DDS for LwCCM
March 2013

virginie.watine@thalesgroup.com
Component Orientation

Component Model =

- A generic packaging format
  - Deployment and configuration external to the application

- Ports to describe
  - Provided & required "services"
    - In initial CCM: facets & receptacles event sinks & sources
    - Other interactions useful for RTE systems
      ➔ Explicit dependencies

- Separation of concerns between business logic (components) and "technical" logic (containers)
  - Containers to be provided by the framework and configured
    - Providing technical support adapted to the domain
    - Mediators with the underlying platform
      ➔ Explicit dependencies
  ➔ All dependencies explicit ➔ reuse & fast integration
<<This RFP solicits submissions for an extension of the current LwCCM to include data distribution using DDS>>

Specification in two parts:

- Definition of a Generic Interaction Support (GIS)
  - Results in Extended Ports and Connectors
- Application of the GIS to DDS
  - Results in DDS Extended Ports and Connectors

Rationale for this two-parts proposal

- Interaction support is the key enabler for component definition
  - Many interaction kinds are relevant - Notably, but not only, DDS
- GIS to allow further extensions with no additional changes
- GIS to favour consistency of the new ports definition
Extended Ports and Connectors
- Generic support for new component interaction

Application to DDS
- DDS/DCPS extended ports and connectors
- DDS/DLRL extended ports and connectors

Conclusion
Generic support for new component interactions

- **Connector** = conceptual interaction entity between components
  - Actual connection may be explicit or implicit (e.g., via DDS)
- A Connector is made of several fragments:
  - Each fragment = part of the Connector implementation co-located with a component
- **Extended Port** = how the Component interact with its Connector fragment
  - Is made of 0..n provided facets ("provides") combined with 0..n required receptacles ("uses")
New keywords introduced to:

- Define a new port (*porttype*)
- Declare an port for a component (*port*)
- Declare an inverse port (*mirrorport*)

An inverse port is the “mirror” of an extended port
  - All *uses* translated to *provides* and vice-versa
  - Inverse ports are very useful at least for connectors
Example:

```cpp
// interfaces
interface Pusher {
    void push (in TheDataType data);
};
interface PushControl {
    void suspend ();
    void resume();
    readonly attribute long nb_waiting;
};

// port type definition
porttype ControlledPusher {
    provides Pusher pusher;
    uses PushControl control;
};

// Component declaration with a port
Component C {
    port ControlledPusher the_port;
};
```
Extended IDL3 (IDL3+) to IDL

Extended IDL3 can be easily translated in plain IDL3

- With simple transformation rules
- Example:

```idl
Component C {
    provides Pusher the_port_pusher;
    uses PushControl the_port_control;
};
```

Extended IDL3 (with extended ports) → IDL3 Equivalent (after transformation) → IDL Local CIF → IDL Equivalent (IDL2) including expanded ports operations
New keyword to declare a Connector

- Connector declaration (**connector**) similar a component declaration
- **Example**

```plaintext
connector cnx {
    mirrorport ControlledPusher controlled_push;
    provides Pusher raw_pusher;
}
```

**This declaration:**

- Is not translated in the plain IDL3 (does not affect the components)
- Is used to provide type information at the assembly level
D&C View of Connectors

Connectors == Composite components

- Connector fragments = plain basic components

At assembly level:

- Connectors are seen as connection entities between components
- Fragments do not appear
- Only connections between extended ports are described

At deployment level:

- Connectors are flattened in fragments (exactly as composite components are substituted by their included components)
- Fragments are part of the CDP (shall be generated at planning phase)
- Basic connections (facet to receptacle) are described
All the new constructs may be parameterized

- Parameterization very useful to capture generic interaction semantics
  - Example: a `ControlledPusher` valid for any data type and not only `TheDataType`

Parameterization placed at module level (template modules)

- Modules are the only grouping constructs in IDL
- Always needed to group in the same template scope port types and connectors (as they need to result in matching instantiated interfaces)

- No need for naming conventions for instantiated types
  - (in line with other IDL practices)
Parameterization: What can be Templated?

Modules

- With the following restrictions
  - A template module cannot embed a template sub-module
    - Rationale: would result in over-complication
  - A template module cannot be re-opened
- Means that all embedded constructs are de facto templates
  - With the consequence that they are in the same template space
    - Embedded modules can provide structuring if needed

Only modules

- Template modules are required for GIS
  - Interfaces “used” or “provided” through extended ports are required to be the same as the ones “provided” or “used” through connectors
- Template modules are easier to add to existing IDL definition
  - Looks as an extension rather than as a revolution
- Template modules offer all the needed flexibility
  - All embedded constructs are de facto parameterised
Templates: Formal Parameters

One to many formal parameters

Type of formal parameters can be:

- Any type
  - introduced by `typename`
- Some more restricted:
  - `interface, valuetype, struct, union, exception, enum, sequence`
- Const primitive types
  - `const { [unsigned] short, [unsigned] long, [unsigned] long long, string...}
  - Purpose is to allow passing any constant
- `sequence<T>`
  - with T being a previous formal parameter
  - The concrete parameter will have to be a previously instantiated sequence<Foo>
    (assuming that instantiation is created for T = Foo)
Templates: Usage

Two steps:
- Definition of the template ➔ parameterized types
- Instantiation with concrete parameters ➔ concrete types

Instantiation rules
- Explicit instantiation required for the template module
  - Explicit instantiation required to allocate a name
  - No on-the-fly instantiation, that would result in an anonymous type
    - rationale: compliance with current IDL strategy regarding template instantiations (anonymous types proven difficult to map in some languages)
- Explicit instantiation of the module ➔ instantiation of the embedded/referenced constructs
  - The embedding structure (module instantiation) provides a namespace that isolates the implied instantiations (no other name required)
    - Embedded constructs
    - Referenced template module (see below)
Templates: Syntax as Simple as Possible (1/2)

Template definition

```
module template-name < {parameter-type formal-parameter}+ > {...}
```

- **parameter-type =**
  - typename
  - interface, valuetype, struct, union, exception, enum, sequence
  - const any-primitive-type
  - sequence<a-previous-formal-parameter>

- **Example**
  - module MyTemplateModule<typename T> {...}

Template instantiation

```
module template-name < {parameter}+ > instantiation-name;
```

- **Example**
  - module MyTemplateModule<Foo> MyFooMod;

- **Rationale:**
  - moduledef is already used in the IFR definition (ModuleDef)
  - typedef would be confusing (it does not provide a type)
  - No new keyword
Template reference

- It is sometimes needed to reference an externally defined template module inside another one and to give it a name
  - Purpose is to reuse definitions
  - Constraint: the formal parameters of the referenced template module have to be a subset of the formal parameters of the referencing one
- Instantiation will be performed when the referencing template is instantiated
- Syntax

```
alias template-name < {formal-parameter}+ > alias-name;
```

- alias-name can be identical to template-name
- Example:
  - module MyTempModule <typename T1> {...}
  - module MySecondTempModule <typename T1, typename T2>{
    alias MyTempModule<T1> MyTempModule;
    ...
  }
GIS Conclusion

Generic Interaction Support provides great flexibility
- By decoupling the programming contract from the actual interaction
- By allowing definition of new interactions without further changes in CCM and D&C

GIS may support sophisticated interactions
- As programming contracts, Extended Ports may combine freely basic facets (caller) and receptacles (callee)
- As interaction artefacts, Connector Fragments are restriction-free

It reuses as much as possible from CCM and D&C
- Extended Ports benefit from the CIF unchanged
- Connectors are plain composite components wrt D&C

Integrated in CCM since v3.2
*(Template modules could be used for many other purposes)*
Extended Ports and Connectors

- Generic support for new component interaction

Application to DDS

- DDS/DCPS extended ports and connectors
- DDS/DLRL extended ports and connectors

Conclusion
Apply Separation of Concerns also to DDS

- Keep the business code apart from the "technical" code
  - For DDS, means externalising the creation and configuration of the DDS entities
- Integrate DDS configuration in the general D&C scheme

Simplify the use of DDS

- Easy “out of the box” DDS operation
  - No entity creation / configuration burden
  - Simpler API
- Easy and secured QoS setting

What should be avoided

- Performance degradation
- Usefulness degradation
How Designing DDS Ports?

Identify most commonly used DDS patterns

A DDS pattern =
- A set of roles
- Their related DDS entities and how they are to be used
- Their related QoS settings (QoS profiles)

Examples of patterns:
- State (Observer / Observable)
- Event (Supplier / Consumer)
- Continuous Data (Supplier / Consumer)
- Alarm (Activator / Handler)
- WatchDog (WatchDog / Monitor)
- Consensus (Participant)
- LoadBalancing (LBServer / LBClient)
- ...

Cette présentation et son contenu sont la propriété du groupe THALES
Patterns ➔ Ports, Connectors, Configuration, etc.

One Role should be supported via (only) one Extended Port

- But no need for a different port type per role
  - One port = one programming contract
    - Usually linked to one Topic (parameter), but several Topics could be handled
- Variability points:
  - Read or write
  - One or Many data instances......

Connector Fragments correspond to Roles:

- Hide related DDS entities and implementation logics (ex: take vs. read)
- Configured by means of an integrated consistent QoS Profile (expressed in XML)
- All the roles of a given pattern can be gathered in a Connector
  - Does NOT imply physical connection between the components
  - Helps logistics
Connector =/= Physical Connection

Component 1

Component 2

Classical DDS Participant

collocated
DDS Normative Ports and Connectors

Following those principles, a very huge set of ports could be defined

**DDS4CCM standardises:**
- A set of DDS/DCPS Ports
  - Covering all sensible programming contracts involving one Topic
- Two very typical DDS/DCPS patterns
  - State Transfer & Event Transfer
- Optionally, a composition rule to define DDS/DLRL ports and connectors

**Nothing prevents to create other DDS ports and/or connectors in order to implement other DDS use patterns**
- In particular, by reusing some standardised interfaces
DDS/DCPS Port IDL Definition - Rules

DDS ports made of several 'basic' ports

- One basic port by 'area of functionality' X 'interaction direction'=
  - Areas of functionality:
    - data access
    - status access
    - DDS entity access
  - Interaction direction = whether the component invokes (uses) operations on the fragment or provides a callback to the fragment

Parameters as simple as possible:

- Simplified ReadInfo to accompany data values
- Use of port attributes to capture recurrent operation settings
- Exceptions to report errors
DDS-DCPS Ports Parameterization

DDS-DCPS ports and connectors meant to be parameterized by one data type (the one of the related Topic)

All are grouped in one module, with a template sub-module for the part specific to the data type:

```
module CCM_DDS {
    // declarations that are not T specific
    module Typed <typename T, sequence<T> Tseq> {
        // declarations that are T specific
        // sequence<T> to be provided at instantiation
    }
}

module CCM_DDS::Typed<Foo, FooSeq> FooPorts;
```

Note: `sequence<T>` provided by the application so as not to create a new type
Data Access – Writer Side
- Writer when the instance lifecycle is not a concern
- Updater when the instance lifecycle is a concern

Data Access – Reader Side
- Reader to just read the data
- Getter to wait for fresh data
- Listener to be notified of a fresh value of an instance whose lifecycle is not a concern
- StateListener to be notified of instance state changes

Other interfaces
- InstanceHandleManager root for Writer and Updater
- DataListenerControl / StateListenerControl
- PortStatusListener / ConnectorStatusListener
**Writer Side**

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

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

**Reader Side**

```cpp
porttype DDS_Read {
    uses Reader data;
    uses DDS::DataReader dds_entity;
    provides PortStatusListener status;
};

porttype DDS_Get {
    uses Reader data;
    uses Getter fresh_data;
    uses DDS::DataReader dds_entity;
    provides PortStatusListener status;
};
```
Reader Side – ctn’d

porttype DDS_Listen {
    uses Reader data;
    uses DataListenerControl data_control;
    provides Listener data_listener;
    uses DDS::DataReader dds_entity;
    provides PortStatusListener status;
};

porttype DDS_StateListen {
    uses Reader data;
    uses StateListenerControl data_control;
    provides StateListener data_listener;
    uses DDS::DataReader dds_entity;
    provides PortStatusListener status;
};

Note that all the reader side extended ports comprise a Reader to set the read context and perform the init phase
State Connector

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

Event Connector

collector DDS_Event : DDS_TopicBase {
  mirrorport DDS_Write supplier;
  mirrorport DDS_Get pull_consumer;
  mirrorport DDS_Listen push_consumer;
};
Same objectives...
- Simplify programming
  - No entity creation / configuration burden
  - Simpler API
- Easy and secured QoS setting, in the general D&C scheme

...But a different realisation
- DLRL offers plain object manipulation that interfaces under the scene with DCPS operations → very simple
- DLRL supporting entities to be created by the Connector fragment
  - Object Homes and Cache
- All topics used by a DLRL Cache are to be managed consistently
  - To be grouped in a single DDS/DLRL Extended Port
- A fixed set of DLRL ports and connectors cannot be designed
  - Instead basic building blocks and a composition rule
DDS/DLRL Ports and Connectors (1/2)

**DLRL Extended Port**
- Should give access to all related objects
- Comprises:
  - A receptacle for each ObjectHome
  - Another receptacle for Cache **functional** operations
    - I.e., excluding all the operations that are related to configuration (thus will be for the only use of the Connector implementation)

**DLRL Connector**
- Natural support to gather all the DLRL extended ports related to the same set of topics in order to master their configuration system-wide
- Could provide as many mirror ports as there are DLRL participants to this set of topics
  - Nothing prevents several DLRL participants to share the same object model, thus the same DLRL port
Example

- **A DDS/DLRL Extended Port**
  
  ```
  porttype MyDlrlPort_1 {
    uses DDS_CCM::CacheOperation cache;
    uses FooHome foo_home;  // entry point for Foo objects
    uses BarHome bar_home;  // entry point for Bar objects
  };
  ```
  
  - DDS_CCM::CacheOperation is a subset of DDS::Cache
  - FooHome and BarHome are provided by the DLRL tooling

- **A DDS/DLRL Connector**
  
  ```
  connector MyDlrlConnector : DDS_CCM:DDS_Base {
    mirrorport MyDlrlPort_1 p1;
    mirrorport MyDlrlPort_2 p2;
    mirrorport MyDlrlPort_3 p3;
  };
  ```
  
  - Inherits from DDS_Base to be given a ConnectorStatusListener and to set domain id and QoS profile
Extended Ports and Connectors
- Generic support for new component interaction

Application to DDS
- DDS/DCPS extended ports and connectors
- DDS/DLRL extended ports and connectors

Conclusion
DDS4CCM specification formally published

- [http://www.omg.org/spec/CORBA/3.3](http://www.omg.org/spec/CORBA/3.3)
- [http://www.omg.org/spec/dds4ccm/1.1](http://www.omg.org/spec/dds4ccm/1.1)

DDS4CCM implemented

- By Thales (Cardamom & MyCCM)
- By RemedyIT in the scope of CIAO / Dance

Simple DDS ports added to CCM – Extension possible

GIS fully generic and root for a CCM revival

- Moved to CCM (now in CORBA specification / part 3 – chapter 7)
- In the process of being used for other CCM ports & connectors
  - AMI (Asynchronous Machine Invocation), Events…
- Note: will allow CCM w/o CORBA (aka UCM)
  - e.g. by implementing those connectors on top of DDS
Thank you for your attention
Questions?