HOW ADLS CAN HELP IN ADAPTING THE CORBA COMPONENT MODEL TO REAL-TIME EMBEDDED SOFTWARE DESIGN

Sylvain Robert[1], Ansgar Radermacher[1], Vincent Seignole[2], Sébastien Gérard[1], Virginie Watine[2], Stéphane Ménoret[2] and François Terrier[1]
[1]CEA-LIST, CEA/SACLAY, 91191 Gif sur Yvette, France; [2]Thales/ALICE pilot program, Thales Communications, 91300 Massy, France

Abstract: Coping with the increasing complexity of software in embedded and distributed real-time systems is becoming a major concern. Even if promising as far as this latter aspect is concerned, design techniques issued from the middleware components (or framework-based) approaches have until now fall short in achieving their breakthrough in the real-time and embedded community. They are usually perceived as complex, monolithic and resulting in oversized applications, and thus, as not adapted to RT/E software development constraints. In an attempt to bridge this gap, we[1] aim at contributing to the adaptation of the lightweight CCM [1] to real-time and embedded systems. The originality of our approach, mainly resides in the emphasis on high-level (or design-time) issues of the development process, on the contrary to the usual focus on low-level ones: we raise QoS issues from implementation level to analysis and design level. In such a process, we have found it would be worth integrating considerations from the software architecture/ADL field in middleware components approaches. We especially claim that interactions configurability at design time is a major requirement in the class of systems we target and that, on this latter aspect, middleware components approaches could benefit from a separation of concerns between computation and interactions, as in most ADLs.

Key words: Real-Time Embedded, CCM, ADL, Connectors, Components

[1] The work set out in this paper is performed in the context of the ICE (Interaction of Components during Execution) research project, lead jointly by CEA-LIST and Thales/ALICE pilot program. For further information, see http://www.carroll-research.org.

1. Introduction

Real-time embedded software is generally considered as a category of software in which resources are constrained: the application design has to take into account such aspects as power and memory consumption, on top of classical real-time constraints. Thus, common reasons why middleware components approaches are considered to be poorly adapted to real-time and embedded systems area is that the runtime platforms consume too much resources (e.g. memory footprint), that they result in "heavyweight" components, and eventually because they do not properly address real-time constraints.

But this stance needs to be moderated: the term "RT/E systems" refers to a large family, where the requirements are quite heterogeneous. In particular, if the assertion made above about resources constraints applies to certain members of this family -this is for instance the case of distributed command and control systems, where sensors/actuators are associated with small units of computation, usually provided with very limited resources-, it is not completely true for all: for instance, some embedded communication systems own larger hardware configurations (about 1-2 Mb of RAM/ROM and even larger, sometimes close to those of common workstations). Then, middleware components approaches have numerous inherent advantages from which RT/E systems could benefit: they provide applications with standard platforms comprising an extensive set of extra-functional services (e.g. distribution management, security); they come with well-defined component's abstract models; they specify architecture of execution frameworks; they natively address deployment and packaging issues, and they provide guidelines which structure the components development process. Moreover, efforts have already been made to cut down the size of middleware platforms, for instance the initiative Lightweight CCM [1] which proposes a "low-fat" version of CCM mostly dedicated to resources-constrained systems. Even real-time aspects have (somehow) been addressed, by extending the original platforms with dedicated mechanisms: we can quote the example of real-time CORBA [3].

However, this continuous upgrading towards "RT/E-compliant" middleware platforms by means of iterative functionalities addition / subtraction diverts attention from several of their initial cons: they are extremely complex to understand (and to use), and they lack design-time configurability. The usefulness of the large range of provided services is mitigated by the difficulty for the developer to appropriate these services in accordance with his specific requirements. Thus, our belief is that major issues for bridging the gap with RT/E reside not only in enhancing the platforms themselves, but also in simplifying their use and, in enabling the application designers to adapt the platforms to their specific needs. The objective is thus to identify key issues which would contribute to answer these two concerns. Researchers from middleware have obviously a bottom-up approach, which consists in a constant rising in abstraction -from, for instance, CORBA to CCM-. In order to have a complementary view, we have chosen to consider approaches which have had a top-down process, hence our focus on ADLs-based approaches. This paper presents the first step of our work, in which the focus is on *interactions between components* in CCM. It explains why we have chosen to introduce the concept of ADL *connector* concept in CCM, and gives some insights about how we intend to proceed. The structure of this paper is as follows: Section 2 is an introductive overview of CCM, with a focus on the aspects linked to our study. In Section 3, our focus on interactions is justified, and we describe the rationale which has lead us to the "CCM connector" choice. Section 4 describes the primitive set of connectors we have built for integration in CCM. Eventually, Section 5 gives some insights about the needed CCM extensions, before concluding.

2. Introducing CCM

This section presents some fundamentals of framework-based approaches and gives an overview of the CCM.

General concepts of framework-based approaches

Frameworks have been recognized to capture best practices in some engineering domains: they provide templated implementations of patterns that were seen as leveraging issues in particular domains. A main characteristic of many frameworks is the inversion of the control flow: the application specific code is invoked by the framework to perform specific tasks and not vice versa (as opposed to a library).

Middleware components approaches use the abstraction of an encapsulated *component.* A key characteristic of a component is that it is loosely coupled, i.e. there are no dependencies to other components – only to certain interfaces. This is achieved by the separation of interface and implementation. A component *implements* a set of interfaces and it *requires* components implementing a set of other interfaces.

A component requiring an interface can be *bound* to another component offering a *compatible* interface (usually compatibility is defined via inheritance hierarchies or via a structural equivalence). The bindings are usually specified by a separate assembly descriptor.

Components are often embedded into *Containers* that mediate requests from and to the component. Containers are part of the *execution framework* for components that provides frequently used services such as persistency or security. It can be tailored towards the application needs. The bindings between components are enabled by means of a packaging format as well as a deployment infrastructure.

The CORBA Component Model

In the CORBA Component model, the external connection points of components are called *ports*. There are four different kinds of ports, called facets, receptacles, event sources and event sinks. Facets correspond to provided (implemented) interfaces, receptacles to required interfaces (containing method calls specified in IDL – Interface Definition Language). Event sources and sinks are the event based counterparts to receptacles and facets. Sources emit events of a certain type, event sinks are named connection points into which events can be pushed. Besides the facets, the component always implements a primary interface, called equivalent interface. Figure 1 shows a CCM component. Receptacles can be connected to facets, as depicted for the second receptacle on the right.

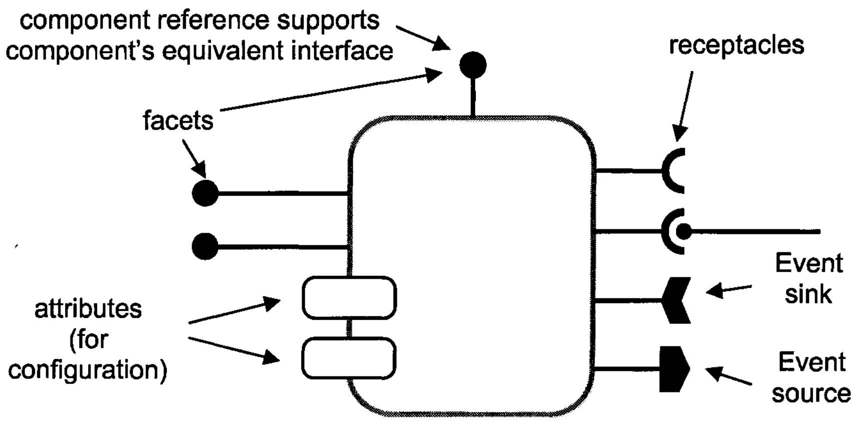
component reference supports component's equivalent interface
receptacles
facets
attributes (for configuration)
Event sink
Event source

Figure 1: CORBA Component Model

Component instances are managed by a component home, which is in charge of instantiating and deleting components, i.e. the components life cycle (not shown).

We give below an example of a component description in IDL3, as well as the description of its associated interfaces, event types, and home:

```
interface intf1 {
  void do_something( in string s );
};
eventtype E {
  public string s;
};
component C {
  provides intf1 a_intf1;
  consumes E a_E;
};
home C_home manages C {
  attribute string foo;
};
```

The CCM introduces a so called *component implementation framework* (CIF) with the objective to separate concerns: the component should not be responsible for instance to manage connections or know how to emit an event via an underlying CORBA service.

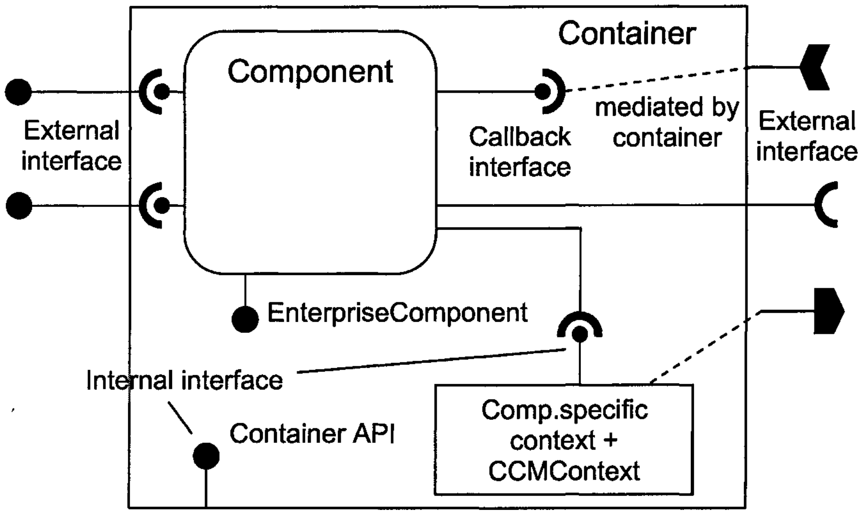

Figure 2: Components are embedded into a container

The *container* is the glue between a component and the underlying execution platform as shown in Figure 2. It provides a programming interface for the component, the internal interface. This interface consists of a standardized container API and a component specific context object. The code of the context object is completely generated from the IDL definition. It provides an interface that allows the component to retrieve references for used interfaces and operations related to the publishing of events (a reference to an implementation of the Context is passed by the container to the component). If the latter are invoked, the context object generates suitable events of the underlying event mechanism, usually the CORBA notification service. For incoming events, the component has to implement a callback interface and the container mediates the event by invoking a method provided on this interface. Please note that this form of event delivery does not allow the component to poll for new events, it is always "pushed" into the component. Since the container shields the component by intercepting its communication, it can implement some non-functional requirements such as logging or security.

The CCM provides an explicit deployment step within the development process. In this phase, component instances (including values of their attributes) as well as their interconnections are specified. This task might be supported by a suitable tool. The specification is done by means of different descriptors files in XML, in particular a component descriptor and component assembly descriptors.

We omit further details of the CCM, since they are not necessary in order to understand the rest of the paper.

3. Answering the need for a CCM interactions improvement

In [14], the author states that (static) configurability is a paramount concern in real-time embedded systems. This is particularly true when talking about components interactions: industrial practices tend to favor different architecture styles depending on the application domain considered. As examples, in the field of communication protocols, a layered approach together with a pipes and filters generalized pattern is commonly used. In the signal processing domain, a data-flow approach is considered most of the time. In some distributed systems like command and control-ones, a common practice is to use variations around publish-subscribe. The variety of practices thus highlights the necessity, to provide developers with means to properly configure the interaction mechanisms to be used in their applications and, to ensure the adherence to specified features (e.g. QoS ones) at execution. And yet, as most middleware components approaches, CCM offers rather poor means to express and configure components' interactions modalities at design time.

In order to bridge this gap, we have first considered various attempts to adapt CBSE to real-time embedded software design. In these latter, three issues are generally distinguished:

- Adaptation of the *component model*, which is an informal representation of what should be a component. It usually specifies the content (e.g. binary [12]) of components, the interaction points with the environment (e.g. event channels [13]), and often a set of non-functional features associated to the component. The originality of these approaches compared to classical ones usually lies in this latter point. For instance, many authors seem to agree on the integration of WCET in components.

However, in some cases, the list of features to integrate can be much larger, e.g., memory needs, deadline, power requirements.

- The *development process* of an application with components is often described as a complement of the component model. This issue often emphasizes the need to enable the reuse of components and the necessity to have a connection with off-the-shelf validation tools, for instance to assess the schedulability of the resulting system. The guidance provided for development process may also be a mean to introduce common real-time architecture patterns [11].
- The last aspect usually addressed is *architectural configuration*, i.e. representation of applications by connected graph of components. The approaches demonstrate the way components may be composed, or how the resulting system architecture may be validated, e.g. with regards to interfaces compatibility or exclusion constraints.

A common point arising from these approaches is the focus on the component model specification. Very little attention is paid to the interaction modalities, which are implicitly specified by the components' interfaces and the representation of the connections between these interfaces in the architectural configuration. This focus is also obvious in [9], where the authors list a set of industrial requirements for the CBSE approaches to be suitable for automotive real-time embedded systems: all the requirements regard either development process or component model issues.

The most relevant answer to our concern was actually found in approaches issuing from the Software Architecture / ADLs field. ADLs provide features for modeling software systems' conceptual architecture, independently from implementation concerns [16]. An ADL usually provides, on top of the *component* modeling concept, the notion of a *connector*, which represents an architectural building block used to model interactions among components and rules that govern those interactions. Connectors are considered as first-class model elements [6] in the sense that they have quite the same attached features as components, e.g. interfaces, semantics, constraints, non-functional properties. For instance, Unicon [4] proposes to specify a protocol for each connector which provides a connector type, and assertions constraining the interaction (e.g. roles). Each connector specification provides also an implementation which may be built-in. Non-functional attributes may also be attached, e.g. real-time ones to perform a schedulability analysis. In [5], a comprehensive framework is provided to perform a classification of all kinds of connectors. This approach chooses to classify connectors according to the service they provide, their types, and the dimensions along which these types may be refined. This work follows a bottom-up pattern: instead of designing connectors and implementing the corresponding mechanism, the authors have made an attempt to perform an exhaustive classification of existing interaction mechanisms in software. In [18], the author proposes a radically different view of connector. After having noticed that usual connectors address rather primitive interactions mechanisms, he proposes to consider connectors as "pattern-like transferable abstractions": connectors express only abstract interactions -mainly specified by roles and protocols- which have no direct mapping to the implementation of the application.

As noted in the introduction, one of the CCM drawbacks is that it does not provide an abstract view of many aspects relevant for RT/E systems (these are hidden in the implementation that may configure for instance a CORBA timeout for synchronous operations). Many facilities are provided by the communication layer, but they require a high level of expertise to be used properly. Introducing connectors would constitute an opportunity to provide a high-level translation of these mechanisms in a more understandable manner. Connectors are also likely to facilitate the integration, reuse and replacement of components, especially when building applications from off-the-shelf ones. These connectors should not, of course, result in a one-to-one representation of the underlying mechanisms. The aim is to provide the developer with easily configurable interaction facilities, shielding him the complexity of the platform. However, unlike [18], we believe that introducing connectors is bound to have an impact on the CCM component model and accompanying artifacts (e.g. IDL), and that connectors shall have an implementation counterpart. On top of that, the native architecture of CCM appears to be adapted to perform the introduction of connectors. For instance, with its intermediate positioning between the communication layer and the application, the components' container is a relevant place holder for an implementation of the connectors. Another important issue to consider is the integration in the development process: our opinion slightly differs from the "connectors as first-call elements" software architecture's leitmotiv. Since we do not intend to act on the communication layer, but only on the CCM level, the connectors to be built will be largely constrained by the native mechanisms of the underlying platform. In our view of the development process, connectors will not be *designed* in the same way components are designed, but a set of primitive connectors will be predefined and provided to the application developer, who will be able only to *configure* them in order to fill the application requirements. Following these high-level requirements, the points to deal with are thus: first to define these primitive connectors, then to ensure their configurability, and at last integrate them in CCM. In the next section, we explain how we have dealt with the first two points, based on both a bibliographical work in real-time platform and on the connector classification framework provided in [5].

4. Interactions reification: building the primitive connectors

Our aim was twofold: identifying main interaction mechanisms available in real-time platforms, finding means to express these mechanisms by means of configurable connectors. The first point has been addressed by trying to cover a large area in terms of technological trends in real-time systems in our bibliographical work. Thus, several standard platforms offering different levels of abstraction have been considered, from operating systems (e.g. OSEK [20]) to middleware layers (e.g. Fractal [21]). Furthermore, the main computation models in real-time systems have been studied (e.g. time-triggered [19] and event-triggered architecture). We have also benefit from the experience acquired in working on the Accord/UML platform [7], a complex real-time systems development facility designed at CEA-List. In order to deal with the second issue, we have chosen to use as a base the conceptual foundations of the connector classification framework provided by [5]. Our rationale is thus the following: defining basic connectors, refining them by attaching them sets of parameters/sub-parameters which may take different values, and assessing these connectors by expressing with them the mechanisms found in the various real-time platforms.

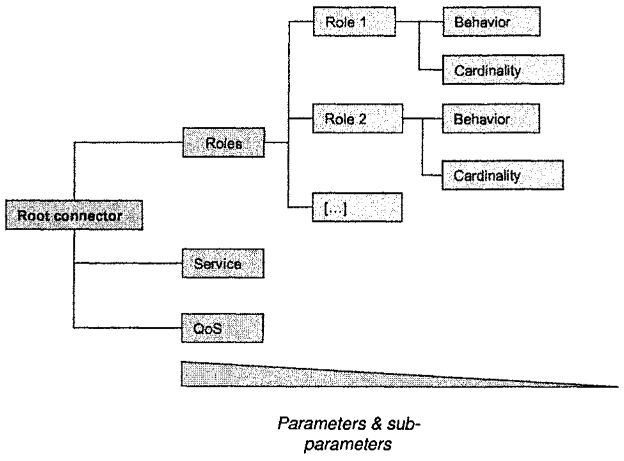

Figure 3: Root Connector

In order to set a starting point in the connectors design, we have tried to clarify the notion of interaction by introducing several basic characteristics: in our rationale, an interaction involves several *participants*, each acting in a given *role* (e.g. sender, subscriber); the *cardinality* of the interaction specifies the number of components instances associated to each role, and the *behavior* of a role precises the actions performed by a component playing this role. Interactions are also to be considered from the *service type* point of view: *communication* (i.e. data transmission) or *coordination* (i.e. synchronization). In RT/E applications, *Quality of Service* requirements (e.g. priority, deadline) may be associated to the interaction. These few characteristics specify a "root connector" (Figure 3) from which all our connectors derive.

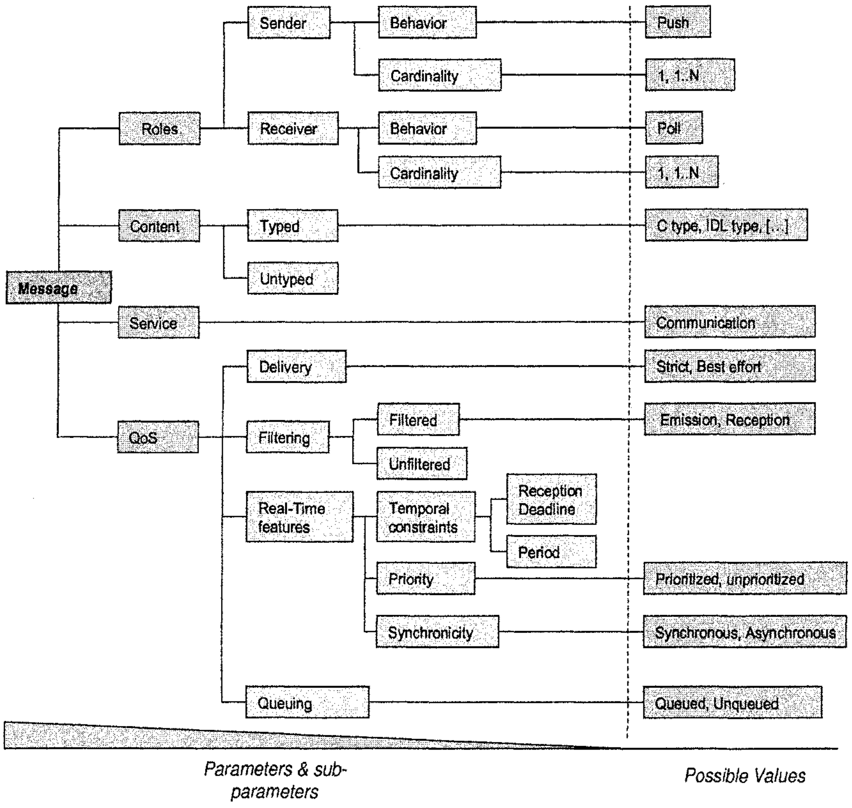

Figure 4: Message connector

From our bibliographical work, we have identified three main interaction mechanisms: message passing, event broadcasting and procedure call. These mechanisms have then been directly mapped to primitive connectors (message, event, and procedure call), and refined by adding parameters and sub-parameters, following the base pattern presented above. In the following, we detail an example of building such a primitive connector: the *Message* one. Message passing is a very common mechanism in real-time platform, which basically consists in a data exchange between tasks/components. To build the associated primitive connector "message", we have considered one by one its base parameters:

- Roles: In all platforms, message passing involves two roles: sender and receiver. Depending on the platform considered, cardinality may be 1 to 1, or n to m. The behavior is homogeneous among the different platform: the sender performs a *push* operation, and the receiver *polls* the incoming messages.
- Service: Message passing is a *communication* support.
- QoS: Depending on the safety requirements of the platform, the *delivery* of a message may be guaranteed (strict) or not (best-effort). Messages may be filtered (for instance, not accepting the same content multiple times) at reception or emission. Several Real-Time features may also be associated to a message connector: *temporal constraints* (period, deadline,...); *priority*: have all the message the same priority (prioritized)? Or can a priority be set for each message (prioritized)? *synchronicity*: asynchronous delivery, or synchronous.

The last aspect, not considered until now as not present in the root connector, is the *content*: depending on the approaches, this content may be typed (e.g. C type in OSEK) or untyped. Figure 4 shows the message connector type resulting from this analysis.

It is equally possible to describe the "event" connector, which is another common interaction mechanism in real-time systems. Briefly, let us precise that the majority of parameters identified in messages are applicable to events. However, roles behaviors are affected (for instance, the receiver will be invoked or will poll the received messages) and the focus is no more on the *content* since what matters is the *occurrence* of the event and not its format.

In the same way, we have built the *procedure call* connector. This set of three connectors has then been assessed with regards to its expressiveness, by using them to represent the interactions mechanisms offered by the studied real-time. Platform-specific mechanisms are expressed as subsets of the primitive connectors: depending on the platform considered, parameters/sub-parameters and values are removed. For instance, representing the POSIX message queues requires to remove from the message connectors the "filtering" QoS sub-parameter (and its associated values), as well as the "unprioritized" priority value, the "unqueued" queuing value, the "synchronous" synchronicity value, and the "strict" delivery value.

Once the primitive connectors built and assessed, we have looked for ways to integrate them in CCM. In the next sections, we list the issues which have to be dealt with, and give for several of them some elements of answer.

5. Lightweight CCM extension strategy

There are two extensions to the CCM model: the first is the introduction of connectors, the second an extension of port abilities.

A connector shares many properties with a component: it will offer ports as interaction points and provide attributes for its configuration. Therefore, its specification in IDL will "look like" a normal component specification in which the keyword component has been exchanged by the keyword connector. Of course, its implementation will be different to that of a component, as shown later. Unlike in the standard CCM, components will normally not be connected directly with each other, but use a connector in between. This means, that a component instance binds one of its ports to a suitable port of a connector instance which in turn will be bound via another port to the target component (in general, it should be possible that the interaction is mediated via additional connectors). QoS aspects can be configured either via the attributes of a connector or a specific, standardized interface.

The connection of components and connectors via ports motivates the second CCM extension: if a component interacts with a connector, it plays a certain *role*. For instance, it could be an event producer or an event consumer. In the CCM, ports always correspond to either implementing or using a certain interface. For general interactions, this is not sufficient, since a single role may imply using a certain interface and implementing another one. For instance, an event producer might want to receive a notification, if an event has been successfully delivered to all subscribed consumers (or if the delivery has failed). In this example, the producer role would imply implementing a delivery-status interface and using a push interface for the delivery at the same time. While it is possible to define this scenario via a pair of uses/provides specification in the IDL, we would not have the possibility to associate a single role-name with this interaction.

Therefore, we propose to extend the notion of a port in CCM into an element that consists of zero or more provided as well as zero or more required interfaces, i.e. closely resembling the UML2 [17] specification of a port – unlike in the current CCM. For the example, we would get the following definition:

```
Port PushWithNotification {
  provides IPush push;                  // provided interface
  uses IDeliveryStatus deliveryStatus;  // required intf.
}
```

A component will have the ability to use or provide ports and supply a role name, using the syntax use_port <port-name> as <role-name> (analogous: provide_port <port-name> with <role-name>). The translation of this extended IDL code into the existing one is straight forward: the use (provision) of a port is replaced by the use (provision) of the interfaces provided by a port and the provision (use) of the interfaces required by it.

The role names are important during the assembly of components and connectors. If a component instance is bound to a connector instance, each use of a port of the component has to match the provision of a port of the connector and vice versa. Provided and used ports match only, if type and role name are identical. This implicit binding via a role name avoids an additional specification (implying a further complexity). We assume that it will not impose a restriction in practice, since the use of a different role-name will almost certainly be accompanied by a different semantics that would inhibit the (unchanged) reuse of a component or connector.

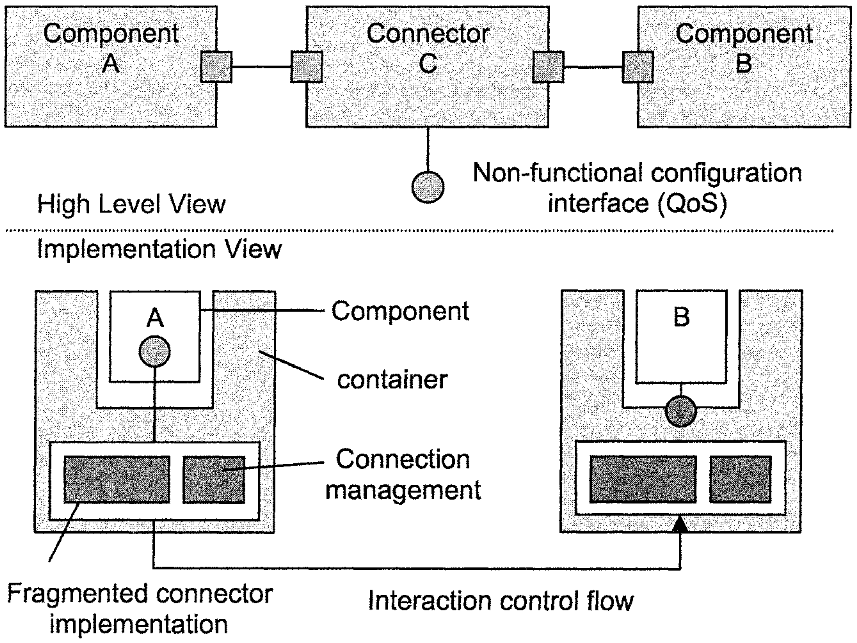

Figure 5: A first draft for architectural integration of connectors

However, the implementation of a connector – i.e. the connector's behaviour – is different to that of a normal component. It might behave almost like a normal component, e.g. in case of an event channel that is located on a specific node. However, in general, connectors can only be efficient, if their implementation consists of several parts, each co-located with a component that uses the connector in a specific role as shown in Figure 5. In addition, the connectors will be integrated in a different way with the container (it is not clear yet, whether the connectors should be considered as part of the container or not). Therefore, the CCM's code generation rules need to be adapted. It seems useful to support the fragmentation of a connector on the implementation level by different *executors* (compliant with CCM terminology; but different from component executors). Each of these executors would correspond to the use or provision of a port and would be co-located with the component bound to it.

The chosen component/connector model allows for the adaptation of invocations without modification: the connector would simply need to provide a port that assumes a certain style of usage and can internally transform and mediate requests to another port. An example is the transformation of a typed event at user level into an untyped event used by the underlying platform. It may be possible to generate the necessary adaptation code from a suitable description of the conversion.

There are two further aspects, connector packaging and assembly descriptors that need to be adapted as well. Since components are packaged into archives (containing descriptors, binary code for component), we propose to package connectors in archives, too. The rationale for this is to integrate the connector in the deployment procedure: it can be needed to instantiate part of the connector on a particular host (think about Event Channels). The component assembly descriptor format needs also to be adapted since, as shortly mentioned in the CCM description, the assembly descriptor format allows to describe direct binding between components (and matching port types), and hence does not allow to insert connectors between them.

6. Conclusions and Perspectives

We have described in this paper our ongoing work in the context of the ICE project, which aims at contributing to the adaptation of the CORBA component model to real-time software design domain. We have followed a process of real-time interaction mechanisms expression by means of connectors, supported and inspired by similar works from the ADLs area. We have laid the foundations for the next stage which will focus in concretely integrating these primitive connectors in CCM, and demonstrating the relevance of our approach through prototyping and application to use cases. The use cases selected have a close connection to "real" application domains and offer enough complexity to constitute an assessment of our conceptual and technical choices. For instance, we plan to deal with a simplified UMTS radio-protocol stack use case.

But CCM enhancement is of course still an open issue. Even if well-matured in some aspects, e.g., the extensive set of services provided with the middleware or the deployment issue, it still lacks major features. The main one regards what ADLs call *architectural configuration*, i.e. the ability to specify an application by means of a graphical representation, as well as addressing the associated issues (e.g. compositionality, refinement, scalability) [16]. Effectively, using CCM in its current state requires a sound knowledge of the underlying CORBA platform, and high skills in platforms implementation languages, which mitigate its usefulness for neophytes. Moreover, this absence of a high-level representation forbids the early validation of the developed application, an action commonly performed with ADLs.

Building this CCM architectural configuration requires in a first step to give an abstract view of the mechanisms provided by the CCM middleware platform. It sets also a need for an enhancement of the CCM component model. Our contribution regards these two issues, with a focus on interactions representation.

7. References

[1] Lightweight CORBA Component Model – OMG draft adopted specification, Object Management Group, 2003.
[2] CORBA Components, version 3.0, Object Management Group, 2002.
[3] RealTime – CORBA specification, version 2.0, Object Management Group, 2003.
[4] Abstractions for Software Architecture and Tools to support them, M. Shaw, Robert Deline et al., Software Engineering, vol. 21, number 4, 1995.
[5] Towards a Taxonomy of Software Connectors, N. R. Mehta, N. Medvidovic and S. Phadke, ICSE 2000.
[6] Software Connectors and their role in component development, D. Bálek & F. Plášil, DAIS'01.
[7] MDA Platform for Complex Embedded Systems Development, C. Mraidha, S. Robert et al., DIPES 2004.
[8] Specification for deployment and configuration of component based applications - draft adopted specification, OMG, 2003.
[9] Software Component Technologies for Real-Time Systems - An Industrial Perspective -, Anders Möller, Mikael Åkerholm et al., RTSS 2003.
[10] Towards Aspectual Component-Based Development of Real-Time Systems, Aleksandra Tešanovic, Dag Nyström et al, RTCSA 2003.
[11] Developing component-based software for Real-Time systems, Janusz Zalewski, 27th Euromicro conference, 2001.
[12] Components in Real-Time Systems, D. Isovic, C. Norström, RTCSA 2002.
[13] An Approach to Component-Based Software Engineering for Distributed Embedded Real-Time System, Uwe Rastofer, Frank Bellosa, WMSCI 2000.
[14] VEST: A toolset for constructing and analyzing component based operating systems for embedded and real-time systems, John A. Stankovic, Lecture Notes in Computer Science, vol. 2211, 2001.
[15] RNTL project ACCORD, http://www.infres.enst.fr/projets/accord.

[16] A Classification and Comparison Framework for Software Architecture Description Languages, N. Medvidovic, R. N. Taylor, IEEE transactions on software engineering, vol. 26, n. 1, 2000.

[17] UML 2.0 Superstructure (Final Adopted specification), Object Management Group, 2003, http://www.omg.org/cgi-bin/doc?ptc/03-08-02

[18] A Connector Model for Object-Oriented Component Integration, Stefan Tai, International Workshop on Component-Based Software Engineering, 1998.

[19] Time-Triggered Real-Time Computing, H. Kopetz, IFAC World Congress, Barcelona, July 2002, IFAC Press.

[20] OSEK/VDX Operating System version 2.2.1, OSEK/VDX, 2003.

[21] The Fractal project, ObjectWeb Consortium, http://fractal.objectweb.org/.