Specification of Communication Components in UML

Eric CARIOU
Irisa, Campus de Beaulieu
35042 RENNES Cedex – France

Antoine BEUGNARD
ENST Bretagne, BP 832
29285 BREST Cedex – France

Abstract: *Reusable software components are being used more and more in application development. Our approach introduces a new type of component: the communication component (or medium). It is a component that encapsulates any kind of communication service or protocol. Other components, that could possibly be distributed, are connected to these mediums and use their communication services. This paper explains how to specify a medium in UML. A medium is defined by a UML collaboration. The constraint language OCL and statecharts are also used to specify the behavior of the services offered by a medium. This specification has two aims: to define the precise behavior of a medium and to use this in a UML CASE tool to validate the medium and automatically generate code.*

Keywords: communication components, UML, UML collaborations, OCL, UML statecharts

1 Introduction

Application development is being based more and more on components. A component is a software element that describes the services it offers and those it needs. Thanks to this knowledge, it is easier to interconnect components to build applications.

Our approach consists in reifying communication in a special kind of component: the communication component (or medium) [2]. A medium is a component that encapsulates any kind of communication service or protocol. For instance, a medium can integrate a broadcast service, a voting service or a consensus protocol. A distributed application is built by interconnecting "classical" components with mediums that manage their communication and distribution.

From a software architecture point of view, using mediums enables the functional concern (what the components do) to be clearly separated from the communication concern (managed by the mediums).

From an analysis point of view, mediums bring a new way of "thinking" about distributed applications, offering the possibility of considering the communication as a structuring element. It allows means of communication to be capitalized on, since mediums are components, and thus, highly reusable.

Previous works [11, 5] have described the interaction between components as important at least as the functional concern. Most of the *Architecture Description Languages* (ADL) [9] contain the notion of *connector*. A connector is a software element that allows components to communicate with each other in accordance with a certain processing. The connectors form the "glueware" coordination between reusable components, *i.e.* the specific part of an application. None of these works consider that some parts of inter-component interaction (modeled by connectors) can be reusable. A medium represents a reusable communication element that can be used in the coordination design of several applications.

On the component model side, Enterprise Java Beans[1] or Corba Component[2] models permit component distribution. But they do not

[1] `http://java.sun.com/products/ejb`
[2] `http://www.omg.org`

integrate the notion of a communication specialized component. Our approach introduces this kind of component.

This paper describes how to formally specify a communication medium. The UML [10] being the new standard in software engineering we have chosen it despite some limitations we have encountered. This specification has two goals. The first is to define a precise contract on the behavior and the processing of a medium and the second is to use this specification in a UML CASE tool to validate the medium and automatically generate code to "map" the specification onto component models (like Enterprise Java Beans, Corba Components, COM+...).

The next section explains how to specify a medium in a general way. Section 3 illustrates a simple but complete example: a medium integrating the Linda coordination model.

2 Medium specification in UML

2.1 What to describe and why

The aim of the specification is to define the precise contract [3] of the processing and behavior of the medium and its services. It details the signatures of the medium services offered (to the components) and those required (on the components), their behavior and the properties of a medium, either global, or local concerning a link with a component.

Depending on their needs, components use some services of the medium, but not all of them. For instance, in a broadcast medium, a component wanting to send information uses only the broadcast service. On the other hand, a component wanting only to receive information uses the receive service. Components are classed depending on the role they play from the medium point of view. With each role is associated a list of services. For instance, the broadcast medium defines a sender and a receiver role.

2.2 Methodology of specification in UML

To describe the contract of the medium, we need to combine three UML "views":

- **a collaboration diagram** for the structural view of a medium. This collaboration is described at the specification level. Collaborations at instance level (the way it is the most commonly used) and sequence diagrams cannot be used because they cannot define a generic system but only a specific instance of a system.

- **OCL constraints** [12, 10] for the specification of medium invariants and the static behavior of a service (specification of pre and postconditions on each service).

- **statecharts** for the specification of dynamic behavior. This view allows temporal constraints, synchronization, locked conditions, etc. to be described.

  Each service can be associated with an interaction (a set of messages) added onto the collaboration diagram, to show the order and nesting of operation calls to realize this service.

The use of OCL "links" all these views formally. We generalize the use of OCL wherever possible. In particular, we use OCL for specifying the guards and conditions of messages in collaborations and in statechart transitions. OCL expressions are defined in a precise context. For a message in a collaboration, the context is the sender of the message. For a transition in a statechart describing a class (or an operation of this class), the context is this class.

2.3 Detail of the structural view

Mediums are represented by UML collaborations. UML collaborations are used to describe interactions between elements. Since we can

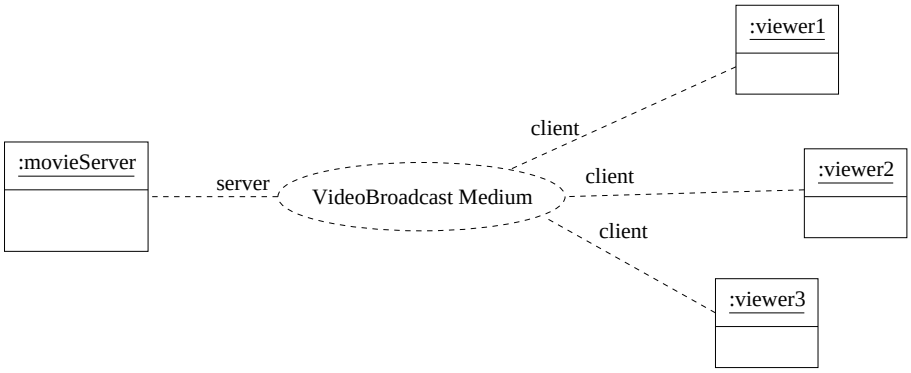

Figure 1: Example of the use of a video stream broadcast medium

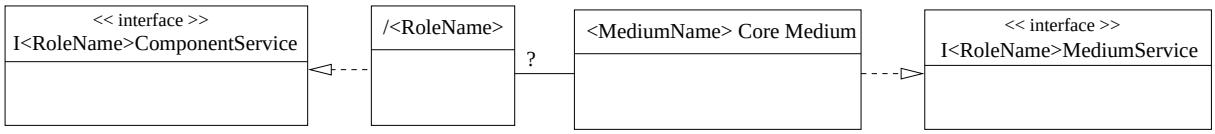

Figure 2: Internal generic description of a medium and a role

consider communication as a kind of interaction, then using collaboration to describe a medium seems to be adapted.

In each collaboration diagram, we retrieve roles played by components connected to the medium. The use of a medium is represented by a class or object diagram in which the collaboration of this medium appears. For instance, figure 1 shows an object diagram in which a medium is used to realize communication between 4 components. The medium called `VideoBroadcast Medium` manages the broadcast of a video stream. One server, the `movieServer` object, (playing the server role in the collaboration) offers the stream to the medium that broadcasts it to the three client components, the `viewerX` objects (all playing the client role). The number of clients is variable and depends of the application.

For each "<RoleName>" role, there are two interfaces:

- I<RoleName>MediumService is the interface of the services offered by the medium to the component playing the role "<RoleName>", *i.e.* the services used by these components.
- I<RoleName>ComponentService is the interface of the services that the component playing the role "<RoleName>" must implement and that are called by the medium.

Interfaces and classes involved in a collaboration describing a medium "<MediumName>" belong to a package named P<MediumName>Medium.

The use of interfaces allows the roles to be typed. A class implementing the PBroadcastMedium::ISenderComponentService interface can be connected to the medium called "Broadcast" and play the Sender role.

Figure 2 shows the classes involved and their relationship for a generic role called <RoleName> and a medium called "<MediumName>". This description is at the collaboration description level, figure 1 being at that of collaboration use (*i.e* in a class or object diagram). The " ? " on the association between /<RoleName> role and <MediumName> Core Medium class is replaced in a "real" description

by the number of components of this role that can be connected to this medium.

Some other classifiers could be associated with the <MediumName> Core Medium class in order to describe the behavior of the medium and its services.

Global properties are attributes of the class <MediumName> Core Medium. Local properties concerning a link component/medium are placed in an association class on the association between the class <MediumName> Core Medium and the appropriate role.

3 Communication medium example

A quite simple but complete example is the description of a medium implementing the Linda coordination model.

3.1 Informal description

The Linda coordination model [8] is based on a shared data space, containing tuples. A tuple is a sequence of fields that contain either a value or a variable. The tuples in the tuple space only contain values. The communication is based on the pattern-matching between a tuple template (a mix between values and variables) and tuples of the space. A template tuple T1 matches a tuple T2 if they have the same number of fields and if T1 contains the same values at the same places as T2. The others fields of T1 are variables.

For instance, consider the following tuple space:

`("12","ab","23","897")`	t_1
`("12","rg","23","567")`	t_2
`("12","ab","23","897","33")`	t_3
`("13","bc","29","432")`	t_4

The `("12",x,"23",y)` template tuple, where `x` and `y` are variables, matches tuples t_1 and t_2, because they have the same number of fields and the same values at the same places (`12` in first position and `23` in third).

The communication is done by adding and removing tuples from the space. Communication services are the same for all the components, so they all play a single role, called Linda (there is no need to distinguish components as in the case for the broadcast medium that classes the components as server or client). A medium manages a single tuple space. The services are the following:

- `void out(Tuple t)`: add the tuple `t` to the space. This operation is atomic.
- `Tuple in(Tuple t)`: return and remove from the space a tuple that fits the `t` template. If more than one tuple fits `t`, the removed tuple is chosen in an indeterministic way. If there is no available tuple fitting `t` when the operation is called, that is blocked until such a tuple is present in the space.

  If we consider the previous example, the calling of `in(("12",x,"23",y))` will return and remove from the space, either t_1 or t_2.
- `Tuple read(Tuple t)`: this behavior is the same as for the `in` operation but only returns a tuple without removing it from the space.

3.2 Collaboration view

Figure 3 describes the collaboration diagram of the Linda medium. The number of components that can connect to the medium is undefined (the "`*`" multiplicity between the `/linda` role and the `Linda Core Medium` class). The `ILindaMediumService` interface contains the service signatures and the `ILindaComponentService` interface is empty because the medium does not need to call services on components connected to it. The `space` association represents the tuple space managed by the medium. A tuple is an instance of the `Tuple` class. The `match` operation returns `true` if the template tuple passed

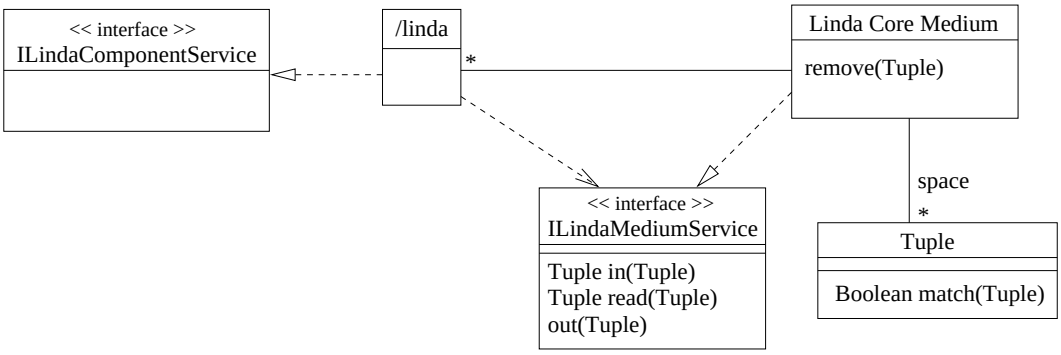

Figure 3: Collaboration diagram of the Linda medium

as a parameter matches the tuple on which this operation is called and `false` otherwise.

3.3 OCL view

Calling the `out` operation adds the tuple passed as a parameter in the space:

```
context LindaCoreMedium::out(t:Tuple)
post:  space = space@pre->including(t)
```

The intuitive way to specify the postcondition of the `in` operation is to say "the returned tuple fits the template and is removed from the space, which notably implies the space becomes the space before the operation without the returned tuple". In OCL, this gives:
`space = space@pre -> excluding(result)`.

Actually, this specification cannot work if the tuple is not in the space when the operation is called. This operation can be blocked for a time long enough to allow some components to add several tuples in the space by `out` operation calls. So, when the blocked `in` operation has finished, `space@pre` references a space that has been modified and if the former OCL expression is validated, that will cancel all the space modifications made during the operation execution and so will not ensure the tuple space coherence.

The solution is to use an atomic tuple removing operation, like the `remove` operation. This operation ensures that the tuple passed as a parameter is in the space when the operation is called, and that once the operation is finished, it has been removed from it.

```
context LindaCoreMedium::
                          remove(t:Tuple)
pre:  space->including(t)
post:  space = space@pre->excluding(t)
```

Now, to specify the `in` operation, we just have to say that "the returned tuple fits the template and it has been removed by a `remove` operation call":

```
context LindaCoreMedium::
                        in(t:Tuple):Tuple
post:
  result.match(t) and
  self.oclCallOperation(remove, result)
```

The *oclCallOperation* is one of our OCL extensions. Here, it means that the `remove` operation has been called with `result` as a parameter on the object `self` during the execution of the `in` operation. (the *oclCallOperation* can be considered as the implementation in OCL of the CallAction defined in UML Action Semantics [1]).

An important issue is to specify that the `remove` operation is atomic. This cannot be done with OCL, but we can consider that if the statechart associated with an operation does not contain blocking or timing constraints, the operation is atomic (this is also true when there is no statechart associated with an operation). However it would be useful to precisely specify this atomicity by adding, for instance,

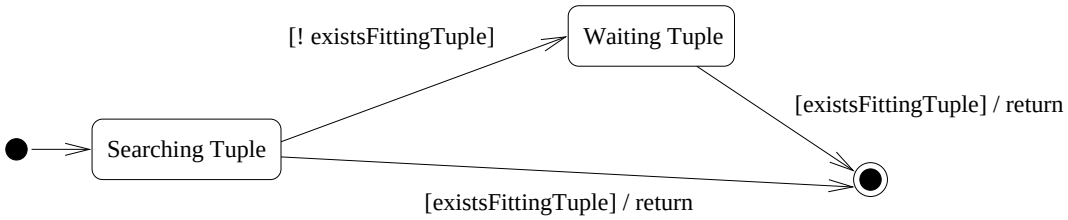

Figure 4: Statechart of the Linda `in(t)` and `read(t)` operations

a constraint like `{atomic}` on the operations.

The `read` operation has the same behavior as the `in` one but without removing the tuple found. So, its OCL specification is quite obvious:

```
context LindaCoreMedium::
                    read(t:Tuple):Tuple
post:  result.match(t)
```

3.4 Statechart view

The OCL specifications are not sufficient to completely specify the behavior of the `in` and `read` operations. In particular, we need to use statecharts associated with each service to describe the blocking conditions of these operations, *i.e.* their dynamic behavior.

Figure 4 represents the statechart associated with the `in` and `read` operations (their dynamic behaviors are the same). At the beginning of the operation, the Searching Tuple state is immediately reached and left, depending on the value of `existsFittingTuple`. This expression is true when (at least) one tuple fitting the tuple passed as parameter (`t`) belongs to the space (at the time of the evaluation of the expression). This expression can be written in OCL, in order to make the link with the OCL operation specifications and ensure coherence between all the views:

```
existsFittingTuple = space->
select(t' | t'.match(t))-> notEmpty
```

where `t` is the parameter passed to the `in` or `read` operation and the OCL context is the LindaCoreMedium class.

So, if there exists a tuple that fits template `t` (guard `[ existsFittingTuple ]`), the operation returns immediately, or else the Waiting Tuple state is reached. It will be left only when the `[ existsFittingTuple ]` guard has become true, *i.e.* when another component has added a tuple by an `out` operation call that fits template `t`.

The statechart view adds dynamic informations to the OCL view that only defined what an operation does (its static behavior).

4 Conclusion and perspectives

Communication mediums are software reusable components integrating any kind of communication services. We have presented how to specify a medium in UML. Then, we have studied a simple medium integrating the Linda coordination model.

Other mediums have already been specified, including: a two-way asynchronous point to point medium (resulting from the composition of two one-way mediums), an event broadcast medium, a voting medium [4] and a broadcast stream medium. During these specifications, we have encountered some expressivity limitations in the definition of synchronization and interaction between components, temporal constraints or OCL specification of non-atomic operations. The generalized use of OCL and some small extensions make it possible to formally link the static and behavioral parts of a specification, and thus, specify a medium in this entirety.

An interactive video application has been already developed with a stream broadcast

medium and a voting medium [4]. A video-conference application has also been developed, using the same two mediums, illustrating the reusability of mediums.

Our current work continues with medium specification in UML. We are also developing some medium prototypes in order to elaborate a medium catalogue. We expect to retro-analyze some real applications putting the communication at the center of the analysis process.

The medium specification methodology will be integrated in a UML CASE tool, like Umlaut[3]. This tool is dedicated to the manipulation of UML models at the semantic level [6] and will then make it possible to validate the design and behavior of a medium [7] (*e.g.* by model checking based techniques or simulation), and automatically generate the code associated with a medium, depending on the component model target choice.

References

[1] Action Semantics Consortium. Uml action semantics, proposal version 9.23, 3 March 2000. `http://people.ce.mediaone.net-/weigert/actionsemantics/home.html`, 2000.

[2] Antoine Beugnard. Un Modèle Architectural à Base de Composants pour les Applications Distribuéés. In Hermes, editor, *LMO'2000*, 2000. *(in french).*

[3] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, pages 38–45, July 1999.

[4] Eric Cariou. Spécification de Composants de Communication en UML. In *OCM'2000*, May 2000. *(in french).*

[5] Chrysanthos Dellarocas. Software Component Interconnection Should Be Treated as a Distinct Design Problem. `http://www.-umcs.maine.edu/~ftp/wisr/wisr8/-papers/dellarocas/dellarocas.html`, 1996.

[6] Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In *Proc. Automated Software Engineering, ASE'99, Florida*, October 1999.

[7] Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. Validating distributed software modelled with UML. In *Proc. Int. Workshop UML98, Mulhouse, France*, June 1998.

[8] Thilo Kielmann. Designing a Coordination Model for Open Systems. In Springer Verlag, editor, *Coordination Languages and Models*, Lecture Notes in Computer Science 1061, 1996.

[9] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, 1997. `http://www.ics.-uci.edu/pub/arch/sw-and-pubs.html`.

[10] OMG. Unified Modeling Language Specification, version 1. 3. `http://www.omg.org`, 1999.

[11] Mary Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In D.A. Lamb, editor, *Studies of Software Design, Proceedings of a 1993 Workshop*. Lecture Notes in Computer Science 1078, Springer-Verlag, pp. 17-32, 1996. `http://www.cs.cmu.edu/~Vit/-paper_abstracts/FirstClConnTR.html`.

[12] Jos Warmer and Anneke Kleppe. *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley, 1998.

[3] `http://www.irisa.fr/PAMPA/umlaut`