OpenSplice DDS

Version 5.x

DLRL Code Generator Guide

PRISMTECH

OpenSplice DDS

DLRL Code Generator Guide

Part Number: OS-DCGG

Doc Issue 14, 11 May 10

Copyright Notice

Contents

Table of Contents

Preface

About the DLRL Code Generator Guide

The *DLRL Code Generator Guide* provides a detailed explanation of the OpenSplice DLRL Code Generator (DCG).

Intended Audience

The *DLRL Code Generator Guide* is intended to be used by developers who are using OpenSplice DDS to develop applications.

Conventions

The conventions listed below are used to guide and assist the reader in understanding the DLRL Code Generator Guide.


Item of special significance or where caution needs to be taken.

i
Item contains helpful hint or special information.

WIN
Information applies to Windows (e.g. XP, 2003, Windows 7) only.

UNIX
Information applies to Unix based systems (e.g. Solaris) only.

C
C language specific

C++
C++ language specific

Java
Java language specific

Hypertext links are shown as *blue italic underlined.*

On-Line (PDF) versions of this document: Items shown as cross references, e.g. *Contacts* on page viii, are as hypertext links: click on the reference to go to the item.

```
%  Commands or input which the user enters on the
   command line of their computer terminal
```

`Courier` fonts indicate programming code and file names.

Extended code fragments are shown in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to "example" and kind field to an empty string
newName[0] = new NameComponent ("example", "");
```

Italics and ***Italic Bold*** indicate new terms, or emphasise an item.

Arial Bold indicate Graphical User Interface (GUI) elements and commands, for example, **File | Save** from a menu.

Step 1: One of several steps required to complete a task.

Contacts

PrismTech can be reached at the following contact points for information and technical support.

USA Corporate Headquarters

PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA

Tel: +1 781 569 5819

European Head Office

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK

Tel: +44 (0)191 497 9900
Fax: +44 (0)191 497 9901

Web: *http://www.prismtech.com*
Technical questions: *crc@prismtech.com* (Customer Response Center)
Sales enquiries: *sales@prismtech.com*

CHAPTER

1 Description and Use

The OpenSplice DLRL Code Generator (DCG) plays a role in generating code for DDS/DLRL specialized interfaces and classes (e.g., ObjectRoot, ObjectHome) from application data definitions defined in IDL and XML for all supported languages.

1.1 Introduction

The OpenSplice DCG accepts two IDL input files and one XML input file. The IDL input files comply with the OMG CORBA specification. One IDL file is used to describe the application classes and data types on DLRL level (i.e., Object Model), while the other (optional) IDL file is used to describe the data types on DCPS level (i.e., Topic Model). If no IDL file containing the Topic Model is specified the DCG will generate this file based upon a set of default rules as defined in Chapter 1.6, *Default Mapping Rules*, on page 20. The XML input file complies with the W3C XML specification and is used to annotate the DLRL application classes and describe the mapping from DLRL Object Model to the underlying DCPS Topic Model. The XML input file must comply with the DTD specified in Chapter 1.5.1, *XML DTD Mapping*, on page 5, this DTD is part of the OpenSplice release. *Figure 1* shows the OpenSplice DCG high-level processing.

The OpenSplice DCG scans and parses the IDL input file(s) and the XML input file. The OpenSplice DCG will, after parsing, determine if all DLRL classes are mapped to DCPS topics, if this is not the case then additional mapping information will be generated following default rules. It will then validate the input files with each other, ensuring that there are no conflicts of any kind.

If the OpenSplice DCG generated any additional mapping information, then this information will be written to output files. In this case one IDL file containing data structures will be generated and used by the OpenSplice DCG as input for the OpenSplice IDL Pre-processor. Also one XML file containing XML information of the input XML file annotated with the newly generated mapping information will be generated. This XML file is generated for convenience purposes and may be used as input in future runs of the OpenSplice DCG if desired or used to see how the DCG annotated the mapping information.

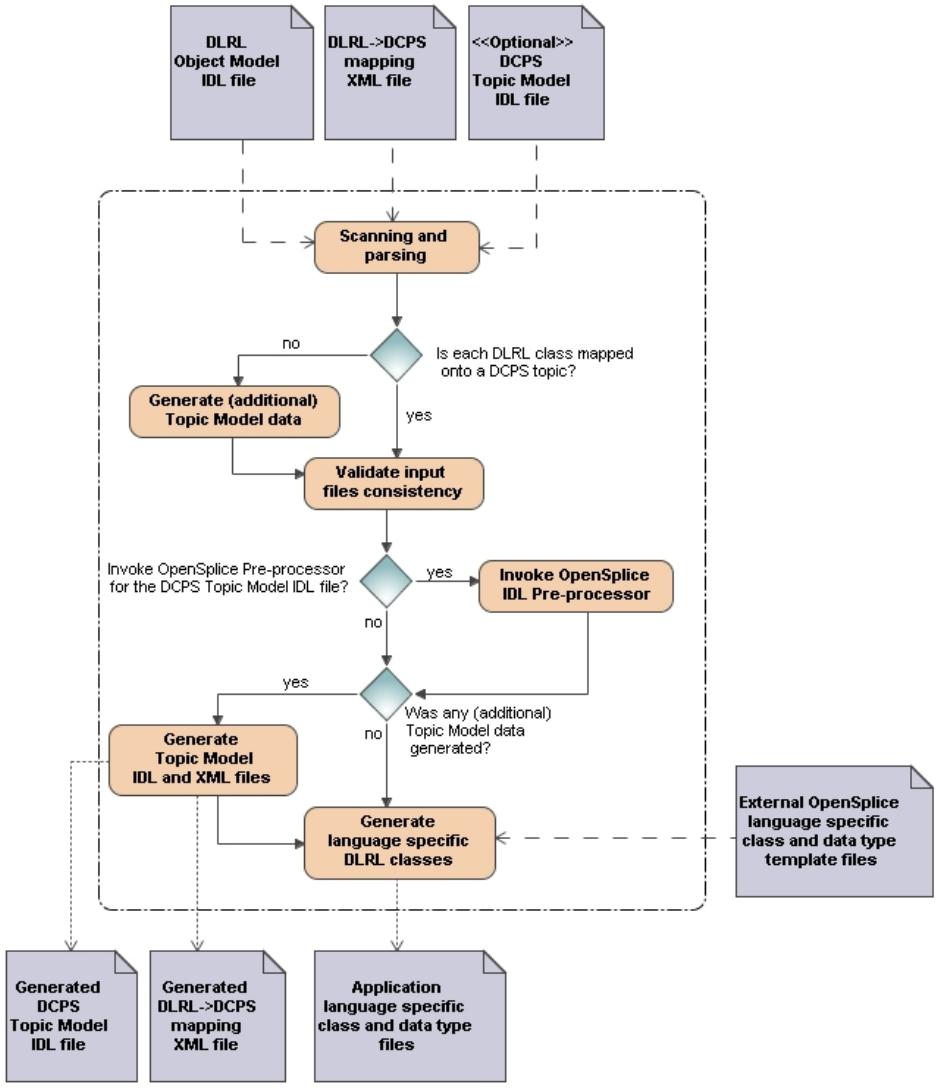

Figure 1 OpenSplice DCG High Level Processes

For the selected language, the OpenSplice DCG generates the specialized interfaces and classes based on specialized class template files which are provided by OpenSplice. Note that the OpenSplice DCG will only generate specialized interfaces and classes for application classes or data types defined in the IDL file containing the Object Model. The IDL file containing the Topic Model should be offered to the OpenSplice IDL Pre-processor as input separately unless a special command line parameter (`-gendcps`) is provided to the OpenSplice DCG, in which case the OpenSplice IDL Pre-processor will be invoked automatically. The

OpenSplice DCG also generates language specific support functions, which are needed to allow the OpenSplice system to handle the application classes and data types.

The OpenSplice DCG generates the language specific application classes and data types according the OMG IDL language mapping that is applicable for the specific target language.

1.2 Prerequisites

UNIX

The OpenSplice environment must be set correctly for UNIX-based platforms before the OpenSplice DCG can be used. Run *release.com* from a shell command line to set the environment. release.com is located in the root directory of the OpenSplice installation (*<OSPL_HOME>*):

```
%   . <OSPL_HOME>/release.com
```

The OpenSplice DCG, *ospldcg*, is started in a command shell:

```
%   ospldcg
```

The ospldcg's command line options are described in Section 1.3, *DCG Command Line Options*, below.

1.3 DCG Command Line Options

The OpenSplice DCG, ospldcg, can be run with the following command line options:

```
[ -dcps <filename> ]
< -dlrl <filename> >
< -mapping <filename> >
[ -o <path> | -o<path> ]
[ -I <path> | -I<path> ]
[ -defaultoid (FullOid | SimpleOid) ]
[ -gendcps ]
[ -v ]
[ -l (JAVA | SACPP) ]
[ -help | -h ]
```

These options are described in detail, below. Options between angle brackets (< >) are required; options between square brackets (**[ ]**) are optional. All options may be specified in random order and are case insensitive subject to the platform: case sensitivity for file names is platform dependant. Spaces in file names should be enclosed by \' or \", for example ***\'filename with spaces\'***.

[-dcps <filename>] - specifies the IDL input file containing the DCPS Topic Model definition. Even when this option is specified, the DCG may still generate a complementary topics IDL file if any additional topic information is required. In such a case the output file is generated in the output directory specified by the '*-o*' option. The OpenSplice DCG will invoke the OpenSplice IDL Pre-Processor tool automatically for this generated file. The name of this generated file is chosen equal to the name of the `Dlrl` element in the corresponding XML mapping file (that is specified by the *-mapping* option), post fixed with `.idl`.

< -dlrl <filename> > - required option specifying the IDL file containing the DLRL Object Model definition.

< -mapping <filename> > - required option specifying the XML mapping file containing mapping information between the Object Model and the Topic Model definitions. If any mapping information is missing the OpenSplice DCG will try to annotate the mapping file and output the result to the output directory specified by the *-o* option. The name of this generated file is chosen equal to the name of the `Dlrl` element in the specified XML mapping input file by this `-mapping` option, post fixed with `.xml`.

[-o <path> | -o<path>] - specifies the output directory where the OpenSplice DCG should generate it's output files. If no output path is specified then the OpenSplice DCG will generate it's output to the current file directory. If necessary the output directory will be created.

[-I <path> | -I<path>] - specifies an IDL include file path. This option may be used zero or more times. The *etc/idl* path in the *OSPL_HOME* release directory has already been set by default.

[-defaultOid (FullOid | SimpleOid)] - specifies the content value of the key descriptions that default generated topics will use. This option is only applicable to DLRL classes that have no corresponding topic definition. If not specified `SimpleOid` will be used by default. Be aware that selecting `FullOid` is less efficient due to the addition of an extra string key.

[-gendcps] - used with the ***-dcps*** option to invoke the OpenSplice IDL Pre-processor for the file specified by `-dcps`. The same language options, include paths and output directory will be used for invoking of the OpenSplice IDL Pre-processor as was used for the invoking of OpenSplice DCG. Using this option without specifying `-dcps` has no affect.

[-validate] - if specified the OpenSplice DCG will only validate the input files and will not generate any output files.

[-v] - prints additional messages to the default output stream.

[-l (JAVA | SACPP)] - specifies the language to generate output for (currently only the Java and SACPP languages are supported). Java is the default language used when this option is not specified.

[-h | -help] - display the options listed here

1.4 DCG IDL Grammar

The OpenSplice DCG accepts IDL grammar which complies with the IDL grammar as defined in section 3.4 of the CORBA 2.4.2, formal/01-02-01 specification (available from obtainable from *www.omg.org*).

1.5 DCG XML Grammar

The OpenSplice DCG accepts XML grammar which adheres to the DTD for the DLRL XML mapping file as defined in Chapter 1.5.1, *XML DTD Mapping*, on page 5. This DTD file is included in the OpenSplice release and can be found in the *etc/dcg* directory located in the root directory of the OpenSplice installation (*<OSPL_HOME>*). The DTD file is called 'Dlrl.dtd'.

Each XML mapping file must include the following directives:

```
<?xml version='1.0' encoding='ISO-8859-1' standalone='no'?>
<!DOCTYPE Dlrl SYSTEM 'Dlrl.dtd'>
```

i Note that it is not required to define the path of the DTD file in the XML mapping file since, the OpenSplice DCG will automatically locate the ***Dlrl.dtd*** file which is located in the installation path.

1.5.1 XML DTD Mapping

The following XML code is the DTD for expressing the DLRL to DCPS mapping.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT Dlrl
       (enumDef | templateDef | associationDef | compoRelationDef |
       classMapping)*>
<!ATTLIST Dlrl
        name            CDATA           #IMPLIED>

<!ELEMENT enumDef
        (value)*>
<!ATTLIST enumDef
        name            CDATA           #REQUIRED>

<!ELEMENT value
        (#PCDATA)>

<!ELEMENT templateDef EMPTY>
<!ATTLIST templateDef
        name            CDATA           #REQUIRED
        pattern         (Set | StrMap | IntMap | List) #REQUIRED
        itemType        CDATA           #REQUIRED>

<!ELEMENT associationDef
```

```
        (relation,relation)>

<!ELEMENT relation EMPTY>
<!ATTLIST relation
        class           CDATA           #REQUIRED
        attribute       CDATA           #REQUIRED>

<!ELEMENT compoRelationDef EMPTY>
<!ATTLIST compoRelationDef
        class           CDATA           #REQUIRED
        attribute       CDATA           #REQUIRED>

<!ELEMENT classMapping
       (mainTopic?,extensionTopic?, (monoAttribute | multiAttribute |
       monoRelation | multiRelation | local)*)>
<!ATTLIST classMapping
        name            CDATA           #REQUIRED
        implClass       CDATA           #IMPLIED
        implPath        CDATA           #IMPLIED>

<!ELEMENT mainTopic
        (keyDescription)>
<!ATTLIST mainTopic
        name            CDATA           #REQUIRED
        typename        CDATA           #IMPLIED>

<!ELEMENT extensionTopic
        (keyDescription)>
<!ATTLIST extensionTopic
        name            CDATA           #REQUIRED
        typename        CDATA           #IMPLIED>

<!ELEMENT monoAttribute
        (placeTopic?,valueField+)>
<!ATTLIST monoAttribute
        name            CDATA           #REQUIRED>

<!ELEMENT multiAttribute
        (multiPlaceTopic,valueField+)>
<!ATTLIST multiAttribute
        name            CDATA           #REQUIRED>

<!ELEMENT monoRelation
        (placeTopic?, validityField?, keyDescription)>
<!ATTLIST monoRelation
        name            CDATA           #REQUIRED>

<!ELEMENT validityField EMPTY>
<!ATTLIST validityField
   name   CDATA  #REQUIRED>

<!ELEMENT multiRelation
        (multiPlaceTopic,keyDescription)>
<!ATTLIST multiRelation
        name            CDATA           #REQUIRED>

<!ELEMENT local EMPTY>
<!ATTLIST local
        name            CDATA           #REQUIRED>

<!ELEMENT placeTopic
```

```
        (keyDescription)>
<!ATTLIST placeTopic
        name            CDATA           #REQUIRED
        typename        CDATA           #IMPLIED>

<!ELEMENT multiPlaceTopic
        (keyDescription)>
<!ATTLIST multiPlaceTopic
        name            CDATA           #REQUIRED
        typename        CDATA           #IMPLIED
        indexField      CDATA           #IMPLIED>

<!ELEMENT keyDescription
        (keyField*)>
<!ATTLIST keyDescription
        content (FullOid | SimpleOid | NoOid) #REQUIRED>

<!ELEMENT keyField
        (#PCDATA)>

<!ELEMENT valueField
        (#PCDATA)>
```

1.5.2 XML Element Details

The fully qualified name is always used to ensure no conflicts when two constructs with the same name exist in different modules when identifying IDL constructs in XML. A fully qualified name uses IDL scoping separators (**::**). For example, a `valuetype` ***Foo*** in module ***Test*** defined in IDL will be referenced as ***Test::Foo*** in XML.

The elements which are used in the DTD for expressing the DLRL to DCPS mapping are described in detail below.

1.5.2.1 Dlrl

This is the main XML Mapping file element and must always be included.

Mandatory Attributes

name - This attribute may contain any name. It will be used by the OpenSplice DCG as the base file name when (additional) mapping information is generated. It will be postfixed with ***.idl*** for the Topics Model IDL file and with ***.xml*** for the annotated mapping XML file which is generated in this case.

Optional Attributes

None

Child Elements

enumDef - This element is not supported by the OpenSplice DCG.

templateDef - Defines a typed collection (giving its pattern as well as the type of its elements); it comes in place of a statement such as `List<Foo>` which is not allowed in IDL.

compoRelationDef - States that a given relation is actually a composition.

associationDef - Associates two relations, so that they make a full association (in the UML sense).

classMapping - Defines the mapping of a DLRL class to DCPS topic(s).

Example

```
<Dlrl name="dlrl_example">
  ...
</Dlrl>
```

1.5.2.2 enumDef

This element is currently not supported by OpenSplice DCG.

1.5.2.3 value

This element is currently not supported by OpenSplice DCG.

1.5.2.4 templateDef

The DLRL supports various collection types such as a Set, List or Map (with a string or integer key). However IDL is too limited to describe such constructs. If we wanted to define a map with a 'string' key of type 'Foo' we would want to include the following (hypothetical) typedef:

```
typedef StrMap<Foo> FooStrMap;
```

As such constructs are not allowed, another way has to be found. The DDS specification states that such collections should be realized by using a 'forward valuetype' declaration in IDL and annotating that declaration in XML. This is the purpose of the templateDef element.

Required Attributes

name - the fully qualified IDL name of the collection type.

pattern - the construct pattern of the Collection. The constructs are *`List`*, *`StrMap`*, *`IntMap`* and *`Set`*. The `List` pattern is currently ignored for generation by the OpenSplice DCG.

itemType - the fully qualified IDL name of each element type in the collection

Optional Attributes

None

Child Elements

None

Example

```
<templateDef name="BarModule::BarStrMap"
             pattern="StrMap"
             itemType="BarModule::Bar"/>
```

1.5.2.5 associationDef

This element is used to associate two relations with each other. Each relation then represents an association end of the newly formed association.

This element is currently ignored for generation by the OpenSplice DCG.

Mandatory Attributes

None

Optional Attributes

None

Child Elements

relation - The `associationDef` element embeds two required relation elements to designate the concerned relations.

Example

```
<associationDef>
  ...
</associationDef>
```

1.5.2.6 relation

This element is used to describe a relation within a DLRL class. This element is only used as a child element of the associationDef element. This element is currently ignored for generation by the OpenSplice DCG.

Mandatory Attributes

class - contains the fully qualified IDL name of the DLRL class.

attribute - contains the name of the attribute that supports the relation inside the DLRL class.

Optional Attributes

None

Child Elements

None

Example Using relation with associationDef

```
<associationDef>
  <relation class="TrackModule::Track"
            attribute="a_radar"/>
  <relation class="RadarModule::Radar"
            attribute="tracks"/>
</associationDef>
```

1.5.2.7 compoRelationDef

This element is used to indicate that a relation is actually a composition. This element is currently ignored for generation by the OpenSplice DCG.

Mandatory Attributes

class - contains the fully qualified scope IDL name of the DLRL class.

attribute - contains the name of the attribute that supports the relation inside the DLRL class.

Optional Attributes

None

Child Elements

None

Example

```
<compoRelationDef class="RadarModule::Radar"
                  attribute="tracks"/>
```

1.5.2.8 classMapping

This element is the root element for any mapping of a single DLRL class to DCPS topic(s).

Mandatory Attributes

name - This attribute contains the fully qualified IDL name of the DLRL class to be mapped.

Optional Attributes

implClass - This attribute contains the fully qualified IDL name for the application-specific class containing the implementation code for any user-defined operations on the DLRL class, as specified for this `classMapping`.

implPath - This attribute is only used in the `SACPP` language binding and contains the `include` filepath for the file containing the class definition for the implementation class, as specified in the `implClass` attribute, for example, *`include/implClass.h`*. The same value may be used for multiple `classMapping` attributes: the file will only be included once. This attribute has no meaning when not using the `SACPP` language binding.

Child Elements

mainTopic - an optional child element that identifies the main topic onto which this DLRL class is mapped. If this element is specified then the entire DLRL class must be mapped, if it is not specified then the OpenSplice DCG will automatically complete the `classMapping` following default rules, but not overriding any custom made mappings within the scope of this `classMapping`. Learn more about types of mapping in Chapter 1.8, *Mapping Modes*, on page 26.

extensionTopic - an optional child element that identifies the so-called extension topic of this DLRL class. Extension topics are used when a DLRL class extends another DLRL class (excluding `DDS::ObjectRoot`).

monoAttribute - maps attributes (excluding collection types and relations to DLRL objects) of the DLRL class onto the correct DCPS topic attribute.

multiAttribute - maps collection type attributes (excluding collections of DLRL classes) of the DLRL class onto a DCPS topic.

monoRelation - maps attributes of the DLRL class which are relations towards other DLRL classes onto the correct DCPS topic attribute(s).

multiRelation - maps collection type attributes with DLRL class elements of the DLRL class onto a DCPS topic.

local - specifies that an attribute will not participate in data distribution and only has meaning in the local application context.

Example

```
<classMapping name="TrackModule::Track">
  ...
</classMapping>
```

1.5.2.9 mainTopic

This element identifies the main DCPS topic of a DLRL class. The main topic is the topic which defines the existence of a DLRL object. A DLRL object is declared as existing if, and only if, there is an instance in that topic matching it's key value.

Mandatory Attributes

name - contains the name of the topic. A topic name is an identifier for a topic and is defined as any series of characters *'a', ..., 'z', 'A', ..., 'Z', '0', ..., 9', '_'*, but may not start with a digit.

Optional Attributes

typename - contains the fully qualified IDL name of the type of the topic, if not specified then the type name is considered to be equal to the topic name. This can thus only be true for topic types which are not contained within a module, as the topic name will never contain the **::** scoping operator.

Child Elements

keyDescription - required child element specifying the *mainTopic*'s keys

Example

```
<mainTopic name="Track_Topic"
           typename="TrackModuleDCPS::Track_Topic">
  <keyDescription ...>
     ...
  </keyDescription>
</mainTopic>
```

1.5.2.10 keyDescription

This element describes the keys to be associated to several elements (mainTopic, extensionTopic, placeTopic and multiPlaceTopic).

Mandatory Attributes

content - describes the `keyDescription's` contents. The values can be *FullOid*, *SimpleOid* or *NoOid*. The content of a `keyDescription` puts restrictions on the number of *keyField* elements contained within the `keyDescription` and on the sequence of those `keyField` elements.

If `FullOid` is selected, then it is implied that there will be two `keyField` child elements. The first `keyField` is used to store the name of the topic field and which indicates the type name of the topic. The second `keyField` is used to store the name of the Object ID (`OID`) field. `FullOid` is meant to be used for Object Models where inheritance plays a role, the topic type name stored in the first field can then be used to deduct the exact type of any given instance. OpenSplice does support `FullOid` but does not require it to resolve inheritance yet.

If `SimpleOid` is selected, then it implies that there will be only one `keyField` child element. This `keyField` is used to store the name of the `OID` field within the DCPS topic.

If `NoOid` is selected it implies that there is no fixed amount of `keyField` elements and no fixed sequence. This option is used mostly in systems where a Topic Model already exists and topics do not contain a field which matches the `OID` definition of DLRL. Take note that when relating classes in DLRL with each other and `NoOid` is used that the `keyField` elements in the `keyDescription` elements should be in the same sequence! If no `keyField` elements are defined it means that at most one instance of this type can exist.

Optional Attributes

None

Child Elements

`keyField` - identifies a field within the containing DCPS topic which is used as a key field. The number of `keyField` elements is dependent on the value of the `content` attribute.

Example Different content types

```
<keyDescription content="FullOid">
  <keyfield>typeNameField</keyField>
  <keyField>someOidField</keyField>
</keyDescription>

<keyDescription content="SimpleOid">
  <keyField>someOidField</keyField>
</keyDescription>

<keyDescription content="NoOid">
  <keyField>someKeyField1</keyField>
  <keyField>someKeyField2</keyField>
  <keyField>someKeyField3</keyField>
</keyDescription>
```

1.5.2.11 keyField

This element is used within `keyDescription` elements to identify a field within a topic.

Mandatory Attributes

None

Optional Attributes

None

Child Elements

`text` - required element which defines a field's name

Example

```
<keyfield>typeNameField</keyField>
```

1.5.2.12 extensionTopic

The *`extensionTopic`* is used to identify the DCPS topic that is used as an extension topic for the attributes of an inherited class. It comprises the same attributes as the mainTopic element as described in Chapter 1.5.2.9, *mainTopic*, on page 12.

The `extensionTopic` element is currently not supported by OpenSplice DCG.

1.5.2.13 placeTopic

This element comprises the same attributes as the `mainTopic` element as described in Chapter 1.5.2.9, *mainTopic*, on page 12. It is used to allow the definition of attributes on external topics, which is useful in systems where some attributes of a DLRL object are updated more frequently then the rest of the DLRL object.

This element is currently no supported by OpenSplice DCG.

1.5.2.14 monoAttribute

This element defines the mapping of an attribute of a DLRL class onto a DCPS topic. Collection types and relations towards DLRL classes are not covered by this element.

Mandatory Attributes

name - defines the name of the attribute within the DLRL class.

Optional Attributes

None

Child Elements

placeTopic - This optional child element defines the DCPS topic where the attribute is located. This child element follows the same pattern as the `mainTopic` element. If no `placeTopic` is defined then the `extensionTopic`, or if also not defined, the `mainTopic` element is used in place of the `placeTopic`.

valueField - This required child element defines the name of the field within the topic that will contain the value of the attribute. OpenSplice DCG does not yet support defining multiple `valueField` child elements.

Example

```
<monoAttribute name="y">
  <placeTopic name="Y_TOPIC"
              typename="SomeModule::Y_TOPIC">
    <keyDescription content="SimpleOid">
       <keyField>oidField</keyField>
    </keyDescription>
  </placeTopic>
  <valueField>Y</valueField>
</monoAttribute>
```

1.5.2.15 valueField

This element is used within various elements to identify a field within a topic.

Mandatory Attributes

None

- Optional attributes:

  None

- Child elements:

text element - The required text element is used to define the name for a field.

Example

```
<valueField>Y</valueField>
```

1.5.2.16 multiAttribute

This element is currently ignored for generation by the OpenSplice DCG.

1.5.2.17 monoRelation

This element defines the mapping of an attribute of a DLRL class onto a DCPS topic. Only attributes which are relations towards other DLRL classes are covered by this element, excluding collection types.

Mandatory Attributes

name - defines the name of the attribute within the DLRL class

Optional Attributes

None

Child Elements

placeTopic - This optional child element defines the DCPS topic where the attribute is - located. This child element follows the same pattern as the mainTopic element. If no placeTopic is defined then the extensionTopic, or if also not defined, the mainTopic element is used in place of the placeTopic.

validityField element - This optional child element may only be used if the `keyDescription` element in this `monoRelation` has a content type of `NoOid`. It defines the name of the field in the DCPS topic which indicates if the value in the field(s) defining this relation should be interpreted or not (i.e., whether those values are valid or not). If the value of the `validityField` field in the topic is a zero value it indicates the DLRL should not interpret the key fields for this relation. If the value is anything else then zero it indicates the DLRL should interpret the key fields.

The `validityField` allows `monoRelation` elements to be defined with a *`0...1`* cardinality instead of the default *`1`* cardinality. For relations mapped as `SimpleOid` or `FullOid` the `validityField` is not needed as such relations have a `0...1` cardinality by default.

keyDescription - required child element which defines the names of the fields that contain the value of the relation.

Example

```
<monoRelation name="a_radar">
  <keyDescription content="SimpleOid">
    <keyField>radarOidField</keyField>
  </keyDescription>
</monoRelation>
```

1.5.2.18 validityField

This element is used to indicate the validityField within a DCPS topic. A validityField is only valid in monoRelation elements with a keyDescription with a content of NoOid. A validityField may be of the following 'countable' IDL types: boolean, long, long long, octet, short, unsigned long, unsigned long long, unsigned short. If the value of this field in an instance is found to be '0' then the relation for which this validityField was defined will not be interpreted. If the field is any other value then the relation will be interpreted.

Mandatory Attributes

name - defines the name of the field within the DCPS topic

Optional Attributes

None

Child Elements

None

Example NoOid keyDescription with specified validityField

```
<monoRelation name="a_radar">
  <validityField name="isRadarValid"/>
  <keyDescription content="NoOid">
    <keyField>someKeyfieldIdentifyingRadar</keyField>
  </keyDescription>
</monoRelation>
```

1.5.2.19 MultiRelation

This element is used to define the mapping for a collection type with a DLRL class as it's element type. Each collection type is mapped onto a separate DCPS topic, where the fields in the topic identify the owner of the collection, the target element and, if applicable, the index field of the element in the collection. This means that every instance of such a topic is an element in a collection, all instances with the same owner key are thus implicitly grouped together into one collection by the DLRL.

Mandatory Attributes

name - defines the name of the attribute within the DLRL class

Optional Attributes

None

Child Elements

multiPlacetopic - required child element defining the topic which the *multiRelation* is mapped to. It contains a *keyDescription* describing the owner class' keys of the collection.

keyDescription - required child element defined in the target class' keys of the collection.

Example

```
<multiRelation name="tracks">
  <multiPlaceTopic ...>
    ...
  </multiPlaceTopic>
  <keyDescription content="SimpleOid">
    <keyField>radarElementOid</keyField>
  </keyDescription>
</multiRelation>
```

1.5.2.20 multiPlaceTopic

This element is used to define topics of collection types. It follows the same definition as the placeTopic/mainTopic/extensionTopic elements with one addition. An optional attribute may be specified to indicate which field in the topic is the index

field in the collection. Only for a set collection should this field not be specified. For the IntMap and List collections this field should be of a 'long' IDL type, for a StrMap collection it should be of a 'string' IDL type.

Mandatory Attributes

name - contains the name of the topic. A topic name is an identifier for a topic and is defined as any series of characters *'a', ..., 'z', 'A', ..., 'Z', '0', ..., 9', '_'*, but may not start with a digit.

Optional Attributes

typename - contains the fully qualified IDL name of the type of the topic, if not specified then the type name is considered to be equal to the topic name. This can thus only be true for topic types which are not contained within a module, as the topic name will never contain the ***::*** scoping operator.

indexField - optional attribute which specifies the field needed for storing the collection index, if applicable.

Child Elements

keyDescription - required child element specifying the *mainTopic*'s keys

Example

```
<multiPlaceTopic name="radar_tracks_topic"
                 typename="SomeModule::Radar_tracks_topic"
                 indexField="index">
  <keyDescription ...>
    ...
  </keyDescription>
</multiPlacetopic>
```

1.5.2.21 Local

This element is used to indicate that the corresponding attribute has to be ignored by the DLRL.

Mandatory Attributes

name - defines the name of the attribute to be ignored

Optional Attributes

None

Child Elements

None

Example

```
<local name="w"/>
```

1.6 Default Mapping Rules

The following mapping rules will be applied by default. This default mapping is overridden by any mapping information provided by the application developer.

The OpenSplice DCG will map each `valuetype` which directly or indirectly extends the *`DDS::ObjectRoot`* `valuetype` onto a `classMapping` element. The `classMapping` element will follow the following default mapping, unless a `classMapping` element already exists and a `mainTopic` element is defined for that `classMapping` in which case no default rules will be applied at all.

Each `valuetype` will be mapped onto a single topic: all single-valued attributes of the class will be located in this topic. The topic's name is the fully qualified IDL name of the DLRL `valuetype` appended with *`_topic`*. All double colon (***`::`***) IDL scoping separators are replaced with underscores (_). This topic has one key field with the name ***`oid`*** and type ***`DDS:DLRLOid`***. The topic is mapped similar to `SimpleOid`.

A simple example will illustrate how this will work. Imagine the following IDL defining DLRL valuetypes as input:

```
module Example{
  valuetype Foo : DDS::ObjectRoot {
    ...
  };
};
```

This will result in the following XML mapping information to be generated:

```
<classMapping name="Example::Foo">
  <mainTopic name="Example_Foo_topic"
             typename="Example::Foo_topic">
    <keyDescription content="SimpleOid">
      <keyField>oid</keyField>
    </keyDescription>
  </mainTopic>
  ...
</classMapping>
```

It will also result in the following IDL information for the OpenSplice PreProcessor to be generated:

```
//relevant IDL file includes will be generated here.
module Example{
  struct Foo_topic{
    DDS::DLRLOid oid;
    ...
  };
#pragma keylist Foo_topic oid
};
```

Each single-valued attribute of the valuetype will be mapped onto the before mentioned topic. The name of the field within the topic will be the same as the attribute's name within the DLRL valuetype. In case of a relation however, multiple fields may be generated, as many as is needed to uniquely identify the target valuetype. The `mainTopic`'s `keyDescription` of the target `valuetype classMapping` element plays a role in how the fields are named. The following naming scheme is used which depends on the value of the content attribute of the `keyDescription` element:

- ***FullOid***
  - the first `keyField` is *`_typeName`* for the DLRL attribute's name
  - the second `keyField` is *`_oid`* for the DLRL attribute's name

DLRL attribute name where the keyField is ''

- ***SimpleOid***
  - the keyField is ***_oid*** for the DLRL attribute's name
- ***NoOid***
  - DLRL attribute name is concatenated with the name of each `keyField` of the target valuetype's `keyDescription`

An example of the naming convention is shown below in the DLRL valuetype's IDL definition of input and where *`Bar`* is also mapped while adhering to the default rules:

Example module

```
  valuetype Bar : DDS::ObjectRoot {
    ...
  };

  valuetype Foo : DDS::ObjectRoot {
    public Bar myBar;
    public long myLong;
    ...
  };
};
```

This results in the generated XML mapping information

```
<classMapping name="Example::Bar">
  ...
</classMapping>

<classMapping name="Example::Foo">
  ...
  <monoAttribute name="myLong">
    <valueField>myLong</valueField>
  </monoAttribute>
  <monoRelation name="myBar">
    <keyDescription content="SimpleOid">
      <keyField>myBar_oid</keyField>
    </keyDescription>
  </monoRelation>
</classMapping>
```

It will also result in the generated OpenSplice PreProcessor's IDL information:

```
//relevant IDL file includes will be generated here.
module Example{
  struct Foo_topic{
    ...
    DDS::DLRLOid myBar_oid;
    long myLong;
    ...
  };
...
};
```

Each of the class' multi-valued attributes will be mapped onto a separate topic. The name of the topic is the fully qualified IDL name of the DLRL *valuetype* concatenated with the attribute name, an underscore (_) and postfixed with *_topic*. All double colon (**::**) IDL scoping separators are replaced with underscores (_).

If the multi-valued attribute is a *List*, *StrMap* or *IntMap* type, then an *index* field will be generated as the key field and called *index*. This field is assigned IDL type *long* if the multi-valued attribute is a List or IntMap and *string* if it's a StrMap. No such attribute is generated for multi-valued attributes that are of the Set type.

Each topic that is generated for a multi-valued attribute contains keys to identify it's *owner* `valuetype` and foreign keys to identify the *target* `valuetype` for each element. The names for each of these fields are constructed by concatenating *`target_`* or `owner_` with the name of corresponding field in the `mainTopic` `keyDescription` definition of the corresponding valuetype.

Multi-valued attributes that are a collection of primitive types have a `valueField` element instead of a `keyDescription` element. These attributes use a similar naming convention as for topics, except they use *`element_`*' prefixed to the `valueField` name instead of *`target_`*.

The following code example illustrates the mapping. The following IDL defines DLRL `valuetypes` as input, where there are two forward valuetype declarations for a `Set` containing `Bar valuetypes` (*`BarSet`*) and a forward valuetype declaration for a `StrMap` (which also contains `Bar valuetypes` (*`BarStrMap`*). Although the `Bar valuetype` itself it not shown here, it is mapped the similarly as shown in the previous example.

```
module Example{
  valuetype BarSet;
  valuetype BarStrMap;

  valuetype Foo : DDS::ObjectRoot {
    ...
    public BarSet myBars;
    public BarStrMap myOtherBars;
    ...
  };
};
```

This will result in the following XML mapping information to be generated:

```
<classMapping name="Example::Foo">
  ...
  <multiRelation name="myBars">
    <multiPlaceTopic name="Example_Foo_myBars_topic"
                     typename="Example::Foo_myBars_topic">
      <keyDescription content="SimpleOid">
        <keyField>owner_oid</keyField>
      </keyDescription>
    </multiPlaceTopic>
    <keyDescription content="SimpleOid">
      <keyField>target_oid</keyField>
    </keyDescription>
  </multiRelation>

  <multiRelation name="myOtherBars">
    <multiPlaceTopic name="Example_Foo_myOtherBars_topic"
                     typename="Example::Foo_myOtherBars_topic"
                     indexField="index">
      <keyDescription content="SimpleOid">
        <keyField>owner_oid</keyField>
      </keyDescription>
    </multiPlaceTopic>
    <keyDescription content="SimpleOid">
```

```
      <keyField>target_oid</keyField>
    </keyDescription>
  </multiRelation>
</classMapping>
```

It will also result in the following IDL information for the OpenSplice PreProcessor to be generated. Take note that the keylist for each topic is different. For a collection topic of a Set the target fields are also included as key fields for the topic, but for other collection types only the index is also included as a key field (next to the owner fields as keyfields in every case):

```
//relevant IDL file includes will be generated here.
module Example{
  struct Foo_topic{
    ...
  };
...
  struct Foo_myBars_topic{
    DDS:DLRLOid owner_oid;
    DDS:DLRLOid target_oid;
  };
#pragma keylist Foo_myBars_topic owner_oid target_oid

  struct Foo_myOtherBars_topic{
    DDS:DLRLOid owner_oid;
    string index;
    DDS:DLRLOid target_oid;
  };
#pragma keylist Foo_myOtherBars_topic owner_oid index
};
```

1.7 Languages and Processing Steps

1.7.1 Stand Alone Java

The OpenSplice DCG generates the application classes from IDL, in accordance with the *DDS Specification* and mapping, and annotates the application with the information from the XML file. The OpenSplice DCG also generates classes for the specialized interfaces and classes.

All generated and application Java code must be compiled together with the Java compiler. All generated (DLRL related) code contains full *JavaDoc* documentation which can be generated into HTML code using the standard JDK `javadoc` operations. The OpenSplice DCG translates each module it encounters into a nested directory structure. The OpenSplice DCG generates the Java classes listed below for each `valuetype` extending *`DDS::ObjectRoot`* defined in the DLRL Object Model IDL file (for example, `valuetype Foo : DDS::ObjectRoot`).

None of the generated files should be edited since they are overwritten, without warning, whenever the OpenSplice DCG is run.

- *`ObjectRoot`* (`Foo.java`)
- *`ObjectImpl`* (`FooImpl.java`)

- *ObjectHome* (FooHome.java)
- *ObjectListener* (FooListener.java)
- *IntMap* (FooIntMap.java)
- *StrMap* (FooStrMap.java)
- *List* (FooList.java)
- *Set* (FooSet.java)
- *Selection* (FooSelection.java)
- *FilterCriterion* (FooFilter.java)
- *SelectionListener* (FooSelectionListener.java)

OpenSplice DCG generates the following files for enumerations, structs or unions (for example *SomeEnum*, *SomeStruct* or *SomeUnion* respectively) defined in the DLRL Object Model IDL file:

- *enum* (SomeEnum.java)
- *struct* (SomeStruct.java)
- *union* (SomeUnion.java)

1.7.2 Stand Alone C++

The OpenSplice DCG generates the application classes from IDL, in accordance with DDS Specification and the language mapping, and annotates the application with the information in the XML file. The OpenSplice DCG also generates classes for the specialized interfaces and classes.

All generated and application C++ code must be compiled with a C++ compiler, such as ***g++***. The OpenSplice DCG generates the following C++ files for the DLRL Object Model IDL file (for example *Foo.idl*):

ccpp_Foo.h - Contains references to all other relevant generated files and is the only one an application should include. This file should not be edited.

Foo.h - Contains the public definition of all types listed in the DLRL Object Model IDL file. This file should ***not*** be edited.

Foo.cpp - Contains the implementation for a subset of the types defined in the *Foo.h* file. This file should ***not*** be edited.

ccpp_Foo_abstract.h - Contains the definitions for all valuetype classes extended with OpenSplice specific information. Each valuetype class listed in this file directly extends the valuetype class listed in the *Foo.h* file. This file should ***not*** be edited.

ccpp_Foo_abstract.cpp - Contains the implementation for all classes and operations listed in the *ccpp_Foo_abstract.h* file. Some of the classes' functions that were not yet implemented in the *Foo.cpp* file are also implemented here. This file should ***not*** be edited.

ccpp_Foo_impl.h - Contains the lowest level implementation classes for each valuetype defined in the Object Model IDL file. Each DLRL object will be instantiated as the corresponding class defined in this file. This file should ***not*** be edited.

ccpp_Foo_impl.cpp - Contains the implementation for the types defined in the *ccpp_Foo_impl.h* file. This file should ***not*** be edited.

FooDlrl.h - Contains all helper classes which are derived of a `valuetype` defined in the Object Model IDL file. For example, if a `valuetype` *Bar* is defined in the Object Model IDL file, then classes such as *FooSelection*, *FooHome*, *FooIntMap*, etc. will be generated in this file. All types defined here comply with the OMG DDS Specification and do not contain any OpenSplice specific code. This file should ***not*** be edited.

FooDlrl.cpp - Contains the implementation for a subset of the types defined in the *FooDlrl.h* file. This file should ***not*** be edited.

ccpp_FooDlrl_impl.h - Contains the same classes that are in the *FooDlrl.h* file and annotated with the OpenSplice specific code. This file should ***not*** be edited.

ccpp_FooDlrl_impl.cpp - Contains the implementation for the classes listed in the *ccpp_FooDlrl_impl.h* file, as well as callbacks which used by the OpenSplice DLRL kernel. This file should ***not*** be edited.

1.8 Mapping Modes

The OpenSplice DCG supports three distinct modes to map DLRL objects to a DCPS topic model. These modes are detected transparently.

1.8.1 Object Model Leading Mode

In this mode only an Object Model is available and no Topic Model exists. The OpenSplice DCG will automatically generate a Topic Model based upon the Object Model following the default mapping rules.

Only two files are required as input in this mode, the IDL file containing the Object Model and the mapping XML file. This mapping XML file will only contain certain annotated Object Model information which can not be specified in IDL, if applicable. In such cases the mapping XML file may be restricted to the following code:

```
<?xml version='1.0' encoding='ISO-8859-1' standalone='no'?>
<!DOCTYPE Dlrl SYSTEM 'Dlrl.dtd'>
<Dlrl name='Minimum_mapping_xml_example'>
</Dlrl>
```

1.8.2 Topic Model Leading Mode

In this mode an Object Model is mapped onto a pre-existing Topic Model, each DLRL class is mapped onto a DCPS topic. All three input files are required in this mode and the DCG will not annotate any mapping information.

1.8.3 Hybrid Mode

This mode is a combination of the before mentioned modes and may feature two or three input files. In this mode some classes may be mapped following the Topic Model leading mode where a full mapping is done from the DLRL class towards the DCPS topics, where these DCPS topics must be defined in the DCPS Topic Model IDL file naturally. Other classes may be mapped following the Object Model leading mode, meaning that the DLRL will automatically generate mapping information for such classes following default rules.

It is also allowed to include incomplete mappings of DLRL classes, meaning that any DLRL class which contains mapping information in the mapping XML file, but without the link towards the DCPS topic will automatically be completed following default rules. This option is very powerful in terms that it allows very localized deviation of the default mapping rules. While there is no direct link from the DLRL class towards the DCPS topic (in other words, no `mainTopic` element is defined on a `classMapping` element in the Mapping XML file) the DLRL will always complete the missing mapping information as it would in Object Model leading mode and validate (not override) the already existing mapping information as it would in Topic Model leading mode. Although the OpenSplice DCG will complete such mapping, in the DTD as specified in Section 1.5.1, *XML DTD Mapping*, on page 5 is still applies: no XML may be used which does not comply to this DTD.

1.9 Adding Local Operations to DLRL valuetypes

It is possible to declare user-defined local operations on DLRL classes when they are specified in IDL. These user-defined operations will be generated on the interface definition of the class, however the application programmer must provide the actual implementation code since it is not generated automatically. The OpenSplice DCG cannot know what the purpose of a user-defined operation is,

therefore it cannot generate the operation's implementation code. Instead, it generates *dummy* implementations for these operations to ensure that the code generated by the OpenSplice DCG can compile "out of the box".

A mechanism is provided to tell the DCG where the application programmer will locate the implementation code, since all files generated by the DCG are overwritten every time the DCG is run. Accordingly, generated files must not be manually edited. Therefore the middleware must know the name and location of each class that contains the user-defined operations' implementation code, since it is responsible for instantiating the DLRL objects.

OpenSplice DCG is able to generate the implementation code for all other, non user-defined operations. However, when local operations are required, then the tasks described below must be performed.

Consider the an IDL specification for a basic DLRL object model (stored in a file called ***FooObjects.idl***):

```
//file name = 'FooObjects.idl'
#include "dds_dlrl.idl"

module test
{
    valuetype Foo : DDS::ObjectRoot
    {
        /* some shared attribute */
        public int id;
        /* A local operation defined for this valuetype */
        void display();
    };
};
```

This object model will be mapped according to the default mapping rules, namely that only minimal XML mapping directives are needed and no existing topic model will be used. However, the OpenSplice DCG must be made aware of the application-specific class that implements the local `display()` operation (as defined in the `Foo` valuetype). This is achieved, in the XML definition, by using the attributes described for the *`classMapping`* element (see Section 1.5.2.8, *classMapping*, on page 10. The XML definition for this example is:

```
<?xml version="1.0" encoding='ISO-8859-1' standalone='no'?>
<!DOCTYPE Dlrl SYSTEM 'Dlrl.dtd'>

<Dlrl name='Local_operations_example_xml'>
    <classMapping name="test::Foo"
        implClass="example::FooCustomImpl"
        implPath="FooCustomImpl.h"/>
</Dlrl>
```

The above XML definition states that for valuetype *Foo* in module *test*, a specific implementation class must be used. Further, for C++ applications the name of the header file that contains the class definition must also be specified to enable it to be generated as an *include* statement in the OpenSplice DCG's output files. (For Java, this is not necessary since the filename and its location always corresponds to the class name and its package.)

The Java implementation of this example must be defined in a separate class, which in this case is called *FooCustomImpl*, located in a package called *example*. The class definition would be:

```
package example;

public class FooCustomImpl extends test.Foo
{
    public void display()
    {
    /* insert implementation code here */
    }
}
```

The code fragment above shows that the *FooCustomImpl* class inherits from the *test.Foo* base class. The user-defined *display()* operation must be implemented locally: its signature can be copied from the dummy implementation generated by the OpenSplice DCG. The pathname for this example's dummy implementation's is (*...)/test/FooImpl.java*, but since Java derives a file name directly from the corresponding class name, this file name does not need to be specified explicitly: just ensure that the location of the *test* package is included in the *CLASSPATH* variable.

The C++ implementation of this example must be defined in a separate class, whose name is specified in a similar way as in Java, and whose header file must be specified in the XML's *classMapping* element's *implPath* attribute. This enables OpenSplice DCG to include it in the generated C++ files. The SACPP language binding file is called *FooCustomImpl.h* in this example.

```
#ifndef SOME_FOO_IMPL_H
#define SOME_FOO_IMPL_H

#include "ccpp_FooObjects_abstract.h"

namespace example
{

    class FooCustomImpl :
        public test::Foo_abstract
    {
        public:
        void display();
    };
```

```
};

#endif
```

The *FooCustomImpl* class inherits from the base class called *test::Foo_abstract*. Therefore, the file defining the base class must be included. In this example, that file is called *ccpp_FooObjects_abstract.h*: it consists of the name of the original DLRL Object IDL file, prepended by ccpp_ and closed by _abstract.h.

The implementation of the user-defined *display()* operation must be provided in a *.cpp* file. The file's name and location is not important to the DCG. However, the file must be compiled and linked into the same executable or library created by the DCG.

The operation signatures for these user-defined implementation classes can be copied from the dummy implementations generated by the OpenSplice DCG. The filename for this example's dummy implementation's is ccpp_FooObjects_impl.h: it consists of the name of the original DLRL Object IDL file, prepended by ccpp_ and ending with _impl.h: it is located in the output directory of the DCG.