The Specification of UML Collaborations as Interaction Components

Eric Cariou and Antoine Beugnard

ENST Bretagne, BP 832, 29285 BREST CEDEX, FRANCE
{Eric.Cariou,Antoine.Beugnard}@enst-bretagne.fr

Abstract. One of the touchstones of Object-Oriented Design is that the management of complexity is seldom located within any single object. It should instead be an emerging property of the collaborations within a society of objects, each one of these being as simple as possible. These collaborations can easily be specified using UML collaboration diagrams. We propose to reify UML collaborations as *interaction components.* This allows the easy handling and reusing of *interaction abstractions* among components at both specification and implementation levels.
This paper focuses on the specification of these components. We propose criteria to define the type and the "frontier" of an interaction abstraction. We present a UML collaboration specification methodology that deals with the constraints of component specification.
Keywords: UML collaborations, specification methodology, interaction abstraction, interaction components

1 Introduction

The essence of the object-oriented paradigm is the modeling of interesting phenomena as a structure of interacting objects [2]. The management of complexity should not be located within any single object. It should instead be an emerging property of the collaborations within a society of objects, each one of these being as simple as possible. The interest of this notion of collaboration has long been recognized in the object-oriented community, and some methodologies of the early nineties (such as CRC [18] or OOram [14]) even concentrated on collaborations as the basic building blocks to carry out object-oriented design. Collaboration diagrams are now well established as a core component of UML [13]. They allow the easy handling and reusing of *interaction abstractions* among components.

Very often, though, these interaction abstractions get lost during the detailed design process, making it difficult to keep good traceability between the design and the implementation. At implementation level very few traces of these abstractions are still visible: collaborations have been refined, split and lost in a set of objects that can be distributed over a network and communicate through "low level" primitives such as remote procedure calls. Given that, at the implementation level, the communication principles used are basic, the designer will perhaps unconsciously not search to build complex or high level collaborations at design time.

In this paper we propose to reify UML collaborations into *interaction components*, in such a way that any given collaboration can be thought of as both a design level component *and* an implementation level component (i.e. at the level of EJB, .NET or CCM). These components are designed to be as complex as required at the implementation level, allowing the designer to handle high-level interaction abstractions throughout the software process. So there are no more "restrictions" for the design of high-level collaborations that can have very interesting properties such as reusability and substitutability.

The rest of the paper is organized as follows. In section 2, we study a car park access control system and show that if the collaboration used is well designed, it leads to several advantages such as reusability. In section 3 we describe how we can design a collaboration as a reusable component, using UML and OCL [17] to precisely specify its properties. We also give guidelines for finding the frontier and the responsibility of a collaboration by defining its component type. Then we discuss related works in section 4, before concluding with some interesting perspectives for interaction components.

2 A car park management application

In this section, we briefly discuss a car park management system. We focus on the UML collaboration used in the application specification and study a better way to define it.

2.1 Specification of the car park application

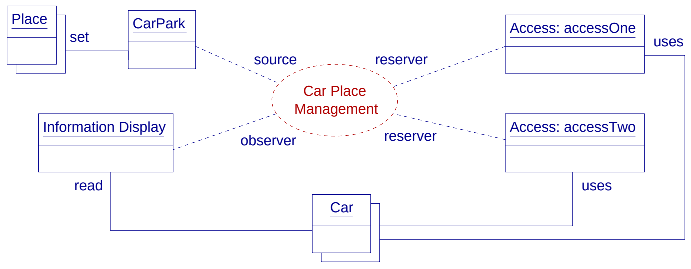


Fig. 1. Car park management application

The car park is composed of a set of parking places. Each place is referenced by a unique identifier (e.g. a number). A car in the car park occupies one identified place. A car can enter and leave the car park using one of two accesses. The place occupied by a car is assigned to it when it enters the car park. Outside the car park, an information monitor displays the number of available places.

Figure 1 is an UML instance diagram representing this application. We can see on this figure the car park, the two accesses and the display that are the four components forming the car park management system. The car park component manages the set of identified places. The cars are objects using this system.

These four components interact among themselves in the following way:

– When a car wants to enter the car park, the access door sends a query to the car park component to get a place identifier. If the car park is full, a special value is returned.
– When a car leaves the car park, the access door must inform the car park component that the place occupied by the car is available again.
– When a car enters or leaves through an access door, the number of available places has to be refreshed accordingly on the display.

So the four components interact through the “Car Place Management” collaboration. In this collaboration, an access plays a `reserver` role because it can reserve (and cancel) a place in the car park for a car. The display plays the `observer` role because it keeps informing on the number of available places. The car park component plays the `source` role because it represents the car park itself and manages the place set.

2.2 What is the complexity of the collaboration ?

The answer to this question depends on what the car park component is and on how the collaboration has been designed.

A first specification. In our application specification, the car park component has the responsibility of managing the set of places. It has to maintain the list of available places in the car park. The collaboration[1] is very simple (see figure 2). It just describes the method calls and their nesting between the roles.

If a reserver wants to reserve a place, it calls the `newCarEntering` method on the source role and gets a place identifier (or a special value if no place is available in the car park). To cancel the reservation of a place by a car, it calls the `carLeaving` method on the source manager with the place identifier as parameter.

Each time one of these two methods is called, the number of available places changes and the observer roles must be informed of this new number: the `nb-AvailablePlaces` method is called on the observer roles.

For more precision, each method can be specified using OCL pre and post-conditions.

[1] Collaborations are generally described at instance level. In this paper, all the collaborations are described at specification level (see [13, page 3-109]) because we want to specify generic interactions and not application dedicated ones.

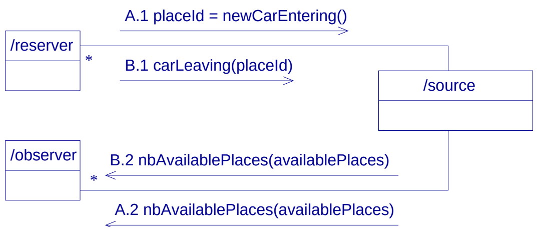

A.1 placeId = newCarEntering()
/reserver
*
B.1 carLeaving(placeId)
/source
/observer
B.2 nbAvailablePlaces(availablePlaces)
*
A.2 nbAvailablePlaces(availablePlaces)

Fig. 2. First specification of the collaboration

A more interesting specification. This interaction pattern is not specific to our car park. It can be used (or more precisely *reused*) in several contexts.

For instance, if the data to reserve are flight seats instead of car places and with minor changes like method renaming, the collaboration can be used in a context of a flight seat reservation system. An airline that wants to sell places for a given flight then plays the source role. The reserver and observer roles are travel agencies that reserve seats on this flight for their clients.

The context of this application is completely different but the structure of the interaction is the same as in the car park management application. The collaboration used will be a slightly-modified version of the former collaboration.

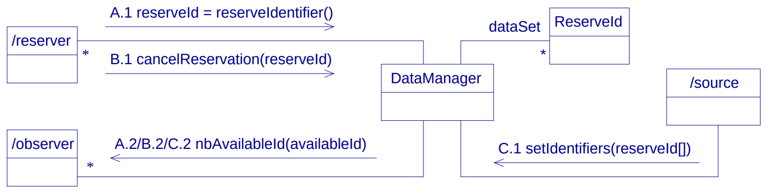
In this flight seat reservation application, the airline component must manage the list of available seats on the flight. This can be done exactly in the same way as for the car park component with its car park places (excepting for the type of data to manipulate: seats on a flight instead of places in a car park but that is a minor change). We can easily imagine that the design of this system will be exactly the same in these two contexts. So, reusing the structure of the collaboration implies reusing the set management system associated with the component that is playing the `source` role. The designer has to keep this in mind if he wants to make a good reuse of the collaboration structure (i.e. if he does not want to design the set management system from scratch).

This observation being made, we can re-design our collaboration. As the set management system is closely correlated to the collaboration, why not put this system *inside* the collaboration and make it work with any kind of data?

The reuse of the collaboration will also lead to the reuse of this system. The component playing the `source` role will not have the responsibility of managing the data set anymore.

This gives us a new collaboration (see figure 3). The main difference with the former is that this collaboration does "more". It is also more complex because it has a state and manages a collection of data, by the way of the `DataManager` component. A data is an instance of the `ReserveId` class. In a given context, in order to manipulate a specific type of data, a sub-class of `ReserveId` class would be defined. The collaboration can then be used independently of the data type needed in a particular application.

The two applications are now described by the following figures:



A.1 reserveId = reserveIdentifier()
/reserver
*
B.1 cancelReservation(reserveId)
dataSet
ReserveId
*
DataManager
/source
/observer
A.2/B.2/C.2 nbAvailableId(availableId)
*
C.1 setIdentifiers(reserveId[])

Fig. 3. New collaboration design

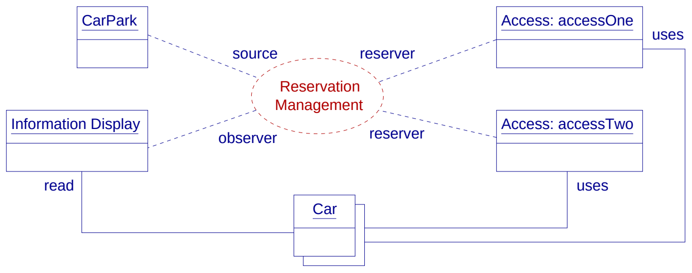


Fig. 4. New car park management application

– Figure 4 for the car park management application
– Figure 5 for the flight seat reservation application

In these two contexts, the collaborations used are exactly the same. They have been directly reused without any modification. The two source roles (`Car Park` and `ErnestAir` components) just have to specify the data set to use (by calling the `setIdentifiers` service on the `DataManager` component) and not manage it themselves.

Comparison of the collaboration designs. The reuse of the first collaboration is almost useless. Indeed, it just describes the call of methods between the components and this interaction is too simple and specific to belong to an interaction catalogue, unlike the second collaboration specification that reuses more elements. In this case, the interaction embeds not only the method calls but also the data management. The interaction is "self-content" and specified independently from any context of use. This makes it easily usable in any application design.

Having such a "complex" collaboration gives us major improvement in software specification. Firstly, we have seen that the collaboration is more reusable.

Secondly, it allows the use of abstraction[2] of interaction in a high-level design (in a class or instance diagram where the collaboration is used).

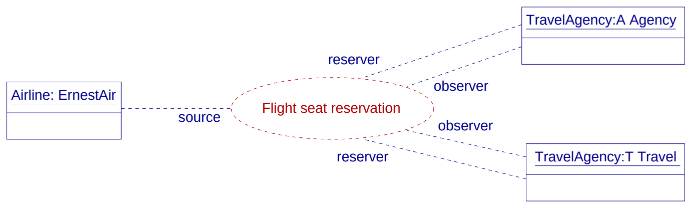


Fig. 5. Flight seat reservation application

3 Design of collaboration as component

We have seen that the specification and the use of high-level and complex communication or interaction abstractions can be easily done, notably in UML with the collaboration diagrams. Moreover, in a class or instance diagram, the use of a collaboration allows the manipulation of a communication abstraction as a single entity; this communication abstraction being precisely defined in its collaboration diagram. But at the implementation level, these complex interactions do not exist as "single manipulable entities" anymore, they have been refined, split and lost in a set of objects that can be distributed over a network and communicate through "low level" primitives such as remote procedure calls.

In order to solve this problem, we propose to manipulate an interaction between components as a component, i.e. to implement a UML collaboration into a software component. This permits the use and the manipulation of the same communication abstraction during the entire software development: from the design to the code.

These components are a bit special because they do not concern the functional part of an application but the interaction, the communication part. So, to differentiate these "interaction" components from the "classical functional" components, we call them *mediums*. An application is the result of the combination of (functional) components and mediums[3, 5].

The protocols, the communication services or the interaction systems implemented in mediums are various in type and complexity: event broadcast, consensus protocol, coordination through a shared memory, multimedia stream broadcast, vote system...

[2] Abstraction is used in the sense that "the details are hidden" and not "fuzzy". It is also precise because the interaction is completely defined in its collaboration diagram.

3.1 Medium specification methodology

As the collaboration has to be refined into a software component, its specification must follow some particular rules.

Although the paradigm of components is now widely accepted by the software community, there is no real consensus on the definition of a component. However, we can summarize its principle properties [16, 8]:

- It is an autonomous and deployable software entity.
- It clearly specifies the services it offers and those it requires. This allows the use of a component without knowing how it works (by looking at the code for instance).
- It can be combined with other components.

The design of a collaboration must deal with the above characteristics: the interfaces of the services offered and required must be present. A special class inside the collaboration represents the medium as a whole.

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 only uses a broadcast service. On the other hand, a component wanting to receive information uses a receive service. Components are differentiated depending on the role they play from the medium point of view. With each role is associated a list of services. For example, a broadcast medium defines a sender and a receiver role. These roles match the roles used in UML collaborations. Since a medium specification is a collaboration, components connecting to a medium play a given role in this collaboration.

As for any classical components, mediums require a good specification in order to make them easily usable. A “good” specification includes all information that describes how to use the component, but also what it does. This could be encapsulated in a contract as we propose in [4]. This contract must include the required and offered services signatures, but can not be limited to these. The semantics and the dynamic behavior of services must also be specified.

The specification in UML of a medium is made by combining three “views” :

- **a collaboration diagram** for the structural view of a medium. This collaboration contains all the roles that can be played by the components, a class representing the medium (called `MediumName Medium`[3]) and all the elements necessary to describe the behavior of the medium and of its services.
  For each role, the class representing the medium in the collaboration implements an interface (called `IRoleNameMediumServices`) containing the services offered to the components playing this role. This interface is the type of the role. A “RoleName” role implements an interface (called `IRoleName-ComponentServices`) providing the services the medium would call back.

[3] We have chosen to use naming conventions for classifiers involved in medium specifications, but of course we could have used UML stereotypes instead. For instance, the class representing the medium could have been stereotyped by `<< medium >>`.

Messages representing interactions resulting from a service call can be added onto the collaboration diagram. An interaction can be associated with each service.
- **OCL constraints** [17] for specification of the medium invariants and the static behavior of a service (specification of pre and postconditions on each service).
- **statecharts** for specification of dynamic behavior. This view allows temporal constraints, synchronization, locked conditions, etc. to be described.

The use of OCL links all these views formally. We generalize the use of OCL everywhere 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 associated with a class (or with an operation of this class), the context is this class.

Of course, other UML diagrams such as sequence or activity diagrams can also be used to specify a medium in addition to those described above. Actually, all the UML features can be used. But we believe that these three views are sufficient in most of the cases.

3.2 Example: specification of the reservation medium

The reservation medium is the medium we have used in the context of the car park and the airline applications. The following specification details the structure of the collaboration according to the specification principles we have given above. Other medium specifications can be found in [5].

Informal description. The reservation medium manages a set of identifiers (seats in a plane, car park places...) that can be reserved by a group of components. Some components can observe the state of the medium: after any reservation or cancellation of a reservation, they are informed on the new number of available identifiers.

The roles of components and their associated services are the following:

source role

this single component owns the set of the reservation identifiers and calls the following service on the medium to initialize it :
- `void setReserveIdSet(ReserveId setId[], Boolean cancelerIsReserver)`: `setId` is the set of the reservation identifiers. `cancelerIsReserver` indicates if the component that cancels a reservation must be the one that has made the reservation (`true` value) or can be any component (`false` value).

reserver role

the components playing this role can make reservations and cancel them by calling the following services :

- `ReserveId reserve()`: return one available reservation identifier of the set (and remove it from the set). Return `null` if there is no more available identifier.
- `Boolean cancel(ReserveId)`: cancel a reservation whose identifier is passed as parameter. Return `false` if the three following conditions are not verified:
  1. The identifier belongs to the original set.
  2. The identifier does not belong to the available set (i.e. it is reserved at the present time).
  3. If `cancelerIsReserver` is true, the component that does the cancellation is the one that has made the reservation.

  Return `true` if these conditions are verified and the identifier is again available to the reservation.

observer role

these components have to implement the following service, in order to be informed of the change of the available identifier number:

- `void nbAvailableId(Integer newValue)` : the parameter indicates the new number of available identifiers.

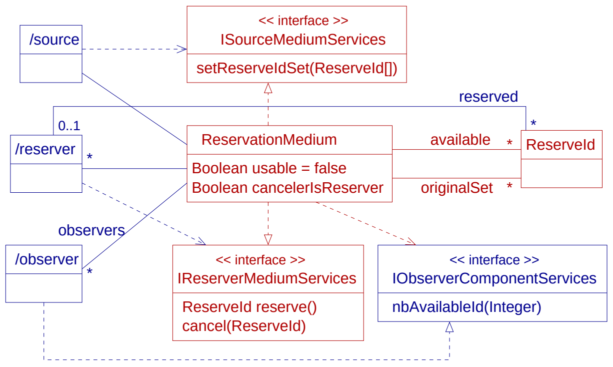


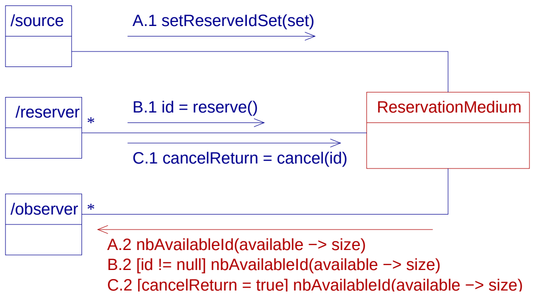
Fig. 6. Collaboration describing the reservation medium

Collaboration diagram. The structural diagram of the collaboration is figure 6. Its design follows the specification rules described in section 3.1.

The `ReservationMedium` class represents the medium as a whole. It implements the two interfaces of offered services. The components playing the `source` and `reserver` roles are each one dependent on one of these interfaces, i.e. they are using their services. The `observer` role must implement the `IObserverComponentServices` in order to be informed of the change of the number of available identifiers.

The `ReservationMedium` class manages the set of available reservation identifiers (link `available`) and keeps a reference on the original set (link `originalSet`). The `reserved` link allows the medium to know if an identifier is reserved or not and by which `reserver` role.

The multiplicity of the links between a role and the `Reservation Medium` class on the role side allows the number of role instantiations to be specified. Here, one and only one `source` role must be present. The number of `reserver` and `observer` roles is unspecified.



/source
A.1 setReserveIdSet(set)
/reserver
*
B.1 id = reserve()
ReservationMedium
C.1 cancelReturn = cancel(id)
/observer
*
A.2 nbAvailableId(available −> size)
B.2 [id != null] nbAvailableId(available −> size)
C.2 [cancelReturn = true] nbAvailableId(available −> size)

Fig. 7. Dynamical view of the reservation medium

Figure 7 is the dynamic view of the collaboration including messages sent among the roles. It shows the relations between the services: each time the set is modified, the `nbAvailableId` method is called on the observer components. We use OCL-based expressions for the parameter of this method and the guards of the messages. The context of these OCL expressions is the sender of the message, here the `MediumReservation` class. For example, the parameter of this method is the result of the OCL expression `avalaible -> size` that returns the size of the set `available`.

OCL specifications. OCL is used to specify the static semantics of the medium offered services and medium properties.

The `setReserveIdSet` service is used for the medium initialization. Figure 8 gives its OCL specification. The goal of this service is to initialize the medium. The call of this service can be done only once and before this call, the medium is considered as unusable, i.e. the medium services can not be called. The precondition then checks the `usable` property that must be false. After the call, it is set to `true`. In the OCL specification of the reserver services, the precondition will check that this property is verified.

The postcondition specifies that the original and available identifier set (respectively the `originalSet` and `available` reference) are equal to the set passed as parameter. The set of the reserved identifiers must be empty for each reserver manager.

OCL specifications are also defined for the reserver services: the `reserve` and `cancel` methods (they are not described in this paper[4]). They ensure that the semantics (described in section 3.2) of a reservation and its cancelling are respected. Medium invariants insuring the set consistency and describing the relations between the `reserved`, `originalSet` and `available` sets will have to be written in OCL.

```
context ReservationMedium::setReserveIdSet(Set idSet, Boolean cancel)
pre: usable = false
post: originalSet = idSet
    and available = idSet
    and usable = true
    and cancelerIsReserver = cancel
    and reserver -> forAll( r | r.reserved -> isEmpty )
```

Fig. 8. OCL specification of the `setReserveIdSet` service

Statecharts. No statechart is needed in the specification of the reservation medium.

3.3 Finding the medium frontier and services

The first section of this paper shows that a collaboration is more reusable if it is "self-contained" and not "too simple". This is a very intuitive way to define the frontier of a collaboration. The frontier allows us to know what the responsibility of the collaboration is (data to handle, services to implement, etc.) and what is external to it. The definition of the frontier of a medium or a classical component is a key point.

Some component-based application specification processes based on UML such as UML Components [7] or Catalysis [8] propose to define a component's frontier and its responsibility by finding its *types*. A type specification is the definition of a set of services (grouped in an interface) and their semantics. All features and attributes that are necessary for specifying the services must be added in a static diagram associated with the type. These features are for instance the data manipulated by the services. OCL constraints are also used in these specifications. A component implements one or several types. These types are defined at an abstract level without any assumptions about implementation; they describe the usage contract of a component.

We propose to adapt this type definition to the context of mediums. As a medium is a component, it must also specify its usage contract and thus

[4] For a complete specification of the reservation medium, the reader is invited to consult our web site: `http://www-info.enst-bretagne.fr/medium/`

its type(s). The abstract specification of a medium is a UML collaboration. Therefore finding the type of a medium allows us to define the responsibility and the frontier of the collaboration at the highest level of specification.

In the case of our reservation applications, the collaborations are used to reserve identifiers. A reserver component needs for instance to call a `reserve` service. This service belongs to the interface of the type. The service will return an identifier that was available. So the available identifier list is mandatory for specifying the service and it belongs to the type specification. Thus, this is the "correct" justification for placing the identifier set in the collaboration (see figure 6) as a service offered manipulates this set. The ReservationMedium has references on the set of identifiers because they are mandatory for specifying the `reserve` service behavior.

This process invites us to place the identifier set inside the medium, but we could have chosen to delegate its management to an external component (as it is done in figure 1. This latter solution has to be rejected since (1) the collaboration abstraction loses its coherence and becomes almost useless and (2) it constrains a design or implementation choice with a premature decision.

In our collaboration, components that play a reserver role need to reserve and cancel identifiers. So the `reserve` and `cancel` services have to be implemented by the medium. The observer components are kept informed on the number of available identifiers. In order to do so, they need to implement the `nbAvailabelId` service. By looking at the need of these two kinds of components, we found two interfaces of services (the `IReserverMediumServices` and `IObserverComponentServices`).

As the collaboration or the medium handles the identifier set, a service must be added for the source component to specify the set to use. This is the `setReserveIdSet` service of the `ISourceMediumServices`. This last service was discovered after a first iteration of the process.

We can summerarize the guidelines for finding the medium frontier and responsibility as follows:

- Find the services offered and required by the medium.
- These services are used to define offered and required interfaces associated with each role.
- Define the service semantics with OCL constraints or other UML features. This implies adding all the necessary classifiers and associations to the collaboration diagram.
- Reiterate the first three steps to find new services if needed.
- No implementation assumptions must be made during these specification steps.

Once the medium type is specified through its collaboration diagram, the interaction frontier and responsibility are completely defined. This is done at an abstract level, independently of any implementation constraints or choices.

4 Related Works

The specification of collaboration or interaction abstractions has been studied extensively. Our methodology is not revolutionary but is simply a new vision of these interaction abstraction specifications in the context of software components. Moreover, an important aspect of our methodology is that it is included in a whole process. The goal of this paper has been only to speak about specification, but readers interested in other parts of the process are invited to read [6] or to consult our web site[5]. In particular, this process is composed of a specification refinement process allowing a high-level medium specification to be transformed into lower level ones according to implementation and deployment constraints. We have also defined an internal medium architecture at the implementation and deployment level and built a framework to implement them.

Some following related works will not only discuss the specification level but also include some other parts of the whole process.

4.1 Architecture Description Languages

A wide field of software engineering concerns software architecture and languages to describe it [1, 12]. Research goals are to improve maintainability, evolutivity, and reusability. The ACME [9] language, for instance, enables us to describe a system as an assembly of components and connectors. Connectors reify interactions among components; they mediate the communication and coordination activities among them.

The main difference between the ADL approach and ours is that event if ADL connectors are, in principle, as abstract as possible, ADLs eventually propose a small set of connectors that are usually very close to existing "low-level" communication services, such as RPC, Unix pipes, SQL-link, http-link, etc.

We consider that the interaction specification has to be used at architectural level with, if possible, no hypothesis on the implementation. An implementation variant resulting from the refinement process also has to be selected depending on the application context (performance, size, middleware, systems, etc.).

4.2 Coordination components and languages

In order to improve the separation of the objects' essence and their interaction requirements, some authors propose object connectors [10, 15]. Connectors contain the required glue to make objects interact. This approach, like ours, reifies interactions which are usually described with a coordination language. In some implementations the refinement process consists of a compilation, which is more abstract, but prevents implementation variants; connectors are dedicated to a specific application making them potentially less reusable than if they were developed from *standard* interactions such as we propose. Another approach [11]

[5] `http://www-info.enst-bretagne.fr/medium/`

uses collaboration contracts which is more flexible since collaboration rules can be dynamically updated.

Most coordination models rely on an explicit interface of components being coordinated [11], but some use introspection and metaprogramming features to coordinate components that hide their interface [10]. However, we consider the coordination problem as a subproblem of communication, limited to message-passing models[6]. Until now, we have not specified a medium dedicated to collaboration but we imagine it would be possible; a collaboration medium would be programmed using a coordination language and would observe events on its entries and trigger events as specified.

4.3 Catalysis

Catalysis [8] is a methodology that is component centered. It uses a notation based on UML to describe models, components and implementations. The concept of collaboration is central in this approach. A collaboration is a collection of actions and the types of objects that participate in them. Moreover, Catalysis defines the notion of collaboration frameworks that are kinds of generic collaboration. Catalysis collaboration frameworks resemble our medium specifications since a catalysis connector is specified by a collaboration framework (a medium being a kind of connector). But Catalysis proposes to define only a small set of connectors that can be used during the implementation. We believe that communication abstractions between distributed components cannot be limited to a small set of low-level interaction patterns, even at the implementation level. We argue that every communication abstraction can be manipulated as a connector or a medium, independently of its complexity.

Finally, although Catalysis offers a methodology to refine its models and to keep track of the successive refinement steps and, although collaborations are proposed to be refined in connectors, Catalysis proposes no real process or implementation target as we do. We imagine the work presented here as being an extension of Catalysis methodology.

5 Conclusion

We have presented an approach to reify high-level interaction abstractions into full-blown software components: interaction components or mediums. At the specification level, a medium is specified by a UML collaboration augmented with OCL constraints and any necessary UML diagrams. Since a medium is a component, the specification methodology we have described follows rules specific to the component context.

A very important issue is, like for any component, to define the medium frontier, i.e. to specify the services it offers and what its responsibility is. We have given guidelines to find this frontier by defining the medium abstract type.

[6] Even Linda which emulates asynchronous message-passing through a shared memory.

This paper has only focused on medium specification methodology but this methodology is involved in a more global process allowing the manipulation and the reutilization of high-level interaction abstractions throughout the software process, from analysis to implementation and deployment. In particular, we have worked on a refinement process that transforms an abstract UML medium specification into an implementation specification according to some implementation and deployment constraints. This allows several implementations of the same interaction abstraction to be made and the designer to choose the right one according to an application context. An interactive video application using two mediums (one for the multimedia stream broadcast and one for the voting system) has also been implemented (thanks to our implementation medium framework) and has shown the benefits of having complex interactions at the implementation level.

Given that it is now possible to easily instantiate high-level interaction abstractions, we believe that this will lead designers to definitively consider and use collaborations as first-class architectural entities at both design and implementation time.

References

1. R. J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, 1997.
2. E. P. Andersen and T. Reenskaug. System design by composing structures of interacting objects. In O. L. Madsen, editor, *ECOOP'92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 133–152, New York, NY, 1992. Springer-Verlag.
3. A. Beugnard. Communication services as components for telecommunication applications. In *14th European Conference on Object-Oriented Programming (ECOOP'2000), Objects and Patterns in Telecom Workshop, Sophia Antipolis and Cannes (France)*, 2000.
4. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *Computer*, pages 38–45, July 1999.
5. E. Cariou and A. Beugnard. Specification of Communication Components in UML. In H. Arabnia, editor, *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '2000)*, volume 2, pages 785–792. CSREA Press, June 2000.
6. E. Cariou, A. Beugnard, and J.-M. Jézéquel. An architecture and a process for implementing distributed collaborations. In *The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2002)*, September 2002.
7. J. Cheesman and J. Daniels. *UML Components - A Simple Process for Specifying Component-Based Software.* Addison-Wesley, 2000.
8. D. D'Souza and A. Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach.* Addison-Wesley, 1998.
9. D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.
10. M. Günter. Explicit connectors for coordination of active objects. Master's thesis, University of Berne, 1998.

11. L.Andrade, J.Fiadeiro, J.Gouveia, A.Lopes, and M.Wermelinger. Patterns for coordination. In G.Catalin-Roman and A.Porto, editors, *Coordination Languages and Models*, pages 317–322. LNCS 1906, Springer-Verlag, 2000.
12. N. Medvidovic and R. 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.
13. OMG. Unified Modeling Language Specification, version 1. 3. `http://www.omg.org`, June 1999.
14. T. Reenskaug. *Working with Objects*. Manning/Prentice Hall, 1996.
15. M. Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In D. Lamb, editor, *Studies of Software Design, Proceedings of a 1993 Workshop*. Lecture Notes in Computer Science 1078, Springer-Verlag, pp. 17-32, 1996.
16. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, N.Y., 1998.
17. J. Warmer and A. Kleppe. *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley, 1998.
18. R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.