DDS Extensible Type and Serialization Framework

Version 1.4

OMG Document Number: ptc/2026-02-06

Date: February 2026

Standard document URL: https://www.omg.org/spec/DDS-XTypes/1.4/

Normative Machine Consumable File(s):

https://www.omg.org/spec/DDS-XTypes/20260301/dds-xtypes_model.xmi

https://www.omg.org/spec/DDS-XTypes/20260301/dds-xtypes_typeobject.idl

https://www.omg.org/spec/DDS-XTypes/20260301/dds-xtypes_discovery_builtin_topics.idl

https://www.omg.org/spec/DDS-XTypes/20260301/dds-xtypes_dynamicdate.idl

---

Copyright ©2008-2026, Object Management Group, Inc.

Copyright ©2008-2026, Real-Time Innovations, Inc.

Copyright ©2008-2020, ADLINK Ltd.

Copyright ©2026, ZettaScale Technology.

Copyright ©2008-2026, Twin Oaks Computing, Inc.

Copyright ©2008-2026, Object Computing, Inc.

Copyright ©2019-2026, Kongsberg Computing, Inc.

Copyright ©2019-2026, Rocket Software Computing, Inc.

Copyright ©2019-2026, Unity Foundation NP.

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 109 Highland Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

C®, CORBA®, CORBA logos®, FIBO®, Financial Industry Business Ontology®, FINANCIAL INSTRUMENT GLOBAL IDENTIFIER®, IIOP®, IMM®, Model Driven Architecture®, MDA®, Object Management Group®, OMG®, OMG Logo®, SoaML®, SOAML®, SysML®, UAF®, Unified Modeling Language®, UML®, UML Cube logo®, VSIPL®, and XMI® are registered trademarks of the Object Management Group, Inc.

For a complete list of trademarks, see: https://www.omg.org/legal/tm_list.htm. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG's Issue Reporting

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: https://issues.omg.org/issues/create-new-issue.

Table of Contents

DDS Extensible Type and Serialization Framework i
Table of Contents v
Tables xi
Figures xiv
Preface xvi
1. Scope 1
2. Conformance Criteria 3
2.1 Programming Interface Profiles 4
2.1.1 Static Programming Interface Profile 4
2.1.2 Dynamic Programming Interface Profile 4
2.2 Network Interoperability Profiles 5
2.2.1 Minimal Network Interoperability Profile 5
2.2.2 Legacy Network Interoperability Profile 5
2.2.3 Safe Network Interoperability Profile 5
2.3 Optional XML Type Representation Profile 6
2.4 XML Data Representation Profile 6
3. Normative References 7
4. Terms and Definitions 9
5. Symbols 11
6. Additional Information 13
6.1 Acknowledgments 13
7. Extensible and Dynamic Topic Types for DDS 17
7.1 Overview 17
7.2 Type System 19
7.2.1 Overview (Informative) 19
7.2.1.1 Type Evolution Example 22
7.2.1.2 Type Inheritance Example 23
7.2.1.3 Sparse Types Example 24

7.2.1.4 Annotation Example 24
7.2.2 Attributes and Metadata 25
7.2.2.1 Type Kind 25
7.2.2.2 Type Naming 25
7.2.2.3 Namespaces and Modules 27
7.2.2.4 Nested Types 27
7.2.2.5 Documentation 27
7.2.2.6 License Information 28
7.2.2.7 Default Value 31
7.2.2.8 Try Construct behavior 31
7.2.2.9 Units 34
7.2.2.10 Range Limits 35
7.2.2.11 External Data 36
7.2.2.12 Annotation Application 37
7.2.3 Type Categories 38
7.2.3.1 Scalar Types 38
7.2.3.2 Constructed Types 47
7.2.3.3 Annotation Definition 77
7.2.4 Type Extensibility and Mutability 78
7.2.5 Type Compatibility 79
7.2.5.1 Constructing objects of one type from objects of another type 80
7.2.5.2 Concept of Delimited Types 81
7.2.5.3 Strong Assignability 81
7.2.5.4 Type Assignability Options 81
7.2.5.5 Assignability Rules 85
7.3 Type Representation 102
7.3.1 IDL Type Representation 104
7.3.1.1 IDL Compatibility 104
7.3.1.2 Representing Attributes and Metadata 106
7.3.1.3 Annotation Language 106

7.3.1.4 Constants and Expressions 125
7.3.1.5 Primitive Types 126
7.3.1.6 Alias Types 126
7.3.1.7 Array and Sequence Types 126
7.3.1.8 String Types 126
7.3.1.9 Fixed Point Decimal Types 126
7.3.1.10 Enumerated Types 126
7.3.1.11 Map Types 127
7.3.1.12 Structure Types 127
7.3.1.13 Union Types 128
7.3.2 XML Type Representation 129
7.3.2.1 Type Representation Management 129
7.3.2.2 Representing Attributes and Metadata 131
7.3.2.3 Basic Types 133
7.3.2.4 String Types 134
7.3.2.5 Fixed Point Decimal Types 134
7.3.2.6 Collection Types 135
7.3.2.7 Aggregated Types 138
7.3.2.8 Aliases 141
7.3.2.9 Enumerated Types 141
7.3.2.10 Modules 142
7.3.2.11 Custom Annotations 142
7.3.2.12 Builtin Annotations 143
7.3.3 Representing Types with TypeIdentifier and TypeObject 145
7.3.3.1 Plain Types 145
7.3.3.2 Type Identifier 145
7.3.3.3 Complete TypeObject 147
7.3.3.4 Encoding of Member Attributes in TypeObject 147
7.3.3.5 Minimal TypeObject 148
7.3.3.6 TypeObject serialization 148

7.3.3.7 Classification of TypeIdentifiers 149
7.3.3.8 Type Equivalence 151
7.3.3.9 Types with mutual dependencies on other types 152
7.3.3.10 Computation of Type identifiers for types with mutual dependencies 154
7.3.3.11 Representing License Information 161
7.4 Data Representation 162
7.4.1 Extended CDR Representation (encoding version 1) 164
7.4.1.1 PLAIN_CDR Encoding 164
7.4.1.2 Parameterized CDR Encoding 168
7.4.1.3 Issues and Limitations 174
7.4.2 Extended CDR Representation (encoding version 2) 174
7.4.3 Extended CDR encoding virtual machine 175
7.4.3.1 Encoding version and format 175
7.4.3.2 XCDR Stream State 176
7.4.3.3 Type and Byte transformations 178
7.4.3.4 Functions related to data types and objects 180
7.4.3.5 Encoding (serialization) rules 182
7.4.4 XML Data Representation 195
7.4.4.1 Schema Qualified XML Data Representation 195
7.4.4.2 Compact XML Data Representation 196
7.4.4.3 Mapping of DDS Data Values to XML 197
7.4.4.4 XML Data Validation Schemas 197
7.5 Language Binding 198
7.5.1 Plain Language Binding 199
7.5.1.1 Primitive Types 200
7.5.1.2 Annotations and Built-in Annotations 202
7.5.1.3 Map Types 213
7.5.1.4 Structure and Union Types 222
7.5.2 Dynamic Language Binding 223
7.5.2.1 UML-to-IDL Mapping Rules 224

7.5.2.2 DynamicTypeBuilderFactory 225
7.5.2.3 AnnotationDescriptor 231
7.5.2.4 TypeDescriptor 233
7.5.2.5 MemberId 236
7.5.2.6 DynamicTypeMember 236
7.5.2.7 MemberDescriptor 238
7.5.2.8 DynamicType 241
7.5.2.9 DynamicTypeBuilder 246
7.5.2.10 DynamicDataFactory 249
7.5.2.11 DynamicData 251
7.6 Use of the Type System by DDS 256
7.6.1 Topic Model 256
7.6.2 Types that may be associated with a DDS Topic 257
7.6.3 Discovery and Endpoint Matching 257
7.6.3.1 Data Representation QoS Policy 258
7.6.3.2 Discovery Built-in Topics 265
7.6.3.3 Built-in TypeLookup service 268
7.6.3.4 Type Consistency Enforcement QoS Policy 277
7.6.4 Local API Extensions 279
7.6.4.1 Operation: `DomainParticipant::create_topic` 279
7.6.4.2 Operation: `DomainParticipant::lookup_topicdescription` 279
7.6.5 Built-in Types 280
7.6.5.1 String 280
7.6.5.2 KeyedString 280
7.6.5.3 Bytes 281
7.6.5.4 KeyedBytes 281
7.6.6 Use of Dynamic Data and Dynamic Type 281
7.6.6.1 Type Support 281
7.6.6.2 `DynamicDataWriter` and `DynamicDataReader` 284

7.6.7 DCPS Queries and Filters 284
7.6.7.1 Member Names 284
7.6.7.2 Optional Type Members 285
7.6.7.3 Grammar Extensions 285
7.6.8 Interoperability of Keyed Topics 286
7.6.8.1 Keyhash Computation 286
7.6.8.2 Serialized representation of the key fields 286
Annex A: XML Type Representation Schema 287
Annex B: Representing Types with TypeObject 288
Annex C: Dynamic Language Binding 290
Annex D: DDS Built-in Topic Data Types 292
Annex E: Built-in Types 293
Annex F: Characterizing Legacy DDS Implementations 299
F.1 Type System 299
F.2 Type Representation 299
F.3 Data Representation 299
F.4 Language Binding 300

Tables

Table 1 – Major type-related concerns in DDS and mechanisms defined by this specification .. 20
Table 2 – TryConstruct examples 33
Table 3 – TryConstruct behavior kinds 34
Table 4 – Primitive Types 40
Table 5 – Enumerated types 49
Table 6 – Bitmask types 50
Table 7 – Alias types 53
Table 8 – Collection Types 58
Table 9 – Aggregated Types 60
Table 10 – Default values for non-optional members 65
Table 11 – Impact of the extensibility kind 79
Table 12 – Type assignability example 80
Table 13 – Definition of the *is-assignable-from* relationship for alias types 85
Table 14 – Definition of the *is-assignable-from* relationship for primitive types 86
Table 15 – Definition of the is-assignable-from relationship for string types 87
Table 14 – Definition of the *is-assignable-from* relationship for fixed point types 89
Table 16 – Definition of the *is-assignable-from* relationship for collection types 89
Table 17 – Definition of the *is-assignable-from* relationship for bitmask and enumerated types 91
Table 18 – Definition of the *is-assignable-from* relationship for aggregated types 92
Table 19 – Alternative Type Representations 103
Table 20 – IDL Built-in Annotations Usage 121
Table 21 – IDL primitive type mapping 126
Table 22 – Attributes and Metadata represented using XML attributes 132
Table 23 – Attributes and Metadata represented using XML elements 133
Table 24 – Primitive and string type names in the XML Type Representation 133
Table 27 – Formats and interpretation of the TypeIdentifier 145
Table 28 – Alternative Data Representations 164
Table 29 – Serialization of primitive types in version 1 encoding 165

Table 30 – Serialization of enumeration types 166
Table 31 – Serialization of bitmask types 167
Table 32 – Reserved parameter ID values 170
Table 33 – Serialization format to use. 176
Table 34 – State variables and constants in the XCDR stream model 176
Table 35 – Stream operations in the XCDR stream model 177
Table 36 – Type and Byte transformations used in the serialization virtual machine 179
Table 37 – Functions operating on objects and types 180
Table 38 – Symbols and notation used in the serialization virtual machine 183
Table 39 – Kinds of Language Bindings 198
Table 40 – Plain Language Binding for Primitive Types in C 200
Table 41 – Plain Language Binding for Primitive Types in C++ 201
Table 42 – Bit mask integer equivalents 203
Table 43 – Configurable behaviors of the copy function when destination is not NULL 205
Table 44 – Behavior of assignment operator 209
Table 45 – Operations for map<KeyType, ElementType> 213
Table 46 – `DynamicTypeBuilderFactory` properties and operations 225
Table 47 – AnnotationDescriptor properties and operations 232
Table 48 – TypeDescriptor properties and operations 234
Table 49 – DynamicMember behavior 237
Table 50 – DynamicTypeMember properties and operations 238
Table 51 – MemberDescriptor properties and operations 239
Table 52 – DynamicType properties and operations 242
Table 53 – DynamicType::member_by_name behavior 244
Table 54 – DynamicTypeBuilder properties and operations 246
Table 55 – DynamicDataFactory properties and operations 250
Table 56 – DynamicData properties and operations 251
Table 57 – Compatibility matrix for the DataRepresentationQosPolicy 261
Table 58 – RTPS representation identifier values 262
Table 59 – Built-in Endpoints added by the XTYPES specification 270

Table 60 – Mapping of the built-in endpoints added by this specification to the availableBuiltinEndpoints 274

Table 61 – New `TypeSupport` operations 281

Table 62 – New `FooTypeSupport` operations 282

Table 63 – DynamicTypeSupport properties and operations 283

Figures

Figure 1 – Packages 1
Figure 2 – Relationships between Type System, Type Representation, Language Binding, and Data Representation 17
Figure 3 – Example Type Representation, Language Binding, and Data Representation 19
Figure 4 – Namespaces 27
Figure 5 – Type System Model 38
Figure 6 – Primitive Types: Integral Types 39
Figure 7 – Primitive Types: Floating Point Types 39
Figure 8 – Primitive Types: Booleans, Bytes, and Characters 40
Figure 9 - String Types 44
Figure 10 – Fixed Point Types 46
Figure 10 – Constructed Types 47
Figure 11 – Enumerated Types 48
Figure 12 – Enumeration Types 49
Figure 13 – Bitmask Types 50
Figure 14 – Alias Types 53
Figure 15 – Collection Types 57
Figure 16 – Aggregated Types 60
Figure 17 – Structure Types 61
Figure 18 – Union Types 62
Figure 19 – Annotation Types 78
Figure 20 – Type Representation 102
Figure 21 – Directed graph, Strongly Connected Components, and Kernel DAG 153
Figure 22 – Dependency graph derived from a set of type definitions 154
Figure 23 – Data Representation—conceptual model 163
Figure 24 – Usage of PID_EXTENDED within the CDR Buffer 172
Figure 25 – Language Bindings—conceptual model 198

Figure 26 – Dynamic Data and Dynamic Type 223
Figure 27 – Annotation Descriptor 231
Figure 28 – Type Descriptor 234
Figure 29 – Dynamic Type Members 237
Figure 30 – Dynamic Type 242
Figure 31 – Dynamic Data and Dynamic Data Factory 251
Figure 32 – Dynamic Type Support 283

Preface

About the Object Management Group

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 https://www.omg.org/.

OMG Specifications

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

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

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- OMG SysML™
- Other Profile specifications

OMG Middleware Specifications

- CORBA/IIOP
- DDS and the DDS Interoperability Protocol, RTPS
- IDL/Language Mappings

- Specialized CORBA specifications
- CORBA Component Model (CCM)

Platform Specific Model and Interface Specifications

- CORBA services
- CORBA facilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications

All of the 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
109 Highland Avenue
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 *https://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

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

```
Courier - 10 pt. Bold:  Programming language elements.
```

Helvetica/Arial - 10 pt: Exceptions

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

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to https://issues.omg.org/issues/create-new-issue.

1. Scope

The Specification addresses four related concerns summarized in Figure 1 below.

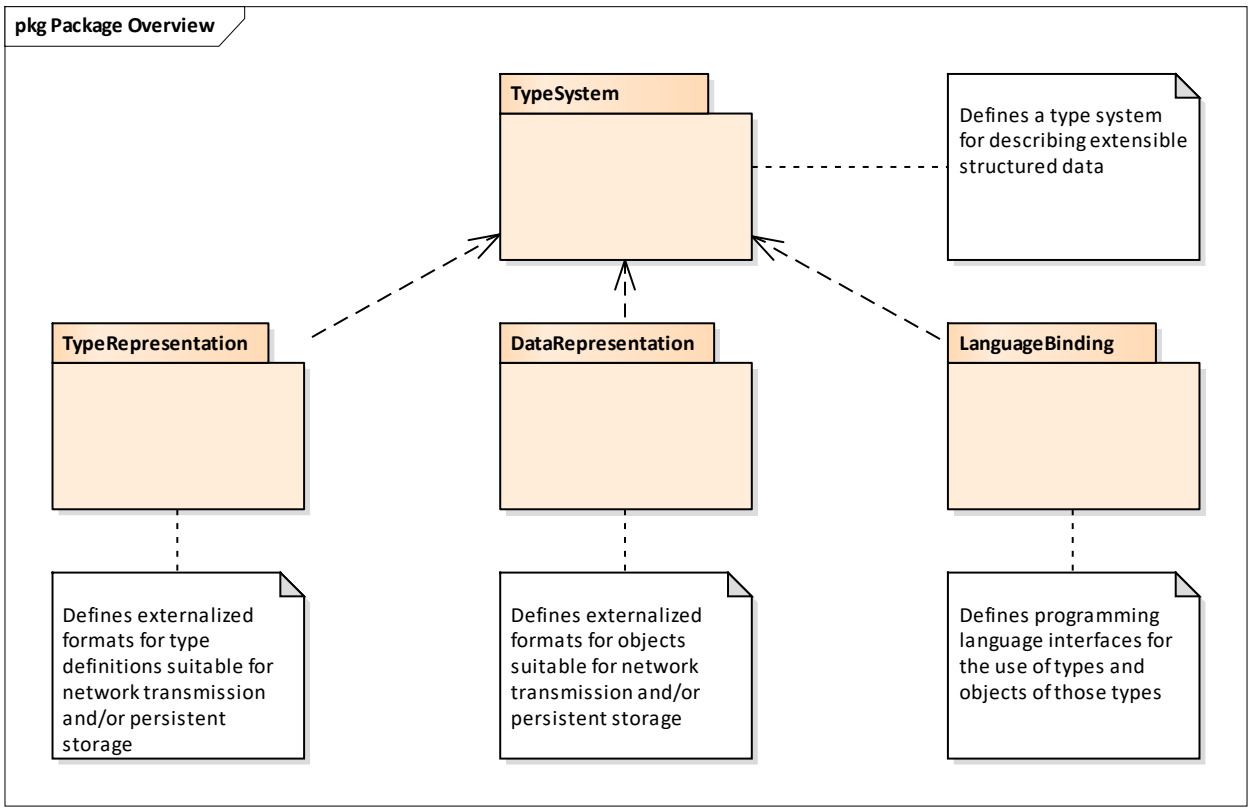

Figure 1 – Packages

> *The specification addresses four related concerns: the type system, the representation of types, the representation of data, and the language bindings used to access types and data. Each of these concerns is modeled as a collection of classes belonging to a corresponding package.*

This specification provides the following additional facilities to DDS [DDS] implementations and users:

Type System. The specification defines a model of the data types that can be used for DDS Topics. The type system is formally defined using UML. The Type System is defined in Clause 7.2 and its sub clauses. The structural model of this system is defined in the Type System Model in Clause 7.2.3. The framework under which types can be modified over time is summarized in Clause 7.2.4, “Type Extensibility and Mutability.” The concrete rules under which the concepts from 7.2.3 and 7.2.4 come together to define compatibility in the face of such modifications are defined in Clause 7.2.5, “Type Compatibility.”

Type Representations. The specification defines the ways in which types described by the Type System may be externalized such that they can be stored in a file or communicated over a network. The specification adds additional Type Representations beyond the one (IDL [IDL])

already implied by the DDS specification. Several Type Representations are specified in the sub clauses of Clause 7.3. These include IDL (7.3.1), XML (7.3.2), XML Schema (XSD) (**Error! Reference source not found.**), and TypeObject (7.3.3).

Data Representation. The specification defines multiple ways in which objects of the types defined by the Type System may be externalized such that they can be stored in a file or communicated over a network. (This is also commonly referred as "data serialization" or "data marshaling.") The specification extends and generalizes the mechanisms already defined by the DDS Interoperability specification [RTPS]. The specification includes Data Representations that support data type evolution, that is, allow a data type to change in certain well-defined ways without breaking communication. Two Data Representations are specified in the sub clauses of Clause 7.4. These are Extended CDR (7.4.1, 7.4.2, and 7.4.3) and XML (7.4.4).

Language Binding. The specification defines multiple ways in which applications can access the state of objects defined by the Type System. The specification extends and generalizes the mechanism currently implied by the DDS specification ("Plain Language Binding") and adds a Dynamic Language Binding that allows application to access data without compile-time knowledge of its type. The specification also defines an API to define and manipulate data types programmatically. Two Language Bindings are specified in the sub clauses of Clause 7.5. These are the Plain Language Binding and the Dynamic Language Binding.

2. Conformance Criteria

This specification recognizes three areas of conformance:

- programming interfaces—that is, related to the DDS API
- network interoperability—that is, related to the RTPS protocol.
- additional type and data representations—that is, external representations used for tooling, code generation, configuration, and data interchange.

In this specification, a conformance profile identifies a specific, independently claimable set of requirements. A conformance level identifies a predefined combination of conformance profiles that together represent a broader portability and/or interoperability capability.

Conformance is organized in terms of profiles:

- Static Programming Interface
- Dynamic Programming Interface
- Minimal Network Interoperability
- Legacy Network Interoperability
- Safe Network Interoperability
- XML Type Representation
- XML Data Representation

These profiles are used to define three conformance levels:

1. ***XTYPES Basic* conformance** requires conformance to the following profiles:
    - Static Programming Interface
    - **Minimal** Network Interoperability
2. ***XTYPES Evolvable* conformance** extends *XTYPES Basic* conformance. It requires conformance to the following additional profiles:
    - **Legacy** Network Interoperability
    - **Safe** Network Interoperability
3. ***XTYPES Dynamic* conformance** requires conformance to all profiles. That is, *XTYPES Evolvable* conformance and the following additional profiles:
    - Dynamic Programming Interface
    - XML Type Representation
    - XML Data Representation

2.1 Programming Interface Profiles

The Programming Interface Profiles define the requirements for conformance at the DDS API level. There are two Programming Interface conformance profiles. An implementation may claim conformance to the **Static Programming profile** or to the **Dynamic Programming profile**. The Dynamic profile extends the Static profile.

This specification extends the *Data Distribution Service for Real-Time Systems* specification [DDS] with an additional optional DDS conformance profile: the "DDS Extensible and Dynamic Types Profile."

Conformance to the "DDS Extensible and Dynamic Types Profile" requires conformance to the DDS specification Minimum Profile and the XTYPES Programming Interface Profile.

2.1.1 Static Programming Interface Profile

Conformance to the Static Programming Interface profile requires conformance to the following clauses of this specification:

- "Type System" (Clause 7.2) excluding "Type Compatibility" (Clause 7.2.5)
- "IDL Type Representation" (Clause 7.3.1)
- "Language Binding" (Clause 7.5)
- "Use of the Type System by DDS" (Clause 7.6) excluding "Use of Dynamic Data and Dynamic Type" (Clause 7.6.6)
- Annex D: "DDS Built-in Topic Data Types"
- Annex E: "Built-in Types"

2.1.2 Dynamic Programming Interface Profile

This profile adds dynamic APIs allowing applications to use data types that are not known at the time the application was compiled.

Conformance with the Dynamic Programming Interface profile requires conformance with the Static Programming Interface profile and the following clauses:

- "Use of Dynamic Data and Dynamic Type" (Clause 7.6.6)
- Annex C: "Dynamic Language Binding"

2.2 Network Interoperability Profiles

There are two Network Interoperability conformance profiles. An implementation may claim conformance to the Minimal **Network profile** or to the **Safe Network profile**. The Safe profile extends the Minimal profile.

Conformance with respect to any of the network interoperability profiles requires conformance to the *Real-Time Publish-Subscribe Wire Protocol* specification [RTPS].

2.2.1 Minimal Network Interoperability Profile

Conformance with the Minimal Network Interoperability profile requires conformance with the following clauses of this specification:

- "Representing Types with TypeIdentifier and TypeObject" (Clause 7.3.3)
- From "Use of the Type System by DDS" (Clause 7.6)
    - "Topic Model" (Clause 7.6.1)
    - "Discovery and Endpoint Matching" (Clause 7.6.3) excluding "Built-in TypeLookup service" (Clause 7.6.3.3)
        - Clause 7.6.3.1.1 "`DataRepresentationQosPolicy`: Conceptual Model", with support limited to version 2 encoding
    - "Interoperability of Keyed Topics" (Clause 7.6.8)
- "Extended CDR Representation (encoding version 2)" (Clause 7.4.2)
- "Extended CDR encoding virtual machine" (Clause 7.4.3)
- Annex B: Representing Types with TypeObject
- Annex D: DDS Built-in Topic Data Types

2.2.2 Legacy Network Interoperability Profile

This profile adds interoperability with implementations that use XCDR version 1 encoding.

Conformance with the Legacy Network Interoperability Profile requires conformance with the Minimal Network Interoperability profile and conformance with the following clauses:

- "Extended CDR Representation (encoding version 1)" (Clause 7.4.1)
- "DataRepresentationQosPolicy: Conceptual Model" (Clause 7.6.3.1.1)

2.2.3 Safe Network Interoperability Profile

This profile adds type safety in the presence of type evolution. It enables checking type compatibility between published and subscribed types as a precondition for matching the endpoints.

Conformance with the Safe Network Interoperability Profile requires conformance with the Minimal Network Interoperability profile and the following clauses:

- “Type Compatibility” (Clause 7.2.5)
- “Built-in TypeLookup service” (Clause 7.6.3.3)

2.3 Optional XML Type Representation Profile

This profile adds support for the XML Type Representation format.

Conformance to this profile requires conformance to the following clauses of this specification:

- “XML Type Representation” (Clause 7.3.2)
- The XML schemas referenced in Annex A: XML Type Representation Schema

2.4 XML Data Representation Profile

This profile adds support for the XML Data Representation format.

Conformance to this profile requires conformance to the following clauses of this specification:

- “XML Data Representation” (Clause 7.4.4)

3. Normative References

The following normative documents contain provisions that, through reference in this text, constitute provisions of this specification.

[**DDS**] *Data Distribution Service for Real-Time Systems Specification*, Version 1.4 (https://www.omg.org/spec/DDS/)

- [**RTPS**] *Real-Time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification*, Version 2.5 (https://www.omg.org/spec/DDSI-RTPS/)
- [**IDL**] *Interface Definition Language*, Version 4.2 (https://www.omg.org/spec/IDL/)
- [**CDR**] *Common Object Request Broker Architecture (CORBA) Specification*, Version 3.1, Part 2 (OMG document formal/2008-01-07), clause 9.3: “CDR Transfer Syntax”
- [**C-LANG**] *Programming languages -- C* (ISO/IEC document 9899:1990)
- [**C++-LANG**] *Programming languages -- C++* (ISO/IEC document 14882:2003)
- [**JAVA-LANG**] *The Java Language Specification, Second Edition* (by Sun Microsystems, http://java.sun.com/docs/books/jls/)
- **[C-MAP]** *C Language Mapping Specification*, Version 1.0 (OMG document formal/1999-07-35)
- [**C++-MAP**] *C++ Language Mapping Specification*, Version 1.2 (OMG document formal/2008-01-09)
- [**JAVA-MAP**] *IDL to Java Language Mapping*, Version 1.3 (OMG document formal/2008-01-11)
- **[DDS-PSM-CXX]** *ISO/IEC C++ 2003 Language DDS PSM™,* Version 1.0 (OMG document formal/2013-11-01)
- **[DDS-XML]** *DDS Consolidated XML Syntax*, Version 1.1, (https://www.omg.org/spec/DDS-XML/)
- [**IDL-XSD**] *CORBA to WSDL/SOAP Interworking Specification*, Version 1.2.1 (OMG document formal/2008-08-03)
- [**LATIN**] *Information technology -- 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1* (ISO/IEC document 8859-1:1998)
- [**UCS**] *Information technology -- Universal Multiple-Octet Coded Character Set (UCS)* (ISO/IEC document 10646:2003)
- **[FNMATCH]** *POSIX fnmatch function* (IEEE 1003.2-1992 clause B.6)
- **[ISO-8601:2004]** *ISO 8601:2004 1988 (E), "Data elements and interchange formats - Information interchange - Representation of dates and times".*
- **[IETF RFC 3339]** *IETF RFC 3339, "Date and Time on the Internet: Timestamps".* https://tools.ietf.org/html/rfc3339.

- **[SPDX-LicenseList]** *The System Package Data Exchange™ (SPDX®) License List.* https://spdx.org/licenses/

- **[UNICODE]** *The Unicode Standard, Version 9.0.0*. (Mountain View, CA: The Unicode Consortium, 2016. ISBN 978-1-936213-13-9). http://www.unicode.org/versions/Unicode9.0.0/.

- **[IEEE-754]** IEEE Standard for Binary Floating-Point Arithmetic, 754-2008 - IEEE Standard for Floating-Point Arithmetic

4. Terms and Definitions

Data Centric Publish-Subscribe (DCPS) – The mandatory portion of the DDS specification used to provide the functionality required for an application to publish and subscribe to the values of data objects.

Data Distribution Service (DDS) – An OMG distributed data communications specification that allows Quality of Service policies to be specified for data timeliness and reliability. It is independent of implementation languages.

5. Symbols

No additional symbols are used in this specification.

6. Additional Information

6.1 Acknowledgments

The following companies submitted and/or supported parts of this specification:

- Real-Time Innovations
- PrismTech Corp
- THALES
- Twin Oaks Computing, Inc.
- Object Computing, Inc.

The following individuals and companies submitted content that was incorporated into this specification:

Submitting contributors:

- (lead) Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations. gerardo.pardo AT rti.com
- Rick Warren, Real-Time Innovations. rick.warren AT rti.com
- Angelo Corsaro, Ph.D., PrismTech. angelo.corsaro AT prismtech.com
- Robert Poth, PrismTech. robert.poth AT prismtech.com
- Virginie Watine, THALES, virginie.watine AT thalesgroup.com

Supporting contributors:

- Clark Tucker, Twin Oaks Computing, Inc. ctucker AT twinoakscomputing.com

Finalization Task Force members and participants:

- (chair) Rick Warren, Real-Time Innovations. rick.warren AT rti.com
- Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations. gerardo.pardo AT rti.com
- Virginie Watine, THALES, virginie.watine AT thalesgroup.com
- Charles Fudge, NSWC Dalghren, charles.fudge AT navy.mil
- Angelo Corsaro, Ph.D., PrismTech. angelo.corsaro AT prismtech.com
- Ken Rode, Kongsberg Gallium, ken.rode AT kongsberggallium.com
- Nikolay Malitsky, Brookhaven National Laboratory, malitsky AT bnl.gov
- John Thier, GD-AIS. john,their AT gd-ais.com
- Char Wales, MITRE, charwing AT mitre.gov

- Johnny Willemsen, Remedy, johnny AT jwillemsen.nl

Version 1.1 members and participants:

- (chair) Sumant Tambe, Ph.D., Real-Time Innovations. sumant.tambe AT rti.com
- Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations. gerardo.pardo AT rti.com
- Clark Tucker, Twin Oaks Computing, Inc. ctucker AT twinoakscomputing.com
- Virginie Watine, THALES, virginie.watine AT thalesgroup.com
- Angelo Corsaro, Ph.D., PrismTech. angelo.corsaro AT prismtech.com
- Charles Fudge, NSWC Dalghren, charles.fudge AT navy.mil
- Ken Rode, Kongsberg Gallium, ken.rode AT kongsberggallium.com

Version 1.2 members and participants:

- (chair) Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations. gerardo.pardo AT rti.com
- Fernando Garcia-Aranda, Real-Time Innovations. fgarcia AT rti.com
- Erin McManus., Real-Time Innovations. erin AT rti.com
- Charles Fudge, NSWC Dalghren, charles.fudge AT navy.mil
- Virginie Watine, THALES, virginie.watine AT thalesgroup.com
- Hakim Souami, THALES, hakim.souami AT thalesgroup.com
- Angelo Corsaro, Ph.D., PrismTech. angelo.corsaro AT prismtech.com
- Adam Mitz, OCI, mitza AT objectcomputing.com
- Clark Tucker, Twin Oaks Computing, Inc. ctucker AT twinoakscomputing.com
- Ken Rode, Kongsberg Gallium, ken.rode AT kongsberggallium.com

Version 1.3 members and participants:

- (chair) Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations. gerardo.pardo AT rti.com
- Charles Fudge, NSWC Dalghren, charles.fudge AT navy.mil
- Erik Hendriks, ADLink Technology. erik.hendricks AT adlink.com
- Adam Mitz, OCI, mitza AT objectcomputing.com
- Clark Tucker, Twin Oaks Computing, Inc. ctucker AT twinoakscomputing.com

- Ornulf Staff, Ph.D., Kongsberg Defence & Aerospace, ornulf.staff AT kongsberg.com

Version 1.4 members and participants:

- (chair) Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations. gerardo.pardo AT rti.com
- Clark Tucker, Twin Oaks Computing, Inc. ctucker AT twinoakscomputing.com
- Erik Hendriks, ZettaScale Technology. erik.hendricks AT zettascale.com
- Ornulf Staff, Ph.D., Kongsberg Defence & Aerospace, ornulf.staff AT kongsberg.com
- Chuck Abbott, OIS, chuck.abbott AT ois.com
- John Draga, Rocket Software, jdraga AT rocketsoftware.com
- Idar Carlsen, Kongsberg Defence & Aerospace, idar.carlsen AT kongsberg.com

7. Extensible and Dynamic Topic Types for DDS

7.1 Overview

A running DDS [DDS] application that publishes and subscribes data must deal directly or indirectly with data types and data samples of those types and the various representations of those objects. The application and middleware perspectives related to data and data types are shown in Figure 2 below.

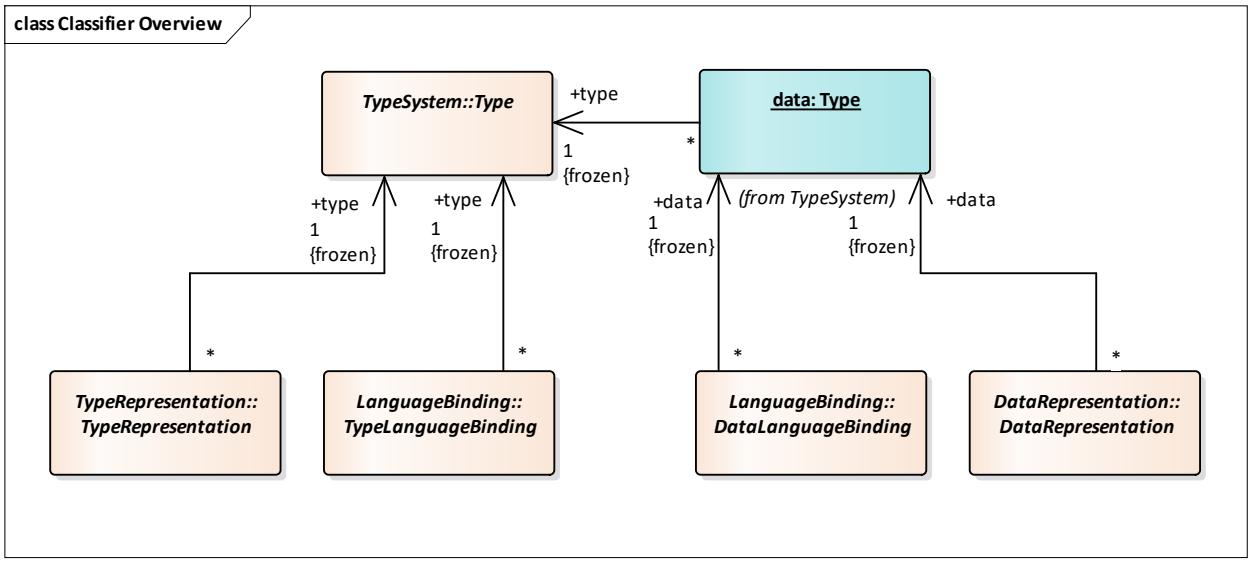

Figure 2 – Relationships between Type System, Type Representation, Language Binding, and Data Representation

> *DDS data objects have an associated data type (in the common programming language sense of the word) that defines a common structure for all objects of the type. From a programming perspective, an object is manipulated using a Language Binding suitable for the programming language in use (e.g., Java). From a network communications and file storage perspective, an object must have a representation (encoding) that is platform neutral and maps into a contiguous set of bytes, whether textual or binary.*
>
> *Similarly, from a programming perspective a data type is manipulated using a Language Binding to the programming language of choice (sometimes known as a reflection API) and must have a representation (encoding) that is platform neutral and maps into a contiguous set of bytes (e.g., XSD or IDL).*

The following example is based on a hypothetical "Alarm" data use case which can be used to explain Figure 2 above.

An application concerned with alarms might use a type called "`AlarmType`" to indicate the nature of the alarm, point of origin, time when it occurred, severity, etc. Applications publishing and subscribing to `AlarmType` must therefore understand to some extent the logical or semantic contents of that type. This is what is represented by the `TypeSystem::Type` class in Figure 2 above.

If this type is to be communicated in a design document or electronically to a tool, it must be represented in some "external" format suitable for storing in a file or on a network packet. This aspect is represented by the `TypeRepresentation::TypeRepresentation` class in Figure 2 above. A realization of the `TypeRepresentation` class may use XML, XSD, or IDL to represent the type.

An application wishing to understand the structure of the Type, or the middleware attempting to check type-compatibility between writers and readers, must use some programming language construct to examine the type. This is represented by the `LanguageBinding::TypeLanguageBinding` class. As an example of this concept, the class `java.lang.Class` plays this role within the Java platform.

An application publishing Alarms or receiving Alarms must use some programming language construct to set the value of the alarm or access those values when it receives the data. This programming language construct may be a plain language object (such as the one generated from an IDL description of the type) or a dynamic container that allows setting and getting named fields, or some other programming language object. This is represented by the `LanguageBinding::DataLanguageBinding` class.

An application wishing to store Alarms on a file or the middleware wishing to send Alarms on a network packet or create Alarm objects from data received on the network must use some mechanism to "serialize" the Alarm into bytes in a platform-neutral fashion. This is represented by the `DataRepresentation::DataRepresentation` class. An example of this would be to use the CDR Representation derived from the IDL Type Representation.

The classes in Figure 2 above represent each of the independent concerns that both application and middleware need to address. The non-normative Figure 3 below indicates their relationships to one another in a less formal way.

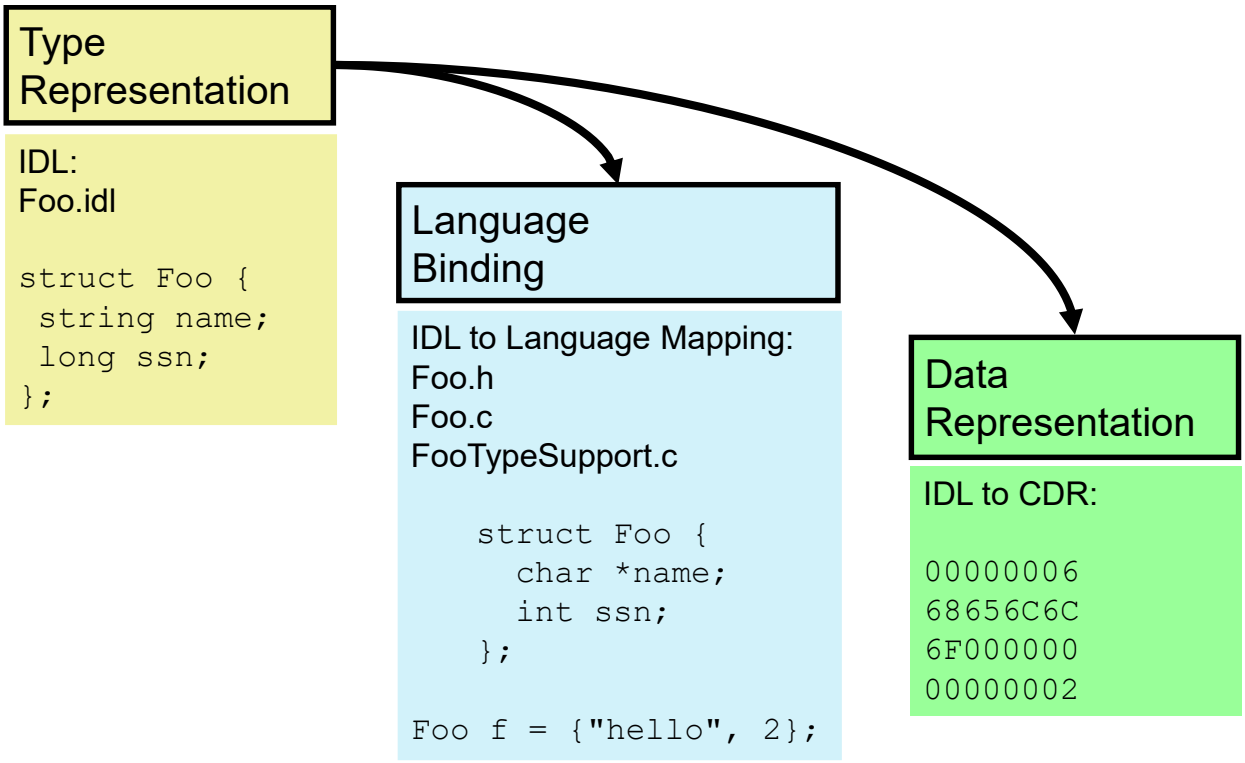

Figure 3 – Example Type Representation, Language Binding, and Data Representation

Type Representation is concerned with expressing the type in a manner suitable for human input and output, file storage, or network communications. IDL is an example of a standard type representation. Language Binding is concerned with the programming language constructs used to interact with data of a type or to introspect the type. Plain language objects as obtained from the IDL language mappings of the IDL representation of the type are one possible Language Binding. Data Representation is concerned with expressing the data in a way that can be stored in a file or communicated over a network or manipulated by a human. The Common Data Representation is a Data Representation optimized for network communications; XML is another representation more suitable for human manipulation.

7.2 Type System

The Type System defines the data types that can be used for DDS Topics and therefore the type of the data that can be published and subscribed via DDS.

7.2.1 Overview (Informative)

The *Data Distribution (DDS)* is the Object Management Group (OMG) standard for data-centric publish-subscribe communication used in real-time and mission-critical systems. DDS provides applications with a data-centric shared-state model in which information is represented as typed Topics and exchanged among DataWriters and DataReaders with well-defined Quality-of-Service (QoS) controls for timeliness, reliability, and resource usage.

Originally, early versions of DDS assumed that application data types were defined and compiled ahead of time using the OMG IDL language mapping. This model offered high performance and static type safety but limited flexibility in systems that evolve or integrate components developed independently.

To address those limitations, OMG introduced the Extensible and Dynamic Topic Types (XTypes) specification, now an integral part of DDS. DDS implementations conformant with XTypes can operate with:

- **Statically defined types**, generated from IDL and used through language-binding classes, and
- **Dynamically defined or discovered types**, created, exchanged, and interpreted at run time without prior compile-time knowledge.

The XTypes model formalizes:

- A Type System that defines the kinds of types DDS can represent and how those types may evolve over time.
- Type Representations—IDL, XML, and TypeObject—that describe how types are defined, stored, and communicated.
- Data Representations, standardized as Extended CDR (XCDR) version 1 and version 2, which serialize data for interoperable transmission. XCDR v1 corresponds to the original CDR encoding extended for schema evolution; XCDR v2 generalizes this with better support for compatible type evolution and more efficient type discovery.
- A Dynamic Language Binding API that allows inspection and manipulation of types and data at run time.

Through XTypes, DDS now supports type evolution and safe interoperability between different versions of a type. Writers and Readers can communicate even when their type definitions differ, subject to assignability rules and TypeConsistencyEnforcement QoS settings.

DDS together with XTypes therefore enables:

- Loose coupling between components that evolve independently,
- Dynamic discovery of types and topics at run time, and
- Efficient binary interoperability through standardized XCDR encodings.

Table 1 summarizes the principal features and mechanisms defined by this specification to address the main DDS type-related concerns.

Table 1 – Major type-related concerns in DDS and mechanisms defined by this specification

Concern	*Features and mechanisms introduced by the Extensible Topics specification*
Type System	Defined in UML, independent of any programming language. Supports:

	• Primitive, string, enumerated, collection, and aggregated types (structures, unions, arrays, sequences, maps) • Type inheritance (single inheritance for aggregated types) • Keys, optional members, and sparse types (types whose instances may omit values for certain fields) • Specification of units and value ranges for numeric members • Inclusion of documentation and license metadata as part of type definitions • Custom (user-defined) annotations to attach metadata to types and type elements.
Type Representation	Multiple standardized representations: • **IDL** – Intuitive, developer-friendly format compatible with existing IDL-based datamodels. • **XML** – Structured text format suitable for configuration and tooling. • **`TypeObject/TypeIdentifier`** – Compact binary representations optimized for network propagation, discovery, and type assignability evaluation.
Data Representation	Multiple standardized encodings: Standardized Extended CDR (XCDR) encodings: • XCDR version 1 – Derived from classic CDR; adds bounded sequences and optional extensibility for evolved types. • XCDR version 2 – Enhanced encoding with more flexible type evolution rules, better suited for modern datamodels. Both encodings ensure binary interoperability and support type evolution across revisions. Human-readable XML encoding may also be used to ease integration with tools and web technologies..

Language Binding	Two complementary approaches: • **Plain Language Binding** – Provides maximum performance, compile-time type safety and smooth integration with native language data types; rerquires code generation. • **Dynamic Language Binding** – Provides reflection and manipulation of types and data at run time without prior compile-time knowledge or code generation; ideal for tooling and bridging.

The non-normative examples in Clauses 7.2.1.1 through 7.2.1.4 below illustrate several a few selected concepts of the DDS Type System—type evolution, inheritance, sparse/optional members, and annotations. Formal definitions are found in Clause 7.2.3.

7.2.1.1 Type Evolution Example

Assume a DDS-based distributed application has been developed that uses the Topic "Vehicle Location" of type `VehicleLocationType`. The type `VehicleLocationType` itself was defined using the following IDL:

```
// Initial Version
struct VehicleLocationType {
   float latitude;
   float longitude;
};
```

As the system evolves it is deemed useful to add additional information to the `VehicleLocationType` such as the estimated error latitude and longitude errors as well as the direction and speed resulting in:

```
// New version
struct VehicleLocationType {
   float latitude;
   float longitude;
   float latitude_error_estimate;    // added field
   float longitude_error_estimate;   // added field
   float direction;                  // added field
   float speed;                      // added field
};
```

This additional information can be used by the components that understand it to implement more elaborate algorithms that estimate the position of the vehicle between updates. However, not all components that publish or subscribe data of this type will be upgraded to this new definition of `VehicleLocationType` (or if they are all upgraded, they will not be upgraded at the same time)

so the system needs to function even if different components use different versions of `VehicleLocationType`.

The Type System supports type evolution so that it is possible to "evolve the type" as described above and retain interoperability between components that use different versions of the type such that:

- A publisher writing the "initial version" of `VehicleLocationType` will be able to communicate with a subscriber expecting the "new version" of the `VehicleLocationType`. In practice what this means is that the subscriber expecting the "new version" of the `VehicleLocationType` will, depending on the details of how the type was defined, either be supplied some default values for the added fields or else be told that those fields were not present.
- A publisher writing the "new version" of `VehicleLocationType` will be able to communicate with a subscriber reading the "initial version" of the `VehicleLocationType`. In practice this means the subscriber expecting the "initial version" of the `VehicleLocationType` will receive data that strips out the added fields.

Evolving a type requires that the designer of the new type explicitly tags the new type as equivalent to, or an extension of, the original type and limits the modifications of the type to the supported set. The addition of new fields is one way in which a type can be evolved. The complete list of allowed transformations is described in Clause7.2.5.

7.2.1.2 Type Inheritance Example

Building upon the same example in Clause 7.2.1.1, assume that the system that was originally intended to only monitor location of land/sea-surface vehicles is now extended to also monitor air vehicles. The location of an air vehicle requires knowing the altitude as well. Therefore the type is extended with this field.

```
// Extended Location

struct VehicleLocation3DType : VehicleLocationType {

   float altitude;

   float vertical_speed;

};
```

VehicleLocation3DType is an extension of `VehicleLocationType`, not an evolution. VehicleLocation3DType represents a new type that extends `VehicleLocationType` in the object-oriented programming sense (IS-A relationship).

The Type System supports type inheritance so that it is possible to "extend the type" as described above and retain interoperability between components that use different versions of the type. So that:

An application subscribing to Topic "Vehicle Position" and expecting to read `VehicleLocationType` CAN receive data from a Publisher that is writing a VehicleLocation3DType. In other words applications can write extended types and read base types.

An application subscribing to Topic “Vehicle Position” and expecting to read VehicleLocation3DType CAN receive data from a Publisher that is writing a `VehicleLocationType`. Applications expecting the derived (extended) type can accept the base type; additional members in the derived type will take no value or a default value, depending on their definitions.

This behavior matches the behavior of the “IS-A” relationship in Object-Oriented Languages,

Intuitively this means that a VehicleLocation3DType is a new type that happens to extend the previous type. It can be substituted in places that expect a `VehiclePosition` but it is not fully equivalent. The substitution only works one way: An application expecting a VehicleLocation3DType cannot accept a `VehiclePosition` in place because it cannot “just” assume some default value for the additional fields. Rather it wants to just read those `VehiclePosition` that corresponds to Air vehicles.

7.2.1.3 Sparse Types Example

Suppose that an application publishes a stream of events. There are many kinds of events that could occur in the system, but they share a good deal of data, they must all be propagated with the same QoS, and the relative order among them must be preserved—it is therefore desirable to publish all kinds of events on a single topic. However, there are fields that only make sense for certain kinds of event. In its local programming language (say, C++ or Java), the application can assign the value null to a pointer to omit a value for these fields. It is desirable to extend this concept to the network and allow the application to omit irrelevant data in order to preserve the correct semantics of the data.

Alternatively, suppose that an application subscribes to data of a type containing many fields, most of which often take a pre-specified “default value” but may, on occasion, deviate from that default. In this situation it would be inefficient to send every field along with every sample. Rather it would be better to just send the fields that take a non-default value and fill the missing fields on the receiving side, or even let the receiving application do that job. This situation occurs, for example, in the DDS Built-in Topic Data. It also occurs in financial applications that use the FIX encoding for the data.

The type system supports sparse types whereby a type can have fields marked “optional” so that a Data Representation may omit those fields. Values for non-optional fields may also be omitted to save network bandwidth, in which case the Service will automatically fill in default values on behalf of the application.

7.2.1.4 Annotation Example

Annotations provide a standardized mechanism for attaching structured metadata to type elements.

The XTYPES type system defines a set of built-in annotations with predefined structure and semantics.

Developers may also define custom annotations.

The structure of an annotation is described by a corresponding **annotation definition** (see 7.2.3.3), while annotations are instantiated through **annotation applications** (see 7.2.2.12).

Annotation definitions have a TypeObject and a TypeIdentifier. They are treated as types by the TypeLookup service (see 7.6.3.3).

The following IDL example illustrates both concepts:

```
@annotation qos_profile {
   string name;   // e.g. <library>::<profile>
};
@qos_profile(name="FutureLibrary::TimeTravel")
struct Delorean {
   Wheel wheels[4];

   @unit("mile/gallon")
   float miles_per_gallon;
};
```

In this example:

- The declaration `@annotation qos_profile { ... }` defines a custom annotation type.
- The use of `@qos_profile(...)` on struct `Delorean` is an annotation application that associates an instance of the `qos_profile` annotation with the `Delorean` structure type.
- The use of `@unit` on member `miles_per_gallon` is an annotation application that associates an instance of the builtin IDL annotation `unit` with parameter value set to `"mile/gallon"` to the member `miles_per_gallon`.

The formal definition of annotation definitions appears in 7.2.3.3 (Annotation Definition) and the rules governing annotation applications appear in 7.2.2.12 (Annotation Application).

7.2.2 Attributes and Metadata

Type Kind

Every type has a *kind* that identifies whether it is a Primitive type (see Table 4), a String type (7.2.3.1.2), an Enumerated Type (7.2.3.2.1), a Collection Type (7.2.3.2.2.1), an Alias Type (7.2.3.2.2), or an Aggregated type (7.2.3.2.4). The type kind also identifies the specific kind within that category, for example (*Int32* versus *Float64* for Primitive, *Array* versus *Sequence* for Collection, *Structure* versus *Union* for Aggregated, etc.).

7.2.2.2 Type Naming

Types in the type system can be classified into three categories:

1. *Primitive Types*. Types with predefined names.
2. *Anonymous Types*. Types that do not have a name.
3. *Named Types.* Types with names provided in the type definition.

7.2.2.2.1 Pre-Named Types

Basic types whose names are predefined by the Type System. These correspond to the Primitive Types (see 7.2.3.1.1). Each has a reserved, globally recognized name (see Table 4).

7.2.2.2.2 Anonymous Types

DDSXTY14-2: Specify more clearly which types have a name and how it is constructed

Basic types constructed inline that do not have a name. These correspond to the String Types (see 7.2.3.1.2) and the Collection Types (see 7.2.3.2.3).

7.2.2.2.3 Named Types

Types whose names are introduced as part of the type-definition process. These are the Constructed Types (7.2.3.2) other than Collections–namely Enumerated Types (7.2.3.2.1), Alias Types (7.2.3.2.2), and Aggregated Types (7.2.3.2.4).

Named types have both an *unqualified* name and a *fully-qualified* name.

7.2.2.2.3.1 Unqualified Name

The string identifier given as part of the type-definition. It shall consist one or more characters matching the following POSIX Extended Regular Expression:

```
^[_A-Za-z][_A-Za-z0-9]*$
```

This means:

- The first character shall be an ASCII alphabetic character (`A-Za-z`) or the underscore (`_`).
- Each subsequent characters shall be an ASCII alphabetic character (`A-Za-z`), digit (`0-9`), or underscore (_).

7.2.2.2.3.2 Fully-qualified name

The string identifier constructed by prepending the names of all the containing namespaces (see 7.2.2.3) to the unqualified name using the "::" as separator.

Example:

```
moduleName1::moduleName2::structName1
```

7.2.2.2.3.3 Scope of Fully Qualified Names Uniqueness

The fully qualified name of a type identifies the type within the namespace hierarchy, but it is not necessarily unique across the entire type system.

Certain type representations (see 7.3) may impose additional constraints, requiring fully qualified names to be unique within a file or across a set of files processed together. When such constraints apply, two distinct types having the same fully qualified name may result in an error.

7.2.2.3 Namespaces and Modules

A **namespace** is a scope used to construct fully qualified type names.

A **module** is a construct used to group related types which are nested inside the module. Modules can contain Named Types as well as other modules. Named types can be belong to at most one module.

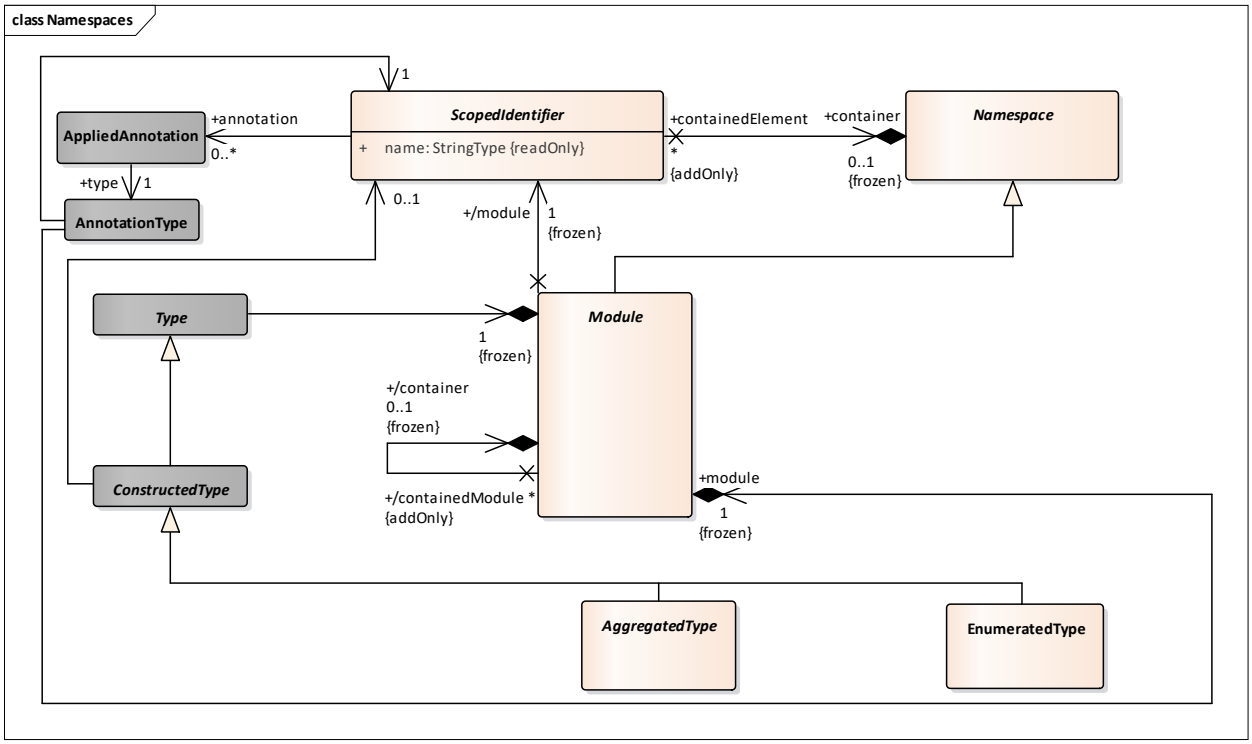

Figure 4 – Namespaces

Namespaces fall into one of two categories:

- *Modules*. Modules have a name which is used a namespace for the Named Types (see 7.2.2.2) within the module.
- Certain kinds of *types* are themselves namespaces with respect to the elements inside of them.

7.2.2.4 Nested Types

Not every type in a user's application will be used to type DDS Topics; some types appear only as the types of members within other types. It is desirable to distinguish these two cases for the sake of efficiency; for example, an IDL compiler need not generate typed `DataWriter`, `DataReader`, and `TypeSupport` classes for types that are not intended to type topics. Types that are not intended to describe topic data are referred to as *nested* types.

7.2.2.5 Documentation

Datamodel authors may need to associate human-readable documentation with types (e.g., aliases, enumerations, structures, unions) as well as type elements (e.g., structure members,

union members, and enumeration literals). Such documentation is intended for generated API docs, reference manuals, or annotated source comments and should be preserved across translations of the Type Representation. Therefore, it is appropriate to manage user-specified documentation within the Type System so that all Type Representation compilers can retain and render it. The `Documentation` class serves this purpose. Each constructed type and each element of an aggregated or enumerated type may refer to one or more instances of this class.

A `Documentation` object defines two properties; each is a string:

- `format` — the format in which the documentation is expressed.
- `text` — the documentation content.

`Documentation` may be attached to Named Types (see 7.2.2.2.3), Aggregated Type Members (see 7.2.3.2.4.4), Enumeration Literals (see 7.2.3.2.1.1), and Bitmask flags (7.2.3.2.1.2).

7.2.2.5.1 Property: Format

This property identifies the representation format of the documentation text. It shall be interpreted in a case-insensitive manner. All Type Representation compilers shall recognize the following format strings; implementations may recognize others in addition:

- "plain" : plain, unformatted text.
- "markdown" : Markdown-formatted text.
- "html" : HTML-formatted text.

The default value is "markdown".

7.2.2.5.2 Property: Text

This property contains the documentation content. The content may be a multi-line string. When generating code or documentation, a Type Representation compiler shall preserve the content and apply any escaping or transformation required by the target language or output format (e.g., escaping quotes in code comments and converting Markdown or HTML to the target doc system if supported).

7.2.2.6 License Information

Datamodel authors may need to associate license information with types so that downstream users can determine the terms under which those types may be used. This specification represents that license information with three distinct classes:

- `LicenseType` — a definition of a license kind (e.g., Apache 2.0), identified by a stable name and a canonical reference link (URL).
- `LicenseUse` — a concrete statement that a collection of artifacts adopts a particular license_type, identified by a unique name and a link (URL) to the license/notice file.
- `License` — an attachment that binds a `LicenseUse` to a type, or to a module.

The `LicenseType` and `LicenseUse` are declarations and shall appear only at the global scope or associated with modules.

A `License` may be attached to modules and Named Types (see 7.2.2.2.3) to indicate the terms under which those artifacts may be used. A module-level `License` applies to all types and nested modules it contains unless overridden by a `License` attached to a nested element.

Toolchains shall preserve `License` associations across Type Representations.

7.2.2.6.1 LicenseType (declaration)

A `LicenseType` declares a kind of license recognized by the Type System. It defines the following string properties:

- `name` — Identifier of the license kind.
- `link` — Canonical URL where the license text is defined.

`LicenseType` declarations may appear at global scope or within modules. They are not associated with specific types.

This specification reserves all the `LicenseType` names that correspond to identifiers in the Software Package Data Exchange (SPDX) License List (see [SPDX-LicenseList]). LicenseTypes with those names are considered predefined with the canonical links URLs specified by the License List (https://spdx.org/licenses/).

Example: The LicenseType with name `"Apache-2.0"` and link `"https://spdx.org/licenses/Apache-2.0.html"` is predefined as it is listed in https://spdx.org/licenses/.

7.2.2.6.2 LicenseUse (declaration)

A `LicenseUse` declares that a specific project, product, or collection of artifacts adopts a particular `LicenseType`. It defines the following string properties:

- `name`: A unique identifier for this usage (e.g. "MyModelLicense:Apache2").
- `license_type_name`: Identifier of the referenced LicenseType; it shall correspond to the name of a defined LicenseType (e.g., "Apache-2.0").
- `link`: URL to the project's license/notice file (e.g., a repository LICENSE.md) indicating that the artifacts are released under that license.

License `LicenseUse` declarations may appear at global scope or within modules. They are not associated with specific types.

7.2.2.6.3 License (attachment)

A `License` associates a `LicenseUse`—and therefore a `LicenseType`—with a module or a Named Type. It defines one string property:

- `license_use_name`: Identifier of the `LicenseUse` that applies; it shall correspond to the name of a defined `LicenseUse` (e.g. " MyModelLicense:Apache2").

A `License` may be attached to modules and Named Types (see 7.2.2.2.3).

- Each element (module or Named Type) shall have at most one License attached.
- A module-level `License` applies to all types and nested modules it contains unless overridden by a `License` attached to a nested element.
- A `License` attached to a Named Type affects only that one type.

7.2.2.6.4 Verbatim Text

System developers frequently require the ability to inject their own text into the code produced by a Type Representation compiler. Such output typically depends on the target programming language, not on the Type Representation. Furthermore, it is desirable to be able to preserve information about such output across translations of the Type Representation. Therefore, it is appropriate to manage user-specified content within the Type System for use by all Type Representations and therefore by Type Representation compilers. The `VerbatimText` class serves this purpose; each constructed type may refer to one or more instances of this class.

A `VerbatimText` object defines three properties; each is a string:

- `language`: The target programming language for which the output text applies.
- `placement`: The location within the generated output at which the output text should be inserted.
- `text`: The literal output text to be copied into the output by the Type Representation compiler.

7.2.2.6.4.1 Property: Language

When a Type Representation compiler generates code for the programming language named (case-insensitively) by this property, it shall copy the string contained in the `text` property into its output.

- The string “c” shall indicate the C programming language [C-LANG].
- The string “c++” shall indicate the C++ programming language [C++-LANG].
- The string “java” shall indicate the Java programming language [JAVA-LANG].
- The string “*” (an asterisk) shall indicate that `text` applies to all programming languages.

7.2.2.6.4.2 Property: Placement

This string identifies where, relative to its other output, the Type Representation compiler shall copy the `text` string. It shall be interpreted in a case-insensitive manner. All Type Representation compilers shall recognize the following placement strings; individual compiler implementations may recognize others in addition.

- `begin-declaration-file`: The `text` string shall be copied at the beginning of the file containing the declaration of the associated type before any type declarations.

For example, a system implementer may use such a `VerbatimText` instance to inject import statements into Java output that are required by literal code inserted by other `VerbatimText` instances.

- `before-declaration`: The `text` string shall be copied immediately before the declaration of the associated type.

  For example, a system implementer may use such a `VerbatimText` instance to inject documentation comments into the output.

- `begin-declaration`: The `text` string shall be copied into the body of the declaration of the associated type before any members or constants.

  For example, a system implementer may use such a `VerbatimText` instance to inject additional declarations or implementation into the output.

- `end-declaration`: The `text` string shall be copied into the body of the declaration of the associated type after all members or constants.

- `after-declaration`: The `text` string shall be copied immediately after the declaration of the associated type.

- `end-declaration-file`: The `text` string shall be copied at the end of the file containing the declaration of the associated type after all type declarations.

7.2.2.6.4.3 Property: Text

The Type Representation compiler shall copy the string contained in this property into its output as described above.

7.2.2.7 Default Value

The type system supports optional specification of the default value associated with Alias types and Aggregated type members.

The default value is used to initialize the type or member in situations where an initialization is required but no other (legal) value is provided. For example:

- If an object of an Aggregated type is constructed with an "empty" constructor that takes no parameters, and therefore does not specify how to initialize its members, then members with associated default values will be initialized accordingly.

- If an Alias type is used as the type of a member or variable, the associated default value is applied when no explicit initialization is provided.

The default value is also used in when the Try Construct behavior (see 7.2.2.8) indicates the object should be constructed by setting a member to its default value.

7.2.2.8 Try Construct behavior

Type evolution can result in a DDS `DataReader` built using type "T1" to be matched with a `DataWriter` built using a different but compatible version of the type "T2". When the

`DataReader` receives an object O2 sent by the `DataWriter` it needs to construct some object of type T1 to hold the data in O2. The expectation is that the constructed object "O1" of type T1 will faithfully capture all the information from O2 that is relevant to the application that was expecting to read objects of type T1.

There are situations where no "obviously reasonable" object of type T1 can be constructed to hold the value of a specific object "O2" of type "T2". A type system could declare types T1 and T2 where this situation may occur to be "incompatible" thus ensuring the situation is never encountered when a `DataWriter` sends data to a matching `DataReader`. However doing so would be too restrictive for the kinds of distributed systems where DDS is deployed.

For example, a system may be deployed with `DataReader` entities reading an Aggregated type (e.g. a structure) called "STRUCT1024" with a member of type string with a maximum length of 1024 characters, see Table 2. Once the system is deployed new applications are added and the deployment extends to resource-constrained environments where the 1024 character strings can be problematic. Moreover as it turned out the value of 1024 was overly generous and in the deployed system the strings never exceed 80 characters. In this situation it becomes desirable to re-define the type as "STRUCT128". STRUCT128 differs from STRUCT1024 in that the string member has maximum length 128, see Table 2. With these definitions there exist objects of type STRUCT1024 that cannot construct any object of type STRUCT128, namely those objects a string member of length greater than 128 characters. This is true even if the application never uses these objects. If the existence of such objects would prevent STRUCT128 from being compatible with STRUCT1024 we would not be able to adjust the type without modifying the already deployed systems, which may not be feasible.

Similar situations can occur for Collection types. For example a type "SEQ1024" that is defined as a sequence whose elements have type string with maximum length 1024 and an evolution of that type "SEQ128" that differs from SEQ1024 in that the element type is string with maximum length 128, see Table 2. Similar to the structure examples there exist objects of type SEQ1024 that cannot construct any object of type SEQ128 and yet in many cases we do not want to consider these types as incompatible.

Table 2 – TryConstruct examples

Example Type IDL definition	*Explanation of the Type*

`struct STRUCT1024 {`
`  string<1024>  member;`
`};`
	Structure Aggregated type with a member of type string with maximum length **1024** characters.

`struct STRUCT128 {`
`  string<128>  member;`
`};`
	Structure Aggregated type with a member of type string with maximum length **128** characters.

`typedef` `sequence< string<1024> > SEQ1024;`	Sequence Collection type with element of type string with maximum length **1024** characters.
`typedef` `sequence< string<128> > SEQ128;`	Sequence Collection type with element of type string with maximum length **128** characters.

To avoid the situation described above the type compatibility relationship defined by this type system (see Clause7.2.5) does not require that all objects of a type "T2" can faithfully construct some object of some other type "T1", as a pre-requisite for compatibility. The type system only requires that a reasonable subset of T2 object can construct some object of type T1 and that the situations where this is not possible are detected and gracefully handled. The rules for this are formally defined in Clause 7.2.5.

Therefore, even when two types T1 and T2 are compatible it may be possible to encounter an object sent by a `DataWriter` of type T2 that cannot be used to construct any object of the T1 type expected by the `DataReader` without losing some potentially critical information. For example, depending on the application truncating a 20-character string sent by the `DataWriter` into a 10-character string that may be the maximum allowed by the `DataReader` could result in misinterpretation and application malfunction. The same could be said for trimming a received sequence to a shorter length.

If no "reasonable" T1 object can be constructed from a given object O2 of type T2, we say that "**O2 cannot construct any object of type T1**".

Object construction for collection and aggregated types is done recursively. To construct the collection/aggregated object it is necessary to construct all nested elements/members. For this reason, failure to construct a nested element/member can prevent the construction of the collection/aggregated type.

There are situations when it is not desirable to fail the construction of a collection or aggregated object of type T1 just because some nested element/member cannot be constructed. The failure to construct the element/member would cause all other nested elements/members to be "lost" and not just the problematic one. In some situations, it may be more desirable to trim the problematic member or set it to some well-known default value. To support these scenarios Collection and Aggregated types may explicitly declare the `TryConstruct` behavior of each of their elements or members.

- Array and Sequence collection types may explicitly declare that their element has one of three kinds of `TryConstruct` behavior, see Table 3.
- Map collection types may explicitly declare that their "key" and or "value" element has one of three kinds of `TryConstruct` behavior, see Table 3.
- Structure and Union types may explicitly declare member has one of three kinds of `TryConstruct` behavior, see Table 3. In the case of Unions this extends to the discriminator member.

The `TryConstruct` behavior kinds are described in Table 3 below. The default behavior unless otherwise specified using the `TryConstruct` annotation is `DISCARD`.

Table 3 – TryConstruct behavior kinds

TryConstruct kind	*Description*
`DISCARD`	**Failure to construct an element or member propagates to the collection or aggregated type that contains it.** If an element or member cannot be constructed, then the collection or aggregated object that contains the element or member cannot be constructed either.
`USE_DEFAULT`	**Failure to construct an element or member is contained—element or member is set to its default value.** If an element or member cannot be constructed, the element/member shall be set to its default value (according to its type as described in Table 10) and does not cause the collection/aggregated object to fail its construction.
`TRIM`	**Failure to construct an element or member is contained—element or member is trimmed.** This option only applies to elements or members of type string, wide string, sequence, or map. The behavior when applied to other element/member types is unspecified and may be treated as an error. The option affects the situation where failure to construct is due to the length of the collection sent exceeding what can be accommodated on the receiving member collection type. In this situation the element or member is constructed trimming the received object to the length that can be accommodated by the receiving member type. The order of the characters in the string or elements in the sequence or map is preserved.

7.2.2.9 Units

The type system supports optional specification of the units associated with Alias types and Aggregated Type members.

Units are intended to capture the physical scale of reference used to measure the values of that type. For example, they may be used to indicate that a data type is intended to hold a length value measured in `"meter"` or an energy value in `"kW h"`.

When specified, the type system uses the units associated with Alias types and Aggregated type members when evaluating type assignability (see 7.2.5.5.3 and 7.2.5.5.9).

Units are specified as a string formatted as a semicolon-separated list of elements. Each element is interpreted as an alternative *unit name*. Whitespace is ignored, and unit names are case-sensitive.

For example:

- `"degrees"` defines a single unit name `"degrees"`
- `"meter; m"` defines two unit names `"meter"` and `"m"`
- `"watt; W; J/s"` defines three unit names: `"watt"`, `"W"`, and `"J/s"`.

Unit names shall use printable ASCII characters in the range 0x22-0x126, except for 0x3B (';'), which is reserved as a separator.

Two unit specifications are considered compatible if they have at least one *unit name* in common.

For example:

- `"meter ; m"` is compatible with `"m"`, `"meter"`, and `" meter;m "`
- `"meter; m"` is not compatible with `"M"` or `"Meter"`
- `"km/h"` is compatible with `"km / h"` and `"km/ h"`
- `"kw h"` is compatible with `"kwh"`
- `"kw h"` is not compatible with `"h kw"`

The reserved unit name `"__ANY__"` is treated as a wildcard: it matches every unit name, as well as not having a unit.

For example:

- `"__ANY__"` is compatible with `"m"` and `"meter; m"`
- `"__ANY__"` is compatible with not having units specified

If only one side specifies units, the compatibility behavior is controlled by the `TypeConsistencyEnforcementQosPolicy`, specifically the `ignore_unspecified_units` flag (see 7.6.3.4.1).

The type system does not attempt to interpret the format or semantics of unit strings (e.g., it does not parse symbols as mathematical operators, perform conversions, or interpret structured notations), beyond applying the rules for type assignability checks.

7.2.2.10 Range Limits

Numeric primitive types can represent only a limited range of values. These limits are typically determined by the intended language binding and data representation. For example, an `Int32` type can only hold integer values that can be represented using 32 bits, that is, -2^{31} to $2^{31} - 1$.

In some cases, it may be desirable to further restrict the range of permissible values to better reflect the intended use of the type. To support this, the type system allows the optional specification of range limits for Alias types and for members of Aggregated types (see 7.2.3.2.2.3 and 7.2.3.2.4.4.12).

Range limits are specified as two numeric values the minimum (inclusive) and the maximum (inclusive). It is also possible to specify only one of the two limits.

It shall be an error to specify a limit that falls outside the representable range of the underlying type (for example, a maximum greater than $2^{31} - 1$ for an `Int32` type).

If a range limit is not specified, it shall be treated as equivalent to specifying the limit that corresponds to the representable range of the underlying type.

When specified, the type system uses the range limits associated with Alias types and members of Aggregated types when evaluating type assignability (see 7.2.5.5.3 and 7.2.5.5.9).

Two range specifications are considered compatible only if their associated limits have a non-empty intersection. For example:

- Limits: `min=-20, max=30` defines the range `-20 <= value <= 30`
- Limits: `max=30` defines the range `value <= 30`
- Limits: `min=-20` defines the range `-20 <= value`
- Limits: `min=-20, max=30` is compatible with Limits: `min=-100, max=100`
- Limits: `min=-20, max=30` is compatible with Limits: `min=5, max=50`
- Limits: `min=-20, max=30` is compatible with Limits: `min=5`
- Limits: `max=30` is compatible with Limits: `min=5`
- Limits: `min=-20, max=30` is **not** compatible with Limits: `min=50, max=100`

Limits: `max=30` is **not** compatible with Limits: `min=50`

7.2.2.11 External Data

In some cases, it is necessary and/or desirable to provide information to a language binding that a certain member's data should be stored, not inline within its containing type, but external to it (e.g., using a pointer).

- For example, the data may be very large, such that it is impractical to copy it into a sample object before sending it on the network. Instead, it is desirable to manage the storage outside of the middleware and assign a reference in the sample object to this external storage.

- For example, the type of the member may be the type of a containing type (directly or indirectly). This will be the case when defining linked lists or any of a number of more complex data structures.

Type Representations shall therefore allow the following type relationships in the case of external members, which would typically cause errors in the case of non-external members:

- An external member of an aggregated type shall be permitted to refer to a type whose definition is incomplete (*i.e.,* is identified only by a forward declaration) at the time of the member's declaration.
- An external member of an aggregated type shall be permitted to refer to the member's containing type.

Each member of an aggregated type—with the exception of the discriminator of a union type—may be optionally marked as *external*. Likewise, the elements of a collection type may be optionally marked as external.

Note that this attribute does *not* provide a means for modeling object graphs.

7.2.2.12 Annotation Application

An annotation application is the attachment of an instance of an annotation (see 7.2.3.3) to a model element. Annotation applications constitute structured metadata that qualify or constrain the semantics of types, members, and other model elements.

Each annotation application denotes an instance of an annotation definition, and supplies values its parameters. Optional parameters may be omitted, in which case the default values defined by the annotation type apply.

Annotations may be applied to modules (see 7.2.2.3), named type declarations (see 7.2.2.2.3), the related type of an Alias type (see 7.2.3.2.2), and elements within Enumerated (see 7.2.3.2.1) and Aggregated (see 7.2.3.2.4) types.

More than one annotation may be applied to the same element. Multiple instances of the same annotation type may be applied if the annotation definition explicitly permits it.

As a general rule, annotations applied at a module level implicitly apply to all types defined within the module unless overridden by a directly applied annotation in a nested element.

7.2.3 Type Categories

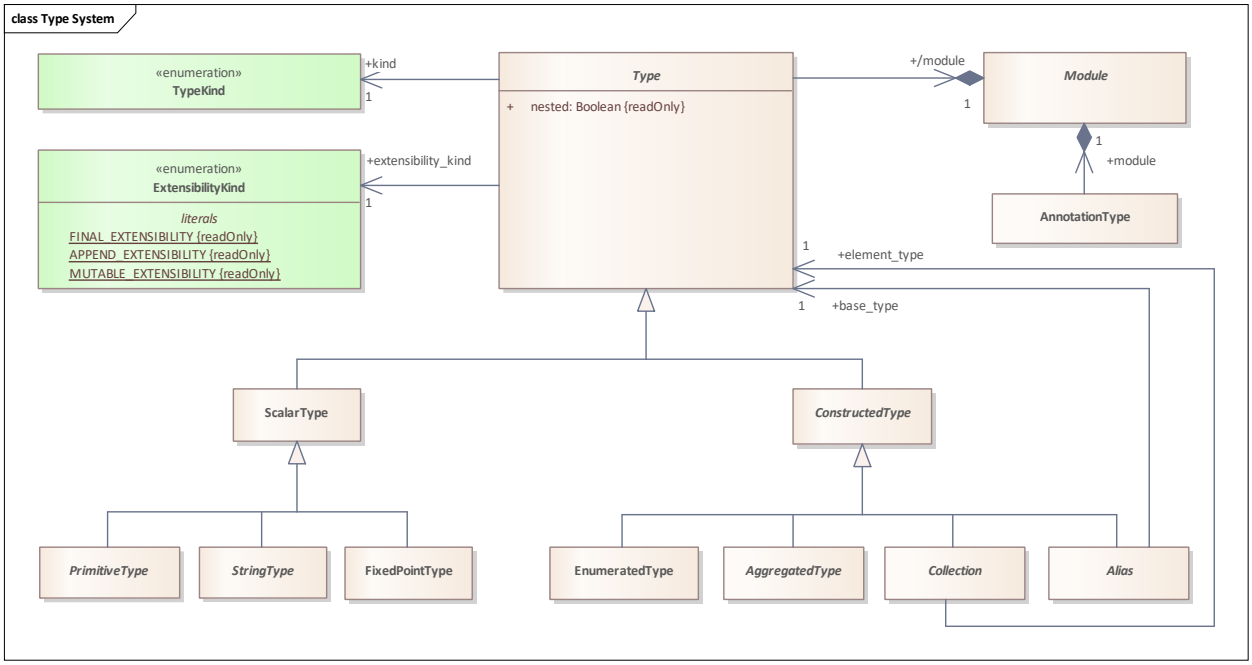

Figure 5 – Type System Model

The definition of a type in the Type System can either be primitive or it can be constructed from the definitions of other types.

The Type System model is shown in Figure 5.

7.2.3.1 Scalar Types

A Scalar Type is a type whose values are single, atomic values and which is not constructed from other XTypes types.

Scalar types include:

- Primitive Types, see 7.2.3.1.1, including integer, floating-point, boolean, and character types.
- String Types, see 7.2.3.1.2, including bounded and unbounded String<Char8> and String<Char16>.
- Fixed-Point Decimal Types, see 7.2.3.1.3.

7.2.3.1.1 Primitive Types

The primitive types in the Type System have parallels in most computer programming languages and are the building blocks for more complex types built recursively as collections or aggregations of more basic types.

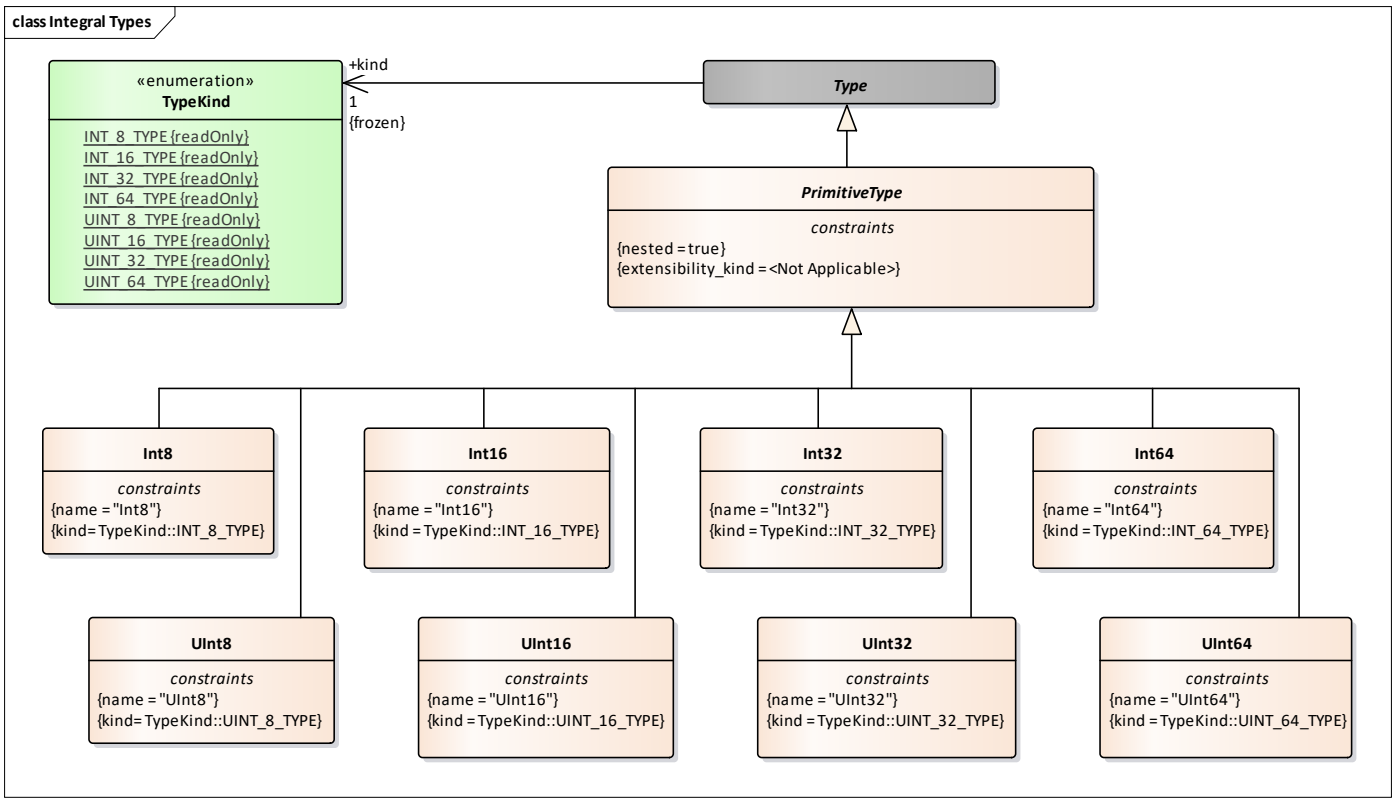

Figure 6 – Primitive Types: Integral Types

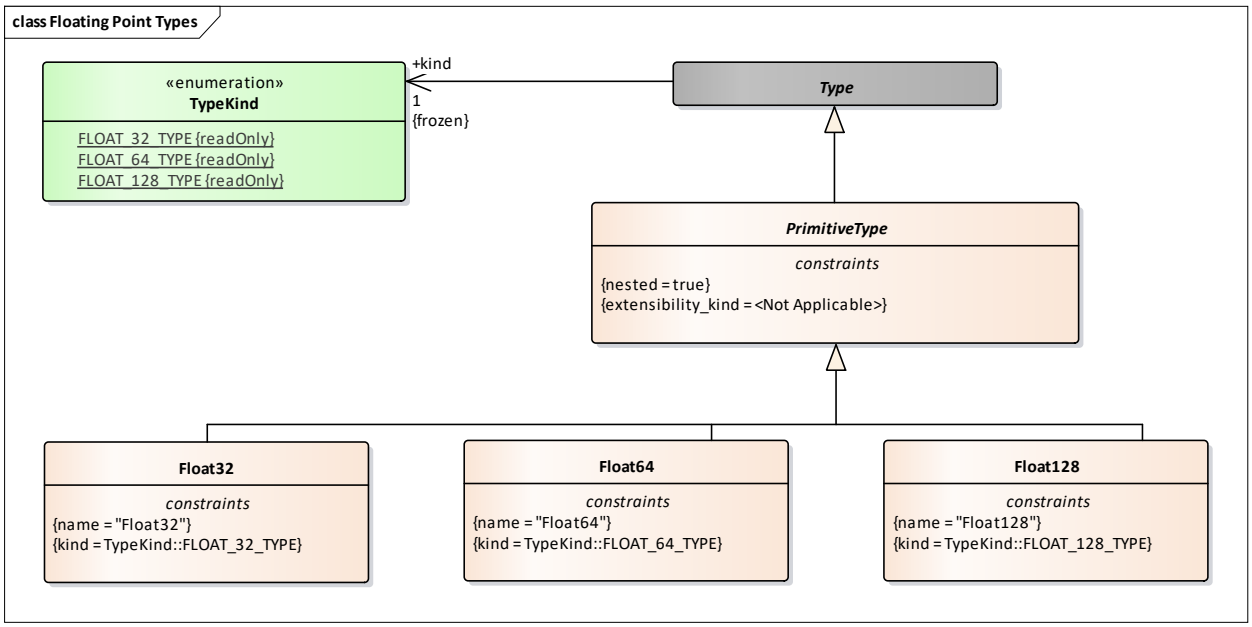

Figure 7 – Primitive Types: Floating Point Types

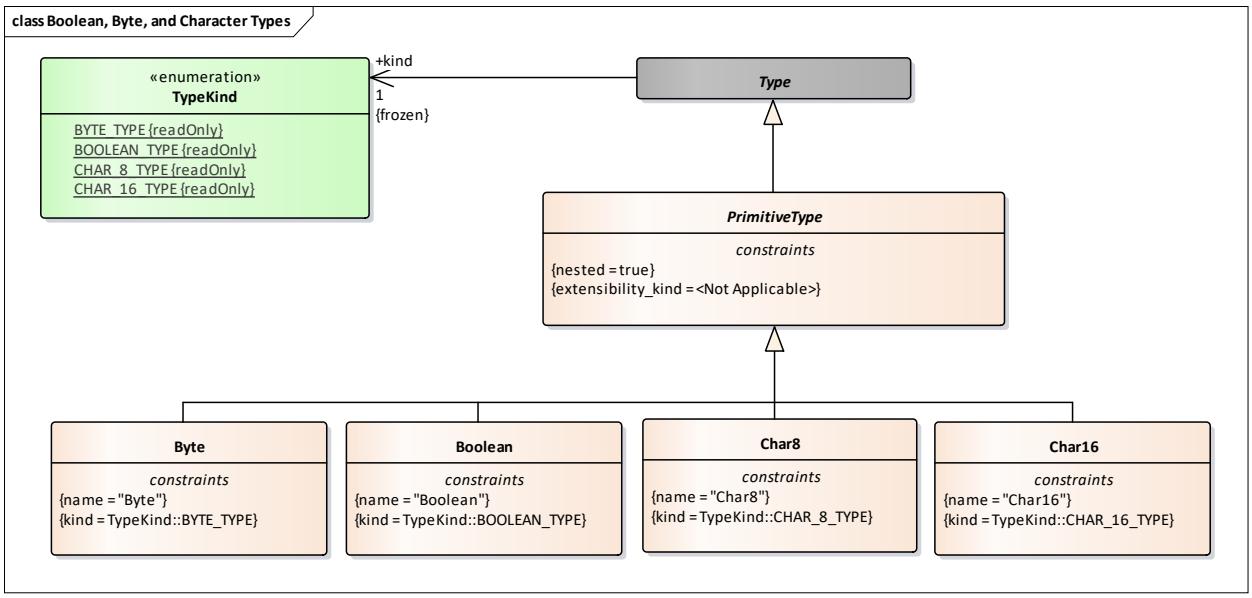

Figure 8 – Primitive Types: Booleans, Bytes, and Characters

Primitive Types include the primitive types present in most programming languages, including Boolean, integer, floating point, and character.

Table 4 below enumerates and describes the available primitive types. Note that value ranges are in this package specified only in terms of upper and lower bounds; data sizes and encodings are the domain of the Type Representation and Data Representation packages.

Table 4 – Primitive Types

Type Kind	***Type Name***	***Description***
`BOOLEAN_TYPE`	`Boolean`	Boolean type. Data of this type can only take two values: true and false.
`BYTE_TYPE`	`Byte`	Single opaque byte. A Byte value has no numeric value.
`INT_8_TYPE`	`Int8`	Signed integer minimally capable of representing values in the range -127 to +128.
`UINT_8_TYPE`	`UInt8`	Unsigned integer minimally capable of representing values in the range 0 to +255.
`INT_16_TYPE`	`Int16`	Signed integer minimally capable of representing values in the range -32768 to +32767.
`UINT_16_TYPE`	`UInt16`	Unsigned integer minimally capable of representing values in the range 0 to +65535.
`INT_32_TYPE`	`Int32`	Signed integer minimally capable of representing values in the range -2147483648 to +2147483647.

UINT_32_TYPE	UInt32	Unsigned integer minimally capable of representing values in the range 0 to +4294967295.
INT_64_TYPE	Int64	Signed integer minimally capable of supporting values in the range -9223372036854775808 to +9223372036854775807.
UINT_64_TYPE	UInt64	Unsigned integer minimally capable of supporting values in the range 0 to +18446744073709551617.
FLOAT_32_TYPE	Float32	Floating point number minimally capable of supporting the range and precision of an IEEE 754 single-precision floating point value.
FLOAT_64_TYPE	Float64	Floating point number minimally capable of supporting the range and precision of an IEEE 754 double-precision floating point value.
FLOAT_128_TYPE	Float128	Floating point number minimally capable of supporting the range and precision of an IEEE 754 quadruple-precision floating point value.
CHAR_8_TYPE	Char8	8-bit character type. There is no encoding specified, it may be ASCII, ISO-8859-1, or used to hold a byte of a multi-byte-encoded character set.
CHAR_16_TYPE	Char16	16-bit character type capable of supporting the Basic Multilingual Plane (BMP) encoded in UTF-16.

The primitive types do not exist within any module; their names are top-level names.

7.2.3.1.1.1 Character Data

The character types identified above require further definition, provided here.

7.2.3.1.1.2 Design Rationale (Non-Normative)

Because the Unicode character set is a superset of the US-ASCII character set, some readers may question why this specification provides two types for character data: `Char8` and `Char16`. These types are differentiated to facilitate the efficient representation and navigation of character data as well as to more accurately describe the designs of existing systems.

Existing languages for type definition—including C, C++, and IDL—distinguish between regular and wide characters (C/C++ `char` vs. `wchar_t`; IDL `char` vs. `wchar`). While other commonly used typing systems do not make such a distinction—in particular Java and the ECMA Common Type System, of which Microsoft's .Net is an implementation—it is more straightforward to map two platform-independent types to a single platform-specific type than it is to map objects of a single platform-independent type into different platform-specific types based on their values.

7.2.3.1.1.3 Character Sets and Encoding

7.2.3.1.1.3.1 Use of Unicode

This specification uses the Unicode Standard (version 9.0, June 2016) as the means to represent characters and strings.

Unicode defines a codespace of 1,114,112 code points in the range 0x000000 to 0x10FFFF. A Unicode code point is referred to by writing "U+" followed by its hexadecimal number (e.g. U+0000F1).

In the Unicode standard, a plane is a continuous group of 2^16 code points. There are 17 planes, identified by the numbers 0 to 16, which corresponds with the possible values 0x00-0x10 of the first two positions in six position format (hhhhhh).

Plane 0 is called the Basic Multilingual Plane (BMP). It contains nearly all commonly used writing systems and symbols. It contains characters U+0000 to U+FFFF. Planes 1–16, are called “supplementary planes”. As of Unicode version 9.0, six of the planes have assigned code points (characters), and four are named.

Unicode can be implemented by different character encodings. The most commonly used encodings are UTF-8, UTF-16, and UTF-32 (in that order). The Unicode code point is shared across all these encodings.

The UTF-8 encoding is backward compatible with the ASCII character set and is the default one used by most C and C++ compilers. The UTF-8 representation of ASCII characters uses one 8-bit code unit. The UTF-8 representation of ISO-8859-1 characters that are not in the ASCII subset uses two 8-bit code units. Any character in the Basic Multilingual Plane is encoded using one to three UTF-8 code units.

The UTF-16 encoding represents the code points in the Basic Multilingual Plane using one 16-bit code unit. The remaining Unicode characters use two 16-bit code units. The representation is numerically equal to the corresponding code points using the selected endianness.

7.2.3.1.1.3.2 CHAR_8_TYPE

This specification does not define an encoding for the `CHAR_8_TYPE`. The only constraint is that it shall be representable using 8 bits.

Rationale

By not specifying an encoding for `CHAR_8_TYPE` it is possible to use the 8-bit code-unit to either store a single ISO-8859-1 character or alternatively a code-unit of a UTF-8 encoded string.

7.2.3.1.1.3.3 Array or Sequence of CHAR_8_TYPE

This specification does not define an encoding for the `CHAR_8_TYPE` that appears as an element of an array or sequence of `CHAR_8_TYPE`.

Rationale

By not specifying an encoding for the elements of an Array or Sequence of `CHAR_8_TYPE` it becomes possible to store the characters of a String type into an Array or Sequence of `CHAR_8_TYPE` regardless of the encoding used in the String.

CHAR_16_TYPE

The `CHAR_16_TYPE` shall be restricted to representing Unicode codepoints in the Basic Multilingual Plane. That is Unicode codepoints from 0x0000 to U+FFFF.

The `CHAR_16_TYPE` encoding shall be UTF-16.

Rationale

UTF-16 is more space efficient than UTF-32. UTF-16 also maps directly to the Java and C# languages, which makes serialization and deserialization simple in those languages.

The BMP captures nearly all commonly used writing systems and symbols. Restricting to the BMP ensures that each codepoint is represented using a single UTF-16 code unit (16 bits)

7.2.3.1.1.3.5 Array or Sequence of CHAR_16_TYPE

The representation of each `CHAR_16_TYPE` element of an array or sequence of `CHAR_16_TYPE` shall be UTF-16 and shall be restricted to being in the Basic Multilingual Plane (Unicode codepoints from 0x0000 to U+FFFF).

String Types

StringTypes are ordered one-dimensional collections of characters. StringTypes are variable-sized; objects of a given string type can have different numbers of elements (i.e., the string object's "length"). Furthermore, the length of a given string object may change between zero and the string type's "bound" (see below) over the course of its lifetime.

A string is logically very similar to a sequence. However, the element type of a string must be either `Char8` or `Char16` (or an alias to one of these); other element types are undefined. These two collections have been distinguished in order to preserve the fidelity present in common implementation programming languages and platforms.

String types are considered global; they do not exist within any module. They are Anonymous types, i.e. they do not have a name.

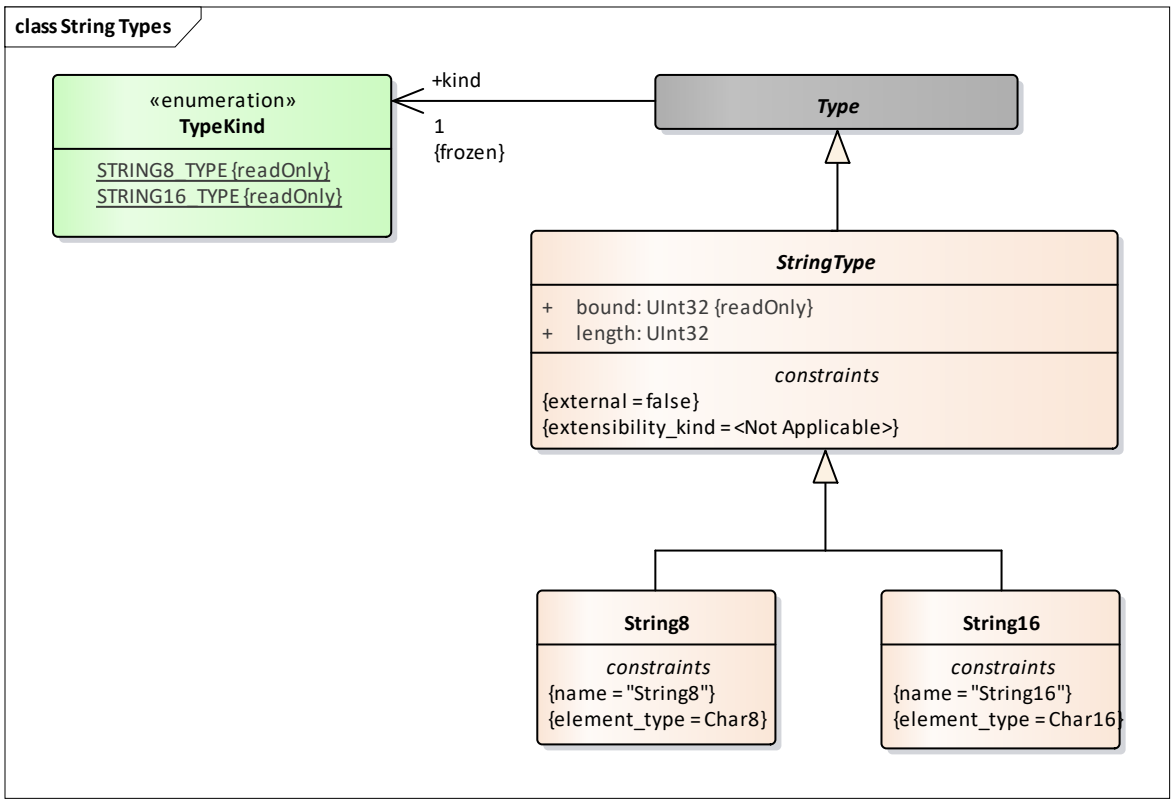

Figure 9 - String Types

7.2.3.1.2.1 String<Char8> type

The default encoding for `String<Char8>` shall be UTF-8. This encoded shall be used for the externalized Data Representation (see clause 7.4). Language bindings (see Clause 7.5) may use the representation that is most natural in that particular language. If this is different from UTF-8 the language binding shall manage the transformation to/from UTF-8 external Data Representation.

String<Char8> types can be **bounded**, indicating that there is a limit in the number of characters they may hold or **unbounded**.

For bounded strings, the string **bound** is the maximum number of octets that can be used to hold the UTF-8 representation of the string. This will match the maximum number of characters when the string consists exclusively of ASCII characters, however, in cases where the UTF-8 representation of some characters takes more than one byte, the number of characters the string can held will be less than the bound.

The specification of the string bound depends on the type representation. For example, when using the IDL type representation a string member with a bound of 32 octets is specified as:

```
struct MyStruct {
    string<32> str;
}
```

The string **length** is defined to be the number of bytes used to represent the string.

The reason for these definitions is to be able to handle the strings without needed to interpret the character encoding and be compatible with what some programming languages do, specifically the Standard C++ `std::string type`.

7.2.3.1.2.2 String<Char16> type

The encoding for `String<Char16>` shall be UTF-16. This encoded shall be used for the externalized Data Representation (see Clause 7.4). Language bindings (see Clause 7.5) may use the representation that is most natural in that particular language. If this is different from UTF-16 the language binding shall manage the transformation to/from UTF-16 external Data Representation.

String<Char16> types can be **bounded**, indicating that there is a limit in the number of characters they may hold or **unbounded**.

For bounded strings, the string **bound** is the maximum number of UTF-16 code units that the string can hold. Note that for the characters in the Basic Multilingual Plane (BMP), each UTF-16, however for characters outside the BMP (e.g. emoji) a single UTF-16 character can take two UTF-16 code units (4 bytes).

The specification of the string bound depends on the type representation. For example, when using the IDL type representation a string member with a bound of 32 (UTF-16) characters is specified as:

```
struct MyStruct {
    wstring<32> str;
}
```

The string **length** is defined to be the number of octets used to represent the string. This will always be twice the number of UTF-16 code units in the string since each UTF-16 code units takes two octets.

7.2.3.1.3 Fixed Point Types

Fixed-Point Types represent signed fractional numeric values encoded using a fixed-precision format parameterized by **precision**, **scale**, and **base**:

- **Precision** (also referred as **Digits**). The total number of significant digits (or discrete magnitude increments) that the representation can encode.
- **Scale**. The number of fractional digits (or fractional increments) to the right of the radix point. It is constrained to the range 0 <= scale <= precision.
- **Base**. The numeric base in which digits are expressed (for example, base 10 decimal).

Fixed-Point Types have an *unscaled_value* and a *scaled_value*. The term *value*, when used without qualification, shall refer to the *scaled_value*.

The *unscaled_value* is a signed integer in the range:

$$-(base^{precision} - 1) <= unscaled_value \leq (base^{precision} - 1)$$

The *scaled_value* is a signed numeric value defined as:

$$scaled_value = unscaled_value * base^{-scale}$$

Example:

Assuming precision=9, scale=2, and base=10. The *unscaled_value* 123456789 corresponds to the *scaled value* 1234567.89

This specification currently defines one fixed-point type:

- Fixed-Point Decimal Types, see 7.2.3.1.3.1.

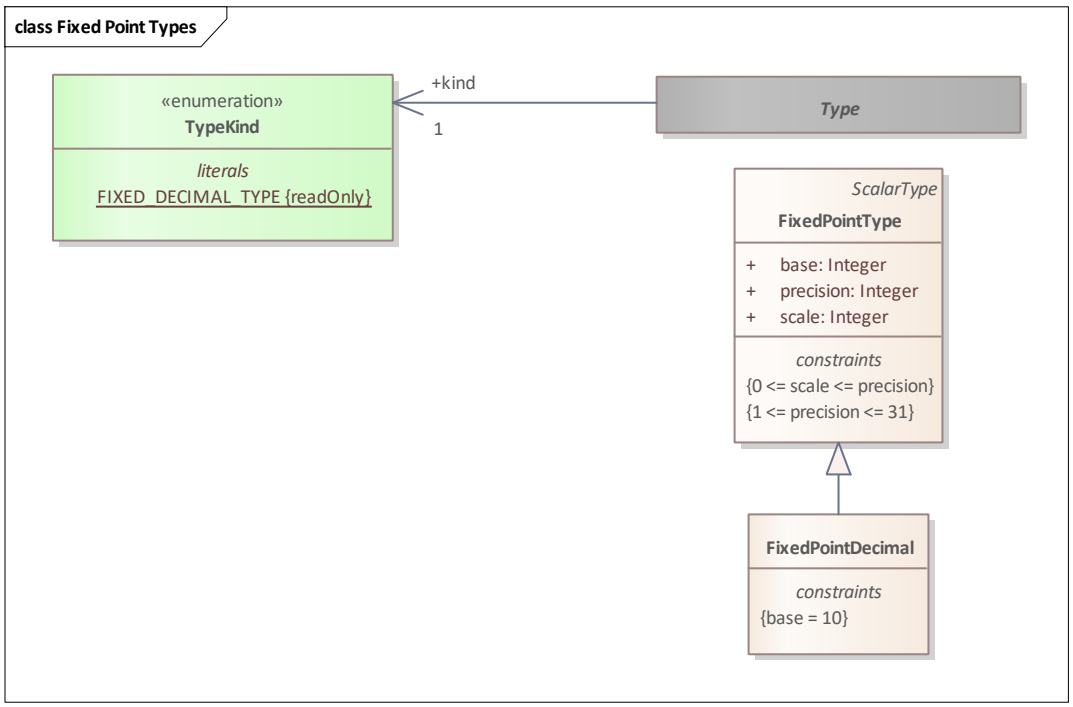

Figure 10 – Fixed Point Types

7.2.3.1.3.1 Fixed Point Decimal Types

A Fixed-Point Decimal type is a Fixed-Point Type whose digits are expressed in base-10, Precision is constrained to be between 1 and 31.

Using the IDL type representation 7.3.1 Fixed Decimal Types are specified using the `fixed` keyword. For example:

```
// First template parameter is precision (number of total digits)
// Second template parameter is scale (number of fractional digits)
typedef AccountBalance fixed<12, 2>;
```

7.2.3.2 Constructed Types

The definitions of these types are constructed from—that is, based upon—the definitions of other types. These other types may be either primitive types or other constructed types: type definitions may be recursive to an arbitrary depth. Constructed types are explicitly defined by a user of an implementation of this specification and are assigned a name when they are defined.

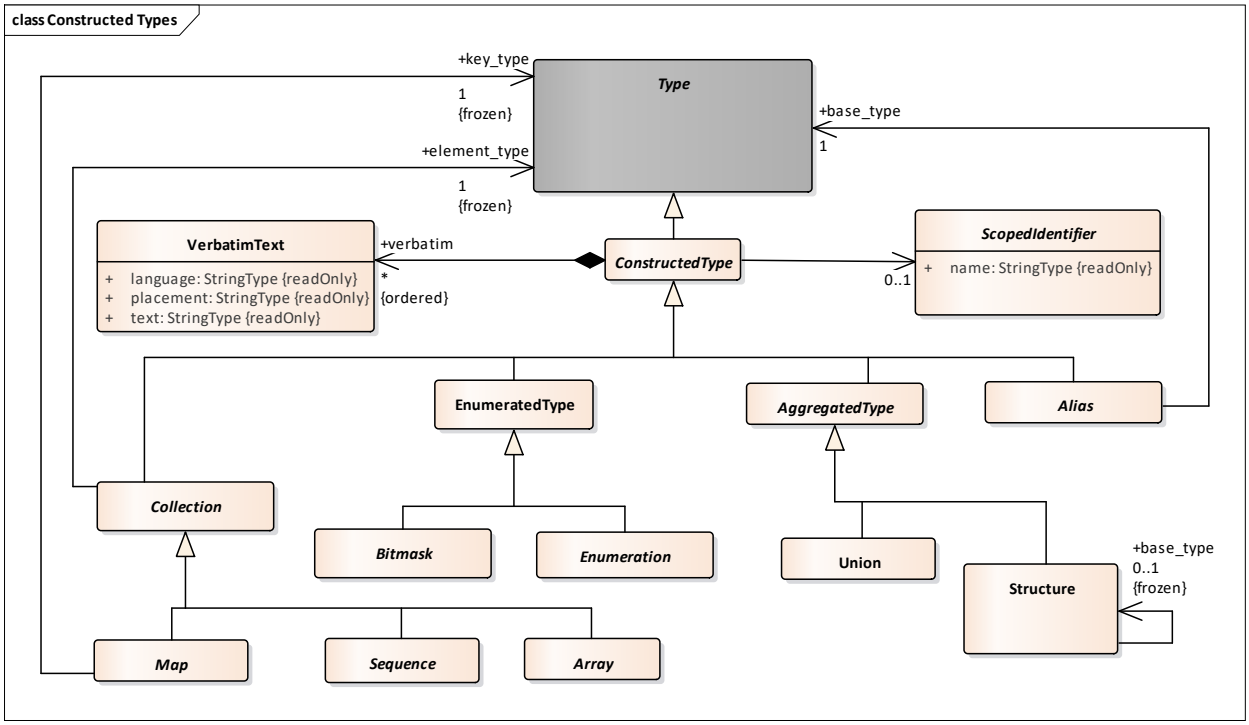

Figure 11 – Constructed Types

There are several kinds of Constructed Types: Collections, Aggregated types, Aliases, and Enumerated types. Collections are homogeneous in that all elements of the collection have the same type. Aggregated types are heterogeneous; members of the aggregated types may have different types. Aliases introduce a new name for another type. Enumerated types define a finite set of possible integer values for the data.

7.2.3.2.1 Enumerated Types

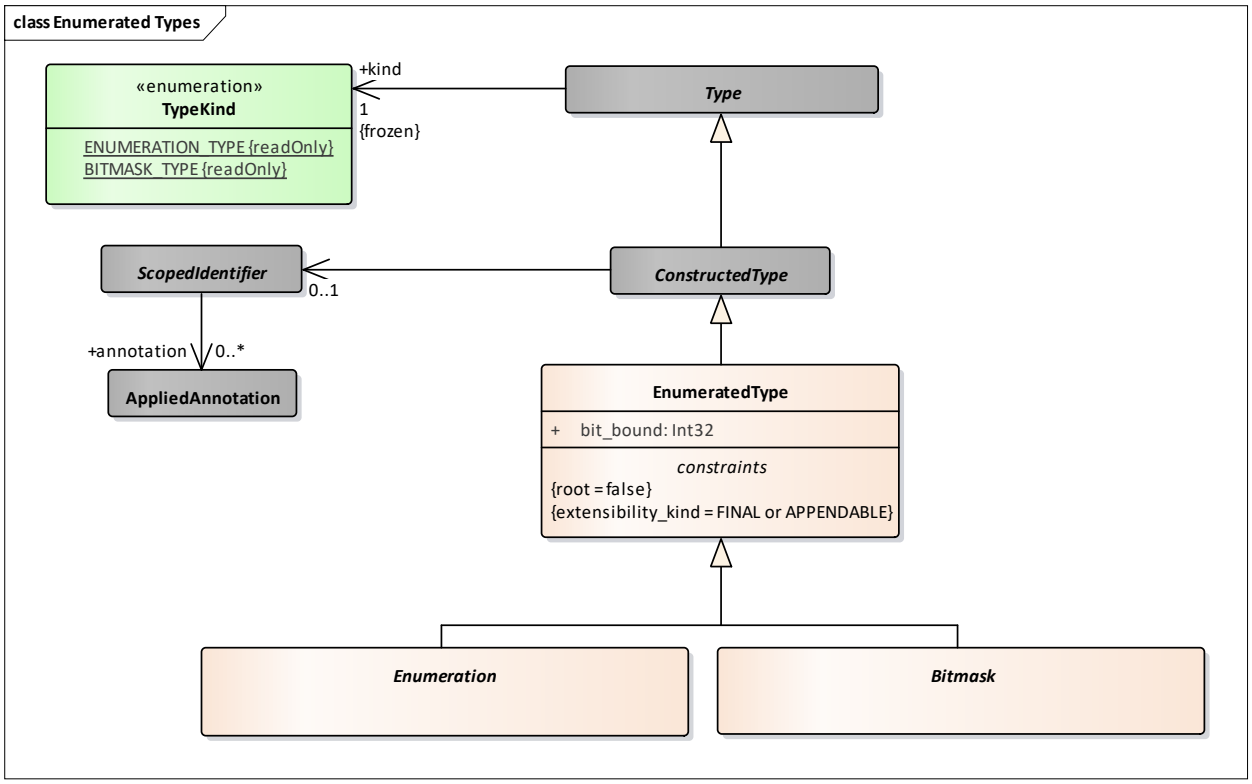

Figure 12 – Enumerated Types

7.2.3.2.1.1 Enumeration Types

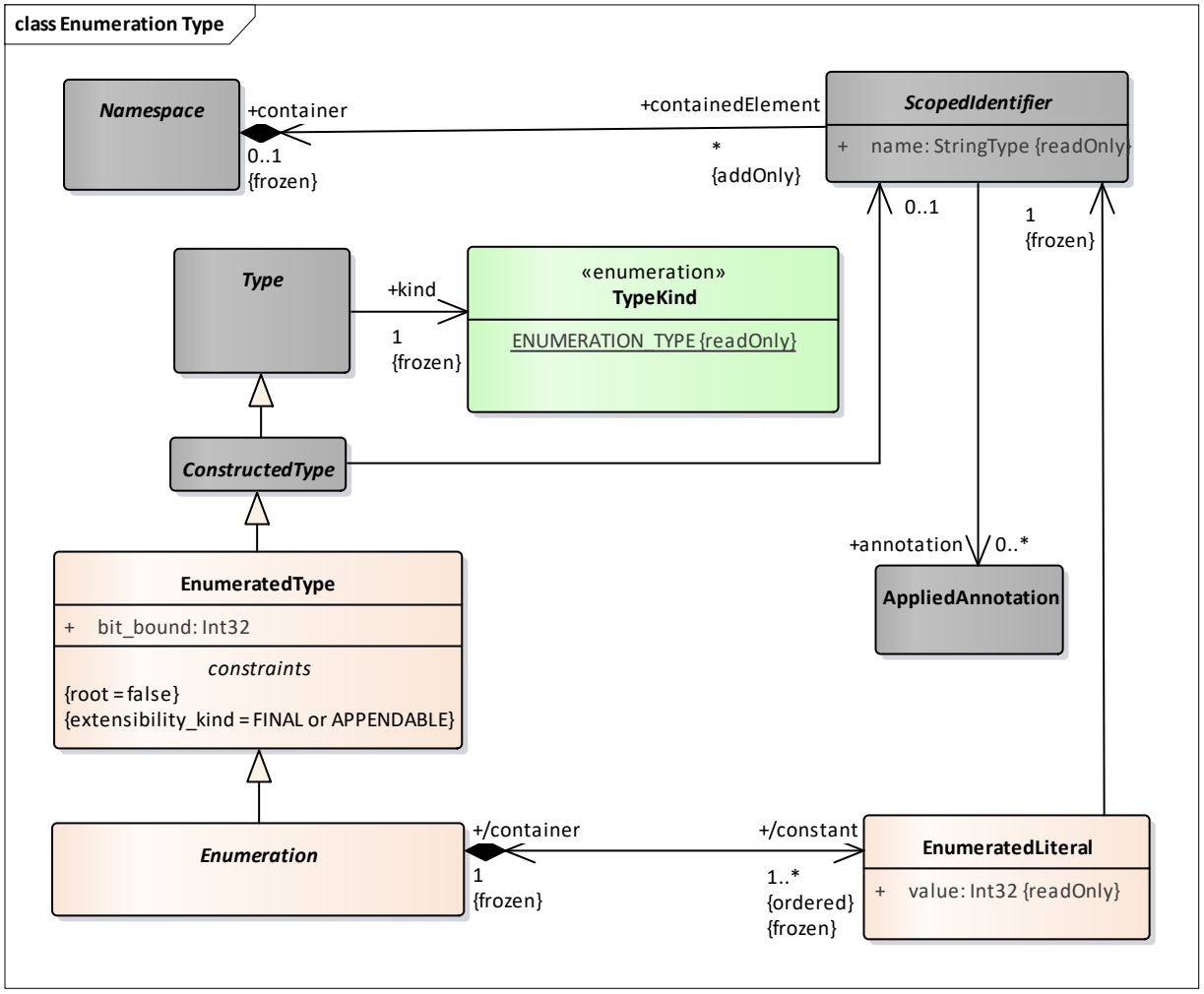

Figure 13 – Enumeration Types

Table 5 – Enumerated types

Type Kind	*Type Name*	*Description*
`ENUMERATION_TYPE`	*Assigned when type is defined*	**Set of literals.** An enumerated type defines a closed set of one or more literal objects of that type. Each object of a given enumerated type has a name and a signed `Integer` value that are each unique within that type. The order in which the literals of an enumerated type are defined is significant to the definition of that type. For example, some type representations may base the numeric values of the literals on their order of definition.

7.2.3.2.1.2 Bitmask Types

Bitmasks, as in the C++ standard library (and not unlike the `EnumSet` class of the Java standard library), represent a collection of Boolean flags, each of which can be inspected and/or set individually.

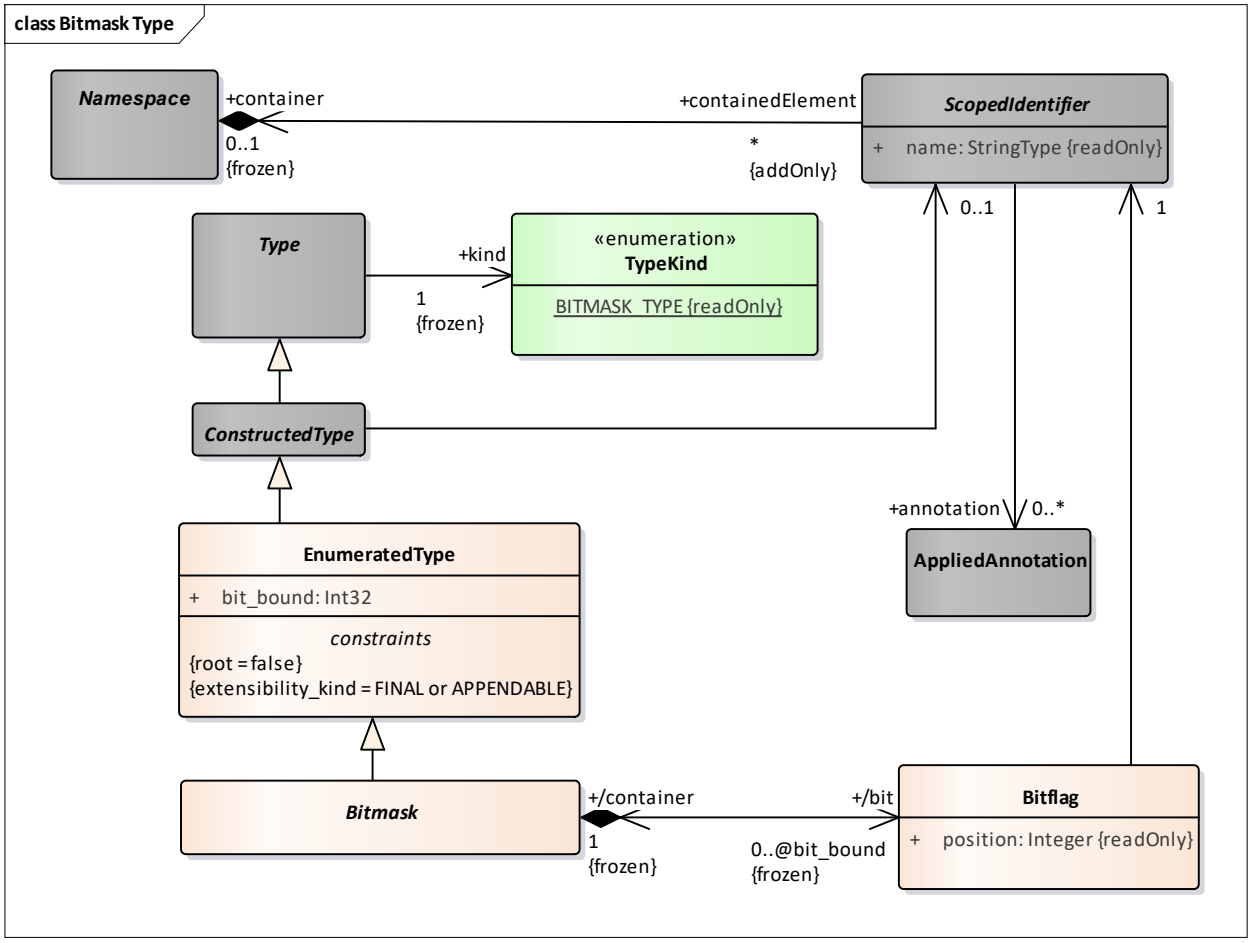

Figure 14 – Bitmask Types

Table 6 – Bitmask types

Type Kind	*Type Name*	*Description*
`BITMASK_TYPE`	*Assigned when type is defined*	**Ordered set of named Boolean flags.** A bitmask defines a bound—the maximum number of bits in the set—and identifies by name certain bits within the set. The bound must be greater than zero and no greater than 64.

A bitmask type reserves a number of "bits" (Boolean flags); this is referred to as its bound. (The bound of a bitmask is logically similar to the bound of an array, except that the "elements" in a bitmask are single bits.) It then identifies some subset of those bits. Each bit in this subset is

identified by name and by an a position, numbered from 0 to (bound – 1). The bitmask need not identify every bit it reserves. Furthermore, the bits it does identify need not be contiguous.

Note that this type exists for the sake of semantic clarity and to enable more efficient data representations. It does not actually constrain such representations to represent each "bit" in the set as a single memory bit or to align the bitmask in any particular way.

Each Bitmask Flag has an associated flag position, which specifies the bit index corresponding to that flag (see 7.2.3.2.1.3.4).

7.2.3.2.1.2.1 Design Rationale (Non-Normative)

It is commonly the case that complex data types need to represent a number of Boolean flags. For example, in the DDS specification, status kinds are represented as `StatusKind` bits that are combined into a `StatusMask`. A bitmask (also referred to as a bit mask) allows these flags to be represented very compactly—typically as a single bit per flag. Without such a concept in the type system, type designers must choose one of two alternatives:

- Idiomatically define enumerated "kind" bits and a "mask" type. Pack and unpack the former into the latter using bitwise operators. As previously noted, this is the approach taken by the DDS specification in the case of statuses, because it predated this enhanced type model. There are several weaknesses to this approach:
    - It is verbose, both in terms of the type definition and in terms of the code that uses the bitmask; this verbosity slows understanding and can lead to programming errors.
    - It is not explicitly tied to the semantics of the data being represented. This weakness can lead to a lack of user understanding and type safety, which in turn can lead to programming errors. It furthermore hampers the development of supporting tooling, which cannot interpret the "bitmask" otherwise than as a numeric quantity.
- Represent the flags as individual Boolean values. This approach simplifies programming and provides semantic clarity. However, it is extremely verbose: a structure of Boolean members wastes at least 7/8 of the network bandwidth it uses (assuming no additional alignment and that each flag requires one bit but occupies one byte) and possible up to 31/32 of the memory it uses (on platforms such as Microsoft Windows that conventionally align Boolean values to 32-bit boundaries).

7.2.3.2.1.3 Enumerated Type Attributes and Metadata

Bit Bound

A bit bound defines the maximum number of bits used to represent a value of Enumeration and Bitmask Types.

The bit bound is a positive integer that specifies the maximum bit-width of the unsigned integer used to hold the value. A bit bound shall be greater than zero and no greater than 64.

When omitted, the bit bound defaults to 32.

7.2.3.2.1.3.2 Enumerator Value

Each enumeration literal has an associated `Enumerator Value` attribute, which identifies the integer value that the literal represents within the enumeration.

If the Type Representation does not explicitly assign a value to a literal, the enumerator value is implicitly assigned by incrementing from the previous literal's value, starting at zero for the first literal.

7.2.3.2.1.3.3 Default Literal

The type system supports optional specification of the `Default Literal` associated with an Enumeration type.

To support this, each literal of an Enumeration type has a "default_literal" attribute that may be set to true or false.

At most one literal may have the default_literal" attribute set to **true**.

The concrete expression of default literal association depends on the Type Representation (see 7.3). Using the IDL type representation (7.3.1), member default values are specified with the annotation `@default_literal`.

Example using IDL:

```
enum Weekday {
    MONDAY,
    TUESDAY,
    @default_literal WEDNESDAY,
    THURSDAY,
    FRIDAY
};
```

If no literal is explicitly marked as `default_literal` then the first literal becomes the default_literal.

```
enum Weekday {
    MONDAY,   // This is the default literal since none is explicitly marked
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY
};
```

7.2.3.2.1.3.4 Flag Position

Each Bitmask Flag has a `Flag Position` attribute that identifies the bit index corresponding to that flag within the Bitmask's underlying unsigned integer representation.

The flag position is a non-negative integer, starting from 0 for the least-significant bit. Each flag within a Bitmask shall have a unique position value.

If the position is not explicitly specified, it is assigned implicitly in increasing order from the previous flag, starting from 0.

7.2.3.2.2 Alias Types

Alias types define a new type that is structurally equivalent to another type but has a different name. The new type may also have additional constraints e.g. specification of valid numeric ranges or engineering units.

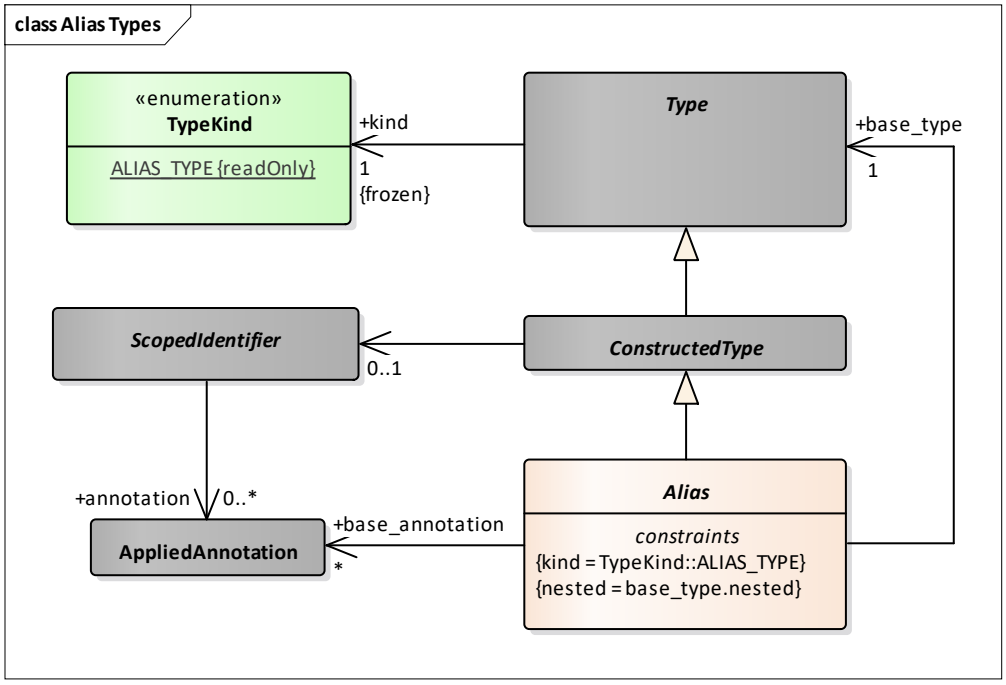

Figure 15 – Alias Types

Table 7 – Alias types

Type Kind	*Type Name*	*Description*
`ALIAS_TYPE`	*Assigned when type is defined*	**Alternative name for another type.** An alias type—also referred to as a *typedef* from its representation in IDL, C, and elsewhere—creates a new type from an already-existing type. The new type has also a new name which can be helpful for suggesting particular uses and semantics to human readers, making it easier to repeat complex type names for human writers, and simplifying certain language bindings.

		Unlike the C and C++ programming languages, an alias/typedef does introduce a new distinct type. The new type is compatible with the referenced type.

7.2.3.2.2.1 Default Value

The type system supports optional specification of the default value associated with Alias types.

The default value association may be direct or indirect:

- Direct association: default value is specified in the declaration of the Alias type.
- Indirect association: default value is inherited from the base type referenced by the Alias type.

The concrete expression of direct default value association depends on the Type Representation (see 7.3). Using the IDL type representation 7.3.1 default values are specified using the annotation `@default`.

Example:

```
typedef @default(32) float Temperature; // direct default value
```

If an Alias type T1 does not directly associate a default value, then it is indirectly associated with the default value of its base type T2, if one exists.

Example:

```
typedef Temperature TFarenheit; // OK. indirect default association (32)
```

If an Alias type specifies a default value directly, its base type shall not be an Alias type that itself specifies a default value directly, nor one that recursively resolves to such an Alias.

Examples:

```
typedef @default(100) Temperature TCelsius; // NOT OK base 'Temperature' has
                                  // directly associated default value

typedef @default(100) TFarenheit TFarenheit2; // NOT OK base 'TFarenheit'
                                  // recursively resolves to 'Temperature'
                                  // which has a direct default value
```

If the declaration of the Alias type does not associate a direct default value then is indirectly associated with the default value its base type. See Table 10.

7.2.3.2.2.2 Units

The type system supports optional specification of the units associated with Alias types (see 7.2.2.9). The unit association may be direct or indirect.

- Direct association: units are specified in the declaration of the Alias type.
- Indirect association: units are inherited from the base type referenced by the Alias type.

The concrete expression of direct unit association depends on the Type Representation (see 7.3). Using the IDL type representation (7.3.1), units are specified with the annotation `@unit`.

Examples:

```
typedef @unit("degrees") int32 Angle; // direct units on 'Angle'
typedef @unit("C; celsius") float Temperature;
typedef @unit("m, m/s, m/s^2") float StateVector[3];
typedef @unit("km/h") sequence<float> LapSpeeds;
```

Note — The example above uses a comma-separated list as a convention to indicate different units for each element of an array. This convention is not interpreted by the type system; it is simply an example usage that remains consistent with the unit compatibility rules.

If the Alias T1 does not directly associate units but its base type T2 has associated units, then T2's units are indirectly associated with T1:

```
typedef Angle Slope; // OK 'Slope' is (indirectly) associated
                     // with the units 'Angle' is associated with
```

If an Alias type has units directly associated with it, its base type shall not be an Alias type that already has associated units (directly or indirectly). Other than this restriction, the base type is unrestricted.

```
typedef @unit("deg")  Angle Slope2; // NOT OK base 'Angle' has
                                    // (directly) associated units
typedef Angle Incline; // OK base 'Incline' has (indirectly)
                       // associated units

typedef @unit("deg") Incline Slope3; // NOT OK base 'Incline' has
                                     // (indirectly) associated units
```

7.2.3.2.2.3 Range Limits

The type system supports optional specification of the range limits associated with Alias types (see 7.2.2.10). The association may be direct or indirect:

- Direct association: limits are specified in the declaration of the Alias type.
- Indirect association: limits are inherited from the base type referenced by the Alias type.

Range limits may be specified only if the base type is a numeric primitive type or recursively resolves to a numeric primitive type through intermediate Aliases.

The concrete expression of direct range limits depends on the Type Representation (see 7.3). Using the IDL type representation 7.3.1 range limits are specified with the annotations: `@range`, `@min`, and `@max`.

Examples:

```
typedef @range(0, 360) int32 Angle; // direct range limits
typedef @min(-273.15) float Temperature; // -273.15 C is absolute zero
```

If the base type T2 for an Alias type T1 has associated range limits, then:

- If T1 does not directly associate limits, then T2's limits are indirectly associated with T1.
- If T1 directly associate limits, then T1's limits shall be contained within T2's limits, or else it shall be treated as an error.

Examples:

```
typedef Angle Slope; // Slope is indirectly associated with
                     // 'Angle' limits: @range(0,360)

typedef @range(0,180) Slope Incline1; // OK 'Incline1' range (0,180) is
                                      // contained in the base type 'Slope'
                                      // range (0,360)

typedef @range(-180,180) Slope Incline2; // NOT OK 'Incline3'
                                      // range (-180,180)IS NOT contained
                                      // in base type 'Slope' range (0,360)
```

7.2.3.2.3 Collection Types

Collections are containers for elements of a homogeneous type. The type of the element might be any other type, primitive or constructed (although some limitations apply; see below) and must be specified when the collection type is defined.

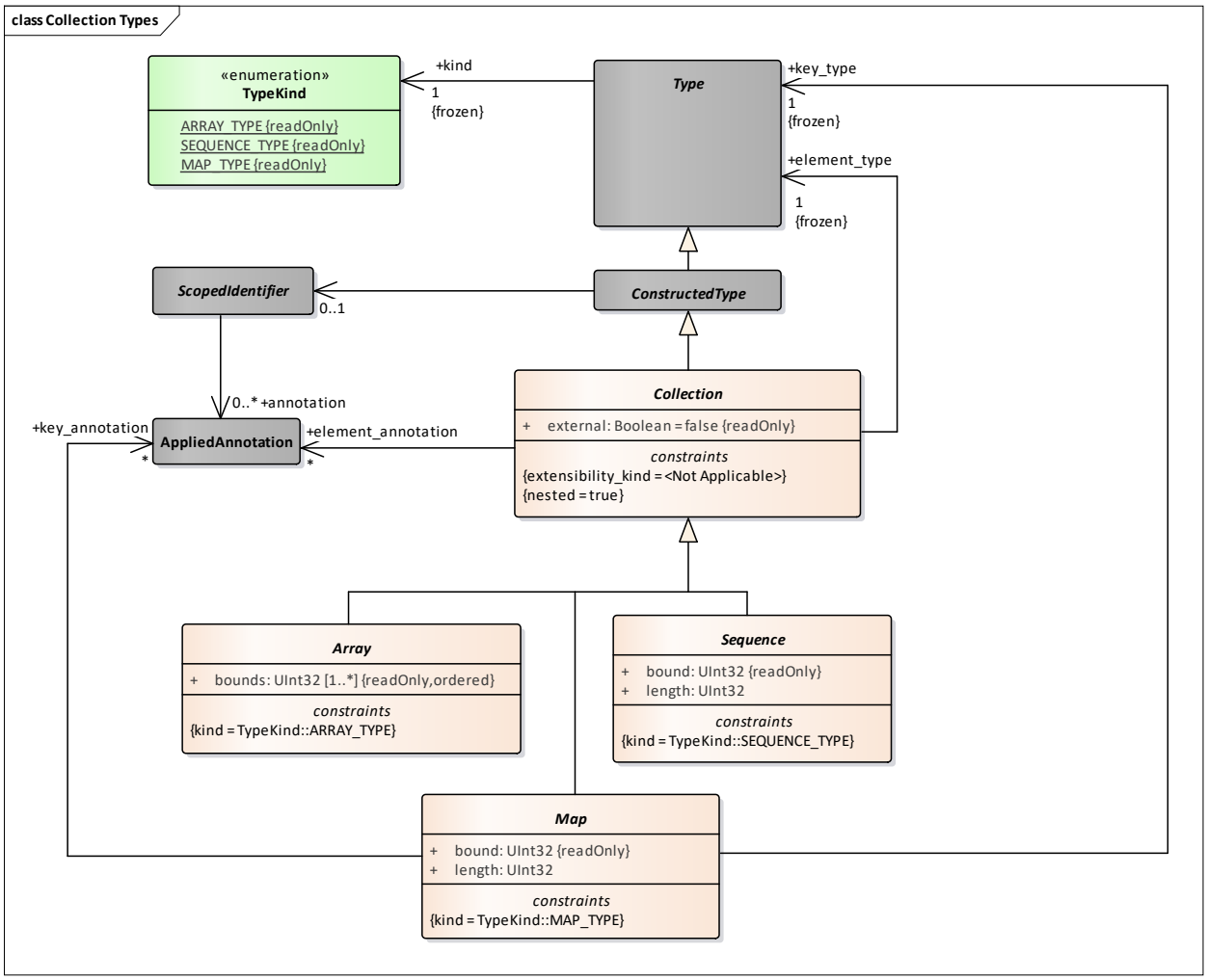

Figure 16 – Collection Types

Collection types are considered global. They do not exist within any module. They are Anonymous types, i.e. they do not have a name.

There are three kinds of Collection Types: `ARRAY`, `SEQUENCE`, and `MAP`. These kinds are described in Table 8.

Table 8 – Collection Types

Type Kind	*Type Name*	*Description*
`ARRAY_TYPE`	*Assigned implicitly*	**Fixed-size multi-dimensional collection.** Arrays are of a fixed size in that all objects of a given array type will have the same number of elements. Elements are addressed by a sequence of indices (one per dimension). Semantically, array types of higher dimensionality are distinct from arrays of arrays of lower dimensionality. (For example, a two-dimensional array is not just an array of one-dimensional arrays.) However, certain type representations may be unable to capture this distinction. (For example, IDL provides no syntax to describe an array of arrays[1], and in Java, all "multi-dimensional" arrays *are* arrays of arrays necessarily.) Such limitations in a given type representation should not be construed as a limitation on the type system itself.
`SEQUENCE_TYPE`	*Assigned implicitly*	**Variable-size single-dimensional collection.** Sequences are variably sized in that objects of a given sequence type can have different numbers of elements (the sequence object's "length"); furthermore, the length of a given sequence object may change between zero and the sequence type's "bound" (see below) over the course of its lifetime. Elements are addressed by a single index.
`MAP_TYPE`	*Assigned implicitly*	**Variable-size associative collection.** Maps are variably sized in that objects of a given map type can have different numbers of elements (the map object's "length"); furthermore, the length of a given map object may change between zero and the map type's "bound" (see below) over the course of its lifetime. "Map value" elements are addressed by a "map key" object, the value of which must be unique within a given map object. The types of both of these are homogeneous within a given map type and must be specified when the map type is defined.

[1] An intermediate alias can help circumvent this limitation; see below for a more formal treatment of aliases.

Collection types are defined implicitly as they are used. Their definitions are based on three attributes:

- **Collection kind**: The supported kinds of collections are identified in Table 8 above.
- **Element type**: The concrete type to which all elements conform. (Collection elements that are of a subtype of the element type rather than the element type itself may be truncated when they are serialized into a Data Representation.)

  In the case of a map type, this attribute corresponds to the type of the *value* elements. Map types have an additional attribute, the *key element type*, that indicates the type of the key elements. Implementers of this specification need only support key elements of signed and unsigned integer types and of narrow and wide string types; the behavior of maps with other key element types is undefined and may not be portable. (**Design rationale, non-normative**: Support for arbitrary key element types would require implementers to provide uniform sorting and/or hashing operations, which would be impractical on many platforms. In contrast, these operations have straightforward implementations for integer and string types.)
- **Bound**: The maximum number of elements the collection may contain (inclusively); it must be greater than zero.

  In the cases of sequences, strings, and maps, the bound parameter may be omitted. If it is omitted, the bound is not specified; such a collection is referred to as "unbounded." (All arrays must be bounded.) In that case, the type may have *no* upper bound—meaning that the collection may contain any number of elements—*or* it may have an *implicit* upper bound imposed by a given type representation (which might, for example, provide only a certain number of bits in which to store the bound) or implementation (which might, for example, impose a smaller default bound than the maximum allowed by the type representation for resource management purposes). Because of this ambiguity, type designers are encouraged to choose an explicit upper bound whenever possible.

  In the cases of sequences, strings, and maps, the bound is a single value. Arrays have independent bounds on each of their dimensions; they can also be said to have an overall bound, which is the product of all of their dimensions' bounds.

For example, a one-dimensional array of 10 integers, a one-dimensional array of 10 short integers, a sequence of at most 10 integers, and a sequence of an unspecified number of integers are all of different types. However, all one-dimensional arrays of 10 integers are of the same type.

7.2.3.2.4 Aggregated Types

7.2.3.2.4.1 Overview

Aggregated types are containers for elements—"members"—of (potentially) heterogeneous types. Each member has a string name, a type, an integer ID, an integer index, and a set of boolean properties, see 7.2.3.2.4.4.

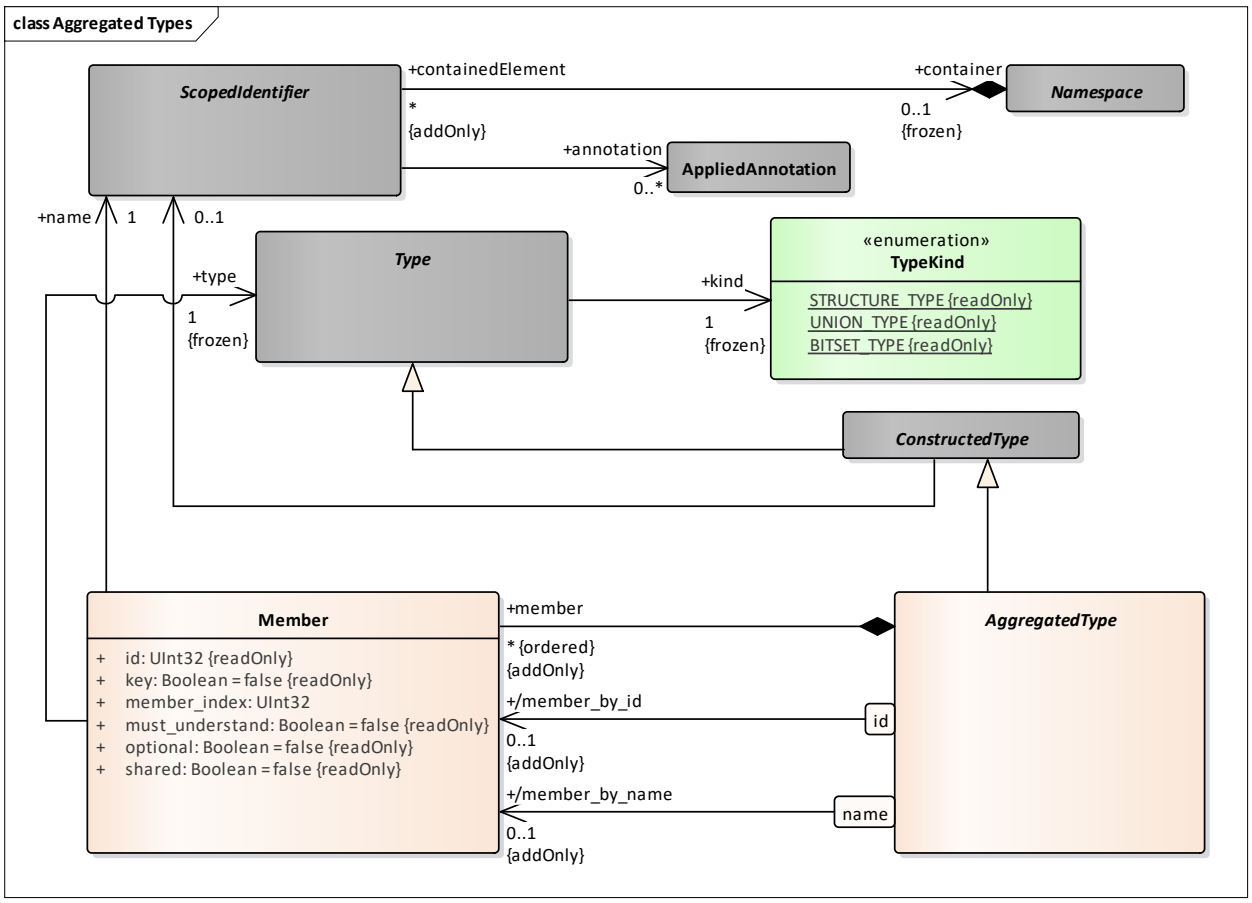

Figure 17 – Aggregated Types

There are three kinds of Aggregated Types: structures, unions, and annotations. These kinds are described in Table 9.

Table 9 – Aggregated Types

Type Kind	*Type Name*	*Description*
`UNION_TYPE`	*Assigned when type is defined*	**Discriminated exclusive aggregation of members.** Unions define a well-known discriminator member and a set of type-specific members.
`STRUCTURE_TYPE`	*Assigned when type is defined*	**Non-exclusive aggregation of members.** A type designer may declare any number of members within a structure. Unlike in a union, there are no implicit members in a structure, and values for multiple members may coexist.

7.2.3.2.4.2 Structure Types

A type designer may declare any number of members within a structure. Unlike in a union, there are no implicit members in a structure, and values for multiple members may coexist.

Subject to the constraints specified in 7.2.3.2.5. structure type can optionally extend a single other structure type, referred to as its `base_type`.

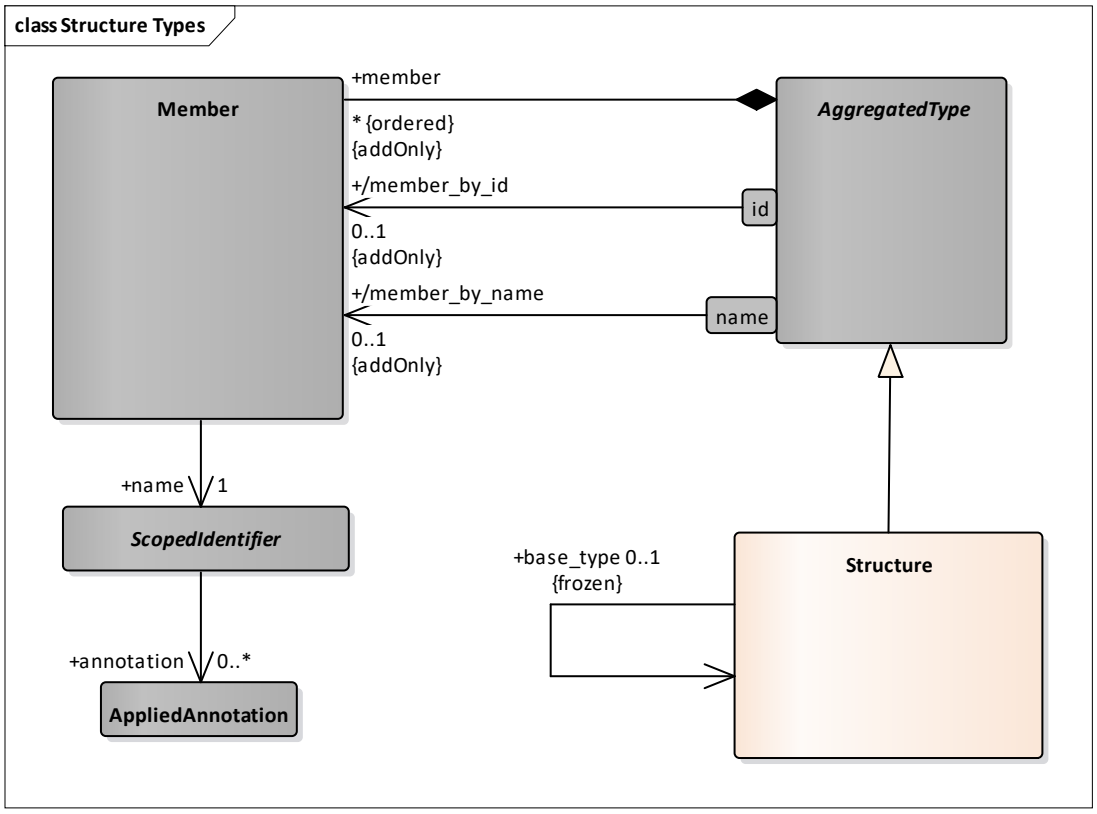

Figure 18 – Structure Types

7.2.3.2.4.3 Union Types

Unions define a well-known discriminator member and a set of type-specific members. The name of the discriminator member is always “discriminator”; that name is reserved for union types and is not permitted for type-specific union members. The discriminator member is always considered to be the first member of a union and has ID 0.

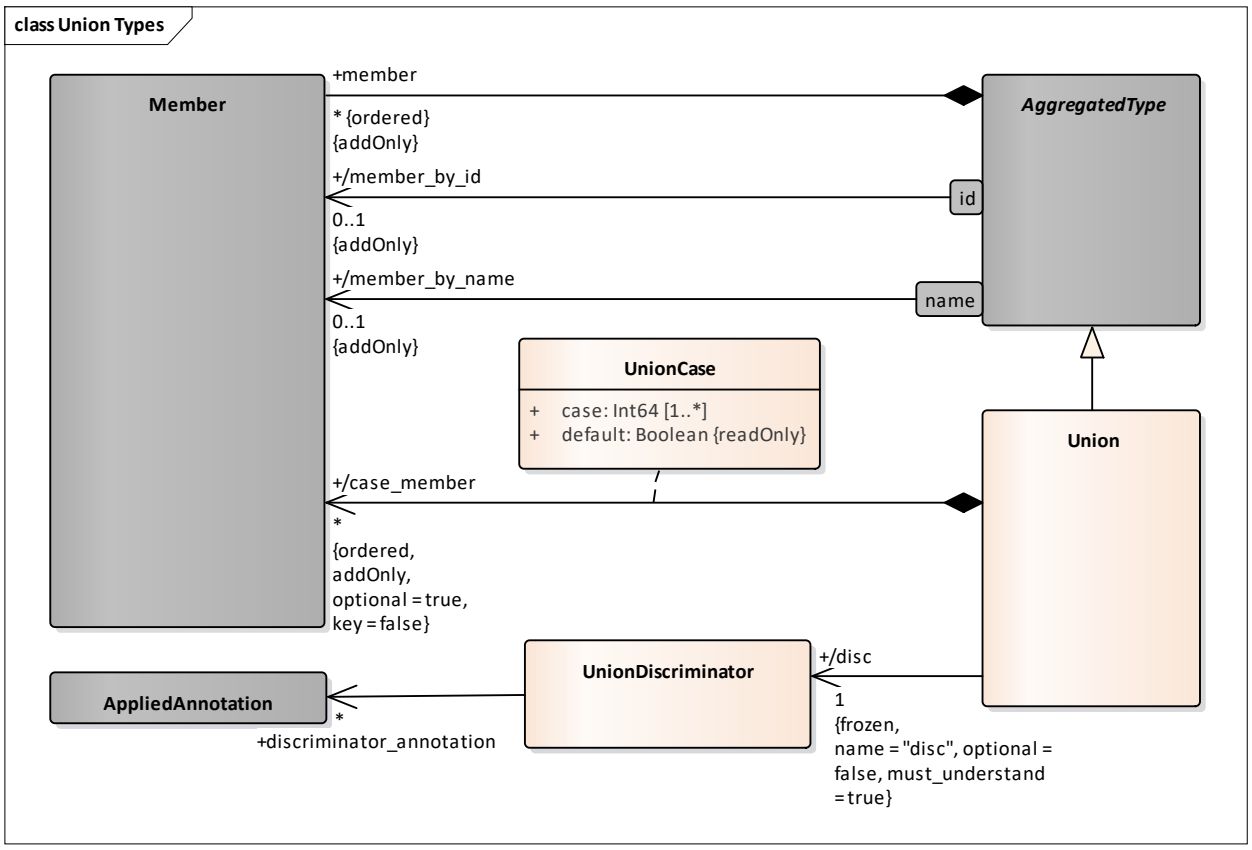

Figure 19 – Union Types

Each type-specific member is associated with one or more values of the discriminator. These values are identified in one of two ways:

(1) Explicitly, by associating a member with a list of discriminator values. It is not allowed for multiple members to explicitly identify the same discriminator value.

(2) Implicitly, by identifying at most one member of the union as the “default” member. In this case, any discriminator value that does not explicitly identify another member is considered to identify the default member.

These two mechanisms together guarantee that any given discriminator value identifies at most one member of the union. (*Note* that it is not required for every potential discriminator value to be associated with a member.) These mappings from discriminator values to members are defined by a union *type* and do not differ from object to object.

The value of the member associated with the current value of the discriminator is the only member value considered to exist in a given object of a union type at a given moment in time. However, the value of the discriminator field may change over the lifetime of a given object, thereby changing which union member’s value is observed. When such a change occurs, the initial value of the newly observed member is undefined by the type system (though it may be defined by a particular language binding). In particular, it is not defined whether, upon switching from a discriminator value *x* to a different value *y* and then immediately back to *x*, the previous value of the *x* member will be preserved.

The discriminator of a union must be of one of the following types:

- `Boolean`
- `Byte`
- `Char8`, `Char16`
- `Int8, UInt8, Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`
- Any enumerated type
- Any alias type that resolves, directly or indirectly, to one of the aforementioned types.

Subject to the constraints specified in 7.2.3.2.5. a union type can optionally extend a single other union type, referred to as its `base_type`.

Example using IDL Type Representation:

```
enum CommandKind {
    START,
    STOP,
    PAUSE,
    DEFAULT_CMD
};

union BaseCommand switch (CommandKind) {
case START:
    uint32 delay;
case STOP:
    uint32 distance;
default:
    string comment;
};

// Valid derived union
union ExtendedCommand : BaseCommand {
case DEFAULT_CMD:
    uint32 default_timeout; // OK: matches default case in base type
case GO_LEFT:
```

```
    float angle;            // OK: New member with unused discriminator
default:
    string notes;           // OK: Derived union defines its own default
};

// Ill-formed derived union
union BadExtendedCommand : BaseCommand {
case START:
    uint32 new_delay;   // NOT OK: discriminator value already used in base
case GO_LEFT:
    uint32 distance;    // NOT OK: member name collides with base's default:
};
```

7.2.3.2.4.4 Member Attributes and Metadata

Member Name

Each member of an Aggregated type is identified by a string name. The name shall be unique within the scope of the Aggregated type and therefore may be used to uniquely identify the member.

7.2.3.2.4.4.2 Member Type

Each member of an Aggregated type has an associated member type.

7.2.3.2.4.4.3 Member Index

Each member has an associated integer index. The member index indicates the declaration order of the member within the Aggregated type. The member index may impact certain Data Representations and Language Bindings. For example, certain Data Representations serialize member in order according to the member index.

Member indices shall start at zero and increment by one with the highest index being equal to the number of members minus one.

7.2.3.2.4.4.4 Member IDs

Each member of an aggregated type is uniquely identified within that type by an integer "member ID." Member IDs are unsigned and have a range that can be represented in 28 bits: from zero to 268,435,455 (0x0FFFFFFF). (The full range of a 32-bit unsigned integer is *not* used in order to allow binary Data Representations the freedom to embed a small amount of meta-data into a single 32-bit field if they so desire.)

Member Name Hashes

As specified in 7.2.3.2.4.4.1, each member of an aggregated type has a name. Some parts of this specification require computing 4-byte hashes of member names.

Unless otherwise noted, the 4-byte hashes computed from strings shall consist of the first 4 bytes of the MD5 hash of the UTF-8 encoding of the string, without any 'nul' terminator.

For example, computing the 4-byte Hash of the member name "color" shall result on the 4 octets {0x70, 0xDD, 0xA5, 0xDF}.

7.2.3.2.4.4.6 Must-Understand Members

A consumer of data may not have the same definition for a type as did the producer of that data. Such a situation may come about as a result of the independent, decoupled definition of the respective types or as a result of a single type's evolution over time. A consumer, upon observing a member value it does not understand, must be able to determine whether it is acceptable to ignore the member and continue processing other members, or whether the entire data sample must be discarded.

Each member of an aggregated type has a Boolean attribute "must-understand" that satisfies this requirement. If the attribute is true, a data consumer, upon identifying a member it does not recognize, must discard the entire data sample to which the member belongs. If the attribute is false, the consumer is permitted to process the sample, omitting the value of the unrecognized member.

In a structure type, each member may have the "must-understand" attribute set to true or false independently.

The default setting of the "must-understand" attribute for structure members is **false**. The default setting may be changed explicitly designating the members for which the "must-understand" attribute shall be set to **true**. The specific syntax depends on the Type Representation used, see 7.3.1 and. 7.3.2 for the syntax when using the IDL and XML representations.

In a union type, the discriminator member shall always have the "must-understand" attribute set to **true**. Non-discriminator members shall have the "must-understand" attribute set to **false**.

The ability of a consumer to detect the presence of an unrecognized member depends on the Data Representation. Each representation shall therefore define the means by which such detection occurs.

7.2.3.2.4.4.7 Optional Members

Each member of an aggregated type has a Boolean attribute that indicates whether it is *optional*. Every object of a given type shall be considered to contain a value for every non-optional member defined by that type. In the event that no explicit value for such a member is ever provided in a Data Representation of that object, that member is considered to nevertheless have the default "zero" value defined in Table 10 below.

Table 10 – Default values for non-optional members

Type Kind	*Default Value*
`BYTE`	0x00
`BOOLEAN`	False

INT_8_TYPE, UINT_8_TYPE, INT_16_TYPE, UINT_16_TYPE, INT_32_TYPE, UINT_32_TYPE, INT_64_TYPE, UINT_64_TYPE, BITMASK_TYPE, FLOAT_32_TYPE, FLOAT_64_TYPE, FLOAT_128_TYPE	0
CHAR_8_TYPE, CHAR_16_TYPE	'\0'
STRING8_TYPE STRING16_TYPE	""
ARRAY_TYPE	*An array of the same dimensions and same element type whose elements take the default value for their corresponding type.*
ALIAS_TYPE	*The default type of the alias's base type.*
SEQUENCE_TYPE	*A zero-length sequence of the same element type.*
MAP_TYPE	*An empty map of the same element type.*
ENUM_TYPE	*The first literal in the enumeration.*
UNION_TYPE	*A union with the discriminator set to the default value for the discriminator type. If this selects a branch, then the selected member is also set to the default value for the member type. Otherwise, the value of the union is fully specified by the discriminator value.*
STRUCTURE_TYPE	*A structure without any of the optional members and with other members set to their default values based on their corresponding types.*

An object may omit a value for any optional member(s) defined by its type. Omitting a value is semantically similar to assigning a null value to a pointer in a programming language: it indicates that no value exists or is relevant. Implementations shall *not* provide a default value in such a case.

The Union discriminator shall have the "optional" attribute set to **false**.

In a structure type, each member may have the “optional” attribute set to true or false independently. The same applies to each member of Union types except for the discriminator.

The default setting of the “optional” attribute for structure and (non-discriminator) union members is **false**. The default setting may be changed explicitly designating the members for which the “optional” attribute shall be set to **true**. The specific syntax depends on the Type Representation used, see 7.3.1 and 7.3.2 for the syntax when using the IDL and XML representations.

The value of a member’s “optional” attribute is unrelated to the value of its “must understand” attribute. For example, it is legal to define a type in which a non-optional member can be safely skipped or one in which an optional member, if present and not understood, must lead to the entire sample being discarded.

7.2.3.2.4.4.8 Key Members

Each member of an aggregated type has a Boolean attribute “is key” indicating that the member is part of that type’s *key*. The presence of one or more members with “is key” set to **true** will determine whether the aggregated Type is considered “Keyed or “Unkeyed”, see 7.2.3.2.6.

The members that have “is_key” attribute set to **true** are called the “Key Members” of the containing type.

The default setting of the “is key” attribute for all members is **false**. The default setting may be changed explicitly designating the members for which the “is key” attribute shall be set to **true**. The specific syntax depends on the Type Representation used, see 7.3.1 and. 7.3.2 for the syntax when using the IDL and XML representations.

Members having the “is key” attribute set to **true** shall also satisfy the following constraints:

- Their “must understand” attribute shall be set to **true** (see 7.2.3.2.4.4.6).
- Their “optional” attribute shall be set to **false** (see 7.2.3.2.4.4.7).

Members of type Map, as well as members that are an Array or Sequence of Map, an Alias to Map, or any recursive combination of these types, **shall not** be designated as key members and their “is key” attribute shall be set to **false**.

The reason to exclude Map types is that maps are not ordered making their comparison and consistent hash computation a very expensive operation.

For structure types, any number of members may have the “is key” attribute set to true, subject to the constraints described above regarding member types and member attributes.

For union types, only the discriminator member can have the “is key” attribute set to true.

If a member of type Array or Sequence has its “is key” attribute set to **true** in an aggregated type T, all elements of the array or sequence shall be considered part of the key of T.

For members of type Sequence, the length of the sequence shall also considered part of the key ahead of the sequence elements.

If a member of an aggregated type T has type K, and K is a Structure or Union Type, then:

- If K is a Keyed Type (see 7.2.3.2.6), only the members of K with “is key” set to **true** shall be considered part of the key of T.
- If K is an Unkeyed Type (see 7.2.3.2.6), all members of K shall be considered part of the key of T, except for the members of type Map, Array or Sequence of Map, Alias to Map, or any recursive combination of these types. Members of K for which the “optional” attribute is set to **true** or the “must understand” is set to **false** shall still be considered part of T’s key.

This relationship is recursive: the key members of *K* may themselves have nested key members.

For example, suppose the key of a medical record is a structure describing the individual whose record it is. Suppose also that the nested structure (the one describing the individual) has a key member that is the social security number of that individual. The key of the medical record is therefore the social security number of the person whose medical record it is.

7.2.3.2.4.4.9 Non Serialized Members

The `Non Serialized` attribute identifies members of an `Aggregated Type` that are not included in the serialized representation of the data objects of that type.

When this attribute is set to true, the member:

- Shall be ignored during serialization and deserialization.
- Shall be ignored for type assignability checks.
- Shall not have the `Key` attribute set to **true**.

Members with `Non Serialized` set to **true** are typically used to model transient or computed values that do not appear on the wire.

If not explicitly specified in the Type Representation, the default value of this attribute is false.

Note — A member that has attributes `Non Serialized` and `Key` both set to **true** is invalid, since a key member must always be serialized.

7.2.3.2.4.4.10 Member Default Value

The type system supports optional specification of the default value associated with members of Aggregated types.

The default value association may be direct or indirect:

- Direct association: the default value is specified in the declaration of the member.
- Indirect association: the default value is inherited from the member type.

To support this, each member of an aggregated type has a “default_value” attribute that contains the default value directly associated with the member, if any.

The concrete expression of direct default value association depends on the Type Representation (see 7.3). Using the IDL type representation (7.3.1), member default values are specified with the annotation `@default`.

Example:

```
struct Config {
    @default(10)  int32 mode;     // direct default value
};
```

If a member does not directly associate a default value, then it is indirectly associated with the default value of its type, if one exists.

```
typedef @default(100) float Altitude;


struct DroneStatus {
    Altitude height; // OK. indirect default association (100)
};
```

If a member associates a default value directly, its type shall not be an Alias type that itself specifies a default value directly, nor one that recursively resolves to such an Alias.

Example:

```
typedef @default(100) float Altitude;


struct DroneStatus2 {
    @default(200) Altitude height; // NOT OK base type 'Altitude'
                                   // already has a direct default value
};
```

Optional members shall not associate a default value directly. When an optional member value is not provided during initialization, it is always initialized as ‘not present.’

If a member does not directly associate a default value, then it is indirectly associated with the default value of its type (if one exists). See Table 10.

7.2.3.2.4.4.11 Member Units

The type system supports optional specification of the units associated with a member of an Aggregated type (see 7.2.2.9). The unit association may be direct or indirect.

- Direct association: units are specified in the declaration of the Member.
- Indirect association: units are inherited from the member’s type.

To support this, each member of an aggregated type has an attribute that contains the units directly associated with the member, if any.

If a member has units directly associated with it, its type shall not be an Alias type that already has associated units (directly or indirectly). Other than this restriction, the member’s type is unrestricted.

The concrete expression of direct unit association depends on the Type Representation (see 7.3). Using the IDL type representation (7.3.1), units are specified with the annotation `@unit`.

Examples:

```
typedef float Elevation;

struct DroneStatus {

    @unit("m/s; meter/sec") float speed;   // direct units

    @unit("meter") Elevation height;       // OK: Elevation has no units

}

typedef @unit("feet") float USElevation;

struct DroneStatus2 {

    @unit("mph") float speed;

    @unit("ft")  USElevation height;  // NOT OK: USElevation

                                      // already has units

}
```

If a member does not directly specify units but its type has associated units, then those units are indirectly associated with the member:

```
typedef @unit("meter") float Elevation;

struct DroneStatus3 {

    Elevation height;   // OK: height indirectly associates

                        // the units from 'Elevation'

}
```

When specified, member units are considered as part of type assignability checks (see 7.2.5.5.9). Two members are assignable only if their associated unit are compatible.

7.2.3.2.4.4.12 Member Range Limits

The type system supports optional specification of the range limits associated with a member of an Aggregated type (see 7.2.2.10). The association may be direct or indirect:

- Direct association: limits are specified in the declaration of the member.
- Indirect association: limits are inherited from the member's type.

To support this, each member of an aggregated type has attributes that contain the minimum and maximum values directly associated with the member, if any.

Range member limits may be specified only if the member type is a numeric primitive type or recursively resolves to a numeric primitive type through intermediate Aliases.

The concrete expression of direct range limits depends on the Type Representation (see 7.3). Using the IDL type representation (7.3.1) range limits are specified with the member annotations: `@range`, `@min`, and `@max`.

Examples:

```
struct Coordinates {
    @range(-90, 90)      float latitude;     // direct range limits
    @min(-180) @max(180) float longitude;    // direct range limits
}
```

If an Aggregated type member has type T1 and T1 has associated range limits, then:

- If the member does not directly specify limits, then T1's limits are indirectly associated with the member.
- If the member directly specifies limits, then they shall be contained within T1's limits, or else it shall be treated as an error.

Examples:

```
typedef @range(0, 360) int32 Angle;
struct Velocity {
    Angle direction;   // Member indirectly associated with
                       // range limis from 'Angle' (0,360)
    float speed;
};
struct Velocity1 {
    @range(0, 180) Angle direction;   // OK: member range (0,180) is
                                      // contained in type 'Angle'
                                      // range (0,360)
    float speed;
};
struct Velocity2 {
    @range(-180, 180) Angle direction; // NOT OK: member range (-180,180) is
                                       // not contained in type 'Angle'
                                       // range (0,360)
    float speed;
};
```

When specified, member range limits are considered as part of type assignability checks (see 7.2.5.5.9). Two types or members are assignable only if their associated limits have a non-empty intersection.

7.2.3.2.5 Inheritance of Aggregated Types

The Type System supports single inheritance of Aggregated Types.

A Structure Type may inherit from another Structure Type, referred to as its `base_type`, as long as the following conditions are met:

- The derived structure either has the extensibility kind unspecified, or else, it has the same extensibility kind as the base type.
    - If the extensibility of the derived structure is unspecified, then it shall be interpreted to have the same extensibility kind as its `base_type`.
- The derived type does not have a member with the same name or with the same memberId as the base type or any of the base structure ancestor types. This rule may be stated differently:
    - Construct a hypothetical structure type containing as members both the members of the derived structure and the members of all ancestor structures.
    - The members of the hypothetical structure must have unique names and member IDs.

If any of the rules above are violated, the definition of the derived structure shall be considered ill-formed.

Derived structures may declare members as being part of the key. However, note that doing so changes the set of members that constitute the derived type's key, and therefore the derived structure shall not be assignable to or from the base structure, as defined in 7.2.5.5.9.

A Union Type may inherit from another Union Type, referred to as its `base_type`, as long as the following conditions are met:

- The derived Union either has the extensibility kind unspecified, or else it has the same extensibility kind as the base type.
    - If the extensibility of the derived union is unspecified, then it shall be interpreted to have the same extensibility kind as its `base_type`.
- The derived Union has the same discriminator as the base Union.
- The base Union, or any of its ancestors have not explicitly defined a "default" member of the Union.
- Any given discriminator value identifies at most one member of the derived Union or any of the base Union ancestor types. This rule may be stated differently:
    - Construct a hypothetical Union Type containing as members both the members of the derived Union and the members of all ancestor Unions.

- The resulting Union must be legal according to the rules in 7.2.3.2.4.3 which require that a discriminator value can select at most one member.

If any of the rules above are violated, the definition of the derived union shall be considered ill-formed.

The derived union may:

- Define additional members corresponding to discriminator values that do not violate the rules above.
- Define members for discriminator values that, in the base type —or recursively, in any base type up the inheritance chain—would select the default member.
- Define its own default member, even if the base type —or recursively, any base type further up the inheritance chain—already defines one.

7.2.3.2.6 Keyed and Unkeyed (Aggregated) Types

If an aggregated type has one of more members with the attribute “is_key” set to **true**, then the Type is called a “Keyed Type”. Otherwise to the type is classified as an “Unkeyed Type”.

Any DDS Topic that is constructed using a Keyed Type as the top-level type will have a key. Conversely DDS Topics constructed using an Unkeyed Types as the top-level will have no key.

7.2.3.2.7 Related Key-Erased type of an Aggregated Type

For each Agregated Type “T” the type system defines a related Aggregated Type KeyErased(T) obtained from “T” by removing the *key* designation from any member that had it (see 7.2.3.2.4.8). In the case the type “T” had no members designated *key* members then KeyErased(T) is the same type “T”.

Example: Assume the following types described in IDL:

```
struct Coordinates {
    float x;
    float y;
    float z;
};

struct Aircraft {
    @key string airline;
    @key int32  flight_number;
    Coordinates current_position;
};
```

```
enum CommandKind {
   START,
   STOP,
   GO_LEFT,
   GO_RIGHT
};

union Command switch (@key CommandKind) {
case START:
   float delay; /* delay until start in seconds */
case STOP:
   float distance; /* distance to stop in meters */
case GO_LEFT:
case GO_RIGHT:
   float angle; /* Angle to change direction in radians */
};
```

The related KeyErased types are:

```
// Types with no key members are unchanged
 type struct Coordinates_KeyErased {
    float x;
    float y;
    float z;
};

struct Aircraft_KeyErased {
    string airline;
    int32  flight_number;
    Coordinates current_position;
};
```

```
// The related KeyErased type definition only applies to aggregated types.
// Is not defined for enum CommandKind.

union Command_TypeErased switch (CommandKind) {
case START:
   float delay; /* delay until start in seconds */
case STOP:
   float distance; /* distance to stop in meters */
case GO_LEFT:
case GO_RIGHT:
   float angle; /* Angle to change direction in radians */
};
```

7.2.3.2.8 Related Key-Holder type of an Aggregated Type

For each Aggregated Type “T” the type system defines a related Aggregated Type KeyHolder(T) obtained from “T” as follows:

- If “T” has any members designated as *key* members (see 7.2.3.2.4.4.8), then KeyHolder(T) removes any members of “T” that are not designated as *key* members.
- If “T” is a structure with no *key* members, then KeyHolder(T) adds a key designator to each member.
- If T” is a union and the discriminator is not marked as key, then KeyHolder(T) is the same type T.

Example: Assume the following types described in IDL:

```
struct Coordinates {
    float x;
    float y;
    float z;
};

struct Aircraft {
    @key string airline;
    @key int32  flight_number;
    Coordinates current_position;
};
```

```
enum CommandKind {
   START,
   STOP,
   GO_LEFT,
   GO_RIGHT
};

union Command switch (@key CommandKind) {
case START:
   float delay; /* delay until start in seconds */
case STOP:
   float distance; /* distance to stop in meters */
case GO_LEFT:
case GO_RIGHT:
   float angle; /* Angle to change direction in radians */
};
```

The related KeyHolder types are:

```
// Coordinates had no key members so every member is now designated as key
 type struct Coordinates_KeyHolder {
    @key float x;
    @key float y;
    @key float z;
};

// Aircraft had key members so non-key members are removed
struct Aircraft_KeyHolder {
    @key string airline;
    @key int32  flight_number;
};
```

```
// The related KeyHolder type definition only applies to aggregated types.
// Is not defined for enum CommandKind.


// Aircraft had key members so non-key members are removed
union Command_KeyHolder switch (@key CommandKind) {
};
```

7.2.3.3 Annotation Definition

An annotation definition is used to declare a new kind of annotation and describe the structured metadata used to instantiate the annotation when it is applied (see 7.2.2.12).

Annotation definitions consist of an annotation name and an optional list of parameters.

Each annotation parameter consists of:

- a name,
- a type, and
- an optional default value.

.

The types permitted for an AnnotationParameter are:

- Primitive types (see 7.2.5.5.4)
- String types (7.2.5.5.5)
- Enumerated types (7.2.3.2.1)

Limiting the permitted parameter types allows the compiler of a Type Representation to be able to efficiently interpret an annotation instantiation; it also simplifies expressing the parameter values as object literals in a variety of Type Representations.

Note - Specific Type Representations (for example, IDL or XML, see 7.3) may define built-in annotations to represent the standardized attributes defined in this specification.

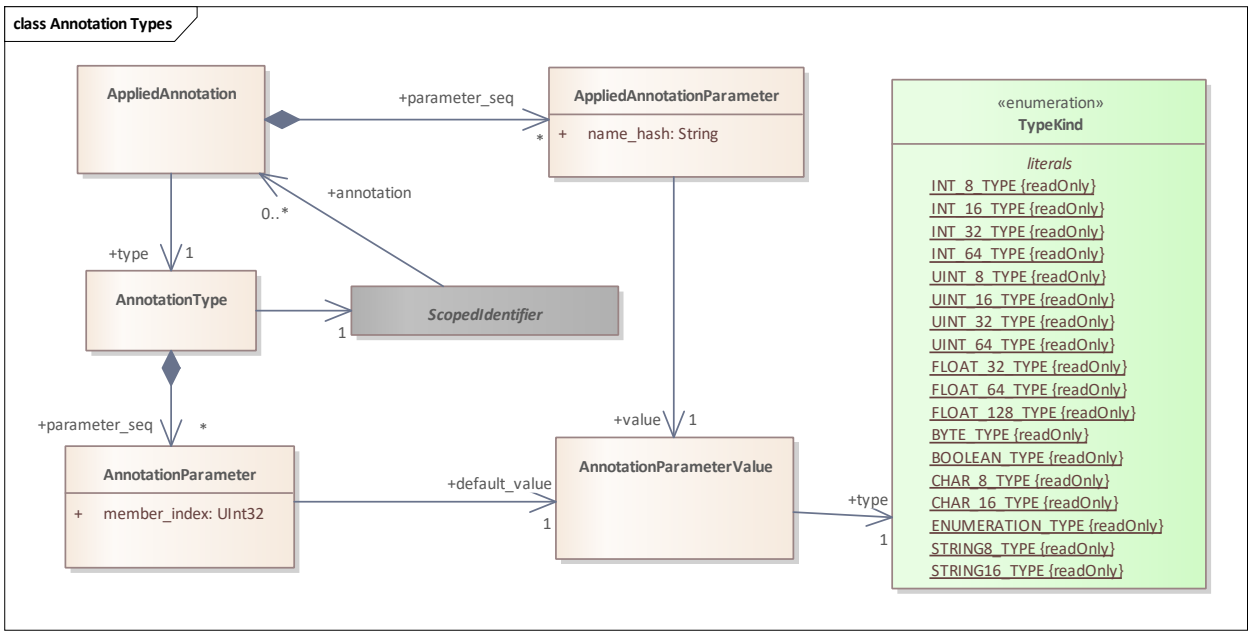

Figure 20 – Annotation Types

7.2.4 Type Extensibility and Mutability

In some cases, it is desirable for types to evolve without breaking interoperability with deployed components already using those types. For example:

- A new set of applications to be integrated into an existing system may want to introduce additional fields into a structure. These new fields can be safely ignored by already deployed applications, but applications that do understand the new fields can benefit from their presence.
- A new set of applications to be integrated into an existing system may want to increase the maximum size of some sequence or string in a Type. Existing applications can receive data samples from these new applications as long as the actual number of elements (or length of the strings) in the received data sample does not exceed what the receiving applications expects. If a received data sample exceeds the limits expected by the receiving application, then the sample can be safely ignored (filtered out) by the receiver.

In order to support use cases such as these, the type system introduces the concept of type extensibility and defines three extensibility kinds: *final*, *appendable* and *mutable*.

A type may be *FINAL*, indicating that the type may not be modified without breaking assignability. In particular, it is not possible to add or remove members from aggregated types while maintaining type assignability.

- A type may be *APPENDABLE*, indicating that two types, where one contains all of the elements/members of the other plus additional elements/members appended to the end, may remain assignable. Note that this was called *EXTENSIBLE* in xtypes version 1.1 and prior.

- A type may be *MUTABLE*, indicating that two types may differ from one another in the additional, removal, and/or transposition of elements/members while remaining assignable.

This attribute may be used by the Data Representations to modify the encoding of the type in order to support its extensibility.

The meaning of these extensibility kinds is formally defined with respect to type compatibility in Clause 7.2.5, "Type Compatibility." It is summarized more generally in Table 11.

Table 11 – Impact of the extensibility kind

Type Kind	*Impact of the extensibility kind*
Aggregated Types: `STRUCTURE_TYPE`, `UNION_TYPE`	Aggregated types may be final, appendable, or mutable on a type-by-type basis. However, the extensibility kind of a structure type with a base type must match that of the base type. It shall not be permitted for a subtype to change the extensibility kind of its base type. Any members marked as keys must be present in all variants of the type.
Collection Types: `ARRAY_TYPE`, `SEQUENCE_TYPE`, `MAP_TYPE`	For these types the extensibility kind has no effect.
Enumerated Types: `ENUMERATION_TYPE`, `BITMASK_TYPE`	These types may be final or appendable on a type-by-type basis.
String Types: `STRING8_TYPE`, `STRING16_TYPE`	For these types the extensibility kind has no effect.
`ALIAS_TYPE`	Since aliases are semantically equivalent to their base types, the extensibility kind of an alias is always equal to that of its base type.
Primitive types	For these types the extensibility kind has no effect.

7.2.5 Type Compatibility

In order to maintain the loose coupling between data producers and consumers, especially as systems change over time, it is desirable that the two be permitted to use slightly different versions of a type, and that the infrastructure perform any necessary translation. To support type evolution and inheritance the type system defines the "is-assignable-from" directed binary relationship between every pair of types in the Type System.

Given two types T1 and T2, we will write:

```
T1  is-assignable-from  T2
```

…if and only T1 is related to T2 by this relationship. The rules to determine whether two types have this relationship are given in the following subclauses.

Intuitively, if T1 *is-assignable-from* T2, it means that in general it is possible, in a structured way, to set the contents of an object of type T1 to the contents of an object of T2 (or perhaps a subset of those contents, as defined below) without leading to incorrect interpretations of that information.

7.2.5.1 Constructing objects of one type from objects of another type

The fact that T1 *is-assignable-from* T2, does not mean that *all* objects of T2 can be used to construct an object of type T1 (for example, a collection may have too many elements).

What the *is-assignable-from* indicates is that the difference between T2 and T1 is such that (*a*) a meaningful subset of T2 objects can construct T1 objects without misinterpretation and that (*b*) the remaining objects of T2—which cannot construct T1 objects—can be detected as such so that misinterpretations can be prevented. For the sake of run-time efficiency, these per-object "can-construct" rules are designed such that their enforcement does not require any inspection of a data producer's type definition. Per-object enforcement can potentially be avoided altogether—depending on the implementation—by declaring a type to be *final*[2], forcing producer and consumer types to match exactly; see Clause 7.2.4.

In the case T1 *is-assignable-from* T2 but an object O2 of type T2 is encountered that cannot construct any object of type T1, the default behavior is to discard the O2 object to avoid misinterpretation. This behavior can be altered when the object O2 is a member of an Aggregated type (e.g. a structure). In this case the behavior is determined by the `TryConstruct` behavior specified for the member. See Clause 7.2.2.8.

Therefore, for each pair of types T1 and T2 this specification defines the rules for T1 to be assignable-from T2. Assuming T1 is-assignable-from T2 the specification also defines which objects of type T2 can be used to construct an object of type T1.

For example:

Table 12 – Type assignability example

T1	***T2***	***Type compatibility***	***Object construction***
Sequence of 10 integers	Sequence of 5 integers	**T1 is assignable from T2:** All objects of type T2 can be used to initialize T1 objects. **T2 is assignable from T1:**	Any object O2 of type T2 can construct an object of type T1. Only T1 objects containing at most 5

[2] DDS-based systems have an additional tool to enforce stricter static type consistency enforcement: the `TypeConsistencyEnforcementQosPolicy`. See Clause 7.6.3.3.

		For any object O1 of type T1 it can either be used to construct some object of type T2 or it can be determined reliably that there is no T2 object that can be initialized from O1.	elements can construct T2 objects.

7.2.5.2 Concept of Delimited Types

Delimited types are those types “T” whose serialized object representation is such that the receivers of an object of that type “T” who only know a type T1 assignable-from type “T” are able to reliably delimit the object within the serialized representation. This means that where appropriate the receiver may “skip” that object and proceed to process other objects that are serialized after.

Primitive and Enumerated types (Enumeration and Bitmask) are delimited types as their serialized size is fixed.

Strings and wide strings are delimited types because the serialization starts with a size from which it is possible to derive the overall serialized length of the string.

Collection types (arrays, sequences, maps) are delimited if the collection element type is delimited. In the case of a map collection the key type must also be delimited. Otherwise the collection is not delimited. The reason is that the receiver of a compatible collection type always knows the length of the collection: Either it is encoded in the serialized representation (sequences and maps) or it is the same as the receiver type in the case of arrays.

Other than the types mentioned above all other types with extensibility kind FINAL **are not** delimited.

Types with extensibility kind APPENDABLE **are** delimited if serialized with encoding version 2 (DELIMITED_CDR). See Clause 7.4.2. They **are not** delimited if serialized with encoding version 1.

Mutable types are also delimited with both encoding version 1 and encoding version 2.

- The serialized representation used for version 1 encoding (PL_CDR) is a list of length-encoded elements ended by a sentinel, which delimits the serialized object. See Clause 7.4.1.2.
- The serialized representation used for version 2 encoding (PL_CDR2) starts with a delimiter header similar to the one used for DELIMITED_CDR, which delimits the serialized object.

7.2.5.3 Strong Assignability

If types T1 and T2 are equivalent using the MINIMAL relation (see Clause 7.3.3.8), or alternatively if T1 is assignable-from T2 and T2 is a delimited type, then T1 is said to be “strongly” assignable from T2.

7.2.5.4 Type Assignability Options

This subclause defines options that influence type assignability checks.

These options are not properties of any type; they are parameters supplied to the assignability evaluation function.

For example, when evaluating the type compatibility between a DDS Data Writer and a DataReader these parameters are configured in the TypeConsistencyEnforcementQosPolicy (see 7.6.3.4).

Assume two types T1 and T2 and the relation T1 assignable-from T2. This relation applies, for example, when T2 is the type used by a DataWriter and T1 is the type expected by a DataReader.

The assignability check shall incorporate the following configuration options:

- force_type_validation
- type_consistency_kind
- prevent_type_widening
- ignore_sequence_bounds
- ignore_string_bounds
- ignore_member_names
- ignore_literal_names
- ignore_units
- ignore_unspecified_units
- ignore_ranges

When any “ignore_*” option is TRUE, it overrides the corresponding stricter requirement that would otherwise apply from the type assignability rules defined in 7.2.5.5..

When an option is FALSE, assignability proceeds according to the normal rules of 7.2.5.5, taking all relevant type metadata—such as bounds, names, ranges, and units—into account.

7.2.5.4.1 Option force_type_validation

The force_type_validation option controls whether full type validation is required in order to determine type assignability.

If force_type_validation is set to TRUE, the assignability algorithm shall perform the checks specified in 7.2.5.5 to determine type assignability. In this case, the type information required to perform those checks shall be available; otherwise, the types shall be considered not assignable.

If force_type_validation is set to FALSE (the default), the assignability algorithm may omit the checks specified in 7.2.5.5 if the type information for one of the types involved is not available. In this case, it is permitted to consider matching type names as a criterion for assignability.

7.2.5.4.2 Option type_consistency_kind

The type_consistency_kind option determines the overall type assignability mode to be applied during evaluation. Its possible values are:

- DISALLOW_TYPE_COERCION
- ALLOW_TYPE_COERCION

If type_consistency_kind is DISALLOW_TYPE_COERCION, the assignability check shall fail unless T1 and T2 are equivalent under the MINIMAL TypeObject equivalence relation (see 7.3.3.8),

If type_consistency_kind is ALLOW_TYPE_COERCION, lack of MINIMAL equivalence does not by itself cause assignability to fail.

In this case, assignability shall be determined according to the detailed rules defined in 7.2.5.5. taking into account the other configuration options in this subclause.

At the DDS level, if coercion is allowed, compatible but non-equivalent types may communicate provided that all other applicable assignability rules are satisfied.

7.2.5.4.3 Option prevent_type_widening

If this option is TRUE, the check shall fail if T2 contains any non-optional members that are not present in T1.

If this option is FALSE (default), the presence of additional non-optional members in T2 that are not in T1 shall not, by itself, cause assignability to fail.

At the DDS level, this option controls whether a DataReader is permitted to accept data written with a DataWriter type that extends it by adding members.

7.2.5.4.4 Option ignore_sequence_bounds

If this option is TRUE, differences in declared maximum lengths (bound) for sequence or map types shall be ignored during assignability checks.

If this option is FALSE (default), assignability between a sequence or map type T1 and T2 requires that the bound of T1 is greater than or equal to that of T2.

At the DDS level, ignoring sequence bounds allows Readers and Writers to communicate even if they use different maximum sequences (or maps) lengths, at the risk of truncation or data loss as as specified by the TryConstruct behavior (see 7.2.2.8).

7.2.5.4.5 Option ignore_string_bounds

If this option is TRUE, differences in declared maximum lengths (bounds) for string and wstring types shall be ignored during assignability checks.

If this option is FALSE (default), assignability between a string type T1 and T2 requires that T1’s bound is greater than or equal to T2’s bound.

ignoring string bounds allows Readers and Writers to communicate even if they use different maximum string lengths, at the risk of truncation, or data loss as specifief by the TryConstruct behavior (see 7.2.2.8).

7.2.5.4.6 Option ignore_member_names

If this option is TRUE, the assignability of aggregated types (see 7.2.3.2.4) shall be determined without considering member names. In this case, matching of members may rely only on member-ID values (see 7.2.3.2.4.4.4 and 7.2.5.5.).

If this option is FALSE (default), the names of members shall be compared according to the rules in 7.2.5.5 involving member name checks. Mismatched names cause assignability to fail.

7.2.5.4.7 Option ignore_literal_names

If this option is TRUE, the assignability of enumerated types shall be determined without considering literal names.

If this option is FALSE (default), the names of literals shall be compared according to the rules in 7.2.5.5 involving literal name checks. Mismatched names cause assignability to fail.

7.2.5.4.8 Option ignore_units

If this option is TRUE, differences in unit metadata (see 7.2.2.9) shall be ignored during assignability checks.

If this option is FALSE (default), assignability between two members requires that any declared units are compatible as defined in 7.2.2.9.

At the DDS level, ignoring units allows components using otherwise assignable types but different unit metadata to communicate, at the cost of disabling the extra layer of safety provided by the unit-compatibility checking.

7.2.5.4.9 Option ignore_unspecified_units

The ignore_unspecified_units option further refines the behavior when ignore_units is TRUE and one of the types being compared omits unit information.

If this option is TRUE (default), a type whose unit is unspecified shall be considered compatible with any other type, even if the other specifies a concrete unit.

In effect, the absence of a unit is treated as equivalent to specifying "__ANY__".

If this option is FALSE, a type that does not specify units shall be compatible only with another type that also omits units or explicitly specifies "__ANY__" (see 7.2.2.9.).

This option has no effect when ignore_units is FALSE.

7.2.5.4.10 Option ignore_ranges

If this option is TRUE, value-range metadata (see 7.2.2.10) shall be ignored during assignability checks.

If this option is FALSE (default), assignability between two members requires that their declared ranges are compatible as defined in 7.2.2.10.

At the DDS level, ignoring ranges allows components using otherwise assignable types but different ranges annotations to communicate, at the cost of partial data loss as specifief by the TryConstruct behavior (see 7.2.2.8).

7.2.5.4.11 Option use_xtypes_1_3_union_assignability

Assignability Rules

The assignability relations defined in this clause are evaluated under the evaluation options described in 7.2.5.4 (Type Assignability Options), which determine how certain aspects of type metadata—such as bounds, names, units, and ranges—impact assignability.

7.2.5.5.1 Assignability of Equivalent Types

If two types T1 and T2 are equivalent according to the MINIMAL relation (see Clause 7.3.3.8), then they are mutually assignable, that is, T1 is-assignable-from T2 and T2 is-assignable-from T1.

The reverse is not always true. The type system contains mutually assignable types that are not equivalent according to the MINIMAL relation.

7.2.5.5.2 Non-serialized Members

Members that are marked as non-serialized, see Sub Clause 7.3.1.3.1.18, shall be ignored during type compatibility checking.

7.2.5.5.3 Alias Types

Table 13 – Definition of the *is-assignable-from* relationship for alias types

T1 Type Kind	*T2 Type Kinds for which T1 is-assignable-from T2 Is True*	*Object construction*
Any `ALIAS_TYPE`	*Any* `ALIAS_TYPE` T2 where: • The units associated with T1 and T2 are compatible (see 7.2.2.9). • The range limits associated with T1 and T2 are compatible (see 7.2.2.10). • T1.base_type *is-assignable-from* T2.base_type. *Any non* `ALIAS_TYPE` T2 where T1.base_type *is-assignable-from* T2.	Construct according to the rules for constructing T1.base_type objects from T2.base_type objects. Construct according to the rules for constructing T1.base_type objects from T2 objects.

Any non `ALIAS_TYPE`	*Any* `ALIAS_TYPE` T2 where: • T1 *is-assignable-from* T2.base_type.	Construct T1 objects according to the rules for constructing T1 from objects of type T2.base_type.

7.2.5.5.4 Primitive Types

Table 14 below defines the *is-assignable-from* relationship for Primitive Types. These conversions are designed to preserve the data during translation. Furthermore, in order to preserve high performance, they are designed to enable the preservation of *data representation*, such that a `DataReader` is not required to parse incoming samples differently based on the `DataWriter` from which they originate. (For example, although a short integer could be promoted to a long integer without destroying information, a binary Data Representation is likely to use different amounts of space to represent these two data types. If, upon receiving each sample from the network, a `DataReader` does not consult the type definition of the `DataWriter` that sent that sample, it would not know how many bytes to read. The runtime expense of this kind of type introspection on the critical path is undesirable.)

Table 14 – Definition of the *is-assignable-from* relationship for primitive types

T1 Type Kind	***T2 Type Kinds for which T1 is-assignable-from T2 Is True***	***Object construction***
Any Primitive Type	The same Primitive Type	Copy the primitive object.
	`ALIAS_TYPE` subject to the conditions in Table 13.	See Table 13.
`UINT8_TYPE`	`BITMASK_TYPE` if and only if T2.bound is between 1 and 8, inclusive.	For each bitflag that is set in the bitmask construct the integer value (1 << position) using the position of that bitflag. Add all those integer values to obtain the resulting object O1 of type T1
`UINT16_TYPE`	`BITMASK_TYPE` if and only if T2.bound is between 9 and 16, inclusive.	
`UINT32_TYPE`	`BITMASK_TYPE` if and only if T2.bound is between 17 and 32, inclusive.	
`UINT64_TYPE`	`BITMASK_TYPE` if and only if T2.bound is between 33 and 64, inclusive.	

7.2.5.5.5 String Types

The *is-assignable-from* relationship for string types is described in Table 15.

Table 15 – Definition of the is-assignable-from relationship for string types

T1 Type Kind	*T2 Type Kinds for which T1 is-assignable-from T2 Is True*	*Object construction (assuming type assignability)*
`STRING8_TYPE`	`STRING8_TYPE` if and only if T1.element_type *is-assignable-from* T2.element_type `ALIAS_TYPE` subject to the conditions in Table 13.	An object O2 of type T2 can-construct an object of type T1 if and only if O2.length <= T1.length Copy each character. O1.length is set to O2.length. If an element of O2 cannot construct T1.element_type (i.e. O2.length > T1.length), then the result depends on the TryConstruct behavior associated with T1 element type. • If DISCARD, O2 cannot construct any object of type T1. • If USE_DEFAULT, O1 is constructed by initializingit to its default value (the empty string). • If TRIM, O1 is constructed by copying as many complete UTF-8 characters from O2 as they can fit within T1.length. Trimming shall be UTF-8-aware so that the resulting string does not end with an incomplete/invalid character.
`STRING16_TYPE`	`STRING16_TYPE` if and only if T1.element_type *is-assignable-from* T2.element_type `ALIAS_TYPE` subject to the conditions in Table 13.	An object O2 of type T2 can-construct an object of type T1 if and only if O2.length <= T1.length Copy each character. O1.length is set to O2.length. If an element of O2 cannot construct T1.element_type (i.e. O2.length > T1.length), then the result depends on

		the TryConstruct behavior associated with T1 element type. • If DISCARD, O2 cannot construct any object of type T1. • If USE_DEFAULT, O1 is constructed by initializing it to its default value (the empty string). • If TRIM, O1 is constructed by copying T1.length characters from characters into O1.

7.2.5.5.5.1 Example: Strings

According to the above rules, any string type of narrow characters is assignable from any other string type of narrow characters. Any string type of wide characters is assignable from any other string type of wide characters. However, string types of narrow characters are not assignable from string types of wide characters, because of the possibility of data misinterpretation. For example, suppose a string of wide characters is encoded using the CDR Representation. If a consumer of strings of narrow characters were to attempt to consume that string, it might consider the first byte of the first character to be a character onto itself, the second byte of the first character to be a second character, and so on. The result would be a string of narrow characters having "junk" contents.

Furthermore, any T2 string *object* containing more characters than the bound of the T1 string type cannot construct any object of type T1 in order to prevent data misinterpretations resulting from truncations. For example, consider two versions of a shopping list application. The list of purchases is represented by a sequence of strings. Version 2.0 of the application increased the bounds of these strings. Supposing that the list items "cat food" and "catsup" were too long to be understood by a version 1.0 consumer, it would be better to come home from the store without either item than to come home with two cats instead.

7.2.5.5.6 Fixed Point Types

The *is-assignable-from* relationship for fixed point types is determined solely by the type's fixed-point parameters. There is no type coercion, narrowing, widening, rounding, or truncation allowed.

Table 16 – Definition of the *is-assignable-from* relationship for fixed point types

T1 Type Kind	*T2 Type Kinds for which T1 is-assignable-from T2 Is True*	*Object construction*
`FIXED_DECIMAL_TYPE`	`FIXED_DECIMAL_TYPE` if and only if: • T2.precision == T1.precision • T2.scale == T1.scale For this type T2.base == T1.base = 10	Copy the object.

7.2.5.5.7 Collection Types

The *is-assignable-from* relationship for collection types is based in part on the same relationship as applied to their element types.

Table 17 – Definition of the *is-assignable-from* relationship for collection types

T1 Type Kind	*T2 Type Kinds for which T1 is-assignable-from T2 Is True*	*Object construction (assuming type assignability)*
`ARRAY_TYPE`	`ARRAY_TYPE` if and only if[3]: • T1.bounds[] == T2.bounds[] • T1.element_type is strongly assignable from T2.element_type • The units associated with T1.element_type and T2.element_type are compatible (see 7.2.2.9). • The range limits associated with T1.element_type and T2.element_type are compatible (see 7.2.2.10). `ALIAS_TYPE` subject to the conditions in Table 13.	To construct an object of type T1 from an object O2 of type T2: Each element of the T1 array shall be constructed from the corresponding element of the O2 array. If an element of T1 cannot be constructed from the O2 element, the result depends on the TryConstruct behavior associated with T1 element type. • If DISCARD, O2 cannot construct any object of type T1. • If USE_DEFAULT or TRIM, the element is constructed accordingly and the array of type T1 is successfully constructed.

[3] Design rationale: This specification allows sequence, map, and string bounds to change but not array bounds. This is because of the desire to avoid requiring the consultation of per-`DataWriter` type definitions during sample deserialization. Without such consultation, a reader of a compact data representation (such as CDR) will have no way of knowing what the intended bound is. Such is not the case for other collection types, which in CDR are prefixed with their length.

SEQUENCE_TYPE	SEQUENCE_TYPE if and only if: • T1.element_type is strongly assignable from T2.element_type • The units associated with T1.element_type and T2.element_type are compatible (see 7.2.2.9). • The range limits associated with T1.element_type and T2.element_type are compatible (see 7.2.2.10). ALIAS_TYPE subject to the conditions in Table 13.	An object O2 of type T2 can construct T1 if and only if O2.length <= T1.length O1.length is set to O2.length. Construct each in O1 from the corresponding O2 element. If an element of O2 cannot construct T1.element_type, the result depends on the TryConstruct behavior associated with T1 element type. • If DISCARD, O2 cannot construct any object of type T1. • If USE_DEFAULT or TRIM, the element is constructed accordingly and the O1 sequence is successfully constructed.
MAP_TYPE	MAP_TYPE if and only if: • T1.key_element_type is strongly assignable from T2.key_element_type • The range limits and units associated with T1.key_element_type are compatible with the ones associated with T2.key_element_type • T1.element_type is strongly assignable from T2.element_type. • The range limits and units associated with T1.element_type are compatible with the ones associated with T2.element_type	An object O2 of type T2 can construct T1 if and only if O2.length <= T1.length The constructed object O1 shall be as if the O1 map were cleared of all elements and subsequently all T2 map entries were added to it. The entries are not logically ordered. If a key element of O2 cannot construct the corresponding key type of T1 the entire map O2 cannot construct any object of type T1. If a value element of O2 cannot construct T1.element_type, the result depends on the TryConstruct behavior associated with T1 element type. • If DISCARD, O2 cannot construct any object of type T1. • If USE_DEFAULT or TRIM, the element is constructed accordingly and the O1 object is successfully constructed.

	ALIAS_TYPE subject to the conditions in Table 13.	

7.2.5.5.8 Enumerated Types

Conversions of bitmask, and enumerated types are designed to preserve the data during translation.

Table 18 – Definition of the *is-assignable-from* relationship for bitmask and enumerated types

T1 Type Kind	*T2 Type Kinds for which T1 is-assignable-from T2 Is True*	*Object construction*
BITMASK_TYPE	BITMASK_TYPE if and only if T1.bit_bound == T2.bit_bound	Preserve bit values by index for all bits identified in both T1 and T2.
	UINT_32_TYPE if and only if T1.bit_bound is between 17 and 32, inclusive.	
	UINT_16_TYPE if and only if T1.bit_bound is between 9 and 16, inclusive.	
	UINT_64_TYPE if and only if T1.bit_bound is between 33 and 64, inclusive.	
	UINT_8_TYPE if and only if T1.bit_bound is between 1 and 8, inclusive.	
	ALIAS_TYPE subject to the conditions in Table 13.	
ENUMERATION_TYPE	ENUMERATION_TYPE if an only if: • T1.extensibility == T2.extensibility • T1.bit_bound and T2.bit_bound are in the same equivalence set. The equivalence sets are defined as the closed integer ranges 1-8, 9-16, and 17-32. • Any literals that have the same name in T1 and T2 also have the same value, and any literals that have the same value in T1 and T2 also have the same name. This behavior may be modified with the ignore_literal_names option, see 7.2.5.4.7.	Choose the corresponding T1 literal if it exists. If the name or value of the T2 object does not exist in T1, the object cannot construct any object of type T1.

	• If extensibility is final T1 and T2 have the same literals. `ALIAS_TYPE` subject to the conditions in Table 13.	

7.2.5.5.9 Aggregated Types

For aggregated types, *is-assignable-from* is based on the same relationship between the types' members. The correspondence between members in the two types is established based on their respective member IDs and on their respective member names.

Table 19 – Definition of the *is-assignable-from* relationship for aggregated types

T1 Type Kind	*T2 Type Kinds for which T1 is-assignable-from T2 Is True*	*Object construction*
`UNION_TYPE`	`UNION_TYPE` if and only if it is possible to unambiguously select the appropriate T1 member based on the T2 discriminator value and to transform both the discriminator and the selected member correctly. Specifically: • T1.extensibility == T2.extensibility. • T1.discriminator.type *is-strongly-assignable-from* T2.discriminator.type. • Either the discriminators of both T1 and T2 are keys or neither are keys. • If the union is mutable, any members in T1 and T2 that have the same name also have the same ID and any members with the same ID also have the same name. This behavior may be modified with the ignore_member_names option, see 7.2.5.4.6. • If the union is appendable or final, any member m1 in T1 for which there is a member m2 in T2 with the same name verifies that:	A union object O2 of type T2 can construct an object of type T1 if and only if: • Either the value of O2.discriminator can construct the type of T1's discriminator. Or else the discriminator has TryConstruct behavior set to DEFAULT. **AND** • Either the selected member "m2" in O2, if any, can construct the selected member "m1" of T1, if any (where m1 and/or m2 may be the default member). Or else the selected member (if any) has TryConstruct behavior set to DEFAULT or TRIM. Assuming O2 can construct an object of type T1, then: • The constructed object O1 discriminator is constructed from the

	○ Both m1 and m2 have the same ID. That is, m1.ID == m2.ID ○ Or else both m1 and m2 have member IDs that match their corresponding member Index. That is m1.ID == m1.index && m2.ID == m2.index This behavior may be modified with the ignore_member_names option, see 7.2.5.4.5 • For any member m2 in T2 that has m2.ID != m2.index, if there is a member m1 in T1 with the same member ID where m1.ID != m1.index, then m1.type is assignable from m2.type. • For each non-default label in T2 with corresponding member m2 that selects a non-default member m1 in T1, then: ○ m1.name == m2.name. This rule can be modified by the ignore_member_names attribute ○ Either m1.ID == m2.ID or else (m1.ID == m1.index || m2.ID == m2.index). • If the union is appendable or final, every non-default label in T2 that has a member with the must_understand==TRUE, must select a non-default member in T1. • For all non-default labels in T2 that select some member in T1 (including selecting the member in T1's default label), then: ○ the type of the selected member in T1 is assignable from the type of the T2 member.	object O2's discriminator or if that is not possible it is set according to its TryConstruct behavior. If the discriminator value selects a member m2 in O2 (which may be the default value), then: • If the discriminator value also selects a member m1 in O1 (which may be the default value), then m1 is constructed from m2 or if that is not possible it is set according to its TryConstruct behavior. • If the discriminator value does not select any member in O1, then there is no value assigned from m2 (i.e., m2 is "truncated"). If the discriminator value does not select any member in O2, then: • If the discriminator value selects a member m1 in O1, then m1 is set to its default value according to its type. • If the discriminator value does not select any member in T1, then there is nothing else to assign or set on T1.

	○ the units and range limits associated with the selected member in T1 are compatible with those associated with the T2 member (see 7.2.2.9 and 7.2.2.10). • If any non-default labels in T1 that select the default member in T2, then: ○ the type of the member in T1 is assignable from the type of the T2 default member. ○ the units and range limits associated with the member in T1 are compatible with those associated with the T2 default member (see 7.2.2.9 and 7.2.2.10). • If T1 and T2 both have default labels, then: ○ the type associated with T1 default member is assignable from the type associated with T2 default member. ○ the units and range limits associated with the default member in T1 are compatible with those associated with T2 default member (see 7.2.2.9 and 7.2.2.10). • If T1 (and therefore T2) extensibility is final then the set of labels is identical. Otherwise, they have at least one common label other than the default label.	
`STRUCTURE_TYPE`	`STRUCTURE_TYPE` if and only if: • T1 and T2 have the same extensibility kind.	Each member “m1” of the T1 object takes the value of the T2 member with the same ID

	• Any members in T1 and T2 that have the same name also have the same ID and any members with the same ID also have the same name. This behavior may be modified with the ignore_member_names option, see 7.2.5.4.6. • There is at least one member “m1” of T1 and one corresponding member “m2” of T2 such that m1.id == m2.id. • For any member “m2” in T2, if there is a member "m1" in T1 with the same member ID, then: o the type KeyErased(m1.type) is-assignable-from the type KeyErased(m2.type). o the member units and range limits associated with m2 in T2 are compatible with those associated with m1 in T1 (see 7.2.2.9 and 7.2.2.10). • If the types are non-mutable, then members of T2 for which must_understand is true also appear in T1 (i.e., have a corresponding member of the same member ID). • If the types are mutable, then members of T2 for which both must_understand is true and optional is false also appear in T1 (i.e., have a corresponding member of the same member ID). • Members marked as key in either T1 or T2 appear (i.e., have a corresponding member of the same member ID) in both T1 and T2. • For any string key member m2 in T2, the m1 member of T1 with the	or name, if such a member exists. Each non-optional member in a T1 object that is not present in the T2 object takes its default value. Each optional member in a T1 object that is not present in the T2 object takes no value. If a “must understand” member in the T2 object is present, then T1 must have a member with the same member ID. Otherwise the object cannot construct T1. This behavior is not affected by the TryConstruct setting. If a member cannot construct the corresponding member in T1, then the behavior is determined by the TryConstruct setting of the member.

	same member ID verifies m1.type.length >= m2.type.length. • For any enumerated key member m2 in T2, the m1 member of T1 with the same member ID verifies that all literals in m2.type appear as literals in m1.type. • For any sequence or map key member m2 in T2, the m1 member of T1 with the same member ID verifies m1.type.length >= m2.type.length. • For any structure or union key member m2 in T2, the m1 member of T1 with the same member ID verifies that KeyHolder(m1.type) is-assignable-from KeyHolder(m2.type). • For any union key member m2 in T2, the m1 member of T1 with the same member ID verifies that: For every discriminator value of m2.type that selects a member m22 in m2.type, the discriminator value selects a member m11 in m1.type that verifies KeyHolder(m11.type) is-assignable-from KeyHolder(m22.type). Note: The rules regarding key members ensure that the key of T2 can be transformed faithfully into the key of T1 without aliasing or loss of information. AND if T1 is appendable, then members with the same member_index have the same member ID, the same setting for the 'optional' attribute and the T1 member type is strongly assignable from the T2 member type. AND if T1 is final, then they meet the same condition as for T1 being	

	appendable and in addition T1 and T2 have the same set of member IDs. For the purposes of the above conditions, members belonging to base types of T1 or T2 shall be considered “expanded” inside T1 or T2 respectively, as if they had been directly defined as part of the sub-type.	

7.2.5.5.9.1 Example: Union case reordering

Consider the following non-mutable union types where the cases have been re-ordered. Reordering of the cases changes the auto-assigned memberIds as shown in the comments.

```
union A_U11 switch (int32) {
  case 1:   int32 x; // auto-assigned memberId = 1
  case 2:   int64 y; // auto-assigned memberId = 2
};
union A_U12 switch (int32) {
  case 2:   int64 y; // auto-assigned memberId = 1
  case 1:   int32 x; // auto-assigned memberId = 2
};
```

A_U11 **is assignable** from A_U12 despite having members with the same name that don’t have the same memberId. This is because the unions are **appendable** (default extensibility) and the memberIDs match the memberIndex which is treated a a special case in the assignability rules.

Consider the following union types, similar to the previous ones, except. They are mutable.

```
@mutable
union M_U11 switch (int32) {
  case 1:   int32 x; // auto-assigned memberId = 1
  case 2:   int64 y; // auto-assigned memberId = 2
};
@mutable
union M_U12 switch (int32) {
  case 2:   int64 y; // auto-assigned memberId = 1
  case 1:   int32 x; // auto-assigned memberId = 2
};
```

M_U11 **is not assignable** from M_U12 because they have members with the same name that don't have the same memberId.

Consider the following union types, similar to the previous ones, except. They are mutable using @autoid(HASH).

```
@mutable @autoid(HASH)
union M_U11 switch (int32) {
  case 1:   int32 x; // auto-assigned memberId = hashid("x")
  case 2:   int64 y; // auto-assigned memberId = hashid("y")
};
@mutable @autoid(HASH)
union M_U12 switch (int32) {
  case 2:   int64 y; // auto-assigned memberId = hashid("y")
  case 1:   int32 x; // auto-assigned memberId = hashid("x")
};
```

M_U11 **is assignable** from M_U12 because members with the same name have the same memberId. The use of @autoid(HASH) causes the memberId to be derived from the name of the member so the union is robust to reordering the cases.

7.2.5.5.9.2 Example: Union incompatibility due to case redefinition

Consider the following union types where a case has changed the name and ID of the member.

```
union U11 switch (int32) {
case 1:
   @id(10) int32 x;
case 2:
   @id(20) int64 y;
};
union U22 switch (int32) {
case 1:
   @id(10) int32 x;
case 2:
   @id(30) int64 weight;
};
```

U11 **is not assignable** from U12 because they the case label '2' selects members with different names and/or IDs.

Consider the following union types where a case has changed the ID of the member.

```
union U11 switch (int32) {
case 1:
   @id(10) int32 x;
case 2:
   @id(20) int64 y;
};
union U22 switch (int32) {
case 1:
   @id(10) int32 x;
case 2:
   @id(30) int64 y;
};
```

U11 **is not assignable** from U12 because they the case label ‘2’ selects members with different IDs.

7.2.5.5.9.3 Example: Union incompatibility due to default case type assignability

Consider the union types U1 and three variations U21, U22, and U23 defined in the IDL below:

```
union U1 switch (int32) {
    case 1:   @id(10) int32 x;
    case 2:   @id(20) int64 y;
    default:  string extra;
};
union U21 switch (int32) {
    case 1:   @id(10) int32 x;
    case 2:   @id(20) int64 y;
    default:  int64 z;
};
union U22 switch (int32) {
    case 1:   @id(10) int32 x;
    case 2:   @id(20) int64 y;
    case 3:   @id(50) int64 z;
};
union U23 switch (int32) {
case 1:
```

```
    @id(10) int32 x;
    default:   @id(50) string z;
};
```

Union U1 is not assignable from U21 because the types of the default members are not assignable.

Union U1 is not assignable from U22 because U22 discriminator value=3 has a member of type `int64` and selects the default member of U1 which has type `string`.

Union U1 is not assignable from U23 because U23 default case has a member of type `string` and some discriminator values, that select U23's default member select U1 members with types not assignable from type `string`. For example, U23's discriminator value=2 selects U1's member "y" with type `int64`.

7.2.5.5.9.4 Example: Type Truncation

Consider the following type for representing two-dimensional Cartesian coordinates:

```
struct Coordinate2D {
   long x;
   long y;
};
```

(This example uses the IDL Type Representation. However, the same principles apply to any other type representation.)

Now suppose that another subsystem is to be integrated. That subsystem is capable of representing three-dimensional coordinates:

```
struct Coordinate3D {
   long x;
   long y;
   long z;
};
```

(The type `Coordinate3D` may represent a new version of the `Coordinate2D` type, or the two coordinate types may have been developed concurrently and independently. In either case, the same rules apply.)

`Coordinate2D` is assignable from `Coordinate3D`, because that subset of `Coordinate3D` that is meaningful to consumers of `Coordinate2D` can be extracted unambiguously. In this case, consumers of `Coordinate2D` will observe the two-dimensional projection of a `Coordinate3D`: they will observe the `x` and `y` members and ignore the `z` member.

7.2.5.5.9.5 Example: Type Inheritance

Type inheritance is a special case of type truncation, which allows objects of subtypes to be substituted in place of objects of supertypes in the conventional object-oriented fashion.

Consider the following type hierarchy:

```
<struct name="Vehicle">
   <member name="km_per_hour" type="int32"/>
</struct>
<struct name="LandVehicle" baseType="Vehicle">
   <member name="num_wheels" type="int32"/>
</struct>
```

(This example uses the XML Type Representation. However, the same principles apply to any other type representation.)

`LandVehicle` is assignable from `Vehicle`. Any consumer of the latter that receives an instance of the former will observe the value of the member `km_per_hour` and ignore the member `num_wheels`.

7.2.5.5.9.6 Example: Type Refactoring

As systems evolve, it is sometimes desirable to refactor data from place in a type hierarchy to another place. For example, consider the following representation of a giraffe:

```
struct Animal {
   long body_length;
   long num_legs;
};

struct Giraffe : Animal {
   long neck_length;
};
```

(This example uses the IDL Type Representation. However, the same principles apply to any other type representation.)

Now suppose that a later version of the system needs to model snakes in addition to giraffes. Snakes are also animals, but they don't have legs. We could just say that they have zero legs, but then should we add `num_scales` to `Animal` and set that to zero for giraffes? It would be better to refactor the model to capture the fact that legs are irrelevant to snakes:

```
struct Animal {
   long body_length;
};
```

```
struct Mammal : Animal {
   long num_legs;
};

struct Giraffe : Mammal {
   long neck_length;
};

struct Snake : Animal {
   long num_scales;
};
```

Because the is-assignable-from relationship is evaluated as if all member definitions were flattened into the types under evaluation, both versions of the `Giraffe` type are assignable to one another. Producers of one can communicate seamlessly with consumers of the other and correctly observe values for all fields.

7.3 Type Representation

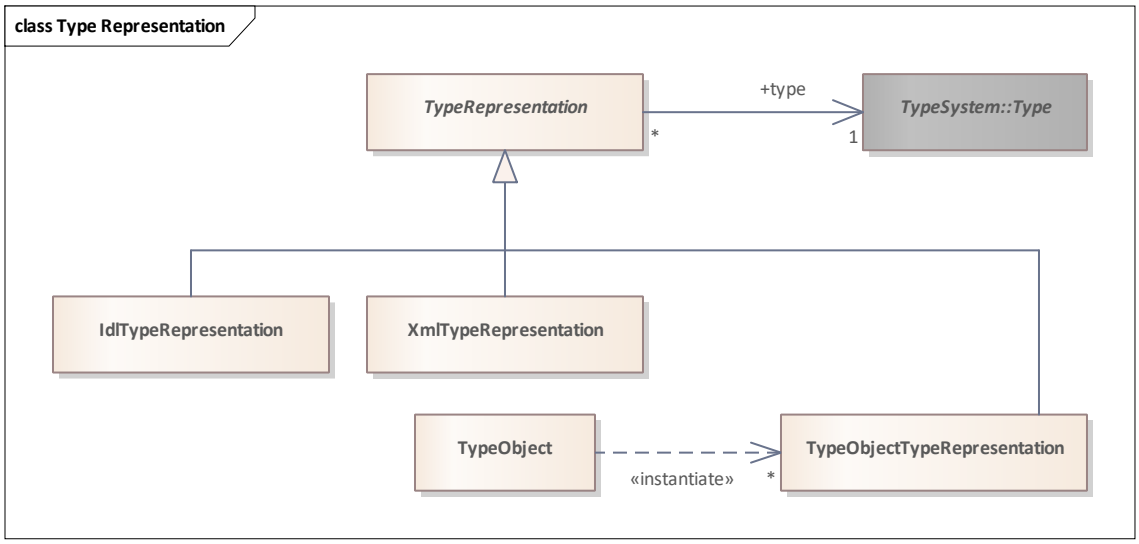

Figure 21 – Type Representation

The Type Representation module specifies the ways in which a type can be externalized so that it may be stored in a file or communicated over the network. Type Representations serve multiple purposes such as:

- Allow a user to describe and document the data type.

- Provide an input to tools that generate code and language-specific constructs to program and manipulate objects of that type.
- Provide an input to tools that want to "parse" and interpret data objects dynamically, without compile-time knowledge of the schema.
- Communicate data types via network messages so that applications can dynamically discover each other's types or evaluate whether relationships such as *is-assignable-from* are true or false.

This specification introduces multiple equivalent Type Representations. The reason for defining multiple type representations is that each of these is better suited or optimized for a particular purpose. These representations are all equivalent because they describe the same Type System. Consequently, other than convenience or performance, there is no particular reason to use one versus the other.

The alternative representations are summarized in Table 20.

Table 20 – Alternative Type Representations

Type Representation	*Reasons for using it*	*Disadvantages*
IDL	Compact Language. Easy to read and write by humans. Familiar to programmers. Uses constructs close to those in programming languages. Has standard language bindings to most programming languages.	Perceived as a legacy language by users who prefer XML-based languages. Not as many tools available (parsers, transformations, syntax-aware editors) as XML-languages. Parsing is complex. Requires extensions to support all concepts in the Type System, e.g., keys, optional members, map types, and member IDs.
TypeObject	Can provide most compact binary representation. Best suited for communication over a network or as an internal representation of a type.	Not human readable or writable.
XML	Compact XML language. Easy to read and write by humans. Defined to precisely fit the Type System so all concepts (including	New language. Based on XML but with a schema that is previously unknown to users.

	keys, optional member, etc.) map well. Syntax can be described using XSD allowing the use of editors that assist and verify the syntax of the type. Well-suited for run-time processing due to availability of packages that parse XML.	

7.3.1 IDL Type Representation

The type system defined by this specification is designed to allow types to be easily represented using IDL [IDL] with minimal extensions.

7.3.1.1 IDL Compatibility

This specification considers two aspects of IDL compatibility:

- *Backward compatibility with respect to type definitions*: Existing IDL type definitions for use with DDS remain compatible to the extent that those definitions were standards-compliant and based on implementation-independent best practices.
- *Forward compatibility with respect to IDL compilers*: With a few exceptions, IDL type definitions formulated according to this specification will be accepted by IDL compilers that do not conform to this specification.

7.3.1.1.1 Backward Compatibility with Respect to Type Definitions

This specification uses a subset of the IDL type definition syntax defined in [IDL]. In particular, it uses the Extensible DDS Profile (Sub Clause 9.3.2 [IDL]), which is composed of the following elements:

- Building Blocks
    - Core Data Types (Sub Clause 7.4.1 [IDL])
    - Extended Data Types (Sub Clause 7.4.13 [IDL])
    - Anonymous Types (Sub Clause 7.4.14 [IDL])
    - Annotations (Sub Clause 7.4.15 [IDL])
- Group of Annotations
    - General Purpose (Sub Clause 8.3.1 [IDL])
    - Data Modeling (Sub Clause 8.3.2 [IDL])
    - Data Implementation (Sub Clause 8.3.4 [IDL])
    - Code Generation (Sub Clause 8.3.5 [IDL]).

This specification retains well-established IDL type definition syntax, such as enumeration, structure, union, and sequence definitions.

Some DDS users may be using constructs for implementation-specific purposes outside the building blocks and group of annotations listed above. These constructs remain legal for use in IDL files provided to IDL compilers compliant with this specification. However, their meanings are undefined with respect to this specification. Compilers that do not support them shall ignore them or issue a warning rather than halting with an error.

7.3.1.1.2 Forward Compatibility with Respect to Compilers

This specification retains well-established IDL type definition syntax, such as enumeration, structure, union, and sequence definitions. This degree of backward compatibility also provides forward compatibility with respect to IDL compilers.

However, this specification also defines new Type System concepts that necessarily had no defined IDL representation, such as maps and annotations. In some cases, such as with annotations, a syntax exists that does not harm compatibility; in other cases, incompatibility is unavoidable.

The following pragma declarations allow IDL type designers to indicate to their tools and to human readers that their IDL file (or a portion of it) makes use of constructs defined by this specification:

```
#pragma dds_xtopics begin [<version_number>]

// IDL definitions

#pragma dds_xtopics end [<version_number>]
```

The optional version number indicates the OMG version number of this specification document. It shall be interpreted without respect to case, and any spaces (for example, in “1.0 Beta 1”) shall be replaced with underscores.

In the event that such `pragma` declarations are nested within one another, the innermost version number specified, if any, shall be in effect. If version numbers are used with “end” declarations, those version numbers should be the same as those used with the matching “begin” declarations.

In the event that such a `pragma` “begin” declaration is not matched with a subsequent closing “end” declaration, the “begin” declaration shall be considered to continue until the end of the IDL input.

For example:

```
#pragma dds_xtopics begin 1.0_Beta_1


struct Base {

   @key long id;

};
```

```
#pragma dds_xtopics begin 1.1


struct Sub : Base {
   long another_member;
};


#pragma dds_xtopics end 1.1
#pragma dds_xtopics end 1.0_Beta_1
```

The above declarations are informative only. The behavior of an IDL compiler upon encountering them is unspecified but may include:

- Silently ignoring them.
- Issuing a warning, perhaps because it does not recognize them, or because it recognizes the pragmas but not the indicated version number.
- Halting with an error, perhaps because it recognized the `pragma`s and knows that it is not compliant with this specification, or because it detected a version mismatch between matching “begin” and “end” declarations.

7.3.1.2 Representing Attributes and Metadata

The attributes and metadata defined in Clause 7.2 are represented in IDL using a combination of IDL language constructs and annotations.

Some properties — such as type names, member names, and type kinds — are represented directly by the intrinsic syntax of the IDL language and therefore do not require annotations.

Other standardized DDS-XTypes attributes and metadata, such as key, optional, id, unit, default, and range, are represented using a defined set of IDL built-in annotations. These annotations provide the syntactic means to express properties of types and members defined abstractly in Clauses 7.2.2 (Attributes and Metadata) and 7.2.3.2.4.4 (Member Attributes and Metadata).

Additional, non-standardized metadata may be represented using user-defined annotations, which extend the same mechanism but have no predefined semantics in this specification.

The built-in annotations used for standardized DDS-XTypes attributes, and the rules governing their use, are described in 7.3.1.3.1 (Built-in Annotations) and 7.3.1.3.2 (Using Built-in Annotations).

7.3.1.3 Annotation Language

This specification makes use of different standard annotation groups defined in [IDL]. It also proposes an alternative annotation syntax for pre-existing IDL compilers.

7.3.1.3.1 Built-in Annotations

This specification uses the IDL built-in annotations described in the following subclauses to model certain attributes and metadata defined in Clause 7.2.

In IDL an annotation may be applied to any construct or subconstruct (see [IDL] clause 7.4.15.2). This specification restricts the applicability of annotations to the types and elements identified in 7.2.2.12 (Annotation Application).

7.3.1.3.1.1 Member IDs

As specified in 7.2.3.2.4.4.4, members of aggregated types have a member ID that uniquely identifies them within their defining type.

Using the IDL Type Representation, Member IDs may be explicitly or automatically assigned. To assign the member ID explicitly the member shall have either the `@id` or the `@hashid` annotation. At most one of these two annotations may appear on a member. Otherwise, it shall be considered an error.

- The `@id` annotation is defined in sub clause 8.3.1.1 in [IDL]). It shall assign the member ID to the value specified in the annotation parameter.
- The `@hashid` annotation is defined below. It shall assign the member ID to a value derived from the member name or a string specified in the annotation parameter.

The `@hashid` annotation is defined as:

```
@annotation hashid {
    string value default "";
};
```

The `@hashid` annotation is useful when one type is using the `@autoid(HASH)` annotation and a new version of the type changes a member's name. The parameter to the `@hashid` annotation can be set to the old member's name, resulting in both members getting assigned the same value for their IDs. If the annotation is used without any parameter or with the empty string as a parameter, then the Member ID shall be computed from the member name.

The explicit assignment of member IDs takes precedence over the automatic mechanisms. If there is no explicit member ID specified, member IDs are automatically assigned either sequentially or as a hash of the member name. The decision is controlled by the `@autoid` annotation defined in sub clause 8.3.1.2 of [IDL]. If no `@autoid` annotation has been specified for the type or any of the containing modules, then the default shall be to use a sequential approach (i.e., the same as using the annotation `@autoid(SEQUENTIAL)`).

The value of the member ID assigned to a member using the sequential approach shall equal to the member ID of the previous member of the same Aggregated Type plus one, subject to the following considerations:

- In the case an aggregated type inherits from another aggregated type the first member of the derived type gets the member ID of the last member in the base type plus one. In

other words, it shall behave as if the base type members appeared at the beginning of the derived type.

- In the case of the first member of Aggregated Type that does not have any base type the member ID shall be set to zero.

Example 1. Consider the following types:

```
@autoid(SEQUENTIAL)
struct MyStruct1 {
    long  s1_m1;
    float s1_m2;
};
@autoid(SEQUENTIAL)
struct MyStruct2 : MyStruct1 {
    string  s2_m1;
};
@autoid(SEQUENTIAL)
struct MyStruct3 : MyStruct2 {
    long  s3_m1;
};
```

In this case the automatic member IDs would be: ID(s1_m1)=0, ID(s1_m2)=1, ID(s2_m1)=2, ID(s3_m1)=3.

By default, member IDs are set automatically following a progression that starts from the most-recently specified ID (using the `@id` annotation defined in Sub Clause 8.3.1.2 in [IDL]) or an implicit value of zero for the first constant—if there is no previous specified value—adding one with each successive member.

This behavior may be altered by two additional annotations. The `@autoid` annotation (defined in Sub Clause 8.3.1.2 in [IDL]), which if set to HASH indicates that all member IDs shall be computed with a hashing algorithm, regardless of the order in which they are declared. And the `@hashid` member annotation, which provides the value to hash to generate the member ID; its definition is as follows:

```
@annotation hashid {
    string value default "";
};
```

The `@hashid` annotation is useful when one type is using the `@autoid` annotation and a new version of the type changes a member's name. The value for this annotation can be set to the old member's name, resulting in both members getting assigned the same hash value for their IDs.

If the annotation is used without any parameter or with the empty string as a value, then the Member ID shall be the hash of the member name.

The computation of a Member ID from a string value, whether the string is the member name (e.g. when using `@autoid(HASH)` or `@hashid` without parameters), or an explicit string parameter (e.g. when using `@hashid` with a string parameter), shall use the following algorithm:

1. Compute a 4-byte hash of the string as specified in 7.2.3.2.4.4.5.
2. Interpret the resulting 4-byte has as a Little Endian unsigned 32-bit integer.
3. Perform a bitwise AND operation with the integer 0x0FFFFFFF to zero the most significant 4-bits of the integer.

Example 2. Consider the following type:

```
@autoid(HASH)

struct MyStruct1 {

    string  color;

    @hashid("shapesize") long size;

};
```

In this case the member IDs would be: ID(color) = 0x0FA5DD70, ID(size) = 0x047790DA.

This is because the 4-byte hash of the member name "color" is {0x70, 0xDD, 0xA5, 0xDF}. When interpreted as a little-endian integer this results in the integer 0xDFA5DD70. Setting the four most significant bits to zero results in the unsigned 32-bit integer 0x0FA5DD70.

Likewise, the 4-byte hash of "shapesize" which is the string parameter specified to @hashid is {0xDA, 0x90, 0x77, 0x14}. When interpreted as a little-endian integer this results in the integer 0x147790DA. Setting the four most significant bits to zero results in the unsigned 32-bit integer 0x047790DA.

7.3.1.3.1.2 Union member IDs

Unions are Aggregated types so their members have an associated member ID which can be automatically or manually assigned via the annotations @autoid, @id, and @hashid, see 7.3.1.3.1.1.

In addition, union member IDs auto-assignment can be controlled with the @union_autoid annotation defined in the IDL below.

```
@annotation union_autoid {

    enum AutoidKind {

        SEQUENTIAL,

        HASH

    };
```

```
    enum UnionSelection {
        ALL_UNIONS,
        MUTABLE_UNIONS
    };

    AutoidKind          value      default HASH;
    UnionSelection      selection  default MUTABLE_UNIONS;
};
```

The `@union_autoid` annotation may be applied to a module or to a union type. Its use overrides the value of the `@autoid` annotation within the application scope.

7.3.1.3.1.3 Optional Members

By default, a member declared in IDL is not optional (see 7.2.3.2.4.4.7). To declare a member optional, users shall apply the `@optional` annotation, which is defined in Sub Clause 8.3.1.3 of [IDL].

It is an error to declare the same member as both optional and as a key.

7.3.1.3.1.4 Key Members

By default, members declared in IDL are not considered part of their containing type's key. To declare a member as part of the key, users shall apply the `@key` or `@key(true)` annotation defined in Sub Clause 8.3.2.1 of [IDL].

It is an error to declare the same member as both optional and as a key.

7.3.1.3.1.5 External Data

By default, members declared in IDL are not considered external data (see 7.2.2.11). To declare a member as external data, users shall apply `@external` or `@external(TRUE)` annotation defined in in Sub Clause 8.3.2.4 of [IDL].

Example:

```
struct T1 {
    @external long my_aggregation_member;
};
```

To declare the elements of a collection type external, apply the annotation to the collection declaration like this:

Sequences:

```
struct T1 {
    sequence<@external Foo, 42> sequence_of_foo;
};
```

Arrays:

```
struct T1 {
    Foo array_of_foo @external [42];
};
```

Maps:

```
struct T1 {
    map<string, @external Foo, 42> map_of_string_to_foo;
};
```

7.3.1.3.1.6 Enumerated Literal Values

For Enumerated types declared using IDL, the value for each enumeration literal is derived from the order in which they appear (see 7.2.3.2.1.3.2). To declare a different value associatwd with a litaral users shall apply `@value` annotation defined in in Sub Clause 8.3.1.5 of [IDL].

Example:

```
enum Weekday {
    MONDAY,    // defaults to 0. See 7.2.3.2.1.3.2
    @value(10)
    TUESDAY,   // explicitly set to 10
    WEDNESDAY, // will be 11, see 7.2.3.2.1.3.2
    THURSDAY,  // will be 12, see 7.2.3.2.1.3.2
    @value(5)
    FRIDAY,    // explicitly set to 5
    SATURDAY,  // will be 6, see 7.2.3.2.1.3.2
    SUNDAY     // will be 7, see 7.2.3.2.1.3.2
};
```

7.3.1.3.1.7 Enumeration and Bistmask Bit Bound

The default bit-size for IDL-defined Enumerated type is 32, see 7.2.3.2.1.3.1. To declare a different bit bound users shall apply `@bit_bound` annotation defined in in Sub Clause 8.3.4.1 of [IDL].

Example:

```
@bit_bound(8) // use only 8 bits to hold the values
enum Weekday {
    MONDAY,
    TUESDAY,
```

```
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};
```

7.3.1.3.1.8 Bitmask Flag Positions

For Bitmask types declared using IDL, the position of each flag is derived from the order in which they appear (see 7.2.3.2.1.3.4). To declare a different position associated with a flag users shall apply `@position` annotation defined in Sub Clause 8.3.1.4 of [IDL].

Example:

```
bistmask Permisions {
    READ_BY_OWNER,    // defaults to (1 << 0) - see 7.2.3.2.1.3.4
    WRITE_BY_OWNER,   // defaults to (1 << 1) - see 7.2.3.2.1.3.4
    EXECUTE_BY_OWNER, // defaults to (1 << 2) - see 7.2.3.2.1.3.4
    @position(20)
    READ_BY_GROUP,     // explicitly set to (1 << 20)
    WRITE_BY_GROUP,    // defaults to (1 << 21) - see 7.2.3.2.1.3.4
    EXECUTE_BY_GROUP,  // defaults to (1 << 22) - see 7.2.3.2.1.3.4
    @position(10)
    READ_BY_OTHER,     // explicitly set to (1 << 10)
    WRITE_BY_OTHER,    // defaults to (1 << 11) - see 7.2.3.2.1.3.4
    EXECUTE_BY_OTHER,  // defaults to (1 << 12) - see 7.2.3.2.1.3.4
};
```

7.3.1.3.1.9 Nested Types

By default, aggregated types and aliases to aggregated types defined in IDL are not considered to be nested types (see 7.2.2.4) and the remaining types are considered nested. This designation may be changed by applying the IDL `@nested` annotation to a type definition, or the `@default_nested` annotation to the enclosing module.

The `@nested` annotation is defined in Sub Clause 8.3.4.3 of [IDL].

The `@default_nested` annotation is defined below:

```
@annotation default_nested {
    boolean value default TRUE;
};
```

The `@default_nested` annotation may be applied to modules. If not present on a module, the value defaults to that of the enclosing module. If a top-level module is not annotated, the default is FALSE.

In addition to the above annotations, IDL compilers shall provide the means to change the default value for non-annotated top-level modules.

7.3.1.3.1.10 Type Extensibility and Mutability

The extensibility kind of a type (see 7.2.4) may be defined by means of a `@extensibility` annotation defined in Sub Clause 8.3.1.6 of [IDL].

In the event that the representation of a given type does not indicate the type's extensibility kind, the type shall be considered appendable. Implementations may provide a mechanism to override this default behavior; for example, IDL compilers may provide configuration options to allow users to specify whether types of unspecified extensibility are to be considered final, appendable, or mutable.

IDL compilers shall also implement the shortcut annotations for the different extensibility kinds. That is, `@final`, `@appendable`, and `@mutable`, which defined in Sub Clauses 8.3.1.7, 8.3.1.8, and 8.3.1.9 of [IDL].

7.3.1.3.1.11 Default value for a member of an Aggregated type

Normally, the default value of a member of an Aggregated type (see 7.2.2.7) is predefined based on the member's type (see 7.2.3.2.4.4.10 and Table 10). For example, a member of type `Int32` has its *default_value* attribute set to zero.

The `@default` annotation, defined in Subclause 8.3 of [IDL], allows the value of the *default_value* to be overridden.

7.3.1.3.1.12 Must Understand Members

Normally, the Must Understand attribute of a member of an IDL-defined Aggregated type (see 7.2.3.2.4.4.6) is set to FALSE. This behavior value may be changed by applying the `@must_understand` or `@must_understand(TRUE)` annotation to a member within a type definition.

The `@must_understand` annotation is defined in Sub Clause 8.3.2.2 of [IDL].

7.3.1.3.1.13 Default Literal for Enumeration

By default, a Enumerated type declared in IDL uses the first literal in the Enimaration as the Default Literal (see 7.2.3.2.1.3.3). To declare explicitly the Default Literal, users shall apply the `@default_literal` annotation.

The `@default_ literal` annotation, is defined in Subclause 8.3.2.3 of [IDL].

Example:

```
enum MyEnum {

    ENUM1,
```

```
    ENUM2,
    @default_literal ENUM3,
    ENUM4
};
```

TryConstruct Elements and Members

By default, a `TryConstruct` behavior of and element of a collection type and/or member of an aggregated type defined in IDL is set to `DISCARD`. (see 7.2.2.8). This behavior value may be changed by applying the `@try_construct` annotation to a member or element in the IDL type definition.

The IDL definition of the `@try_construct` annotation is:

```
enum TryConstructFailAction {
   DISCARD,
   USE_DEFAULT,
   TRIM
};
@annotation try_construct {
   TryConstructFailAction value default USE_DEFAULT;
};
```

As specified in 7.2.2.8 the default behavior is `DISCARD`. Therefore if the `@try_construct` annotation is not used it is the same as if it had been explicitly set to `DISCARD`. For example:

```
struct T1 {
   long important_member;
   @try_construct(DISCARD) string<4> m1;
};
```

Is the same as:

```
struct T1 {
   long important_member;
   string<4> m1;
};
```

If the annotation is specified without a value, or if the value is set to `USE_DEFAULT`, then the behavior is set to `USE_DEFAULT` as specified in 7.2.2.8.

Examples:

```
struct T1 {
   long a_long;
   @try_construct(USE_DEFAULT) string<5> member;
};
struct T1 {
   long a_long;
   @try_construct(TRIM) string<5> member;
};
struct T1 {
   long a_long;
   @try_construct(TRIM) sequence<long,4> member;
};

typedef string<5> String5;
// specify try construct behavior of sequence elements
struct T1 {
   long a_long;
   sequence<@try_construct(TRIM) String5, 10> member;
};

enum T1Enum {
   ENUM1,
   @default_literal ENUM2
};

// specify try construct behavior of union discriminator
union T1 switch ( @try_construct T1Enum ) {
   case ENUM1:
      long e1_value;
   case ENUM2:
      long e2_value;
};
```

7.3.1.3.1.15 Verbatim Text

Verbatim Text objects (see 7.2.2.5) associated with a constructed type declaration shall be indicated using the `@verbatim` annotation defined in Sub Clause 8.3.5.1 of [IDL].

7.3.1.3.1.16 Documentation

Documentation objects associated with a type or type element (see 7.3.1.3.1.16) shall be indicated using the `@doc` annotation defined below.

```
@annotation doc {
  string value  default "";
  string format default "markdown";
};
```

Implementations compliant with this specification shall recognize the following format values: "plain", "markdown", and "html". See 7.2.2.5.

The `@doc` annotation may be repeated on the same element.

Examples:

```
@doc(
  format="html",
  value="<p>Unit quaternion representing a right-handed 3-D rotation. "
        "Components are ordered <code>(x, y, z, w)</code>. "
        "See <a href=\"https://www.iso.org/standard/87028.html\">"
        "ISO/IEC 18026:2025</a>. "
        "The quaternion shall be normalized (||q|| = 1); "
        "q and −q encode the same orientation.</p>"
)
typedef float Quaternion[4]; // [x,y,z,w]


@doc("Geographic coordinates on the WGS 84 ellipsoid.")
@doc(
  format="html",
  value="The <strong>geographic (latitude/longitude)</strong> coordinates "
        "on the WGS 84 ellipsoid.")
struct GPSPosition {
  @doc("Angle of the line normal to the ellipsoid "
       "at the point, in **decimal degrees**.")
```

```
  float latitude;
  @doc(
    format="markdown",
    value="Angle around the ellipsoid's spin axis "
          "from a prime meridian, in **decimal degrees**.")
  string longitude;
  @doc(
    format="text",
    value="Distance above the ellipsoid in meters.")
  string height;
};
```

7.3.1.3.1.17 License information

License information (see 7.3.1.3.1.17) shall be indicated using the following IDL annotations:

```
@annotation license_type {
  string name;  // identifier of the license kind,
                // e.g., "ACMESoftLicense" or "Apache-2.0"

  string link;  // canonical URL where the license text is defined,
              // e.g., "https://www.acme.com/licenses/ACME_CustomLicense.pdf"
};

@annotation license_use {
  string name;               // identifier of this usage,
                             // e.g., "ACMETypes_LicenseTerms"

  string license_type_name;  // shall match license_type.name
                             // of a defined @license_type,
                             // e.g., "ACME_CustomLicense"

  string link;               // URL to the project's LICENSE/NOTICE document
                             // stating that the project is released under
                             // the referenced license_type_name,
```

```
                        // e.g., "https://www.acme.com/ACMETypes/LICENSE.txt"
};

@annotation license {
  string license_use_name;   // shall match license_use.name
                             // of a defined @license_use,
                             // e.g., "ACMETypes_LicenseTerms"
};
```

The annotations `@license_type` and @`license_use` may only be applied to modules.

The reserved module name `"__BuiltinLicenseDefinitions__"` may be used to indicate that the annotations apply to the global scope.

The annotation `@license` may be applied to modules and Named Types (see 7.2.2.2.3).

If `@license` is applied to a module, it shall be interpreted as the default license for that module and all Named Types contained within that module and its nested modules, except for those elements that have a `@license` annotation directly attached to them.

If `@license` is applied to a Named Type, it applies only to that Named Type.

The following is a non-normative example:

```
@license_type(
    name="DDS-Open-Testing",
    link="https://github.com/omg-dds/licenses/blob/master/DDS-Open-Testing.md")

@license_use_name(
    name="DDS-RTPS-License",
    license_type_name="DDS-Open-Testing",
    link="https://github.com/omg-dds/dds-rtps/blob/master/LICENSE.md")

module __BuiltinLicenseDefinitions__ {}

// Shapes Demo Types
@license("DDS-RTPS-License")
module ShapesExample {

    const int32 COLOR_MAX_LEN = 32;
```

```
    @appendable
    @license("DDS-RTPS-License")
    struct ShapeType {
        @key
        string<COLOR_MAX_LEN> color;
        int32 x;
        int32 y;
        int32 shapesize;
    };
};
```

7.3.1.3.1.18 Non-serialized Members

By default, all members declared of ID-defined types have the Non-serialized attribute (see 7.3.1.3.1.18) set to FALSE. To declare that a member should have the attribute set to TRUE users must apply the `@non_serialized` or `@non_serialized(TRUE)` annotation defined below:

```
@annotation non_serialized {
   boolean value default TRUE;
};
```

Example:

```
    struct ShapeType {
        @key
        string<COLOR_MAX_LEN> color;
        int32 x;
        int32 y;
        int32 shapesize;
        @non_serialized float speed;
    };
```

7.3.1.3.1.19 Constrained Data Representations

By default, the Data Representation that may be used by IDL-defined types is not constrained (see 7.3.1.3.1.19). This behavior may be changed using `@data_representation` annotation.

The IDL definition of the `@data_representation` annotation is:

```
// Positions are defined to match the values of the DataRepresentationId_t
// XCDR_DATA_REPRESENTATION, XML_DATA_REPRESENTATION, and
// XCDR2_DATA_REPRESENTATION
```

```
@bit_bound(32)
bitmask DataRepresentationMask {
   @position(0) XCDR1,
   @position(1) XML,
   @position(2) XCDR2
}

@annotation data_representation {
   DataRepresentationMask allowed_kinds;
};
```

The bits set on the allowed_kinds indicate the data representations that are allowed for objects of the annotated data type. For example, objects of the type Foo defined below, can use the Extended CDR DataRepresentation versions 1 and 2 (7.4.1 and 7.4.2) but not the XML DataRepresentation (7.4.4).

```
@data_representation(XCDR | XCDR2)
struct Foo {
   long my_long;
};
```

If the `@data_representation` annotation is not present it is interpreted as if the DataRepresentationMask value was set to 0xffffffff. In this case the representation is not constrained by the type declaration and it may be decided by the code generation or the DataRepresentationQos policy on the DataWriter and DataReader, see 7.6.3.1.

This annotation may be applied to type declarations and module declarations.

7.3.1.3.1.20 Explicit declaration of Topic types

A type may be explicitly annotated to indicate the intent to use it as a top-level DDS Topic. Absent this annotation only non-nested types may be used as DDS Topics. The `@topic` annotation overrides that default behavior.

The IDL definition of the `@topic` annotation is:

```
@annotation topic {
     string name default "";
     string platform default "*";
};
```

The name parameter allows specification of a default topic name associated with the type. The specification does not define how this default is used but a toolchain could take advantage of it in its code generation, e.g. to generate example code.

The defined values for the platform parameter are as follows:

"DDS" indicates that the topic is intended to be used over DDS.

"*" (an asterisk) indicates any platform. This is the default value.

The `@topic` annotation can only be used on types that are legal as DDS Topic types, see 7.6.2.

The `@topic` annotation overrides the `@nested` and `@default_nested` annotations.

7.3.1.3.2 Using Built-in Annotations

The application of the annotations listed above is restricted to the type elements specified in Table 21.

Table 21 – IDL Built-in Annotations Usage

Annotation	*Applicable type elements*
`@must_understand`	Structure Members
`@optional, @id, @hashid, @external, @non_serialized`	Structure Members, Union members (except union discriminator)
`@try_construct`	Structure Members, Union Members (including discriminator), Collection Element
`@key`	Structure Members, Union discriminator
`@bit_bound`	Enumerated Types, Bit Mask Types
`@extensibility, @mutable, @appendable, @final, @nested, @license`	Type declarations, except for Aliases
`@min, @max, @range, @default`	Structure Members, Union Members (including discriminator), Alias Type declarations
`@unit`	Structure Members, Union Members (except discriminator), Alias Type declarations
`@default_literal, @value`	Enumerated Literals
`@position`	Bitmask Values
`@autoid, @license_type, @license_use`	Module declarations, Structure declarations, Union declarations
`@union_autoid`	Module declarations, Union declarations
`@verbatim, @doc`	All elements

7.3.1.3.3 Alternative Annotation Syntax

It is anticipated that it will take vendors some amount of time to implement the syntax defined in [IDL]. During this time, existing customers may have the need to share IDL files between products that do support this specification and those that do not. In such a case, the extended annotation syntax defined here could be problematic. Therefore, this specification defines an alternative syntax for annotations that will not cause problems for pre-existing IDL compilers.

This alternative syntax uses special comments containing at-signs ('@'), much like the way JavaDoc used "at" comments to attach metadata to declarations prior to the introduction of an annotation to the Java language. (For example, the conventional way to deprecate a method prior Java 5 was to place `@deprecated` in the documentation. In Java 5 and above, the preferred way is to use `@deprecated` in the source code itself, but the JavaDoc-based mechanism is still supported.)

As an alternative to prefixing a declaration with an annotation, it is legal to follow the declaration with a single-line comment containing the annotation string. To distinguish such comments from regular comments, there must be no space in between the double slash ("//") and the at-sign ('@'). For example:

```
struct Gadget {

    long my_integer; //@my_member_annotation("Hello")

}; //@my_type_annotation
```

If multiple annotations are to be applied to the same element, the at-sign of each shall be preceded by a double slash and no white space. For example:

```
struct Gadget {

    long my_integer; //@my_annotation1(greeting="Hello")

//@my_annotation2

}; //@my_type_annotation
```

7.3.1.3.4 Defining Annotations

To declare an annotation, users shall use the syntax Sub Clause 7.4.15.4.1 of [IDL]. For example:

```
@annotation MyAnnotation {

   // ...

};
```

Annotation identifiers are orthogonal to any other kind of type and therefore do not conflict with other types that may use the same identifier name even when defined in the same module. This is because the application of an annotation prefixes the annotation identifier with the "@" character, see Sub Clause 7.3.1.3.5.

IDL annotation definitions may specify any number of parameters with optional default values. For example:

```
@annotation MyAnnotation {
   long my_annotation_member_1;
   double my_annotation_member_2 = 3.14; // defaults to specified value
};
```

Annotation parameters have additional constraints that are described in the Type System Model, see 7.2.3.3.

As defined in 7.4.15.4.1 (Defining Annotations) of [IDL], annotation members can take default values; these are expressed by using the keyword "`default`" in between the attribute name and the semicolon, followed by the default value. This value must be a valid IDL literal that is type compatible with the type of the member. For example:

```
@annotation RequestForEnhancement {
   long id;             // identify the RFE
   string synopsis;     // describe the RFE
   string engineer      default "[unassigned]";    // engineer to implement
   string date          default "[unimplemented]"; // date to implement
};
```

The specified default value may be any legal IDL literal compatible with the declared member type.

7.3.1.3.5 Applying Annotations

To apply an annotation, users shall use the syntax Sub Clause 7.4.15.4.2 (Applying Annotations) of [IDL]. For example:

```
struct Delorean {
   Wheel wheels[4];
   float miles_per_gallon;
   @RequestForEnhancement boolean can_travel_through_time;
};
```

IDL syntax permits a member declaration to contain multiple member declarators that share the same type specification (see [IDL] 7.4.15.3 Declarators).

For example:

```
struct ShapeType {
   @key string<32> color;
   int32 x, y; // member declaration with multiple members
   int32 shapesize;
};
```

While this syntax is valid IDL, it is discouraged in the context of DDS-XTypes because it can lead to ambiguity or inconsistent interpretation when annotations are applied to the declaration.

If an annotation is applied to a declaration that contains multiple member declarators sharing the same type, implementations may either:

- treat the declaration as an error, or
- interpret the annotation as applying to each member declarator within that declaration and optionally emit a warning message

The behavior shall be consistent within an implementation.

Example:

```
struct ShapeType {
   @key string<32> color;
   @range(0,255) int32 x, y; // @range annotation applies to both x and y
   int32 shapesize;
};
```

In this example, the `@range` annotation applies to both members x and y, as if each were individually declared with the annotation as illustrated below:

```
struct ShapeType {
   @key string<32> color;
   @range(0,255) int32 x;
   @range(0,255) int32 y;
   int32 shapesize;
};
```

Annotations can be applied to the implicit discriminator member of a union type by applying them to the discriminator type declaration in the header of the union type's definition:

```
union MyUnion switch (@MyAnnotation long) {
   case 0:
      string member_0;
   default:
      long default_member;
};
```

As with any IDL identifier, the name of an annotation and its members are *not* case-sensitive. To specify multiple annotations, place them one after another, separated by white space.

To specify values for any or all of the annotation type's members, follow the name of the annotation with a parenthesis, and place the member values in a comma-delimited list in between

them, where each list item is of the form "*member_name = member_value*." Each value must be a compile-time constant. For example:

```
struct Delorean {
   @RequestForEnhancement(
      id = 10,
      synopsis = "Enable time travel",
      engineer = "Mr. Peabody",
      date = "4/1/3007"
   )
   boolean can_travel_through_time;
};
```

An annotation with an empty list of member values is equivalent to a member list that is omitted altogether.

An annotation with a single value (omitting the parameter name) shall be treated as equivalent to the same annotation with a single named parameter with the name being the first declared parameter and the specified value. For example:

```
struct Delorean {
   @RequestForEnhancement(10)
   boolean can_travel_through_time;
};
```

Shall be treated as equivalent to:

```
struct Delorean {
   @RequestForEnhancement(id = 10)
   boolean can_travel_through_time;
};
```

Any parameter may be omitted when the annotation is applied. If a value for a given parameter is omitted, and that parameter has a defined default value, it will take that value. If an omitted parameter does not have a specified default, it will take the default value specified for its type in Clause 7.2.3.2.4.4.7.

7.3.1.4 Constants and Expressions

IDL allows the declaration of global and namespace-level constant values. It also allows the use of compile-time mathematical expressions, which may include constants, enumeration values, and numeric literals. Such declarations and expressions remain legal IDL. However, they are not reflected directly in the Type System specified here, which assumes that all compile-time-constant values have already been evaluated.

7.3.1.5 Primitive Types

The primitive types specified here directly correlate to the primitive types that already exist in IDL.

Table 22 – IDL primitive type mapping

Type System Model Type	*IDL Type*	*Type System Model Type*	*IDL Type*
`Int8`	`int8`	`Float32`	`float`
`UInt8`	`uint8`	`Float64`	`double`
`Int16`	`short`	`Float128`	`long double`
`UInt16`	`unsigned short`	`Char8`	`char`
`Int32`	`long`	`Char16`	`wchar`
`UInt32`	`unsigned long`	`Boolean`	`boolean`
`Int64`	`long long`	`Byte`	`octet`
`UInt64`	`unsigned long long`		

7.3.1.6 Alias Types

Aliases as described in this specification are fully compatible with the IDL `typedef` construct.

7.3.1.7 Array and Sequence Types

Arrays and sequences as described in this specification are fully compatible with the IDL constructs of the same names.

7.3.1.8 String Types

The string container defined by this specification has two element types for which the behavior is defined: `Char8` and `Char16`. Strings of `Char8` shall be represented by the IDL type `string`. Strings of `Char16` shall be represented by the IDL type `wstring`. In either case, any bound shall be retained.

7.3.1.9 Fixed Point Decimal Types

The fixed point decimal types defined by this specification are fully compatible with the IDL `fixed` construct.

7.3.1.10 Enumerated Types

Enumerations and bitmasks as described in this specification are fully compatible with the IDL constructs of the same name.

7.3.1.11 Map Types

Map types as described in this specification are fully compatible with the IDL constructs of the same name defined in the Extended Data-Types Building Block of [IDL].

7.3.1.12 Structure Types

Structures as described in this specification are fully compatible with the IDL constructs of the same name.

7.3.1.12.1 Specification of must understand members

When using the IDL type representation members are specified as “must understand” annotating the member with `@must_understand` or `@must_understand (true)`.

Members annotated with `@must_understand (false)` shall be treated the same as is it had no `@must_understand` annotation. It does not specify the member as “must understand.”

7.3.1.12.2 Specification of optional members

When using the IDL type representation members are specified as “optional” annotating the member with `@optional` or `@optional(true)`.

Members annotated with `@optional(false)` shall be treated the same as is it had no `@optional` annotation. It does not specify the member as “optional.”

7.3.1.12.3 Specification of key members

In the IDL type representation members are designated as “key” members annotating the member with `@key` or `@key(true)`. These members shall have the “is_key” attribute set to **true** (see 7.2.3.2.4.4.8**).**

Members annotated with `@key(false)` shall be treated the same as is it had no `@key` annotation. These members shall have the “is_key” attribute set to **false** (see 7.2.3.2.4.4.8).

In the IDL example below, the designated key members are: `item_number`, `manufacturer_name` and `manufacturer_country`.

```
struct MyInventoryItem {
    @key uint32 item_number;
    @key(true) string manufacturer_name;
    @key string manufacturer_country;
    @key(false) string expiration_date;
    string manufacture_date;
    int32 price;
    int32 available_units;
};
```

7.3.1.13 Union Types

Unions as described in this specification are fully compatible with the IDL constructs of the same name. Compliant IDL parsers shall implement the Building Block Extended Data-Types of [IDL], which adds support for `Byte` (`octet`) and `Char16` (`wchar`) type discriminators.

7.3.1.13.1 Specification of key discriminator member

As specified in 7.2.3.2.4.4.8**,** for a union type, only the discriminator can be designated as a key member.

In the IDL type representation the discriminator is designated as "key" member annotating the discriminator with `@key` or `@key(true)`. In this case the discriminator member shall have the "is_key" attribute set to **true** (see 7.2.3.2.4.4.8**).**

If the discriminator is annotated with `@key(false)` or it does not have the `@key` annotation then the discriminator member shall have the "is_key" attribute set to **false** (see 7.2.3.2.4.4.8).

Example:

```
enum CommandKind {
   START,
   STOP,
   GO_LEFT,
   GO_RIGHT
};

union MyCommand switch (@key CommandKind) {
  case START:
   float delay; /* delay until start in seconds */
  case STOP:
   float distance; /* distance to stop in meters */
  case GO_LEFT:
  case GO_RIGHT:
   float angle; /* Angle to change direction in radians */
};
```

7.3.1.13.2 Mutable unions should use @autoid(HASH) or set member IDs explicitly

Mutable union assignability rules (see 7.2.5.5.9) require members that have the same name to also have the same ID. By default, unions use `@autoid(SEQUENTIAL)`, which assigns member IDs based on the order of the union members. As a result, reordering members in a mutable union that relies on sequentially assigned member IDs causes the unions to be incompatible. For example:

```
@mutable
union U1 switch(int32) {
  case 1:   int32 x; // auto-assigned memberId = 1
  case 2:   int64 y; // auto-assigned memberId = 2
};
@mutable
union U2 switch(int32) {
  case 2:   int64 y; // auto-assigned memberId = 1
  case 1:   int32 x; // auto-assigned memberId = 2
};
```

Although U1 and U2 define the same cases and members, the automatically assigned member IDs differ due to the change in declaration order, and therefore the two unions are not assignable.

Note that `@autoid(SEQUENTIAL)` is the default setting, so if `@autoid` has not been specified any members that don't use `@id` or `@hashid`, would end up with their IDs assigned sequentially.

For this reason, it is recommended that mutable unions use `@autoid(HASH)`, or `@union_autoid(HASH)`, or else explicitly assign member IDs to all union members (except the default member) using `@id` or `@hashid`.

IDL compiler toolchains are encouraged to issue a warning or error when a mutable union relies on `@autoid(SEQUENTIAL)`.

For backward compatibility, it is recommended that IDL compiler toolchains provide a mechanism (for example, a command-line option) to suppress such warnings or errors.

7.3.2 XML Type Representation

Types may be defined and expressed in an easy-to-read, easy-to-process XML format. This format is defined by the XML schema documents (XSD) referenced in **Annex A: XML Type Representation Schema**.

The XML Type Representation is equivalent, in terms of expressiviness, to the IDL Type representation. A tool may convert between theXML and IDL representations without loss of information.

The sections below provide additional context and detail on the use of XML to represent types in the DDS-XTypes type system.

7.3.2.1 Type Representation Management

The XML Type Representation provides several features that do not directly affect the Type System Model. However, these features support the organization and management of type declarations and are therefore useful for tooling. These features are described in this clause.

7.3.2.1.1 File Inclusion

XML Type Representation documents may include other XML Type Representation documents. Such inclusions shall not be considered semantically meaningful with respect to the Type System Model, they exist solely as a mechanism for organizing and managing type definitions.

The following is a non-normative example:

```
<dds:types xmlns:dds="http://www.omg.org/dds">

   <dds:include file="my_other_types.xml"/>

</dds:types>
```

Conformant XMLType Representation compilers are not required to support inclusion of files expressed using other type representations. For example, a conformant XML Type Representation compiler need not support the inclusion of IDL files from within an XML Type Representation document.

Design Rationale (non-normative)

XML provides other mechanisms to include one document within another, such as entitity definitions. However, these mechanisms are expanded during XML parsing and therefore cannot provide functionality equivalent to the `#include` directive in IDL.

For example, consider a type `X` defined in `X.xml` and a type `Y` defined in `Y.xml` both of which depend on a type `Z` defined in `Z.xml`. If `X.xml` and `Y.xml` include `Z.xml` using XML entity expansion, the XML parser will expand the contents of `Z.xml` directly into both documents. As a result, a code generator will encounter multiple definitions of Z, potentially causing build failures due to multiply defined symbols.

By contrast, the inclusion mechanism defined here allows the code generator to observe the intent to include `Z.xml` and to generate an appropriate language-specific include directive (for example, `#include <Z.h>` in C or C++), thereby avoiding multiple definition problems.

7.3.2.1.2 Forward Declarations

As in IDL, C, and C++, a usage of a type must be preceded by a declaration of that type. Accordingly, the XML Type Representation provides for *forward declarations* of types.

Forward declarations are provided for the convenience of code generator implementations and shall have no representation in the Type Representation Model.

A forward declaration in the XMLType Representation shall be considered semantically equivalent to an IDL forward declaration. A formal definition is in **Annex A: XML Type Representation Schema**.

The following is a non-normative example:

```
<dds:types xmlns:dds="http://www.omg.org/dds">

   <dds:forward_dcl kind="struct" name="MyStructure"/>

</dds:types>
```

7.3.2.1.3 Constants

As in the IDL Type Representation, the XML Type Representation supports declaration of compile-time constant values.

Constants can appear at the top level of a Type Representation file or within a module declaration.

Constants are not reflected directly in the Type System. Instead, mathematical expressions shall be considered to be evaluated at compile time.

The string specified in the `value` attribute described below shall have the same syntax as the **<const_exp>** production in the IDL grammar [IDL].

The following is a non-normative example:

```
<dds:types xmlns:dds="http://www.omg.org/dds">
   <dds:const name="MY_CONSTANT" type="int32" value="2 + 3"/>
</dds:types>
```

7.3.2.2 Representing Attributes and Metadata

The attributes and metadata defined in Clause 7.2 are represented in the XML Type Representation using XML elements and XML attributes defined in **Annex A: XML Type Representation Schema**.

In this representation, all aspects of a type — including its name, structure, members, and associated metadata — are expressed using elements and attributes defined in the schema.

This includes both the intrinsic properties of the type (such as its kind, members, and nesting) and the standardized attributes and metadata (such as key, optional, id, unit, default, range, and documentation).

The choice between representing a property as an XML attribute or as an XML element depends on whether its value is a simple scalar or a longer / more structured construct:

- Simple scalar or literal properties (for example, key, optional, id, unit) are represented as XML attributes.
- Structured or descriptive metadata (for example, documentation, license, or extensibility information) are represented as predefined XML elements nested within the corresponding type or member element.

This representation preserves the semantics of the abstract type model defined in Clause 7.2 while conforming to the syntactic and schema constraints of XML.

The specific XML elements and attributes used to represent each standardized DDS-XTypes property are summarized in the tables below and formally defined in Annex A: XML Type Representation Schema.

Table 23 lists the type-system attributes and metadata that are represented as XML attributes and Table 24 the ones represented as XML elements. In case of an inconsistency between these tables and the XML schema, the schema defined in Annex A: takes precedence.

Table 23 – Attributes and Metadata represented using XML attributes

Type System Attribute	*XML Attribute*	*Applicable XML elements*
Extensibility	extensibility	<struct>, <union>
Nested Type	nested	<struct>, <union>
Default Value	default_value	<member> within <struct>
Try Construct behavior	tryConstruct	<member> within <struct> or <union>
Units	unit	<member> within <struct> or <union> <discriminator> <typedef>
Range Limits	min, max	<member> within <struct> or <union> <discriminator> <typedef>
Key	key	<member> within <struct> <discriminator>
Optional	optional	<member> within <struct> or <union>
Member Id (by value)	id	<member> within <struct>
Member Id	hashid	<member> within <struct>
Member Id	autoid	<module>, <struct>
External Data	external	<member> within <struct> or <union>
Non Serialized	nonSerialized	<member> within <struct> or <union>
Must Understand	mustUnderstand	<member> within <struct> or <union>
Bit Bound	bit_bound	<enum>, <bitmask>
Default Value	default	<member> within <struct> or <union> <typedef>
Default Literal	default_literal	<enumerator>
Literal Value	value	<enumerator>
Flag Position	position	<flag>
License	license	<module> All elements used to define types

Table 24 – Attributes and Metadata represented using XML elements

Type System Attribute	*XML Element*	*Containing XML elements*
`Documentation`	`<doc>`	<module> All elements used to define types All elements used to define type elements
`License Type`	`<license_type>`	<module>
`License Use`	`<license_use>`	<module>
`Verbatim Text`	`<verbatim>`	All Elements

7.3.2.3 Basic Types

This Type Representation represents type names with a combination of XML attributes, defined according to the following pattern:

- A "type" attribute, typed by an enumeration `allTypeKind`, indicates whether the type is "basic" (i.e., is a primitive or string)—and if so, which one—or if it is "non-basic" (i.e., any other type).

  Design rationale: As even basic types have identifier names, the use of the `allTypeKind` enumeration does not add to the expressiveness of this Type Representation. However, since primitive types are used frequently, the enumeration allows XML editors to provide context-sensitive completions, improving the user experience.

- A "non-basic type name" attribute indicates the name of the type if it is a non-basic type. It is an error to include this attribute if the `type` attribute does not indicate a non-basic type.

- If the type is a collection type, additional attributes describe its bound(s); see below.

The names of the basic types in this Type Representation have been chosen to resemble terse versions of the corresponding names in the Type System Model.

Table 25 – Primitive and string type names in the XML Type Representation

Type System Model Name	*XML Type Representation Name*
`Boolean`	`boolean`
`Byte`	`byte`
`Char8`	`char8`
`Char16`	`char16`
`Int32`	`int32`
`UInt32`	`uint32`

Int8	int8
UInt8	uint8
Int16	int16
UInt16	uint16
Int64	int64
UInt64	uint64
Float32	float32
Float64	float64
Float128	float128
String<Char8, …>	string
String<Char16, …>	wstring
FixedPointDecimal	decimal

7.3.2.4 String Types

As described above, strings (whether of narrow or wide characters) are considered to be basic types in this Type Representation. Nevertheless, the description of their bounds requires additional attributes.

The `stringMaxLength` attribute, if present, indicates the string's bound. If the attribute is omitted, the string shall be considered unbounded.

The presence of this attribute is legal only when a member's type is a string, a wide string, or an alias to string or wide string. The following examples are non-normative:

```
<struct name="MyStructure">
   <member name="unbounded_string_1" type="string"/>
   <member name="unbounded_string_2" type="string" stringMaxLength="-1"/>
   <member name="bounded_string" type="string"
           stringMaxLength="2 + MY_CONSTANT"/>
</struct>
```

7.3.2.5 Fixed Point Decimal Types

Fixed Point Decimals are considered to be basic types in this Type Representation. Nevertheless, the description of their precision and scale requires additional attributes.

The `decimalPrecision` attribute, if present, indicates the Fixed Pooint `precision` and `scale`. If the attribute is omitted, the `precision` defaults to 10 and the `scale` defaults to 2.

When present, the `decimalPrecision` attribute shall contain either:

- One integer, interpreted as precision (with scale = 2)
- Two integers separated by a comma, interpreted as precision and scale.

The presence of this attribute is legal only when a member's type is a `decimal`.

The following examples are non-normative:

```
<struct name="MyStructure">
   <!-- precision defaults to 10, scale defaults to 2 -->
   <member name="fixed_decimal_1" type="decimal"/>
   <!-- scale defaults to 2 -->
   <member name="fixed_decimal_2" type="decimal" decimalPrecision="6"/>
   <!-- Both precision and scale are explicit -->
   <member name="fixed_decimal_3" type="decimal" decimalPrecision="12,4"/>
</struct>
```

7.3.2.6 Collection Types

The element type identified by the `type` and `nonBasicTypeName` attributes correspond to the type of a member itself when the member identifies a single value, to the element type when the member is of a sequence or array collection, or to the "value" type of map collection if the member is of a map type. This clause and its sub clauses summarize these rules; the formal grammar can be found in "Annex A: XML Type Representation Schema."

Collection bounds are indicated by attributes named according to the convention `<`*`collection`*`>MaxLength`: `stringMaxLength`, `sequenceMaxLength`, and `mapMaxLength`. The types of these attributes are strings, not integers: the values of these attributes may be any constant expression as defined by the **<const_exp>** production in the IDL grammar [IDL]. The literal expression "-1" shall indicate an unbounded collection; no other "negative" value is permitted.

The `element_external` property of the Type System Model shall be represented by an attribute `external`.

7.3.2.6.1 Array Types

The presence of the `arrayDimensions` attribute shall indicate that given member is an array. Array dimensions are represented as a comma-delimited list of dimension bounds in the same order in which those bounds would be given in IDL. Whitespace is allowed around each bound and is not significant.

Compile-time-constant mathematical expressions are also permitted; their syntax shall be defined by the **<const_exp>** production in the IDL grammar [IDL]. As in the IDL Type Representation, such expressions are not expressed directly in the Type System Model but are evaluated first. For example, the following are all valid:

- `arrayDimensions="1"`

- `arrayDimensions="2, MY_CONSTANT + 3"`
- `arrayDimensions=" 6,2, 3 "`

For example:

```
<struct name="MyStructure">
   <member name="my_array_of_42_integers" type="int32" arrayDimensions="42"/>
</struct>
```

7.3.2.6.2 Sequence Types

The `sequenceMaxLength` attribute, if present, shall indicate that the member is of a sequence type. This attribute is required for all sequence types. In case of an unbounded sequence, sequenceMaxLength shall be set to "-1".

The following is a non-normative example:

```
<struct name="MyStructure">
   <member name="my_unbounded_sequence_of_integers" type="int32"
         sequenceMaxLength="-1"/>
   <member name="my_bounded_sequence_of_structures" type="nonBasic"
         nonBasicTypeName="MyOtherStructure"
         sequenceMaxLength="6 * 3"/>
</struct>
```

7.3.2.6.3 Map Types

Map types must include the following information:

- The mapMaxLength attribute, if present, shall indicate that the member is of a map type. This attribute is required for all map types. In case of an unbounded map, mapMaxLength shall be set to "-1".
- The type of the map's "key" elements shall be indicated by the `mapKeyType` attribute. This attribute is required for all map types. This attribute is exactly parallel to the `type` attribute (which describes the type of the map's "value" elements): it indicates whether the "key" elements of the map are of a basic or non-basic type and, if basic, which basic type. If the type is non-basic, the `mapKeyNonBasicTypeName` attribute is also required and is parallel to the `nonBasicTypeName` attribute. If the "key" type is basic, the `mapKeyNonBasicTypeName` attribute is not allowed.
- Only if the map's "key" type is a string type, the attribute `mapKeyStringMaxLength`, if present, shall indicate the bound of that string type. If the "key" type is a string type, and this attribute is omitted, the string shall be considered unbounded. If the "key" type is not a string type, this attribute is not allowed.

The following is a non-normative example:

```
<struct name="MyStructure">
```

```
   <member name="my_unbounded_maps_of_integers_to_floats" type="float32"
         mapKeyType="int32"
         mapMaxLength="-1"/>
   <member name="my_bounded_map_of_strings_to_structures"
         mapKeyType="string"
         mapKeyStringMaxLength="128"
         type="nonBasic"
         nonBasicTypeName="MyOtherStructure"
         mapMaxLength="6 * 3"/>
</struct>
```

7.3.2.6.4 Combinations of Collection Types

A type may be a sequence of arrays, a map of strings to sequences, or some other complex combination of collection types. It's therefore important to understand, if some *combination* of `sequenceMaxLength` and `mapMaxLength` are present, which takes precedence. The following list is ordered from most-tightly-binding to least-tightly-binding:

- Sequence designations, including `sequenceMaxLength`
- Array designations, including `arrayDimensions`
- Map designations, including `mapMaxLength`.

To indicate a type composed in a different order (for example, a sequence of arrays), it is necessary to interpose an alias definition.

For example, a member specifying all of these would define a map whose values are arrays of sequences of strings. Further examples follow:

```
<struct name="MyStructure">
   <member name="my_array_of_strings"
            type="string"
            stringMaxLength="-1"
            arrayDimensions="20"/>
   <member name="my_array_of_sequences_of_integers"
            type="int32"
            sequenceMaxLength="6 * 3"
            arrayDimensions="20"/>
</struct>
```

7.3.2.7 Aggregated Types

Aggregated types include those types that define internal named members taking per-instance values: annotations, structures, and unions.

The Type System defines a number of properties for aggregated types and their members:

- `extensibility_kind`
- `nested`
- `key`
- `optional`
- `must_understand`, etc.

The IDL Type Representation is based on IDL, which provides no syntax to provide values for these attributes; therefore, that Type Representation makes use of built-in annotations for this purpose. In contrast, the XML Type Definition is able to express these properties directly.

For example, structures and unions may indicate whether they are appendable/mutable and/or nested types:

```
<struct name="MyStructure" extensibility="mutable" nested="true">
  …
</struct>
```

In the event that the representation of a given type does not indicate the type's extensibility kind, an implementation may make its own determination. In particular, type representation compilers shall provide configuration options to allow users to specify whether types of unspecified extensibility will be considered final, appendable, or mutable.

7.3.2.7.1 Structures

Structures contain three kinds of declarations:

- Applied annotations
- Verbatim text
- Members

Applied annotations are described in clause **Error! Reference source not found.**. The other elements are described in the sections below.

7.3.2.7.1.1 Verbatim Text

As described in Clause 7.2.2.6.4, types may store blocks of text to be used by Type Representation compilers. These are represented within a structure's declaration as shown in the following non-normative example:

```
<struct name="MyStructure">
   <verbatim language="Java" placement="before-declaration">
```

```
            /**
             * This is a JavaDoc comment.
             */
   </verbatim>
   ...
</struct>
```

7.3.2.7.1.2 Members

Each structure type shall include one or more members. Each member of a structure type can indicate individually whether or not it is a key member and whether or not it is an optional or must_understand member.

```
<struct name="structMemberDecl">
   <member name="my_key_field" type="int32" key="true"/>
   <member name="my_regular_field" type="int32"
           optional="true" must_understand="true"/>
</struct>
```

7.3.2.7.1.2.1 Specification of must understand members

When using the XML type representation members are specified as “must understand” using the attribute `must_understand="true"` within the member element.

Members with the attribute `must_understand="false"` shall be treated the same as is it had no `must_understand` attribute present. It does not specify the member as “must understand.”

7.3.2.7.1.2.2 Specification of optional members

When using the XML type representation members are specified as “optional” using the attribute `optional="true"` within the member element.

Members with the attribute `optional="false"` shall be treated the same as is it had no `optional` attribute present. It does not specify the member as “optional.”

7.3.2.7.1.2.3 Specification of key members

When using the XML type representation members are specified as “key” using the attribute `key="true"` within the member element. These members shall have the “is_key” attribute set to **true** (see 7.2.3.2.4.4.8**).**

Members with the attribute `key="false"` shall be treated the same as is it had no `key` attribute present. These members shall have the “is_key” attribute set to **false** (see 7.2.3.2.4.4.8).

7.3.2.7.1.3 Inheritance

A structure declaration’s `baseType` attribute indicates the name of the structure’s base type, if any; if it is omitted, then the structure has no base type. For example:

```
<struct name="MyStructure" baseType="MyOtherStructure">
   ...
</struct>
```

7.3.2.7.2 Unions

In addition to the `annotate` and `verbatim` elements they share with other aggregated types (see above), unions contain two kinds of members: exactly one discriminator member (identified by a `discriminator` element) and one or more cases (identified by `case` members). The discriminator member must be declared before the others.

Each case of a union contains one or more discriminator values (`caseDiscriminator` elements) and one data `member`. A case discriminator is a string expression, the syntax of which shall be defined by the **<const_exp>** production in the IDL grammar [IDL]. The literal "default" is also allowed; it indicates that the corresponding case is the default case—there can only be one such within a given union declaration.

For example:

```
<union name="MyUnion">
   <discriminator type="int32"/>
   <case>
      <caseDiscriminator value="1"/>
      <caseDiscriminator value="2"/>
         <member name="small_value" type="float32"/>
   </case>
   <case>
      <caseDiscriminator value="default"/>
      <member name="large_value" type="float64"/>
   </case>
</union>
```

The example above is equivalent to the following IDL type:

```
union MyUnion switch (long) {
   case 1:
   case 2:
      float small_value;
   default:
      double large_value;
};
```

7.3.2.7.2.1.1 Specification of key discriminator member

As specified in 7.2.3.2.4.4.8**,** for a union type, only the discriminator can be designated as a key member.

In the XML type representation, the discriminator is designated as "key" member using the attribute `key="true"` within the discriminator element. In this case the discriminator member shall have the "is_key" attribute set to **true** (see 7.2.3.2.4.4.8**).**

If the discriminator has the attribute with `key="false"` or it does not have the `key` attribute then the discriminator member shall have the "is_key" attribute set to **false** (see 7.2.3.2.4.4.8).

For example:

```
<union name="MyKeyedUnion">
   <discriminator type="int32" key="true"/>
   <case>
      <caseDiscriminator value="1"/>
      <caseDiscriminator value="2"/>
         <member name="small_value" type="float32"/>
   </case>
   <case>
      <caseDiscriminator value="default"/>
      <member name="large_value" type="float64"/>
   </case>
</union>
```

7.3.2.8 Aliases

Alias definitions are defined in `typedef` elements. They have syntax very similar to that of structure members.

For example:

```
<typedef name="MyAliasToSequenceOfStructures"
      type="nonBasic"
      nonBasicTypeName="MyStructure"
      sequenceMaxLength="16"/>
```

7.3.2.9 Enumerated Types

7.3.2.9.1 Enumerations

Enumerations consist of a list of enumeration literals, each of which has a name and a value. The syntax of the value shall be defined by the **<const_exp>** production in the IDL grammar [IDL]. If the value is omitted, it shall be assigned automatically.

For example:

```
<enum name="MyEnumeration" bitBound="16">
   <enumerator name="LITERAL_1" value="0"/>
   <enumerator name="LITERAL_2" value="0+1"/>
   <enumerator name="LITERAL_3"/>
</enum>
```

7.3.2.9.2 Bitmasks

A bitmask type defines a sequence of flags, each of which shall identify one of the bits in the bitmask.

For example:

```
<bitmask name="MyBitmask" bitBound="64">
   <flag name="FIRST_BIT" position="0"/>
   <flag name="SECOND_BIT" position="1"/>
</bitmask>
```

7.3.2.10 Modules

A module groups type declarations and serves as a namespace for those definitions.

```
<module name="MyModule1">
   <struct name="MyStructure">
      <member name="my_member" type="int64"/>
   </struct>
</module>


<module name="MyModule2">
   <struct name="MyStructure">
   <member name="my_member" type="nonBasic"
         nonBasicTypeName="MyModule1::MyStructure"/>
   </struct>
</module>
```

7.3.2.11 Custom Annotations

There are two related XML constructs involved in custom annotations:

(1) Annotation definition. An annotation is defined using the <annotation> element. The declaration lists the annotation's own parameters (name, type, etc.).

(2) Annotation application. An instance of that annotation is then applied to a type or to a type member using the <custom> element nested inside the <annotations> element. The

<custom> element references the defined annotation and supplies values for its parameters.

The following is a non-normative example:

```
<annotation name="MyAnnotation">
   < parameter name="widgets" type="int32"/>
</annotation>

<struct name="MyStructure">
   <annotations>
      <custom annotation_ref="MyAnnotation">
         <parameter name="widgets" value="5"/>
      </custom>
   </annotations>
   ...
</struct>
```

7.3.2.12 Builtin Annotations

Builtin annotations are pre-defined so there is no corresponding `<annotation>` element declaring them in the XML.

Depending on the annotaton, a builtin annotation is applied in one of two ways:

- As an attribute on the XML element that declares the target.
- As a child element inside an `<annotations>` element, using a reserved element name.

For example:

- License information is applied as a `license="..."` attribute on the `<struct>` element.
- Documentation is applied inside `<annotation>` element using a `<doc>` element that may carry attributes (such as `format="markdown"`) and text content.

The following is a non-normative example:

```
<struct name="MyStructure" license="APACHE2.MYPACKAGE">
   <annotations>
      <doc format="markdown">
         Documentation associated with
         *MyStructure* type
```

```
      </doc>
   </annotations>

   <member name="slope" type="int32"
           min="0" max="90" unit="degree">
      <annotations>
         <doc>Indicates the angle to the horizontal</doc>
      </annotations>
   </member>
   ...
</struct>
```

7.3.2.12.1 License Information

LicenseType and LicenseUse are declared using the XML elements <license_type> and <license_use>, respectively.

License information is applied to Named Types and modules using the license attribute in the corresponding element.

The following is a non-normative example:

```
<license_type
    name="DDS-Open-Testing"
    link="https://github.com/omg-dds/licenses/blob/master/DDS-Open-Testing.md" />

<license_use
    name="DDS-RTPS-License"
    license_type_name="DDS-Open-Testing"
    link="https://github.com/omg-dds/dds-rtps/blob/master/LICENSE.md" />

<module name="ShapesExample" license="DDS-RTPS-License">

    <const name="COLOR_MAX_LEN" value="32"
                type="int32"/>

    <struct name="ShapeType" extensibility="append">
        <member name="color" key="true"
            type="string" stringMaxLength="COLOR_MAX_LEN"/>
        <member name="x"
```

```
            type="int32"/>
        <member name="y"
            type="int32"/>
        <member name="shapesize"
            type="int32"/>
    </struct>
</module>
```

7.3.3 Representing Types with TypeIdentifier and TypeObject

Any possible type within the XTYPES type system is uniquely identified by a `TypeIdentifier`. In the case of simple types such as primitive types, string, or certain sequences of primitive types, the `TypeIdentifier` completely describes the Type. For more complex types, the `TypeIdentifier` only identifies the type and its full description uses a `TypeObject`.

See "Annex B: Representing Types with TypeObject" for the formal definition of the `TypeIdentifier` and `TypeObject` types.

7.3.3.1 Plain Types

This specification uses the term **Plain Collection** type to refer to anonymous collection types (array, sequence, and map) that have no annotations beyond `@external` and `@try_construct`.

This specification uses the term **Plain** type to refer to Scalar types (see 7.2.3.1) and plain collection types. The remaining types are called **Non-Plain** types.

Plain types only have a `TypeIdentifier`. Non-plain types have both a `TypeIdentifier` and a `TypeObject`.

7.3.3.2 Type Identifier

The type identifier provides a unique way to identify each type within the XTYPES type system. More precisely it identifies each equivalence class of types, see Clause 7.3.3.7.

The definition of the type identifier uses the structure `TypeIdentifier` declared in IDL; see "Annex B: Representing Types with TypeObject".

`TypeIdentifier` is a discriminated union allowing the format of the identifier to vary depending on the type. Table 21 below lists the `TypeIdentifier` discriminator values and their use.

Table 26 – Formats and interpretation of the TypeIdentifier

TypeIdentifier discriminator value	***Types***	***Notes***
`TK_NONE`	N/A	Invalid identifier
`TK_BOOLEAN, TK_BYTE, TK_INT8, TK_UINT8, TK_INT16, TK_INT32,`	Primitive Types	Plain Type. No TypeObject

TK_INT64, TK_UINT16, TK_UINT32, TK_UINT64, TK_FLOAT32, TK_FLOAT64, TK_FLOAT128, TK_CHAR8, TK_CHAR16		Fully described by the discriminator. No further information in TypeIdentifier.
TI_STRING8_SMALL, TI_STRING8_LARGE	String Types	Plain Type. No TypeObject Fully described by the discriminator and the bound of the string. The SMALL discriminators have a bound represented as an octet. It is used for unbounded strings or strings with bounds smaller than 256. The LARGE discriminators are used for the remaining strings
TI_STRING16_SMALL, TI_STRING16_LARGE	Wide String types	Plain Type. No TypeObject Fully described by the discriminator and the bound of the string. SMALL and LARGE indicate representation of bound.
TI_FIXED_POINT	Fixed Point Types	Plain Type. No TypeObject Fully described by precision, scale, and base.
TI_PLAIN_SEQUENCE_SMALL, TI_PLAIN_SEQUENCE_LARGE	Plain sequence Collection	Plain Type. No TypeObject TypeIdentifier contains maximum length of sequence and the TypeIdentifier of element. SMALL and LARGE indicate representation of maximum length.
TI_PLAIN_ARRAY_SMALL, TI_PLAIN_ARRAY_LARGE	Plain array Collection	Plain Type. No TypeObject TypeIdentifier contains array dimensions and the TypeIdentifier of element. SMALL and LARGE indicate representation of dimensions.
TI_PLAIN_MAP_SMALL, TI_PLAIN_MAP_LARGE	Plain map Collection	Plain Type. No TypeObject

		TypeIdentifier contains length of map and the TypeIdentifier of key and element. SMALL and LARGE indicate representation of maximum length.
`TI_STRONGLY_CONNECTED_COMPONENT`	Types with mutual dependencies on other types	Not plain type. Has TypeObject. Uses a Hash computed on the TypeObjects of the set of mutually-dependent types. See clause 7.3.3.9.
`EK_COMPLETE`	Not mutually dependent on other types	Not plain type. Has TypeObject. Uses a Hash of the Complete TypeObject that describes the type. See 7.3.3.3
`EK_MINIMAL`	Not mutually dependent on other types	Not plain type. Has TypeObject. Uses a Hash of the Minimal TypeObject that describes the type. See 7.3.3.5.
`TK_ANNOTATION`	Annotation Declaration	Not plain type. Has TypeObject. Uses Hash of the TypeObject representation of the Annotation declaration
`TI_EXTENDED`		Reserved for future extensions

7.3.3.3 Complete TypeObject

The Complete `TypeObject` is a type representation with the same expressive power as the IDL (7.3.1, XML (7.3.2), and XSD (**Error! Reference source not found.**) representations. Any non-plain type represented in IDL can be converted to the Complete TypeObject representation and back to IDL with no information loss, other than formatting (e.g. presence of whitespace).

The Complete `TypeObject` provides an alternative representation of types suitable for programming and tooling.

The complete `TypeObject` is defined by its IDL representation; see the declaration of structure `CompleteTypeObject` in "Annex B: Representing Types with TypeObject".

7.3.3.4 Encoding of Member Attributes in TypeObject

The values of member attributes defined in the Type System—such as key, must_understand, optional, try_construct, and external_data—shall be encoded in the `TypeObject` using the `MemberFlag` bitmask defined by the normative IDL for TypeObject (see Annex B: Representing Types with TypeObject).

Implementations shall map the attribute values from these clauses onto the corresponding IDL-defined `MemberFlag` bits as follows:

- `TRY_CONSTRUCT1` and `TRY_CONSTRUCT2`: two-bit field encoding the `TryConstruct` behavior for the member (`DISCARD`, `USE_DEFAULT`, `TRIM`), see 7.2.2.8.
- `IS_OPTIONAL`: encodes the optional attribute (true if the member is optional), see 7.2.3.2.4.4.7.
- `IS_MUST_UNDERSTAND`: encodes the must_understand attribute (true if the member must be understood by consumers), see 7.2.3.2.4.4.6.
- `IS_KEY`: encodes the key attribute (true if the member is part of the type's key), see 7.2.3.2.4.4.8.
- `IS_DEFAULT`: marks the default union branch and default enum literal, as defined for union members (see 7.2.3.2.4.3) and enumerated literals (see 7.2.3.2.1.1).
- `IS_EXTERNAL`: encodes the external_data attribute, see 7.2.2.11.

For clarity, the TypeObject IDL also defines type-specific aliases of MemberFlag which constrain applicability: StructMemberFlag, UnionMemberFlag, UnionDiscriminatorFlag, AliasTypeFlag, etc.

7.3.3.5 Minimal TypeObject

The Minimal `TypeObject` provides a compact way to represent the type information relevant for a remote application to determine type assignability. This representation does not include information on the type that would not impact type assignability. For example user-defined annotations or the order of members for types with extensibility kind `MUTABLE`.

The Minimal `TypeObject` reduces the amount of information that applications need to send on the network in order to check type assignability between `DataWriters` and `DataReaders`.

The Minimal `TypeObject` is defined by its IDL representation; see the declaration of structure `MinimalTypeObject` in "Annex B: Representing Types with TypeObject".

7.3.3.6 TypeObject serialization

The serialization of a `TypeObject` shall happen in accordance to its IDL declaration and the general serialization rules defined in this specification (see Clause 7.4) for XCDR encoding version 2. Additional restrictions are placed such that the serialized result is bitwise identical independently of the vendor or platform where the serialization occurs. Specifically:

- The serialization shall use Little Endian encoding.
- The elements in `AnnotationParameterSeq` shall be ordered in increasing values of their `paramname_hash_typeid`.
- The elements in `AppliedAnnotationSeq` shall be ordered in increasing values of their `annotation_typeid`.

- The elements in `CompleteStructMemberSeq` shall be ordered in increasing values of the `member_index`.
- The elements in `MinimalStructMemberSeq` shall be ordered in increasing values of the `member_index`.
- The elements in `CompleteUnionMember` shall be ordered in increasing values of the `member_index`.
- The elements in `MinimalUnionMember` shall be ordered in increasing values of the `member_index`.
- The elements in `CompleteAnnotationMemberSeq` shall be ordered in increasing values of the `member_index`.
- The elements in `MinimalAnnotationMemberSeq` shall be ordered in increasing values of the `member_name hash`.
- The elements in `CompleteEnumeratedLiteralSeq` shall be ordered in increasing values of their numeric value.
- The elements in `MinimalEnumeratedLiteralSeq` shall be ordered in increasing values of their numeric value.
- The elements in `CompleteBitflagSeq` shall be ordered in increasing values of their `position`.
- The elements in `MinimalBitflagSeq` shall be ordered in increasing values of their `position`.
- The elements in `CompleteBitfieldSeq` shall be ordered in increasing values of their `position`.
- The elements in `MinimalBitfieldSeq` shall be ordered in increasing values of their `position`.

7.3.3.7 Classification of TypeIdentifiers

7.3.3.7.1 Fully-descriptive TypeIdentifiers

Some TypeIdentifiers do not involve computing the hash of any TypeObject. These are called Fully-descriptive TypeIdentifiers because they fully describe the Type. These are:

The TypeIdentifiers for Primitive and String types.

- The TypeIdentifiers of plain collections where the element (and key) TypeIdentifer is a fully descriptive TypeIdentifier. They are recognized by the contained PlainCollectionHeader having EquivalenceKind set to EK_BOTH.

7.3.3.7.2 Hash TypeIdentifiers

Some `TypeIdentifiers` are computed using a hash function applied over a serialized representation of the TypeObject. These are called HASH TypeIdentifiers. These are:

- Those with discriminator EK_MINIMAL, EK_COMPLETE, or TI_STRONG_COMPONENT
- Those with discriminator TI_PLAIN_SEQUENCE_SMALL, TI_PLAIN_SEQUENCE_LARGE, TI_PLAIN_ARRAY_SMALL, TI_PLAIN_ARRAY_LARGE, TI_PLAIN_MAP_SMALL, or TI_PLAIN_MAP_LARGE where the contained PlainCollectionHeader has EquivalenceKind EK_MINIMAL or EK_COMPLETE.

In contrast to the Fully-descriptive Identifiers HASH identifiers only identify a Type but do not provide a complete description of the type without the auxiliary TypeObjects whose hashes are included in the TypeIdentifier.

HASH TypeIdentifiers are further classified along two orthogonal dimensions:

- Direct vs. Indirect. This classification looks at the nature of their dependency on the TypeObjects.
- Minimal vs Complete. This classification looks at the kind of TypeObjects involved.

7.3.3.7.3 Direct Hash TypeIdentifiers

These are the HASH `TypeIdentifiers` with discriminator EK_MINIMAL, EK_COMPLETE, or TI_STRONG_COMPONENT.

7.3.3.7.4 Indirect Hash TypeIdentifiers

These are the `TypeIdentifiers` for plain collections with the element type identified using a HASH `TypeIdentifier`. They are distinguished by:

1. Having discriminator TI_PLAIN_SEQUENCE_SMALL, TI_PLAIN_SEQUENCE_LARGE, TI_PLAIN_ARRAY_SMALL, TI_PLAIN_ARRAY_LARGE, TI_PLAIN_MAP_SMALL, or TI_PLAIN_MAP_LARGE.
2. Having the contained PlainCollectionHeader with EquivalenceKind EK_MINIMAL or EK_COMPLETE.

7.3.3.7.5 Minimal Hash TypeIdentifiers

These are HASH `TypeIdentifiers` that involve hashing serialized MINIMAL TypeObjects. They consist of:

- those with discriminator EK_MINIMAL
- those with discriminator TI_STRONG_COMPONENT where the contained TypeObjectHashId has discriminator EK_MINIMAL.

- those for plain collections where the contained PlainCollectionHeader EquivalenceKind is EK_MINIMAL

7.3.3.7.6 Complete Hash TypeIdentifiers

These are HASH `TypeIdentifiers` that involve hashing serialized COMPLETE TypeObjects. They consist of:

- those with discriminator EK_COMPLETE
- those with discriminator TI_STRONG_COMPONENT where the contained TypeObjectHashId has discriminator EK_COMPLETE.
- those for plain collections where the contained PlainCollectionHeader EquivalenceKind is EK_COMPLETE

7.3.3.8 Type Equivalence

A distributed type system where types can be defined at different locations using different representations leads to the need of defining equivalence relations between types.

In set theory an "equivalence" relation is one satisfying the reflexive, symmetric, and transitive properties. Using the "~" sign to represent the relation, the three properties can be expressed as:

- Reflexive: T ~ T for every type "T" in the set of possible types.
- Symmetric: T1 ~ T2 implies T2 ~ T1
- Transitive: T1 ~ T2 and T2 ~ T3 implies T1 ~ T3

An equivalence relation partitions a set into disjoint subsets (equivalence classes) where each contains all the elements that are equivalent to each other. Being a "partition" each element belongs to exactly one of the equivalence classes.

An equivalence relation between types captures the intuitive notion that the related types "behave the same way" under a certain set of operations or use cases> because of this they can be considered to be "the same" from the perspective of those operations/use-cases.

When defining two equivalence relations R1 and R2 on the same set it may be the case that all elements that are equivalent under (R1) are also equivalent under the other (R2). In this case it is said that R1 is finer than R2, or alternatively that R2 is coarser than R1.

When this happens the finer relationship (R1) further partitions each equivalence class of the coarser (R2) in its own finer R1-equivalence classes. Said differently elements considered equivalent according to R2 may be differentiated by the R1 relation.

This specification defines two equivalence relations between types: **Complete** and **Minimal**.

- **Complete equivalence** relates types that can be considered the same for all practical uses of the type system, including code generation or displaying type information to the user.

- **Minimal equivalence** relates types that can be considered the same with regards to the type compatibility/assignability between a DataWriter and a DataReader as well as with regards to the data objects published by the DataWriter and received by the DataReader.

The formal definition of these equivalence relations is done in terms of TypeIdentifiers and TypeObjects.

- Two types are equivalent according to the Complete equivalence relation if and only if either they have equal Fully-Descriptive TypeIdentifiers, or else they have equal Complete TypeIdentifiers.
- Two types are equivalent according to the Minimal equivalence relation if and only if either they have equal Fully-Descriptive TypeIdentifiers, or else they have equal Minimal TypeIdentifiers.

From the definition of the Complete and Minimal TypeIdentifier it is clear that two types that are equivalent according to the complete relation are also equivalent according to the Minimal relation.

7.3.3.9 Types with mutual dependencies on other types

The XTYPES type system includes types that have mutual dependencies on other types. These types are used to express "recursive" data structures such as trees. For example:

```
struct NodeData {
   long l_data;
};
struct TreeNode;
struct TreeNode {
   NodeData data;
   sequence<@external TreeNode> children;
};
```

More complex dependency cycles are possible where one type depends on another, which depends on another forming a dependency chain that eventually points back to the original type.

The "simple" algorithm to compute the `TypeIdentifier` of a type based on a hash of its `TypeObject` fails when types have mutual dependencies on each other because the construction of the `TypeObject` requires knowledge of the `TypeIdentifier` of all the dependent types, creating a circular dependency.

7.3.3.9.1 Background: Basic graph theory

The problem of types with mutual dependencies can be formulated in terms of directed graphs (digraphs). Given a set of types we define the "Type Dependency" digraph for those types as follows:

- The vertices in the graph are the types.

- The edges in the graph represent the direct dependencies between types, that is, if type T1 directly references type T2 (e.g., T1 is a structure and T2 is the type of a member, or T1 is a collection, and T2 is the type of the collection element).

A 'directed path" in a digraph is a sequence of vertices where each vertex is connected to the next by a directed edge.

A "directed cycle" is a directed path that starts and ends on the same vertex.

Reachability relation: A vertex V1 is reachable from vertex V2 in the digraph if and only if there is a directed path from V2 to V1.

Strong connectivity relation: Two vertices V1 and V2 are **strongly connected** if and only they are mutually reachable, that is, V1 is reachable from V2 and V2 is also reachable from V1.

Strong connectivity is an equivalence relation. The resulting partitions are called **Strongly Connected Components**.

The **kernel DAG** is defined as the digraph created by "combining" strongly connected components into a single vertex:

- Kernel DAG vertices: The strongly connected components
- Kernel DAG edges: There is an edge from a strongly connected component SCC1 to a strongly connected component SCC2 if and only the original digraph contains some vertex belonging to SCC1 with an edge to a vertex belonging to SCC2.

A basic theorem in graph theory proves that Kernel DAG is acyclic, hence the name DAG which stands for Directed Acyclic Graph.

Figure 22 below shows an example digraph, its strongly connected components, and the corresponding Kernel DAG.

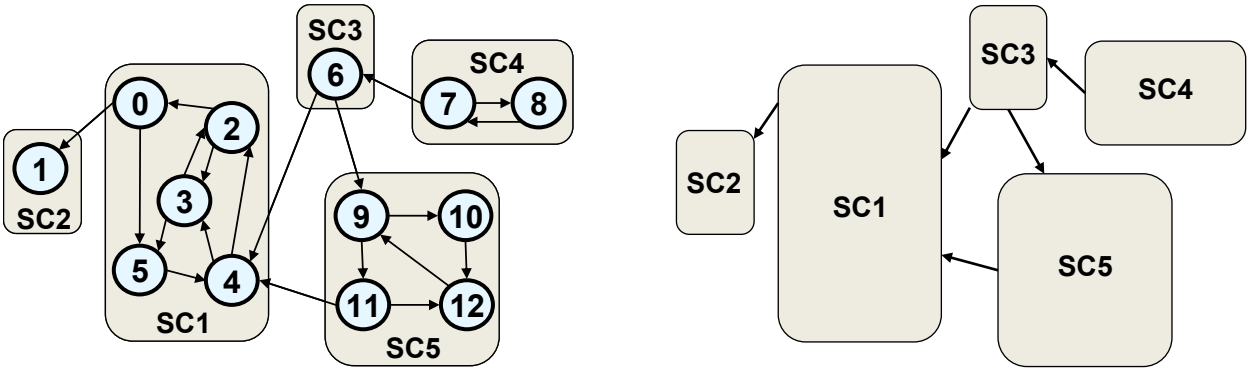

Figure 22 – Directed graph, Strongly Connected Components, and Kernel DAG

*The strongly connectivity relation partitions the vertices in a digraph into subsets called **strongly connected components**. This is shown on the left part of the figure. The right side shows the Kernel DAG constructed using the strongly connected components as vertices. It is always a directed acyclic graph (DAG).*

7.3.3.10 Computation of Type identifiers for types with mutual dependencies

7.3.3.10.1 Introduction

Mutual dependencies between types appear as directed cycles in the type dependency digraph. For example, the type dependency graph for the "tree" types declared above has a directed cycle involving the vertices "TreeNode" and "sequence<TreeNode>". This is shown in Figure 23 below.

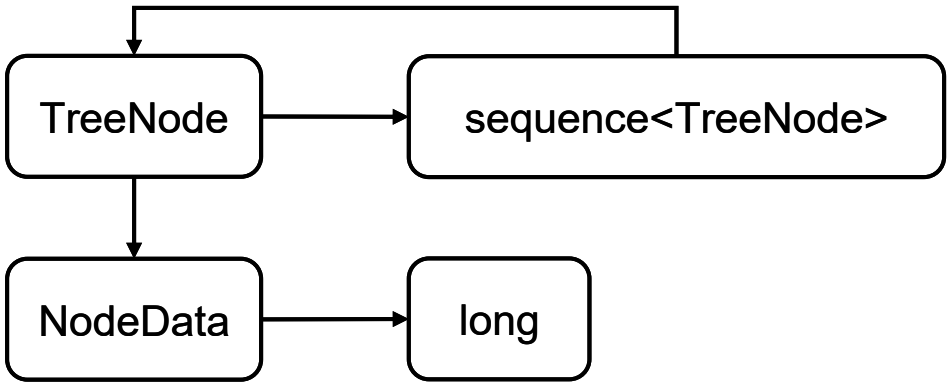

Figure 23 – Dependency graph derived from a set of type definitions

Type representation and type dependencies operate on the equivalence classes defined by the COMPLETE and MINIMAL type relations defined in Clause 7.3.3.8. Types belonging to the same equivalence class have the same TypeObject so they are treated as "the same type". Depending on the relation (MINIMAL or COMPLETE) selected we will end up with a different set of types and type dependencies.

The algorithm to generate the `TypeObjects` and `TypeIdentifiers` is the same regardless of the equivalence relation chosen. To generate both the algorithm will be run two times, one for each equivalence relation.

The "basic" algorithm to compute Hash `TypeIdentifier` consists of hashing the serialized `TypeObject`. The construction of a `TypeObject` requires having the `TypeIdentifiers` of all the types the `TypeObject` depends on. Therefore this "basic" algorithm can handle only situations where the dependency graph does not have cycles, that it, it is a DAG.

- The following clause defines a more general algorithm to construct `TypeIdentifiers` and `TypeObjects` that can also handle cycles in the dependency graph.

7.3.3.10.2 Sorting Label

Label for the remaining type categories (see 7.2.2.2):

- Named Types
- Anonymous Types

7.3.3.10.2.1 Sorting Label for Named types

The sorting label for Named Types shall be equal to the Type's fully-qualified name. See 7.2.2.2.

7.3.3.10.2.2 Sorting label for Anonymous types

Anonymous types can be grouped into three categories:

- Anonymous types used to define an Alias type
- Anonymous types used to define the type of a structure or unions member
- Anonymous types used to define the type of a collection element

The sub-sections that follow define the sorting label for each of these.

7.3.3.10.2.2.1 Anonymous types used to define an Alias type

These types shall be assigned a sorting label obtained by concatenating the fully-qualified name of the Alias type with the ".t" suffix.

Example:

```
module MyModule {
    typedef sequence<T> Sequence1;
    typedef sequence<T, 10> Sequence2;
    typedef map<KT,VT> Map1;
    typedef map<KT,VT, 10> Map2;
    typedef T Array1[10];
    typedef T Array2[10][20];
};
```

The sorted label for the (bolded) types referenced by the typedefs above are:

- "MyModule::Sequence1.t"
- "MyModule::Sequence2.t"
- "MyModule::Map1.t"
- "MyModule::Map2.t"
- "MyModule::Array1.t"
- "MyModule::Array2.t"

7.3.3.10.2.2.2 Anonymous types used to define the type of a structure or union member

These types shall be assigned a sorting label obtained by concatenating the fully-qualified name of the containing type with the member name using "." as the separator.

Example:

```
module MyModule {
    struct MyStruct {
        sequence<T>  seqMember1;
```

```
        sequence<T, 10> seqMember2;
        map<KT, VT>  mapMember1;
        map<KT, VT, 10>  mapMember2;
        T arrMember1[10];
        T arrMember2[10][20];
    };
};
```

The names for the (bolded) types associated with members of MyStruct above are:

- "MyModule::MyStruct.seqMember1"
- "MyModule::MyStruct.seqMember2"
- "MyModule::MyStruct.mapMember1"
- "MyModule::MyStruct.mapMember2"
- "MyModule::MyStruct.arrMember1"
- "MyModule::MyStruct.arrMember2"

7.3.3.10.2.2.3 Anonymous types used to define the type of a collection element

These anonymous types are used to define the type associated with Array and Sequence elements as well as the types associated with Map key and value elements.

These types shall be assigned a sorting label obtained by concatenating the sorting of the containing anonymous type with a suffix that depends on the kind of containing collection:

- If the containing collection is a sequence, the suffix shall be ".s"
- If the containing collection is an array, the suffix shall be ".a"
- If the containing collection is a map then:
    - If the anonymous type is used to type the map key, the suffix shall be ".k"
    - If the anonymous type is used for type the map value, the suffix shall be ".v"

This rule applies recursively if the element, key, or value type is another collection.

Example1:

```
module MyModule {
    typedef string<10> String1;
    typedef sequence<sequence<string<10>> String2;
    typedef sequence<sequence<string<10>>> String3;
    typedef sequence<sequence<T>> Sequence1;
    typedef sequence<sequence<T>> Sequence2;
```

```
    typedef sequence<sequence<sequence<T>>> Sequence3;
    typedef sequence<sequence<sequence<T>>> Sequence4;
    typedef map<T1, sequence<T2, 20>> Map1;
    typedef map<T1, sequence<T2, 20>> Map2;
    typedef map<T1, sequence<sequence<T2, 10>>> Map3;
    typedef map<T1, sequence<sequence<T2, 10>>> Map4;
    typedef map<T1, map<T2, sequence<T3, 10>>> Map5;
    typedef map<T1, map<T2, sequence<T3, 10>>> Map6;
    typedef sequence<T>> Array1[10];
    typedef sequence<sequence<T>> Array1[10][20];
    typedef sequence<sequence<T>> Array1[10][20];
};
```

The names for the (bolded) types referenced by the typedefs above are:

- "MyModule::String1.t"
- "MyModule::String1.t.s"
- "MyModule::Sequence3.t.s.s"
- "MyModule::Sequence1.t"
- "MyModule::Sequence2.t.s"
- "MyModule::Sequence3.t.s"
- "MyModule::Sequence4.t.s.s"
- "MyModule::Map1.t"
- "MyModule::Map2.t.v"
- "MyModule::Map3.t.v"
- "MyModule::Map4.t.v.s"
- "MyModule::Map5.t.v"
- "MyModule::Map6.t.v.v"

Example2:

```
module MyModule {
    struct MyStruct {
        string<10>            strMember1;
        sequence<string<10>>  strMember2;
        sequence<sequence<T>>  seqMember1;
        sequence<sequence<T>>  seqMember2;
```

```
        map<KT, sequence<T>>  mapMember1;
        map<KT, sequence<T>>  mapMember2;
        map<KT1, map<KT2, sequence<T>, 10>>  mapMember3;
        map<KT1, map<KT2, sequence<T>, 10>>  mapMember4;
        map<KT1, map<KT2, sequence<T>, 10>>  mapMember5;
        map<KT1, map<KT2, sequence<sequence<T>>, 10>>  mapMember6;
        sequence<sequence<T>> arrMember1[10];
        sequence<sequence<T>> arrMember2[10];
        sequence<map<KT, sequence<T>>> arrMember3[10];
    };
};
```

The names for the (bolded) types associated with members of MyStruct above are:

- "MyModule::MyStruct::strMember1"
- "MyModule::MyStruct::seqMember2.s"
- "MyModule::MyStruct.seqMember1"
- "MyModule::MyStruct.seqMember2.s"
- "MyModule::MyStruct.mapMember1"
- "MyModule::MyStruct.mapMember2.v"
- "MyModule::MyStruct.mapMember3"
- "MyModule::MyStruct.mapMember4.v"
- "MyModule::MyStruct.mapMember5.v.v"
- "MyModule::MyStruct.mapMember6.v.v.s"
- "MyModule::MyStruct.arrMember1"
- "MyModule::MyStruct.arrMember2.s"
- "MyModule::MyStruct.arrMember3.s.v”

7.3.3.10.3 Algorithm

Let **EK** be the desired equivalence kind. Either EK_COMPLETE or EK_MINIMAL, which selects whether we are constructing the `TypeObjects` and `TypeIdentifiers` according to the MINIMAL or the COMPLETE equivalence relation.

Let **Types(EK)** a self-contained set of types (i.e., type equivalence classes) for the selected equivalence relation EK. By self-contained we mean a set of types that does not depend on any type outside the set.

Let T be an element of **Types(EK)** whose `TypeObject` and `TypeIdentifier` we wish to compute. The algorithm will construct the `TypeObject` and `TypeIdentifier` for all types in **Types(EK)** but it can be started with any type as an entry point.

Let **TypeDependencyDG(T)** be the dependency digraph that contains only the types that are reachable from T. If this graph has no cycles, then T is not affected by mutual dependencies and the `TypeIdentifier` can use the regular algorithm of hashing the serialized `TypeObjects`, which can be constructed recursively. Otherwise proceed to step 2.

2. Let **ReducedDependencyDG(T)** be the subdigraph of **TypeDependencyDG(T)** where all the vertices that have no outgoing edges are removed. These represent types that do not depend on any other types so their `TypeIdentifier` (and `TypeObject`) can be computed directly.
3. Identify the Strongly Connected Components of the **ReducedDependencyDG(T)**. Let **DependencyKernelDAG(T)** be the Kernel DAG of **ReducedDependencyDG(T)**.
4. Use a depth-first algorithm to compute the `TypeIdentifier` of the types on each Strongly Connected Component in **DependencyKernelDAG(T)**:
    a. If the Strongly Connected Component (**SCC**) has a single type, then use the regular algorithm to compute its `TypeIdentifier` based on the type identifiers of all types it depends on. The depth first order ensures that those identifiers have already been computed.
    b. If the Strongly Connected Component (**SCC**) has multiple types, then sort them using the lexicographic order of the type's Sorting Label (see 7.3.3.10.2) the sort index of each type **U** belonging to the **SCC** starting with index 1 for the first type.
        i. Temporarily set the `TypeIdentifier` of each U belonging to the SC to:
            - discriminator = TI_STRONGLY_CONNECTED_COMPONENT
            - sc_component_id =
                - scc_hashid = {discriminator=**EK**, hash= 0}
                - scc_length = Number of types in SCC
                - scc_index = SCCIndex(U) . Note that 1 <= scc_index <= scc_length
        ii. Construct the `TypeObject` of all the types in the SC using the temporary `TypeIdentifier` for references to other types in the SCC. The depth first order ensures that `TypeIdentifier` for other types that the SCC depends on have already been computed.
    c. Place computed `TypeObjects` from step 4.b into a sequence `TypeObjectSeq` in the order of their scc_index.
    d. Serialize the `TypeObjectSeq` using the XCDR serialization for sequences with encoding version 2 and little endian.

e. Compute the MD5 hash of the serialized buffer. Let **EquivalenceHash(SC)** be the first 14 bytes.

f. Set the TypeIdentifier of each of the types in SCC to:

- discriminator = TI_STRONGLY_CONNECTED_COMPONENT
- strong_component_id = StronglyConnectedComponentId where:
  - scc_hashid = {discriminator=**EK**, hash= **EquivalenceHash(SC)** }
  - scc_length = Number of types in SCC
  - scc_index = SCCIndex(U)

Implementation notes: (non-normative):

- The strongly connected component of a vertex V can be constructed as the set of vertices W reachable from V both by backwards and forwards traversal. If we define Forward(V) as the vertices reachable from V and Backward(V) as the set of vertices from which it is possible to reach V. Then:
  - StronglyConnectedComponent(V) = Forward (V) ∩ Backward (V).
  - Forward (V) can be computed using depth first search (DFS) from V.
  - Backward (V) can be computed using DFS on the transpose graph obtained by inverting every edge.
- There are simple linear time algorithms (e.g. Kosaraju-Sharir) that compute the strongly connected components of a graph.

7.3.3.10.4 Strongly Connected Components Identifier (SCCIdentifier)

Each type in a Strongly Connected Component (SCC) is uniquely identified by a **StronglyConnectedComponentId.** The **StronglyConnectedComponentId** is constructed using the algorithm specified in 7.3.3.10.3.

The **StronglyConnectedComponentId** contains the number of types in the strongly connected component (field ***scc_length***) and a hash of all the corresponding `TypeObjects` (field ***sc_component_id.scc_hashid.hash***).

From the **StronglyConnectedComponentId** of any type in the SCC it is possible to derive the `TypeIdentifiers` of all the types in the SCC. The `TypeIdentifiers` of all the types belonging to the same SCC only differ on the ***sc_component_id.scc_index*** field, which always takes values from 1 to ***sc_component_id.scc_length***.

There are situations where an SCC needs to be identified without referencing a concrete type inside the SCC. In this situation a `TypeIdentifier` is constructed the same way as for any of the types in the SCC except the ***sc_component_id.scc_index*** field is set to 0. We refer to this special `TypeIdentifier` recognizable by its discriminator being equal to TI_STRONGLY_CONNECTED_COMPONENT and ***scc_index*** = 0 AS the **SCCIdentifier**.

The `TypeIdentifier` of any type in the SCC contains the information needed to construct the **SCCIdentifier** and vice versa.

7.3.3.11 Representing License Information

`License` information may be associated any Named Type or with a module (see 7.2.2.6.3). That association conceptually includes the following information, derived from the applicable `LicenseUse` (see 7.2.2.6.2) and `LicenseType` (see 7.2.2.6.1):

- The `LicenseUse` name and link (i.e., the identifier of the specific usage of the license within a product or data model, and the URL to the project's license/notice file).
- The `LicenseType` name and link (i.e., the identifier of the license kind, such as "Apache-2.0", and its canonical URL).

This information is considered part of the type definition and therefore must appear in the Complete `TypeObject`.

One way to encode this information would be to attach it directly as an applied annotation containing four string values. However, doing so would replicate multiple arbitrary-length strings in every Complete `TypeObject`. To improve efficiency, this specification defines two related concepts:

- an `AppliedLicenseTypeIdentifierHelper`, and
- its corresponding `AppliedLicenseTypeIdentifier`.

These allow the Complete `TypeObject` to contain only an identifier (a hash) of the `License` Information rather than four strings.

7.3.3.11.1 AppliedLicenseTypeIdentifierHelper

An `AppliedLicenseTypeIdentifierHelper` is an annotation declaration that is logically constructed for the purpose of capturing the effective license metadata of a particular Named Type. The license information is captured as default values of the annotation's members.

For each Named Type with associayed License information. the annotation shall be constructed as follows:

```
@annotation AppliedLicenseTypeIdentifierHelper {
    string license_type_name  default "<name from LicenseType>";
    string license_type_link  default "<link from LicenseType>";
    string license_use_name   default "<name from LicenseUse>";
    string license_use_link   default "<link from LicenseUse>";
};
```

In this declaration:

- *license_type_name* and *license_type_link* reflect the associated `LicenseType`
- *license_use_name* and *license_use_link* identify the associated `LicenseUse`.

The default values assigned to these members are the four strings obtained by following the `License` chain in effect for the Named Type to its `LicenseUse` and `LicenseType` (see 7.2.2.6).

An `AppliedLicenseTypeIdentifierHelper` is an annotation declaration. Therefore, it has both a `TypeObject` and a `TypeIdentifier`.

7.3.3.11.2 AppliedLicenseTypeIdentifier

The term `AppliedLicenseTypeIdentifier` refers to the `TypeIdentifier` computed from the `TypeObject` of an `AppliedLicenseTypeIdentifierHelper`.

Because the `AppliedLicenseTypeIdentifier` is derived from the hash of the `AppliedLicenseTypeIdentifierHelper` annotation's `TypeObject`, two Named Types that carry the same effective license information (i.e., the same four strings) will, by construction, have the same `AppliedLicenseTypeIdentifier`.

7.3.3.11.3 Computing the AppliedLicenseTypeIdentifier for a Named Type

When a `Named` Type T has an associated `License` (see 7.2.2.6.3), the XTYPES implementation shall compute an `AppliedLicenseTypeIdentifier` for T as follows:

(1) Determine the effective license for T, after applying module-level defaults and overrides as described in 7.2.2.6.3. This yields four strings: license_type_name, license_type_link, license_use_name, and license_use_link.

(2) Synthesize (or reuse, if already synthesized for identical values) an `AppliedLicenseTypeIdentifierHelper` annotation declaration whose members have those four strings as their default values.

(3) Generate the `TypeObject` that represents this helper annotation declaration.

(4) Compute the corresponding `TypeIdentifier` from that `TypeObject`. This is the `AppliedLicenseTypeIdentifier` for T.

(5) Include that `AppliedLicenseTypeIdentifier` in the Complete `TypeObject` representation of T according to the encoding defined in Annex B: (Representing Types with TypeObject).

A receiver may resolve an `AppliedLicenseTypeIdentifier` back to its corresponding helper-annotation `TypeObject` using the Builtin `TypeLookup` Service (see 7.6.3.3), in the same manner used to resolve any other `TypeIdentifier`.

7.4 Data Representation

The Data Representation module specifies the ways in which a data object of a given type can be externalized so that it can be stored in a file or communicated over the network. This is also commonly referred as "data serialization" or "data marshaling."

Data Representations serve multiple purposes such as:

- Represent data in a “byte stream” so it can be sent over the network
- Represent data in a “byte stream” so it can be stored in a file
- Represent data in a human-readable form so it can be displayed to the user
- Provide a language for the user to enter data-values to a tool or specify them in a file

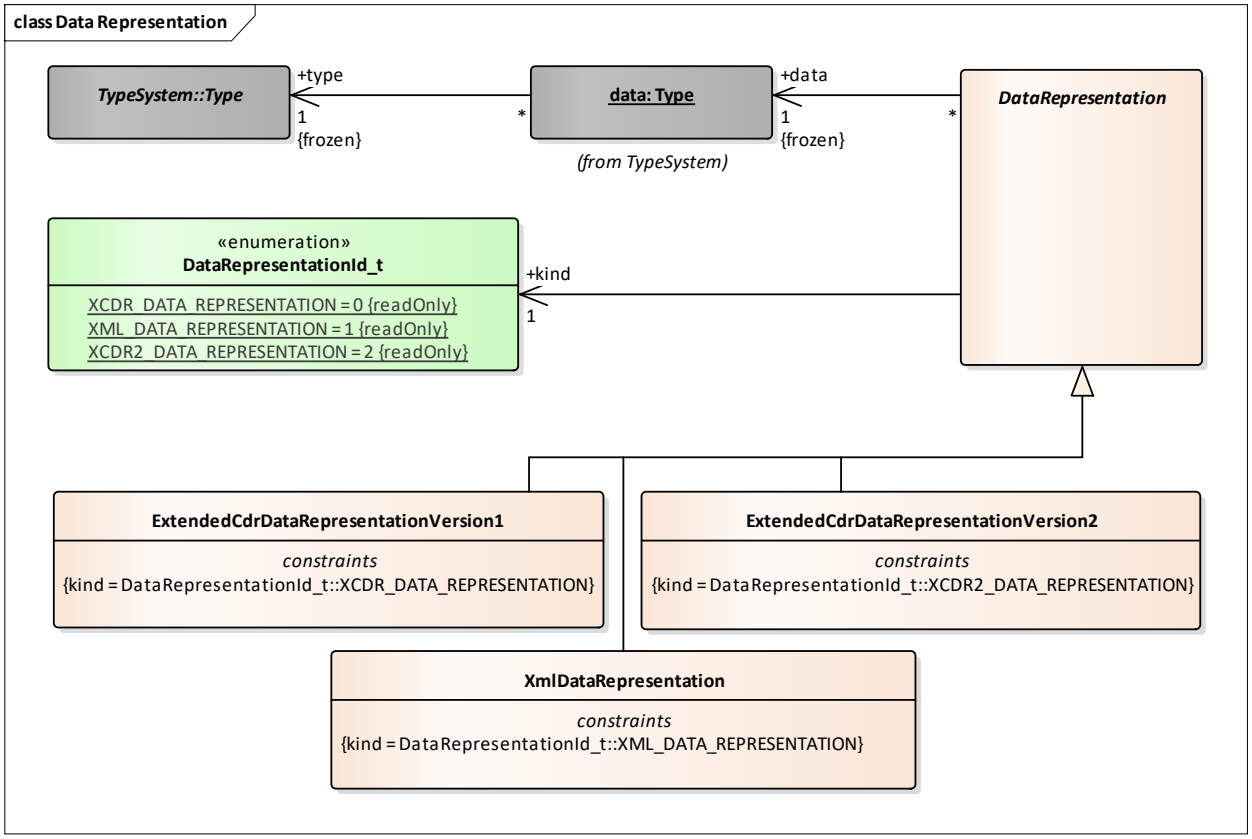

Figure 24 – Data Representation—conceptual model

This specification introduces multiple Data Representations. The reason for defining multiple type representations is that each of these is better suited or optimized for a particular purpose. These representations are all mostly equivalent. Consequently, other than convenience or performance, there is little reason to use one versus the other.

The alternative representations are summarized in Table 27.

Table 27 – Alternative Data Representations

Data Representation	*Reasons for using it*	*Disadvantages*
Extended CDR, encompassing both "traditional" CDR and parameterized CDR	Compact and efficient binary representation. Minimizes CPU and Bandwidth used. Supports type evolution. Existing international OMG Standard. (Traditional CDR from CORBA [CDR]; parameterized CDR from RTPS [RTPS].) Already in used in the DDS Interoperability Protocol.	Not human readable.
XML	Human Readable Easily parsed and transformed with standard tools	CPU Intensive Uses 10 or 20 times more space than CDR

7.4.1 Extended CDR Representation (encoding version 1)

This specification defines extensions of the OMG CDR representation [CDR] able to accommodate both optional members and appendable/mutable types. These extensions result in two encoding formats: PLAIN_CDR and PL_CDR.

Both encoding formats leverage the OMG CDR representation for all primitive types and non-mutable constructed types where the (traditional) CDR representation is well defined:

- PLAIN_CDR introduces extensions to CDR in order to handle optional members, bitmasks, and maps.
- PL_CDR leverages the RTPS Parameter List encoding to handle mutable types.

7.4.1.1 PLAIN_CDR Encoding

The PLAIN_CDR encoding shall be used for final and appendable types, including (trivially) primitive types. It shall also be used for all string, sequence, and map types. Aggregated types declared as mutable shall use the PL_CDR encoding described in Clause 7.4.1.2.

The PLAIN_CDR encoding is based on the traditional CDR representation format [CDR] with the minimal extensions described below needed to handle the new types and concepts introduced by this specification.

The [RTPS] specification states that following the serialized data submessage element, padding bytes shall be added so that the next submessage starts at a 4-byte offset relative to the beginning

of the RTPS message. This XTYPES specification further requires that any padding bytes added at the end of the serialized data shall be set to zero.

7.4.1.1.1 Primitive types

The PLAIN_CDR encoding for primitive types shall be the same as in "traditional" CDR [CDR]. Specifically:

- The serialized data shall be encoded at an offset that aligned to the size of the primitive type.
- An endianness byte swap shall be performed in case the native system endianness is different from the one currently configured in the XCDR stream (XCDR.cendian).

Table 28 below summarizes the serialization of various primitive types.

Table 28 – Serialization of primitive types in version 1 encoding

Primitive Type	*Encoded Size*	*Alignment (version 1)*	*Byte representation*
`Byte`	1	1	The byte value
`Boolean`	1	1	0 for false, 1 for true
`Char8`	1	1	The character value encoded as described in 7.2.3.1.1.3
`Char16`	2	2	The character value encoded as described in 7.2.3.1.1.3
`Int8` `UInt8`	1	1	The integer value using two's complement notation
`Int16` `UInt16`	2	2	The integer value using two's complement notation
`Int32` `UInt32`	4	4	The integer value using two's complement notation
`Int64` `UInt64`	8	8	The integer value using two's complement notation
`Float32`	4	4	IEEE standard for normalized single-precision floating-point numbers [IEEE-748]
`Float64`	8	8	IEEE standard for normalized double-precision floating-point numbers [IEEE-748]
`Float128`	16	8	IEEE standard for normalized quadruple-precision floating-point numbers [IEEE-748]

7.4.1.1.2 Character Data

Objects of `Char8` type shall not be interpreted to have a specific encoding and shall be serialized as-is in the same way as the Byte primitive type.

Objects of `String<Char8>` type shall be represented using the UTF-8 character encoding. The serialized length of an object of type `String<Char8>` shall be the number of bytes in the CDR buffer taken by the `String<Char8>` characters, including the terminating NUL character. The serialized length may not be the same as the number of Unicode characters because a single Unicode character encoded using the UTF-8 encoding may take one to four bytes.

Objects of `String<Char16>` type shall be represented using the UTF-16 character encoding. The serialized length of an object of type `String<Char16>` shall be the number of bytes in the CDR buffer taken by the `String<Char16>` characters. This is twice the number of characters in the string because a single character (in the Basic Multilingual Plane) encoded using UTF-16 takes 2 bytes to serialize.

The UTF-16 representation of object of type `String<Char16>` shall not include a Byte Order Mark (BOM). The representation shall also not include any terminating NUL character(s).

Rationale: By setting the serialized length equal to the number of bytes the representation could support sending UTF-16 encoded Unicode characters outside the BMP (which map to two UTF-16 units). In this case, the serialized length would still indicate the number of bytes until the end of the string. The byte order used by the UTF-16 representation can be inferred from the one already available in the RTPS Encapsulation Identifier (see Clause 7.6.3.1.2), therefore the BOM is not needed. Finally terminating UTF-16 encoded strings with NUL characters is not considered best practice and the latest versions of OMG CDR do not do it.

7.4.1.1.3 Enumerated Types

7.4.1.1.3.1 Enumeration Types

Objects of enumerated types shall be serialized as integers, the sizes of which shall depend on the “bit bound” of their associated type.

Table 29 – Serialization of enumeration types

Corresponding Primitive Type	*Bit Bound*
`Int8`	1-8
`Int16`	9-16
`Int32`	17-32 (32 bits is the default size, and corresponds to all enumerated types prior to this specification)

7.4.1.1.3.2 Bitmask Types

Objects of bitmask types shall be serialized in the same way as the following primitive types, depending on the bitmask’s bound:

Table 30 – Serialization of bitmask types

Bound	*Corresponding Primitive Type*
[1..8]	`UInt8`
[9..16]	`UInt16`
[17..32]	`UInt32`
[33..64]	`UInt64`

Bit indexes are counted from zero starting at the least-significant bit of the full byte size of the bitmask. In the case where the bound of the bitmask is less than the number of bits in the corresponding primitive type, the states of the remaining serialized bits are not specified, and those bits are not considered to be part of the bitmask.

7.4.1.1.4 Map Types

Objects of map types shall be represented according to the following equivalent IDL:

```
struct MapEntry_<key_type>_<value_type>[_<bound>] {
   <key_type> key;
   <value_type> value;
};

typedef sequence<MapEntry_<key_type>_<value_type>[_<bound>][, <bound>]>
Map_<key_type>_<value_type>[_<bound>];
```

The <*key_type*> and <*value_type*> names are as defined the Type System. See also Clause 7.2.3.2.2.1, which defines the implicit names of collection types.

For example, objects of the following IDL map type:

```
map<long, float>
```

…shall be serialized as if they were of the following IDL sequence type:

```
struct MapEntry_Int32_Float32 {
   long key;
   float value;
};
typedef sequence<MapEntry_Int32_Float32> Map_Int32_Float32;
```

7.4.1.1.5 Structures

Objects of structure type shall be represented as defined by the CDR specification [CDR], augmented as described below.

7.4.1.1.5.1 Inheritance

The members defined by the base type, if any, shall be serialized before the members of their derived types. The representation shall be exactly as if all of the members had been defined, in the same order, in the most-derived type.

7.4.1.1.5.2 Optional Members

Structure members marked as optional shall be preceded by a parameter header as described in Clause 7.4.1.2, “Parameterized CDR ”, below.

7.4.1.2 Parameterized CDR Encoding

The parameterized CDR encoding is based on the RTPS Parameter List CDR encoding defined in [RTPS].

Each element, or parameter, within a parameter list data structure is simply a CDR-encapsulated block of data. Preceding each one is a parameter header consisting of a two-byte parameter ID followed by a two-byte parameter length. One parameter follows another until a list-terminating sentinel is reached.

Unlike it is stated in [RTPS] Sub Clause 9.4.2.11 “ParameterList”, the value of the parameter length is the exact length of the serialized member. It does not account for any padding bytes that may follow the serialized member. Padding bytes may be added in order to start the next parameterID at a 4 byte offset relative to the previous parameterID.

This data representation uses elements of the parameter list data structure for two purposes:

- Any object of a mutable aggregated type shall be serialized as a parameter list. Each of its members shall correspond to a single parameter within that list.
- Any optional member of a final or appendable structure shall be preceded by a parameter header describing that member. If the member takes no value within that particular object, the data length indicated by the header shall be zero. This reuse of the parameter header data structure does not constitute a complete parameter list: the optional member shall not be followed by list-terminating sentinel.

7.4.1.2.1 Interpretation of Parameter ID Values

As described in Clause 9.6.2.2.1, *ParameterId space*, of the RTPS Specification, the 16-bit-wide parameter ID range may be interpreted as a two-bit-wide bitmask followed by a 14-bit wide unsigned integer.

- The first bit of the bitmask—the most-significant bit of 16-bit-wide the parameter ID as a whole—indicates whether the parameter has an implementation-specific interpretation. This specification refers to this bit as `FLAG_IMPL_EXTENSION`.
- The second bit of the bitmask indicates whether the parameter, if its ID is not recognized by the consuming implementation, may be simply ignored or whether it causes the entire data sample to be discarded. This specification refers to this bit as `FLAG_MUST_UNDERSTAND`. This bit shall be set if and only if the `must_understand` property of the member being encapsulated is set to true.

Within the 14-bit-wide integer region of the parameter ID, this specification further reserves the largest 255 values—from 16,129 (0x3F01) to 16,383 (0x3FFF)—for use by the OMG in this specification and future specifications. Table 31 below identifies the reserved parameter ID values. For a parameter to be recognized as one of the well-known values in Table 31, the `FLAG_IMPL_EXTENSION` bit must be set to zero. Refer to Table 31 for the value of the `FLAG_MUST_UNDERSTAND` bit.

Table 31 – Reserved parameter ID values

Name	*14-Bit Hex Value(s)*	*FLAG_ MUST_UNDERSTAND set?*	*Description*
PID_EXTENDED	0x3F01	Yes	Allows the specification of large member ID and/or data length values; see below
PID_LIST_END	0x3F02	Yes	Indicates the end of the parameter list data structure. RTPS specifies that the PID value 1 shall be used to terminate parameter lists within the DDS built-in topic data types. Rather than reserving this parameter ID for all types, thereby complicating the member ID-to-parameter ID mapping rules for all producers and consumers of this Data Representation, *Simple Discovery types* shall be subject to a special limitation: member ID 1 shall not be used and parameter ID 1 shall terminate the parameter list to provide backwards compatibility. Implementations shall be robust to receiving parameter ID 0x3F02 to indicate the end of a list as well. These types consist of the built-in topic data types, and those other types that contain them as members, as defined by [RTPS].
PID_IGNORE[4]	0x3F03	No	All consumers of this Data Representation shall ignore parameters with this ID.
Reserved for OMG	0x3F04-0x3FFF	*N/A*	*Reserved for OMG*

When writing data, implementations of this specification shall set the FLAG_MUST_UNDERSTAND bit as described in Table 31. When reading data, implementations of this specification shall be robust to any setting of the FLAG_MUST_UNDERSTAND bit and accept the parameter nevertheless.

[4] **Design rationale (non-normative)**: RTPS uses PID 0 ("PID_PAD"), corresponding to member ID 0, as a padding field. PID_IGNORE applies this concept to all data types using this Data Representation. The additional reservation of PID 0 is not

This specification extends the parameter list data structure to permit 32-bit parameter IDs and data lengths up to 4 Giga-Bytes. This extension uses the reserved must-understand 16-bit parameter ID `PID_EXTENDED` to indicate that a member's parameter ID and/or length require 32-bits. The member ID (long member ID) and member length (long member length) follow in the 8 bytes directly after the `PID_EXTENDED` parameter ID and accompanying 16-bit length.

The value of the `PID_EXTENDED` with the must understand flag set is 0x7F01 (that is 0x4000 + 0x3F01).

The four bytes following the `PID_EXTENDED` and length shall be a serialized `UINT32` value "eMemberHeader" that is constructed by combining four 1-bit flags with the 28-bit member ID. The flags occupy the 4 most significant bits of the `UINT32` value. The flags are combined with the `memberId` as shown below:

```
FLAG_1 = 0x80000000

FLAG_2 = 0x40000000

FLAG_3 = 0x20000000

FLAG_4 = 0x10000000

eMemberHeader = FLAG_1 & FLAG_IMPL_EXTENSION
              + FLAG_2 & FLAG_MUST_UNDERSTAND
              + FLAG_3 & FLAG_UNSPECIFIED1
              + FLAG_4 & FLAG_UNSPECIFIED2
              + memberId
```

As indicated by the formula above, FLAG_1 encodes the implementation extension flag, FLAG_2 encodes the must understand flag, and FLAG_3 and FLAG_4 are left for future extensions.

The second four bytes following the `PID_EXTENDED` and length shall be interpreted as a 32-bit unsigned integer (llength) that contains the length of the serialized member. Note that llength is the exact length of the serialized member and does not account for any padding that may follow the member.

The value of the 16-bit length associated with the `PID_EXTENDED` (`slength`) shall be equal to eight.

The serialized member shall start immediately after the long member length (`llength`). That is exactly 12 bytes from the beginning of the `PID_EXTENDED` parameter.

See Figure 25 for an example of the layout of the CDR buffer where `PID_EXTENDED` is used.

```
Big Endian Representation

0...2...4.......8...............16.............24.............32
```

necessary: because the types defined by RTPS do not use member ID 0, consumers of those types will naturally ignore any incidence of its corresponding PID that they encounter.

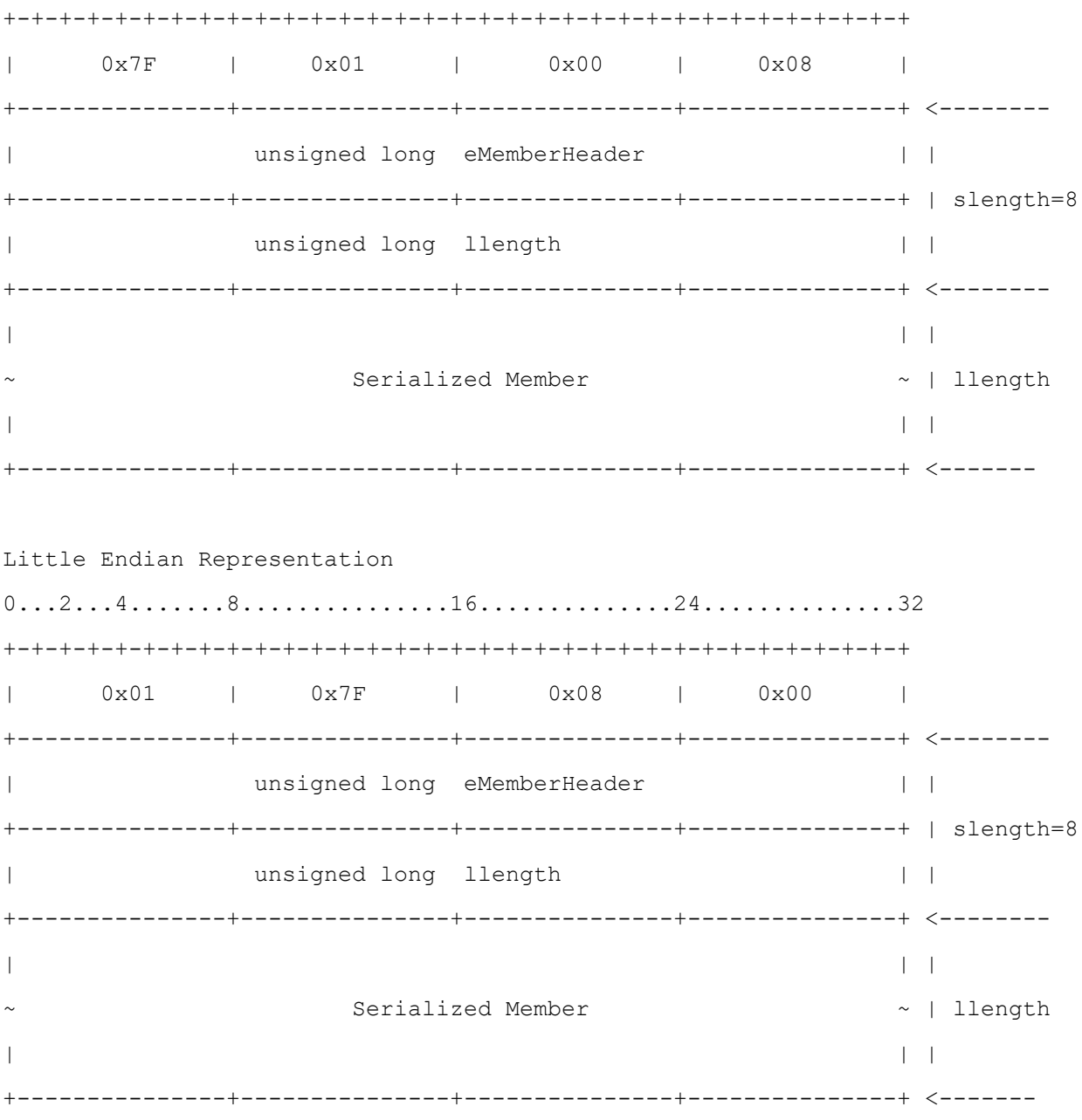

Figure 25 – Usage of PID_EXTENDED within the CDR Buffer

The setting of the `FLAG_IMPL_EXTENSION` and `FLAG_MUST_UNDERSTAND` bits in the 16-bit parameter ID shall not be interpreted to apply to the extended parameter as well. Instead, the first most-significant bit of the four-bitmask of flags within the extended parameter header shall represent the value of `FLAG_IMPL_EXTENSION` for the data member. The second most-significant bit shall represent the `FLAG_MUST_UNDERSTAND` value of the data member. The remaining two bits, unless specified by some other OMG specification, should be set to zero.

These extended parameter headers, based on `PID_EXTENDED`, shall be legal within the parameter list data structures used to serialize objects of mutable aggregated types. They shall also be legal when preceding optional members of final or appendable structures, as described above.

The alignment rules for extended parameters shall be the same as those for non-extended parameters, which are defined in [RTPS] Clause 9.4.2.11.

7.4.1.2.2 Member ID-to-Parameter ID Mapping

The mapping from member IDs to parameters shall be as follows:

- Member IDs from 0 to 16,128 (0x3F00) inclusive shall be represented exactly in the lower 14 bits of the parameter ID.
- All other member IDs must be expressed using the extended parameter header format.
- Almost any parameter can legally be expressed using extended parameter headers. There is no requirement that parameters that *could* be described with the shorter header defined by the RTPS Specification *must* be described that way; if a parameter could be described using a short parameter header or an extended header, the short and extended expressions of that header shall be considered totally equivalent. This mapping ensures that members of user-defined data types will never set the `FLAG_IMPL_EXTENSION` bit. Currently, the `FLAG_IMPL_EXTENSION` bit is used only for RTPS discovery-defined data types, which may or may not have the bitmask as defined by the RTPS Specification itself.

7.4.1.2.3 Omission and Reordering of Members of Aggregated Types

Because each parameter (type member, in this case) is explicitly identified, and identification of mutable structure members occurs based on the IDs of those parameters, members of mutable structures may appear in any order. Furthermore, from the receiver's point of view, any mutable structure member's value may be omitted. For non-optional members this may be due to the sender having a compatible type that does not have that member. In such a case, if the receiver's member is not optional, it logically takes its default value. If the member is optional, it takes no value at all.

When encoding data using Parameterized CDR, the encoded data shall contain all non-optional members, even if a member value matches the default value for the member. This is because decoding of the data representation may be done for a different (assignable) data type. Therefore the data may be decoded assuming a different default value for the omitted member.

Objects of final or appendable structures are not serialized as full parameter lists, even if some members are optional. Therefore, the members of these types may not be omitted or reordered.

Because union members are identified based on a discriminator value, the value of the discriminator member must be serialized before the value of the current non-discriminator member. The discriminator value shall not be omitted.

7.4.1.2.4 Nested Objects

In the case where an object of an aggregated mutable type contains another object of an aggregated mutable type, one parameter list will contain another. In that case, parameter IDs are interpreted relative to the innermost type definition. (For instance, a type `Foo` may contain an instance of type `Bar`. Both `Foo` and `Bar` may define a member with ID 5. Inside the parameter list corresponding to the `Bar` object, an occurrence of parameter ID 5 shall be considered to refer to `Bar`'s member 5, not to `Foo`'s member 5.)

Likewise, an occurrence of `PID_LIST_END` indicates the conclusion of the innermost parameter list.

7.4.1.3 Issues and Limitations

For legacy backward-compatibility reasons, XCDR version 1 uses the PLAIN_CDR encoding for all Appendable and Final types (see 7.4.1.1).

Because the PLAIN_CDR encoding does not encode a delimiter header (DHEADER, see 7.4.3.4.1), a receiving implementation cannot skip unknown members.

As a result, XCDR version 1 cannot satisfy the type-assignability requirements of Appendable types defined in 7.2.5.5. Specifically, it cannot ensure correct assignability when Appendable types evolve by adding or removing members at the end of the type.

The only exception is when the Appendable type is used as the top-level Topic type, in which case the DDSI-RTPS encapsulation includes the Encapsulation Header that delimits the serialized payload. This allows limited interoperability for top-level Appendable types across revisions, but not for Appendable types used as nested or member-level constructs.

Therefore, interoperability of evolved Appendable types requires the use of XCDR version 2 (see 7.4.2), which provides a delimiter header and alignment rules that enable assignability-compliant evolution.

Conformance Requirement

XTYPES implementations shall detect the situation in which XCDR version 1 is used and the types being compared for assignability are Aggregate types declared as Appendable and are not used as the top-level Topic type.

In this case, the implementation shall treat those types as having FINAL extensibility for the purposes of type-assignability checking (see 7.2.5.5).

This ensures consistent assignability semantics across all conformant implementations and prevents erroneous interpretation of received data.

7.4.2 Extended CDR Representation (encoding version 2)

This specification defines three encoding formats used with encoding version 2: PLAIN_CDR2, DELIMITED_CDR, and PL_CDR2.

The three encoding formats leverage the PLAIN_CDR encoding. They enhance the encodings used in version 1 to improve type assignability and reduce the size of serialized data.

- PLAIN_CDR2 shall be used for all primitive, strings, and enumerated types. It is also used for any type with extensibility kind FINAL. The encoding is similar to PLAIN_CDR except that INT64, UINT64, FLOAT64, and FLOAT128 are serialized into the CDR buffer at offsets that are aligned to 4 rather than 8 as was the case in PLAIN_CDR.
- DELIMITED_CDR shall be used for types with extensibility kind APPENDABLE. It serializes a UINT32 delimiter header (DHEADER) before serializing the object using

PLAIN_CDR2. The delimiter encodes the endianness and the length of the serialized object that follows.

- PL_CDR2 shall be used for **aggregated types** with extensibility kind MUTABLE. Similar to DELIMITED_CDR it also serializes a DHEADER before serializing the object. In addition, it serializes a member header (EMHEADER) ahead each serialized member. The member header encodes the member ID, the must-understand flag, and length of the serialized member that follows.

7.4.3 Extended CDR encoding virtual machine

The encoding formats are specified using a virtual machine that acts on an XCDR stream object. The XCDR stream holds the bytes resulting from the incremental serialization of data objects into the stream.

The XCDR stream model consists of:

- A linear byte buffer where the serialized objects are placed.
- A set of internal state variables that may affect the serialization of future objects serialized into the stream. See Table 33.
- A set of operations on the stream that modify the state variables. See Table 34.
- A "stream insertion" operation that serializes objects onto the stream with a format that depends on the object type, its composition, and the value of the state variables. The append operation is represented using the operator symbol "<<". See Table 34.

7.4.3.1 Encoding version and format

The encoding format is determined by the encoding version and the extensibility kind of the object being serialized. Table 32 specifies the format that shall be used in each case.

Table 32 – Serialization format to use.

Extensibility Kind	*Encoding Version*	*Encoding format on the wire*
FINAL	1	PLAIN_CDR
FINAL	2	PLAIN_CDR2
APPENDABLE	1	PLAIN_CDR
APPENDABLE	2	DELIMITED_CDR
MUTABLE	1	PL_CDR
MUTABLE	2	PL_CDR2

7.4.3.2 XCDR Stream State

7.4.3.2.1 XCDR stream state variables

The state of the XCDR stream is described by the value of the variables (the XCDR state variables) defined in Table 33.

Table 33 – State variables and constants in the XCDR stream model

XCDR state variable	*meaning*
NENDIAN	Constant that represents the native endianness used by the system. It is dependent on the processor architecture, compiler, and operating system. There are two possible values: LITTLE_ENDIAN and BIG_ENDIAN
cendian	Choice variable representing the current endianness. This is the endianness that will be used to serialize subsequent objects into the stream. It affects integer types, floating-point types, enumerated types, and the Char16 type.
offset	Integer variable representing the offset into the byte stream where the next serialized byte will be placed. XCDR.offset is computed relative to the beginning of the stream so that XCDR.offset counts the number of bytes currently serialized into the stream. Each byte serialized into the stream causes XCDR.offset to be incremented.
origin	Integer state variable representing the offset into the stream used as the “logical beginning of the stream” for alignment operations. Each Type “T” has a default alignment (T.dalignment). This is the alignment used by default when an object of that type is serialized into a stream. An object O of type T shall be serialized at an offset that verifies: ((XCDR.offset - XCDR.origin) % T.dalignment) == 0

	If the current XCDR.offset does not satisfy the above condition, the serialization shall insert the minimum “padding bytes” needed to advance XCDR.offset so that the condition is met.
eversion	Octet state variable used to identify the version of the encoding rules used to serialize the stream. The pre-defined values are: {0x00} -- VERSION_NONE {0x01} -- VERSION1 {0x02} -- VERSION2
maxalign	Integer state variable representing the maximum value for the alignment that will be used for future objects serialized into the stream. This value overrides the required alignment for the object being serialized, so the alignment condition for any object O of type O.type becomes: ((XCDR.offset - XCDR.origin)% MALIGN(O))== 0 Where MALIGN(O) = MIN(O.type.alignment, XCDR.maxalign) This value is automatically set from the XCDR.eversion. XCDR.maxalign == MAXALIGN(XCDR.eversion)

7.4.3.2.2 Operations that change the XCDR stream state

The XCDR stream state is modified as a result of the serialization of data objects into the stream. It can also be modified as a result of performing the operations shown in Table 34.

Table 34 – Stream operations in the XCDR stream model

XCDR stream operation	*meaning*
INIT(V1=<nv1>, V2=<nv2>,…)	Initializes (constructs) the XCDR stream and sets the state variables V1, V2, … as specified. The notation <?> indicates that the value can be chosen by the implementation.
SET(VARIABLE=<newvalue>)	Sets the specified XCDR stream variable VARIABLE to <newvalue>. The notation <?> indicates that the new value can be chosen by the implementation.
SET(V1=<nv1>,V2=<nv2>,…)	A shortcut for calling SET() multiple times with the listed variables and new values.

MAXALIGN(<eversion>)	This operation returns the maximum alignment used for a given version of the encoding: MAXALIGN(VERSION2) = 4 MAXALIGN(VERSION1) = 8 MAXALIGN(VERSION_NONE) = 8
ALIGN(N)	This operation is used to advance the XCDR stream to achieve a desired alignment of the XCDR.offset. Advancing the XCDR.offset is done by inserting "padding bytes" into the stream. The value of the padded bytes is left unspecified. The actual number of bytes advanced depends not only on "N" but also on the value of the XCDR.maxalign. Specifically the stream is aligned to neededalign: neededalign = MIN(N, XCDR.maxalign) After the operation is performed the following condition shall be true: (XCDR.offset - XCDR.origin) % neededalign == 0
XCDR << { O : T }	The "append" stream operation. Serializes (using the Extended CDR representation) an object "O" of type "T" onto the XCDR stream starting at offset XCDR.offset.

7.4.3.2.3 XCDR Stream Initialization

The XCDR stream shall be initialized with an empty buffer.

The endianness shall be set as desired by the implementation, although a common setting for best performance is the native system endianness (NENDIAN).

The encoding version (eversion) shall be set as configured on the DataWriter. In this version of the DDS-XTypes specification it may be set to 1 or 2.

The first 2 octets in the XCDR stream shall be the Encapsulation Header (ENC_HEADER) indicating the endianness, encoding version, and encoding algorithm of the top-level type. See Table 36. This is the type associated with the DataWriter.

7.4.3.3 Type and Byte transformations

The operation of the serialization virtual machine uses a set of helper type and byte-buffer transformations.

The type transformations transform a type into another type, typically modifying its extensibility kind.

The byte-buffer transformations perform byte swaps in arrays of bytes or allow reinterpreting an object of a primitive type as an array of bytes.

These transformations are used to decompose the serialization of one type as a set of serializations of other types which have already been described.

Table 35 defines the type and byte transformations.

Table 35 – Type and Byte transformations used in the serialization virtual machine

Type or Object transformation	***meaning***
AsFinal(T) for any type T	This transformation only affects Aggregated types. For other types AsFinal(T) returns T. For the affected types AsFinal(T) is a new type which is declared the same as T except that its extensibility kind is FINAL.
AsNested(T) for any type T	This transformation treats the type as a Nested type for serialization purposes.
AsBytes(O) for any object O of a PRIMITIVE_TYPE	This transformation reinterprets the primitive object as an array of bytes. The resulting bytes are ordered as they appear in the processor memory according to the native Endianness (NENDIAN) used by the system.
ESWAP(B, <doit>) where B is a stream of 1, 2, 4, or 8 bytes	Conditionally swaps the bytes on the input stream B based on whether the current XCDR endianness (XCDR.cendian) matches the native Endianness (NENDIAN). This operation returns the same input stream if the input is a single byte or if XCDR.cendian == NENDIAN. Otherwise the operation produces a new stream of bytes with the same length as the input performing an (endianness) byte swapping according to the length of the input stream: For length 2: { B[1], B[0] } For length 4: { B[3], B[2], B[1], B[0] } For length 8: { B[7], B[6], B[5], B[4], B[3], B[2], B[1], B[0] }

7.4.3.4 Functions related to data types and objects

The operation of the serialization virtual machine uses a set of helper functions that return bytes or data to append to the XCDR stream. The notation and meaning is defined in Table 36.

Table 36 – Functions operating on objects and types

function	*meaning*
ENC_HEADER(<E>, <eversion>, T) for any type "T"	ENC_HEADER is an array of 2 octets used to identify the type of encoding (serialization), version of the encoding (<eversion>) and the endianness used by the stream (<E>). The possible values shall be one of the Encapsulation Identifiers defined in section 7.6.3.1.2 Table 57.
<OPTIONS>	<OPTIONS> is an array of 2 octets used to encode data-representation options. The <OPTIONS> are specific to each Data Representation. For Extended CDR version 1 and 2 they shall be set as specified in 7.6.3.1.2
EVERSION(T) for any type "T"	EVERSION is an octet used to identify the version of the encoding rules used to serialize the stream. The values are: 0x00 -- unspecified version (understood as version 1) 0x01 -- version 1 0x02 -- version 2
DHEADER(O) for any object O of type T	A UInt32 header value defined as: DHEADER(O) = O.ssize Where O.ssize is the number of bytes following the header that are required to hold the serialized representation of O.
EMHEADER1(M) Where M is a member of a structure	EMHEADER1 is the first 4 bytes of the Enhanced Mutable Header (EMHEADER) is used by the PL_CDR2 encoding format. It is a UINT32 value computed as: EMHEADER1 = (M_FLAG<<31) + (LC<<28) + M.id Where: M_FLAG is the value of the Must Understand option for the member LC is the value of the Length Code for the member.

LC(M) Where M is a member of a structure	LC is a 3-bit length code used to construct the EMHEADER1. It determines whether EMHEADER header has an additional 4 bytes (the NEXTINT) and is also used to encode the serialized size of the member that follows.
NEXTINT(M) Where M is a member of a structure	NEXTINT is the second 4 bytes of the Enhanced Mutable Header (EMHEADER). It is a UInt32 value. NEXTINT is only present if LC(M)>=4. NEXTINT is used in combination with LC to encode the serialized size of the member that follows.

7.4.3.4.1 Delimiter Header (DHEADER)

The DELIMITED_CDR and PL_CDR encoding formats prepend a UInt32 delimiter header (DHEADER) ahead of the serialization of the object content.

The DHEADER encodes the serialized size of the object that follows (not including the DHEADER itself). It is defined as:

```
DHEADER(O) = O.ssize
```

In this expression, O.ssize is constrained to being smaller than 4 Giga Bytes (2^32 Bytes).

The serialization of the DHEADER being a Uint32 type forces a 4-byte alignment relative to XCDR.origin, this may insert into the stream up to 3 padding bytes prior to the DHEADER.

The serialization of the DHEADER uses the endianness active in the XCDR stream at the time it is serialized (XCDR.cendian).

7.4.3.4.2 Member Header (EMHEADER), Length Code (LC) and NEXTINT

The PL_CDR2 encoding format serializes aggregated types using a member-by-member Type-Length encoding.

A member header precedes the serialization of each member. The member header can be either 4 or 8 bytes.

The first four bytes are the serialized representation of a UInt32 integer called EMHEADER1. EMHEADER1 shall be serialized using the XCDR stream endianness current at the place the serialization occurs (XCDR.cendian).

The second 4 bytes, if present, are the serialized representation of a UINT32 integer called NEXTINT. It shall be serialized with the same endianness as EMHEADER1.

EMHEADER1 is constructed from three parts: The must understand flag (M_FLAG), the length code (LC) and the member ID.

```
EMHEADER1 = (M_FLAG << 31) + (LC << 28) + (MemberId & 0x0fffffff)
```

The must understand flag (M_FLAG) shall be set to 1 if the corresponding member must be understood by the receiver, see Clause 7.2.3.2.4.4.6. Otherwise it shall be set to zero.

The length code provides the means to determine the serialized size of the member. There are eight possible values from 0 to 7 both included (0b000 to 0b111 in binary). These are interpreted as follows:

- LC values between 0 and 3 indicate that the member header is 4 bytes. That is, there is no NEXTINT. The value of LC encodes the length of the serialized member directly:
    - LC = 0 = 0b000 indicates serialized member length is 1 Byte
    - LC = 1 = 0b001 indicates serialized member length is 2 Bytes
    - LC = 2 = 0b010 indicates serialized member length is 4 Bytes
    - LC = 3 = 0b011 indicates serialized member length is 8 Bytes
- LC values between 4 and 7 indicate that the member header is 8 bytes. That is, a second integer (NEXTINT) immediately follows EMHEADER1. The value of LC combined with the value NEXTINT encode the length of the serialized member:
    - LC = 4 = 0b100 indicates serialized member length is NEXTINT
    - LC = 5 = 0b101 indicates serialized member length is also NEXTINT
    - LC = 6 = 0b110 indicates serialized member length is 4*NEXTINT
    - LC = 7 = 0b111 indicates serialized member length is 8*NEXTINT

EMHEADER1 with LC values 5 to 7 also affect the serialization/deserialization virtual machine in that they cause NEXTINT to be reused also as part of the serialized member. This is useful because the serialization of certain members also starts with an integer length, which would take exactly the same value as NEXTINT. Therefore the use of length codes 5 to 7 saves 4 bytes in the serialization.

7.4.3.5 Encoding (serialization) rules

The logic of the virtual machine is expressed as a collection of rules. Each rule has the form:

XCDR[vv] “<<” <match criteria> “=” XCDR “<<” <serialization action1>

“<<” <serialization action2>

“<<” ...

XCDR represents the stream containing the serialization of an object. It has a state represented by its state variables (see Clause 7.4.3.1) and it also holds the bytes from previously serialized objects. The [vv] indicates the encoding version that the DataWriter uses. This is configured on each DataWriter. A stream has its encoding version set when it is initialized and it cannot be modified.

A rule with left hand side XCDR[vv] only applies if the XCDR stream is using encoding version vv. A rule with left hand side XCDR applies for all xtypes encoding versions.

The **<match criteria>** represents the object that is being serialized into the XCDR stream.

When serializing an object each rule is evaluated in sequence and the first one that has a matching version and criteria is applied.

The application of a rule consists of executing each one of the serialization actions. Each action may change state variables of the stream or indicate that new objects (or modifications to existing objects) shall be serialized. This may recursively trigger the application of new rules.

The rules shall be applied until completion. Once completed, the XCDR stream contains the serialized representation of the object that initiated the serialization.

The rules are written from the point of view of a writer that is constructing the RTPS SerializedData buffer to send. Therefore the entrypoint is a so-called "Top Level" type which indicates a non-nested type that can be published by a DDS DataWriter. This entry point ensures the XCDR stream includes the SerializedData encapsulation header required by the DDS-RTPS protocol. Other entry points are possible if the intent is to simply serialize an object and not embed it within an RTPS SerializedData.

7.4.3.5.1 Notation used for the match criteria

Table 37 shows the symbols and notation used by the serialization virtual machine.

Table 37 – Symbols and notation used in the serialization virtual machine

notation	*meaning*
O : T	An object "O" of type "T" • O.type is another way to refer to the object type "T" • O.ssize is the size in bytes required to hold the serialized representation of O in an XCDR stream that has XCDR.offset aligned to the T.dalignment.
O : TOP_LEVEL_TYPE	An object O being serialized as the top-level Topic-Type. That is as the object written directly by a data-writer and not a nested object.
O : PRIMITIVE_TYPE	An object O of a primitive type as defined in 7.2.3.1.1.
O : STRING_TYPE	An object O of a string type which Char8 elements as defined in 7.2.3.2.2.1 O.ssize is one plus the string length as defined for the String<Char8> type (see 7.2.3.1.2.1). In other words, it is the number of bytes uses to represent the string when including the terminating NUL character.
O : WSTRING_TYPE	An object O of a string type with Char16 elements as defined in 7.2.3.2.2.1 O.ssize is twice the string length as defined for the String<Char16> type (see 7.2.3.1.2.2). In other words, it is twice the number of UTF-16 code units in the string, it does not including any terminating NUL character.

O: FIXED_DECIMAL_TYPE	An object O of a Fixed Point Decimal type as defined in 7.2.3.1.3.1
O : ENUM_TYPE	An object “O” of an Enumerated type as defined in 7.2.3.2.1 • O.holder_type is either Int8, Int16 or Int32 depending on the value of the @bit_bound annotation. • O.value is the (integer) value of the enumeration.
O : BITMASK_TYPE	An object O of a BitMask type as defined in 7.2.3.2.1.2 • O.holder_type is UInt8, UInt16, UInt32, or UInt64 depending on the value of the @bit_bound annotation. • O.value is the (integer) value of the bitmask.
O : ALIAS_TYPE	An object O of an Alias type as defined in 7.2.3.2.2 • O.base_type is the equivalent (aliased) type.
O : ARRAY_TYPE	An object “O” of an Array type as defined in 7.2.3.2.2.1 • O.element_type is the element type • O.length is the total number of elements in the array (accounting for all the dimensions) For single- dimensional arrays O[i] is the “ith” element in the array. Multi-dimensional arrays are treated for serialization purposes as a single dimensional array containing all the elements ordered such that the index of the first dimension varies most slowly, and the index of the last dimension varies most quickly.
O: FARRAY_TYPE	Same as ARRAY_TYPE except that its extensibility kind is FINAL.
O: PARRAY_TYPE	An ARRAY_TYPE whose element type is primitive.
O : SEQUENCE_TYPE	An object “O” of a Sequence type as defined in 7.2.3.2.2.1 • O.element_type is the element type • O.length is the number of elements in the sequence. Empty sequences have O.length==0 For non empty sequences O[i] is the “ith” element in the sequence. Sequence indices are zero-based so O[0] is the first element in the sequence and O[O.length-1] is the last element in the sequence.

O : PSEQUENCE_TYPE	Same as SEQUENCE_TYPE except that O.element_type is a primitive type. These sequences are intrinsically delimited in the sense that the CDR representation allows determining the serialized size of the entire sequence without iterating over each element.
O: FSEQUENCE_TYPE	Same as SEQUENCE_TYPE except that its extensibility kind is FINAL.
O : MAP_TYPE	An object "O" of a Map type as defined in 7.2.3.2.2.1 • O.key_type is the key type • O.element_type is the element type • O.length is the number of keys in the map, which is also the number of elements in the map. For non empty maps O[i].key is the "ith" key in the map, O[i].element is the (value) element that corresponds to that key. Map indices are zero-based so O[0].key is the first key in the map and O.key[O.length-1] is the last key in the map.
O : FMAP_TYPE	A MAP_TYPE whose extensibility kind is FINAL.
O : PMAP_TYPE	A MAP_TYPE whose element and key are primitive types.
O : UNION_TYPE	An object "O" of a Union type as defined in 7.2.3.2.4.3 • O.disc is the discriminator member. • O.disc.value is the value of the discriminator member. • O.disc.type is the type of the discriminator member. • O.selected_member is the member of the union selected based on the value of the discriminator. Note that certain discriminator values may select no member. • O.selected_member.value is the value of the selected member, if any. • O.selected_member.type is the type of the selected member.
O: FUNION_TYPE	Same as UNION_TYPE except that its extensibility kind is FINAL.

O : STRUCT_TYPE	An object “O” of a Struct type as defined in 7.2.3.2.4.2 • O.base_type is the type of the base Structure in case O.type inherits from another structure. • O.member_count is the number of members. For non empty structures: • O.member[i] is the “ith” member in the structure. It is a holder for the object that contains the value of the member and contains additional information. • Member indices are zero-based so O[0] is the first member. See definition of MEMBER.
O : FSTRUCT_TYPE	Same as STRUCT_TYPE except that its extensibility kind is FINAL.
O : MSTRUCT_TYPE	Same as STRUCT_TYPE except that its extensibility kind is MUTABLE. Unlike FSTRUCT_TYPE, O.member[i].id is the MemberId of O.member[i] as defined in 7.2.3.2.4.4 which may be different from “i”.
M : MEMBER	A member of an Aggregated type, 7.2.3.2.4. • M.id is the member ID. • M.value is the object holding the value of the member. • M.value.type is the type of the object. • M.value.ssize is the serialized size of the object holding the value of the member.
M : FMEMBER	A member (see MEMBER) of an Aggregated type that has extensibility kind FINAL.
M : OPT_FMEMBER	An optional member (see Clause 7.2.3.2.4.4.7) of an Aggregated type with extensibility kind final (FMEMBER).
M : NOPT_FMEMBER	A non-optional member (see Clause 7.2.3.2.4.4.7) of an Aggregated type with extensibility kind final (FMEMBER).
M : MMEMBER	A member (see MEMBER) of an Aggregated type that has extensibility kind MUTABLE.
O : FINAL_TYPE	An object O of a type with extensibility kind FINAL.
O : APPENDABLE_TYPE	An object O of a type with extensibility kind APPENDABLE. This is the default for collection types and structured types.

7.4.3.5.2 Encoding of Optional Members

PLAIN_CDR serializes optional members by prepending either a ShortMemberHeader or a 12 byte LongMemberHeader. See Clause 7.4.1.1.5.2. The associated size is set to zero if the optional member is not present or to the actual serialized size if the member is present. These headers are serialized at a 4-byte offset relative to the current stream origin (XCDR.origin) and adjust the alignment origin to zero for the serialization of the member itself.

PLAIN_CDR2 and DELIMITED_CDR serialize optional members by first serializing a boolean (<is_present>) that indicates whether the member is present or not. The serialized boolean shall be set to 0 if the member is not present and to 1 if it is. If the member present (<is_present> = 1) it shall be serialized following the <is_present> boolean. If it is not present, the member shall be omitted from the serialization.

PL_CDR and PL_CDR2 serialize optional members as it would with regular members except that if the optional member is not present, then the corresponding member header and serialized member are omitted from the serialized stream.

7.4.3.5.3 Complete Serialization Rules

```
(1)  XCDR << {O : TOP_LEVEL_TYPE}    =
                XCDR
                << INIT( OFFSET=0,   ORIGIN=0,
                         CENDIAN=<E>, EVERSION=<eversion> )
                << { ENC_HEADER(<E>, <eversion>, O.type) : Byte[2] }
                << SET( EVERSION = <eversion> )
                << SET( MAXALIGN = MAXALIGN(<eversion>) )
                << { <OPTIONS> : Byte[2] }
                << SET( ORIGIN = <offset> )
                << { O : AsNested(O.type) }

(2)  XCDR << {O : PRIMITIVE_TYPE}  =
                  XCDR
                    << ALIGN( O.ssize )
                    << ESWAP( AsBytes(O) )
```

```
(3)  XCDR << {O : STRING_TYPE}      =
                    XCDR
                      << { O.ssize : UInt32 }   // includes NUL
                      << { O[i] : Byte }*       // includes NUL

(4)  XCDR << {O : WSTRING_TYPE}   =
                    XCDR
                      << { O.ssize : UInt32 }   // No NUL
                      << { O[i] : Char16 }*     // No NUL

(5)  XCDR << {O : ENUM_TYPE}        =
                    XCDR
                      << { O.value : O.holder_type }

(6)  XCDR << {O : BITMASK_TYPE}   =
                    XCDR
                      << { O.value : O.holder_type }

(7)  XCDR << {O : ALIAS_TYPE}       =
                    XCDR
                      << { O      : O.base_type }

// Arrays of primitive element type (version 1 and 2 encoding)
(8)  XCDR    << {O : PARRAY_TYPE} =
                    XCDR
                      << { O[i] : O.element_type }*
```

```
// Arrays using version 2 encoding
(9) XCDR[2] << {O : ARRAY_TYPE}  =
                    XCDR
                      << { DHEADER(O) : UINT32 }
                      << { O[i] : O.element_type }*

// Arrays  using version 1 encoding
(10) XCDR[1] << {O : ARRAY_TYPE}  =
                    XCDR
                      << { O[i] : O.element_type }*

// Sequences of primitive element type (version 1 and 2 encoding)
(11) XCDR    << { O : PSEQUENCE_TYPE } =
                    XCDR
                      << { O.length : UInt32 }
                      << { O[i] : O.element_type }*

// Sequences using version 2 encoding
(12) XCDR[2] << {O : SEQUENCE_TYPE}  =
                    XCDR
                      << { DHEADER(O) : UINT32 }
                      << { O.length : UINT32 }
                      << { O[i] : O.element_type }*

// Sequences using version 1 encoding
(13) XCDR[1] << {O : SEQUENCE_TYPE}  =
                    XCDR
                      << { O.length : UInt32 }
                      << { O[i] : O.element_type }*
```

```
// Maps of primitive key and element type (version 1 and 2 encoding)
(14) XCDR << {O : PMAP_TYPE}  =
                    XCDR
                      << { O.length : UInt32 }
                      << { (O[i].key : O.key_type),
                           (O[i].element : O.element_type) }*

// Maps using version 2 encoding
(15) XCDR[2] << { O : MAP_TYPE }  =
                    XCDR
                       << { DHEADER(O) : UINT32 }
                       << { O.length : UINT32 }
                       << { (O[i].key : O.key_type),
                            (O[i].element : O.element_type) }*

// Maps using version 1 encoding
(16) XCDR[1] << {O : MAP_TYPE}  =
                    XCDR
                      << { O.length : UInt32 }
                      << { (O[i].key : O.key_type),
                           (O[i].element : O.element_type) }*

// Structures with extensibility FINAL (version 1 and 2 encoding)
// FMEMBER can be NOPT_FMEMBER (18) or OPT_FMEMBER (19)
(17) XCDR << {O : FSTRUCT_TYPE} =
                    XCDR
                      << { O.member[i] : FMEMBER }*

// Non-optional member of final Aggregated type (structure, union)
(18) XCDR << {M : NOPT_FMEMBER}   =
```

```
                XCDR
                  << { M.value : M.value.type }

// Optional member of final Aggregated type (structure, union), version 1
// see (26) and (27) for MMEMBER serialization
(19) XCDR[1] << {M : OPT_FMEMBER}    =
                XCDR
                  << { M : MMEMBER }

// Optional member of final aggregated type (structure, union), version 2
(20) XCDR[2] << {M : OPT_FMEMBER}  =
                XCDR
                  << { <is_present> :  BOOLEAN }
                  << IF (<is_present>) { M.value : M.value.type }

// Structures extensibility APPENDABLE handled by generic APPENDABLE rules:
// (29)-(30)

// Structures with extensibility MUTABLE, version 2 encoding
(21) XCDR[2] << {O : MSTRUCT_TYPE} =
                XCDR
                  << { DHEADER(O) : UInt32 }
                  << { O.member[i] : MMEMBER }*

// Member of mutable aggregated type (structure, union), version 2 encoding
(22) XCDR[2] << {M : MMEMBER}  =
                XCDR
                  << { EMHEADER1(M) : UInt32 }
                  << IF (LC(M)>=4) { NEXTINT(M)  : UInt32 }
                  << IF (LC(M)>=5)  XCDR.offset = XCDR.offset-4
```

```
                    << { M.value : M.value.type }

// Structures with extensibility MUTABLE, version 1 encoding
(23) XCDR[1] << {O : MSTRUCT_TYPE}  =
                    XCDR
                      << { O.member[i]  : MMEMBER }*
                      << ALIGN(4)
                      << { PID_LIST_END : UInt16 }
                      << { length = 0   : UInt16 }

// Member of mutable aggregated type (structure, union), version 1 encoding
// using short PL encoding when both M.id <= 2^14 and M.value.ssize <= 2^16
(24) XCDR[1] << {M : MMEMBER}  =
                    XCDR
                      << ALIGN(4)
                      << { FLAG_I + FLAG_M + M.id : UInt16 }
                      << { M.value.ssize          : UInt16 }
                      << SET( ORIGIN = <offset> )
                      << { M.value : M.value.type }

// Member of mutable aggregated type (structure, union), version 1 encoding
// using long PL encoding
(25) XCDR[1] << {M : MMEMBER}  =
                    XCDR
                      << ALIGN(4)
                      << { FLAG_I + FLAG_M + PID_EXTENDED : UInt16 }
                      << { slength=8     : UInt16 }
                      << { M.id          : UInt32 }
                      << { M.value.ssize : UInt32 }
                      << SET( ORIGIN = <offset> )
```

```
                    << { M.value : M.value.type }

// Unions with extensibility FINAL (version 1 and 2 encoding)
// see (18) to (20) for NOPT_FMEMBER and FMEMBER serialization
(26) XCDR    << {O : FUNION_TYPE}  =
                    XCDR
                      << { O.disc : NOPT_FMEMBER }
                      << { O.selected_member : FMEMBER }?

// Unions extensibility APPENDABLE handled by generic APPENDABLE rules:
// (29)-(30)

// Unions with extensibility MUTABLE, version 2 encoding
// see (22) for serialization of MMEMBER using version 2 encoding
(27) XCDR[2] << {O : MUNION_TYPE} =
                    XCDR
                      << { DHEADER(O) : UInt32 }
                      << { O.disc : MMEMBER }
                      << { O.selected_member : MMEMBER }?

// Unions with extensibility MUTABLE, version 1 encoding
// see (25)-(26) for serialization of MMEMBER using version 1 encoding
(28) XCDR[1] << {O : MUNION_TYPE}  =
                    XCDR
                      << { O.disc : MMEMBER }
                      << { O.selected_member : MMEMBER }?
                       << ALIGN(4)
                      << { PID_LIST_END : UInt16 }
                      << { length = 0   : UInt16 }
```

```
// Extensibility APPENDABLE (Aggregated types), version 1 encoding
(29) XCDR[1] << {O : APPENDABLE_TYPE} =
                XCDR
                  << { O : AsFinal(O.type) }
// Extensibility APPENDABLE (Aggregated types), version 2 encoding
(30) XCDR[2] << {O : APPENDABLE_TYPE} =
                XCDR
                  << { DHEADER(O) : UInt32 }
                  << { O : AsFinal(O.type) }
// Nibble is defined as a Half Octet, i.e. 4 bits.
// See 7.2.3.1.3 for the definition of O.precision and O.digits
// O.digit[i] represents each decimal digit of O starting with the most significant
(31) XCDR[1] << {O : FIXED_DECIMAL_TYPE} =
                XCDR
                  << IF (O.precision % 2 == 0) { 0x0  : Nibble }
                  << { O.digits[i]: Nibble } *      // for 1<= i <= O.precision
                  << IF (O.value >=  0) { 0xC  : Nibble }  ELSE { 0xD  : Nibble }
// See 7.2.3.1.3 for the definition of O.unscaled_value
(31) XCDR[2] << {O : FIXED_DECIMAL_TYPE} =
                XCDR
                  << IF (O.precision <= 2)          { O.unscaled_value  : Int8 }
                  << IF (2 < O.precision <= 4)      { O.unscaled_value  : Int16 }
                  << IF (4 < O.precision <= 9)      { O.unscaled_value  : Int32 }
                  << IF (9 < O.precision <= 18)     { O.unscaled_value  : Int64 }
                  << IF (18 < O.precision <= 31) {
                        LET hi64 and lo64 be integer values satisfying:
                            (a)  0 <= lo64 < 2^64
                            (b)  -2^63 <= hi64 < 2^63 -1
                            (c) O.unscaled_value = hi64 × 2^64 + lo64
                       << { hi64 : Int64 }
```

<< (lo64 : Uint64 }

}

7.4.4 XML Data Representation

This clause defines the XML Data Representation for DDS data values. The XML Data Representation specifies how instances of DDS types are encoded as XML documents for storage, interchange, inspection, or processing using XML-based tools.

The XML Data Representation applies only to data values. DDS types are defined using any of the Type Representations specified in clause 7.3.

This specification defines two equivalent syntactic forms of the XML Data Representations:

- the Schema-Qualified XML Data Representation
- the Compact XML Data Representation

Both forms represent the same XML data model and logically conform to the same XML Data Validation Schema. They differ only in their use of XML namespaces and schema declarations.

The examples that follow make use of the following non-normative example type definitions:

```
module MyModule1 { module MyModule2 {
   @nested
   struct MyInnerStructure {
      long my_integer;
   };
   struct MyStructure {
      MyInnerStructure inner;
      sequence<double> my_sequence_of_doubles;
   };
}}
```

7.4.4.1 Schema Qualified XML Data Representation

The Schema-Qualified XML Data Representation of a data-value explicitly declares the XML namespaces and schema location information required to validate the document against the XML Data Validation Schema.

In the event that the XML Data Validation Schema of the sample's type does not specify an explicit target namespace, the modules that scope that type shall imply the namespace for the document. This implied namespace shall take the form `ddstype://www.omg.org/<`*`module path`*`>`, where `<`*`module path`*`>` is a list of enclosing modules, separated by forward slashes, from outermost to innermost. The namespace prefix is not specified.

For example, the Schema-Qualified XML Data Representation of an object of the example type defined above would be as follows:

```
<my:MyStructure xmlns:my="ddstype://www.omg.org/MyModule1/MyModule2">
   <my:inner>
      <my:my_integer>5<my:my_integer>
   </my:inner>
   <my:my_sequence_of_doubles>
      <my:item>10.0</my:item>
      <my:item>20.0</my:item>
      <my:item>30.0</my:item>
   </my:my_sequence_of_doubles>
</my:MyStructure>
```

7.4.4.2 Compact XML Data Representation

The Compact XML Data Representation of a data-value shall be the same as the Schema-Qualified Data Representation escept that it shall *not* declare or use the namespace(s) against which it may be validated, regardless of whether a target namespace was specified in the XML Data Validation Schema of the corresponding sample's type. In other words, the document shall be well-formed but not valid. This limitation allows the document to be more compact in cases where the namespace is not needed or can be inferred by the recipient.

For example, the Compact XML Data Representation of an object of the example type defined above would be as follows:

```
<MyStructure>
   <inner>
      <my_integer>5<my_integer>
   </inner>
   <my_sequence_of_doubles>
      <item>10.0</item>
      <item>20.0</item>
      <item>30.0</item>
   </my_sequence_of_doubles>
</MyStructure>
```

Non-normative note: Schema-Qualified XML data representation can be nearly as small as the Compact XML data presentation by using a default namespace. The syntax to select the default namespace is xmlns=”ddstype://www.omg.org/…”. No prefix is necessary at every element name as they now default to the default namespace. That said, for really small datatypes (e.g., a 2d point) even the overhead of including the default namespace may be non-trivial. In such cases, the Compact XML data presentation may be preferred.

7.4.4.3 Mapping of DDS Data Values to XML

The XML Data Representation encodes the value of a DDS data instance in XML according to the structure of its DDS type.

The mapping is defined in [DDS-XML], Clause 7.2 “XML Representation of Resources Defined in the DDS IDL PSM”. The main aspects are summarized below:

- Primitive types are represented using XML simple content.
- Structures are represented as XML elements containing child elements corresponding to the structure members.
- Unions are represented by encoding the discriminator value together with the value of the selected union member.
- Sequences and arrays are represented as ordered collections of elements.
- Maps are represented as collections of key/value entries.

The XML Data Representation preserves the logical value of the DDS data instance. Type-system metadata such as keys, extensibility, and other type and member annotations are not encoded in the XML Data Representation.

7.4.4.4 XML Data Validation Schemas

To support validation of XML-encoded DDS data values of specific types it is necessary to define a set of XML Schema (XSD) definitions, referred to as XML Data Validation Schemas tailored to those data types.

The XML Data Validation Schemas:

- are derived from DDS type definitions,
- constrain the structure of XML Data Representations of data values, and
- may be used to validate Valid XML Data Representations using standard XML Schema validation mechanisms.

The construction of XML Data Validation Schemas from the type definitions is described in [DDS-XML], Clause 7.3.3 “Building Block Types”.

Note that XML Data Validation Schemas are not a mechanism for defining DDS types. They do not introduce new type semantics and shall not be used as an authoritative source of DDS type information. In fact, the XML Data Validation Schemas intentionally omit type-definition metadata and semantics as this is only relevant to DDS type definitions and not the serialized representation of data values.

7.5 Language Binding

The Language Binding Module specifies the alternative programming-language mechanisms an application can use to construct and introspect types as well as objects of those types. These mechanisms include a Dynamic API that allows an application to interact with types and data without compile-time knowledge of the type. Note that language-specific PSMs might overrule some or all of the language binding rules specified below.

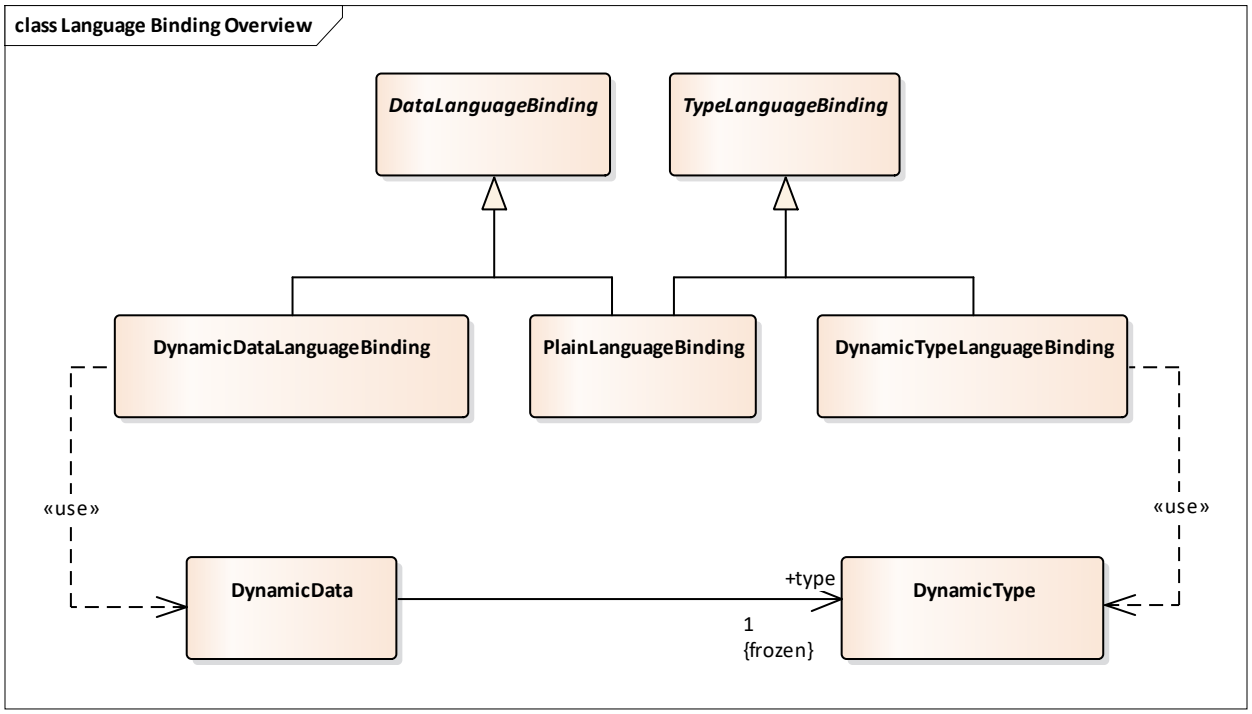

Figure 26 – Language Bindings—conceptual model

The specification defines two language bindings: Plain Language Objects and Dynamic Data. The main characteristics and motivation for each of these bindings are described in Table 38.

The Type Language Binding provides an API to manipulate types. This includes constructing new types as well as introspecting existing types. The API is the same regardless of the type, allowing applications to manipulate types that were not known at compile time. This API is similar in purpose to the `java.lang.Class` class in Java.

The principal mechanism to interact with a Type is the `DynamicType` interface. This interface is described in Clause 7.5.

Table 38 – Kinds of Language Bindings

Data Representation	*Description*	*Reasons and drawbacks*
Plain Language Binding	Each data type is mapped into the most natural "native" construct in the programming language of choice.	Advantages: • Natural - Well integrated in the programming language • Very compact notation

	For example a STRUCT type is mapped into a class in Java where each member of the STRUCT appears as a field in the class.	• Very efficient Disadvantages • Requires compile-time knowledge of the data type • Changes require recompilation • Support for type evolution and sparse data can be cumbersome
Dynamic Language Binding	All data types are mapped into a single Language "Dynamic Data" construct which contains operations to do introspection and access the data within.	Advantages: • Does not require compile-time knowledge of the data type • Does not require code-generation • Well suited for type evolution and sparse data Disadvantages • No compile-time checking • More cumbersome to use than plain data objects • May be lower performance to use than plain data objects

7.5.1 Plain Language Binding

This mapping reuses the OMG-standard IDL language mappings [C-MAP, C++-MAP, JAVA-MAP]. It extends the most commonly used of these bindings in order to express the extended IDL constructs defined in this specification.

The following steps define this language binding in all supported programming language for a particular type.

1. First, express the type in IDL as specified in Clause 7.3.1.
2. Then, apply the OMG Standard IDL to Language Mapping to the IDL in step 2.

3. Finally, apply any programming language-specific transformations to the generated code, if applicable. These transformations are defined below.

Note that any of the following language bindings may be overridden in a language-specific PSM, such as [DDS-PSM-CXX].

7.5.1.1 Primitive Types

7.5.1.1.1 C

The Service shall provide `typedef`s with the following names to types available on the underlying platform that have the appropriate sizes and representations.

Programmers concerned with DDS portability should use the Plain Language Binding types in Table 39 below. However, some may feel that using these types impairs readability. Therefore, compliant implementations have the following degrees of freedom:

- On platforms where a native C type (e.g. `int`) is guaranteed to be identical to a DDS type, the implementation may generate the equivalent native C type.
- On platforms compliant with the C99 specification, the implementation may generate equivalent C99-compatible types.

These degrees of freedom are not expected to impact code portability, as all of these `typedef`s will map to the same underlying native C types.

Table 39 – Plain Language Binding for Primitive Types in C

DDS Type	*Plain Language Binding Type*	*Equivalent C99 Type*
`Int32`	`DDS_Int32`	`int32_t`
`UInt32`	`DDS_UInt32`	`uint32_t`
`Int8`	`DDS_Int8`	`int8_t`
`UInt8`	`DDS_UInt8`	`uint8_t`
`Int16`	`DDS_Int16`	`int16_t`
`UInt16`	`DDS_UInt16`	`uint16_t`
`Int64`	`DDS_Int64`	`int64_t`
`UInt64`	`DDS_UInt64`	`uint64_t`
`Float32`	`DDS_Float32`	(*unspecified*)
`Float64`	`DDS_Float64`	(*unspecified*)
`Float128`	`DDS_Float128`	(*unspecified*)
`Char8`	`DDS_Char8`	(*unspecified*)
`Char16`	`DDS_Char16`	(*unspecified*)
`Boolean`	`DDS_Boolean`	`_Bool`

Byte	DDS_Byte	(*unspecified*)

With respect to DDS::Boolean, only the values 0 and 1 are defined. Other values result in unspecified behavior.

With respect to DDS::Char16, compliant implementations may consider wchar_t to be an equivalent C type if the platform supports it and it is of sufficient size. Otherwise, they may map Char16 to an equivalent integer type.

7.5.1.1.2 C++

The Service shall provide typedefs with the following names to types available on the underlying platform that have the appropriate sizes and representations.

Programmers concerned with DDS portability should use the Plain Language Binding types in Table 40 below. However, some may feel that using these types impairs readability. Therefore, compliant implementations have the following degrees of freedom:

- On platforms where a native C++ type (e.g. int) is guaranteed to be identical to a DDS type, the implementation may generate the equivalent native C++ type.
- On platforms compliant with the C99 specification, the implementation may generate equivalent C99-compatible types.

Table 40 – Plain Language Binding for Primitive Types in C++

DDS Type	*Plain Language Binding Type*	*Equivalent C99 Type*
Int32	DDS::Int32	[std::]int32_t
UInt32	DDS::UInt32	[std::]uint32_t
Int8	DDS::Int8	[std::]int8_t
UInt8	DDS::UInt8	[std::]uint8_t
Int16	DDS::Int16	[std::]int16_t
UInt16	DDS::UInt16	[std::]uint16_t
Int64	DDS::Int64	[std::]int64_t
UInt64	DDS::UInt64	[std::]uint64_t
Float32	DDS::Float32	(*unspecified*)
Float64	DDS::Float64	(*unspecified*)
Float128	DDS::Float128	(*unspecified*)
Char8	DDS::Char8	(*unspecified*)
Char16	DDS::Char16	(*unspecified*)
Boolean	DDS::Boolean	bool *or* _Bool

`Byte`	`DDS::Byte`	(*unspecified*)

With respect to `DDS::Boolean`, only the values 0 and 1 are defined. Alternatively, the C++ keywords true and false may be used. Other values result in unspecified behavior.

With respect to `DDS::Char16`, compliant implementations may consider `wchar_t` to be an equivalent C++ type if the platform supports it and it is of sufficient size. Otherwise, they may map `Char16` to an equivalent integer type. This means that `DDS::Char16` may not be distinguishable from integer types for purposes of overloading.

Types `DDS::Boolean`, `DDS::Char8`, and `DDS::Byte` may all map to the same underlying C++ type. This means that these types may not be distinguishable for the purposes of overloading.

All other mappings for basic types shall be distinguishable for the purposes of overloading. That is, one can safely write overloaded C++ functions for `DDS::Int16`, `DDS::UInt16`, `DDS::Int32`, and so on.

7.5.1.2 Annotations and Built-in Annotations

IDL annotations, including the built-in annotations, impact the language binding as defined below.

7.5.1.2.1 Enumerated Literal Values

Literals in an enumerated type may be given explicit values, as defined in Clause 7.2.3.2.1. This addition to the language impacts the bindings for C, C++, and Java in the following ways.

7.5.1.2.1.1 C

The OMG-standard IDL mapping to C language [C-MAP] transforms an IDL enumeration into a series of `#define` directives, each corresponding to one of the literals in the enumeration. The values to which these definitions correspond shall be the actual values of the enumerated literals on which the definitions are based, whether implicitly or explicitly defined.

7.5.1.2.1.2 C++

The OMG-standard IDL mapping to C++ mapping [C++-MAP] transforms an IDL enumeration into a C++ enumeration. The C++ programming language supports custom values for enumerated literals. Therefore, for any enumerated literal in IDL that bears the Value annotation, the corresponding C++ enumerated literal definition shall be followed by an equals sign (‘=’) and the value of the data member of the annotation.

7.5.1.2.1.3 Java

The OMG-standard IDL mapping to Java [JAVA-MAP] uses the pre-Java 5 “type-safe enumeration” design pattern. The value of each IDL enumerated literal is given in a Java integer constant of the following form:

```
public static final int _<label> = <value>;
```

…where <*label*> is the name of the IDL constant and <*value*> is its numeric value. As per *this* specification, that numeric value shall be set according to the explicit or implicit value assigned according to the operative Type Representation.

7.5.1.2.2 Bitmask Types

The language binding for bitmask types is defined based on the language binding for enumerations.

For each bitmask type defining flags FLAG_0 through FLAG_*n*, the language binding shall be as if there was an enumeration definition like the following:

```
@bit_bound(<bit_bound_value>)
enum <TypeName>Bits {
   @value(1 << <flag_value_0>)
   FLAG_0,
   …
   @value(1 << <flag_value_n>)
   FLAG_n,
};
```

Furthermore, the language binding shall be as if there was a typedef like the following, used to represent collections of flags from the previously defined enumeration:

```
typedef <unsigned_integer_equivalent> <TypeName>;
```

…where the type <*unsigned_integer_equivalent*> is chosen based on the bound of the bitmask type as defined in Table 41 below.

Table 41 – Bit mask integer equivalents

Bitmask Bound	***Unsigned Integer Equivalent***
1–8	`uint8`
9–16	`unsigned short`
17–32	`unsigned long`
33–64	`unsigned long long`

For example, consider the following IDL definition:

```
@bit_bound(19)
bitmask MyFlags {
   FIRST_FLAG,
   @position(14)
   SECOND_FLAG,
```

```
   THIRD_FLAG,
};
```

The language binding shall be as if the previous definition were replaced by the following:

```
enum MyFlagsBits {
   @value(1 << 0)
   FIRST_FLAG,
   @value(1 << 14)
   SECOND_FLAG,
   @value(1 << 15)
   THIRD_FLAG,
};
typedef unsigned long MyFlags;
```

7.5.1.2.3 External Members

The storage for a member of an aggregated type may be declared to be external to the storage of the enclosing object of that type. This is desirable, for instance, when the memory for a member may already exist somewhere and an application wants to combine it with other members and publish it as a unit without making additional copies. Another use case is sharing the data associated with the member among members in different objects.

The language bindings for C, Traditional C++, C++ for the DDS-PSM-CXX, and Java are provided in the following sub clauses.

7.5.1.2.3.1 C

External members shall be represented using pointers. Specifically:

- String and wide string members are already represented using pointers, so the mappings for these members do not change. The same applies to aliases to string and wide string types.
- Other external members are mapped like non-external members except that a member of type *X* shall instead be mapped as type *pointer-to-X*. For example, `short` shall be replaced by `short*`.

The constructor/initializer of the enclosing object shall set the external member pointers to NULL.

The destructor of the enclosing object shall delete the objects referenced by non-NULL external member pointers. It is the responsibility of the application to set the external member pointers to NULL before destroying the enclosing object if they do not want to delete specific referenced objects.

The copy function of the enclosing object shall do a deep copy of the external members. If the destination external member is NULL it shall be allocated. If the destination external member is

not NULL it shall be filled with a copy of the source member (i.e., perform logically a recursive call to `copy(destination->pointer-to-X, source->pointer-to-X)`). If the (recursive call to the) copy operation of the external member fails, then the copy function of the containing object shall fail as well. This may happen when the destination member is not large enough to hold a copy of the source.

There may be an additional copy function that takes in arguments which allow the user to control the behavior of the copy operation. This additional copy function shall allow the user to choose whether a shallow or deep copy is made as well as whether any existing memory pointed by the member is reused, released, or replaced during the copy.

In the case that a shallow copy is made and the destination member is NULL then the destination member pointer will be set to the source member pointer.

In the case that a deep copy is made and the destination member pointer is NULL, memory for the destination member will be allocated and then copied into.

For the behaviors supported by the additional copy function when the destination member is not NULL, see Table 42.

Table 42 – Configurable behaviors of the copy function when destination is not NULL

Copy Type	***Action when destination member is not NULL***	***Description***
Shallow Copy	Replace	Destination will now point to the same memory address as source. The existing memory pointed to by destination is released before making the assignment.
	Release	Destination will now point to the same memory address as source. The existing memory pointed to by destination is released before making the assignment.

Deep Copy	Reuse	(Default) Try to reuse the existing memory to copy into. If the existing member is not large enough, this operation shall fail.
	Replace	Replace the destination member. Allocate new memory to copy into and replace the existing memory without releasing it. It is the application's responsibility to release the replaced memory.
	Release	Release the existing memory before allocating new memory to copy into.

7.5.1.2.3.1.1 External Optional Members

A member that is both external and optional shall be mapped as if it was just external. The difference is that it is valid for the member to be NULL when writing a sample containing this member. If the member is only external but not optional, then it is not allowed for the member to be NULL at the time of a write.

7.5.1.2.3.2 Traditional C++

This mapping extends the IDL to C++ language mapping defined in [C++-MAP].

External members shall be represented by any type that behaves similarly to a pointer (e.g., a plain pointer or a `_var` type). The chosen type must support the concept of being "unset." For example, a plain pointer is considered unset if its value is NULL.

- In cases where the non-external mapping already uses a type similar to a pointer, it shall remain unchanged.
- In cases where the non-external mapping uses a member of type `X`, `X` shall be replaced by `pointer-to-X`. For example, if plain pointers are used, short shall be replaced by `short*`.

The behavior of the constructor, destructor, and copy functions shall be the same as specified for C.

7.5.1.2.3.2.1 External Optional Members

A member that is both external and optional shall be mapped as if it was just external. The difference is that it is valid for the member to be unset when writing a sample containing this member. If the member is only external but not optional, then it is not allowed for the member to be unset at the time of a write.

7.5.1.2.3.3 Modern C++

This mapping extends the IDL to C++ language mapping defined in [DDS-PSM-CXX].

External members shall be represented as an instantiation of a template class `external<T>`, where `T` is the type of the external member. This is a "smart pointer" class that wraps a shared pointer, `ptr_` for automatic reference counting and a boolean `locked_` that controls the assignment behavior. The destruction of the object referenced by an external member is always managed by the underlying shared pointer.

The value of the `locked_` attribute dictates whether copying an external member performs a deep copy or shallow copy of the referenced member. It can also be used to prevent sharing of the referenced object. This control is useful in some situations, for example, to prevent sharing a reference to memory that belongs to a `DataReader` in a DDS application. See Sub Clauses 7.5.1.2.3.3.4 and 7.5.1.2.3.3.5 for details about the copy constructor and assignment operator.

The `locked_` attribute is set at the time the external member is constructed and cannot be modified. The `locked_` attribute can only be set to `true` when the shared pointer is set to a non-NULL value.

The `external<T>` class shall be generated inside of an appropriate namespace. In the case of [DDS-PSM-CXX], this namespace is `dds::core`.

```
namespace dds { namespace core {

template <typename T>

class external {

public:

    external();

    external(T* p, bool locked = false);

    external(shared_ptr<T> p);

    external(const external& other);

    ~external();

    external& operator=(const external& other);

    T& operator*();

    const T& operator*() const;

    T* get();

    const T* get() const;

    shared_ptr<T> get_shared_ptr();

    T* operator->();

    const T* operator->() const;

    bool operator==(const external<T>& other) const;

    bool operator!=(const external<T>& other) const;
```

```
    operator bool() const;
    bool is_locked() const;
    void lock();

private:
    shared_ptr<T> ptr_;
    bool locked_;
};
}} // namespace dds::core
```

7.5.1.2.3.3.1 Operation: Default Constructor

Create an empty `external<T>` object with an empty `ptr_` and `locked_` initialized to false.

7.5.1.2.3.3.2 Operation: Constructor from a T*

Create a new `external<T>` object referencing the provided managed object. The attribute `locked_` is set to false and `ptr_` is initialized with p.

Parameter `p` - The object for `ptr_` to manage.

Parameter `locked` - Whether or not the constructed `external<T>` should be locked. This is an optional parameter with a default value of false.

7.5.1.2.3.3.3 Operation: Constructor from a shared pointer to T object

Create a new `external<T>` object that references the same object managed by the specified shared pointer `p`. The attribute `locked_` is set to false and `ptr_` is initialized with p.

Parameter `p` - The `shared_ptr<T>` holding the `T*` reference that will be shared with the new `external<T>` object.

7.5.1.2.3.3.4 Operation: Copy Constructor

Creates an external object from an existing external object (other). The behavior of this operation depends on the value of the locked_ attribute of the existing external object (other).

- If `other.is_locked()` is false, then the new `external<T>` object shares the reference with other. In other words this operation will not create a `T` object, instead it will perform a shallow copy of `T*` pointer.
- If `other.is_locked()` is true, then a new `T` object is created and `ptr_` is initialized with a reference to the newly created `T` object. The contents of newly-allocated object are initialized with a copy from the contents of other. In other words this operation will create a new `T` object and do a deep copy.

Either way, the newly constructed `external<T>` object will have locked_ set to false.

Parameter `other` - The external object used to initialize the new constructed `external<T>` object.

7.5.1.2.3.5 Operation: Assignment Operator

Assigns an external object to another.

The behavior of this operation depends on the value of the `locked_` attribute both on the source of the copy as well as on the destination.

The behavior specified in Table 43 below shall be applied when assigning an `external<T>` object source to another `external<T>` object destination:

Table 43 – Behavior of assignment operator

Destination	***Destination***	***Source***	***Source***	***Behavior of assignment operator***
`locked_`	`ptr_`	`locked_`	`ptr_`	
TRUE	<any>	<any>	<any>	Error. Operation cannot be called when `destination.is_locked() == TRUE`
FALSE	<any>	<any>	EMPTY	The destination is reset. Result is `destination.ptr_` is EMPTY.
FALSE	EMPTY	TRUE	Not EMPTY	Create new object for `destination.ptr_` Perform deep copy from `source.ptr_` to `destination.ptr_`.
FALSE	Not EMPTY	TRUE	Not EMPTY	Reuse existing `destination.ptr_` Perform deep copy from `source.ptr_` into the existing `destination.ptr_`.
FALSE	<any>	FALSE	Not EMPTY	Perform shallow copy. The `destination.ptr_ == source.ptr_` Destination will reference same object as source

Parameter `other` - The external object whose contents are assigned to this external object.

7.5.1.2.3.6 Operation: Destructor

Destroy the external object. If `ptr_` is the last reference to the managed object, then the managed object will be released, otherwise the reference count will simply be decreased.

7.5.1.2.3.7 Operation: operator* (const and non-const versions)

Get a reference to the underlying managed object that `ptr_` points at.

7.5.1.2.3.8 Operation: get (const and non-const versions)

Obtains a pointer to the managed object.

7.5.1.2.3.3.9 Operation: get_shared_ptr

Obtains a shared pointer to the managed object.

7.5.1.2.3.3.10 Operation: operator-> (const and non-const versions)

Allows accessing members of the managed object.

7.5.1.2.3.3.11 Operation: operator==

Returns whether two external objects manage the same object or are both empty.

7.5.1.2.3.3.12 Operation: operator!=

Returns whether two external objects do not manage the same object.

7.5.1.2.3.3.13 Operation: operator bool

Checks if there is a managed object (is not NULL) or not (is NULL).

7.5.1.2.3.3.14 Operation: is_locked

Indicates whether this object is locked or not.

7.5.1.2.3.3.15 Operation: lock

Sets the `locked_` attribute to true. This prevents of the `external<T>` object from modifying the referenced `T` object. This means that future assignment operations to the `external<T>` object will fail and any copies from `external<T>` will be deep copies (i.e., not share a reference to the same underlying `T` object).

7.5.1.2.3.3.16 External Optional Members

A member that is both external and optional shall be mapped as if it was just external. The difference is that it is valid for `ptr_` to be empty when writing a sample containing this member. If the member is only external but not optional, then it is not allowed for `ptr_` to be empty at the time of a write.

7.5.1.2.3.4 Java

This mapping extends the IDL to Java language mapping defined in [JAVA-MAP].

External members shall be represented using object references. Since *all* objects are referred to by reference in Java, the mappings for external members of non-primitive types are identical to those of non-external members. For IDL types that map to Java primitive types, those Java primitive types shall be replaced by the corresponding object box types from the `java.lang` package. For example, `short` shall be replaced by `java.lang.Short`.

7.5.1.2.4 Optional Members

A member of an aggregated type may be declared to be optional, meaning that its value may be omitted from sample to sample of that type. This concept impacts the language bindings for C, C++, and Java in the following ways.

7.5.1.2.4.1 C

Optional members shall be represented using pointers. Specifically:

- String and wide string members are already represented using pointers, so the mappings for these members shall not change. The same shall apply to aliases to string and wide string types.
- Other optional members are mapped like non-optional members except that a member of type *X* shall instead be mapped as type *pointer-to-X*. For example, `short` shall be replaced by `short*`.

A `NULL` pointer shall indicate an omitted value.

7.5.1.2.4.2 C++

Optional members shall be represented using plain pointers rather than automatic values or smart pointers.

- In cases where the mapping of non-optional members already uses a plain pointer, it shall remain unchanged.
- In cases where the mapping of non-optional members uses a "`_var`" smart pointer, the `_var` type shall be replaced by the corresponding plain pointer type. For example, `MyType_var` is replaced by `MyType*`.
- In cases where the mapping of non-optional members uses an automatic member of type *X*, *X* shall be replaced by *pointer-to-X*. For example, `short` shall be replaced by `short*`.

A `NULL` pointer shall indicate an omitted value.

7.5.1.2.4.3 Java

Optional members shall be represented using object references. Since *all* objects are referred to by reference in Java, the mappings for optional members of non-primitive types are identical to those of non-optional members. For IDL types that map to Java primitive types, those Java primitive types shall be replaced by the corresponding object box types. For example, `short` shall be replaced by `java.lang.Short`.

A `null` pointer shall indicate an omitted value.

7.5.1.2.4.4 Optional Arrays in C and C++

Optional arrays having element type "T" shall be mapped to type *pointer-to-array-of-type-T* rather than to type *array-of-pointers-to-type-T*.

For example, the structure `MyStruct` containing an optional array of ten integers defined by the IDL:

```
// IDL declaration
struct MyStruct {
   @optional long array_member[10];
};
```

Should be mapped in C and C++ to the type:

```
// Mapping to C/C++
struct MyStruct {
   int32_t (*array_member)[10];
}
```

Without the parentheses, `array_member` is an array of ten `int32_t` pointers, rather than a pointer to an array of ten `int32_t` values.

7.5.1.2.5 Nested Types

An IDL compiler need not (although it may) generate `TypeSupport`, `DataReader`, or `DataWriter` classes for any nested type.

7.5.1.2.6 User-Defined Annotation Types

A type designer may define his or her own annotation types. The language bindings for these shall be as follows in Java. In programming languages that lack the concept of annotations, an implementation of this specification may choose to ignore user-defined annotations with respect to this language binding.

7.5.1.2.6.1 Java

Each user-defined IDL annotation type shall be represented by a corresponding Java annotation type. An IDL annotation type defining operations `op_1` through `op_`*`n`* shall be represented by the following Java annotation types:

```
public @interface <TypeName> {
   <op_1_type> <op_1_name>() [default <default>];
   ...
   <op_n_type> <op_n_name>() [default <default>];
}

public @interface <TypeName>Group {
   <TypeName>[] value();
}
```

The <*op_type*> shall be the Java type corresponding to the return type of the IDL operation. If a default value is specified for a given member, it shall be reflected in the Java definition. Otherwise, the Java definition shall have no default value.

A Java annotation type may itself be annotated (for example, by annotation types in the `java.lang.annotation` package). The presence or absence of any such annotations is undefined.

For each IDL element to which a single instance user-defined annotation is applied, the corresponding Java element shall be annotated with the Java annotation of the same name. For each IDL element to which multiple instances of the annotation are applied, the corresponding Java element shall be annotated with the generated annotation bearing the "Group" suffix; each application of the user-defined annotation shall correspond to a member of the array in the group.

7.5.1.3 Map Types

The language bindings for C, Traditional C++, C++ for the DDS-PSM-CXX, and Java are provided in the following sub clauses.

Implementations are only required to support keys of types `UInt32`, `UInt64`, and `String<Char8>`. Implementations may choose to support other key types; however, to reduce complexity, maps declared to use any other key type may not be declared as an anonymous type in the IDL. If a Type Representation compiler encounters an anonymous map with key type that it does not support, it shall fail with an error.

7.5.1.3.1 Operations

Map types support operations to create, delete, and manipulate their contents. These operations are described in Table 44 below. Each of the language bindings support logically equivalent operations which are further described below if they are not supported natively by the language.

Table 44 – Operations for map<KeyType, ElementType>

map<KeyType, elementType>		
Operations		
`new`		`map<KeyType, ElementType>`
`delete`		`void`
`initialize`		`void`
`finalize`		`void`
`copy`		`ReturnCode_t`
	`source`	`map<KeyType, ElementType>`
	`autogrow`	`Boolean`
`get_size`		`unsigned int`
`get_max_size`		`unsigned int`
`set_max_size`		`ReturnCode_t`

	max_size	unsigned int
clear		void
insert		ReturnCode_t
	key	KeyType
	element	ElementType
insert_or_assign		ReturnCode_t
	key	KeyType
	element	ElementType
erase		ReturnCode_t
	key	KeyType
get_first		ReturnCode_t
get_next		ReturnCode_t
	inout: entry	MapEntry
find_element		ElementType
	key	KeyType
find_entry		MapEntry
	key	KeyType
get_pair		Boolean
	entry	MapEntry
	out: key	KeyType
	out: element	ElementType

7.5.1.3.2 C

This mapping extends the IDL to C language mapping defined in [C-MAP].

Map types shall be represented as a collection of structures that contain a member of the key type followed by a member of the element type. A set of methods which create, delete and manipulate objects of the map type shall also be generated. The name of the map type is specified in this language binding.

7.5.1.3.2.1 Map Type Name

For maps whose key type is a Primitive Type the name of the map type shall be constructed by combining the key type name with the element type name. The combination shall follow the schema below:

```
[key_type][fully_qualified_element_type]Map
```

For example, the names of the maps with element type `Foo` for each of the three mandatory key types would be:

`StringFooMap`

`UInt32FooMap`

`UInt64FooMap`

The concrete language binding is not specified, implementers may choose any language binding (e.g., a structure or a sequence) as long as its name and operations comply with what is specified here.

For any type `T`, the declaration and implementation of the map types having element type `T` and key types `UInt32`, `UInt64`, and `String` shall be generated alongside the implementation code for element type `T`.

Note: each of the following operations except for new take the map to be operated on as the first parameter.

7.5.1.3.2.2 Operation: new

Allocate a new map. If this operation fails in an implementation-specific way, this operation shall return NULL.

7.5.1.3.2.3 Operation: delete

Delete the map and all of its contents.

7.5.1.3.2.4 Operation: initialize

Initialize the map. The initial size and capacity of the map shall be 0.

7.5.1.3.2.5 Operation: finalize

Finalize the map. The entries in the map will be deleted, and both the size and maximum size set to 0.

This is equivalent to calling `clear()` followed by `set_max(0)`.

7.5.1.3.2.6 Operation: copy

Overwrite the contents of this (destination) map with the contents of another (source) map. Any entries that are not present in the source map are erased from the destination map. The source map shall not be modified by this operation.

If the size of the source map is greater than the maximum size of the destination map, the behavior depends on the autogrow parameter. If autogrow is TRUE, the operation shall grow the maximum size of the destination map as needed. If autogrow is FALSE, the operation shall fail and return `DDS_RETCODE_PRECONDITION_NOT_MET`. In this case the destination map shall remain unchanged.

If the size of the source is less than the maximum size of the destination, then it is left to the implementation to decide whether the maximum size of the destination map is trimmed to match the source or left unchanged.

If this operation fails in an implementation-specific way, the operation shall return `DDS_RETCODE_ERROR`.

Parameter `source` – The map whose contents are to be copied. If this argument is NULL, the operation shall fail with `DDS_RETCODE_BAD_PARAMETER`.

Parameter `autogrow` – Controls the behavior in case the destination map `max_size` is insufficient to hold the source map.

7.5.1.3.2.7 Operation: get_size

Get the current size of the map. The size of the map is how many entries are currently present in the map.

7.5.1.3.2.8 Operation: get_max_size

Get the current maximum size of the map. The maximum size limits the number of entries the map may contain.

7.5.1.3.2.9 Operation: set_max_size

Set the maximum size of the map.

This operation shall fail with `DDS_RETCODE_ERROR` if it fails for any implementation-specific reason.

Parameter `max_size` – The new maximum size of the map. If the new `max_size` is less than the current size of the map, the operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`.

7.5.1.3.2.10 Operation: clear

Clear all of the entries from the map. The size of the map is set to 0 and the maximum size does not change.

7.5.1.3.2.11 Operation: insert

Insert a new entry into the map with the given key and element values. If the key already exists in the map, the operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`. If successful, the size shall be increased by 1. If inserting a new entry into the map would increase the size past the current maximum size, then this operation shall fail with `DDS_RETCODE_PRECONDITION_NOT_MET`.

This operation shall fail with `DDS_RETCODE_ERROR` if it fails for any implementation-specific reason.

Parameter `key` – The key value of the entry to insert. If this argument is NULL, this operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`. For keys with primitive types, this argument shall be generated as the type and not as a pointer to the primitive type.

Parameter `element` – The element value of the entry to insert. If this argument is NULL, this operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`. For elements with primitive types, this argument shall be generated as the type and not as a pointer to the primitive type.

7.5.1.3.2.12 Operation: insert_or_assign

Insert an entry into the map with the given key and element values. If the key already exists in the map, then the corresponding element shall be replaced. If the key value did not already exist in the map, then the entry shall be inserted with the same behavior specified for the insert operation.

This operation shall fail with `DDS_RETCODE_ERROR` if it fails for any implementation-specific reason.

Parameter `key` – The key value of the entry to insert. If this argument is NULL, this operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`. For keys with primitive types, this argument shall be generated as the type and not as a pointer to the primitive type.

Parameter `element` – The element value of the entry to insert. If this argument is NULL, this operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`. For elements with primitive types, this argument shall be generated as the type and not as a pointer to the primitive type.

7.5.1.3.2.13 Operation: erase

Remove the entry with the given key from the map. If successful, the size of the map shall be decreased by 1.

Parameter `key` – The key value of the entry to erase. If this argument is NULL, this operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`. For keys with primitive types, this argument shall be generated as the type and not as a pointer to the primitive type.

7.5.1.3.2.14 Operation: get_first

Retrieves a `MapEntry` referencing the first entry in the map. The returned `MapEntry` may be a sentinel if the map is empty.

7.5.1.3.2.15 Operation: get_next

Advance the `MapEntry` to the next entry in the Map. If the `MapEntry` was referencing the last entry, the `MapCursor` will be advanced to a sentinel and the operation will return FALSE, otherwise it will return TRUE.

7.5.1.3.2.16 Operation: find_element

Retrieve the element whose key matches the specified one from the map. If the key exists, then return the element corresponding to the key, otherwise return NULL.

Parameter `key` – The key value of the element to search for. If this argument is NULL, this operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`. For keys with primitive types, this argument shall be generated as the type and not as a pointer to the primitive type.

7.5.1.3.2.17 Operation: find_entry

Retrieve the `MapEntry` whose key matches the specified one from the map. If the key exists, then return a `MapEntry` referencing the entry (key and element), otherwise return a sentinel.

Parameter `key` – The key value of the element to search for. If this argument is NULL, this operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`. For keys with primitive types, this argument shall be generated as the type and not as a pointer to the primitive type.

7.5.1.3.2.18 Operation: get_pair

Retrieve the key and element associated with the `MapEntry`. If the `MapEntry` was a sentinel the operation will return FALSE, otherwise it will return TRUE and fill the output parameters with references to the key and element.

Parameter `entry` – The `MapEntry` whose key and element we wish to retrieve. If this argument is NULL, this operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`.

Parameter `key` (output) – The key value associated with the `MapEntry`. If this argument is NULL, this operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`.

Parameter `element` – The element value associated with the `MapEntry`. If this argument is NULL, this operation shall fail and return `DDS_RETCODE_BAD_PARAMETER`.

7.5.1.3.2.19 Example (Non-Normative)

For a `struct MyElementType` defined by the IDL:

```
// IDL definition
module MyModule {
   struct MyElementType {
       // ...members
   };
};
```

The following structures and operations should be generated for `map<unsigned long, MyElementType>`:

```
struct UInt32MyModule_MyElementTypeMapElement {
    uint32_t key;
    MyModule_MyElementType element;
};


typedef sequence<UInt32MyModule_MyElementTypeMapElement>
UInt32MyModule_MyElementTypeMap;


// Operations on UInt32MyModule_MyElementTypeMap
UInt32MyModule_MyElementTypeMap* UInt32BarMap_new();
void UInt32MyModule_MyElementTypeMap_delete(
```

```
        UInt32MyModule_MyElementTypeMap *map);
void UInt32MyModule_MyElementTypeMap_initialize(
        UInt32MyModule_MyElementTypeMap *map);
void UInt32MyModule_MyElementTypeMap_finalize(
        UInt32MyModule_MyElementTypeMap *map);
DDS_ReturnCode_t UInt32MyModule_MyElementTypeMap_copy(
        UInt32MyModule_MyElementTypeMap *map,
        UInt32MyModule_MyElementTypeMap *other,
        bool autogrow);
uint32_t UInt32MyModule_MyElementTypeMap_get_size(
        UInt32MyModule_MyElementTypeMap *map);
DDS_ReturnCode_t UInt32MyModule_MyElementTypeMap_set_size(
        UInt32MyModule_MyElementTypeMap *map,
        uint32_t size);
uint32_t UInt32MyModule_MyElementTypeMap_get_max_size();
DDS_ReturnCode_t UInt32MyModule_MyElementTypeMap_set_max_size(
        UInt32MyModule_MyElementTypeMap *map,
        uint32_t max_size);
void UInt32MyModule_MyElementTypeMap_clear();
DDS_ReturnCode_t UInt32MyModule_MyElementTypeMap_insert(
        UInt32MyModule_MyElementTypeMap *map,
        uint32_t key,
        MyModule_MyElementType *element);
DDS_ReturnCode_t UInt32MyModule_MyElementTypeMap_insert_or_assign(
        UInt32MyModule_MyElementTypeMap *map,
        uint32_t key,
        MyModule_MyElementType *element);
DDS_ReturnCode_t UInt32MyModule_MyElementTypeMap_erase(
        UInt32MyModule_MyElementTypeMap *map,
        uint32_t key);
MapEntry UInt32MyModule_MyElementTypeMap_get_first(
        UInt32MyModule_MyElementTypeMap *map);
bool UInt32MyModule_MyElementTypeMap_get_next(
```

```
        UInt32MyModule_MyElementTypeMap *map,
        MapEntry *entry);
MyElementType* UInt32MyModule_MyElementTypeMap_find_element(
        UInt32MyModule_MyElementTypeMap *map,
        uint32_t key );
MapEntry UInt32MyModule_MyElementTypeMap_find_entry(
        UInt32MyModule_MyElementTypeMap *map,
        uint32_t key );
bool UInt32MyModule_MyElementTypeMap_get_pair(
        UInt32MyModule_MyElementTypeMap *map,
        MapEntry *entry,
        uint32_t *key,
        MyElementType **element);
```

7.5.1.3.3 Traditional C++

This mapping extends the IDL to C++ language mapping defined in [C++-MAP].

This C++ language binding differs only slightly from the C language binding. Instead of a C structure with accompanying functions, C++ defines a class with methods.

7.5.1.3.3.1 Map Class Name and operations

The map class shall be named the same as the C structure, see Sub Clause 7.5.1.3.2, except that it is placed in the same namespace as the element type declaration.

For example, the XTYPES map with key of type `UInt32` and element type `MyElementType` belonging to module `MyModule` would be bound to the class:

```
namespace MyModule {
class UInt32MyElementTypeMap {
public:
    UInt32MyElementTypeMap();
    ~UInt32MyElementTypeMap();
    ReturnCode_t copy(
            const UInt32MyElementTypeMap &other,
            bool autogrow = true);
    uint32_t get_size() const;
    ReturnCode_t set_size(uint32_t size);
    uint32_t  get_max_size() const;
```

```
    ReturnCode_t set_max_size(uint32_t max_size);
    void clear();
    ReturnCode_t insert(
            uint32_t key,
            const MyElementType &element,
            bool replace = true );
    ReturnCode_t erase( uint32_t key );
    MapEntry get_first();
    bool get_next(MapEntry &entry);
    MyElementType* find_element( uint32_t key);
    MapEntry find_entry( uint32_t key );
    bool get_pair(
            const MapEntry &entry,
            uint32_t *key,
            MyElementType **element);
};
}
```

Refer to the C language binding for the behavior of each of the above methods, with the exceptions described below.

The C++ operation insert behaves as the C `insert()` if the `replace()` parameter is false and it behaves as the C `insert_or_assign()` if replace parameter is true.

7.5.1.3.4 Modern C++

This mapping extends the IDL to C++ language mapping defined in [DDS-PSM-CXX].

The Map type shall be bound to an instantiation of the `std::map` template. The C++ Standard [C++-LANG] defines the `std::map` container as follows:

```
namespace std {
   template<class Key,
         class T,
         class Compare   = less<Key>,
         class Allocator = allocator<pair<const Key,T> >
    > class map;
}
```

The `std::map` template shall be instantiated with the `K` class parameter being the C++ type corresponding to the key type and the `T` parameter is the C++ type corresponding to the element type.

When a map has keys of a string type, the Compare function shall operate on the character contents of the strings; it shall not operate on the strings' pointer values (as `std::less` does). The instantiations for the Compare and Allocator parameters are otherwise undefined and may or may not take their default values.

For example, the XTYPES map with key of type `UInt32` and element type `MyElementType` belonging to module `MyModule` would be bound to the following template instantiation:

```
std::map<uint32_t, MyModule::MyElementType *>
```

7.5.1.3.5 Java

An IDL map type shall be represented in Java by an implementation of the standard `java.util.Map` interface. The implementation class to be used is not defined, nor is it defined whether Java 5+ generic syntax should be used. (The OMG-standard IDL mapping to Java [JAVA-MAP] predates Java 5, and implementations of it may retain compatibility with earlier versions of Java.)

The key objects for such maps shall be of the Java type corresponding to the IDL key element type. The value objects shall be of the Java type corresponding to the IDL value element type. If either of these Java types is a primitive type, then the corresponding object box type (e.g., `java.lang.Integer` for `int`) shall be used in its place.

7.5.1.3.6 Other Programming Languages

In all languages for which no language-specific mapping is specified, the language binding for map types shall be based on the equivalent IDL definition given in 7.4.1.1.4.

7.5.1.4 Structure and Union Types

The Plain Language Binding for structure and union types shall correspond to the IDL language mappings for IDL structures and unions as amended below.

7.5.1.4.1 Inheritance

A structure type that inherits from another shall be represented as follows.

7.5.1.4.1.1 C++

The C++ `struct` corresponding to the subtype shall publicly inherit from the C++ `struct` corresponding to the supertype.

7.5.1.4.1.2 Java

The Java class corresponding to the subtype shall extend the Java class corresponding to the supertype.

7.5.1.4.1.3 Other Programming Languages

The language binding shall be generated as if an instance of the base type were the first member of the subtype with the name “parent,” as in the following equivalent IDL definition:

```
struct <struct_name> {

   <base_type_name> parent;

   // ... other members

};
```

7.5.2 Dynamic Language Binding

The Dynamic Type Language Binding provides an API to manipulate types. This includes constructing new types as well as introspecting existing types. The API is the same regardless of the Type, allowing applications to manipulate types that were not known at compile time. This API is similar in purpose to the `java.lang.Class` class in Java.

The Dynamic Data Language Binding provides an API to manipulate objects of any Type. This includes creating data objects, setting fields and getting fields, as well as accessing the Type associated with the data object. The API is the same regardless of the type of the object, allowing applications to manipulate data objects of types not known at compile time.

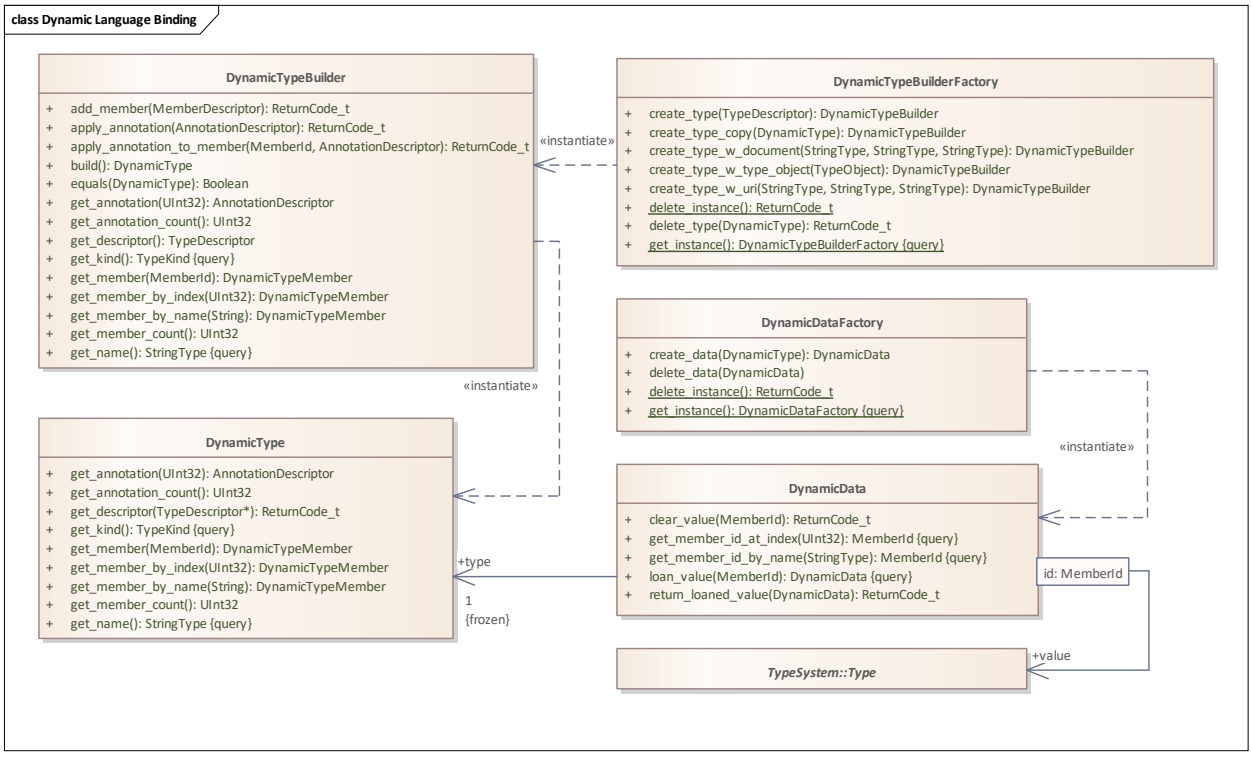

Figure 27 – Dynamic Data and Dynamic Type

There are a small number of fundamental classes to understand in this model, as well as a few helper classes:

- **DynamicType**: Objects of this class represent a type's schema: its physical name, type kind, member definitions (if any), and so on.
- **DynamicTypeBuilderFactory**: This type is logically a singleton. Its instance is responsible for creating `DynamicType` and `DynamicTypeSupport` objects.
- **DynamicData**: A `DynamicData` object represents an individual data sample. It provides reflective getters and setters for the members of that sample.
- **DynamicDataFactory**: This type is logically a singleton. Its instance is responsible for creating `DynamicData` objects.

7.5.2.1 UML-to-IDL Mapping Rules

Each type in this Language Binding has an equivalent IDL API. These APIs are specified using the IDL Type Representation defined in this document with the addition of other standard IDL syntax. These latter parts of IDL are used to describe portions of the UML model that have requirements that go beyond those addressed by the IDL Type Representation (for example, local operations).

Specifically, UML constructs shall be mapped to IDL as described below.

- UML enumerations are mapped to IDL enumerations.
- UML classifiers with value semantics are represented as IDL valuetypes. Classifiers with reference semantics are represented as local interfaces.
- UML structural properties in most cases are represented as IDL fields or attributes.
    - Properties of classifiers mapped as valuetypes are represented as plain fields. Properties of classifiers mapped as interfaces are represented as attributes; if the property value is read-only, so is the attribute.
    - Properties with multiplicity [1] (the default if not otherwise noted) are mapped as-is.
    - Properties with multiplicity [0..1] are defined as `@optional`.
    - Properties with multiplicity [*] (equivalent to [0..*]) or [1..*] may be mapped *either* simply as sequences (in cases where the number of objects is expected to be small and the required level of abstraction low) *or*—in more complex scenarios—a set of methods:

      ```
      unsigned long get_<property_name>_count();

      DDS::ReturnCode_t get_<property_name>(
            inout <property_type> value,
            in unsigned long idx);
      ```

      In addition, if and only if the property value can be modified:

      ```
      DDS::ReturnCode_t set_<property_name>(
      ```

```
      in unsigned long idx,
      in <property_type> value);
```

The "get" operation shall fail with `RETCODE_BAD_PARAMETER` if the given index is outside of the current range. The "set" operation shall do the same with one exception: it shall allow an index one past the end (i.e., equal to the current count); setting with this index shall have the effect of appending a new value to the end of the collection. Either operation shall fail with `RETCODE_BAD_PARAMETER` if either argument is nil.

Each type mapping below indicates which of these two mappings it uses in which cases.

- Qualified association ends (representing mappings from one value to another) are mapped to a set of operations:

```
DDS::ReturnCode_t get_<property_name>(
      inout <property_type> value,
      in <qualifier_type> key);
DDS::ReturnCode_t get_all_<property_name>(
      inout map< <qualifier_type>, <property_type> > value);
```

In addition, if and only if the property value can be modified:

```
DDS::ReturnCode_t set_<property_name>(
      in <qualifier_type> key,
      in <property_type> value);
```

The "get" operation shall return with `RETCODE_NO_DATA` if no value exists for the given key. Either operation shall return with `RETCODE_BAD_PARAMETER` if either argument is nil.

- UML operations are represented as IDL operations.
  - Static operations are commented, as IDL does not formally support static operations. It is up to the implementer to reflect these operations properly in each programming language to which the IDL may be transformed.

These rules may be qualified or overridden below on a case-by-case basis.

The complete IDL API can be found in "Annex C: Dynamic Language Binding."

7.5.2.2 DynamicTypeBuilderFactory

This class is logically a singleton (although it need not technically be a singleton in practice). Its "only" instance is the starting point for creating and deleting `DynamicTypeBuilder` objects.

Table 45 – `DynamicTypeBuilderFactory` properties and operations

DynamicTypeBuilderFactory

Operations		
static get_instance		DynamicTypeBuilderFactory
static delete_instance		ReturnCode_t
get_primitive_type		DynamicType
	kind	TypeKind
create_type		DynamicTypeBuilder
	descriptor	TypeDescriptor
create_type_copy		DynamicTypeBuilder
	type	DynamicType
create_type_w_type_object		DynamicTypeBuilder
	type_object	TypeObject
create_string_type		DynamicTypeBuilder
	bound	UInt32
create_wstring_type		DynamicTypeBuilder
	bound	UInt32
create_sequence_type		DynamicTypeBuilder
	element_type	DynamicType
	bound	UInt32
create_array_type		DynamicTypeBuilder
	element_type	DynamicType
	bound	UInt32 [1..*]
create_map_type		DynamicTypeBuilder
	key_element_type	DynamicType
	element_type	DynamicType
	bound	UInt32
create_bitmask_type		DynamicTypeBuilder
	bound	UInt32
create_type_w_uri		DynamicTypeBuilder
	document_url	string<Char8>
	type_name	string<Char8>
	include_paths	string<Char8> [*]

create_type_w_document		DynamicTypeBuilder
	document	string<Char8>
	type_name	string<Char8>
	include_paths	string<Char8> [*]
delete_type		ReturnCode_t
	type	DynamicType

7.5.2.2.1 Operation: `create_array_type`

Create and return a new `DynamicTypeBuilder` object representing an array type. All objects returned by this operation should eventually be deleted by calling `delete_type`.

All array types having equal element types, an equal number of dimensions, and equal bounds in each dimension shall be considered equal. An implementation may therefore elect whether to always return a new object from this method or whether to pool objects and to return previously created type objects consistent with these rules.

If an error occurs, this method shall return a nil value.

Parameter `element_type` – The type of all objects that can be stored in an array of the new type. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

Parameter **`bound`** - A collection of unsigned integers, the length of which is equal to the number of dimensions in the new array type, and the values of which are the bounds of each dimension. (For example, a three-by-two array would be described by a collection of length two, where the first element had a value of three and the second a value of two.) If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.2.2 Operation: `create_bitmask_type`

Create and return a new `DynamicTypeBuilder` object representing a bitmask type. All objects returned by this operation should eventually be deleted by calling `delete_type`.

If an error occurs, this method shall return a nil value.

Parameter **`bound`** - The number of reserved bits in the bitmask. If this value is out of range, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.2.3 Operation: `create_map_type`

Create and return a new `DynamicTypeBuilder` object representing a map type. All objects returned by this operation should eventually be deleted by calling `delete_type`.

All map types having equal key and value element types and equal bounds shall be considered equal. An implementation may therefore elect whether to always return a new object from this method or whether to pool objects and to return previously created type objects consistent with these rules.

If an error occurs, this method shall return a nil value.

Parameter `key_element_type` – The type of all objects that can be stored as keys in a map of the new type. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

Parameter `element_type` – The type of all objects that can be stored as values in a map of the new type. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

Parameter **`bound`** – The maximum number of key-value pairs that may be stored in a map of the new type. If this argument is equal to `LENGTH_UNLIMITED`, the map type shall be considered to be unbounded.

7.5.2.2.4 Operation: `create_sequence_type`

Create and return a new `DynamicTypeBuilder` object representing a sequence type. All objects returned by this operation should eventually be deleted by calling `delete_type`.

All sequence types having equal element types and equal bounds shall be considered equal. An implementation may therefore elect whether to always return a new object from this method or whether to pool objects and to return previously created type objects consistent with these rules.

If an error occurs, this method shall return a nil value.

Parameter `element_type` – The type of all objects that can be stored in a sequence of the new type. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

Parameter `bound` – The maximum number of elements that may be stored in a map of the new type. If this argument is equal to `LENGTH_UNLIMITED`, the sequence type shall be considered to be unbounded.

7.5.2.2.5 Operations: `create_string_type`, `create_wstring_type`

Create and return a new `DynamicTypeBuilder` object representing a string type. The element type of the result returned by `create_string_type` shall be `Char8`. The element type of the result returned by `create_wstring_type` shall be `Char16`.

All string types having equal element types and equal bounds shall be considered equal. An implementation may therefore elect whether to always return a new object from this method or whether to pool objects and to return previously created type objects consistent with these rules.

If an error occurs, this method shall return a nil value.

Parameter `bound` – The maximum number of elements that may be stored in a string of the new type. If this argument is equal to `LENGTH_UNLIMITED`, the string type shall be considered to be unbounded.

7.5.2.2.6 Operation: `create_type`

Create and return a new `DynamicTypeBuilder` object as described by the given type descriptor. This method is the conventional mechanism for creating structured, enumerated, and alias types, although it can also be used to create types of other kinds. All objects returned by this operation should eventually be deleted by calling `delete_type`.

Parameter `descriptor` – The properties of the new type to create. If this argument is nil or inconsistent (as indicated by its `is_consistent` operation), this operation shall fail and return a nil value.

7.5.2.2.7 Operation: `create_type_copy`

Create and return a new `DynamicTypeBuilder` object with a copy of the state of the given type. All objects returned by this operation should eventually be deleted by calling `delete_type`.

Parameter `type` – The initial state of the new type to create. If this argument is nil, this operation shall fail and return a nil value.

7.5.2.2.8 Operation: `create_type_w_type_object`

Create and return a new `DynamicTypeBuilder` object that describes a type identical to that described by the given `TypeObject` object. Subsequent changes to the new `DynamicTypeBuilder` object shall not be reflected in the input `TypeObject` object. All objects returned by this operation should eventually be deleted by calling `delete_type`.

Parameter `type_object` – The initial state of the new type to create.

7.5.2.2.9 Operation: `delete_instance`

Reclaim any resources associated with any object(s) previously returned from `get_instance`. Any references to these objects held by previous callers of this operation may become invalid at the discretion of the implementation.

This operation shall fail with `RETCODE_ERROR` if it fails for any implementation-specific reason.

7.5.2.2.10 Operation: `delete_type`

Delete the given `DynamicType` object, which was previously created by this factory.

Some “deletions” shall always succeed but shall have no observable effect:

- Deletions of nil
- Deletions of objects returned by `get_primitive_type`

Parameter `type` – The type to delete. If this argument is an object that was already deleted, and the implementation is able to detect that fact (which is not required), this operation shall fail with `RETCODE_ALREADY_DELETED`. If an implementation-specific error occurs, this method shall fail with `RETCODE_ERROR`.

7.5.2.2.11 Operation: `get_instance`

Return a `DynamicTypeBuilderFactory` instance that behaves like a singleton, although the caller cannot assume pointer equality for the results of multiple calls. The implementation may return the same object every time or different objects at its discretion. However, if it returns different objects, it shall ensure that they behave equivalently with respect to all programming interfaces specified in this document.

Calling this operation is legal even after `delete_instance` has been called. In such a case, the implementation shall recreate or restore the state of the "singleton" as necessary in order to return a valid object to the caller.

If an error occurs, this method shall return a nil value.

7.5.2.2.12 Operation: `get_primitive_type`

Retrieve a `DynamicType` object corresponding to the indicated primitive type kind.

The memory management regime underlying this method is unspecified. Implementations may return references to pre-created objects, they may return new objects with every invocation, or they may take an intermediate approach (for example, lazily creating but then caching objects). Whatever the implementation, the following invariants shall hold:

If an error occurs, this method shall return a nil value.

Parameter `kind` – The kind of the primitive type whose representation is to be returned. If the given kind does not correspond to a primitive type, the operation shall fail and return a nil value.

7.5.2.2.13 Operation: `create_type_w_uri`

Create and return a new `DynamicType` object by parsing the type description at the given URL.

Applications shall be able to reclaim resources associated with the type returned by this method by calling `delete_type`, just as if the resultant type was created by one of the `create` methods of this class.

If an error occurs, this method shall return a nil value.

Parameter `document_url` – A URL that indicates a type description document, which shall be parsed to create the `DynamicType` object. Implementations shall minimally support the `file://` URL scheme and may support additional schemes. Implementations shall minimally support the XML Type Description format for loaded documents and may support additional Type Descriptions. (Implementations are recommended to provide a tool or other means of translating among their supported Type Representations.)

Parameter `type_name` – The fully qualified name of the type to be loaded from the document that is the target of the URL. If no type exists of this name in the document (which will trivially be the case if the name is nil or the empty string), the operation shall fail and return a nil result.

Parameter `include_paths` – A collection of URLs to directories to be searched for additional type description documents that may be included, directly or indirectly, by the document that is the target of `document_url`. The directory in which the target of `document_url` resides shall be considered on the inclusion search path implicitly and need not be included in this collection. Implementations shall minimally support the `file:` URL scheme and may support additional schemes.

7.5.2.2.14 Operation: `create_type_w_document`

Create and return a new `DynamicType` object by parsing the type description contained in the given string.

Applications shall be able to reclaim resources associated with the type returned by this method by calling `delete_type`, just as if the resultant type was created by one of the `create` methods of this class.

If an error occurs, this method shall return a nil value.

Parameter `document` – A type description document, which shall be parsed to create the `DynamicType` object. Implementations shall minimally support the XML Type Description format for loaded documents and may support additional Type Descriptions. (Implementations are recommended to provide a tool or other means of translating among their supported Type Representations.)

Parameter `type_name` – The fully qualified name of the type to be loaded from the `document`. If no type exists of this name in the document (which will trivially be the case if the name is nil or the empty string), the operation shall fail and return a nil result.

Parameter `include_paths` – A collection of URLs to directories to be searched for additional type description documents that may be included, directly or indirectly, by the `document` argument. Implementations shall minimally support the `file://` URL scheme and may support additional schemes.

7.5.2.3 AnnotationDescriptor

An `AnnotationDescriptor` packages together the state of an annotation as it is applied to some element (not an annotation type). `AnnotationDescriptor` objects have value semantics, allowing them to be deeply copied and compared.

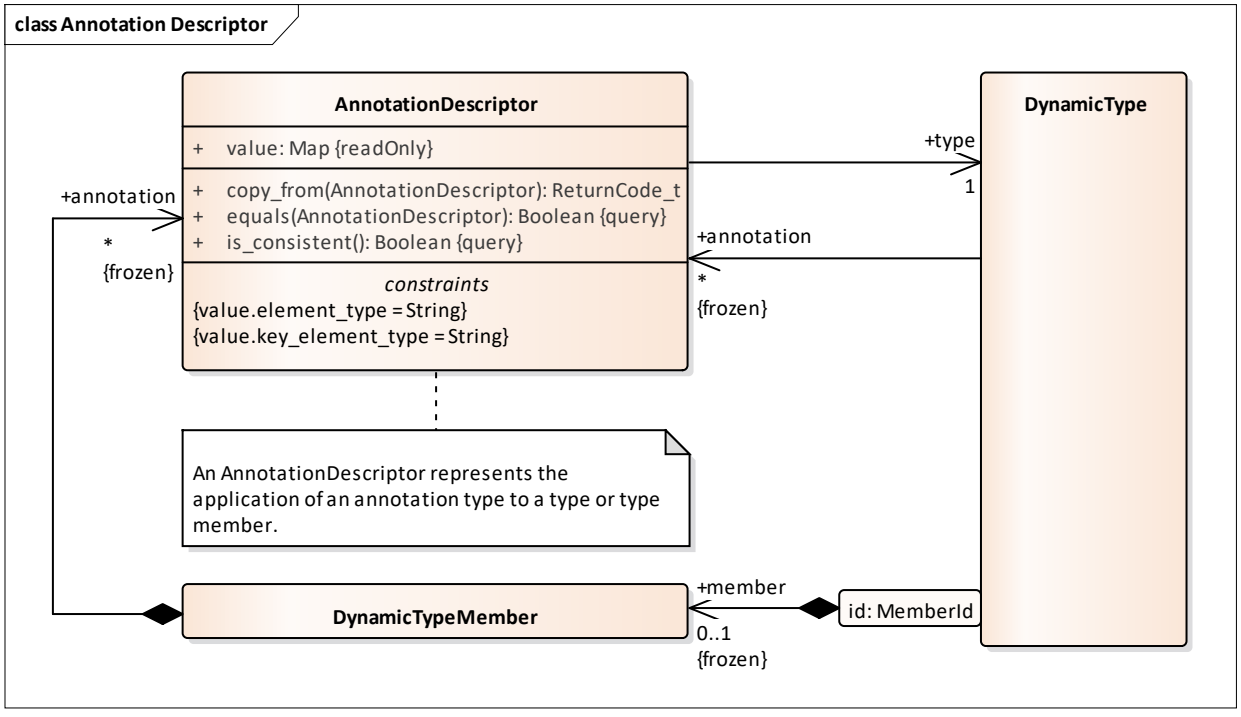

Figure 28 – Annotation Descriptor

Table 46 – AnnotationDescriptor properties and operations

AnnotationDescriptor		
Properties		
type	DynamicType	
value	Map<String<Char8,256>, String<Char8,256>>	
Operations		
copy_from		ReturnCode_t
	other	AnnotationDescriptor
equals		Boolean
	other	AnnotationDescriptor
is_consistent		Boolean

7.5.2.3.1 Operation: `copy_from`

Overwrite the contents of this descriptor with those of another descriptor such that subsequent calls to `equals`, passing the same argument as to this method, return `true`. The other descriptor shall not be changed by this operation.

If this operation fails in an implementation-specific way, this operation shall return `RETCODE_ERROR`.

Parameter `other` – The descriptor whose contents are to be copied. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.3.2 Operation: `equals`

Two annotation descriptors *ad1* and *ad2* are considered equal if and only if all of the following apply:

- Their type properties refer to equal types.
- For every string *s1* for which *ad1*`.value[`*s1*`]` does not exist, *ad2*`.value[`*s1*`]` also does not exist.
- For every string *s1* for which *ad2*`.value[`*s1*`]` does not exist, *ad1*`.value[`*s1*`]` also does not exist.
- For every string *s1* for which *ad1*`.value[`*s1*`]` is a non-nil string *ad1-s2*, *ad2*`.value[`*s1*`]` is a non-nil string *ad2-s2* such that *ad1-s2* equals *ad2-s2*.
- For every string *s1* for which *ad2*`.value[`*s1*`]` is a non-nil string *ad2-s2*, *ad1*`.value[`*s1*`]` is a non-nil string *ad1-s2* such that *ad1-s2* equals *ad2-s2*.

Parameter `other` – Another descriptor to compare to this descriptor. If this argument is nil, this operation shall return `false`.

7.5.2.3.3 Operation: `is_consistent`

Indicate whether this descriptor describes a valid annotation type instantiation. An annotation descriptor is considered consistent if and only if all of the following qualities apply:

- The `type` property refers to a non-nil type of kind `ANNOTATION_TYPE`.
- For every pair of strings *s1* and *s2* such that `value[`*s1*`]` equals `value[`*s2*`]`:
    - String *s1* is the name of an attribute defined by the annotation type referred to by the `type` property.
    - String *s2* is a well-formed string representation of an object of the type of the attribute named by *s1*.

7.5.2.3.4 Property: `type`

The `type` property contains a reference to the annotation type, of which this descriptor describes an instantiation.

When an annotation descriptor is newly created, this reference shall be nil.

7.5.2.3.5 Property: `value`

This property contains a mapping from the names of attributes defined by `type` to valid values of that type. Any attribute defined by `type` but for which no name appears in this property shall be considered to have its default value.

Every attribute value in this property is represented as a string although annotation type members can have other types as well. A string representation of a data value is considered well-formed if it would be a valid IDL literal of the corresponding type with the following qualifications:

- String and character literals shall not be surrounded by quotation characters (‘"’ or ‘'’).
- All expressions shall be fully evaluated such that no operators or other non-literal characters occur in the value. For example, “`5`” shall be considered a well-formed string representation of the integer quantity *five*, but “`2 + ENUM_VALUE_THREE`” shall not be.

7.5.2.4 TypeDescriptor

A `TypeDescriptor` packages together the state of a type. `TypeDescriptor` objects have value semantics, allowing them to be deeply copied and compared.

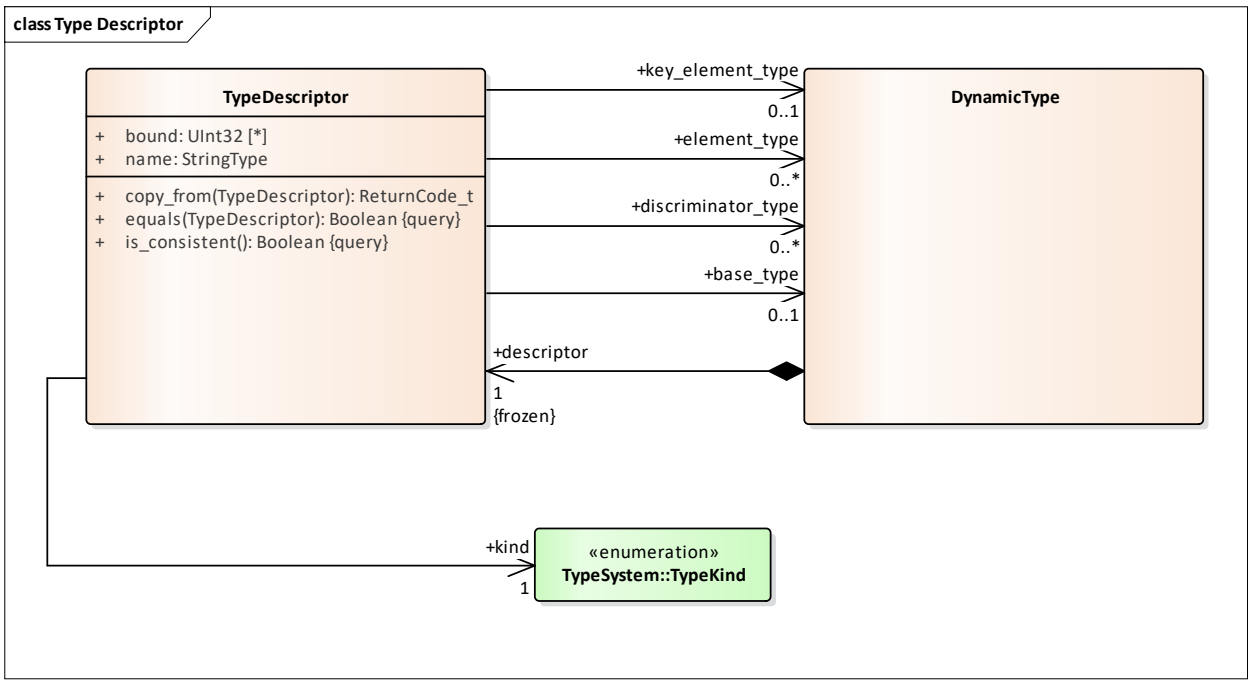

Figure 29 – Type Descriptor

Table 47 – TypeDescriptor properties and operations

TypeDescriptor		
Properties		
kind	TypeKind	
name	string<Char8,256>	
base_type	DynamicType [0..1]	
discriminator_type	DynamicType [0..1]	
bound	UInt32 [*]	
element_type	DynamicType [0..1]	
key_element_type	DynamicType [0..1]	
Operations		
copy_from		ReturnCode_t
	other	TypeDescriptor
equals		Boolean
	other	TypeDescriptor
is_consistent		Boolean

7.5.2.4.1 Property: `base_type`

Another type definition, on which the type described by this descriptor is based. Specifically:

- If this descriptor represents a structure type, `base_type` indicates the supertype of that type. A nil value of this property indicates that the structure type has no supertype.
- If this descriptor represents an alias type, `base_type` indicates the type being aliased. A nil value for this property is not considered consistent.

In all other cases, a consistent descriptor shall have a nil value for this property.

7.5.2.4.2 Property: `bound`

The `bound` property indicates the bound of collection and similar types.

- If this descriptor represents an array type, the length of the property value indicates the number of dimensions in the array, and each value indicates the bound of the corresponding dimension.
- If this descriptor represents a sequence, map, bitmask, or string type, the length of the property value is one and the integral value in that property indicates the bound of the collection.

In all other cases, a consistent descriptor shall have a nil value for this property.

7.5.2.4.3 Operation: `copy_from`

Overwrite the contents of this descriptor with those of another descriptor such that subsequent calls to `equals`, passing the same argument as to this method, return `true`. The other descriptor shall not be changed by this operation.

If this operation fails in an implementation-specific way, this operation shall return `RETCODE_ERROR`.

Parameter `other` – The descriptor whose contents are to be copied. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.4.4 Property: `discriminator_type`

If this descriptor represents a union type, `discriminator_type` indicates the type of the discriminator of the union. It must not be nil for the descriptor to be consistent.

If this descriptor represents any other kind of type, this property must be nil for this descriptor to be consistent.

7.5.2.4.5 Property: `element_type`

If this descriptor represents an array, sequence, or string type, this property indicates the element type of the collection. It must not be nil for the descriptor to be consistent.

If this descriptor represents a map type, this property indicates the *value* element type of the map. It must not be nil for the descriptor to be consistent.

If this descriptor represents a bitmask type, this property must indicate a Boolean type for the descriptor to be consistent.

If this descriptor represents any other kind of type, this property must be nil for the descriptor to be consistent.

7.5.2.4.6 Operation: `equals`

Two type descriptors are considered equal if and only if the values of all of the properties identified in Table 47 above are equal in each of them.

Parameter `other` – Another descriptor to compare to this one. If this argument is nil, the operation shall return `false`.

7.5.2.4.7 Operation: `is_consistent`

Indicates whether the states of all of this descriptor's properties are consistent. The definitions of consistency for each property are given in the clause corresponding to that property.

7.5.2.4.8 Property: `key_element_type`

If this descriptor represents a map type, this property indicates the *value* element type of the map. It must not be nil for the descriptor to be consistent.

If this descriptor represents any other kind of type, this property must be nil for the descriptor to be consistent.

7.5.2.4.9 Property: `kind`

An enumerated value that indicates what "kind" of type this descriptor describes: a structure, a sequence, etc.

7.5.2.4.10 Property: `name`

The fully qualified name of the type described by this descriptor. To be consistent, this name must be a valid identifier for the given type kind, as defined elsewhere in this document.

7.5.2.5 MemberId

The type `MemberId` is an alias to `UInt32` and is used for the purpose of representing the ID of a member of a structured type. The range of MemberId values is constrained as specified in 7.2.3.2.4.4.4.

It is also used to type the constant `MEMBER_ID_INVALID`, which is a sentinel indicating a member ID that is missing, irrelevant, or otherwise invalid in a given context.

7.5.2.6 DynamicTypeMember

A `DynamicTypeMember` represents a "member" of a type. A "member" in this sense may be a member of an aggregated type, a constant within an enumeration, or some other type substructure. Specifically, the behavior is as described in Table 48 below based on the `TypeKind` of the `DynamicType` to which the member belongs.

Table 48 – DynamicMember behavior

Type Kind	*Meaning*
`ANNOTATION_TYPE`	For these aggregated types, a "member" in this sense has the same meaning as it does in the definition of aggregated types generally.
`STRUCTURE_TYPE`	
`UNION_TYPE`	
`BITMASK_TYPE`	Each named flag in a bitmask shall be considered to be a "member" of that bitmask with `Boolean` type.
`ENUMERATION_TYPE`	Each literal in the enumeration shall be considered a "member" of the type. These members shall have the type of the enclosing enumeration itself.
`ALIAS_TYPE`	The behavior is as it would be for the alias's base type.

No other type kinds are considered to have members.

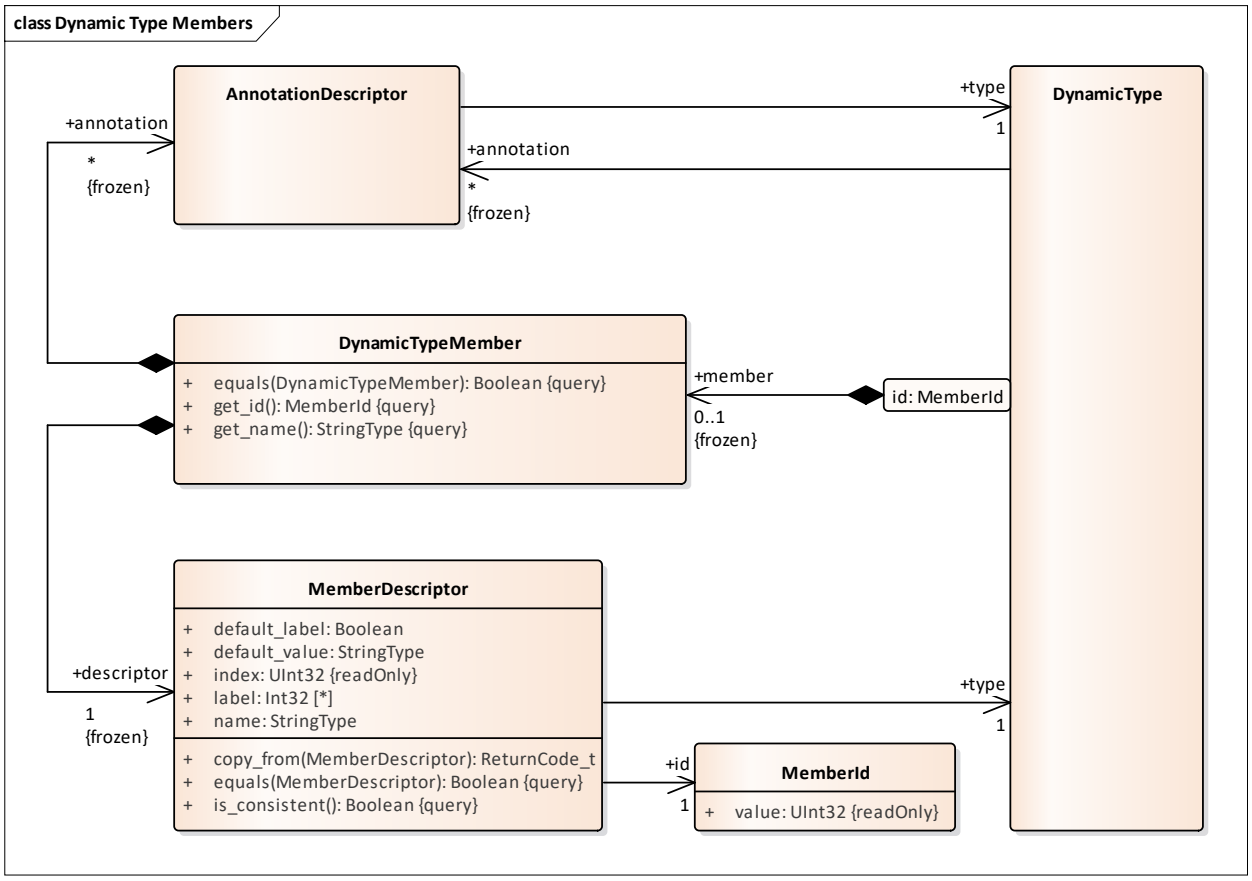

Figure 30 – Dynamic Type Members

`DynamicTypeMember` objects have reference semantics; however, there is an `equals` operation to allow them to be deeply compared.

Table 49 – DynamicTypeMember properties and operations

DynamicTypeMember		
Properties		
annotation	read-only AnnotationDescriptor [*]	
Operations		
get_descriptor		DDS::ReturnCode_t
	inout descriptor	MemberDescriptor
equals		Boolean
	other	DynamicTypeMember
get_name		string<Char8,256>
get_id		MemberId

7.5.2.6.1 Property: `annotation`

This property provides all annotations previously applied to this member.

7.5.2.6.2 Operation: `get_descriptor`

This operation provides a summary of the state of this type. It overwrites the state of an application-provided object.

If the argument is nil, this operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.6.3 Operation: `equals`

Two members shall be considered equal if and only if they belong to the same type and all of their respective properties, as identified in Table 49 above, are equal.

7.5.2.6.4 Operation: `get_id`

This convenience operation provides the member ID of this member. Its result shall be identical to the ID value that is a member of the `descriptor` property.

7.5.2.6.5 Operation: `get_name`

This convenience operation provides the name of this member. Its result shall be identical to the name string that is a member of the `descriptor` property.

7.5.2.7 MemberDescriptor

A `MemberDescriptor` packages together the state of a `DynamicTypeMember`. `MemberDescriptor` objects have value semantics, allowing them to be deeply copied and compared.

Table 50 – MemberDescriptor properties and operations

MemberDescriptor		
Properties		
name	String<Char8,256>	
id	MemberId	
type	DynamicType	
default_value	string	
index	read-only UInt32	
label	Int64 [*]	
default_label	Boolean	
Operations		
copy_from		ReturnCode_t
	other	MemberDescriptor
equals		Boolean
	other	MemberDescriptor
is_consistent		Boolean

7.5.2.7.1 Operation: `copy_from`

Overwrite the contents of this descriptor with those of another descriptor such that subsequent calls to `equals`, passing the same argument as to this method, return `true`. The other descriptor shall not be changed by this operation.

If this operation fails in an implementation-specific way, this operation shall return `RETCODE_ERROR`.

Parameter `other` – The descriptor whose contents are to be copied. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.7.2 Property: `default_label`

For this descriptor to be consistent, this property must be true if this descriptor identifies the default member of a union type or false if not. A default union member may have additional explicit labels (indicated in the `label` property), but these are semantically irrelevant, as the default member would be in effect or not regardless of their presence or absence.

7.5.2.7.3 Property: `default_value`

This property provides the member's default value in string form. A string representation of a data value is considered well-formed if it would be a valid IDL literal of the corresponding type with the following qualifications:

- String and character literals shall not be surrounded by quotation characters (‘"’ or ‘'’).
- All expressions shall be fully evaluated such that no operators or other non-literal characters occur in the value. For example, “`5`” shall be considered a well-formed string representation of the integer quantity *five*, but “`2 + ENUM_VALUE_THREE`” shall not be.

A nil or empty string indicates that the member takes the default value for its type (see Table 10). This rule shall *always* be used when the member is of a type for which IDL provides no syntax to express a literal value (for example, structures or maps) and *may* be used for any other type.

Design rationale: An instance of `DynamicData` might have been used here as an alternative. However, since every default literal can be expressed as a string anyway (i.e., as it is in IDL), and string objects are expected to be more lightweight that `DynamicData` implementations, that representation was preferred.

7.5.2.7.4 Operation: `equals`

Two descriptors are considered equal if and only if the values of all of the properties identified in Table 50 above are equal in each of them.

Parameter `other` – Another descriptor to compare to this one. If this argument is nil, the operation shall return `false`.

7.5.2.7.5 Property: `id`

If this member belongs to an aggregated type, this property indicates the member’s ID.

- When a descriptor is used to add a new member to a type, this property may be set to `MEMBER_ID_INVALID`; in that case, the implementation shall select an ID for the new member that is one more than the current maximum member ID in the type. If the value of this property is *not* `MEMBER_ID_INVALID`, it must be set to a value within a legal range.
- When a descriptor is retrieved from an existing member, this property shall reflect the actual ID of the member. It shall therefore not be `MEMBER_ID_INVALID`, and it shall fall within a legal range.

If this member does not belong to an aggregated type, this property *must* be `MEMBER_ID_INVALID`, or the descriptor is not consistent.

7.5.2.7.6 Property: `index`

This property indicates the order of definition of this member within its type, relative to the type’s other members. The first member shall have index zero, the next one, and so on.

When a descriptor is used to add a new member to a type, any value greater than the current largest index value in the type shall be taken to indicate that the new member will become the last member, whatever the index; member indices within a type shall not be discontiguous. Alternatively, if this property is set to an index at which a member already exists, that member and all those after it shall be shifted up by a single index value to make room for the new member.

When a descriptor is retrieved from an existing member, this property shall always reflect the actual index at which the member exists.

7.5.2.7.7 Operation: `is_consistent`

A descriptor shall be considered consistent if and only if all of the values of its properties are considered consistent. The meaning of consistency for each of these is defined here in the appropriate clause.

7.5.2.7.8 Property: `label`

If the type to which the member belongs is a union, this property indicates the case labels that apply to this member. If `default_label` is false, it must not be empty. In addition, no two members of the same union can specify the same label value.

If the type to which the member belongs is *not* a union, this property's value must be empty to be consistent.

7.5.2.7.9 Property: `name`

This property indicates the name of this member. The value must be a well-formed member name.

7.5.2.7.10 Property: `type`

This property indicates the type of the member's value. It must not be nil and must indicate a type that can legally type a member according to the Type System Model.

7.5.2.8 DynamicType

A `DynamicType` object represents a particular type defined according to the Type System. `DynamicType` objects have reference semantics because of the large number of references to them that are expected to exist (e.g., in each `DynamicData` object created from a given `DynamicType`). However, the type nevertheless provides operations to allow copying and comparison by value.

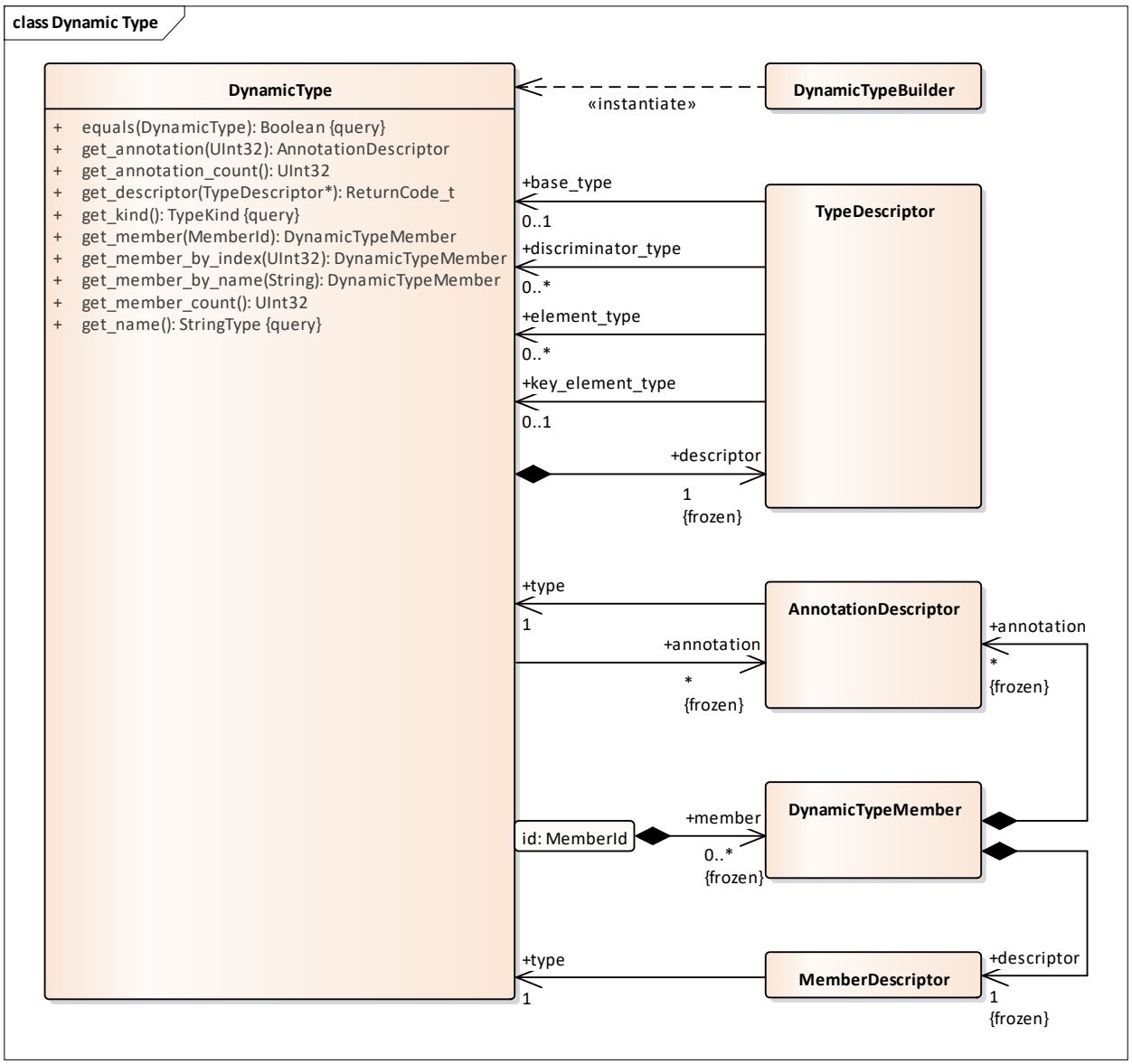

Figure 31 – Dynamic Type

Table 51 – DynamicType properties and operations

DynamicType	
Properties	
member_by_name	read-only string<Char8,256> → DynamicTypeMember [0..1]
member	read-only MemberId → DynamicTypeMember [0..1]
annotation	read-only AnnotationDescriptor [*]

Operations		
equals		Boolean
	other	DynamicType
get_ annotation		ReturnCode_t
	inout: descriptor	AnnotationDescriptor
	index	UInt32
get_annotation_count		UInt32
get_descriptor		ReturnCode_t
	inout descriptor	TypeDescriptor
get_kind		TypeKind
get_member		ReturnCode_t
	inout: member	DynamicTypeMember
	member_id	MemberId
get_member_by_index		ReturnCode_t
	inout: member	DynamicTypeMember
	index	UInt32
get_member_by_name		ReturnCode_t
	inout: member	DynamicTypeMember
	name	String
get_member_count		UInt32
get_name		string<Char8,256>

7.5.2.8.1 Property: `annotation`

This property provides all annotations that have previously been applied to this type.

7.5.2.8.2 Property: `member`

This property contains a mapping from the member ID of a member of this (aggregated) type to the member itself.

- If this type is an aggregated type, the collection of members available through this property shall be equal to (element order notwithstanding) that available through the `member_by_name` property.
- If this type is *not* an aggregated type, the collection of members available through this property shall be empty.

7.5.2.8.3 Property: `member_by_name`

This property contains a mapping from the name of a member of this type to the member itself. As described in Table 52 below, not only members of aggregated types are considered "members" here: the constituents of enumerations, bitmasks, and other kinds of types are also considered to be "members" for the purposes of this property.

Table 52 – DynamicType::member_by_name behavior

Type Kind	*Behavior*
`ANNOTATION_TYPE`	The member descriptor must describe a consistent annotation type member. If the descriptor does not satisfy these constraints, the operation shall fail with `RETCODE_BAD_PARAMETER`.
`ALIAS_TYPE`	The behavior is as it would be for the alias's base type. If adding a member is not defined for the alias's base type, this operation shall fail with `RETCODE_PRECONDITION_NOT_MET`.
`BITMASK_TYPE`	The member descriptor must describe a Boolean flag with a value within the bound of this bitmask type. If the descriptor does not satisfy these constraints, the operation shall fail with `RETCODE_BAD_PARAMETER`.
`ENUMERATION_TYPE`	The member descriptor must describe a literal with the type of this enumeration. If the descriptor does not satisfy these constraints, the operation shall fail with `RETCODE_BAD_PARAMETER`.
`STRUCTURE_TYPE`	The member descriptor must describe a consistent structure member. If the descriptor does not satisfy this constraint, the operation shall fail with `RETCODE_BAD_PARAMETER`.
`UNION_TYPE`	The member descriptor must describe a consistent union member. If the descriptor does not satisfy this constraint, the operation shall fail with `RETCODE_BAD_PARAMETER`.

The lifecycle of a `DynamicTypeMember` object is governed by that of the `DynamicType` that contains it. The former shall be considered to exist logically from the time the corresponding member is added to the latter and until such time as the latter is deleted. Implementations may allocate and de-allocate `DynamicTypeMember` objects more frequently, provided that:

- Users of the `DynamicTypeMember` class are not required to explicitly delete objects of that class.
- Changes to one `DynamicTypeMember` object representing a given member shall be reflected in all observable `DynamicTypeMember` objects representing the same member.
- All `DynamicTypeMember` objects representing the same member shall compare as equal according to their `equals` operations.

7.5.2.8.4 Operation: `equals`

Two types shall be considered equal if and only if all of their respective properties, as identified in Table 51 above, are equal.

7.5.2.8.5 Operation: `get_annotation`

This operation returns the annotation that corresponds to the specified index, if any.

The operation shall fail if the specified index is greater than the current annotation count. In this case it shall return `RETCODE_BAD_PARAMETER`.

7.5.2.8.6 Operation: `get_annotation_count`

This operation returns the current number of annotations applied to the type.

7.5.2.8.7 Operation: `get_descriptor`

This operation provides a summary of the state of this type. It overwrites the state of an application-provided object.

If the argument is nil, this operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.8.8 Operation: `get_kind`

This convenience operation indicates the kind of this type (e.g., integer, structure, etc.). Its result shall be the same as the kind indicated by the type's `descriptor` property.

7.5.2.8.9 Operation: `get_member`

This operation returns the member that corresponds to the specified member ID, if any.

If there is no member with the specified member ID, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.8.10 Operation: `get_member_by_index`

This operation returns the member that corresponds to the specified index, if any.

The operation shall fail if the specified index is greater than the current member count. In this case it shall return `RETCODE_BAD_PARAMETER`.

7.5.2.8.11 Operation: `get_member_by_name`

This operation returns the member that corresponds to the specified name, if any.

If there is no member with the specified name, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.8.12 Operation: `get_member_count`

This operation returns the current number of members.

7.5.2.8.13 Operation: `get_name`

This convenience operation provides the fully qualified name of this type. It shall be identical to the name string that is a member of the `descriptor` property.

7.5.2.9 DynamicTypeBuilder

A `DynamicTypeBuilder` object represents a transitional state of a particular type defined according to the Type System. It is used to instantiate concrete `DynamicType` objects.

Table 53 – DynamicTypeBuilder properties and operations

DynamicTypeBuilder		
Properties		
member_by_name	read-only string<Char8,256> → DynamicTypeMember [0..1]	
member	read-only MemberId → DynamicTypeMember [0..1]	
annotation	read-only AnnotationDescriptor [*]	
Operations		
add_member		ReturnCode_t
	descriptor	MemberDescriptor
apply_annotation		ReturnCode_t
	descriptor	AnnotationDescriptor
apply_annotation_to_member		ReturnCode_t
	member_id	MemberId
	descriptor	AnnotationDescriptor
build		DynamicType
equals		Boolean
	other	DynamicType
get_ annotation		ReturnCode_t
	inout: descriptor	AnnotationDescriptor
	index	UInt32
get_annotation_count		UInt32
get_descriptor		ReturnCode_t
	inout descriptor	TypeDescriptor
get_kind		TypeKind
get_member		ReturnCode_t

	inout: member	DynamicTypeMember
	member_id	MemberId
get_member_by_index		ReturnCode_t
	inout: member	DynamicTypeMember
	index	UInt32
get_member_by_name		ReturnCode_t
	inout: member	DynamicTypeMember
	name	String
get_member_count		UInt32
get_name		string<Char8,256>

7.5.2.9.1 Property: `annotation`

This property provides all annotations that have previously been applied to this type with `apply_annotation`.

7.5.2.9.2 Property: `member`

This property contains a mapping from the member ID of a member of this (aggregated) type to the member itself.

- If this type is an aggregated type, the collection of members available through this property shall be equal to (element order notwithstanding) that available through the `member_by_name` property.
- If this type is not an aggregated type, the collection of members available through this property shall be empty.

7.5.2.9.3 Property: `member_by_name`

This property contains a mapping from the name of a member of this type to the member itself. As described in the case of `add_member`, not only members of aggregated types are considered "members" here: the constituents of enumerations, bitmasks, and other kinds of types are also considered to be "members" for the purposes of this property.

The lifecycle of a `DynamicTypeMember` object is governed by that of the `DynamicTypeBuilder` that contains it. The former shall be considered to exist logically from the time the corresponding member is added to the latter and until such time as the latter is deleted. Implementations may allocate and de-allocate `DynamicTypeMember` objects more frequently, provided that:

- Users of the `DynamicTypeMember` class are not required to explicitly delete objects of that class.
- Changes to one `DynamicTypeMember` object representing a given member shall be reflected in all observable `DynamicTypeMember` objects representing the same member.

- All `DynamicTypeMember` objects representing the same member shall compare as equal according to their equals operations.

7.5.2.9.4 Operation: `add_member`

Add a "member" to this type, where the new "member" has the meaning defined in the specification of the `DynamicTypeMember` class. Specifically, the behavior shall be as described in Table 52 in Clause 7.5.2.8.3, "Property: `member_by_name`". For type kinds not given in that table, this operation shall fail with `RETCODE_PRECONDITION_NOT_MET`.

Following a successful return, the new member shall appear in the member property and possibly in the `member_by_id` property, based on the definition of that property.

Parameter `descriptor` – A descriptor of the new member to be added. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.9.5 Operation: `apply_annotation`

Apply the given annotation to this type. It shall subsequently appear in the `annotation` property.

Parameter `descriptor` – A consistent descriptor for the annotation to apply. If this argument is not consistent, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.9.6 Operation: `apply_annotation_to_member`

Apply the given annotation to this member. It shall subsequently appear in the `annotation` property of the identified member.

Parameter `member_id` – Identifies the member to which the annotation shall be applied.

Parameter `descriptor` – A consistent descriptor for the annotation to apply. If this argument is not consistent, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.9.7 Operation: `build`

Create an immutable `DynamicType` object containing a snapshot of this builder's current state. Subsequent changes to this builder, if any, shall have no observable effect on the states of any previously created `DynamicType`s.

7.5.2.9.8 Operation: `get_descriptor`

This operation provides a summary of the state of this type. It overwrites the state of an application-provided object.

If the argument is nil, this operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.9.9 Operation: `equals`

Two types shall be considered equal if and only if all of their respective properties, as identified in Table 53 above, are equal.

7.5.2.9.10 Operation: `get_annotation`

This operation returns the annotation that corresponds to the specified index, if any.

The operation shall fail if the specified index is greater than the current annotation count. In this case it shall return `RETCODE_BAD_PARAMETER`.

7.5.2.9.11 Operation: `get_annotation_count`

This operation returns the current number of annotations applied to the type.

7.5.2.9.12 Operation: `get_kind`

This convenience operation indicates the kind of this type (e.g., integer, structure, etc.). Its result shall be the same as the kind indicated by the type's descriptor property.

7.5.2.9.13 Operation: Operation: `get_member`

This operation returns the member that corresponds to the specified member ID, if any.

If there is no member with the specified member ID, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.9.14 Operation: `get_member_by_index`

This operation returns the member that corresponds to the specified index, if any.

The operation shall fail if the specified index is greater than the current member count. In this case it shall return `RETCODE_BAD_PARAMETER`.

7.5.2.9.15 Operation: `get_member_by_name`

This operation returns the member that corresponds to the specified name, if any.

If there is no member with the specified name, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.9.16 Operation: `get_member_count`

This operation returns the current number of members.

7.5.2.9.17 `get_name`

This convenience operation provides the fully qualified name of this type. It shall be identical to the name string that is a member of the descriptor property.

7.5.2.10 DynamicDataFactory

This class is logically a singleton (although it need not technically be a singleton in practice). Its "only" instance is the starting point for creating and deleting `DynamicData` and objects, just like the singleton `DomainParticipantFactory` is the starting point for creating `DomainParticipant` objects.

Table 54 – DynamicDataFactory properties and operations

DynamicDataFactory		
Operations		
static get_instance		DynamicDataFactory
static delete_instance		ReturnCode_t
create_data		DynamicData
	type	DynamicType
delete_data		ReturnCode_t
	data	DynamicData

7.5.2.10.1 Operation: `create_data`

Create and return a new data sample. All objects returned by this operation should eventually be deleted by calling `delete_data`.

Parameter `type` - The type of the sample to create.

7.5.2.10.2 Operation: `delete_data`

Dispose of a data sample, reclaiming any associated resources.

Parameter `data` - The data sample to delete.

7.5.2.10.3 Operation: `delete_instance`

Reclaim any resources associated with the object(s) previously returned from `get_instance`. Any references to these objects held by previous callers may become invalid at the implementation's discretion.

This operation shall return `RETCODE_ERROR` if it fails for any implementation-specific reason.

7.5.2.10.4 Operation: `get_instance`

Return a `DynamicDataFactory` instance that behaves like a singleton, although callers cannot assume pointer equality across invocations of this operation. The implementation may return the same object every time or different objects at its discretion. However, if it returns different objects, it shall ensure that they behave equivalently with respect to all programming interfaces specified in this document.

It is legal to call this operation even after `delete_instance` has been called. In such a case, the implementation shall recreate or restore the "singleton" as necessary to ensure that it can return a valid object to the caller.

If an error occurs, this method shall return a nil value.

7.5.2.11 DynamicData

Each object of the `DynamicData` class represents a corresponding object of the type represented by the `DynamicData` object's `DynamicType`.

`DynamicData` objects have reference semantics; however, there is an `equals` operation to allow them to be deeply compared.

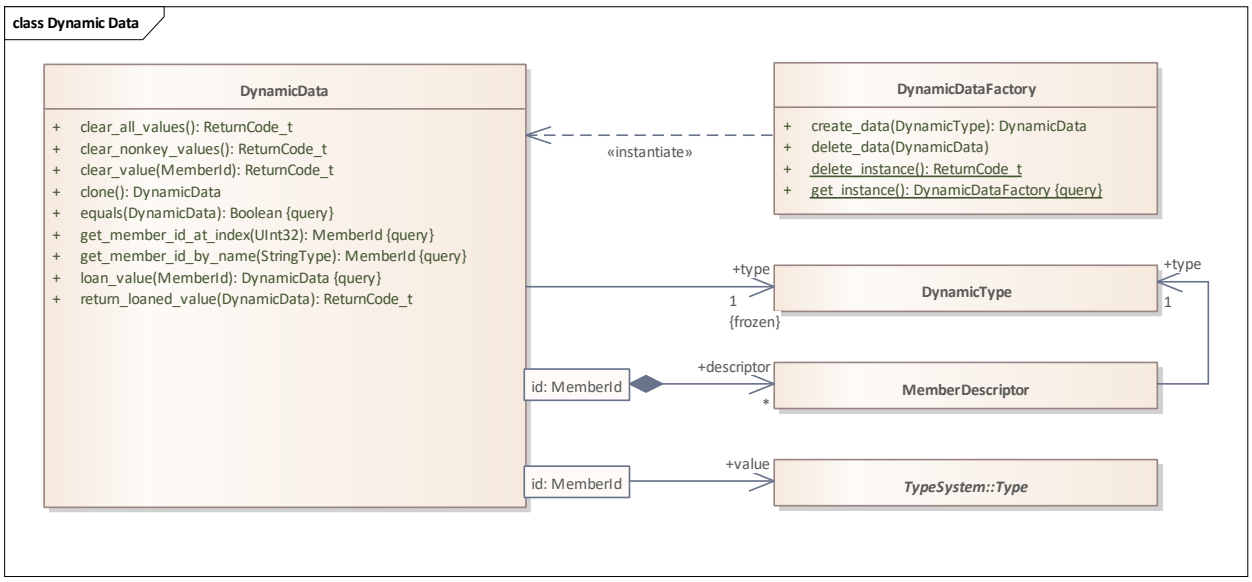

Figure 32 – Dynamic Data and Dynamic Data Factory

Table 55 below summarizes the properties and operations supported by `DynamicData` objects.

Table 55 – DynamicData properties and operations

DynamicData		
Properties		
value	MemberId → Type [0..1]	
type	read-only DynamicType	
descriptor	MemberId → MemberDescriptor	
Operations		
get_member_id_by_name		MemberId
	name	string<Char8,256>
get_member_id_at_index		MemberId
	index	UInt32
get_item_count		UInt32
equals		Boolean
	other	DynamicData

clear_all_values		ReturnCode_t
clear_nonkey_values		ReturnCode_t
clear_value		ReturnCode_t
	id	MemberId
loan_value		DynamicData
	member_id	MemberId
return_loaned_value		ReturnCode_t
	value	DynamicData
clone		DynamicData

7.5.2.11.1 Property: `value`; Operations: `get_member_id_by_name` and `get_member_id_at_index`

Many of the properties and operations defined by this class refer to values within the sample, which are identified by name, member ID, or index. What constitutes a value within a sample, and which means of accessing it are valid, depends on the type of this sample.

- If this object is of an aggregated type, values correspond to the type's members and can be accessed by name, member ID, or index.
- If this object is of a sequence or string type, values correspond to the elements of the collection. These elements must be accessed by index; the mapping from index to member ID is unspecified.
- If this object is of a map type, values correspond to the values of the map. Map keys are implicitly converted to strings and can thus be used to look up map values by name. Map values can also be accessed by index, although the order is unspecified.
- If the object is of an array type, values correspond to the elements of the array. These elements must be accessed by index; the mapping from index to member ID is unspecified. If the array is multi-dimensional, elements are accessed as if they were "flattened" into a single-dimensional array in the order specified by the IDL specification.
- If the object is of a bitmask type, values correspond to the flags within the bitmask and are all of `Boolean` type. Named flags can be accessed using that name; any bit within the bound of the bitmask may be accessed by its index. The mappings from name and index to member ID are unspecified.
- If the object is of an enumeration or primitive type, it has no contained values. However, the value of the sample itself may be indicated by "name" using a nil or empty string, by "ID" by passing `MEMBER_ID_INVALID`, or by "index" by passing index 0.

Note that indices used here are always relative to other values *in a particular* `DynamicData` *object*. Even though member *definitions* within aggregated types have a well-defined order, the same is *not* true within data samples or across data samples. Specifically, the index at which a member of an aggregated type appears in a particular data sample may not match that in which it appears in the corresponding type and may not match the index at which it appears in a different data sample. There are several reasons for these inconsistencies:

- The producer of the sample may be using a slightly different variant of the type than the consumer, which may add to, or omit elements from, the set of members known to the consumer.
- An optional member may have no value; in such a case, it will be omitted, thereby decreasing the index of every subsequent member.
- A non-optional member may likewise be omitted (which semantically is equivalent to it taking its default value). An implementation may discretionarily omit such members (e.g., to save space).
- Preserving member order is not necessary or even desirable (e.g., for performance reasons) for certain data representations.

7.5.2.11.2 Property: `descriptor`

This property shall contain a descriptor for each value in this object, identified by the member ID. The meaning of the member ID shall be as it is described for the `value` property.

7.5.2.11.3 Clearing Values: Operations `clear_value`, `clear_all_values`, and `clear_nonkey_values`

The meaning of "clearing" a member depends on the type of data represented by this sample:

- If this sample is of an aggregated type, and the indicated member is optional, remove it. If the indicated member is not optional, set it to its default value.
- If this sample is of a variable-length collection type, remove the indicated element, shifting any subsequent elements to the next-lowest index.
- If the sample is of an array type, set the indicated element to its default value.
- If the sample is of a bitmask type, clear the indicated bit.
- If the sample is of an enumerated type, set it to the first value of the enumerated type.
- If the sample is of a primitive type, set it to its default value.

The `clear_all_members` takes the above action for each value in turn. The `clear_nonkey_value` operation has exactly the same effect as `clear_all_values` with one exception: the values of key fields of aggregated types retain their values.

7.5.2.11.4 Operation: `clone`

Create and return a new data sample with the same contents as this one. A comparison of this object and the clone using `equals` immediately following this call will return `true`.

7.5.2.11.5 Operation: `equals`

Two data samples are considered to be equal if and only if all of the following conditions hold:

- Their respective type definitions are equal.
- All contained values are equal and occur in the same order.
- If the samples' type is an aggregated type, the previous rule shall be amended as follows:
    - Members shall be compared without regard to their order.
    - One of the samples may omit a non-optional member that is present in the other if that member takes its default value in the latter sample.

7.5.2.11.6 Operation: `get_item_count`

The "item count" of the data depends on the type of the object.

- *If the object is of a collection type*, return the number of elements currently in the collection. In the case of an array type, this value will always be equal to the product of the bounds of all array dimensions.
- *If the object is of a bitmask type*, return the number of named flags that are currently set in the bitmask.
- *If the object is of a structure or annotation type*, return the number of members in the object. This value may be different than the number of members in the corresponding `DynamicType`—for example, some optional members may be omitted.
- *If the object is of a union type*, return the number of members in the object. This number will be two if the discriminator value selects a member and one otherwise.
- *If the object is of a primitive or enumerated type*, it is atomic: return one.
- *If the object is of an alias type*, return the value appropriate for the alias's base type.

7.5.2.11.7 Operations: `loan_value` and `return_loaned_value`

The "loan" operations loan to the application a `DynamicData` object representing a value within this sample. These operations allow applications to visit values without allocating additional `DynamicData` objects or copying values. This loan shall be returned by the `return_loaned_value` operation.

A given `DynamicData` object may support only a single outstanding loan at a time. That is, after calling a "loan" operation, an application must subsequently call `return_loaned_value` before calling a loan operation again. If an application violates this constraint, the loan operation shall return a nil value.

A loan operation shall also return a nil value if the indicated value does not exist.

The `return_loaned_value` operation shall return `RETCODE_PRECONDITION_NOT_MET` if the provided sample object does not represent an outstanding loan from the sample on which the operation is invoked.

7.5.2.11.8 Property: `type`

This property provides the type that defines the values within this sample. Its value shall not be nil.

7.5.2.11.9 Platform-Specific Model: IDL

The programming language-specific APIs for the Dynamic Type and Dynamic Data classes and their companion classes shall be based on the following IDL definitions, transformed according to the IDL language mapping described above, as expanded below.

The conceptual model refers to the type Object, objects of which may be of any concrete type supported by the Type System defined by this specification. The mapping to IDL below represents this multiplicity of concrete types by multiplying the methods implied by the properties, qualifying each method with a concrete type. For example, a qualified association `foo: Int32 → Object` would expand to `get_int32_foo`, `get_int16_foo`, etc. Specifically, the mapping uses the following type expansions:

- Each primitive type has its own expansion. Primitive types can be implicitly promoted to larger primitive types as defined below.
- Strings of `Char8` and `Char16` elements have their own expansions qualified by “string” and “wstring” respectively.
- Enumerated types shall be implicitly converted to any signed integer type having at least as many bits as the enumerated type’s `@bit_bound`. They are thus accessible through those primitive methods.
- Bitmasks shall be implicitly converted to any unsigned integer type having at least as many bits as the bitmask’s `@bit_bound`. They are thus accessible through those primitive methods.
- Alias types shall be implicitly converted to their ultimate base type and are thus accessible through the methods appropriate for that type.
- Sequences of primitive types and strings have their own expansions in which the name of the property has been made plural. Arrays shall also be accessible through these methods.
- Expansions that operate on `DynamicData` objects, qualified by “complex,” catch the remaining cases and offer an alternative approach to accessing values of any of the above types.

If a `DynamicData` object represents an object of a resizable collection type (string, sequence, or map), these setters may also be used to append new elements to the collection.

- For a string or sequence type, use `get_member_id_at_index` to obtain an ID for the index one greater than the current length.
- For a map type, use `get_member_id_by_name` to obtain an ID for the new map key.

If a `DynamicData` object represents an object of a Union type, the union discriminator may be accessed using the standard set and get operations by specifying member ID 0, see 7.2.3.2.4.3.

As mentioned above, it shall be possible to implicitly promote integral types. These shall be supported during both "get" and "set" operations such that a smaller type promotes to a large type but not vice versa. For example, it shall be possible to get the value of a short integer field as if it were a long integer, and it shall be possible to set the value of a long integer as if it were a short integer. Specifically, the following promotions shall be supported:

- `Int8` → `Int16, Int32`, `Int64`, `Float32`, `Float64`, `Float128`
- `Int16` → `Int32`, `Int64`, `Float32`, `Float64`, `Float128`
- `Int32` → `Int64`, `Float64`, `Float128`
- `Int64` → `Float128`
- `UInt8` → `Int16`,`Int32`, `Int64`, `UInt16`, `UInt32`, `UInt64`, `Float32`, `Float64`, `Float128`
- `UInt16` → `Int32`, `Int64`, `UInt32`, `UInt64`, `Float32`, `Float64`, `Float128`
- `UInt32` → `Int64`, `UInt64`, `Float64`, `Float128`
- `UInt64` → `Float128`
- `Float32` → `Float64`, `Float128`
- `Float64` → `Float128`
- `Float128` → (*none*)
- `Char8` → `Char16`, `Int16`, `Int32`, `Int64`, `Float32`, `Float64`, `Float128`
- `Char16` → `Int32`, `Int64`, `Float32`, `Float64`, `Float128`
- `Byte` → (*any*)
- `Boolean` → `Int8`, `Int16`, `Int32`, `Int64`, `UInt8`, `UInt16`, `UInt32`, `UInt64`, `Float32`, `Float64`, `Float128`

The complete IDL representation may be found in "Annex C: Dynamic Language Binding."

7.6 Use of the Type System by DDS

This clause describes how DDS uses the type system.

7.6.1 Topic Model

A DDS topic exists in two senses of the word:

1. **On the network**, with respect to interoperability: This is the sense in which we say that a reader and a writer share the "same" topic, even though they obtain the topic's definition independently within their implementations.
2. **In application code**, with respect to portability: Each component that uses a topic creates or looks up a local proxy for that topic.

On the network, a given topic is associated with one or more types. A given writer or reader endpoint belongs to one topic and is associated with one of the types of that topic. If a writer and a reader share the same topic, it is assumed that they are intended to communicate with one another. At that point, the Service evaluates the two endpoints to make sure that they specify consistent types (see Clause 7.6.3.4.2, "Rules for Type Consistency Enforcement") and compatible QoS (see [DDS]).

Typically, in application code, a topic is associated with a single type (as has always been the case in the [DDS] API)[5]. Therefore, multiple API topics may correspond to (different views of) the same network topic. A given reader or writer endpoint is associated with one of them. See Clause 7.6.4, "Local API Extensions", for definitions of the programming interfaces that support this polymorphism.

Generic services (e.g., logger, monitor) may discover a topic associated with one or more types. Such services may be able to handle all representations of the types, without ever having type specific knowledge hardcoded into them.

7.6.2 Types that may be associated with a DDS Topic

The only types that may be associated with a DDS Topic are the Aggregated Types defined in 7.2.3.2.4, that is, Structure types (7.2.3.2.4.2) and Union types (7.2.3.2.4.3). The reason is that these are the only types that support defining key members and can therefore be used to model Topics that contain multiple data-instances.

Any type in the type system may appear nested within the (Aggregated Type) associated with a Topic.

7.6.3 Discovery and Endpoint Matching

The enhanced Type System and the richer set of available Data Representations necessitate extensions to the discovery and endpoint matching process defined by the DDS specification, which may be divided into three categories:

- **Data Representation**: The multiplicity of data representations introduced by this specification creates the possibility that different `DataWriter` and `DataReader` endpoints in a single system may support different combinations of representations. It is therefore necessary to define a mechanism whereby endpoints can inform each other of the representations they support and thereby negotiate communication.

[5] **Design rationale (non-normative)**: This constraint keeps the programming model the same for both XTypes-supporting and non-XTypes-supporting implementations, and it keeps the mental model simple for the majority of programmers, who will not be aware of the presence of multiple types in their topics.

- **Discovery-Time Data Typing**: The dynamic features of this specification depend on the ability of components to discover the data types used by their peers.
- **Type Consistency Enforcement**: One of the criteria for `DataWriter-DataReader` matching defined by DDS is that the type names of each must match exactly. In complex dynamic systems, this restriction can prove overly limiting. Based on the type compatibility rules defined by this specification, matching endpoints shall be permitted to declare types that are not identical but nevertheless have well-defined relationships with one another.

These extensions are defined in the following sections.

7.6.3.1 Data Representation QoS Policy

With multiple standard data Representations available, and vendor-specific extensions possible, `DataWriters` and `DataReaders` must be able to negotiate which data representation(s) to use. This negotiation shall occur based on a new QoS policy: `DataRepresentationQosPolicy`.

7.6.3.1.1 `DataRepresentationQosPolicy`: Conceptual Model

The conceptual model for data representation negotiation consists of several parts:

The identification of data representations.

- The specification of supported (requested) representations by `DataReader`s and used (offered) representation by `DataWriters`.
- The algorithm by which a suitable representation is chosen for a particular `DataReader`/`DataWriter` pair, given the supported representations of each.

Each data representation shall be identified by a two-byte signed integer value, the "representation identifier." Within the range of such a value, the negative values shall be reserved for definition by DDS implementations. The remainder of the range shall be reserved for the OMG for use in future specifications, including this specification.

Within the OMG-reserved range, this specification defines three representation identifiers:

- `XCDR`, which corresponds to the Extended CDR Representation encoding version 1 and takes the value 0.
- `XML`, which corresponds to the XML Data Representation and takes the value 1.
- `XCDR2`, which corresponds to Extended CDR Representation encoding version 2 and takes the value 2.

Each `Topic`, `DataReader` and `DataWriter` shall have a QoS policy `DataRepresentationQosPolicy`. This policy shall contain a list of representation identifiers. This policy has request-offer semantics, and its value cannot be changed after the entity in question has been enabled [DDS].

- Writers offer a single representation. A writer will use its offered policy to communicate with its matched readers.

(Because the policy structure includes a sequence, it is technically possible for the writer to offer more than one representation. Implementers of this specification may use this fact in order to offer extended functionality; however, this specification does not specify any meaning for the representation identifiers after the first, and implementations may ignore them.)

Writers belonging to implementations of XTYPES version 1.1 or earlier shall not announce the XCDR2 representation identifier.

- By default, Writers belonging to implementations of XTYPES version 1.2 and later:
  - Shall generate or include run-code that can serialize using version 2 encodings.
  - Shall default to announce the XCDR2 representation identifier.
  - Optionally may generate or include run-code that can serialize using version 1 encodings. In this case, they shall offer the means to configure at run-time the encoding version used by the DataWriter and adjust the offered representation identifiers in the DataRepresentationQosPolicy accordingly.

- Readers request one or more representations.
  - Readers requesting the XML Data Representation shall be prepared to receive either valid or merely well-formed XML documents. If a received document is well-formed but does not include any XML namespace declarations, the reader shall assume that the document could be validated using the XSD Data Validation Schema of the corresponding sample's type if it were to include such namespace declarations.
  - Readers belonging to implementations of XTYPES version 1.1 or earlier shall not announce the XCDR2 representation identifier.
  - By default, Readers belonging to implementations of XTYPES version 1.2 and later:
    - Shall generate or include run-time code that can deserialize version 2 encodings.
    - Shall request XCDR2 encoding.
    - Optionally may generate or include run-time code that can deserialize version 1 encodings. In this case they shall also request XCDR encoding in addition to XCDR2 encoding.

- When representations are specified in the `TopicQos`, the first element of the sequence applies to writers of the Topic, and the whole sequence applies to readers of the Topic.
- If a writer's offered representation is contained within a reader's sequence, the offer satisfies the request and the policies are compatible. Otherwise, they are incompatible.

The default value of the `DataRepresentationQosPolicy's` value member shall be an empty sequence. An empty sequence shall be taken to be equivalent to a sequence containing the single element `XCDR`.

The `DataRepresentationQosPolicy` shall not be changeable after its corresponding Entity has been enabled.

The rules defined in this clause result in a compatibility matrix shown in Table 56.

Table 56 – Compatibility matrix for the DataRepresentationQosPolicy

DataWriter offered DataRepresentationId_t	*DataReader requested DataRepresentationId_t*	*Encoding compatibility check*
XCDR DataWriter will encode data according to version 1 encoding rules. Either the DataWriter is a legacy (xtypes 1.1) DataWriter or else it has been configured to use XCDR VERSION1.	**XCDR** DataReader is a legacy (xtypes 1.1) DataReader	**Compatible**. DataWriter finds its encoding among the ones understood by DataReader. DataReader finds its encoding among the ones understood by DataWriter.
	XCDR and XCDR2 DataReader is a (xtypes 1.2) DataReader	**Compatible**. DataWriter finds its encoding among the ones understood by DataReader. DataReader finds its encoding among the ones understood by DataWriter.
XCDR2 DataWriter will encode data according to version 2 encoding rules. DataWriter is a new (xtypes 1.2) DataWriter and it has been configured to use the version 2 encoding.	**XCDR** DataReader is a legacy (xtypes 1.1) DataReader	**Not Compatible**. DataWriter does not find its encoding among the ones understood by DataReader. DataReader does not find its encoding among the ones understood by DataWriter.
	XCDR and XCDR2 DataReader is a new (xtypes 1.2) DataReader	**Compatible**. DataWriter finds its encoding among the ones understood by DataReader. DataReader finds its encoding among the ones understood by DataWriter.

7.6.3.1.2 Use of the RTPS Representation Identifier and Options

As specified in the RTPS standard, the data representation used for the serialized data (the SerializedPayload submessage element) is identified by a two-byte value known as the *RepresentationIdentifier* ([RTPS], clause 10.2). RTPS further specifies that the representation identifier values used to send application Topics shall correspond to those defined in Table 57 below ([RTPS], clause 10.5).

RepresentationIdentifier values with the first byte is in the range 0xC0 to 0xFF (inclusive) shall be reserved for use by DDS implementations and shall be interpreted according to the RTPS vendor ID. The remaining values shall be reserved for the OMG[6] for use in future specifications, including revisions of this specification.

The valid *RepresentationIdentifier* values and their corresponding interpretations are specified in Table 57 below.

Table 57 – RTPS representation identifier values

Representation	***Extensibility Kind***	***Encoding Version***	***Endianness***	***RTPS RepresentationIdentifier***	***Identifier value***
XCDR	FINAL	1	Big Endian	`CDR_BE`	`{0x00, 0x00}`
XCDR	FINAL	1	Little Endian	`CDR_LE`	`{0x00, 0x01}`
XCDR	APPENDABLE	1	Big Endian	`CDR_BE`	`{0x00, 0x00}`
XCDR	APPENDABLE	1	Little Endian	`CDR_LE`	`{0x00, 0x01}`
XCDR	MUTABLE	1	Big Endian	`PL_CDR_BE`	`{0x00, 0x02}`
XCDR	MUTABLE	1	Little Endian	`PL_CDR_LE`	`{0x00, 0x03}`
XCDR	FINAL	2	Big Endian	`CDR2_BE`	`{0x00, 0x06}`
XCDR	FINAL	2	Little Endian	`CDR2_LE`	`{0x00, 0x07}`
XCDR	APPENDABLE	2	Big Endian	`D_CDR2_BE`	`{0x00, 0x08}`
XCDR	APPENDABLE	2	Little Endian	`D_CDR2_LE`	`{0x00, 0x09}`
XCDR	MUTABLE	2	Big Endian	`PL_CDR2_BE`	`{0x00, 0x0a}`
XCDR	MUTABLE	2	Little Endian	`PL_CDR2_LE`	`{0x00, 0x0b}`
XML	any	any	any	`XML`	`{0x00, 0x04}`

[6] Note that all RTPS-specified encapsulation identifier values fall within the OMG-reserved range.

As specified in the RTPS standard, the *RepresentationIdentifier* is immediately followed by a two-byte *RepresentationOptions* field ([RTPS], Clause 10.2). RTPS further specifies that this field shall be followed by the serialized data, encoded using the data representation indicated by the *RepresentationIdentifier*.

The RTPS specification does not assign any semantics to the *RepresentationOptions* field. This DDS-XTYPES specification extends its definition by specifying the use of certain bits within it.

Implementations conforming to this (DDS-XTYPES) specification shall set the least significant two bits in the second byte of the RepresentationOptions field to a value that encodes the number of padding bytes added after the end of the serialized payload.

These padding bytes ensure that the position immediately following the payload corresponds to the next 4-byte aligned offset, consistent with the alignment rules defined in RTPS.

Specifically, the least significant two bits shall be set to binary 00 if no padding bytes are added, or to binary 01, 10, or 11 if one, two, or three padding bytes, respectively, are added. These bits shall be interpreted by the receiver to determine the precise end of the serialized data.

For example, assume structures `TypeA`, `TypeB`, and `TypeC` defined by the following IDL:

```
struct TypeA {
  short member1;
};
struct TypeB {
  short member1;
  char  member2;
};
struct TypeC {
  short member1;
  char  member2;
  char  member3;
};
```

Furthermore assume an object O1 of type `TypeA` with value O1.member1 = 0x11, an object O2 of type `TypeB` with value O2.member1= 0x23, O2.member2 = ‘b’, and an object O3 of type TypeC with value O3.member1= 0x23, O3.member2 = ‘b’, O3.member3 = ‘c’. The CDR big endian representation of these two objects, including *RepresentationIdentifier* and *RepresentationOptions* would be:

Object O1 representation:

```
0...2...4.......8...............16.............24.............32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   CDR_BE { 0x00, 0x00 }       |        0x00   |     0x02      |
```

```
+---------------+---------------+---------------+---------------+
|       O1.member1 = 0x11       | padding (2 bytes) {0x00, 0x00}|
+---------------+---------------+---------------+---------------+
NEXT RTPS SUBMESSAGE...
```

Object O2 representation:

```
0...2...4.......8...............16..............24..............32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   CDR_BE { 0x00, 0x00 }       |        0x00   |     0x01      |
+---------------+---------------+---------------+---------------+
|       O2.member1 = 0x23       |O2.member2 ='b'| padding {0x00}|
+---------------+---------------+---------------+---------------+
NEXT RTPS SUBMESSAGE...
```

Object O3 representation:

```
0...2...4.......8...............16..............24..............32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   CDR_BE { 0x00, 0x00 }       |        0x00   |     0x00      |
+---------------+---------------+---------------+---------------+
|       O3.member1 = 0x23       |O3.member2 ='b'|O3.member2 ='c'|
+---------------+---------------+---------------+---------------+
NEXT RTPS SUBMESSAGE...
```

7.6.3.1.3 `DataRepresentationQosPolicy`: Platform-Specific API

The conceptual model defined above shall be transformed into the IDL definitions `RepresentationId_t`, `RepresentationIdSeq`, `DATA_REPRESENTATION_QOS_POLICY_ID`, `DATA_REPRESENTATION_QOS_POLICY_NAME`, and `DataRepresentationQosPolicy`. These definitions are given in "Annex D: DDS Built-in Topic Data Types."

The topic, publication, and subscription built-in topic data types shall each indicate the data representation of the associated entity with a new member:

```
@id(0x0073) DDS::DataRepresentationQosPolicy data_representation;
```

The `data_representation` member (member ID `0x73`), may be omitted in the topic, publication, and subscription built-in topic data types. In this case, the receiver shall interpret the `data_representation` as if it was set to the empty sequence (sequence of length 0). As specified in `DataRepresentationQosPolicy`: Conceptual Model, the empty sequence value for a `DataRepresentationQosPolicy` shall be interpreted as equivalent to a sequence containing the single element XCDR.

7.6.3.2 Discovery Built-in Topics

7.6.3.2.1 Type Information

A DDS `DomainParticipant` needs to have type information on remote DomainParticipant Topics that are also being published or subscribed by the local DomainParticipant. That way the DomainParticipant can ensure type compatibility with the remote endpoints it matches.

XTYPES 1.1 optionally included the `TypeObject` information into the Publication and Subscription discovery built-in topic data. The TypeObject in XTYPES version 1.1 (TypeObjectV1) was defined as a "library" that contained not only the data-type for the Topic-Type, but also any data-types that were recursively needed to understand the Topic-Type (e.g. the data-types of the members of a structure). That way a DomainParticipant that discovered the endpoint would have all the type information readily available.

XTYPES 1.2 redefines the structure of the `TypeObject` (TypeObjectV2) and introduces a different mechanism that avoids sending `TypeObjects` to DomainParticipants that are not interested in it (e.g., they already know the `TypeObject`, or they are not publishing or subscribing an affected Topic). The XTYPES 1.2 approach is:

- Send TypeInformation that include `TypeIdentifiers` (instead of `TypeObjects`) in the discovery built-in topics.
- Uses the `TypeIdentifiers` to determine which types a DomainParticipant is interested in.
- Uses a new pair of built-in endpoints to request the TypeObjects for those `TypeIdentifiers` the DomainParticipant is interested in, and receive the reply.

The content of the type information is defined in the IDL below:

```
@extensibility(APPENDABLE)
struct TypeIdentifierWithSize {
    TypeIdentifier  type_id;
    unsigned long   typeobject_serialized_size;
};

@extensibility(APPENDABLE)
struct TypeIdentifierWithDependencies {
    TypeIdentifierWithSize    typeid_with_size;
    // The total additional types related to minimal_type
    long dependent_typeid_count;
    sequence<TypeIdentifierWithSize> dependent_typeids;
};
```

```
typedef sequence<TypeIdentifierWithDependencies>
        TypeIdentifierWithDependenciesSeq;


@extensibility(MUTABLE)
struct TypeInformation {
    @id(0x1001) TypeIdentifierWithDependencies minimal;
    @id(0x1002) TypeIdentifierWithDependencies complete;
};
typedef sequence<TypeInformation> TypeInformationSeq;
```

The `TypeInformation` includes information on the data-type associated with the Endpoint (`DataWriter` or `DataReader`, i.e., the `TopicType`. It includes two fields, minimal and complete.

The field ***minimal*** contains the MINIMAL Hash `TypeIdentifiers` for the `TopicType` and types that it depends on:

- The field *minimal.typeid_with_size* shall contain the MINIMAL Hash `TypeIdentifier` of the TopicType and the serialized size of the associated `TypeObject`.
- The field *minimal.dependent_typeid_count* shall contain the total number of other MINIMAL Hash `TypeIdentifiers` that correspond to data-types the TopicType depends on. This field may be set to -1 to indicate it is not being provided.
- The field *minimal.dependent_typeids* may contain some of the MINIMAL Hash `TypeIdentifiers` of the types the TopicType depends on, along with the serialized size of the respective TypeObjects.

The field ***complete*** contains the COMPLETE Hash `TypeIdentifiers` for the TopicType and types that it depends on:

- The field *complete.typeid_with_size* shall contain the COMPLETE Hash `TypeIdentifier` of the TopicType and the serialized size of the associated TypeObject.
- The field *complete.dependent_typeid_count* shall contain the total number of other COMPLETE Hash `TypeIdentifiers` that correspond to data-types the TopicType depends on. This field may be set to -1 to indicate it is not being provided.
- The field *complete.dependent_typeids* may contain some of the COMPLETE Hash `TypeIdentifiers` of the types the TopicType depends on, along with the serialized size of the respective TypeObjects.

As mentioned the field *dependent_typeids* may be used to optionally announce some of the Hash `TypeIdentifiers` the TopicType recursively depends on. The decision of which types to include in the *dependent_typeids* is left to the implementation: It may be set to the empty sequence, or include all the Hash `TypeIdentifiers` that the TopicType depends on, or

something in between. If *dependent_typeid_count* is not -1, then length of the *dependent_typeids* sequence shall be less or equal to *dependent_typeid_count*.

The `TypeIdentifiers` included in the `TypeInformation` shall include only **direct HASH** TypeIdentifiers (see Clause 7.3.3.7.3). In addition it shall not contain individual type identifiers for types belonging to Strongly Connected Component (i.e., those with discriminator TI_STRONG_COMPONENT), instead it shall include the identifier of the whole Strongly-Connected Component (SCCIdentifier, see Clause 7.3.3.10.4).

A DomainParticipant can use the `TypeInformation` to determine if it already knows the associated TopicType and determine the type compatibility with local endpoints. In case some of the `TypeIdentifiers` announced by a remote endpoint are not known to a DomainParticipant, it can use the built-in `TypeLookup` Service to retrieve the `TypeObject` of the types associated with those `TypeIdentifiers`.

7.6.3.2.2 Data Representation Detail

The `DataRepresentationDetail` may be used to provide additional details of the serialized data representation used by a `DataWriter` or expected by a `DataReader only`. The purpose of this information is to support diagnostics and improve interoperability in cases where clarifications to this specification may affect the serialized form of certain types used by the endpoint.

The content of the data representation detail is defined in the IDL below:

```
module XTypes {
  @bit_bound(32)
  bitmask DataRepresentationDetailMask {
    @position(0) map_xcdr2_ommits_length;
  };
  @extensibility(APPENDABLE)
  struct DataRepresentationDetail {
    DataRepresentationDetailMask detail_mask;
  };
}; // module XTypes
```

The presence of the `detail_mask with the` with the `map_xcdr2_omitts_lengts` flag set indicates that the XCDR2 serialized presentation of non-primitive map types uses a modified version of rule `(35)` in 7.4.3.5.3 that omits the serialization of `O.length` that follows the `DHEADER`.

7.6.3.2.3 Additional members included in discovery built-in Topics

The topic, publication, and subscription built-in topic data structures shall each indicate the type(s) used for communication by the associated entity. These declarations shall be as follows:

```
@id(0x0007) ObjectName type_name;
```

```
@id(0x0072) @optional TypeObjectV1 type;                          // XTYPES 1.1
@id(0x0075) @optional XTypes::TypeInformation type_information; // XTYPES 1.2
@id(0x0076) @optional XTypes::DataRepresentationDetail
                                    data_representation_detail; // XTYPES 1.4
```

`TypeObjectV1` corresponds to the `TypeObject` data type specified in "Annex B: Representing Types with TypeObject" of DDS-XTYPES Version 1.1 [DDS-XTYPES11]. Likewise, the `type` member shall be set as specified in Clause 7.3.4 of [DDS-XTYPES11].

Any members of type `TypeInformation` that appear within the built-in Topics shall be serialized using XCDR2, even if the built-in Topic type itself (and its other members) are serialized using XCDR1 as specified in [RTPS].

Any members of type `DataRepresentationDetail` that appear within the built-in Topics shall also be serialized using XCDR2, even if the built-in Topic type itself (and its other members) are serialized using XCDR1 as specified in [RTPS].

.

Non-normative note: When the `TypeObjectV1` and `TypeInformation` members (called `type` and `type_information`) are omitted from the built-in topic samples, `type_name` is the only way to resolve entity matching and as a consequence, it is possible that incompatibility between topic-types is not recognized.

7.6.3.3 Built-in TypeLookup service

7.6.3.3.1 Introduction

This specification defines two built-in Topics that are used to query DomainParticipant for type information. This includes getting the TypeObjects associated with `TypeIdentifiers` as well as determining the list of types that a given type depends on recursively:

- One built-in topic is used for TypeLookup requests. It has two built-in endpoints, a DataWriter to send the request and a DataReader to receive that request.
- The second built-in topic is used for TypeLookup replies. It has two built-in endpoints, a DataWriter to send the reply and a DataReader to receive that reply.

The data types associated with the TypeLookup Request/Reply topics are defined in accordance with the Basic Service Mapping from the [DDS-RPC] specification. It is not, however, a requirement to implement the DDS-RPC specification in order to claim compliance with this specification. The only requirement is to implement the `TypeLookup` built-in endpoints as defined in this XTYPES specification.

In order to facilitate the reading of this specification, some type definitions from DDS-RPC Clause 7.5.1.1.1 have been copied in the next clause.

7.6.3.3.2 Types reused from DDS-RPC

The Extensibility (see section 7.2.3) of IDL constructs in this section shall be FINAL.

```
/* BEGIN of definitions copied from DDS-RPC */
module dds {
typedef octet GuidPrefix_t[12];

struct EntityId_t {
    octet entityKey[3]; octet entityKind;
};

struct GUID_t {
    GuidPrefix_t guidPrefix;
    EntityId_t entityId;
};

struct SequenceNumber_t {
    long high;
    unsigned long low;
};

struct SampleIdentity {
    GUID_t writer_guid;
    SequenceNumber_t sequence_number;
};

} // module dds

// Module dds::rpc

module dds { module rpc {
    typedef octet UnknownOperation;
    typedef octet UnknownException;
    typedef octet UnusedMember;
};
```

```
enum RemoteExceptionCode_t {
    REMOTE_EX_OK,
    REMOTE_EX_UNSUPPORTED,
    REMOTE_EX_INVALID_ARGUMENT,
    REMOTE_EX_OUT_OF_RESOURCES,
    REMOTE_EX_UNKNOWN_OPERATION,
    REMOTE_EX_UNKNOWN_EXCEPTION
};

typedef string<255> InstanceName;

struct RequestHeader {
    SampleIndentity_t requestId;
    InstanceName instanceName;
};

struct ReplyHeader {
    dds::SampleIdentity relatedRequestId;
    dds::rpc::RemoteExceptionCode_t remoteEx;
};
} }  // module dds::rpc
/* END of definitions copied from DDS-RPC */
```

7.6.3.3.3 TypeLookup Types and Endpoints

Compliant implementations shall include the four built-in service endpoints shown in Table 58 below.

Table 58 – Built-in Endpoints added by the XTYPES specification

Built-in Endpoint	*RTPS EntityId_t*	*Associated Topic Type*	*Topic Name*
TypeLookupService RequestDataWriter	ENTITYID_TL_SVC_REQ_WRITER = {{00, 03, 00}, c3}	TypeLookup_Request	"DCPSTypeLookup_Request"
TypeLookupService RequestDataReader	ENTITYID_TL_SVC_REQ_READER ={{00, 03, 00}, c4}	TypeLookup_Request	"DCPSTypeLookup_Request"

TypeLookupService ReplyDataWriter	ENTITYID_TL_SVC_REPLY_WRITER = {{00, 03, 01}, c3}	TypeLookup_Reply	"DCPSTypeLookup_Reply"
TypeLookupService ReplyDataReader	ENTITYID_TL_SVC_REPLY_READER = {{00, 03, 01}, c4}	TypeLookup_Reply	"DCPSTypeLookup_Reply"

The pair `TypeLookupServiceRequestDataWriter` and `TypeLookupServiceReplyDataReader` is used to invoke the built-in `TypeLookup` Service (send the request and receive the reply).

The pair `TypeLookupServiceRequestDataReader` and `TypeLookupServiceReplyDataWriter` is used to implement the `TypeLookup` Service (receive the request and send the reply).

The Quality of Service for the four-built-in endpoints shall match the default Qos for service endpoints defined in Clause 7.10.2 of [DDS-RPC], specifically the RELIABILITY policy shall be DDS_RELIABLE_RELIABILITY_QOS, the HISTORY policy to DDS_KEEP_ALL_HISTORY_QOS and the DURABILITY policy to DDS_VOLATILE_DURABILITY_QOS.

The Data Representation used by these endpoints shall be Extended CDR Representation (encoding version 2), as specified in section 7.4.2.

The associated data-types are defined using IDL below. Types without explicit extensibility annotations shall be APPENDABLE.

```
module dds { module builtin {

// computed from @hashid("getTypes")

const int32 TypeLookup_getTypes_HashId = 0x018252d3;

// computed from @hashid("getDependencies");

const int32 TypeLookup_getDependencies_HashId = 0x05aafb31;


// Query the TypeObjects associated with one or more TypeIdentifiers

@extensibility(MUTABLE)

struct TypeLookup_getTypes_In {

    @hashid sequence<TypeIdentifier>  type_ids;

};


@extensibility(MUTABLE)

struct TypeLookup_getTypes_Out {

    @hashid sequence<TypeIdentifierTypeObjectPair>  types;

    @hashid sequence<TypeIdentifierPair>  complete_to_minimal;
```

```
};

union TypeLookup_getTypes_Result switch(int32) {
  case DDS::RETCODE_OK:
    TypeLookup_getTypes_Out   result;
};

// Query TypeIdentifiers that the specified types depend on
@extensibility(MUTABLE)
struct TypeLookup_getTypeDependencies_In {
    @hashid sequence<TypeIdentifier>  type_ids;
    @hashid sequence<octet, 32>       continuation_point;
};

@extensibility(MUTABLE)
struct TypeLookup_getTypeDependencies_Out {
    @hashid sequence<TypeIdentifierWithSize> dependent_typeids;
    @hashid sequence<octet, 32>             continuation_point;
};

union TypeLookup_getTypeDependencies_Result switch(int32){
case DDS::RETCODE_OK:
    TypeLookup_getTypeDependencies_Out   result;
};

// Service Request
union TypeLookup_Call switch(int32) {
  case TypeLookup_getTypes_HashId:
    TypeLookup_getTypes_In             getTypes;
  case TypeLookup_getDependencies_HashId:
    TypeLookup_getTypeDependencies_In  getTypeDependencies;
};
```

```
@RPCRequestType
@final
struct TypeLookup_Request {
    dds::rpc::RequestHeader header;
    TypeLookup_Call         data;
};

// Service Reply
union TypeLookup_Return switch(int32) {
  case TypeLookup_getTypes_HashId:
    TypeLookup_getTypes_Result   getType;

  case TypeLookup_getDependencies_HashId:
    TypeLookup_getTypeDependencies_Result getTypeDependencies;
};

@RPCReplyType
@final
struct TypeLookup_Reply {
    dds::rpc::RequestHeader header;
    TypeLookup_Return       return;
};
}} // dds::builtin
```

The "_In" and "_Out" types are used to represent the request and reply parameters to the service. These types are defined with extensibility kind MUTABLE. Therefore they can be modified without breaking interoperability.

Implementers may add their own members to these MUTABLE types. If they do they shall use member IDs obtained using the `@hashid` annotation with a string value that has an Internet domain name owned by the implementor prefix. This avoids member ID conflicts with additions from other implementations. For example:

```
// Implementation from company acme.com adds parameters
// extra1 and extra2 to the getTypes request.
struct TypeLookup_getTypes_In {
    @hashid sequence<TypeIdentifier>    type_ids;
    @hashid("acme.com/extra1")  long    extra1;
```

```
    @hashid("acme.com/extra2")  string  extra2;
};
```

7.6.3.3.4 Use of the TypeLookup Service

The DDS Interoperability Wire Protocol [RTPS] specifies that the `ParticipantBuiltinTopicData` shall contain the attribute called `availableBuiltinEndpoints` that is used to announce the built-in endpoints that are available in the `DomainParticipant`. See Clause 8.5.3.2 of [RTPS]. The type for this attribute is an array of `BuiltinEndpointSet_t`.

For the UDP/IP PSM the `BuiltinEndpointSet` is mapped to a bitmap represented as type UInt32. Each built-in endpoint is represented as a bit in this bitmap with the bit values defined in Table 9.4 (Clause 9.3.2) of [RTPS].

This DDS XTypes specification reserves additional bits to indicate the presence of the corresponding built-in end points for the `TypeObjectLookup` Service. These bits shall be set on the `availableBuiltinEndpoints`. The bit that encodes the presence of each individual endpoint is defined in Table 59 below.

Table 59 – Mapping of the built-in endpoints added by this specification to the availableBuiltinEndpoints

Built-in Endpoint	***Bit in the ParticipantBuiltinTopicData availableBuiltinEndpoints***
TypeLookupServiceRequestDataWriter	(0x00000001 << 12)
TypeLookupServiceRequestDataReader	(0x00000001 << 13)
TypeLookupServiceReplyDataWriter	(0x00000001 << 14)
TypeLookupServiceReplyDataReader	(0x00000001 << 15)

Participants implementing (as a server) the `TypeLookup` service shall implement the `TypeObjectServiceRequestDataReader` and `TypeObjectServiceReplyDataWriter`.

The Service *instanceName* that appears in `the dds::rpc::RequestHeader` shall be set to the string obtained by concatenating:

1. the prefix "`dds.builtin.TOS.`", and
2. a 32-character string that encodes the `DomainParticipant` GUID.

The `DomainParticipant` GUID is an array of 16 octets. The GUID shall be encoded into 32 ASCII characters by representing each octet, in order, with two lowercase hexadecimal digits (0–9, a–f). No "0x" prefix shall be used.

For example: "`dds.builtin.TOS.00112233445566778899aabbccddeeff`"

Participants using (as a client) the `TypeLookup` shall implement the `TypeObjectServiceRequestDataWriter` and `TypeObjectServiceReplyDataReader`.

If a participant implements the `TypeLookup` it shall respond to requests for any `TypeIdentifier` that it announced within the `TypeInformation` included in the `PublicationBuiltinTopicData` or `SubscriptionBuiltinTopicData`.

The `dds::rpc::RequestHeader` in the `TypeLookup_Request` and the `TypeLookup_Reply` shall be set as specified in the [DDS-RPC] specification.

7.6.3.3.4.1 Service operation getTypeDependencies

When a `DomainParticipant` receives an incomplete list of `TypeIdentifiers` in a `PublicationBuiltinTopicData` or `SubscriptionBuiltinTopicData`, it may request the additional type dependencies by invoking the `getTypeDependencies` operation.

The `TypeLookup_getTypeDependencies_In` structure shall be filled as follows:

- The field `type_ids` shall contain the sequence of `TypeIdentifiers` for which the Participant wants to get the dependencies.
    - The TypeIdentifiers shall be only **direct HASH** Identifiers.
    - The `TypeIdentifiers` shall be either all MINIMAL hash `TypeIdentifiers` or all COMPLETE hash `TypeIdentifiers`. That is there shall be not be mixed.
    - The `TypeIdentifiers` shall not include identifiers for individual types in Strongly Connected Components (SCCs). Instead it shall use the identifier for the whole SCC (**SCCIdentifier**, see Clause 7.3.3.10.4).
- The field `continuation_point` shall not be present if the requester wants the response to include all the types that the specified types in `type_ids` depend on. Otherwise it shall be set to the `continuation_point` of the `TypeLookup_getTypeDependencies_Out` received in response to a previous call to `getTypeDependencies` with the same `type_ids`. This mechanism is used when the response of the service to a previous call to getDependencies did not return all the types and provided a `continuation_point`.

The `TypeLookup_getTypeDependencies_Out` structure shall be filled as follows:

- The field `dependent_typeids` shall exclusively contain of **direct HASH** `TypeIdentifiers` that are recursive dependencies from at least one of the `TypeIdentifiers` in the request.
- In the case where the `TypeIdentifierWithSize.type_id` corresponds to a **SCCIdentifier**, the `typeobject_serialized_size` shall be set to the serialized size of the sequence of `TypeObjects` corresponding to all the types in that Strongly Connected Component.
- The field `continuation_point` shall not be present if the response contains the complete list of types, otherwise it shall contain an opaque value that the requester shall use in a subsequent request for type identifiers.

7.6.3.3.4.2 Service operation getTypes

A `DomainParticipant` may invoke the operation `getTypes` to retrieve the `TypeObjects` associated with a list of `TypeIdentifiers`.

A `DomainParticipant` may find out about `TypeIdentifiers` of interest as part of the information received in a `PublicationBuiltinTopicData` or `SubscriptionBuiltinTopicData`. It may also find out `TypeIdentifiers` in reply to a `getDependencies` request, or it may find them inside `TypeObjects` received in reply to a `getTypes` request. Regardless of the source it can use the `getTypes` to get the associated `TypeObjects`.

The `TypeLookup_getTypes_In` structure shall be filled as follows:

- The field `type_ids` shall contain the **direct HASH** `TypeIdentifiers` for which the participant is requesting the TypeObjects.
- The field `type_ids` shall not include individual `TypeIdentifiers` belonging to a Strongly Connected Component (SCC). Instead it shall use the identifier for the whole SCC (SCCIdentifier, see Clause 7.3.3.10.4).

The `TypeLookup_getTypes_Out` structure shall be filled as follows:

- The field `types` shall contain TypeObjects that correspond to the `TypeIdentifiers` in the request.
    - If the request had a COMPLETE `TypeIdentifiers`, the `types` shall contain COMPLETE TypeObjects.
    - If the request had MINIMAL TypeIdentifiers the `types` may contain either MINIMAL or COMPLETE TypeObjects.
        - The field `complete_to_minimal` shall contain the mapping from COMPLETE TypeIdentifiers to MINIMAL TypeIdentifiers for any COMPLETE TypeIdentifiers that appear within COMPLETE TypeObjects that were sent in response to a query for a MINIMAL TypeIdentfier.
        - The use of the `complete_to_minimal` field allows an implementation to only send COMPLETE TypeObjects in response to the getTypes request, even if the requested `TypeIdentifiers` are MINIMAL `TypeIdentifiers`. The combination of a COMPLETE `TypeObject` and the mapping of MINIMAL to COMPLETE `TypeIdentifiers` makes it possible for the receiver to reconstruct the MINIMAL `TypeObject`.
- If a TypeIdentifier was a SCCIdentifier (see Clause 7.3.3.10.4), then the response shall threat the TypeObjects within the Strongly Connected Components atomically. Either include all in the reply or none.

7.6.3.4 Type Consistency Enforcement QoS Policy

The Type Consistency Enforcement QoS Policy defines the rules for determining whether the type used to publish a given data stream is consistent with that used to subscribe to it. It applies to `DataReader`s.

This policy corresponds directly to the Type Compatibility Options defined in 7.2.5.4.

The policy contains one enumerated field, kind, that specifies the overall type-consistency mode, and a set of Boolean fields that further refine the assignability behavior, see 7.2.5.4.

The effective values of these parameters are passed to the assignability evaluation defined in 7.2.5.5.

7.6.3.4.1 `TypeConsistencyEnforcementQosPolicy`: Conceptual Model

This policy applies only to `DataReader`s; it does not have request-offer (RxO) semantics [DDS]. The value of this policy cannot be changed after the entity in question has been enabled.

This policy provides a way to declare that type information must be available in order for two endpoints to match, they cannot match solely on type names. See Sub Clause 7.6.3.4.2 for more details on how matching between a `DataWriter` and `DataReader` occurs in the presence and absence of type information.

- The `force_type_validation` option requires type information to be available in order to complete matching between a `DataWriter` and `DataReader` when set to TRUE, otherwise matching can occur without complete type information when set to FALSE. The default value is false.

This policy defines a *type consistency kind*, which allows applications to select from among a set of predetermined policies. The following consistency kinds are specified:

- **`DISALLOW_TYPE_COERCION`**
- **`ALLOW_TYPE_COERCION`**

The meaning of each value is defined in 7.2.5.4.2.

Further details of these policies are provided in Clause 7.6.3.4.2.

The default enforcement kind shall be `ALLOW_TYPE_COERCION`. However, when the Service is introspecting the built-in topic data declaration of a remote `DataWriter` or `DataReader` in order to determine whether it can match with a local reader or writer, if it observes that no `TypeConsistencyEnforcementQosPolicy` value is provided (as would be the case when communicating with a Service implementation not in conformance with this specification), it shall assume a kind of `DISALLOW_TYPE_COERCION`[7]. This behavior is consistent with the type

[7] **Design rationale (non-normative)**: This behavior is critical to ensure that conformant and non-conformant Service implementations reach the same conclusion regarding whether or not a `DataWriter` and a given `DataReader` are using consistent types.

member defaulting rules defined in Clause 7.2.3.2.4.4.7, which state that unspecified values of enumerated types take the first value defined for their type.

The policy defines a number of additional fields, see below, each corresponding one of the Type Assignability Options defined in 7.2.5.4. Their semantics shall be exactly those defined in the referenced subclauses:

- prevent_type_widening – see 7.2.5.4.3
- ignore_sequence_bounds – see 7.2.5.4.4
- ignore_string_bounds – see 7.2.5.4.5
- ignore_member_names – see 7.2.5.4.6
- ignore_literal_names – see 7.2.5.4.7
- ignore_units – see 7.2.5.4.8
- ignore_unspecified_units – see 7.2.5.4.9
- ignore_ranges – see 7.2.5.4.10
- use_xtypes_1_3_union_assignability – see **Error! Reference source not found.**.

7.6.3.4.2 Rules for Type Consistency Enforcement

Implementations of this specification shall use the type-consistency-enforcement rules defined in this clause when matching a `DataWriter` with a `DataReader`, each associated with a `Topic` of the same name. These rules are based on the data types of these entities and on the type consistency kind of the `DataReader`.

The type-consistency-enforcement rules consist of two steps.

Step 1. If both the Publication and the Subscription specify a `TypeObject`, consider it first. If the Subscription allows type coercion, then `the_type` indicated there must be assignable from `the_type` of the Publication, taking into account the values of `prevent_type_widening`, `ignore_sequence_bounds`, `ignore_string_bounds`, and `ignore_member_names`. If the Subscription does not allow type coercion, then its type must be equivalent to the type of the Publication.

If the subscription allows type coercion and the `ignore_member_names` flag is true in `TypeConsistencyEnforcementQoSPolicy`, assignability checking shall ignore the member names in both Subscription and Publication types. That is, only member IDs will impact assignability. Likewise, if `ignore_literal_names` flag is true, assignability checking shall ignore the enumeration literal names in both Subscription and Publication types.

Step 2. If either the Publication or the Subscription does not provide a `TypeObject` definition, then the type names are consulted. The Subscription and Publication `type_name` fields must match exactly, as in [DDS] prior to this specification. This step will fail if `force_type_validation` is true, regardless of the type names.

If either Step 1 or Step 2 fails, then the `Topic`s associated with the `DataWriter` and `DataReader` are considered to be inconsistent: the `DataWriter` and `DataReader` shall not communicate with each other, and the Service shall trigger an `INCONSISTENT_TOPIC` status change for both the `DataReader`'s `Topic` and the `DataWriter`'s `Topic`.

If both Step 1 and Step 2 succeed, then the `Topic`s are considered to be consistent, and the matching shall proceed to check other aspects of endpoint matching, such as the compatibility of the QoS, as defined by the DDS specification.

Note that the `DataWriter` and the `DataReader` can each execute the algorithm independently, having access to its own metadata as well as that of the other endpoint as communicated via DDS discovery (see Clause 7.6.4). Moreover, the algorithm is such that both sides are guaranteed to arrive at the same conclusion. That is, either both succeed or both fail.

7.6.3.4.3 `TypeConsistencyEnforcementQosPolicy`: Platform-Specific API

The conceptual model defined above shall be transformed into the IDL definitions `TypeConsistencyKind`, `ignore_member_names`, `ignore_literal_names`, `TYPE_CONSISTENCY_ENFORCEMENT_QOS_POLICY_ID`, `TYPE_CONSISTENCY_ENFORCEMENT_QOS_POLICY_NAME`, and `TypeConsistencyEnforcementQosPolicy`. These definitions are given in "Annex D: DDS Built-in Topic Data Types."

The subscription built-in topic data type shall indicate the type consistency requirements of the corresponding reader:

```
@id(0x0074) DDS::TypeConsistencyEnforcementQosPolicy type_compatibility;
```

7.6.4 Local API Extensions

The following sub clauses define changes in behavior to existing operations defined by [DDS].

7.6.4.1 Operation: `DomainParticipant::create_topic`

As defined in [DDS], a local `Topic` object is uniquely identified by its name. In implementations conforming to this specification, that restriction shall be removed. The Service may instantiate multiple objects of the same name, provided that all of them represent type-based subsets of "the same" network topic; therefore, they must have consistent QoS with one another.

7.6.4.2 Operation: `DomainParticipant::lookup_topicdescription`

As defined in [DDS], a local `TopicDescription` object is uniquely identified by its name. In implementations conforming to this specification, that restriction shall be removed. The definition of `lookup_topicdescription` operation shall be modified from the one in the [DDS] specification as follows.

The `lookup_topicdescription` operation shall accept an optional *in* `unsigned long` argument called `index`. This shall be the last argument.

When the operation is called with only `topic_name`. It shall behave as if called with `index` = 0.

When the operation is called with both a `topic_name` and an `index`, the operation shall return one of the `TopicDescription` associated with the `DomainParticipant` with a matching `topic_name`. The value of the `index` parameter shall be treated as an "iterator" over the sequence of `TopicDescription` instances that match that `topic_name`. Each value of the index shall return a unique (different) `TopicDescription`. Values of the `index` from 0 to one less than the number of different `TopicDescriptions` match the `topic_name` shall return a `TopicDescription` and values of the index outside the range shall return *nil*.

7.6.5 Built-in Types

DDS shall provide a few types preregistered "out of the box" to allow users to address certain simple use cases without the need for code generation, dynamic type definition, or type registration. These types are:

- **DDS::String**: A single unbounded string; a data type without a key.
- **DDS::KeyedString**: A pair of unbounded strings, one representing the payload and a second representing its key.
- **DDS::Bytes**: An unbounded sequence of bytes, useful for transmitting opaque or application-serialized data.
- **DDS::KeyedBytes**: A payload consisting of an unbounded sequence of bytes plus a key field, an unbounded string.

The built-in types shall be defined as in the following sections and shall be automatically registered by the Service under their fully qualified physical names (as above) with each `DomainParticipant` at the time it is enabled.

Like all non-nested types used with DDS, the built-in types shall have corresponding type-specific `DataWriter` and `DataReader` classes. These shall instantiate the type-specific operations defined by the DDS specification as defined in the following sections; they shall also provide additional overloads.

The built-in types are described briefly below; their complete definitions may be found in "Annex E: Built-in Types."

7.6.5.1 String

The `DDS::String` type is a simple structure wrapper around a single unbounded string. The wrapper structure exists in order to provide the Service implementation with a non-nested type definition and as a basis of the `TypeObject` object propagated with the built-in topics. But the `StringDataWriter` and `StringDataReader` APIs are defined based on the built-in `string` type for convenience.

7.6.5.2 KeyedString

The `DDS::KeyedString` type is similar to `DDS::String`, but it is a keyed type; the key is an additional unbounded string. `DDS::KeyedStringDataWriter` provides additional overloads that "unwrap" this structure, allowing applications to pass the two strings directly.

7.6.5.3 Bytes

The `DDS::Bytes` type is a simple structure wrapper around a single unbounded sequence of bytes. The wrapper structure exists in order to provide the Service implementation with a non-nested type definition and as a basis of the `TypeObject` object propagated with the built-in topics. The `BytesDataWriter` API is defined based on the underlying sequence for convenience; the `BytesDataReader` API is based on `DDS::Bytes` because of the awkwardness of sequences of sequences.

7.6.5.4 KeyedBytes

The `DDS::KeyedBytes` type is similar to `DDS::Bytes`, but it is a keyed type; the key is an unbounded string. `DDS::KeyedBytesDataWriter` provides additional overloads that "unwrap" this structure, allowing applications to pass the string and sequence directly.

7.6.6 Use of Dynamic Data and Dynamic Type

Using the `DynamicData` and `DynamicType` APIs applications can publish and subscribe data of any type without having compile-type knowledge of the type.

The API is still strongly typed; each specific Type must be registered with the `DomainParticipant`. The `DynamicType` interface can be used to construct the Type and register it with the `DomainParticipant`. The `DynamicData` interface can be used to create objects of a specified Type (expressed by means of a `DynamicType`) and publish and subscribe data objects of that type.

In order to for an application to use a type for publication or subscription the type must first be registered with the corresponding `DomainParticipant` in the same manner as a type defined at compile time.

7.6.6.1 Type Support

Application code (i.e., business logic) generally depends statically on particular types and their members. In contrast, infrastructure code (i.e., logic that is independent of particular applications) generally must not depend on application-specific types, because such dependencies prevent that code from being reused. These two kinds of code can exist within a single component.

Therefore, it is desirable to allow conversions among static and dynamic bindings for the same types and samples. These conversions shall be provided by operations on the generic `TypeSupport` interface and its extended interfaces.

7.6.6.1.1 `TypeSupport` Interface

The following operations shall be added to the `TypeSupport` interface defined by [DDS]. (The operations on this interface already defined in [DDS] are unchanged.)

Table 60 – New `TypeSupport` operations

Operations		
`get_type`		`DynamicType`

7.6.6.1.1.1 Operation: `get_type`

Get a `DynamicType` object corresponding to the `TypeSupport`'s data type.

7.6.6.1.2 `FooTypeSupport` Interface

The following operations shall be added to the `FooTypeSupport` interface defined by [DDS]. (The operations on this interface already defined in [DDS] are unchanged.)

Table 61 – New `FooTypeSupport` operations

Operations		
create_sample		Foo
	src	DynamicData
create_dynamic_sample		DynamicData
	src	Foo

7.6.6.1.2.1 Operation: `create_sample`

Create a sample of the `TypeSupport`'s data type with the contents of an input `DynamicData` object.

Parameter `src` – The source object whose contents are to be reflected in the resulting object. This method shall fail with a nil return result if this object is nil or if the `DynamicType` of this object is not compatible with the `TypeSupport`'s data type.

7.6.6.1.2.2 Operation: `create_dynamic_sample`

Create a `DynamicData` object with the contents of an input sample of the `TypeSupport`'s data type.

Parameter `src` – The source object whose contents are to be reflected in the resulting object. This method shall fail with a nil return result if this object is nil.

7.6.6.1.3 DynamicTypeSupport

The `DynamicTypeSupport` interface extends the `FooTypeSupport` interface defined by the DDS specification where "Foo" is the type `DynamicData`.

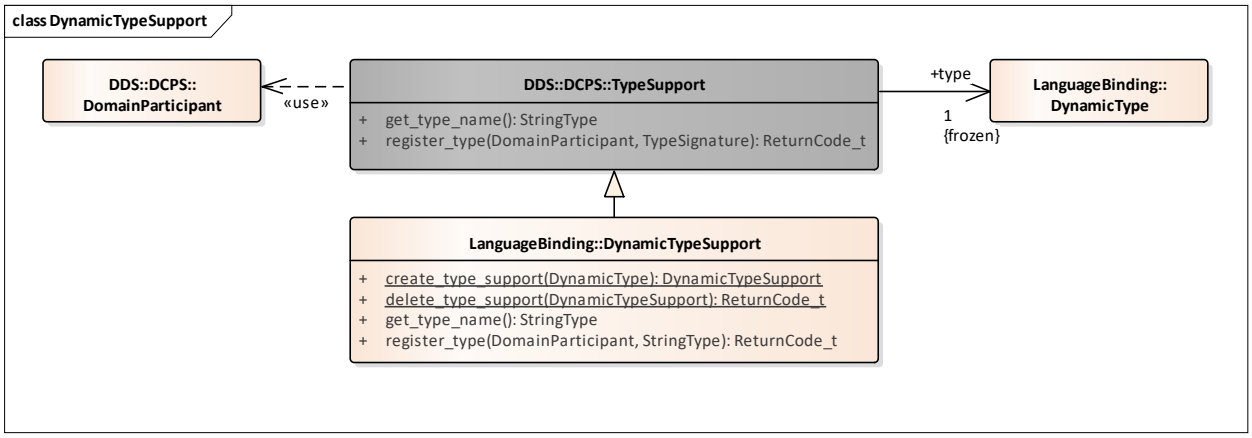

Figure 33 – Dynamic Type Support

Table 62 – DynamicTypeSupport properties and operations

DynamicTypeSupport		
Operations		
register_type		ReturnCode_t
	participant	DomainParticipant
	type_name	string<Char8,256>
get_type_name		string<Char8,256>
static create_type_support		DynamicTypeSupport
	type	DynamicType
static delete_type_support		ReturnCode_t
	support	DynamicTypeSupport

7.6.6.1.4 Operations: `register_type`, `get_type_name`

These operations are defined by, and described in, the DDS specification.

7.6.6.1.5 Operation: `create_type_support`

Create and return a new `DynamicTypeSupport` object capable of registering the given type with DDS `DomainParticipant`s. The implementation shall ensure that the new type support has a "copy" of the given type object, such that subsequent changes to, or deletions of, the argument object do not impact the new type support. All objects returned by this operation should eventually be deleted by calling `delete_type_support`.

If an error occurs, this method shall return a nil value.

Parameter `type` - The type for which to create a type support. If this argument is nil or is a nested type, the operation shall fail and return a nil value.

7.6.6.1.6 Operation: `delete_type_support`

Delete the given type support object, which was previously created by this factory.

If this argument is nil, the operation shall return successfully without having any observable effect.

Parameter `type_support` – The type support object to delete. If this argument is an object that was already deleted, and the implementation is able to detect that fact (which is not required), this operation shall fail with `RETCODE_ALREADY_DELETED`. If an implementation-specific error occurs, this method shall fail with `RETCODE_ERROR`.

7.6.6.2 `DynamicDataWriter` and `DynamicDataReader`

The `DynamicDataWriter` interface instantiates the `FooDataWriter` interface defined by the DDS specification where "Foo" is the type `DynamicData`.

The `DynamicDataReader` interface instantiates the `FooDataReader` interface defined by the DDS specification where "Foo" is the type `DynamicData`.

These types do not define additional properties or operations.

7.6.7 DCPS Queries and Filters

[DDS] defines the syntax for content-based filters, queries, and joins in "*Annex A: Syntax for DCPS Queries and Filters*". This syntax shall be extended as follows.

7.6.7.1 Member Names

[DDS] Clause A.2 defines the syntax for referring to a member of a (potentially nested) data structure. Such a reference is known as a `FIELDNAME`. The syntax shall be extended as follows:

Arrays and sequences: Elements in these ordered collections shall be indicated by a zero-based subscript enclosed in square brackets, e.g. `my_collection[0]`. Such an expression shall be considered to have the type that is the element type of the collection.

- *Maps*: Value elements in these unordered collections shall be indicated by a string representation of a corresponding key element, according to the syntax of `STRING`, enclosed in square brackets, e.g. `my_map['key']`. The key shall be expressed as a string even if the map's key type is an integer type; this distinguishes a map lookup from an index into an ordered collection. Such an expression shall be considered to have the type that is the value element type of the map.
- *Bitmasks*: A flag in a bitmask shall be indicated by its name, according to the syntax of `ENUMERATEDVALUE`, enclosed in square brackets, e.g. `my_bitmask['MY_FLAG']`. Such an expression shall be considered to have a Boolean type: true if the bit is set or false if it is not. Comparisons with the integer literals 1 and 0 shall also be allowed.

- *Unions*: Union members shall be referenced using the name of the member. The union discriminator may also be referenced using the name "`#discriminator`". The evaluation of an expression involving a union member shall also check that the value of the discriminator selects the named member; if the discriminator does not select the named member, the result of the filter evaluation shall be false.

7.6.7.2 Optional Type Members

A member of an aggregated type may be compared to the special value `null`. Such comparisons obey the following rules:

- If the member is optional, and it takes no value in the given object, it shall be considered equal to `null`.
- If the member is optional, and it does take a value in the given object, it shall not be considered equal to `null`.
- No non-optional member shall ever be considered equal to `null`.

Inequalities expressed relative to null shall never evaluate to true—no value is greater than or less than `null`.

7.6.7.3 Grammar Extensions

The `Parameter` production in the grammar given in [DDS] Clause A.1 shall be redefined as follows:

```
Parameter ::=
          | CHARVALUE
          | FLOATVALUE
          | STRING
          | ENUMERATEDVALUE
          | BOOLEANVALUE
          | NULLVALUE
          | PARAMETER
          .
```

(New tokens have been highlighted in **bold**.)

The `BOOLEANVALUE` token shall be either `true` or `false` (case-insensitive).

The `NULLVALUE` token shall always be `null`.

7.6.8 Interoperability of Keyed Topics

7.6.8.1 Keyhash Computation

The key hash for a given object of a keyed type shall be computed using the algorithm described in RTPS 2.5, Clause 9.6.4.8, “KeyHash (PID_KEY_HASH)” (see [RTPS]).

7.6.8.2 Serialized representation of the key fields

The following algorithm shall be used when computing the serialized representation of the value of the key of an application-defined data object for the purpose of inclusion in the RTPS SerializedPayload submessage element (see RTPS 2.5 section 9.4.2.12).

Given an AggregatedType "Foo" and an object “FooObject” of type “Foo”, the Key serialization for FooObject shall use the following algorithm:

Step 1. Define a new type “FooKeyHolder” as follows:

- Start with FooKeyHolder being defined the same way as the original Foo type.
- Remove the non-key members from FooKeyHolder.

Step 2. Define a new object “FooKeyHolderObject” from the FooObject, by setting the members present in FooKeyHolderObject to the same values as the corresponding members in FooObject.

Step 3. Apply steps 1 and 2 recursively to the members of FooKeyHolder if they are themselves AggregatedTypes.

Step 4. Serialize the FooKeyHolderObject using the same DataRepresentation used to serialize the FooObject. The serialization shall be performed on a buffer that is initially aligned to the maximum alignment for the type of serialization used.

Annex A: XML Type Representation Schema

This specification reuses the schema definitions specified in the DDS-XML specification, version 1.2 [DDS-XML]. In particular, it reuses the following schemas:

- https://www.omg.org/spec/DDS-XML/20260301/dds-xml_type_definitions_nonamespace.xsd
- https://www.omg.org/spec/DDS-XML/20260301/dds-xml_type_definitions.xsd

The `dds-xml_type_definitions_nonamespace.xsd` schema does not declare a `targetNamespace`. This design allows other specifications to include the schema without requiring explicit namespace declarations. This approach follows the *Chameleon Namespace* design pattern, in which a schema with no `targetNamespace` takes on the target namespace of the schema that includes it.

The `dds-xml_type_definitions.xsd` schema includes the "nonamespace" schema, declares the appropriate target namespace for use with DDS (that is, `http://www.omg.org/dds`), and defines the root element for XML documents that contain DDS type definitions.

Annex B: Representing Types with TypeObject

The TypeObject is formally described by the IDL contained in the normative, machine-readable file `dds-xtypes_typeobject.idl` that accompanies this specification.

This file can be accessed at the following URL:

https://www.omg.org/spec/DDS-XTypes/20260301/dds-xtypes_typeobject.idl

Annex C: Dynamic Language Binding

The Dynamic Language Binding is formally described by the IDL contained in the normative, machine-readable file `dds-xtypes_dynamicdata.idl` that accompanies this specification.

This file can be accessed at the following URL:

https://www.omg.org/spec/DDS-XTypes/20260301/dds-xtypes_dynamicdata.idl

Annex D: DDS Built-in Topic Data Types

Previously, the standard DDS type system (based solely on IDL prior to the extensions introduced by this specification) was insufficiently rich to represent the built-in topic data to the level specified by DDS [DDS] and RTPS [RTPS]. This specification remedies this situation.

The following are expanded definitions of the built-in topic data types that contain all of the meta-data necessary to represent them as defined by the existing DDS and RTPS specifications.

The machine-readable file `dds-xtypes_discovery_builtin_topics.idl` that accompanies this specification contains the expanded definitions of the built-in topic data types that contain all of the meta-data necessary to represent them as defined by the existing DDS and RTPS specifications.

This file can be accessed at the following URL:

https://www.omg.org/spec/DDS-XTypes/20260301/dds-xtypes_discovery_builtin_topics.idl

Annex E: Built-in Types

DDS shall provide a few very types preregistered “out of the box” to allow users to address certain simple use cases without the need for code generation, dynamic type definition, or type registration. These types are defined below[8].

```
module DDS {
    @extensibility(APPENDABLE)
    struct _String {
        string value;
    };

    interface StringDataWriter : DataWriter {
        /* This interface shall instantiate the type FooDataWriter defined by
         * the DDS specification where "Foo" is an unbounded string.
         */
    };

    interface StringDataReader : DataReader {
        /* This interface shall instantiate the type FooDataReader defined by
         * the DDS specification where "Foo" is an unbounded string.
         */
    };

    interface StringTypeSupport : TypeSupport {
        /* This interface shall instantiate the type FooTypeSupport
         * defined by the DDS specification where "Foo" is an unbounded
         * string.
         */
    };

    @extensibility(APPENDABLE)
    struct KeyedString {
```

[8] The leading underscore in the declaration of the `String` structure is necessary to prevent collision with the IDL keyword “string.” According to the IDL specification, it is treated as an escaping character and is not considered part of the identifier.

```
    @key string key;
    string value;
};
typedef sequence<KeyedString> KeyedStringSeq;

interface KeyedStringDataWriter : DataWriter {
    /* This interface shall instantiate the type FooDataWriter defined by
     * the DDS specification where "Foo" is KeyedString. It also defines
     * the operations below.
     */
    InstanceHandle_t register_instance_w_key(
        in string key);
    InstanceHandle_t register_instance_w_key_w_timestamp(
        in string key,
        in Time_t source_timestamp);

    ReturnCode_t unregister_instance_w_key(
        in string key);
    ReturnCode_t unregister_instance_w_key_w_timestamp(
        in string key,
        in Time_t source_timestamp);

    ReturnCode_t write_string_w_key(
        in string key,
        in string str,
        in InstanceHandle_t handle);
    ReturnCode_t write_string_w_key_w_timestamp(
        in string key,
        in string str,
        in InstanceHandle_t handle,
        in Time_t source_timestamp);

    ReturnCode_t dispose_w_key(
```

```
        in string key);
    ReturnCode_t dispose_w_key_w_timestamp(
        in string key,
        in Time_t source_timestamp);

    ReturnCode_t get_key_value_w_key(
        inout string key,
        in InstanceHandle_t handle);

    InstanceHandle_t lookup_instance_w_key(
        in string key);
};

interface KeyedStringDataReader : DataReader {
    /* This interface shall instantiate the type FooDataReader defined by
     * the DDS specification where "Foo" is KeyedString.
     */
};

interface KeyedStringTypeSupport : TypeSupport {
    /* This interface shall instantiate the type FooTypeSupport
     * defined by the DDS specification where "Foo" is KeyedString.
     */
};


@extensibility(APPENDABLE)
struct Bytes {
    ByteSeq value;
};
typedef sequence<Bytes> BytesSeq;

interface BytesDataWriter : DataWriter {
```

```
    /* This interface shall instantiate the type FooDataWriter defined by
     * the DDS specification where "Foo" is an unbounded sequence of
     * bytes (octets). It also defines the operations below.
     */
    ReturnCode_t write_w_bytes(
          in ByteArray bytes,
          in long offset,
          in long length,
          in InstanceHandle_t handle);
    ReturnCode_t write_w_bytes_w_timestamp(
          in ByteArray bytes,
          in long offset,
          in long length,
          in InstanceHandle_t handle,
          in Time_t source_timestamp);
};

interface BytesDataReader : DataReader {
    /* This interface shall instantiate the type FooDataReader defined by
     * the DDS specification where "Foo" is Bytes.
     */
};

interface BytesTypeSupport : TypeSupport {
    /* This interface shall instantiate the type FooTypeSupport
     * defined by the DDS specification where "Foo" is Bytes.
     */
};


@extensibility(APPENDABLE)
struct KeyedBytes {
    @key string key;
```

```
    ByteSeq value;
};
typedef sequence<KeyedBytes> KeyedBytesSeq;

interface KeyedBytesDataWriter : DataWriter {
    /* This interface shall instantiate the type FooDataWriter defined by
     * the DDS specification where "Foo" is KeyedBytes. It also defines
     * It also defines the operations below.
     */
    InstanceHandle_t register_instance_w_key(
        in string key);
    InstanceHandle_t register_instance_w_key_w_timestamp(
        in string key,
        in Time_t source_timestamp);

    ReturnCode_t unregister_instance_w_key(
        in string key);
    ReturnCode_t unregister_instance_w_key_w_timestamp(
        in string key,
        in Time_t source_timestamp);

    ReturnCode_t write_bytes_w_key(
          in string key,
          in ByteArray bytes,
          in long offset,
          in long length,
          in InstanceHandle_t handle);
    ReturnCode_t write_bytes_w_key_w_timestamp(
          in string key,
          in ByteArray bytes,
          in long offset,
          in long length,
          in InstanceHandle_t handle,
```

```
            in Time_t source_timestamp);

        ReturnCode_t dispose_w_key(
            in string key);
        ReturnCode_t dispose_w_key_w_timestamp(
            in string key,
            in Time_t source_timestamp);

        ReturnCode_t get_key_value_w_key(
            inout string key,
            in InstanceHandle_t handle);

        InstanceHandle_t lookup_instance_w_key(
            in string key);
    };

    interface KeyedBytesDataReader : DataReader {
        /* This interface shall instantiate the type FooDataReader defined by
         * the DDS specification where "Foo" is KeyedBytes.
         */
    };

    interface KeyedBytesTypeSupport : TypeSupport {
        /* This interface shall instantiate the type FooTypeSupport
         * defined by the DDS specification where "Foo" is KeyedBytes.
         */
    };
};  // end module DDS
```

Annex F: Characterizing Legacy DDS Implementations

Prior to the adoption of this specification, no formal definition existed of the DDS Type System or of those portions of IDL that corresponded to it. This annex provides a non-normative description of what is believed to be the consensus Type System, Type Representation, Data Representation, and Language Binding of DDS implementations that do not conform to this specification. It is provided for the convenience of implementers and evaluators who may wish to compare and contrast DDS implementations or to distinguish those parts of this specification that are novel from those that merely codify previous de-facto-standard practice.

F.1 Type System

The following portions of the Type System are believed to be supported by the majority of DDS implementations, regardless of their compliance with this specification:

- *Namespaces and modules*.
- *All primitive types* except for int8 and uint8, albeit named according to their mappings in the IDL Type Representation.
- *Enumerations* of bit bound 32 with automatically assigned enumerator values.
- *Aliases*, typically referred to as "`typedef`s" based on their mappings in the IDL Type Representation.
- *Arrays*, both single-dimensional and multi-dimensional.
- *Sequences*, both bounded and unbounded.
- *Strings* of narrow or wide characters, both bounded and unbounded.
- *Structures* without inheritance. User-defined structures have `FINAL` extensibility. Members are typically non-optional, non-shared, and do not expose member IDs. DDS-RTPS-compliant implementations support `MUTABLE` extensibility and the `must_understand` attribute with respect to the built-in topic data types. Otherwise, these attributes are not generally supported. Key members are generally supported.
- *Unions* with `FINAL` extensibility and without key members. Discriminators of wide character and octet types are not generally supported.

F.2 Type Representation

The IDL Type Representations of those portions of the Type System enumerated above are generally supported.

F.3 Data Representation

The Extended CDR Representations of those portions of the Type System enumerated above are generally supported. The exception is the extended parameter ID and length facility based on `PID_EXTENDED`, which is not generally supported.

F.4 Language Binding

The Plain Language Bindings of those portions of the Type System enumerated above are generally supported.