RTI Connext

Core Libraries and Utilities

Getting Started Guide

Addendum for Extensible Types

Version 5.0

Your systems. Working as one.

Trademarks

Copy and Use Restrictions

Technical Support

Real-Time Innovations, Inc.
232 E. Java Drive
Sunnyvale, CA 94089

Phone:	(408) 990-7444
Email:	support@rti.com
Website:	https://support.rti.com/

Contents

Getting Started with Extensible Types

1 Introduction

This release of *RTI Connext* includes partial support for the "Extensible and Dynamic Topic Types for DDS" (DDS-XTypes) specification[1] from the Object Management Group (OMG). This support allows systems to define data types in a more flexible way, and to evolve data types over time without giving up portability, interoperability, or the expressiveness of the DDS type system.

Specifically, these are now supported:

- Type definitions are now checked as part of the *Connext* discovery process to ensure that *DataReaders* will not deserialize the data sent to them incorrectly.
- Type definitions need not match exactly between a *DataWriter* and its matching *DataReaders*. For example, a *DataWriter* may publish a subclass while a *DataReader* subscribes to a superclass, or a new version of a component may add a field to a preexisting data type.
- Data-type designers can annotate type definitions to indicate the degree of flexibility allowed when the middleware enforces type consistency.
- The above features are supported in the RTI core middleware in the C, C++, Java, and .NET programming languages.

The following Extensible Types features are not supported:

- These types: BitSet, Map
- These built-in annotations: ID, Optional, BitSet, Verbatim
- Annotation definition
- Standard syntax to apply annotations (see Section 7)
- Mutable types
- Optional members
- Member IDs
- XML data representation
- Dynamic language binding compliant with the Extensible Types specification: DynamicType and DynamicData.
- DataRepresentationQosPolicy

1. http://www.omg.org/spec/DDS-XTypes/

- ❏ type member in PublicationBuiltinTopicData and SubscriptionBuiltinTopicData
- ❏ Association of a topic to multiple types within a single *DomainParticipant*

To see a demonstration of Extensible Types, run *RTI Shapes Demo*, which can publish and subscribe to two different data types: the "Shape" type or the "Shape Extended" type. If you don't have *Shapes Demo* installed already, you can download it from RTI's Downloads page (www.rti.com/downloads) or the RTI Customer Portal (https://support.rti.com/). The portal requires an account name and password. If you are not already familiar with how to start *Shapes Demo*, please see the *Shapes Demo User's Manual*.

Besides *RTI Shapes Demo*, several other RTI components include partial support for Extensible Types.

2 Type Safety and System Evolution

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, or create extended types using inheritance. 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.

To support use cases such as the above, the type system introduces the concept of extensible and mutable types. A type may be final, extensible, or mutable:

- ❏ **Final**: The type's range of possible data values is strictly defined. In particular, it is not possible to add elements to members of a collection or aggregated types while maintaining type assignability.
- ❏ **Extensible:** Two types, where one contains all of the elements/members of the other plus additional elements/members appended to the end, may remain assignable.
- ❏ **Mutable:** Two types may differ from one another with the addition, removal, and/or transposition of elements/members while remaining assignable.

  For example, suppose you have:

  ```
  struct A {
      long a;
      long b;
      long c;
  }
  ```

  and

  ```
  struct B {
      long b;
      long a;
      long x;
  }
  ```

  In this case, if a *DataWriter* writes [1, 2, 3], the *DataReader* will receive [2, 1, 0] (because 0 is the default value of x, which doesn't exist in A's sample).

Important: *This release does not provide support for mutable types.*

The type being written and the type(s) being read may differ—maybe because the writing and reading applications have different needs, or maybe because the system and its data design have evolved across versions. Whatever the reason, the data bus must detect the differences and mediate them appropriately. This process has several steps:

1. Define what degree of difference is acceptable for a given type.
2. Express your intention for compatibility at run time.
3. Verify that the data can be safely converted.

At run time, the data bus will compare the types it finds with the contracts you specified.

2.1 Defining Extensible Types

A type's kind of extensibility is applied with the **Extensibility** annotations seen in Table 2.1. If you do not specify any particular extensibility, the default is extensible.

Table 2.1 **Extensibility Annotations**

IDL	`struct MyFinalType {`
	`   long x;`
	`}; //@Extensibility FINAL_EXTENSIBILITY`
	`struct MyExtensibleType {`
	`   long x;`
	`}; //@Extensibility EXTENSIBLE_EXTENSIBILITY`
XML	`<struct name="MyFinalType" extensibility="final">`
	`   <member name="x" type="long"/>`
	`</struct>`
	`<struct name="MyExtensibleType" extensibility="extensible">`
	`   <member name="x" type="long"/>`
	`</struct>`
XSD	`<xsd:complexType name="MyFinalType">`
	`  <xsd:sequence>`
	`  <xsd:element name="x" minOccurs="1" maxOccurs="1" type="xsd:int"/>`
	`  </xsd:sequence>`
	`</xsd:complexType>`
	`<!-- @struct true -->`
	`<!-- @extensibility FINAL_EXTENSIBILITY -->`
	`<xsd:complexType name="MyExtensibleType">`
	`   <xsd:sequence>`
	`      <xsd:element name="x" minOccurs="1" maxOccurs="1" type="xsd:int"/>`
	`   </xsd:sequence>`
	`</xsd:complexType>`
	`<!-- @struct true -->`
	`<!-- @extensibility EXTENSIBLE_EXTENSIBILITY -->`

2.2 Verifying Type Consistency: Type Assignability

Connext determines if a *DataWriter* and a *DataReader* can communicate by comparing the structure of their topic types.

In previous *Connext* releases, the topic types were represented and propagated on the wire using TypeCodes. The Extensible Types specification introduces TypeObjects as the wire representation for a type.

To maintain backward compatibility, *Connext* can be configured to propagate both TypeCodes *and* TypeObjects. However, type comparison is only supported with TypeObjects.

Depending on the value for extensibility annotation used when the type is defined, *Connext* will use a different set of rules to determine if matching shall occur. In this release, only the assignability of extensible and final types is enforced.

If the type extensibility is *final*, the types will be assignable if they are structurally identical. If they are declared as *extensible*, one type can have more fields at the end as long as they are not keys. The common fields have to be identical, with the following allowed exceptions:

- ❏ Sequences and strings can have a different maximum length
- ❏ Enumerations can be extended, adding new constants at the end

In the case of union types, it has to be possible, given any possible discriminator value in the *DataWriter's* type (T2), to identify the appropriate member in the *DataReader's* type (T1) and to transform the T2 member into the T1 member. If T2 has a default case, then T1 and T2 have to be equal.

For more information on the rules that determine the assignability of two types, refer to the DDS-XTypes specification[1].

By default, the TypeObjects are compared to determine if they are assignable in order to match a *DataReader* and a *DataWriter* of the same topic. You can control this behavior in the *DataReader's* TypeConsistencyEnforcementQosPolicy (see Type-Consistency Enforcement (Section 2.3)).

The *DataReader's* and *DataWriter's* TypeObjects need to be available in order to be compared; otherwise their assignability will not be enforced. Depending on the complexity of your types (how many fields, how many different nested types, etc.), you may need to change the default resource limits that control the internal storage and propagation of the TypeObject (see TypeObject Resource Limits (Section 5)).

If the logging verbosity of is set to NDDS_CONFIG_LOG_VERBOSITY_WARNING or higher, *Connext* will print a message when a type is discovered that is not assignable, along with the reason why the type is not assignable.

2.3 Type-Consistency Enforcement

The TypeConsistencyEnforcementQosPolicy defines the rules that determine whether the type used to publish a given data stream is consistent with that used to subscribe to it.

The QosPolicy structure includes the member in Table 2.1.

This QoSPolicy defines a type consistency **kind**, which allows applications to choose either of these behaviors:

- ❏ DISALLOW_TYPE_COERCION: The *DataWriter* and *DataReader* must support the same data type in order for them to communicate. (This is the degree of type consistency enforcement required by the DDS specification prior to the Extensible Types specification).
- ❏ ALLOW_TYPE_COERCION: The *DataWriter* and *DataReader* need not support the same data type in order for them to communicate as long as the *DataReader*'s type is assignable from the *DataWriter's* type. The concept of assignability is explained in section Section 2.2.

This policy applies only to *DataReaders*; it does not have request-offer semantics. The value of the policy cannot be changed after the *DataReader* has been enabled.

The default enforcement kind is ALLOW_TYPE_COERCION. However, when the middleware 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 *DataReader* or *DataWriter*, if it observes that no

1. http://www.omg.org/spec/DDS-XTypes/

TypeConsistencyEnforcementQosPolicy value is provided (as would be the case when communicating with a Connext implementation not in conformance with this specification), the middleware assumes a kind of DISALLOW_TYPE_COERCION.

Table 2.1 **DDS_TypeConsistencyEnforcementQosPolicy**

Type	Field Name	Description
DDS_TypeConsistencyKind	kind	Can be either: • DISALLOW_TYPE_COERCION: The *DataWriter* and *DataReader* must support the same data type in order for them to communicate. • ALLOW_TYPE_COERCION: The *DataWriter* and *DataReader* need not support the same data type in order for them to communicate, as long as the reader's type is assignable from the writer's type. (Default)

2.3.1 Rules For Type-Consistency Enforcement

The type-consistency enforcement rules consist of two steps applied on the *DataWriter* and *DataReader* side:

Step 1. If both the *DataWriter* and *DataReader* specify a TypeObject, it is considered first. If the DataReader allows type coercion, then its type must be assignable from the *DataWriter's* type. If the *DataReader* does not allow type coercion, then its type must be structurally identical to the type of the *DataWriter*.

Step 2. If either the *DataWriter* or the *DataReader* does not provide a TypeObject definition, then the registered type names are examined. The *DataReader's* and *DataWriter's* registered type names must match exactly, as was true in *Connext* releases prior to 5.0.0.

If either Step 1 or Step 2 fails, the *Topics* associated with the *DataReader* and *DataWriter* are considered to be inconsistent (see Section 2.4).

2.4 Notification of Inconsistencies: INCONSISTENT_TOPIC Status

Every time a *DataReader* and a *DataWriter* do not match because the type-consistency enforcement check fails, the INCONSISTENT_TOPIC status is increased.

Notice that the condition under which the middleware triggers an INCONSISTENT_TOPIC status update has changed (starting in release 5.0.0) with respect to previous *Connext* releases where the change of status occurred when a remote *Topic* inconsistent with the locally created *Topic* was discovered.

2.5 Built-in Topics

The type consistency value used by a *DataReader* can be accessed using the **type_consistency** field in the DDS_SubscriptionBuiltinTopicData (see Table 2.2).

Table 2.2 **New Field in Subscription Builtin Topic Data**

Type	New Field	Description
DDS_TypeConsistency EnforcementQosPolicy	type_consistency	Indicates the type_consistency requirements of the remote *DataReader* (see Section 2.3).

You can retrieve this information by subscribing to the built-in topics and using the *DataReader's* **get_matched_publication_data()** operations.

3 Type System Enhancements

3.1 Structure Inheritance

A structure can define a base type as seen in Table 3.1. Note that when the types are extensible, MyBaseType is assignable from MyDerivedType, and MyDerivedType is assignable from MyBaseType.

Table 3.1 **Base Type Definition in a Structure**

IDL	`struct MyBaseType {`
	`  long x;`
	`};`
	`struct MyDerivedType : MyBaseType {`
	`  long y;`
	`};`
XML	`<struct name="MyBaseType">`
	`    <member name="x" type="long"/>`
	`</struct>`
	`<struct name=" MyDerivedType" baseType="MyBaseType">`
	`    <member name="y" type="long"/>`
	`</struct>`
XSD	`<xsd:complexType name="MyBaseType">`
	`  <xsd:sequence>`
	`<xsd:element name="x" minOccurs="1" maxOccurs="1" type="xsd:int"/>`
	`   </xsd:sequence>`
	`</xsd:complexType>`
	`<!-- @struct true -->`
	`<xsd:complexType name="MyDerivedType">`
	`  <xsd:complexContent>`
	`    <xsd:extension base="tns:MyBaseType">`
	`      <xsd:sequence>`
	`      <xsd:element name="y" minOccurs="1" maxOccurs="1" type="xsd:int"/>`
	`      </xsd:sequence>`
	`    </xsd:extension>`
	`  </xsd:complexContent>`
	`</xsd:complexType>`
	`<!-- @struct true -->`

Starting with *Connext* 5.0, value types are equivalent to structures. You can still use value types, but support for this feature may be discontinued in future releases.

For example:

```
struct MyType {
    long x;
};

valuetype MyType {
    public long x;
};
```

The above two types are considered equivalent. Calling the method **equal()** in their TypeCodes will return true. Calling the method **print_IDL()** in the valuetype's TypeCode will print the value type as a struct.

4 Type Representation

Previous versions of *Connext* (prior to 5.0) used TypeCodes as the wire representation to communicate types over the network and the TypeCode API to introspect and manipulate the types at run time.

The Extensible Types specification uses TypeObjects as the wire representation and the DynamicType API to introspect and manipulate the types. Types are propagated by serializing the associated TypeObject representation.

Connext 5.0 supports TypeObjects as the wire representation. To maintain backward compatibility with previous releases, *Connext* 5.0 still supports propagation of TypeCodes. However, the support for this feature may be discontinued in future releases.

Important: Connext does not support TypeObjects for types containing bitfields or for sparse types.

Connext 5.0 does not currently support the DynamicType API described in the Extensible Types specification. Therefore you must continue using the TypeCode API to introspect the types at run time.

You can introspect the discovered type independently of the wire format by using the **type_code** member in the PublicationBuiltinTopicData and SubscriptionBuiltinTopicData structures.

Important: One important limitation of using TypeCodes as the wire representation is that their serialized size is limited to 65 KB. This is a problem for services and tools that depend on the discovered types, such as *RTI Routing Service* and *RTI Spreadsheet Add-in for Microsoft Excel*. With the introduction of TypeObjects, this limitation is removed since the size of the serialized representation is not bounded.

To summarize:

	Connext 5.0	**Connext 4.5f and Earlier**
Wire Representation	TypeObjects or TypeCodes (for backwards compatibility)	TypeCodes
For Introspection at Run Time	TypeCode API (DynamicType API currently not supported)	TypeCode API
Maximum Size of Serialized Representation	When using TypeObjects: Unbounded When using TypeCodes: 65 KB	65 KB

4.1 XML and XSD Type Representations

The XML and XSD type-representation formats available in *Connext* formed the basis for the DDS-XTypes specification of these features. They support the new features introduced in *Connext* 5.0. However, they have not been completely updated to the new standard format.

5 TypeObject Resource Limits

Table 5.1 lists fields in the DomainParticipantResourceLimitsQosPolicy that control resource utilization when the TypeObjects in a *DomainParticipant* are stored and propagated.

Note that memory usage is optimized; only one instance of a TypeObject will be stored, even if multiple local or remote *DataReaders* or *DataWriters* use it.

Table 5.1 **New TypeObject Fields in DomainParticipantResourceLimitsQosPolicy**

Type	Field	Description
DDS_Long	type_object_ max_serialized_length	The maximum length, in bytes, that the buffer to serialize a TypeObject can consume. This parameter limits the size of the TypeObject that a *DomainParticipant* is able to propagate. Since TypeObjects contain all of the information of a data structure, including the strings that define the names of the members of a structure, complex data structures can result in TypeObjects larger than the default maximum of 3072 bytes. This field allows you to specify a larger value. It cannot be UNLIMITED. Default: 3072
DDS_Long	type_object_ max_deserialized_length	The maximum number of bytes that a deserialized TypeObject can consume. This parameter limits the size of the TypeObject that a *DomainParticipant* is able to store. Default: UNLIMITED
DDS_Long	deserialized_type_object_ dynamic_ allocation_threshold	A threshold, in bytes, for dynamic memory allocation for the deserialized TypeObject. Above it, the memory for a TypeObject is allocated dynamically. Below it, the memory is obtained from a pool of fixed-size buffers. The size of the buffers is equal to this threshold. Default: 4096

The TypeObject is needed for type-assignability enforcement.

By default, *Connext* will propagate both the pre-standard TypeCode and the new standard TypeObject. It is also possible to send either or none of them:

To propagate TypeObject only:	Set type_code_max_serialized_length = 0
To propagate TypeCode only:	Set type_object_max_deserialized_length = 0
To propagate none:	Set type_code_max_serialized_length = 0 and type_object_max_deserialized_length = 0
To propagate both (default):	Use the default values of type_code_max_serialized_length and type_object_max_serialized_length or modify them if the type size requires so.

6 ContentFilteredTopics

Writer-side filtering using the built-in filters (SQL and STRINGMATCH) is supported as long as the filter expression contains members that are present in both the *DataReader's* type and the *DataWriter's* type. For example, consider the following types:

DataWriter:

```
struct MyBaseType {
    long x;
};
```

DataReader:

```
struct MyDerivedType : MyBaseType {
    public long y;
```

};

If the *DataReader* creates a ContentFilteredTopic with the expression "*x*>5", the *DataWriter* will perform writer-side filtering since it knows how to find x in the outgoing samples.

If the *DataReader* creates a ContentFilteredTopic with the expression *"x>5 and y>5"* the *DataWriter* will not do writer side filtering since it does not know anything about "y". Also, when the *DataWriter* tries to compile the filter expression from the *DataReader,* it will report an error such as the following:

```
DDS_TypeCode_dereference_member_name:member starting with [y > ] not found
PRESParticipant_createContentFilteredTopicPolicy:content filter compile
error 1
```

7 Annotations

The standard syntax for applying annotations is not supported in this release. The old, pre-standard format used by RTI is still in use for the built-in annotations added in *Connext* 5.0.

For example, the following DDS-XTypes conformant structure:

```
@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct MyType {
    @Key long x;
    @Shared long y;
};
```

Can be defined using:

```
struct MyType {
    long x; //@Key
    long * y;
};
//@Extensibility EXTENSIBLE_EXTENSIBILITY
//@top-level false
```

And this:

```
enum MyEnum {
    @Value(10) CONSTANT_1,
    @Value(20) CONSTANT_2
};
```

Can be defined using:

```
enum MyEnum {
    CONSTANT_1 = 10,
    CONSTANT_2 = 20
};
```

8 RTI Spy

rtiddsspy includes limited support for Extensible Types:

- ❑ *rtiddsspy* can subscribe to topics associated with final and extensible types.
- ❑ *rtiddspy* will automatically create a *DataReader* for each version of a type discovered for a topic. In Connext 5.0, it is not possible to associate more than one type to a topic within a single *DomainParticipant*, therefore each version of a type will require its own *DomainParticipant*.

- ❏ The TypeConsistencyEnforcementQosPolicy's **kind** in each of the *DataReaders* created by *rtiddsspy* is set to DISALLOW_TYPE_COERCION. This way, a *DataReader* will only receive samples from *DataWriters* with the same type, without doing any conversion.
- ❏ The **-printSample** will print each of the samples using the type version of the original publisher.

For example:

```
struct A {
  long x;
};
struct B {
  long x;
  long y;
};
```

Let's assume that we have two *DataWriters* of Topic "T" publishing type "A" and type "B" and sending TypeObject information. After we start spy we will get an output like this:

```
RTI Data Distribution Service Spy built with NDDS version 1.6a.00--
C1.6a.00--C++1.6a.00
Copyright 2012 Real-Time Innovations, Inc.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
NddsSpy is listening for data, press CTRL+C to stop it.

source_timestamp   Info  Src HostId  topic               type
-----------------  ----  ----------  ------------------  ------------------

1345847910.453969  W +N  0A1E01C0    Example A           A
1345847912.056410  W +N  0A1E01C0    Example B           B
1345847914.454385  d +N  0A1E01C0    Example A           A

x: 1

1345847916.056787  d +N  0A1E01C0    Example B           B

x: 2
y: 3

1345847918.455104  d +M  0A1E01C0    Example A           A

x: 2

1345847920.057084  d +M  0A1E01C0    Example B           B

x: 4
y: 6
```

8.1 Type Version Discrimination

rtiddsspy uses the rules described in Section 2.3.1 to decide whether or not to create a new *DataReader* when it discovers a *DataWriter* for a topic "T".

For *DataWriters* created with previous *Connext* releases (prior to 5.0), *rtiddsspy* will select the first *DataReader* with a registered type name equal to the discovered registered type name, since *DataWriters* created with previous releases do not send TypeObject information.

9 Compatibility with Previous Connext Releases

This section describes important behavior differences between *Connext* 5.0 and previous releases. Please read this section to learn about possible incompatibility issues when communicating with pre-*Connext* 5.0 applications.

9.1 Type-Consistency Enforcement

By default, *Connext* 5.0 determines if a *DataWriter* and a *DataReader* can communicate by comparing the structure of their topic types (see Section 2.2 and Section 2.3).

In previous releases, *Connext* considered only the registered type names.

This change in default behavior may cause some applications that were communicating when using previous releases to not communicate when ported to *Connext* 5.0.

For example, let's assume that we have the following two applications:

❑ The first application creates a *DataWriter* of Topic, **Square**, with the registered type name, **ShapeType**, and the type, **EnglishShapeType**:

```
struct EnglishShapeType {
    long x;
    long y;
    long size;
    float angle;
};
```

❑ The second application creates a *DataReader* of Topic, **Square**, with the registered type name **ShapeType**, and the type **SpanishShapeType**:

```
struct SpanishShapeType {
    long x;
    long y;
    long tamagno;
    float angulo;
};
```

Before *Connext* 5.0: The *DataWriter* and *DataReader* in the above example *will* communicate. *Connext* only considers the registered type name to determine whether or not the types were consistent; therefore the *DataWriter* and *DataReader* in the above example match because they both use the same registered type name, **ShapeType**.

Starting with *Connext* 5.0: The *DataWriter* and *DataReader* in the above example will *not* communicate. The "Extensible and Dynamic Topic Types for DDS" specification does not consider **EnglishShapeType** and **SpanishShapeType** to be compatible. *Types are incompatible if they have members with same member ID*[1] *but different names.* In this case, **size** and **tamagno** have the same ID but different names (same situation for **angle** and **angulo**).

To make these two applications compatible, you can disable usage of TypeObjects, as explained in Section 5; however by doing so, the system will no longer check type-consistency at run time by looking at the structure of the topic types.

For more details on type consistency and assignability, see Section 2.2 and Section 2.3.

9.2 Trigger for Changes to INCONSISTENT_TOPIC Status

The condition under which the middleware triggers an INCONSISTENT_TOPIC status update is different starting in release 5.0.

1. For information on member IDs, see the XTypes specification (http://www.omg.org/spec/DDS-XTypes/).

Before *Connext* 5.0: The change of status occurred when a remote *Topic* inconsistent with the locally created *Topic* was discovered. This check was based only on the registered type name.

Starting with *Connext* 5.0: The change of status occurs when a *DataReader* and a *DataWriter* on the same *Topic* do not match because the type-consistency enforcement check fails.

9.3 Wire Compatibility

An endpoint (*DataWriter* or *DataReader*) created with *Connext* 5.0 will not be discovered by an application that uses a previous *Connext* release if that endpoint's TypeObject is sent on the wire *and* its size is greater than 65535 bytes.

This is because TypeObjects with a serialized size greater than 65535 bytes require extended parameterized encapsulation when they are sent as part of the endpoint discovery information. This parametrized encapsulation is not understood by previous *Connext* releases.