Spécification de Composants de Communication en UML

Eric CARIOU

ENST Bretagne – Irisa

ENST Bretagne
BP 832, 29285 Brest Cedex
Eric.Cariou@enst-bretagne.fr

Résumé : Les composants logiciels réutilisables sont de plus en plus utilisés dans la construction d'applications. Notre approche introduit un type de composant particulier : le médium de communication. Il s'agit d'un composant implémentant un service ou un protocole de communication de type quelconque. Les autres composants de l'application, qui seront éventuellement distribués, utiliseront ces services. Cet article explique comment et pourquoi décrire formellement un médium de communication à l'aide d'UML. Un médium est défini par une collaboration UML. Le langage de contrainte OCL ainsi que les diagrammes d'états d'UML permettent de spécifier le comportement des services de communication offerts par un médium. Cette spécification a deux buts : permettre au composant utilisant le médium de savoir exactement ce que fait le médium et de générer automatiquement du code vers un modèle de composant existant (Entreprise JavaBeans, composants CORBA…).

Mots-clés : UML, collaborations UML, OCL, composants de communication

1 Introduction

Le développement d'applications s'appuie aujourd'hui de plus en plus sur le paradigme de composant. Un composant est un élément logiciel décrivant clairement les services qu'il offre et ceux qu'il requiert (chez d'autres composants). Ainsi, grâce à ces connaissances, il devient plus aisé d'interconnecter ces composants entre eux afin de construire une application.

Notre approche consiste à réifier la notion de communication dans un composant particulier : le composant (ou médium) de communication [Beu00]. Un médium intègre un protocole ou un service de communication quelconque. Par exemple, un médium pourra implémenter un service de diffusion, un service de vote, un protocole de consensus… Une application distribuée est construite en interconnectant des composants « classiques » par l'intermédiaire de médiums qui gèrent leur communication ainsi que leur distribution.

En terme d'architecture d'application, cela permet de bien découpler l'aspect fonctionnel (ce que font les composants) de l'aspect communication (gérée uniquement par les médiums).

En terme d'analyse, les médiums apportent une nouvelle façon de « penser » les applications distribuées en offrant la possibilité de considérer la communication comme un élément structurant. Et en permettant de capitaliser les moyens de communication car les médiums sont des composants, et à ce titre, sont fortement réutilisables.

Des travaux précédents [Sha94, Del96] ont décrit la notion d'interaction entre composants comme étant aussi importante que l'aspect fonctionnel. La plupart des langages de description d'architecture [MT97] (les ADL : *Architecture Description Languages*) possèdent la notion de *Connector* qui est un élément permettant à des composants de communiquer en fonction d'un certain traitement. Mais aucun de ces travaux ne considère des parties de ces interactions inter-composants comme réutilisables dans plusieurs applications. Un médium représente un type d'interaction inter-composants mais qui a la propriété essentielle d'être réutilisable (comme tout composant).

Du coté des composants, des modèles comme les *Entreprise Java Beans*[1] (EJB) ou les composants CORBA[2] autorisent la distribution de composants. Mais ils n'intègrent pas de notion de composant

[1] `http://java.sun.com/products/ejb/`

assurant la communication. Notre approche introduit un nouveau type de composant, axé sur la communication.

Cet article présente comment spécifier formellement un médium de communication. Dans le domaine du génie logiciel, le standard est l'UML [OMG99]. C'est pourquoi nous l'avons choisi pour décrire notre proposition architecturale, bien qu'il soit limité pour certains aspects.

Cette spécification a deux buts. Tout d'abord, elle permet d'écrire un *contrat* de fonctionnement de médium, c'est-à-dire de décrire exactement ce qu'il fait. Ensuite, dans un atelier de génie logiciel UML, elle servira de base à la génération de code automatique, afin de « projeter » cette spécification sur différents modèles comme les Entreprise JavaBeans, les composants CORBA…

La partie suivante explique de manière générale comment spécifier un médium. Ensuite, pour illustrer, nous détaillerons la description de deux médiums : un médium très simple de communication point-à-point unidirectionnelle asynchrone et un médium de vote. Nous verrons également des exemples de leur utilisation.

2 Spécification de médium en UML

2.1 Informations à décrire

Les médiums offrent des services de communication aux autres composants de l'application. Il faut spécifier formellement ces services ainsi que leur fonctionnement. Dans certains cas, les composants voulant se connecter à un médium doivent implémenter certains services qui sont appelés par le médium. Le médium peut posséder également des propriétés de configuration qui sont, soit globales, soit locales à une liaison avec un composant. Tout cela doit être également clairement spécifié.

Les composants utilisent, en fonction de leur besoin, certains services du médium mais pas tous. Dans un médium de diffusion par exemple, un composant voulant émettre utilise le service d'émission. Les autres composants désirant seulement recevoir des informations n'ont besoin que d'un service de réception. Les composants sont donc classés en différents *rôles*, en fonction de leur besoin de communication. A chaque rôle est associé un certain nombre de services. Pour le médium de diffusion par exemple, il y a un rôle d'émetteur et un rôle de récepteur.

2.2 Vue statique d'un médium

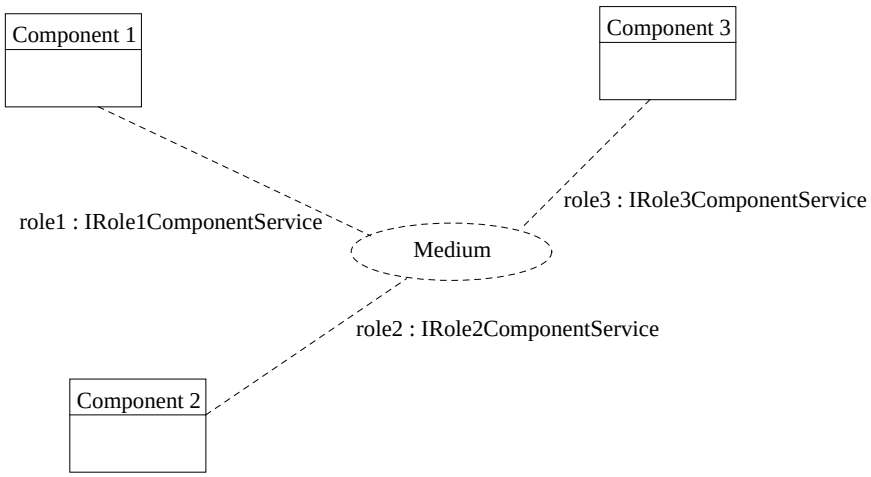

FIG. 1 : Liaisons génériques médium/composant

[2] http://www.omg.org

Les médiums sont représentés par des collaborations UML. En effet, elles permettent de décrire des interactions entre éléments et l'on peut considérer la communication comme une certaine forme d'interaction. Utiliser des collaborations pour décrire des communications, et donc des médiums, semble être bien adapté.

Dans chaque diagramme de collaboration se retrouvent les rôles joués par les composants se connectant au médium. L'utilisation d'un médium est représentée par un diagramme de classe (ou d'instance) utilisant la collaboration de ce médium, comme dans la figure 1, où un médium offre trois rôles différents joués par trois classes de composant.

Pour chaque rôle « <RoleName> » de composant, il existe deux interfaces :

- I<RoleName>MediumService qui est une interface regroupant les services offerts par le médium aux composants jouant le rôle « <RoleName> ». C'est-à-dire les services utilisés par ces composants.
- I<RoleName>ComponentService qui est l'interface regroupant les services qui sont implémentés par le composant jouant le rôle <RoleName> et qui sont appelés par le médium.

Les différentes interfaces et classes intervenant dans une collaboration représentant un médium <MediumName> sont regroupées dans un paquetage particulier, nommé P<MediumName>Medium.

L'utilisation et l'implémentation d'interfaces permet de typer les rôles. Une classe implémentant une interface PDiffusionMedium::ISenderComponentService peut se connecter au médium nommé « Diffusion » et dans la collaboration décrivant le médium, jouer le rôle Sender.

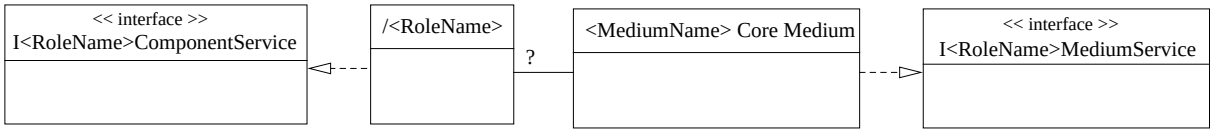

FIG. 2 : Description interne générique d'un médium et d'un rôle

La figure 2 représente les classes impliquées et leur relation pour un rôle générique de nom <RoleName>[3] et un médium de nom <MediumName>. Elle se situe au niveau de la description de la collaboration (diagramme de collaboration), la figure 1 étant à celui d'utilisation (diagramme de classe). Le « ? » de l'association entre la classe /<RoleName> et la classe <MediumName> Core Medium est remplacé dans une vraie description par le nombre de composant de ce rôle qui peuvent se connecter à ce médium.

Un certain nombre d'éléments (classes, interfaces…) pourront être en relation avec la classe <MediumName> Core Medium afin de décrire le fonctionnement du médium et de ses services.

Les propriétés globales du médium sont représentées par des attributs de la classe <MediumName> Core Medium. Les propriétés locales à une liaison composant/médium le sont dans une classe d'association de l'association entre les classes <MediumName> Core Medium et le rôle adéquat.

2.3 Vue dynamique et comportementale

Des expressions OCL [WK98] permettent de décrire une partie du comportement du médium en spécifiant ses invariants ainsi que les pré et postconditions associées à chacun de ses services.

En UML, un diagramme d'interaction peut se décomposer en deux parties. La première est le diagramme de collaboration qui décrit l'aspect structurel (les relations entre les éléments intervenant dans l'interaction) ainsi que les messages échangés lors de cette interaction. La deuxième est le diagramme de séquence qui met l'accent sur le point de vue temporel, l'ordre chronologique des

[3] La manière d'écrire le nom d'un rôle dépend de son contexte d'utilisation. Un rôle nommé Sender dans nos descriptions informelles se transformera en /sender dans un diagramme de collaboration (le nom de la classe représentant le rôle) et en `sender` pour une utilisation de collaboration (le nom du lien représentant le rôle).

événements. Malheureusement ce type de diagramme se situe au niveau instance, ce qui ne permet pas de décrire de manière générale une interaction mais juste de montrer un exemple de ce qu'elle pourrait être. Il n'est donc pas envisageable d'utiliser des diagrammes de séquence pour spécifier un médium ce qui est dommageable car ils contiennent des primitives de gestion de la séquentialité d'appel d'opérations et de contraintes temporelles. Il faut donc utiliser d'autres vues pour exprimer ce type de contrainte.

La séquentialité d'appel d'opérations est définissable à l'aide de l'utilisation de messages dans le diagramme de collaboration représentant le médium. Ces messages étant numérotés voire imbriqués, il définissent un ordre d'appel.

Les contraintes temporelles sont spécifiables à l'aide de diagrammes d'états (opérations `after` et `when`). Typiquement, pour exprimer un blocage, une synchronisation ou une durée maximale d'exécution d'une opération, il faut lui associer un diagramme d'états. Ce type de diagramme contient un état d'attente à partir duquel partent des transitions qui sont déclenchées lorsqu'un certain temps est écoulé ou qu'une condition de déblocage est vérifiée. Ces transitions mènent vers des états que l'on pourra référencer dans une postcondition en OCL à l'aide de la primitive `oclInState`. Par exemple, la figure 7 montre comment gérer un temps maximum `timeout`. Quand ce temps est écoulé, l'état TimeOut est atteint.

L'utilisation d'OCL est intéressante car ce langage permet d'écrire formellement et précisément des contraintes et des expressions booléennes. Nous proposons donc d'étendre son utilisation à tous les diagrammes où des expressions booléennes sont utilisées. Dans notre cas, cela signifie que nous pouvons écrire les gardes des messages des diagrammes de collaboration et les transitions des diagrammes d'état avec des expressions en OCL.

Les expressions OCL sont définies dans un contexte bien précis. Dans le cas d'un message entre deux classes pour un diagramme de collaboration, le contexte est la classe émettrice du message. Pour un diagramme d'états décrivant une classe, le contexte est cette classe. Pour un diagramme d'états décrivant une opération d'une certaine classe, le contexte est cette classe.

2.4 Extensions d'OCL

Nous avons ajouté deux notions à OCL :

- Le pseudo-attribut *`caller`*. Il n'est utilisable que dans la définition des pré et des postconditions d'une opération et représente l'instance qui a appelé cette opération. Il est utilisé de la même façon que l'attribut `self`.
- La primitive *oclCallOperation* :
  Sa signature est : `object.oclCallOperation(opName [, param]*)`. Elle est utilisable uniquement dans une postcondition et permet de préciser qu'une opération `opName` avec une série de paramètre (`[, param]*`) a été appelée sur l'objet `object`. Cet appel a lieu pendant l'exécution de l'opération dont la postcondition contient cette contrainte. Si cette opération retourne une valeur, celle-ci peut être récupérée et permettre notamment d'initialiser une variable.
  Prenons l'exemple suivant :

```
context MaClasse::op1()
post:
      obj.oclCallOperation(op2, true)
```

  Cette spécification précise que pendant l'exécution de l'opération `op1`, la fonction `op2` a été appelée avec le paramètre `true` sur l'objet `obj`.

Dans les exemples, ces extensions seront notées en *italique* afin de bien les différencier des primitives standards d'OCL.

2.5 Résumé de la méthodologie

Un médium est spécifié à l'aide de trois « vues » différentes :

- Un diagramme de collaboration pour décrire l'aspect structurel du médium. Des messages pourront être ajoutés afin de décrire les appels d'opération réalisés dans le cadre de l'exécution d'un service (plusieurs interactions, correspondant chacune à un service, seront décrites si besoin). La numérotation de ces messages permet de spécifier la séquentialité de ces appels.
- Une vue « contraintes OCL » qui permet de décrire les invariants de classe du médium (voire des rôles si besoin) et des pré et postconditions sur les services qu'il rend.
- Des diagrammes d'états associés au médium ou à ces services afin de gérer les contraintes temporelles et de synchronisation. Ils seront utilisés notamment dès qu'une opération n'est pas atomique.

La généralisation de l'utilisation d'OCL aux diagrammes d'états et aux messages des collaborations permet d'être plus précis dans l'écriture des contraintes et de faire le « lien » entre les différentes vues. De plus, dans un outil, les contraintes écrites en OCL seront automatiquement analysables, contrairement à des expressions booléennes écrites en langage naturel ou non clairement défini.

3 Exemples de médium de communication

Un premier exemple très simple est la description d'un médium de communication point-à-point asynchrone unidirectionnelle. Un deuxième exemple plus complexe, concernant un médium de vote, est ensuite détaillé ainsi que deux applications différentes l'utilisant.

3.1 Médium de communication point-à-point asynchrone unidirectionnelle

3.1.1 Description informelle

Il y a deux composants pouvant se connecter au médium : un émetteur et un récepteur. La communication est unidirectionnelle (de l'émetteur vers le récepteur). Elle se fait dans le style « boîte aux lettres » : les messages sont envoyés de manière asynchrone (l'émetteur et le récepteur ne sont jamais bloqués). Les deux composants connectés doivent être différents[4]. Les messages sont ordonnés, ils sont lus dans l'ordre où ils sont envoyés.

3.1.2 Liste des services offerts par le médium

Le composant émetteur joue le rôle Sender et le récepteur le rôle Receiver. Les services offerts par le médium sont les suivants :

- rôle Sender : `void send(Message msg)` : envoyer un message `msg` à l'autre composant (le récepteur)
- rôle Receiver : `Message receive()` : lire le premier message non lu envoyé par l'autre composant (l'émetteur). Si aucun message n'est disponible, l'opération renvoie `null`.

3.1.3 Diagramme de collaboration du médium

La figure 3 représente la collaboration décrivant le médium. On y retrouve les deux rôles Sender et Receiver. Aucun de ces rôles n'a besoin d'implémenter de services qui seront appelés par le médium, donc les interfaces ISenderComponentService et IReceiverComponentService sont vides. Les interfaces ISenderMediumService et IReceiverMediumService contiennent les signatures des services du médium.

[4] Le fait que les deux composants soit différents est un choix de conception. Il aurait été naturellement possible d'autoriser un composant à communiquer avec lui-même, bien que l'intérêt en soit limité pour une spécification de communication entre composants distants.

Au médium est associée une file de messages de type « FIFO », représentée par l'association ordonnée entre les classes PTP Core Medium et Message. Dans cette file seront déposés les messages émis par le rôle Sender (service `send`) et lus par le rôle Receiver (service `receive`).

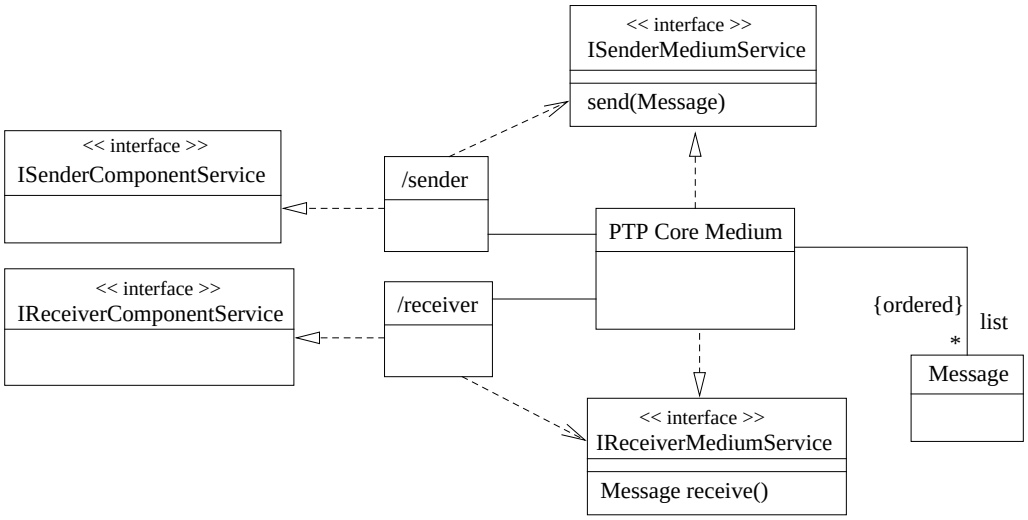

FIG. 3 : Médium point-à-point asynchrone unidirectionnel

3.1.4 Spécification des invariants et des services en OCL

Les composants émetteur et récepteur sont différents (un composant ne peut communiquer avec lui-même en utilisant le médium) :

```
context PTP Core Medium
    inv: self.sender <> self.receiver
```

L'appel de `send` ajoute un message à la fin de la liste :

```
context PTP Core Medium::send(Message msg)
    post: file = file@pre->append(msg)
```

L'appel de `receive` renvoie le premier message de la file et l'en retire, ou `null` si aucun message n'est disponible :

```
context PTP Core Medium::receive() : Message
    post:
        if file->isEmpty
        then result = null
        else
            file = file@pre->subSequence(2,file@pre->size)
            and result = file@pre->first
        endif
```

Le comportement du médium est entièrement spécifié en OCL. Pour des médiums plus complexes, l'utilisation de diagramme d'états permet de bien détailler le comportement dynamique.

3.1.5 Exemple d'utilisation

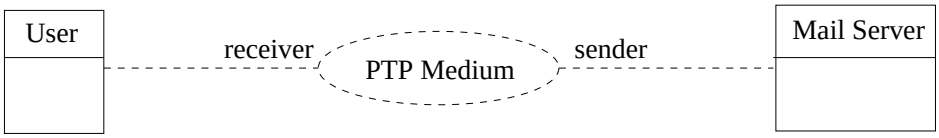

FIG. 4 : Exemple d'utilisation du médium point-à-point

La figure 4 montre un exemple d'utilisation du médium point-à-point asynchrone unidirectionnel. Il sert à gérer les mails d'un utilisateur. Chaque mail reçu par le serveur de mail est envoyé au médium (la classe Mail Server jouant le rôle Sender et qui appelle le service `send` avec en paramètre un mail). L'utilisateur appelle un service du médium pour lire ses mails (la classe User qui joue le rôle Receiver et utilise le service `receive`). Le médium s'occupe de toute la communication entre l'utilisateur et le serveur, y compris le stockage temporaire des mails non lus.

3.2 Médium de vote

3.2.1 Description informelle

Le médium de vote permet à deux types de composants différents de communiquer. Certains composants sont initiateurs de votes : ils proposent une liste de valeurs dans laquelle, les autres composants qui sont des votants choisiront chacun un élément. L'initiateur de vote précise la durée maximale du vote pendant laquelle les composants votants pourront voter. Le vote n'est pas obligatoire. Le service est bloquant tant que le vote n'est pas terminé (c'est-à-dire tant que tous les votants n'ont pas votés ou que le temps maximum de vote n'est pas écoulé). Plusieurs votes peuvent avoir lieu simultanément.

3.2.2 Liste des services

Il y a deux rôles :
- VoteProposer pour les composants initiateurs de votes.
- Voter pour les composants votants.

Pour le rôle VoteProposer, les services sont les suivants :
- du coté du médium :
  - `Value[] askForVote(Value[] prop, Integer timeout)` : demande de voter à tous les votants. `prop` est la liste des valeurs parmi lesquelles les votants choisissent. `timeout` est le temps maximum qui leur est imparti pour voter. En retour, le service renvoie la liste des valeurs choisies par les composants ayant voté. Ce service est bloquant tant que le vote n'est pas terminé.

Pour le rôle Voter, les services sont les suivants :
- du coté du composant :
  - `void askForVote(Value prop[], Integer timeout, Integer voteId)` : demande de vote. `prop` est la liste des valeurs parmi lesquelles le composant doit en choisir une. `timeout` est le temps maximum qu'il a pour voter et `voteId` est l'identificateur du vote permettant de différencier les différents votes en cours.
- du coté du médium :
  - `void vote(Value choice, Integer voteId)` : réponse à un vote. `voteId` est l'identificateur du vote auquel participe le composant et `choice` est la valeur qu'il a choisie.

3.2.3 Diagramme de collaboration du médium

Le diagramme de collaboration du médium de vote est décrit par la figure 5. La classe Vote représente un vote. Un vote est identifié par la valeur de `voteId` dans l'association qualifiée entre les classes Vote Core Medium et Vote. L'attribut `finished` permet de savoir si le vote est terminé (valeur `true`) ou non (valeur `false`). L'association `proposed` représente la liste des valeurs proposées pour le vote et `chosen` celle des valeurs choisies par les votants.

L'association qualifiée `choice` entre le rôle Voter et la classe Value permet de préciser si un composant a déjà voté ou non pour un vote identifié par la valeur de `voteId` (il a déjà voté si l'association existe).

L'opération `initVote(Value prop[])` de la classe Vote Core Medium sert à initialiser un vote (création d'un objet Vote et son association `proposed`) avant de demander à tous les votants de voter.

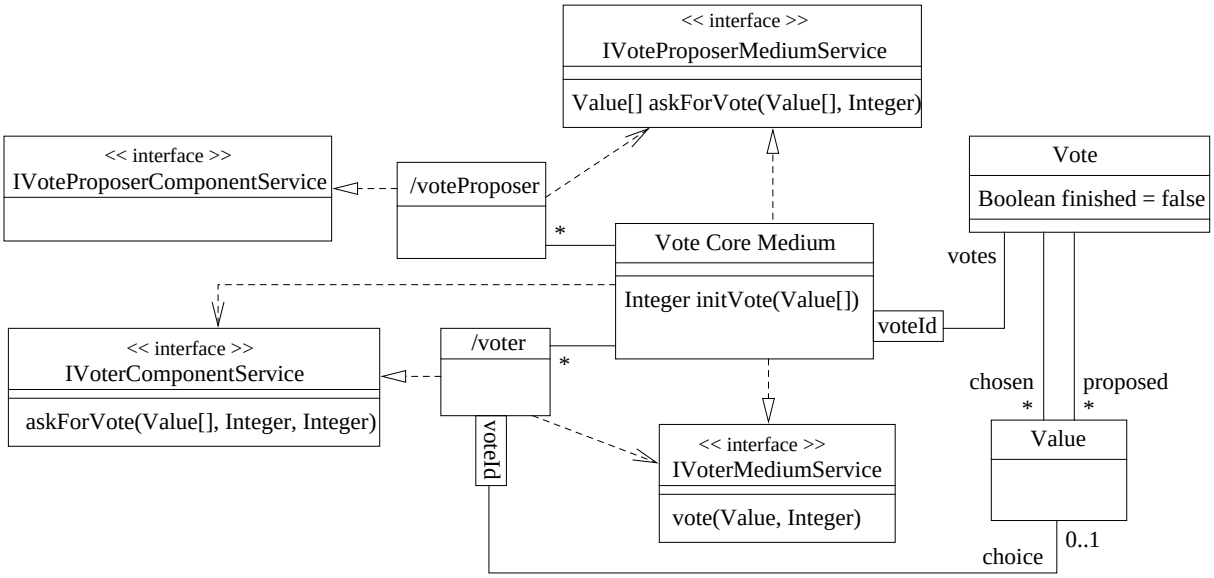

FIG. 5 : Diagramme de collaboration du médium de vote

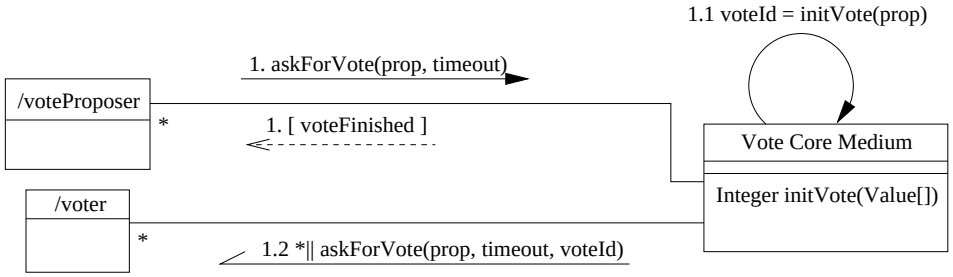

Fig. 6 : Diagramme d'interaction pour le service `IvoteProposerMediumService::askForVote(Value[] prop)`

La figure 6 représente un sous-ensemble du diagramme de collaboration du médium de vote (voir la figure 5) auquel sont ajoutés les messages constituant l'interaction correspondant à l'opération `askForVote` (celle de l'interface `IVoteProposerMediumService`). Les seuls éléments qui ont été conservés sont les deux rôles et la classe représentant le médium.

L'interaction est la suivante :

1. L'opération `askForVote(prop, timeout)` est appelée (message 1). `prop` est la liste des valeurs parmi lesquelles choisir et `timeout` le temps maximum de vote.
2. Le vote est initialisé par appel de l'opération `initVote(prop)` (message 1.1) qui retourne l'identificateur du vote courant (`voteId`).
3. La demande de vote est envoyée ensuite en parallèle à tous les composants jouant le rôle Voter (message 1.2)
4. Certains composants Voter votent mais pas obligatoirement tous (c'est pour cela qu'il n'est pas possible d'ajouter un message indiquant l'appel de l'opération `vote`).
5. L'opération `askForVote` du message 1 retourne lorsque le vote est terminé, ce qui est représenté par la garde `[ voteFinished ]`. Celle-ci est vraie lorsqu'une des deux conditions suivantes est vérifiée :
   - le temps `timeout` est écoulé.
   - tous les votants ont répondu.

Ultérieurement nous verrons comment spécifier formellement `voteFinished` à l'aide d'OCL.

3.2.4 Spécification à l'aide d'OCL

Opération : Integer initVote(Value[] prop)

Cette opération initialise un vote à l'aide de la liste des valeurs (paramètre `prop`) parmi lesquelles les votants en choisissent une. Il retourne un nouvel identificateur de vote qui est unique (qui n'existait pas avant, c'est-à-dire qu'un vote ayant cet identificateur n'existe pas).

```
context Vote Core Medium::initVote(Value[] prop) : Integer
post:
      votes[result]@pre -> isEmpty and          -- il n'existait pas de vote
                                                -- ayant result comme
                                                -- identificateur
      votes[result].proposed = prop and
      votes[result].oclIsNew                    -- nouveau vote initialisé
```

`result` contient l'identificateur du nouveau vote.

Opération IVoteProposerMediumService::askForVote(Value[] prop, Integer timeout)

```
context Vote Core Medium::askForVote(Value prop[],Integer timeout) :Value[]
post:
      let voteId = self.oclCallOperation(initVote, prop) in
      voter.olcCallOperation(askForVote, prop, timeout, voteId) and
      votes[voteId].finished=true  and
      voteFinished = true and
      result = votes[voteId].chosen
```

Cette opération retourne, une fois le vote terminé, la liste des choix faits par les votants, qui est représentée par l'association `chosen` pour le vote courant. A la fin de l'opération, le vote est déclaré comme étant terminé : l'attribut `finished` contient `true` et l'expression `voteFinished` est évaluée à `true` également. Ici, `voteFinished` est la même expression que celle de la garde du message 1 de la figure 6.

Cette opération appelle `voteInit` ainsi que `askForVote` chez tous les votants pour leur demander de voter. Cela est spécifiable dans la postcondition de l'opération écrite en OCL. Par contre, l'ordre d'appel de ses deux opérations n'est pas déterminable. Or il est essentiel que `initVote` soit appelée avant `askForVote`. Cet ordre est défini dans le diagramme d'interaction de la figure 6.

Opération : IVoterComponentService::askForVote(Value[], Integer, Integer)

Cette opération n'a pas à être spécifiée car la demande de vote est interprétée par le composant votant comme il le désire. Aucune contrainte de traitement de cette opération ne lui est imposée.

Opération : IVoterMediumService::vote(Value choice, Integer voteId)

Cette opération permet à un votant de répondre à une demande de vote. `choice` est le choix effectué pour le vote identifié par `voteId` par le composant votant. Le vote sera valide si :

- il existe bien un vote identifié par `voteId.`
- pour ce vote, le choix fait est bien dans la liste des valeurs proposées.
- le composant n'a pas déjà participé à ce vote.

Le problème du temps limite de réponse est moins critique ; il est accepté qu'un composant participe à un vote déjà terminé. Dans ce cas, son choix ne sera pas pris en compte.

```
context Vote Core Medium::vote(Value choice, Integer voteId)
pre:
      caller.isKindOf(/voter) and          -- il n'y qu'un composant jouant
                                           -- le rôle Voter qui peut appeler
                                           -- le service
      votes[voteId] -> notEmpty and        -- le vote existe bien
      votes[voteId].proposed -> includes(choice) and       -- choix correct
      caller.choice[voteId] -> isEmpty     -- le composant n'a pas déjà
                                           -- participé à ce vote
post:
      if votes[voteId].finished = false   -- vote non terminé
      then
            votes[voteId].chosen = votes[voteId].chosen@pre ->
                                       including(choice) and
                                            -- on ajoute le choix à la liste
                                            -- des votes
            caller.choice[voteId] = choice -- le votant a voté
      else
            votes[voteId].chosen = votes[voteId].chosen@pre
                                            -- pas de changement car le vote
                                            -- est terminé
      endif
```

3.2.5 Diagrammes d'états

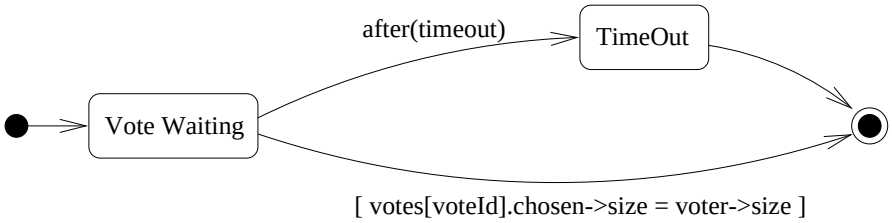

FIG. 7 : Diagramme d'états de l'opération
`Vote Core Medium::askForVote(prop, timeout)`

La figure 7 représente le diagramme d'état de l'opération `askForVote` (appelée par un composant jouant le rôle `VoteProposer`). Il est relativement simple et sert à préciser quand l'opération se termine, c'est à dire quand le vote est terminé.

L'état Vote Waiting est atteint dès le départ et il sera quitté lorsque le vote sera terminé. C'est-à-dire dans l'un des deux cas suivants (à chacun d'entre eux correspond une transition) :

- tous les votants ont répondus. Comme il ne peuvent voter chacun qu'une seule fois, si le nombre de réponses est égal au nombre de votants, ils auront tous votés :
  `voter->size = votes[voteId].chosen->size`
  Avec `voteId` ayant pour valeur celle de la spécification OCL de l'opération `askForVote` décrite précédemment (la valeur du vote courant).
- le temps `timeout` est écoulé (transition « `after(timeout)` »). La valeur de `timeout` est celle passée en paramètre de l'opération. Si cette transition est déclenchée, l'état TimeOut est atteint. Il s'agit d'un pseudo état de terminaison.

Une fois ce diagramme d'état explicité, il est possible de définir l'expression `voteFinished` vue précédemment :

```
voteFinised = (voter->size = votes[voteId].chosen->size) or
              oclInState(TimeOut)
```

Grâce à cela, comme la condition `voteFinished` doit être vraie à la fin de l'exécution de `askForVote` (voir sa postcondition), cette opération ne pourra forcément se terminer que si le temps `timeout` est écoulé ou que tous les votants ont répondus.

4 Exemples d'utilisation de médium

Dans cette partie nous allons étudier deux applications utilisant les mêmes médiums. L'un d'entre eux est le médium de vote spécifié dans la section précédente. L'autre est un médium de diffusion de flux vidéo (appelé « VideoBroadcast Medium »). Il ne sera pas détaillé ici mais son principe est le suivant : un serveur unique (jouant le rôle Server) est connecté au médium et diffuse un flux vidéo qui est reçu par tous les composants clients (jouant le rôle Client).

La première application fait de la diffusion interactive de vidéo. La seconde est une application de visioconférence.

4.1 Application de vidéo interactive

Cette application permet à des utilisateurs de choisir la fin du film qu'ils regardent : un film comportant plusieurs fins est diffusé, et quelques minutes avant qu'il ne se termine, la diffusion s'interrompt, et les spectateurs doivent voter pour choisir la fin qu'ils désirent. La diffusion continue ensuite en fonction du choix majoritaire.

Cette application utilise deux médiums :

- un médium de diffusion de vidéo pour envoyer les films aux utilisateurs
- un médium de vote pour choisir la fin du film

Le serveur de vidéo joue le rôle Server pour le médium de diffusion et le rôle VoteProposer pour le médium de vote car c'est lui qui initiera le vote, une fois qu'il aura terminé de diffuser la première partie du film.

Les spectateurs, dont le nombre est quelconque, jouent le rôle Client pour le médium de diffusion et Voter pour celui de vote car c'est à eux de choisir la fin du film.

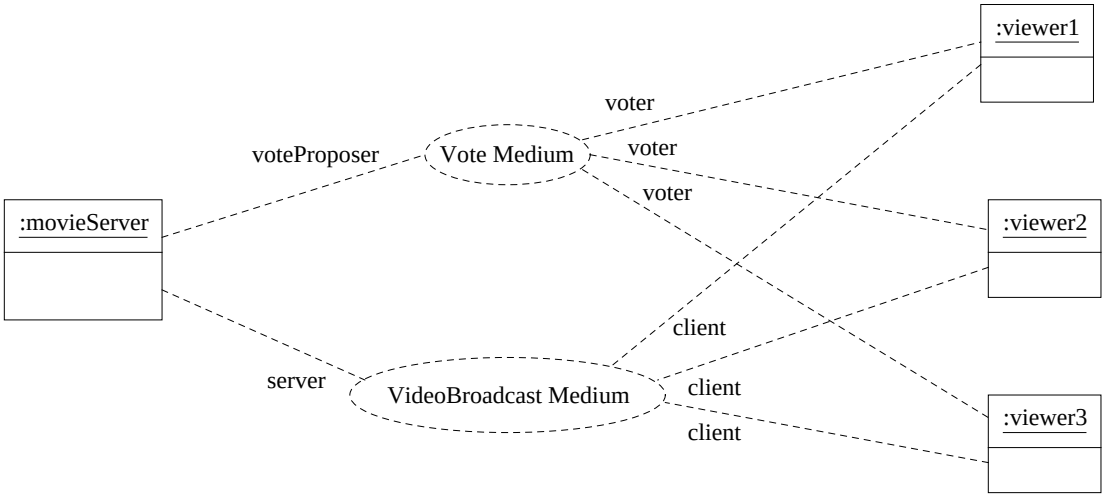

FIG. 8 : Exemple d'application de vidéo interactive

La figure 8 montre un diagramme d'instances représentant cette application. L'objet `movieServer` est l'émetteur du film et l'initiateur du vote. Les trois objets `viewerX` sont les spectateurs du film et en choisissent la fin.

4.2 Application de visioconférence

Cette application permet à des utilisateurs distants de participer à une visioconférence. Chacun d'entre eux est filmé et son image est diffusée à tous les autres utilisateurs. Un participant à la conférence a la possibilité d'initier un vote, auquel participeront tous les utilisateurs connectés à cet instant.

Cette application utilise les mêmes médiums que l'application précédente de vidéo interactive. Un médium de vote sert aux votes et un médium de diffusion est associé à chaque participant pour diffuser son image.

Chaque participant peut initier des votes et voter à tous les votes. Il joue donc pour le médium de vote, à la fois les rôles VoteProposer et Voter. L'image d'un participant est projetée à tous les autres. Comme le médium de diffusion ne permet d'envoyer qu'un seul flux en même temps, il faut qu'il y ait un médium par participant, pour lequel il jouera le rôle Server. Afin de voir l'image des autres participants, chacun d'entre eux doit se connecter en jouant le rôle Client pour tous les médiums de diffusion (sauf le sien).

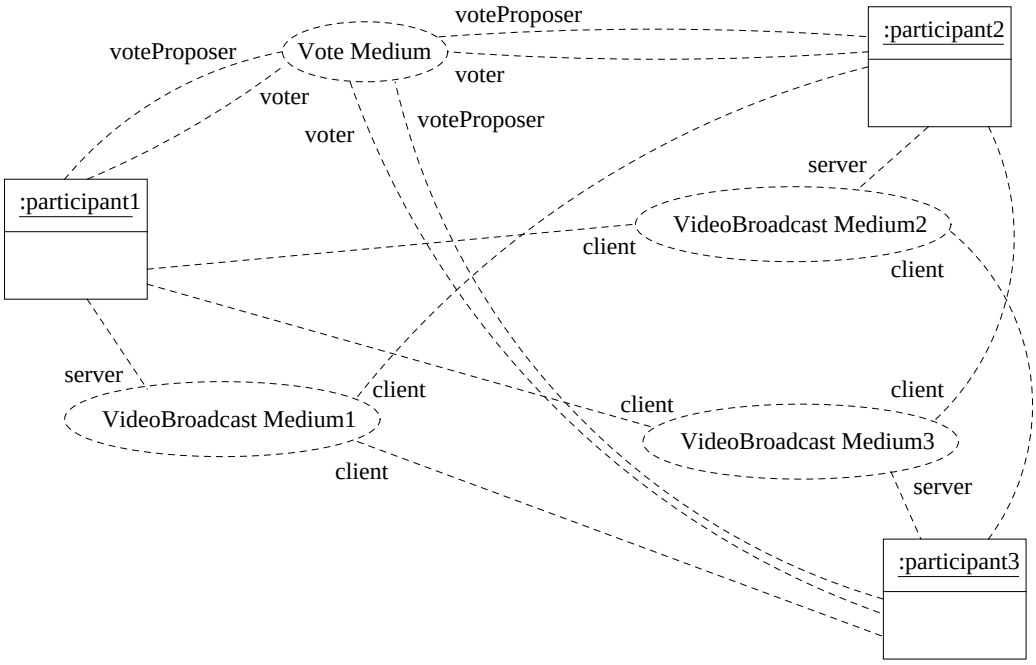

FIG. 9 : Exemple d'application de visioconférence

La figure 9 montre un diagramme d'instances représentant une application de visioconférence à laquelle participe 3 utilisateurs (les objets `participantX`).

5 Conclusion et perspectives

Les médiums de communication sont des composants logiciels réutilisables intégrant des services de communication de tout type. Nous avons présenté comment spécifier un médium de communication en UML. Un exemple simple de médium point-à-point et d'application l'utilisant ont ensuite été étudiés, puis un médium de vote, plus complexe, a été spécifié. Deux applications différentes (une de vidéo interactive et une de visioconférence) utilisant ce médium de vote ont été décrites. Elles ont également été implémentées en réutilisant les deux mêmes médiums. Cela montre bien qu'un médium est réutilisable dans plusieurs contextes.

Nous avons déjà spécifié d'autres médiums, dont notamment un médium de communication point-à-point asynchrone bidirectionnelle (résultant de la composition de deux médiums unidirectionnels), un médium de diffusion d'événements, un médium implémentant le mécanisme de communication Linda [Kie96] et l'une de ses variantes.

Lors de ces spécifications, nous nous sommes heurtés assez rapidement à des problèmes d'expressivité d'UML, concernant principalement la spécification comportementale et dynamique. Comme le montre le médium de vote, il n'est pas possible de se contenter d'un diagramme de collaboration et de contraintes OCL pour décrire un médium. Dès qu'il faut spécifier un comportement dynamique (contraintes temporelles, synchronicité, séquentialité d'opérations…), il devient nécessaire d'utiliser des diagrammes d'états et éventuellement de décrire plusieurs interactions sur un même

diagramme de collaboration. Le lien et la cohérence entre ces différentes vues sont faits à l'aide d'expressions OCL dont nous avons généralisé l'usage partout où cela était possible. Nous avons également été obligé d'ajouter des extensions à OCL, car il ne nous permettait pas d'exprimer des choses essentielles comme de pouvoir référencer l'appelant d'une opération ou de spécifier qu'une opération a été appelée. En conclusion, la spécification d'un médium en UML reste assez complexe voire difficile, même si le fonctionnement du médium est intuitivement simple, comme c'est le cas pour le médium de vote.

Nos travaux continuent à s'orienter vers la spécification et l'implémentation de nouveaux médiums afin d'avoir à notre disposition un catalogue de composants de communication, qui nous permettra de spécifier et de construire de futures applications. Nous allons également faire de la rétro-ingénierie sur des applications existantes, en plaçant la communication – et donc les médiums – au cœur de l'analyse, afin de voir l'impact que pourrait avoir l'utilisation des médiums sur l'analyse des systèmes. En ce qui concerne la génération automatique de code à partir de spécifications de médium, le modèle que nous avons décrit dans cet article va être intégré à un atelier de génie logiciel en cours de développement à l'Irisa[5].

Bibliographie

[Beu00] Antoine Beugnard. *Un Modèle Architectural à Base de Composants pour les Applications Distribuéés*. LMO'2000. Hermes, 2000.

[Del96] Dellarocas, Chrysanthos. *Software Component Interconnection Should Be Treated as a Distinct Design Problem.* 1996

[MT97] Nenad Medvidovic et Richard N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*. Rapport technique, UCI-ICS-97-02, Feb 97. `http://www.ics.uci.edu/pub/arch/sw-and-pubs.html`

[OMG99] OMG. *Unified Modeling Language Specification, version 1. 3.* OMG, June 1999. `http://www.omg.org`

[Kie96] Thilo Kielmann. *Designing a Coordination Model for Open Systems*. Coordination Languages and Models. Lecture Notes in Computer Science 1061, Springer Verlag, 1996.

[Sha94] Mary Shaw. *Procedure Calls Are the Assembly Language of Software Interconnection : Connectors Deserve First-Class Status*. CMU-CS-94-107, Jan 1997

[WK98] Jos Warmer, Anneke Kleppe. *The Object Constraint Language : Procise Modeling with UML*. Addison-Wesley, 1998.

[5] http://www.irisa.fr/pampa/UMLAUT