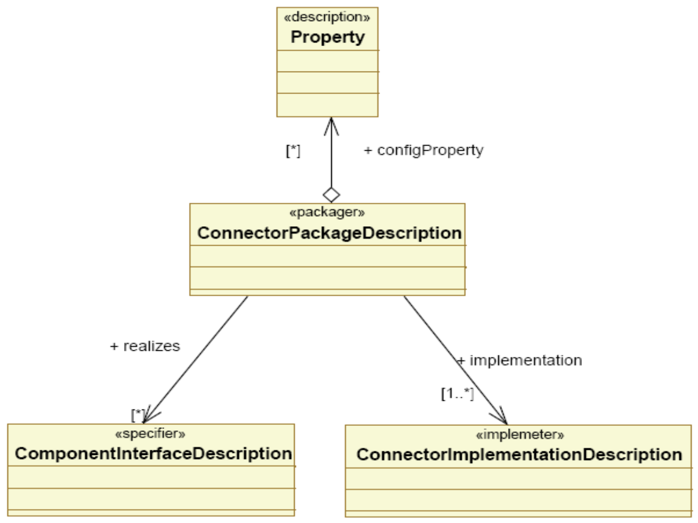

Figure 7: ConnectorPackageDescription Class

[2] 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.

[3] The association between `ComponentUsageDesription` and `ConnectorPackageConfiguration` is in mutual exclusion with those defined in the initial component data model between `ComponentUsageDescription` and `ComponentPackage Description`.

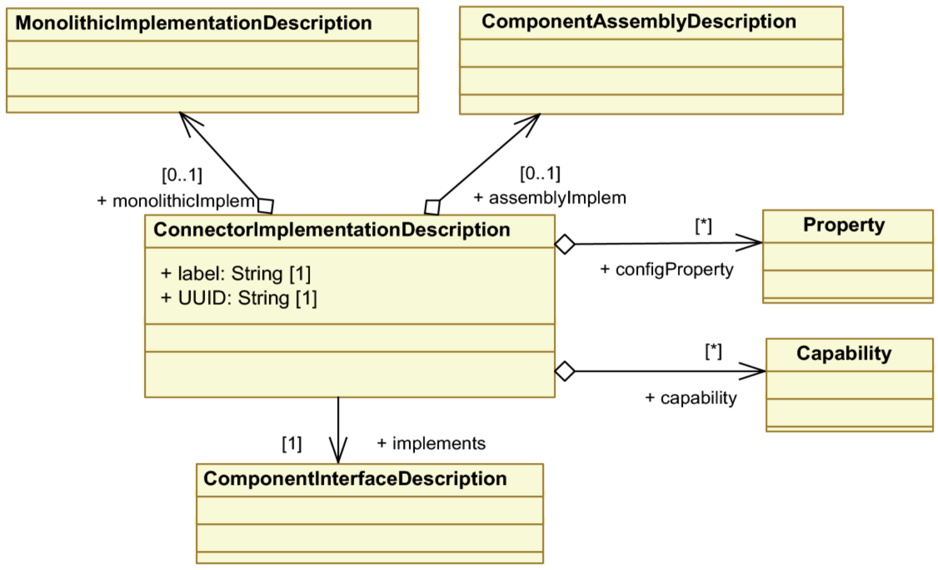

Figure 8: ConnectorImplementationDescription Class

7.2.4.2.2 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.

7.2.4.2.3 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 **CCMPortKind** enumeration is extended with two values: **ExtendedPort**, **MirrorPort** .

7.2.4.2.4 ComponentInterfaceDescription

The added **ComponentPortDescription::templateParam** (String [0..*]) contains all the template parameters 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 **CCMComponentPortKind** attribute shall be **ExtendedPort** or **MirrorPort**

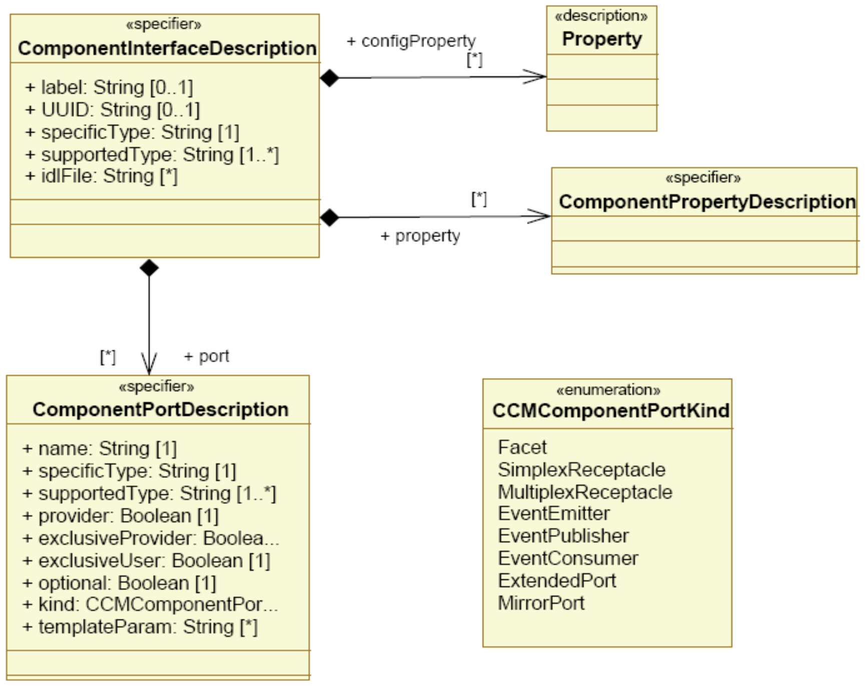

Figure 9: Support for Extended Ports

The connector description will be part of .ccd files.

7.2.4.2.5 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** need 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.

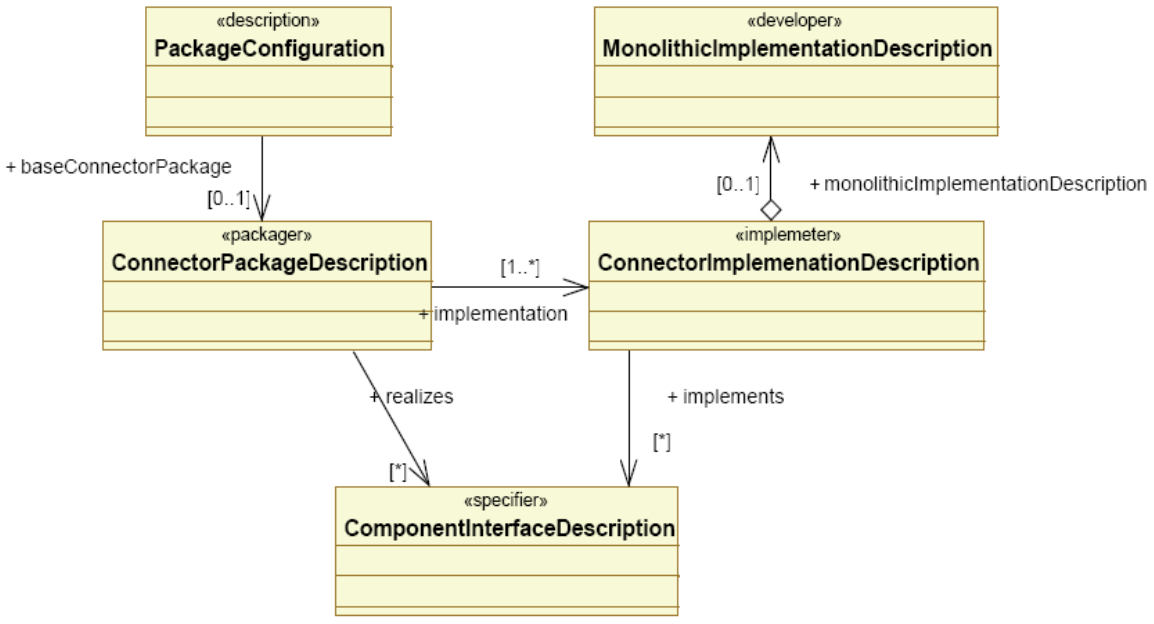
«description»
PackageConfiguration
«developer»
MonolithicImplementationDescription
+ baseConnectorPackage
[0..1]
[0..1]
+ monolithicImplementationDescription
«packager»
ConnectorPackageDescription
«implemeter»
ConnectorImplemenationDescription
[1..*]
+ implementation
+ realizes
+ implements
[*]
[*]
«specifier»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

Figure 10: AssemblyConnectionDescription Extension

7.2.4.3 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 different a 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 typed 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.

7.2.4.3.1 Compliance with Entry Points

This section refers to the 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.

7.2.4.4 Connector Configuration

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

Each fragment of a connector providing an implementation of an extended-port, have to be linked to its dual one, the mirror-port. In some cases, to be configured and linked together, fragment to configure have to query some data from the dual one. Taken into account that, the two fragments can be installed on different nodes, the process of configuration has to be remote.

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>
```

7.2.4.5 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

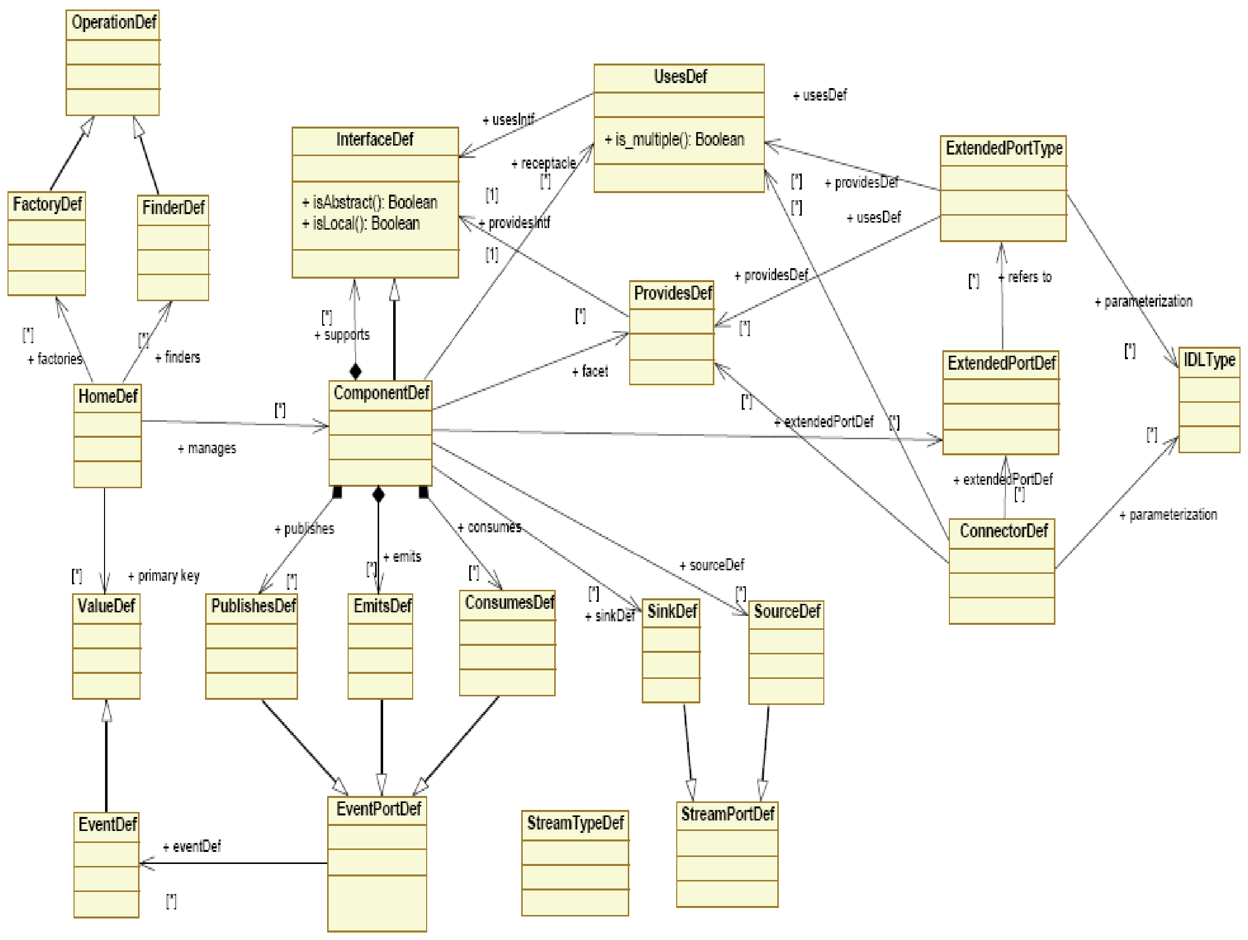

Figure 11: Component Meta-Model – With Extended Ports and Connectors

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**.

In addition to the **ExtendedPortDef** and **ExtendedPortType**, the concept of **connector** is introduced, represented in the meta-model extract below:

8 DDS-DCPS Application

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

8.1 Introduction

8.1.1 Rationale for DDS Extended Ports and Connectors Definition

DDS is a very versatile middleware. It allows to accommodate 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 which may lead to errors or malfunctions due to mistaken uses.

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

- 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)
- 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 (e*ach 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).

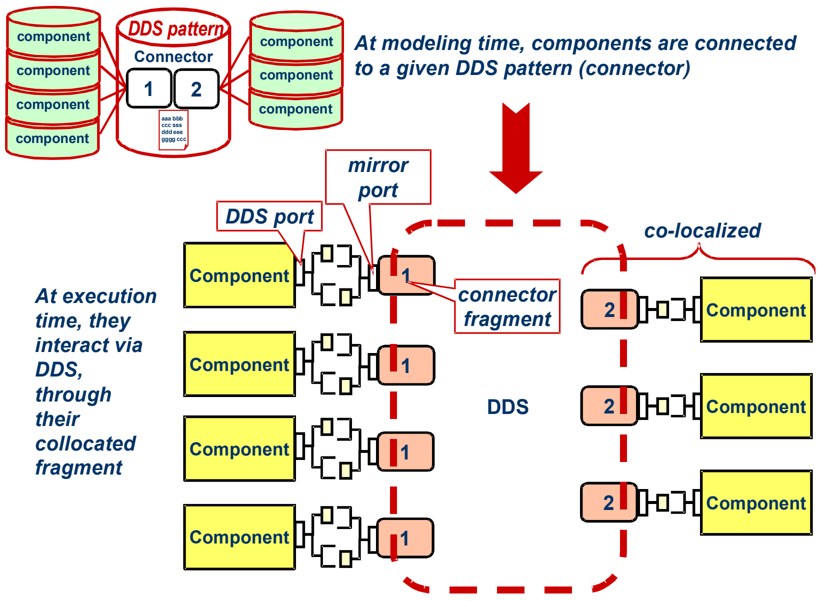

Figure 12: From Modeling to Actual Deployment

8.2 DDS-DCPS Extended Ports

8.2.1 Design Rules

8.2.1.1 Parameterization

DDS-DCPS ports and connectors will be parameterized by 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 to define 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 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 keep only 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.

8.2.1.4 Simplicity versus Richness Trade-off

The goal of this specification is not is not to prevent the advanced user to make 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 a 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 with this starting point.

Note: The proposed DDS-DCPS ports are of large potential usage. However as the Generic Interaction support allows to define new port types, nothing prevents users to define 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 proposed interfaces for basic ports and then assemble them to make the DDS ports.

8.2.2.1 DDS-DCPS Basic Port Interfaces

8.2.2.1.1 Data Access – Publishing Side

Several interfaces allow to write DDS data:

- A **Writer**, allows to publish data on a given topic without paying any attention to the instance lifecycle. Therefore it just allows writing values of the related data type.
- An **Updater** allows to publish data on a given topic when you do care of instance lifecycle. Therefore it allows creating, updating and deleting instances of the related data type. It can be configured to actually check the lifecycle or not.
- A **MultiWriter** is a **Writer** which doesn't act on instances one by one (as a **Writer** does), but per group (sequences). In addition, it may be configured so that each of its actions is 'coherent' as far DDS is concerned (i.e. embedded in a couple of **begin_coherent_updates**, **end_coherent_updates**)
- A **MultiUpdater** is an **Updater** which doesn't act non on instances one by one (as an **Updater** does), but per group (sequences). In addition, it may be configured so that each of its actions is 'coherent' as far DDS is concerned (i.e. embedded in a couple of **begin_coherent_updates**, **end_coherent_updates**)

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

Interface Writer

```
interface Writer<typename T> {              // T assumed to be a data type
        void write (in T an_instance)
                raises (InternalError);
        };
```

Interface MultiWriter

```
interface MultiWriter<typename T> {         // T assumed to be a data type
        typedef sequence<T> T$Seq;
        unsigned long write(in T$Seq instances)  // returns nb of written
                raises (InternalError);
        attribute boolean is_coherent_write;
        };
```

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

- If **is_coherent_write** is TRUE, the write orders are embedded between DDS **begin_coherent_updates** and a **end_coherent_updates**
- The write orders are stopped at the first error (and the index of the erroneous instance is reported in the raised exception)

Interface Updater

```
interface Updater<typename T> {            // T assumed to be a data type
        void create (in T an_instance)
                raises (AlreadyCreated,
                        InternalError);
        void update (in T an_instance)
                raises (NonExistent,
                        InternalError);
        void delete (in T an_instance)
                raises (NonExistent,
                        InternalError);
        readonly attribute boolean is_lifecycle_checked;
        };
```

Behavior of an **Updater** is as follows:

- If **is_lifecycle_checked** is TRUE, then create checks that the instance is not already existing and update and delete that the instance is existing; **AlreadyCreated** and **NonExistent** exceptions may be raised.
- If **is_lifecycle_checked** is FALSE, then those checks are not performed.

Note: This check 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 it should be restricted to architectures where a single writer is involved.

Interface MultiUpdater

```
interface MultiUpdater<typename T> {            // T assumed to be a data type
        typedef sequence<T> T$Seq;
        unsigned long create (in T$Seq instances)       // returns nb of created instances
                raises (AlreadyCreated,
                        InternalError);
        unsigned long update (in T$Seq instances)       // returns nb of updated instances
                raises (NonExistent,
                        InternalError);
        unsigned long delete (in T$Seq instances)       // returns nb of deleted instances
                raises (NonExistent,
                        InternalError);
        readonly attribute boolean is_lifecycle_checked;
        attribute boolean is_coherent_write;
        };
```

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

- If **is_lifecycle_checked** is TRUE, then create checks that the instances are not already existing and update and delete that the instances are existing; **AlreadyCreated** and **NonExistent** exceptions may be raised.
- These checks are performed before any attempt to write or dispose and are applied to all the submitted instances. All the erroneous instances are reported in the exceptions (by means of their index in the submitted sequence)
- If **is_lifecycle_checked** is FALSE, then those checks are not performed.

- If **is_coherent_write** is TRUE, the write orders are embedded between DDS **begin_coherent_updates** and a **end_coherent_updates**
- The write or dispose orders are stopped at the first error (and the index of the erroneous instance is reported in the raised exception)

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 to retrieve data values from DDS data readers.

- A **Reader** allows to read one or several instance values on a given topic according to a given criterion, with no wait.
- A **Getter** allows to get one or several new values on a given topic according to a given criterion. It may block to get the proper information.
- A **RawListener** allows to get pushed with a new value on a given topic, according to a given criterion, regardless the instance status.
- A **StateListener** allows to get pushed with a new value on a given topic, according to a given criterion; Different operations are called depending on the instance state.
- A **MultiListener** allows to get pushed with a sequence of new values on a given topic, according to a given criterion.

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_UPDATED,
        INSTANCE_DELETED
        };

struct ReadInfo {
        AccessStatus        access_status;
        InstanceStatus      instance_status;
        DDS::Time_t         timestamp;
        unsigned long       instance_rank;
        };

typedef sequence<ReadInfo> ReadInfoSeq;
```

ReadInfo is the simplified version of DDS **SampleInfo**. Each read or gotten is accompanied with a **ReadInfo** which specifies:

- Whether the value has already been seen or not by the component (**AccessStatus**)
- The instance status (**InstanceStatus**)
- The DDS **timestamp**
- The belonging instance rank within the returned sequence (**instance_rank**)

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                  query;
        StringSeq               query_parameters
        };
```

QueryFilter gathers in a single structure a query and its related parameters. The **QueryFilter** attribute placed on the following interfaces acts as a filter for the read or get operations. A void **query** means no query.

Interface Reader

```
interface Reader <typename T> {                 // T assumed to be a data type
        typedef sequence<T> T$Seq;
        void read_all (out T$Seq instances, out ReadInfoSeq infos)
                raises (InternalError);
        void read_all_history (out T$Seq instances, out ReadInfoSeq infos)
                raises (InternalError);
        void read_one (inout T an_instance, out ReadInfo info)
                raises (NonExistent,
                        InternalError);
        void read_one_history (in T an_instance,
                            out T$Seq instances, out ReadInfoSeq infos)
                raises (NonExistent,
                        InternalError);
        attribute QueryFilter filter
                setraises (BadParameter);
        };
```

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 **filter** ("" means no query)
- **read_all** returns the last sample of all instances
- **read_all_history** returns all samples of all instances
- **read_one** returns the last sample of a given instance; parameter **an_instance** is supposed to be passed filled with the key value and will be passed back with all its fields.
- **read_one_history** returns all the samples of a given instance; parameter **an_instance** is supposed to be passed filled with the key value.

Interface Getter

```
interface Getter<typename T> {
        typedef sequence<T> T$Seq;
        boolean get_all (out T$Seq instances, out ReadInfoSeq infos)
                raises (InternalError);
        boolean get_all_history (out T$Seq instances, out ReadInfoSeq infos)
                raises (InternalError);
        boolean get_one (inout T an_instance, out ReadInfo info)
                raises (NonExistent,
                        InternalError);
        boolean get_one_history (in T an_instance,
                             out T$Seq instances, out ReadInfoSeq infos)
                raises (NonExistent,
                        InternalError);
        boolean get_next (out T an_instance, out ReadInfo info)
                raises (InternalError);
        attribute QueryFilter           filter
                setraises (BadParameter);
        attribute DDS::Duration_t       time_out;
        };
```

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

- **Get** operations are performed with the following parameters
    - **SampleStateMask**: NO_READ
    - **ViewStateMask**: NEW or NOT_NEW
    - **InstanceStateMask**: ALIVE or NOT_ALIVE
    - Through the query as specified in the **filter** ("" means no query)
    - Within the time limit specified in **time_out**
- They all receives as result a **boolean** that indicates whether actual data are provided (**true**) or if the time-out occurred (**false**).
- **get_all** returns the last sample of all instances.
- **get_all_history** returns all samples of all instances.
- **get_one** returns the last sample of a given instance; parameter **an_instance** is supposed to be passed filled with the key value and will be passed back with all its fields.
- **get_one_history** returns all the samples of a given instance; parameter **an_instance** is supposed to be passed filled with the key value.
- **get_next** returns each gotten samples, one by one.

Note: **get_all** or **get_all_history** are especially useful in the application init phase.

Interface RawListener

```
interface RawListener<typename T> {
        void on_data (in T an_instance, in ReadInfo info);
        };
```

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

- The semantics of **on_data** is similar to the one of **Getter::get_next**, except that it is in push mode instead of pull mode

- Query filter (if any) will be found in the associated **Reader** (see below – definition of DDS extended ports)

Interface StateListener

```
interface StateListener<typename T> {
        void on_creation (in T an_instance, in DDS::Time_t timestamp);
        void on_update (in T an_instance, in DDS::Time_t timestamp);
        void on_deletion (in T an_instance, in DDS::Time_t timestamp);
        };
```

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

- The semantics of the operations is similar to the one of **Getter::get_next**, except that it is in push mode instead of pull mode and that the exact operation called depends on the state of the instance:
- on_creation is called when a new instance is delivered,
- on_update is called when an already existing instance is updated:
- on_deletion is called when an instance is said as non longer alive by underlying DDS; in this case, the only fields valid in the provided instance parameter are the inees that form the key.
- Query filter (if any) will be found in the associated **Reader**.

Interface MultiListener

```
enum GroupingMode {
        INSTANCE_HISTORY,
        LAST_SAMPLE_ALL_INSTANCES,
        ALL_SAMPLES_ALL_INSTANCES
        };

interface MultiListener<typename T> {
        typedef sequence<T> T$Seq;
        void on_data (in T$Seq instances, in ReadInfoSeq infos);
        attribute GroupingMode grouping_mode;
        };
```

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

- **On_data** is here called with a sequence of new values. Depending of **grouping_mode**, each sequence will contain 1) all the samples of an instance, 2) all new last samples or 3) all new samples.
- Query filter (if any) will be found in the associated **Reader** (see below – definition of DDS extended ports).

8.2.2.1.3 Listener Control

The following interface allows to enable the listeners that are attached to the port. The initial value of the attribute is false meaning that the listeners are a priori not enabled.

Interface ListenerControl

```
interface ListenerControl {
        attribute boolean enabled;
        };
```

8.2.2.1.4 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 ones 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 ones are made available through a **ConnectorStatusListener**
- The last ones are kept for internal implementation of connectors fragments and therefore not reported.

Interface PortStatusListener

```
    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

```
    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);
            };
```

All the operations of those two listeners mimic exactly the related DDS ones, 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<typename T> {
            uses Writer<T> data;
            uses DDS::DataWriter dds_entity;
            };
    porttype DDS_MultiWrite<typename T> {
            uses MultiWriter<T> data;
            uses DDS::DataWriter dds_entity;
            };

    porttype DDS_Update <typename T> {
            uses Updater<T> data;
            uses DDS::DataWriter dds_entity;
            };

    porttype DDS_MultiUpdate <typename T> {
            uses MultiUpdater<T> data;
            uses DDS::DataWriter dds_entity;
            };

    porttype DDS_Read<typename T> {
            uses Reader<T> data;
            uses DDS::DataReader dds_entity;
            provides PortStatusListener status;
            };

    porttype DDS_Get<typename T> {
            uses Getter<T> data;
            uses DDS::DataReader dds_entity;
            provides PortStatusListener status;
            };

    porttype DDS_RawListen<typename T> {
            uses Reader<T> data;
            uses ListenerControl control;
            provides RawListener<T> listener;
            uses DDS::DataReader dds_entity;
            provides PortStatusListener status;
            };

    porttype DDS_StateListen<typename T> {
            uses Reader<T> data;
            uses ListenerControl control;
            provides StateListener<T> listener;
            uses DDS::DataReader dds_entity;
            provides PortStatusListener status;
            };

    porttype DDS_MultiListen<typename T> {
            uses Reader<T> data;
            uses ListenerControl control;
            provides MultiListener<T> listener;
            uses DDS::DataReader dds_entity;
            provides PortStatusListener status;
            };
```

All proposed DDS ports combine at least a basic port to access data with a basic port to access underlying DDS entity. **DDS_RawListen**, **DDS_StateLsisten** and **DDS_MultiListen** split the data access functionality in two ports; the first one (**Reader**) is there to set the read criterion and provide operations for the init phase, while the second one is rather intended to be used in the application processing loop. All the ports intended for subscribing side comprise also a port to be notified of the relevant statuses.

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 which allow to specify, all the elements that are needed to properly instantiate them, namely:

- The name of the DDS Topic which 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).

Having all these 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 read only attributes to store the Domain identifier and the QoS profile (c.f. section 8.4.2 for more details on QoS profile). The QoS profile could be given either as a file URL or as the XML string itself.

All DDS connectors should inherit from that base.

```
connector DDS_Base [
        uses ConnectorStatusListener             error_listener;
        readonly attribute DDS:DomainId_t        domain_id;
        readonly attribute string                qos_profile;     // File URL or XML string
        };
```

DDS_TopicBase extends the **DDS_Base** with the name of one topic. they should be the base for all mono-topic connectors

```
connector DDS_TopicBase : DDS_Base {
        readonly attribute string                topic_name;
        readonly attributre StringSeq            key_fields;
        };
```

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

The connector definition is as follows:

```
    connector DDS_State<typename T> : DDS_TopicBase {     // T assumed to be a data type
            mirrorport DDS_Update<T>              observable;
            mirrorport DDS_Read<T>                passive_observer;
            mirrorport DDS_Get<T>                 pull_observer;
            mirrorport DDS_StateListen<T>         push_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 other 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<typename T> : DDS_TopicBase {    // T assumed to be a data type
            mirrorport DDS_Write<T>               supplier;
            mirrorport DDS_Get<T>                 pull_consumer;
            mirrorport DDS_Listen<T>              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 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 a XML string and can gather *QoS*[4] 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

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

default values QoS policies as specified in [DDS], are in annexes and , 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_ccm>** and end with **</dds_ccm>**.
- The primitive types for tag values are specified in the following table:

Table 12: QoS Profile: Supported Tag Values

Type	Format	Notes
Boolean	**yes**, **1**, **true** or **BOOLEAN_TRU**E: these all mean TRUE	Not case-sensitive
	no, **0**, **false** or **BOOLEAN_FALSE**: these all mean 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>**

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 **DDS_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.

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 required 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_SEC</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>DDS_KEEP_ALL_HISTORY_QOS</kind>
		</history>
		<reliability>
			<kind>DDS_RELIABLE_RELIABILITY_QOS</kind>
		</reliability>
	</datawriter_qos>
	<datareader_qos>
		<history>
			<kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
		</history>
		<reliability>
			<kind>DDS_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, which 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>DDS_KEEP_ALL_HISTORY_QOS</kind>
		</history>
		<reliability>
			<kind>DDS_RELIABLE_RELIABILITY_QOS</kind>
		</reliability>
	</datawriter_qos>
	<datawriter_qos topic_filter="B*">
		<history>
			<kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
```

```
				</history>
				<reliability>
					<kind>DDS_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>DDS_KEEP_ALL_HISTORY_QOS</kind>
		</history>
	</datawriter_qos>
```

is equivalent to the following:

```
	<qos_profile name="KeepAllWriter">
		<writer_qos>
			<history>
				<kind>DDS_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 which 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 section instantiates the Generic Interaction Support described in section 7, in order to define ports and connectors for DDS-DLRL. This section assumes an a-priori knowledge of DDS specification (in particular of 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 at 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 contains 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.

```
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 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 DDS_CCM::CacheOperation  cache;
        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 DCPOS entities (if specified in the connector),
- Enable the infrastructure so that DLRL objects can be created and used DLRL way.

9.3 DDS-DLRL Connectors

A DLRL connector is also essentially variable in its composition for it contains as many mirror ports as there are different object models to share the related topics. It 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 : DDS_CCM: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 8.4.2and 8.4.3), 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 likely to be used

Annex A

(normative)

IDL3+ of DDS-DCPS Ports and Connectors

```
#include "dds_rtf2_dcps.idl"

module CCM_DDS {

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

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

exception InternalError{
            unsigned long error_code;// DDS codes that are relevant:
                                     // ERROR (1); UNSUPPORTED (2); OUT_OF_RESOURCE (5)
            unsigned long index;     // of the erroneaous
            };

exception BadParameter {};

// ------------------------------------
// Interfaces to be 'used' or 'provided'
// ------------------------------------

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

interface Writer<typename T> {                               // T assumed to be a data type
            void write (in T an_instance)
                        raises (InternalError);
            };

interface MultiWriter<typename T> {
            typedef sequence<T> T$Seq;
            unsigned long write(in T$Seq instances)  // returns nb of written
                        raises (InternalError);
            attribute boolean is_coherent_write;
            // behaviour:
            // - attempt to write is stopped at the first error
            // - if is_coherent_write, write orders are placed betwen begin/end
            //          coherent updates (even if an error occurs)
            };
```

```
interface Updater<typename T> {
          void create (in T an_instance)
                    raises (AlreadyCreated,
                              InternalError);
          void update (in T an_instance)
                    raises (NonExistent,
                              InternalError);
          void delete (in T an_instance)
                    raises (NonExistent,
                              InternalError);
          readonly attribute boolean is_lifecycle_checked;
          // behaviour:
          // - exceptions AlreadyCreated or NonExistent are raised only if
          //        is_lifecycle_checked
          // - note: this check requires to previously attempt to read (not free)
          // - note: this check is not 100% guarantee as a creation or a deletion may
          //        occur between the check and the actual write od dispose order
          };

interface MultiUpdater<typename T> {
          typedef sequence<T> T$Seq;
          unsigned long create (in T$Seq instances)        // returns nb of created instances
                    raises (AlreadyCreated,
                              InternalError);
          unsigned long update (in T$Seq instances)        // returns nb of updated instances
                    raises (NonExistent,
                              InternalError);
          unsigned long delete (in T$Seq instances)        // returns nb of deleted instances
                    raises (NonExistent,
                              InternalError);
          readonly attribute boolean is_lifecycle_checked;
          attribute boolean is_coherent_write;
          // behaviour:
          // -  exceptions AlreadyCreated or NonExistent are raised only if
          //        is_lifecycle_checked
          // - global check is performed before actual write or dispose
          //        (in case of error, all the erroneous instances are reported
          //        in the exception)
          // - attempt to write or dispose is stopped at the first error
          // - if is_coherent_write, write orders are placed betwen begin/end
          //        coherent updates (even if an error occurs)
          };

// Data access - subscribing side
// -----------------------------
//        read => no wait
//        get  => wait

enum AccessStatus {
          FRESH_INFO,
          ALREADY_SEEN
          };

enum InstanceStatus {
          INSTANCE_CREATED,
          INSTANCE_UPDATED,
          INSTANCE_DELETED
          };
```

```
struct ReadInfo {
        AccessStatus    access_status;
        InstanceStatus  instance_status;
        DDS::Time_t     timestamp;
        unsigned long   instance_rank;
        };
typedef sequence<ReadInfo> ReadInfoSeq;

struct QueryFilter {
        string                  query;
        StringSeq       query_parameters;
        };

interface Reader <typename T> {
        typedef sequence<T> T$Seq;
        void read_all (out T$Seq instances, out ReadInfoSeq infos)
                raises (InternalError);
        void read_all_history (out T$Seq instances, out ReadInfoSeq infos)
                raises (InternalError);
        void read_one (inout T an_instance, out ReadInfo info)
                raises (NonExistent,
                        InternalError);
        void read_one_history (in T an_instance,
                            out T$Seq instances, out ReadInfoSeq infos)
                raises (NonExistent,
                        InternalError);
        attribute QueryFilter filter
                setraises (BadParameter);
        // behaviour
        // - read operations are performed with the following parameters
        //      - READ or NO_READ
        //      - NEW or NOT_NEW
        //      - ALIVE
        //      - through the query as specified in the filter ("" means no query)
        // - data returned:
        //      - read_all returns for each living instance, its last sample
        //        ordered by instance first and then by sample
        //      - read_all_history returns all the samples of all instances
        //        ordered by instance first and then by sample
        //      - read_one returns the last sample of the given instance
        //      - read_one_history returns all the samples for the given instance
        };
```

```
interface Getter<typename T> {
        typedef sequence<T> T$Seq;
        boolean get_all (out T$Seq instances, out ReadInfoSeq infos)
                raises (InternalError);
        boolean get_all_history (out T$Seq instances, out ReadInfoSeq infos)
                raises (InternalError);
        boolean get_one (inout T an_instance, out ReadInfo info)
                raises (NonExistent,
                        InternalError);
        boolean get_one_history (in T an_instance,
                             out T$Seq instances, out ReadInfoSeq infos)
                raises (NonExistent,
                        InternalError);
        boolean get_next (out T an_instance, out ReadInfo info)
                raises (InternalError);
        attribute QueryFilter           filter
                setraises (BadParameter);
        attribute DDS::Duration_t       time_out;
        // behaviour
        // - 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 filter ("" means no query)
        //      - within the time limit specified in time_out
        // - all operations returns TRUE if data are provided,
        //      FALSE if time-out occurred
        // - data returned:
        //      - get_all returns for all the instances their last sample
        //      - get_all_history returns all the samples of all instances
        //      - get_one returns the last sample of the given instance
        //      - get_one_history returns all the samples for the given instance
        //      - get_next returns each read sample one by one
        };

interface RawListener<typename T> {
        void on_data (in T an_instance, in ReadInfo info);
        // behaviour
        // - similar to a get_next, except that in push mode instead of pull mode
        // - triggered only if enabled is the associated ListenerControl
        // - query filter (if any) in the associated Reader
        };

interface StateListener<typename T> {
        void on_creation (in T an_instance, in DDS::Time_t timestamp);
        void on_update (in T an_instance, in DDS::Time_t timestamp);
        void on_deletion (in T an_instance, in DDS::Time_t timestamp);
        // behaviour
        // - similar to a get_next, except that different operations are called
        //      depending on the instance state
        // - triggered only if enabled is the associated ListenerControl
        // - query filter (if any) in the associated Reader

enum GroupingMode {
        INSTANCE_HISTORY,
        LAST_SAMPLE_ALL_INSTANCES,
        ALL_SAMPLES_ALL_INSTANCES
        };
```

```
interface MultiListener<typename T> {
	typedef sequence<T> T$Seq;
	void on_data (in T$Seq instances, in ReadInfoSeq infos);
	attribute GroupingMode grouping_mode;
	// behaviour
	// - depending on grouping_mode similar to get_one_history(any new instance),
	// get_all or get_all_history, except that in push mode instead of
	// pull mode
	// - triggered only if enabled is the associated ListenerControl
	// - query filter (if any) in the associated Reader
	};

interface ListenerControl {
	attribute boolean enabled;
	};

// Status Access
// -------------

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 { // 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);
	};

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

porttype DDS_Write<typename T> {
	uses Writer<T> data;
	uses DDS::DataWriter dds_entity;
	};

porttype DDS_MultiWrite<typename T> {
	uses MultiWriter<T> data;
	uses DDS::DataWriter dds_entity;
	};

	porttype DDS_Update <typename T> {
		uses Updater<T> data;
		uses DDS::DataWriter dds_entity;
		};

	porttype DDS_MultiUpdate <typename T> {
		uses MultiUpdater<T> data;
		uses DDS::DataWriter dds_entity;
		};

	porttype DDS_Read<typename T> {
		uses Reader<T> data;
		uses DDS::DataReader dds_entity;
		provides PortStatusListener status;
		};

	porttype DDS_Get<typename T> {
		uses Getter<T> data;
		uses DDS::DataReader dds_entity;
		provides PortStatusListener status;
		};

	porttype DDS_RawListen<typename T> {
		uses Reader<T> data;
		uses ListenerControl control;
		provides RawListener<T> listener;
		uses DDS::DataReader dds_entity;
		provides PortStatusListener status;
		};

	porttype DDS_StateListen<typename T> {
		uses Reader<T> data;
		uses ListenerControl control;
		provides StateListener<T> listener;
		uses DDS::DataReader dds_entity;
		provides PortStatusListener status;
		};

	porttype DDS_MultiListen<typename T> {
		uses Reader<T> data;
		uses ListenerControl control;
		provides MultiListener<T> listener;
		uses DDS::DataReader dds_entity;
		provides PortStatusListener status;
		};

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


	connector DDS_Base {
		provides ConnectorStatusListener			error_listener;
		readonly attribute DDS::DomainId_t			domain_id;
		readonly attribute string			qos_profile;	// File URL or XML string
		};

	connector DDS_TopicBase : DDS_Base {
		readonly attribute string		topic_name;
		readonly attribute StringSeq	key_fields;
		};
```

```
connector DDS_State<typename T> : DDS_TopicBase {     // T assumed to be a data type
            mirrorport DDS_Update<T>                        observable;
            mirrorport DDS_Read<T>                          passive_observer;
            mirrorport DDS_Get<T>                           pull_observer;
            mirrorport DDS_StateListen<T>   push_observer;
            };

connector DDS_Event<typename T> : DDS_TopicBase {    // T assumed to be a data type
            mirrorport DDS_Write<T>                         supplier;
            mirrorport DDS_Get<T>                           pull_consumer;
            mirrorport DDS_Listen<T>                        push_consumer;
            };
```

Annex B

(normative)

IDL for DDS-DLRL Ports and Connectors

```
#include "dds_rtf2_dlrl.idl"

module DDS_CCM {

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);
        };
};
```

Annex C

(normative)

XML Schema for QoS Profiles

```
<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

(non normative)

Default QoS Profile

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_ccm 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_ccm>
```

Annex E

(non normative)

QoS Policies for the DDS Patterns

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 section 8.3.2 and Event Transfer Pattern as defined in section 8.3.3)

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_UNLIMITED		service_cleanup_delay=0 history_kind=KEEP_LAST history_depth=1 max_*=LENGTH_UNLIMITED		
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_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		

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					