SPECIFYING A UNIFIED COMPONENT MODEL WITH UML AND ITS EXTENSION MECHANISM

Ansgar Radermacher
Arnaud Cuccuru
Sara Tucci
Chokri Mraidha

{FIRSTNAME.LASTNAME}@CEA.FR
CEA LIST / DILS
• UCM key concepts
• Why UML?
• UML component model extensions
• Modeling UCM concepts with UML
  ▪ Focus on software model
  ▪ Show examples
• Conclusion
UCM KEY CONCEPTS

- Flexibility with respect to ports
  - Extended ports as standardized in DDS4CCM / AMI4CCM

- Flexibility with respect to interaction components
  - Generic interaction support (GIS) as standardized in DDS4CCM
  - Adaptation to application types via templates

- Flexibility with respect to container
  - “Plug-in” capability
  - QoS4CCM
  - Concurrency management policies

- Deployment
  - Local and distributed deployment
WHY USE UML?

• OMG Standard
• Base component model with
  ▪ Components owning ports
  ▪ Composite components, assembly of parts within
  ▪ Configuration via instance specification
  ▪ Possibility to export IDL as PSM

• Extensibility via profiles, notably
  ▪ MARTE
    Non-functional properties (resource/time constraints)
    GCM (Flow-ports - Client/server ports)
  ▪ FCM (Flex-eWare component model)
    Unify component models of CCM and Fractal
    Flexible (more later)
WHY USE UML?

• UML & MARTE & FCM developed/used in several projects

• Tool support for modeling
  ▪ Papyrus – configurable towards a specific domain
    o [link](http://www.eclipse.org/papyrus)
  ▪ RSA, Rhapsody, Enterprise architect, …

• Lesson learned from UML: flexibility needs guidance
• Flexibility for the following concepts:
  ▪ Generic interaction support
  ▪ Extended ports adapted to interaction mechanisms
  ▪ Configurable container services
    ⇔ Shares UCM objectives

• Typical system, consisting of three parts

  ▪ A model describing the software components,
  ▪ A model describing the hardware platform and
  ▪ A model describing component instances, including
    ○ Configuration and allocation to hardware components (as in DEPL)
• Decomposition of the system into components.

Since producer and consumer call operations (the latter to actively retrieve data), the two ports would be incompatible without using a specific interaction component, in this case, a FIFO. Use the context menu on a selected connector to specify the interaction component.
Package with template signature

⇒ Group elements that share same signature
⇒ Simply relationships between these
⇒ Similar to DDS for CCM (e.g. Typed <typename T,…>).
(Flow-) ports defined by abstract interaction component

FIFO_impl

pushP

+ m_fifo : dataType [*]
+ m_size : Long

Configuration attribute: the size of the FIFO

This is an implementation of a FIFO of a configurable size (default: 20 elements).

KeepLast

pushP

pullC

Simple storage of last data that has been received. Unlike FIFO, data can be read several times.
EXAMPLE – SOCKET CONNECTOR

- Composite class encapsulating fragments

A composite implementation of a socket-based connector. It consists of 3 parts:
- Client: sends data to socket associated with location of target component
- Server: receives request from socket runtime, registers itself with the socket runtime
- sr: shared by all sockets. Manages a list of server targets. It is also required by the client, since its static method "getSocket" is called directly, i.e. without port/connector by the client implementation. This method only works, after the initialization (activate operation of life-cycle interface) has been called.
• Models must respect compatibility between ports (validation)
• Ports: CCM facet and receptacle (offer or call operations)
• Other port kinds defined by mapping rule (extensible) from port type to provided and required interfaces.
• Mirror-port = port conjugation in UML
• DDS extended port:
• Example: port for publishing is typed with UML data-type
  ▪ Associated mapping: operation to send instance of data type
  ▪ Keep core specification relatively small, add flexibility via extension mechanisms.
• Ports are characterized by kind and type

**Standard Facet**

<table>
<thead>
<tr>
<th>LifeCycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>lc: ILiFeCycle</td>
</tr>
</tbody>
</table>

**Abstract class ⇒ component type**

Part of the Qompass model library. Provides the specific port "Ic" (short for lifecycle). The "activate" operation of this port is called by generated boot-code after instantiation and configuration of a component. The deactivation is called before disposing a component.

**Variant of the consumer component that uses a "pull style", i.e. actively demands whether new data is available. Since the component is active, it needs to have its own thread. This is realized by the combination of the "activate" operations that executes the activation operation in its own thread. Furthermore, the PullConsumer supports the "runStartThread" interceptor which intercepts the lifecycle port.**

**interface PullConsumer_MyData {**
| MyData pull(); |
| boolean hasData(); |
**}**

**Pull flow port, typed with datatype, derived required interface**

kind=PullConsumer, requiredInterface=D_PullConsumer_PubData
• Ports are characterized by kind and type

Extended port definition as UML

Component

LifeCycle

 DDSConsumer_impl

DDS consumer port, derived required interface

kind=DDS_Listen, requiredInterface=D_DDS_Listen_R_PubData

Derived interface

interface DDS_Listen_R_MyData {
    void reader_read_last(inout data, inout infos);
    void reader_read_one_all(in datum, inout ...);
    void dds_entity_read (…)
}

Concatenate port & operation name
EXTENDED PORTS

• Should we allow extended ports owning extended ports?

• Extended port is similar to AADL port group
  \[\Rightarrow\text{Well defined UML/MARTE or SysML equivalent}\]

• Should we support different options or fix a single one?
  \[\begin{itemize}
    \item Flatten extended ports (as in DDS4CCM)
    \item Single port with merged interfaces (as shown)
    \item Port is realized by dedicated class (nested class)
  \end{itemize}\]

  \[\text{Balance flexibility vs. source code compatibility}\]
Composite class encapsulating executor
Handles technical issues, e.g. thread protection

Container elements
Intercept or add (extend) the component definition

Interception via connector

Separation of concerns
CONTAINER – HOW TO SPECIFY?

Specify container via *rules*

Rule = composite class, parts = container elements

Define semantics of multiple rule application (merge)
- Concatenate interceptions
- Aggregate additional elements
Combination of interception and extension
Close to UML specification of state machine behavior
• Problem: thread management integrated into connector
• E.g. socket listener thread with leader/follower pattern

• Container needs to supply thread to connector
• Need standardized interfaces

• Options
  ▪ Use container for the connector
  ▪ Container encapsulates executor and connector fragments
  ▪ Container accesses connector via additional port (or bi-directional)
SAMPLE CONTAINER & INTERACTION COMPONENTS

- **Container**
  - Reflective information (self-description, timing, …)
  - Transparent diagnostics/runtime monitoring
  - Access control

- **Interaction components**
  - Abstraction from communication mechanism (socket, CORBA, …)
  - Fault tolerance (in conjunction with voter, …)
  - Link simulation (error injection, delay)

Both are part of a *domain specific model library*
COMPONENT TO OO-MAPPING

• Code generation from UML well defined for OO concepts
• Define generation for component concepts
  ▪ Ports – internal (code inside) and external API
  ▪ Connections – assembly and delegation
• Existing CCM mapping (CIF) not only suitable solution
  ▪ Robotic middleware Orocos maps all ports consuming data to a single operation (distinct in-parameter for each flow port).

⇒ May need more flexibility
  … but don’t want to loose code compatibility (?)
- Hierarchical composition
- Connectors represent physical connections
- Ports represent physical interaction points such as I/O ports.

The hardware architecture is also defined by means of a composite class (comparable to the software architecture). The types (NodeA and NodeB) can be stereotyped with information about the used target platform (FOM stereotype target). As for software architecture, instances specification for nodes are derived from this class. Use "Q compass: create platform definition" in the context menu.

The physical connection between two nodes is represented by a UML connector. Currently, this information is not evaluated by Q compass, but it is planned to analyze for instance whether the throughput (specified by means of MARTE) is sufficient with requirements of the software.
• Task of the deployment model:
  1. Allocation software to hardware component (component to threads/processes, processes to nodes)
  2. Configure instances of software components (standard UML instance specifications with slot/values).

MARTE allocation between parts or instances
Reuse of standard UML ⇒ code generators

M2M transformations
1. container,
2. interactions,
3. component to OO
CONCLUSIONS

• UML is suitable for representing flexible component models
• Right level for model validation (and architectural exploration), e.g.
  ▪ Port compatibility
  ▪ Schedulability analysis
• Production of IDL/XML artefacts for compatibility
• Need to balance flexibility and compatibility
  ▪ Among different UCM variants
  ▪ Already used MW solutions, such as robotic MW
  ▪ Added value compared to less flexible solutions?
Questions?

ansgar.radermacher@cea.fr
• Need to instantiate behavior with formal parameter

• E.g. Class = formal parameter for a container extension

⇒ Behavior description in form of text template (Acceleo)

```
[template (clazz : Class)]
[for (sm : StateMachine | ownedBehavior->select(oclIsKindOf(StateMachine)))]
switch(m_currentState) {
  [for (state : State | sm.region.subvertex->select(oclIsKindOf(State)))]
  case [clazz.name/][state.name/]:
    ...
  [/for] ...
```
Model transformation (UML $\Rightarrow$ UML) replace a UML connector with a part

**Connector type & implementation need to be adapted to communicate Data**
- Calculate binding based on port type
- Instantiate package template
- May imply transformation of source code embedded in model

Declarative information about connector type
• Code generation directly from UML, i.e. no intermediate IDL mapping.

• Our idea: code generators should be rather simple
  ▪ support variability via model transformations executed immediately before code generation.
  ▪ Code generation for OO concepts is pretty much the same when targeting an OO language, there are different ways to map ports and connectors.