ENHANCING INTERACTION SUPPORT IN THE CORBA COMPONENT MODEL

Sylvain Robert[1], Ansgar Radermacher[1], Vincent Seignole[2], Sébastien Gérard[1], Virginie Watine[2] and François Terrier[1]
[1]CEA-LIST, DRT/LIST/DTSI/SOL, CEA Saclay, 91191 Gif sur Yvette Cedex, France; [2]Thales Alice pilot program, Thales Communications, 91300 Massy, France

Abstract: Even if promising with respect to software complexity management, component-based approaches, like CCM and EJB, have until now fall short in achieving their breakthrough in the real-time and embedded community. Our aim is to adapt one of these approaches - namely the CCM - to the specific needs of this area. In such a process, we have identified several crucial points, among which is interaction management. The current CCM runtime interaction support is actually poor and lacks flexibility. That is the reason why, drawing our inspiration from similar works of the ADLs field, we propose to gather all interaction-related processing in connectors. This paper details the rationale underlying our choice, and outlines all modifications needed to introduce the connector meta-element in the CCM. It also illustrates the relevance of CCM connectors in the scope of a telecommunication use case.

Keywords: CCM, ADL, Components, Connectors, Real Time, Embedded, Interaction

1. INTRODUCTION

The current trend of real time embedded software is towards complexity. Complexity of the developed software, of course, but also complexity of the development processes. Component-based approaches, like CCM[2] or EJB[11], are likely to help developers coping with this issue. However, they have until now been considered to be poorly adapted to embedded software design, notably because of their associated runtime infrastructures which consume

too much resources (e.g. memory footprint). For instance, using the CCM implies using a CORBA-compliant middleware layer, which is generally not affordable in the embedded domain. Hence, a gap has to be bridged, before these approaches can spread in embedded software development practices.

We aim to bridge this gap in the scope of the CCM approach, within ongoing collaborative research projects*. In such a process, we have identified three main steps. First, CCM containers shall be extended with real-time specific support (e.g. scheduling). Then, deployment guidance (i.e. the way the application is installed and launched on the target) has to be adapted. At last, our execution framework should also be usable on systems with limited memory resources. This latter goal is achieved by two measures: (i) choosing a lightweight version of the CCM[1], (ii) making it possible to use light (which means lighter than CORBA) execution infrastructures, like real time operating systems.

The paper focuses on the second aspect, i.e. the integration of new interaction mechanisms that reduce the dependencies to CORBA. As a side effect, these interactions offer more design flexibility. The approach we propose adds a brand new entity in the CCM: the *connector*. While components are loci for business logic, connectors are loci for interaction management logic[8]. They encapsulate all interaction-related processing, thus providing a better separation of concerns. Integrating connectors with CCM is not straightforward, since it imposes a modification of all CCM facilities (e.g. extending the IDL language to be able to define connectors, or adapting deployment and configuration). In the following, we describe our approach to this technical issue and illustrate its relevance an example. The structure of this paper is as follows: Section 2 is an introductive overview of CCM. In Section 3, we describe our view of the CCM connector concept, and show how connectors integrate with the original CCM development process. We also give a short illustration of the introduced concepts. Eventually, section 4 gives a short overview of related work, before concluding.

2. AN OVERVIEW OF THE CCM

The CCM specification[2] covers the whole software development process, from specification to components packaging. It is completed by the OMG[†] Deployment & Configuration specification[5], which provides guidelines for applications configuration and deployment. Describing all CCM facilities

* The IST COMPARE project (http://www.ist-compare.org) and the ITEA MERCED project (http://www.itea-merced.org)

† Object Management Group, http://www.omg.org

would be tedious and not relevant in the context of this paper. Thus, in the following, we sketch only CCM main features: the component model, the execution architecture, and the development process.

In the CCM, components are basically software entities providing services to any of their counterparts. Conversely, they specify the services they need to execute properly. In CCM terminology, a component owns facets (i.e. provided interfaces) and receptacles (required interfaces). On top of that, components may own event sources and sinks, which are the event based equivalents of receptacles and facets. Attributes may also be defined for each component, mostly to enable components configuration at instantiation. Components are defined and declared thanks to a specific language, the Interface Definition Language (IDL). Writing components specifications in IDL is the first step in a CCM application development process. As an example, we provide the declaration of a component "C", providing a port "a_intf" of "IMyIntf" type.

```
component C {
        provides IMyIntf a_intf; // A facet
        consumes E a_E; // An event sink
};
```

The CCM proposes an execution model based on the *component-container* pattern, with the objective to separate business (components) and non-functional (containers) concerns. Containers are the glue between components and the underlying execution / communication platform. They mediate all interactions between components, be they remote or co-located. The container is in particular responsible for providing to the component a *context object*, which can then be used by the component to perform various actions: for instance retrieving a reference to a facet of another component, or publishing an event.

Two main development processes may be differentiated: developing a (single) CCM component and building an application using existing CCM components. The first consists in developing a component corresponding to an interface specification. Fig. 1 shows the successive phases of such a process. Once the interface of the component is written using the IDL, a component skeleton (written in the targeted implementation language) may be obtained using an IDL compiler. This skeleton has then to be completed by the functional (or business) code, and the whole is compiled, in order to obtain a component implementation. This latter step is repeated as many times as necessary: for instance, in order to obtain one Linux, and one Windows implementation. At last, the implementations of the component are packaged together with the component descriptors (XML files describing the contents of the packages) and IDL files.

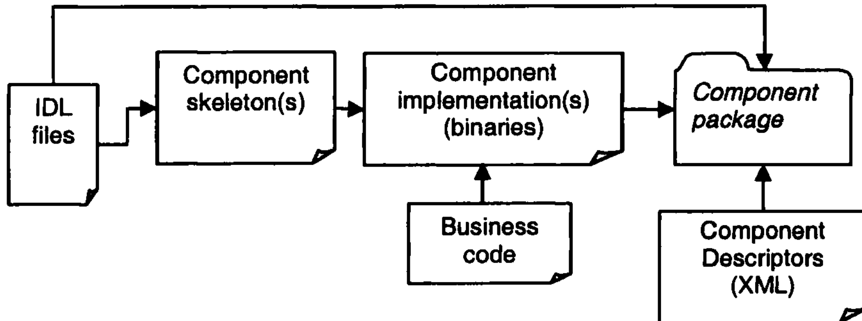

Figure 1. Developing a CCM component

Supposing that all components needed for a given application are identified and the corresponding packages are available, the CCM provides further guidance[5] to properly build, configure, and run the application. First, the application architecture (components instances, connections) have to be defined in dedicated (XML) assembly files. These files are used as an input to a deployment tool, which is then in charge of components instantiation and configuration, also performs the specified connections, and eventually launches the application.

3. CCM CONNECTORS OVERVIEW

In this section, we highlight the interest we have found in defining connectors for CCM and outline the modifications made to CCM in order to integrate this new artifact. The impact of connectors on the CCM development process is also evaluated. In the end, we provide a short illustration of connectors usage.

3.1 Why a CCM connector?

In its native field (Architecture Description Languages[9]), the *connector*[7] is a clearly distinguished entity (from components), dedicated to interaction management. Depending on the work considered, the capabilities of "connectors" may vary a lot. For instance in Unicon[8], the connector is given numerous functional features: a type, an interaction protocol, and other functional features like real-time ones. The elements that may be considered as connectors are also very different from one work to another: sometimes, all kinds of interaction supports are considered (from pipes to scheduler) and even HW artefacts may be parts of the application architecture design[6].

In the scope of the CORBA component model, introducing connectors would have two major impacts: enriching CCM interactions models (CCM originally supports only synchronous method invocation and a specific form of event delivery), and making CCM components interactions *independent from CORBA*. This latter issue is particularly important for embedded systems, since the HW platforms can scarcely afford embedding a CORBA implementation. An additional motivation is that the way interactions are dealt with is part of the application domain's expertise. Thus, building reusable interaction media enables to gather and reuse (i.e. capitalize) software practitioners' knowledge on interaction management. At last, it is obvious that connectors could facilitate component assembly (for instance, two components having provided/required interfaces of different types, but potentially compatible could be linked by means of an "adaptation" connector).

3.2 CCM connectors main features

A CCM connector is an artifact that mediates some interactions between two or more components, while performing intermediary processing. Since components are potentially distributed, it is obvious that connectors are not monolithic, but fragmented. A connector is actually an aggregation of what we call *connector fragments*, each of these fragments being linked to and co-localized with a participant in the interaction. For instance, Fig. 2 shows the deployment view of a connector split into two fragments that are co-located with the components using them. Each fragment has to be bound to the component that will use it. Please note that the connection between a component and a connector fragment is always a *local method invocation*. The remote connection, if necessary, occurs only between the two connector fragments. A major point is that the way the communication between the connector fragments is performed does not forcedly rely on CORBA, i.e. *the communication layer used is specific to the connector under consideration.*

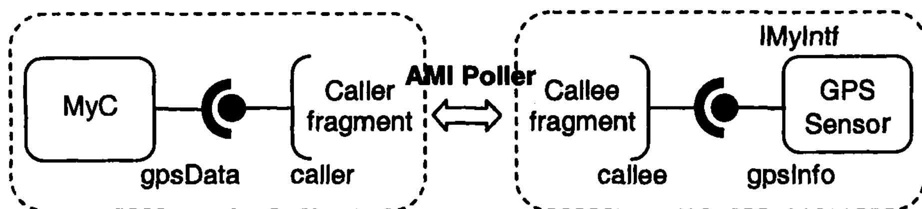

Figure 2. Distributed connector fragments

Connector fragments are connected to the components participating to the interaction. This means that these fragments have to *exhibit CCM ports* that match those of the components. Hence, the abstract model of connector fragments has to be the same as the one of components, basically a combination of (provided and required) ports. However, due to the introduction of connectors, ports will only define the interface in terms of operations and are not responsible for the interaction mechanism that is chosen. Therefore, event-based ports are no longer desirable, for they require using a CORBA middleware event service. Thus, in a CCM design using connectors, components and connector fragments *will only exhibit receptacles and facets*. At last, it is quite obvious that connector fragments will - like components - require configuration means. Hence, we have decided that connector fragments will own attributes as well.

While working on connectors definition, it has rapidly occurred to us that from a methodological point of view, two classes of connectors had to be distinguished. We call those two categories *adaptive* and *fixed* connectors. Adaptive connectors have the particularity that their definition is not fixed, but templated. They ensure a well-defined interaction mechanism (e.g. asynchronous method call), but the interfaces they exhibit are strictly dependent on those of the components they are connected to. Adaptive connectors are interesting in situations where a given interaction mechanism is likely to be used for a variety of interface definitions. For instance, a connector for asynchronous invocations would not be useful, if its port would provide always the same interface: it has to be possible to instantiate the connector and its ports with a certain interface. The advantage of adaptive connectors is that they impose no constraints on components. Fixed connectors have, on the contrary, a fixed set of provided and required interfaces. Therefore, they constrain the definition of the components. Actually, these connectors are very similar to CCM components, excepting their fragmented (and distributed) structures. From a methodological point of view, these two types of connectors are fundamentally different. Adaptive connectors will be preferably used to connect already existing components. On the contrary, using fixed connectors requires an early awareness from the component designer, who will have to define the components in accordance with the connectors he plans to use. Moreover, depending on the class of connectors used, the development processes (and the corresponding support tools) will be different, as shown in the next section.

3.3 A CCM design with connectors

For fixed connectors, the process is the same as for components. Connectors are first described in IDL. For this purpose, we have extended the IDL with the *connector* keyword, and customized accordingly our IDL compiler. Then, based on this specification, several implementations are produced. For instance, one implementation running on top of CORBA, and another that uses Java RMI. These implementations and the IDL are eventually packaged together with descriptor files (an extension has been made on the OMG D&C specification[5] that enables connectors-related features descriptions).

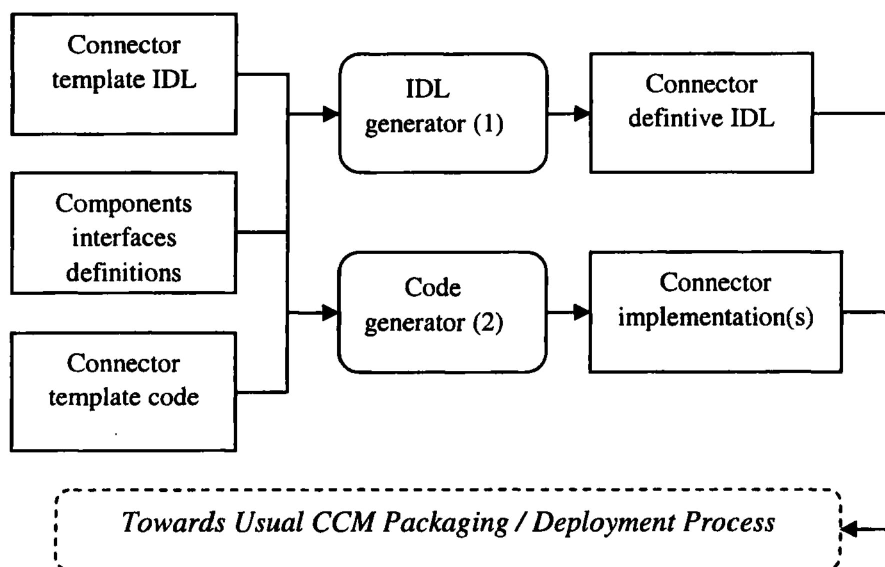

Figure 3. Developing adaptive CCM connectors

The development of adaptive connectors involves more steps. Since these connectors must have the capability to adapt themselves to components interfaces, what shall be provided initially are connector's "raw material", and the associated generation tool. Raw materials are a (template) connector IDL description and one or more (template) connector implementation(s). The generators enable definitive IDL and connector code generation. The corresponding generation process is schematized in Fig. 3. The template declarations are performed by means of IDL extensions we propose. For instance, here follows the template declaration of a connector RemoteMethodCall, with an ISyncInterface template parameter.

```
connector RemoteMethodCall<ISyncInterface> {
        provides ISyncInterface to_caller;
        uses ISyncInterface to_callee;
};
```

In the section dealing with CCM introduction, we indicated that the CCM specification also describes the application building and deployment process. Connectors can be integrated well in this process, since they have basically the same shape as components. It was only necessary to define an extension of the OMG D&C specification[5] to add a connector meta-element in the deployment artifacts. Connectors are packaged strictly the same way as components: an archive (e.g. .zip) is built, which contains connector IDL description, connector implementation(s), and the corresponding descriptor files. However, in the case of adaptive connectors, all connector items have to be obtained *prior to packaging*. With connectors, the deployment process is as follows: (1) Instantiate components and connector fragments on their respective nodes; (2) perform connections between (co-localized) components and connectors fragments; (3) perform connections between remote connector fragments (note that this step depends on the communication mechanisms used by the considered connector implementation); and (4) launch the application. Note that from components point of view, nothing has changed: they obtain a reference for each of their facets without "being aware" that this reference points to a connector fragment.

3.4 Using connectors with CCM: an illustration

In order to test and assess our approach, we are currently carrying out an experimental design. Our application will simulate voice transmission through a simplified UMTS[10] protocol stack. In this scope, we have defined a set of connectors that have significantly eased our design. For instance, we have designed a connector which ensures a "slotted aloha" access protocol. This protocol, which is frequently used in the telecommunication domain, is the following:

- The client initiate the slotted aloha protocol
- A randomly temporized call is performed on the server until the client receives a positive or negative acknowledgement, or the maximum number of calls is reached

Using a "slotted aloha" connector, the process is as follows: (1) the connector is properly configured (e.g. maximum number of calls); (2) the client component initiates the protocol by calling a method of the connector – the call is asynchronous, so the client may then keep on its own

processing; (3) the connector performs the slotted aloha protocol by randomly calling server component, and waits for an acknowledgement. The process is stopped when an acknowledgement is received, or the maximum number of calls is reached; (4) The connector continuously exhibits the acknowledgement status, which can be polled any time by the client component. Using a connector in this case offers several advantages. First, a better separation of concerns, since all the protocol is managed by the connector instead of being ensured by the client component. Then, the connector can be easily reused. For instance, in the UMTS protocol stack, we will use it at two different places, only by modifying connector configuration.

4. RELATED WORK

Besides the foundations in the ADL domain[7], the Qedo (QoS Enabled Distributed Objects) project[12] develops a CCM implementation that adds interaction via streaming, but it does not provide a general framework to add new interaction mechanism. The QuO (Quality Objects) project at BBN[13] adds Quality of service and connectors (as well a connector setup language, CSL) to CORBA. The integration of QuO into the CCM implementation CIAO has been investigated[14] and has led to some aspects also considered by us. However, their intent was more on the resource management and allocation aspects than on fine-grain interaction support. Eventually, we shall precise that even if we closely follow ongoing works on CORBA at the OMG (e.g. minimum CORBA[3] or real-time CORBA[4]), we have no direct relation with these latter, for our focus is strictly on the CCM.

5. CONCLUSIONS AND PERSPECTIVES

We have described in this paper our strategy to improve interaction support in the CORBA Component Model. By analogy with Architecture Description Languages, we have defined a new entity, called "connector", which is dedicated to interaction management. Using CCM connectors enables a better separation between business logic and non-functional logic. Connectors also capitalize interaction management expertise, while reducing design efforts. And, most important in the scope of embedded systems, connectors make CCM-based applications independent from CORBA. We have performed all necessary modifications to the CCM, and are currently assessing our approach on a use-case. The first feedbacks from the design are positive, and we plan to have further usage of connectors. But CCM

enhancement with regards to real-time and embedded systems design (which is the global target of the projects we are involved in) remains an ongoing work. In particular, further real-time support has to be provided by the execution framework (container) to the applicative components. In the next parts of the projects, we plan for instance to evaluate the inclusion of scheduling facilities at framework level.

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. Minimum CORBA specification, version 1.0, formal/02-08-01, Object Management Group, 2002.
4. Real-time CORBA specification, version 1.2, formal/05-01-04, Object Management Group, 2005.
5. Specification for deployment and configuration of component based applications - draft adopted specification, OMG, 2003.
6. Towards a Taxonomy of Software Connectors, N. R. Mehta, N. Medvidovic and S. Phadke, ICSE 2000.
7. Software Connectors and their role in component development, D. Bálek & F. Plášil, DAIS'01.
8. Abstractions for Software Architecture and Tools to support them, M. Shaw, Robert Deline et al., Software Engineering, vol. 21, number 4, 1995.
9. 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.
10. General UMTS Architecture v5.0.1, 3rd Generation Partnership Project, Technical Specification Group Services and System Aspects, 2004.
11. Enterprise JavaBeans Specification version 2.1, Sun Microsystems, 2003.
12. Qedo, QoS Enabled Distributed Objects. http://www.qedo.org
13. Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration, P. Pal et al., Proceedings of ISORC 2000, The 3rd IEEE International Symposium on Object-Oriented Real-time distributed Computing, March 15 - 17, 2000, Newport Beach, CA, http://quo.bbn.com/
14. A Qos-aware CORBA component model for distributed real-time and embedded system development. Nanbor Wang and Chris Gill. OMG Real-time and embedded workshop 2003, Arlington VA. see http://www.omg.org/ workshops/proceedings/