N° d'ordre : 2851

THÈSE

présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

Éric Cariou

Équipes d'accueil :	IRISA/Triskell, ENST Bretagne/Département Informatique
École Doctorale :	MATISSE
Composante universitaire :	IFSIC

Titre de la thèse :

Contribution à un processus de réification d'abstractions de communication

Soutenue le 17 juin 2003 devant la commission d'examen

Composition du jury :

M.	Michel	Raynal	Président
Mmes	Isabelle	Borne	Rapporteurs
	Laurence	Duchien	
M.	Antoine	Beugnard	Examinateurs
Mme	Mireille	Blay-Fornarino	
M.	Jean-Marc	Jézéquel	

Remerciements

Je tiens tout d'abord à remercier Michel Raynal de m'avoir fait l'honneur de présider mon jury de thèse. J'adresse mes sincères remerciements à Isabelle Borne et Laurence Duchien pour avoir accepté de rapporter ce travail de thèse. Je remercie également Mireille Blay-Fornarino pour sa participation à mon jury.

Cette thèse s'est déroulée dans un cadre particulier en étant bi-localisée entre l'ENST Bretagne à Brest et l'IRISA à Rennes. Je tiens donc à remercier chaleureusement les deux personnes qui m'ont encadrées durant cette thèse : Antoine Beugnard à l'ENST Bretagne et Jean-Marc Jézéquel de l'IRISA qui a dirigé ma thèse. Je les remercie pour leurs précieux conseils tout au long de cette thèse et pour toutes ces intéressantes et constructives discussions.

J'ai donc eu la chance de faire partie partie de deux équipes : le département Informatique de l'ENST Bretagne et l'équipe Triskell de l'IRISA. Je tiens à remercier les membres de ces équipes pour ces quatre agréables années passées entre Brest et Rennes.

Enfin, je tiens à vivement remercier Janet Ormrod et Josette Jouas pour la relecture et la correction des articles que j'essayais tant bien que mal d'écrire dans un anglais correct.

Résumé

Lors de la conception et du développement d'applications distribuées, la communication et les interactions entre les composants répartis représentent un point crucial. La spécification d'abstractions d'interaction entre ces composants est donc un élément primordial lors de la définition de l'architecture d'une application. Si bien souvent des abstractions d'interaction de haut niveau sont définies au niveau de la spécification, à celui de l'implémentation il est par contre plus courant de ne trouver que des abstractions bien plus simples (comme des appels de procédure à distance ou de la diffusion d'événements). Pendant le raffinement qui mène à l'implémentation, ces abstractions de haut niveau ont été diluées, dispersées à travers les spécifications et implémentations des composants de l'application.

Nous proposons un processus de réification d'abstractions de communication ou d'interaction sous forme de composants logiciels. Ce processus permet de conserver l'unité et la cohérence d'une abstraction d'interaction pendant tout le cycle de développement d'une application. Que l'on soit à un niveau de conception abstraite, de conception d'implémentation, d'implémentation ou de déploiement, l'abstraction réifiée et manipulée est toujours la même. Ces composants logiciels sont aussi appelés médiums pour les différencier des autres composants d'une application.

Le processus est composé de plusieurs éléments. Le premier est une méthodologie de spécification de médiums en UML. Cette spécification est réalisée à un niveau abstrait, indépendamment de toute implémentation. Dans la terminologie du MDA (Model-Driven Architecture) de l'OMG, il s'agit d'une spécification de niveau PIM (Platform Independent Model). Cette spécification est le contrat du médium qui doit ensuite être respecté quel que soit le niveau de manipulation considéré. Le deuxième élément est une architecture de déploiement de médiums qui offre la flexibilité nécessaire pour implémenter a priori n'importe quelle abstraction d'interaction. Enfin, le troisième élément est un processus de raffinement qui permet de transformer une spécification abstraite de médium en une ou plusieurs spécifications d'implémentation prenant en compte l'architecture de déploiement et différents choix de conception ou d'implémentation. Dans la terminologie du MDA, le processus de raffinement sert à transformer une spécification de niveau PIM en une ou plusieurs spécifications de niveau PSM (Platform Specific Model).

Mots-clés : architecture et composants logiciels, modélisation, abstractions de communication, collaborations UML, processus de raffinement.

Abstract

During the development of distributed applications, communications and interactions among remote components are a key-point. The specification of interaction or communication abstractions among these components is thus a major task during the definition of an application architecture. If high-level abstractions are very often defined at the specification level, yet, it is common to find only much simpler abstractions at the implementation level (such as remote procedure calls or event broadcasting). During the refinement process, i.e. from specification to implementation, these high-level interaction abstractions are lost and split among the component specifications and implementations.

We propose a process of reification of interaction or communication abstractions into software components. The process allows the consistency and the unity of an interaction abstraction to be maintained, from abstract specification to implementation. The abstraction manipulated remains unchanged throughout the software process, from abstract specification to implementation and deployment through specification implementation. These components are also called mediums in order to differentiate them from "standard" software components of an application.

The process is composed of three parts. The first one is a medium specification methodology in UML, at an abstract level, independently of any implementation. In the MDA (Model-Driven Architecture, defined by the OMG) terminology, this specification is done at PIM (Platform Independent Model) level. This specification is the medium contract that has to be fulfilled at all the levels of the software development process. The second part is a medium deployment architecture that provides the necessary flexibility to implement, in principle, any kind of communication or interaction abstraction. The last part is a refinement process, enabling the transformation of an abstract specification into one or several implementation specifications, conforming to our deployment architecture and to the design or implementation choices. In the MDA terminology, the process enables the transformation of a PIM specification into one or several PSM (Platform Specific Model) specifications.

Key-words: software architecture and components, software modeling, communication abstractions, UML collaborations, refinement process.

Table des matières

Table des figures

Introduction

Lors de la conception et la réalisation d'applications, la technologie objet s'est imposée depuis déjà quelques années grâce à ses qualités, telles que la modularité, la flexibilité ou la réutilisabilité [77]. Un objet n'est jamais isolé, il doit collaborer, interagir et communiquer avec d'autres objets pour réaliser les tâches qui lui sont imparties [6, 58]. La collaboration et la communication entre les objets sont donc un point essentiel dans la construction d'une application. Par nature, les applications distribuées mettent elles aussi particulièrement en avant la collaboration et la communication entre objets distribués comme un élément central. Enfin, le nouveau paradigme de composants logiciels [116] accentue encore cet aspect. En effet, un composant logiciel est une entité qui spécifie clairement ses interfaces d'interaction avec les autres composants (services offerts et requis) et qui est conçue pour être composée avec d'autres composants. La communication ou l'interaction[1] entre des objets ou des composants est donc un point crucial, et cela plus particulièrement dans le cadre d'applications distribuées.

Il n'est cependant pas simple de pouvoir manipuler une interaction, ou plus exactement une abstraction de cette interaction, à toutes les étapes de conception et de développement du logiciel. Ce problème se pose plus particulièrement lors de la phase d'implémentation et d'instantiation. Les plates-formes de composants ou les intergiciels qui sont généralement utilisés pour développer ces applications n'offrent en effet qu'un nombre limité de formes d'interaction entre composants ou objets distribués. Nous pouvons citer notamment les classiques « appel de procédure à distance » ou diffusion d'événements. Il existe peu de plates-formes ou de techniques permettant d'implémenter et d'utiliser tout type d'interaction, indépendamment de sa complexité (qu'il s'agisse de la complexité de l'interaction du point de vue de l'abstraction qu'elle représente ou de la complexité de son implémentation).

Cette difficulté à implémenter des interactions de haut niveau est d'autant plus dommageable, qu'au niveau de la conception et de la spécification d'abstractions d'interaction, de nombreux travaux existent. Ainsi, plusieurs méthodologies de spécification considèrent les interactions ou les collaborations entre objets ou composants comme des éléments architecturaux de premier ordre [40, 15, 100]. De plus, dans le langage de modélisation UML (pour *Unified Modeling Language* [25]), les diagrammes de collaboration

[1] Dans ce document, quand nous parlons de communication, nous en parlons de manière très générale, avec un sens « commun » et non spécifique à un domaine particulier de l'informatique. La communication concerne toutes les façons utilisées par des entités pour échanger des informations et les interpréter. La communication sert donc de support à la réalisation d'interactions, de collaborations ou de coordination entres entités. Parmi ces dernières formes, nous pensons que l'interaction est une des plus importantes. La coordination et la collaboration nécessitent des échanges d'informations (de la communication de manière générale) et des interactions. À ce titre, nous parlons donc d'abstractions de communication ou d'interaction, pour qualifier de manière globale les différentes formes de communication.

permettent d'exprimer facilement des interactions de haut niveau. Malheureusement, pendant le processus de raffinement de la spécification d'une application, ces abstractions de haut niveau disparaissent. Elles ne laissent place au niveau de la spécification de l'implémentation, qu'à des formes de communication souvent limitées du point de vue de l'interaction, comme par exemple des appels de procédures à distance. En effet, elles se basent alors sur les primitives de communication offertes par la plate-forme de composants ou l'intergiciel sur lequel s'appuie la réalisation de l'application.

Contribution de cette thèse

Ainsi, il n'est pas facile de pouvoir manipuler une abstraction de communication ou d'interaction comme une entité unique et cohérente à tous les niveaux du développement du logiciel. Afin de résoudre ce problème, nous proposons dans cette thèse de réifier une abstraction de communication ou d'interaction dans un composant logiciel, que nous appelons un *composant de communication ou d'interaction*. Ces composants sont également appelés *médiums* afin de bien les différencier des composants logiciels « fonctionnels » habituels. Un composant logiciel existe classiquement à toutes les étapes du cycle de vie d'une application, de la spécification abstraite jusqu'à l'implémentation et l'instantiation, en passant par la spécification d'implémentation. En réifiant une abstraction de communication dans un composant, il est aisé de la manipuler et de conserver son unité et sa cohérence pendant tout le cycle de développement du logiciel. Afin de manipuler en pratique ces médiums, nous avons défini un processus complet de réification d'une abstraction de communication, de sa spécification abstraite jusqu'à son ou ses implémentations. Ce processus est composé de trois parties :

- Une méthodologie de spécification de médiums en UML à un niveau de conception abstraite.
- Une architecture de déploiement de médiums dans un contexte distribué.
- Un processus de raffinement permettant de passer de la spécification abstraite à une ou plusieurs spécifications d'implémentation, en prenant en compte des choix de conception et se basant sur l'architecture de déploiement de médiums.

Pour le niveau le plus abstrait, nous avons défini une méthodologie de spécification en UML. Nous avons choisi d'utiliser ce langage car il est le standard de fait dans le domaine de la modélisation ou de la spécification d'applications à base d'objets ou de composants. Les collaborations UML sont par principe bien adaptées pour spécifier des abstractions d'interaction ou de communication. Il était alors naturel de les utiliser lors de la spécification de médiums. Notre méthodologie spécifie comment définir une collaboration UML dans le contexte de la spécification d'un composant de communication. Les médiums peuvent donc aussi être vus comme une manière d'implémenter des collaborations UML. Cette méthodologie, ainsi qu'un exemple de spécification de médium (un système de réservation d'identificateurs), ont été publiés dans [31]. D'autres exemples de spécification de composants de communication ont été présentés dans [29] (un médium de vote) et [30] (un médium intégrant un système *à la Linda* [1]).

Nous avons défini une architecture de déploiement de ces composants dans un contexte distribué. Cette architecture, bien que simple, est bien adaptée pour implémenter la plupart des interactions. Et cela, indépendamment de la complexité d'une interaction, qu'il s'agisse de la complexité de l'abstraction qu'elle représente ou de son implémentation.

Afin de pouvoir passer de la spécification abstraite, qui est définie indépendamment de toute implémentation, à la spécification d'une implémentation donnée et adaptée à l'architecture de déploiement, nous proposons un processus de raffinement de spécifications. De nombreux choix ou contraintes non-fonctionnelles peuvent influencer l'implémentation d'une interaction. Par exemple, introduire des contraintes de sécurité ou de fiabilité modifie beaucoup la manière de réaliser une interaction. Ainsi, il est possible de spécifier plusieurs versions d'implémentation d'une même abstraction et d'utiliser celle qui est la plus adaptée à une application ou à un contexte donné. Une même spécification abstraite peut donc conduire à la définition de plusieurs conceptions d'implémentation. Le processus de raffinement est constitué d'une série de transformations de modèles et de spécifications UML. Il s'inscrit dans une démarche de type MDA[2] [88]. En effet, la spécification abstraite d'un médium est un modèle indépendant d'une plate-forme (niveau PIM[2]) et les différentes conceptions d'implémentation – qui tiennent compte de l'architecture de déploiement – sont des modèles dépendants d'une plate-forme (niveau PSM[2]). L'architecture de déploiement et le processus de raffinement appliqué dans le cas de la résistance à la montée en charge (qui peut être considérée comme une contrainte non-fonctionnelle) pour le médium de réservation ont été publiés dans [32].

Finalement, afin de valider ces concepts, nous avons développé une plate-forme pour faciliter l'implémentation et l'utilisation de médiums. Grâce à elle, nous avons réalisé plusieurs médiums et applications les utilisant. Cela nous a permis de constater que la manipulation d'abstractions d'interaction de haut niveau était aisée à l'implémentation et présentait de nombreux avantages.

Plan du document

Le chapitre 1 présente un état de l'art dans le domaine des abstractions de communication.

Les deux chapitres suivants présentent en détail la contribution majeure de cette thèse. Le chapitre 2 définit la notion de composant de communication. Il commence par montrer comment peut s'intégrer la notion de médium dans le cadre de la spécification ou de l'implémentation d'applications à base de composants distribués. Ensuite, il décrit la méthodologie de spécification abstraite de médiums en UML. À titre d'exemple, les spécifications complètes de trois médiums sont données. Il s'agit tout d'abord d'un médium réifiant une abstraction simple de communication de type boîte aux lettres entre deux composants. Le second réalise une interaction de vote et le dernier une interaction via la réservation d'identificateurs. Enfin, ce chapitre se termine par la présentation de l'architecture de déploiement des médiums.

Le chapitre 3 définit le processus de raffinement de spécifications de médiums. Ce processus est composé de plusieurs étapes. Chaque étape transforme la spécification de l'étape précédente en une nouvelle spécification. Un exemple complet d'application de ce processus est donné à partir de la spécification du médium de réservation d'identificateurs. Nous présentons également dans ce chapitre l'approche MDA et expliquons comment notre processus s'inscrit dans cette démarche.

Le chapitre 4 décrit la plate-forme que nous avons développée pour aider à la construction de composants de communication et à leur utilisation. Nous montrons comment cette

[2]MDA est l'abbréviation de *Model Driven Architecture*, PIM de *Platform Independent Model* et PSM de *Platform Specific Model*

plate-forme permet de gérer des contraintes d'instantiation ainsi que de connexion ou de déconnexion dynamique de composants dans un contexte distribué.

Finalement, nous concluons en résumant les principales contributions de cette thèse et discutons ses perspectives.

Chapitre 1

État de l'art

> *Communication : n.f. 1. le fait de communiquer, d'établir une communication avec (qqun, qqch) 2. action de communiquer, résultat de cette action 3. la chose que l'on communique 4. moyen technique par lequel des personnes communiquent ; messages qu'elles se transmettent 5. ce qui permet de communiquer ; passage d'un lieu à un autre.*
>
> *Communiquer : v. 1. faire connaître (qqch.) à qqn. 2. faire partager 3. rendre commun à, transmettre (qqch.)*
>
> Définitions du Petit Robert [101]

La communication est une notion assez générale qui peut, selon le contexte, prendre plusieurs formes et également recouper des notions comme l'interaction, la collaboration, la coordination voire encore d'autres notions. Nous allons commencer par donner nos définitions de ces différents termes. Dans un contexte informatique, la communication concerne l'échange, la transmission et le partage d'informations et de connaissances entre des entités logicielles. Ces entités peuvent être des objets, des processus, des agents, des composants, etc. La communication est un mécanisme qui permet de réaliser ce partage et ces échanges. De manière plus générale, elle sert également de support à la réalisation d'interactions entre entités logicielles. En effet, une interaction correspond à un ensemble d'actions exécutées mutuellement par un groupe d'entités. Pour que ces actions soient réalisées, il faut que des échanges d'informations aient lieu (il faut envoyer des requêtes d'exécution d'action aux entités). À partir de communications (c'est-à-dire d'échanges d'information) et d'interactions, il est possible de réaliser de la coordination ou des collaborations entre entités. Parmi toutes ces formes, la communication et l'interaction nous semblent donc être celles essentielles à la réalisation de tout ce qui concerne l'échange d'informations, les inter-dépendances et les appels d'actions entre entités. À ce titre, dans ce document, nous insistons plus particulièrement sur ces deux formes. Nous pouvons aussi considérer toutes ces formes (communication, interaction, coordination ou collaboration) comme relevant de la communication de manière générale et globale.

L'abstraction est une opération dont le but est d'isoler et de considérer à part un élément parmi un ensemble plus large. Le mot abstraction est utilisé également dans le sens « les détails sont cachés » et non pas « la notion est floue et mal définie ». Une abstraction de communication est donc la représentation d'un type ou mode de communication entre des entités logicielles. Voici quelques exemples d'abstractions de communication : un appel de procédure à distance, un système de diffusion d'événements,

un protocole de consensus, une mémoire partagée, un système de diffusion de flux vidéo, etc. Comme nous pouvons le constater, ces abstractions peuvent être de niveau et de complexité très différents.

La réification est l'opération de transformation en quelque chose de concret. Le résultat de la réification d'une abstraction de communication est une entité manipulable et utilisable correspondant à un type ou mode de communication.

Dans ce chapitre, nous étudions les méthodologies, techniques ou travaux traitant de la réification d'abstractions de communication ou d'interaction, ou bien permettant de réaliser cette réification. Nous nous intéressons pour chacune d'entre elles aux phases du cycle de développement du logiciel qu'elles traitent ou auxquelles il est possible de les rattacher. Ces phases correspondent, entre autres, à la spécification abstraite, la spécification d'implémentation et l'implémentation. La spécification abstraite est réalisée à un niveau qui ne tient pas compte de l'implémentation, elle définit la sémantique d'une abstraction. La spécification d'implémentation est un raffinement de la spécification abstraite qui fait apparaître des contraintes que doit respecter l'implémentation. Lorsque cela est possible, nous traitons aussi du raffinement et des liens entre les différentes phases ou niveaux pour chacune des approches considérées.

Ces méthodologies et techniques sont variées. Elles concernent des aspects très différents de la gestion des communications et des interactions dans les applications. Cela va des phases d'implémentation avec les intergiciels à des niveaux conceptuels et abstraits comme les patrons de conception.

Nous proposons une présentation de ces travaux en trois parties. La première traite des approches à un niveau système. Nous y trouvons les protocoles, les composants logiciels, les intergiciels et la programmation orientée aspect interactionnelle. Ensuite nous présentons l'aspect conception et méthodologie, avec des travaux comme les patrons de conception, le langage de modélisation UML et des méthodologies comme Catalysis et UML Components. Enfin, nous décrivons des langages et modèles permettant de définir des abstractions de communication : les langages de description d'architecture (ADL), les langages et modèles de coordination et les systèmes multi-agents. Parmi tous ces travaux, nous détaillons plus en avant ceux qui nous semblent avoir un lien important avec notre approche. Il s'agit des langages de description d'architecture, des langages et modèles de coordination et de Catalysis. Cette classification en système, conception/modèle et langages n'est certainement pas parfaite. Certaines approches recoupent plusieurs domaines ce qui ne rend pas aisée leur classification. Néanmoins, la classification que nous avons choisie est celle qui nous a semblée être la plus appropriée.

1.1 La vision système des abstractions de communication

Dans cette section, nous décrivons des approches qui proposent des outils et des techniques pour la réalisation et la mise en œuvre d'applications distribuées. Nous étudions les solutions qu'elles offrent pour réifier des abstractions de communication ou d'interaction. Ces approches sont les protocoles, les intergiciels, les plates-formes de composants logiciels et la programmation orientée aspect interactionnelle.

1.1.1 Les protocoles et composition de protocoles

Un protocole est un ensemble de règles qui déterminent le déroulement d'une communication entre plusieurs entités. Les protocoles sont très courants en télécommunication et en informatique. Dans le cadre des réseaux informatiques, nous pouvons citer par exemple les célèbres protocoles IP, TCP ou HTTP. Dans un contexte distribué, des ouvrages comme [98, 99] s'attachent à définir des protocoles et algorithmes relatifs aux problèmes d'exclusion mutuelle, d'interblocage ou d'accès et de partages de données. De nombreux travaux s'attachent également à l'étude de protocoles basés sur le consensus entre des entités distribuées, comme notamment [56, 38].

Tous ces protocoles sont des abstractions de communication. Les travaux sur les protocoles portent généralement sur leur spécification. Il n'existe pas ou peu de techniques permettant de générer le code correspondant à la spécification d'un protocole ni traitant du raffinement de ce protocole. L'implémentation d'un protocole se fait donc généralement de manière ad hoc.

Plusieurs approches s'intéressent également à la composition de protocoles pour en créer de nouveaux. Ces outils et plates-formes permettent alors de construire de nombreux protocoles en combinant entre eux des protocoles plus simples ou des briques de protocole. L'assemblage des briques de protocole peut se faire de différentes façons, notamment en créant une pile de briques ou en les composant entre elles. Ces briques servent par exemple à définir de la synchronicité, de l'envoi fiable de message, du cryptage, etc. Dans ce domaine de composition de protocoles, nous pouvons citer notamment BAST [45, 46], Ensemble [53] ou Coyotte [23, 54]. Généralement ces approches définissent un langage ou un formalisme pour définir le protocole et offrent un support d'exécution pour son utilisation. Elles couvrent donc les phases de spécification et d'implémentation.

Analyse de cette approche

Les plates-formes de composition de protocoles permettent de définir simplement de nouveaux protocoles et donc de nouvelles abstractions de communication. Elles permettent également de générer l'implémentation de ces abstractions et de les utiliser dans une application. Nous pouvons noter tout de même quelques limitations. La première est que la spécification est liée à l'implémentation. En effet, chaque brique de base possède une spécification et correspond à une brique logicielle lors de l'implémentation. La spécification d'un protocole est donc réalisée en décrivant la composition de ces briques logicielles. Si cela présente l'avantage de pouvoir spécifier formellement un protocole à l'aide d'éléments réutilisables, cela réduit le niveau d'abstraction de cette spécification. En effet, certains protocoles pourraient être définis de manière plus abstraite ou plus simple que sous la forme de composition d'éléments de protocoles. Spécifier un protocole de cette façon revient à définir une spécification d'implémentation adaptée à la plate-forme de composition considérée, et non pas une spécification abstraite c'est-à-dire indépendante de toute implémentation.

La deuxième limitation concerne les protocoles qu'il est possible de définir avec ces approches. Les briques de base sont très bien conçues pour construire certains types de protocoles, comme des protocoles de diffusion fiable ou atomique par exemple. Mais elles ne permettent pas de définir tout type d'abstraction de communication. Par exemple, des abstractions ayant un état comme la communication à travers une mémoire partagée. Ces

plates-formes ne conviennent donc pas à la spécification de ce genre d'abstractions.

Nous proposons une approche permettant de spécifier tout type d'abstraction et à un niveau plus abstrait que celui des approches dont nous venons de parler. La composition de briques de base est une solution parmi d'autres pour définir une spécification et son implémentation. Nous ne préconisons pas de solution particulière sur ce point, ce qui permet d'utiliser au besoin différents moyens de conception et d'implémentation. Néanmoins, la composition telle qu'elle est utilisée par ces outils serait un ajout intéressant à notre approche et ces plates-formes pourraient être une base intéressante pour le développement de médiums.

1.1.2 Les intergiciels

Une application distribuée peut être formée de composants hétérogènes qui ne fonctionnent pas tous sur les mêmes systèmes d'exploitation et ne sont pas écrits dans les mêmes langages. Un intergiciel (*middleware* en anglais) est conçu pour gérer cette hétérogénéité et les problèmes liés à la distribution de ces composants. Un intergiciel peut être vu comme une abstraction qui offre à tous les composants un support pour leur communication. Il s'intercale entre le niveau applicatif et le système d'exploitation.

[42] liste trois types d'intergiciel :

- Les intergiciels transactionnels qui sont utilisés pour interagir avec des bases de données.
- Les intergiciels orientés messages ou événements qui permettent l'envoi et la diffusion de messages à plusieurs composants simultanément.
- Les intergiciels de type « appel de procédure à distance » ou RPC (pour *Remote Procedure Call*) qui permettent à un composant d'appeler une procédure sur un autre composant.

La dernière catégorie d'intergiciel est très courante et dans un contexte de programmation objet, nous y trouvons des technologies comme CORBA [86, 110, 49] de l'OMG ou les RMI [113, 39, 3] (pour *Remote Method Invocation*) du langage Java.

Les intergiciels offrent une gestion très intéressante de la distribution et de l'hétérogénéité au niveau de l'implémentation, mais en terme d'abstractions d'interaction offertes, ils sont souvent limités. En général, un intergiciel offre une abstraction de communication unique ou un petit nombre d'abstractions. Par exemple, pour les intergiciels orientés messages, l'abstraction d'interaction est la diffusion de message. Une application distribuée construite sur la base d'un intergiciel doit donc s'adapter au type de communication offert par l'intergiciel [22]. En effet, les intergiciels offrent peu de facilités ou d'outils pour aider à implémenter de nouvelles abstractions de communication[1].

Néanmoins certains intergiciels offrent tout de même ce genre de facilités. Les dernières versions de la norme CORBA définissent notamment la notion d'intercepteurs qui offrent la possibilité de modifier le fonctionnement de l'appel de méthode à distance. L'utilisation d'intercepteurs pourrait aider à réaliser plus facilement de nouvelles fonctionnalités de communication et donc d'ajouter de nouvelles abstractions de communication à un intergiciel de type CORBA.

[1] Il est malgré tout possible d'implémenter une abstraction d'interaction de haut niveau en se basant sur un intergiciel. Mais ce que nous voulons dire ici, c'est que les intergiciels, bien que gérant la communication entre des composants distribués, ne sont pas pour la plupart conçus pour être facilement étendus en leur ajoutant d'autres services ou abstractions de communication.

Analyse de cette approche

Les intergiciels forcent les concepteurs d'applications à n'utiliser qu'une ou un petit nombre d'abstractions de communication. L'intergiciel est en effet vu comme le support de communication unique dans une application. Cela force le programmeur à adapter l'architecture et le fonctionnement d'une application à un intergiciel en particulier. De plus, l'utilisation d'un intergiciel unique dans une application ne permet pas de résoudre tous les problèmes. Par exemple, lors de la diffusion de flux multimédia, un intergiciel de type RPC n'est pas adapté. En d'autres termes, avoir accès à différents services ou abstractions de communication lors de l'implémentation est utile.

Nous proposons à l'implémentation de pouvoir utiliser plusieurs abstractions de communication différentes, en fonction des besoins. Pour cela, la communication entre les composants d'une application se fait grâce à plusieurs médiums. Ainsi, un concepteur d'applications peut utiliser plusieurs services de communication au lieu de l'unique service offert par l'intergiciel. Cela permet de découpler l'implémentation des aspects communication du reste des composants formant l'application. En effet, le code correspondant à la réalisation des services ou des abstractions de communication est uniquement localisé dans les médiums, et non pas dans les composants qui doivent intégrer une partie de la gestion de leurs interactions lorsqu'un intergiciel est utilisé. Nous reviendrons sur ce point dans la section 2.1.1 page 48.

Néanmoins, les intergiciels offrent un support évolué pour la gestion de la distribution. Ils peuvent donc être intéressants comme support de réalisation de médiums dont le but est de faire communiquer des composants répartis.

1.1.3 Les composants logiciels

Définition des composants logiciels

Les composants logiciels se sont imposés depuis quelques temps comme un nouveau paradigme pour le développement d'applications. De nombreuses plates-formes pour l'implémentation d'applications distribuées à base de composants sont apparues ces dernières années. Paradoxalement, si la notion de composant logiciel semble avoir été acceptée et reconnue par la communauté du logiciel, il n'existe pas vraiment de définition « officielle » ou du moins un consensus complet sur ce qu'est un composant. Néanmoins, certaines caractéristiques et propriétés reviennent très souvent dans les différentes définitions des composants. D'après les définitions et descriptions de [40, 116], voici ce que l'on peut dire à propos des composants logiciels.

Un composant est une unité logicielle qui a les propriétés suivantes :

- Il spécifie clairement ses interactions avec l'extérieur en définissant les services qu'il offre et ceux qu'il requiert (ceux qu'il a besoin d'appeler pour son fonctionnement). Ces services sont généralement regroupés en interfaces de services offerts et interfaces de services requis.
- Il est sujet à composition avec d'autres composants afin de réaliser une application ou un autre composant.
- Grâce à la spécification précise de ses interactions avec les autres composants, il est réutilisable dans différents contextes et substituable (par un composant « compatible »).
- C'est une unité de déploiement.

La spécification des interactions et des interfaces de services offerts et requis ne se limite pas en général à définir la liste de ces services mais spécifie également leur sémantique. Cela revient à définir le *contrat* [21, 76, 77] d'un composant. Avec ce contrat, il est possible de savoir ce que fait ce composant et ce dont il a besoin pour fonctionner (avec la spécification des interfaces de services requis). Ainsi, il est possible d'utiliser et de réutiliser ce composant sans avoir besoin de s'intéresser au détail de son implémentation. Cette séparation de l'implémentation et de la spécification d'un composant est également une caractéristique essentielle des composants.

Un composant existe sous plusieurs formes et peut être manipulé à différentes phases du cycle de développement du logiciel. Tout d'abord, il existe à l'analyse et à la conception sous forme d'une spécification de contrat de composant. Ensuite, il est possible à un niveau plus bas, de spécifier une implémentation de ce composant. Bien entendu, le composant existe aussi lors de l'implémentation et de l'exécution. À chacun de ces niveaux il est également possible de définir des assemblages de composants afin de spécifier ou de réaliser soit un composant plus complexe, soit une application résultant de cet assemblage.

Pour prendre un exemple concret, considérons un correcteur orthographique sous forme de composant. Celui-ci définit deux interfaces de services, l'une pour l'administration du composant (ajouter des dictionnaires et gérer les langues) et l'autre pour effectuer des corrections et des vérifications d'orthographe de mots. Un traitement de texte qui veut offrir une fonction de correction automatique du texte entré par l'utilisateur, définit une interface de service requis contenant des services de correction et de vérification d'orthographe. Si ces services requis sont compatibles avec ceux de notre correcteur orthographique, alors ces deux composants peuvent être connectés et le traitement de texte peut utiliser les services du correcteur orthographique pour corriger le texte de l'utilisateur. Notre correcteur orthographique est substituable par tout autre correcteur qui respecte la définition des services requis par le traitement de texte. Il est également réutilisable dans d'autres contextes comme par exemple un logiciel de courrier électronique voulant lui aussi offrir une fonctionnalité de correction du texte entré par l'utilisateur.

Les modèles et plates-formes de composants logiciels

Il existe aujourd'hui plusieurs plates-formes ou modèles de composants logiciels. Nous pouvons citer les Enterprise Java Beans (ou EJB) de Sun [114, 3], le récent .Net de Microsoft [79, 78, 112] ou bien encore Fractal [26] du consortium ObjectWeb. L'OMG a également défini et normalisé un modèle de composants, appelé CORBA Component Model (ou CCM) [85, 68], dont une réalisation est OpenCCM [91, 68]. Le but principal de ces plates-formes est de pouvoir implémenter des applications distribuées à base de composants logiciels. Pour cela, elles offrent un support d'exécution et de gestion des composants.

En terme de communication, ces plates-formes fonctionnent sur le même principe que les intergiciels. En fait, on peut considérer une plate-forme de composants comme étant composée d'un intergiciel servant à faire communiquer les composants distants (dans la plupart des cas via des appels de services à distance et d'émissions d'événements) et d'un support d'exécution pour les composants. Ce dernier a la charge de gérer les composants en offrant des outils comme des fabriques de composants pour les instantier et les déployer. Il offre également des services de gestion de la persistance de composants, de transactions vers des bases de données, des fonctionnalités assurant de la sécurité, etc. Ainsi, ces

plates-formes permettent de réaliser des composants et de gérer leur exécution. Pour plus d'informations sur les modèles de composants, le lecteur est invité à consulter [69].

Analyse de cette approche

Du point de vue de la communication, les plates-formes de composants ont les mêmes caractéristiques et limitations que les intergiciels sur lesquels elles se reposent. Elles n'offrent qu'un nombre limité de services de communication et ne sont pas intrinsèquement conçues pour aider à la réalisation d'abstractions de communication.

De plus, la plupart des modèles de composants se basent sur des interfaces de services mono-localisées. Tous les points d'accès aux services d'un composant sont localisés sur un seul site (et cela même si le composant peut parfois être éclaté sur plusieurs sites). Cela limite la manière dont les composants peuvent interagir entre eux et contraint leur architecture. Nous pensons qu'il est utile qu'un composant puisse offrir plusieurs points d'accès sur des sites différents (et donc aussi qu'il puisse être éclaté). C'est d'ailleurs une des bases de l'architecture de déploiement de médiums que nous avons définie.

1.1.4 La programmation orientée aspect interactionnelle

La programmation par aspects [59] propose une nouvelle façon d'organiser le code des applications. Une application est généralement découpée en plusieurs modules ou composants, chacun réalisant une ou plusieurs fonctionnalités ou préoccupations. Néanmoins, le code correspondant à certaines fonctionnalités est parfois éclaté entre plusieurs modules. Cette fonctionnalité est dite transverse et correspond à un aspect. Le principe de la programmation orientée aspect est de représenter une application comme formée d'un corps principal et d'un ensemble d'aspects. Ces aspects sont ensuite tissés sur le corps principal pour générer le programme final.

Par exemple, ajouter de la sécurité dans un programme nécessite souvent de modifier plusieurs modules ou composants de celui-ci. Il s'agit donc d'une fonctionnalité transverse. Avec la programmation par aspect, il faut définir un aspect de sécurité qui contient les actions à effectuer pour gérer cette sécurité. Ensuite, il faut préciser à quels endroits du programme (ces endroits sont appelés points de coupure) cet aspect doit être appliqué. La programmation par aspect permet donc d'organiser différemment certaines fonctionnalités. Elle permet une plus grande maintenabilité et évolutivité d'une application, en évitant de devoir intervenir à plusieurs endroits dans une application pour modifier une fonctionnalité ou une préoccupation.

[93] est une approche orientée aspect pour la gestion des interactions entre les objets formant une application. Son principe est de proposer de raffiner des interactions (des appels de services) au sein d'un groupe d'objets par des aspects. Le but est de pouvoir ajouter des préoccupations (comme par exemple de l'authentification ou de la sécurité) en les associant à des interactions entre objets. Cette approche de programmation orientée aspect interactionnelle définit un langage de spécification de ces aspects interactionnels. Ce langage est basé sur le λ-calcul. Une plate-forme permet de réaliser et d'utiliser ces aspects avec le langage de programmation Java dans un contexte distribué.

Analyse de cette approche

Cette approche ne permet pas directement de réifier des abstractions de communication. Un aspect interactionnel n'est pas une abstraction d'interaction ou de communication. Un aspect s'applique sur des interactions entre objets. Ces interactions ne doivent pas – d'après un des principes de cette approche – être modifiées par l'application de l'aspect. L'aspect représente un raffinement de l'interaction et non pas une modification de celle-ci. Une abstraction de communication ou d'interaction doit donc être réalisée directement par les objets eux-mêmes.

Par contre, cette approche permet de réaliser des variantes d'implémentation de ces interactions. En effet, un aspect peut par exemple rajouter de l'authentification ou de la sécurité au niveau des interactions entre les objets. Ainsi, il est possible de rajouter des contraintes ou des préoccupations non-fonctionnelles[2] sur une abstraction de communication.

1.2 La vision conception et méthodologie

Nous nous intéressons maintenant à la réification d'abstractions de communication lors de la conception d'applications. Tout d'abord, nous décrivons les patrons de conception puis le langage de modélisation UML et plus particulièrement ses diagrammes de collaboration. Ensuite, nous étudions des méthodologies ou processus de développement d'applications, qui partent de la phase d'analyse et qui vont jusqu'à l'implémentation, en passant par la conception. Dans cette catégorie, nous détaillons d'abord Catalysis puis plus succinctement UML Components.

1.2.1 Les patrons de conception

Lors de la spécification ou l'implémentation d'applications, un concepteur se retrouve souvent confronté à des problèmes récurrents et similaires. À chaque fois, il doit concevoir une nouvelle solution à chacun de ces problèmes. Bien entendu, toutes les solutions apportées à ce problème récurrent sont très proches. Chaque solution particulière est une adaptation à un contexte donné. L'apparition des *design patterns* ou patrons de conception vient de ce constat. Les patrons de conception servent à définir et lister des solutions à ces problèmes récurrents. Un patron ne définit pas une solution unique à un problème donné mais donne une forme générale de solution pour une classe de problème. Un patron n'est donc pas une solution tout faite que l'on applique mais une forme, un schéma de solution qui est utile pour résoudre un type de problème.

Les patrons ne sont pas spécifiques à l'informatique mais existent dans beaucoup de domaines d'ingénierie. Les premiers patrons sont d'ailleurs apparus dans le monde de l'architecture [2]. En restant dans ce domaine nous pouvons donner un exemple simple de ce que sont les patrons. Lorsqu'un architecte conçoit un bâtiment, il doit bien souvent prévoir des passages d'une pièce à l'autre de ce bâtiment. Le patron qui résout ce problème est une « porte ». Une porte est en effet un dispositif qui permet la communication ou le passage d'une pièce à une autre. Ainsi, plutôt que de réinventer le concept de la porte à chaque fois qu'un architecte est confronté à ce problème, il lui suffira d'adapter ce concept

[2]Dans le sens où elles ne modifient pas la sémantique de l'abstraction du point de vue des objets qui communiquent via cette abstraction.

au cas particulier qu'il a à traiter. En effet, il existe de très nombreux modèles de portes différentes mais qui ont toutes des caractéristiques communes. Dans la définition du patron « porte », le but n'est pas de définir un modèle précis de porte, car elle ne sera pas adaptée à tous les contextes, mais de définir justement ces caractéristiques communes et essentielles qui servent à définir ce qu'est une porte et dans quels cas l'utiliser.

En informatique et en génie logiciel, le monde des patrons est un vaste champ de recherche. Un des ouvrages les plus connus est [44] qui est un catalogue de patrons. On y trouve notamment les patrons observateur, composite, adaptateur ou proxy. Nous pouvons aussi citer [27] qui se focalise plus particulièrement sur les patrons d'architecture d'applications, [105] qui définit des patrons dans un contexte distribué et concurrent, [37] qui traite de patrons pour différents domaines et usages ou bien encore [115, 62] qui s'intéressent à la spécification de patrons en UML.

Certains patrons traitent d'aspects de collaboration ou de communication entre des objets. Ils réifient donc une forme d'abstraction de communication ou d'interaction. Par exemple, si l'on prend le patron observateur comme il est défini dans [44], il est clair qu'il représente une abstraction d'interaction.

Analyse de cette approche

Un des problèmes avec les patrons de conception est qu'à chaque utilisation d'un patron, il faut concevoir et implémenter une version différente en fonction du contexte. De plus, à l'implémentation, lorsque l'on se place dans un contexte distribué, une difficulté supplémentaire est la gestion de la distribution. Cela peut augmenter grandement la durée et la complexité de la réalisation de l'application particulière d'un patron.

À l'aide de nos médiums, il est possible de réaliser dans un médium une version particulière d'un patron, en faisant des choix concernant les noms des services offerts et le fonctionnement précis du patron. Ainsi, cela permet de réutiliser facilement une abstraction de communication (correspondant à une version particulière d'un patron) à l'aide d'un médium et permet d'éviter de concevoir et implémenter plusieurs fois des parties de code très semblables.

Cela dit, il faut néanmoins reconnaître que ce type d'utilisation des patrons s'éloigne de leur principe (un patron étant une forme de solution et non pas une solution unique complètement définie).

1.2.2 Le langage de modélisation UML et les diagrammes de collaboration

Quelques mots sur UML

UML (pour *Unified Modeling Language*) est un langage de modélisation qui doit son origine à trois spécialistes de la modélisation que sont Booch, Rumbaugh et Jacobson qui avaient respectivement développé les méthodologies Booch [24], OMT [104] et OOSE [57]. Forts de leurs expériences passées, ils se sont attachés à la définition d'un processus unifié de développement d'applications basé sur l'approche objet. Mais rapidement, il est apparu qu'avant de spécifier un processus de conception d'application, il fallait développer un langage pour définir les différents artéfacts et modèles créés et manipulés par ce processus. Leurs efforts ont alors été recentrés vers cette problématique et cela a abouti à la définition du langage de modélisation unifié UML, dans sa version 0.8. Ensuite UML a été normalisé

par l'OMG, ce qui a amené à la définition de la version 1.0. Actuellement, la dernière version est la 1.4 [84] et la 2.0 qui devrait mettre plus en avant les aspects de composants logiciels est en cours de définition. La version d'UML utilisée pour les spécifications dans ce document est la 1.3 [83] qui est très proche de la 1.4.

UML est un langage principalement graphique. Son but est de modéliser différents artéfacts ou diagrammes lors de la conception d'une application. UML est ciblé sur l'approche objet. Il permet de définir 9 types de diagrammes afin de modéliser plusieurs aspects ou points de vue d'une application :

- Statique avec les diagrammes de classes et d'instances,
- Dynamique avec les diagrammes d'interaction (diagrammes de collaboration et de séquence), les diagrammes d'états et les diagrammes d'activités,
- Fonctionnel avec les diagrammes de cas d'utilisation,
- Structurel avec les diagrammes de déploiement et de composants.

En plus de ces différents diagrammes, UML inclut désormais le langage de contraintes OCL (pour *Object Constraint Language*) [117]. Ce langage permet de définir la sémantique d'une opération en définissant une précondition qui doit être valide lors de l'appel de l'opération et une postcondition qui doit être respectée à la fin de l'exécution de l'opération. Il permet aussi de définir sur des objets des invariants qui doivent être respectés en permanence. UML est un langage qui est extensible à l'aide de mécanismes tels que les stéréotypes et les profils. Les stéréotypes permettent de marquer et de préciser des contraintes sur certains éléments qui doivent jouer un rôle particulier. Un profil définit un ensemble de règles et de contraintes permettant de spécifier et décrire un domaine donné. Un modèle UML peut alors être défini pour ce domaine particulier et en suivre les règles.

Le lecteur intéressé par plus de détails et d'informations sur UML et ses différents diagrammes pourra se reporter aux ouvrages [81, 25, 103] ainsi qu'à la norme de l'OMG [84]. Le langage OCL est également défini dans cette norme ainsi que dans [117].

Les diagrammes de collaboration UML

Sans détailler les neuf types de diagrammes UML, nous allons nous intéresser plus particulièrement aux diagrammes de collaboration, car ils sont à la base de notre méthodologie de spécification de médiums comme nous le verrons dans le chapitre suivant. Les diagrammes de collaboration permettent de définir des interactions entre des éléments. Il existe deux types de diagrammes de collaboration, d'après la norme 1.3 d'UML [83] :

- Les diagrammes de collaboration au niveau spécification qui décrivent des interactions entre des classes.
- Les diagrammes de collaboration au niveau instance qui décrivent des interactions entre des objets.

Il existe deux types de diagrammes en UML permettant de représenter des interactions : les diagrammes de collaboration et les diagrammes de séquence. Les premiers insistent plus sur la structure et les liens entre les éléments qui interagissent alors que les seconds s'attachent plus à montrer cette interaction d'un point de vue temporel. Ces deux types de diagrammes peuvent donc décrire une même interaction selon un point plutôt structurel ou plutôt temporel. Curieusement, si les diagrammes de collaboration existent au niveau classe et au niveau instance, les diagrammes de séquence existent uniquement au niveau instance. Il n'y a a priori aucune raison logique pour laquelle les diagrammes

de séquence n'existent pas au niveau classe, mais la norme UML de l'OMG ne les définit pas.

Dans notre méthodologie, les diagrammes de collaboration que nous utilisons sont des diagrammes au niveau spécification. Ces diagrammes ne sont pas très couramment utilisés[3], nous allons donc détailler un exemple simple de leur utilisation. Globalement, ils ressemblent très fortement aux diagrammes de collaboration au niveau instance à l'exception de l'utilisation de classes à la place d'objets.

Exemples de diagrammes UML

Dans cette section, nous détaillons succinctement quelques diagrammes UML, dans un contexte de gestion de compte bancaire.

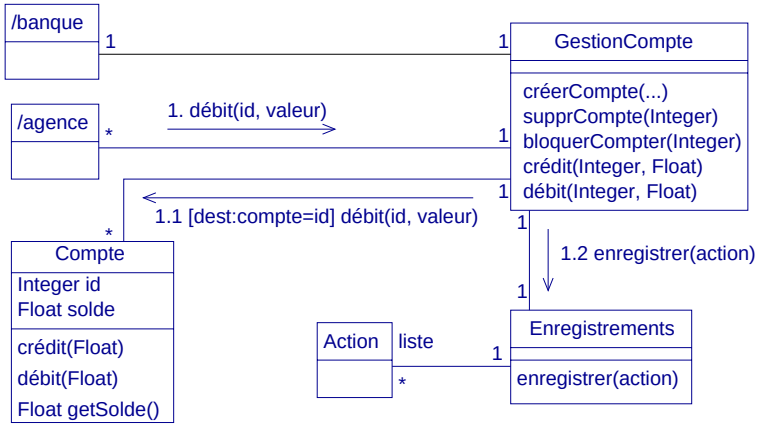

FIG. 1.1: Exemple de collaboration UML au niveau spécification

La figure 1.1 représente un diagramme de collaboration UML au niveau spécification. Dans ce diagramme nous trouvons des éléments identiques à ceux d'un diagramme de classes : les classes `GestionCompte`, `Compte`, `Enregistrements` et `Action` avec leurs attributs et méthodes ainsi que leurs liens entre elles (comme l'association `liste` par exemple). Un diagramme de collaboration sert à définir l'interaction entre un ensemble d'éléments et leurs relations structurelles. Ces éléments peuvent être des classes ou des rôles. Un rôle représente une classe sous une facette donnée. Lors de l'utilisation de la collaboration dans un diagramme de classe ou d'instance, des classes ou objets jouent ces rôles dans la collaboration. Sur notre diagramme, nous pouvons voir deux rôles: `/agence` et `/banque`. Un rôle est représenté par le même symbole qu'une classe et son nom doit commencer par « / ».

[3]En effet, de très nombreux outils de création et de manipulation de diagrammes UML – si ce n'est la totalité – permettent de définir des diagrammes de collaboration au niveau instance mais pas au niveau spécification. Nous n'avons pas trouvé d'explication convainquante à ce choix qui fait que la plupart de ces outils ne sont pas des outils respectant la norme, bien que cela soit pourtant annoncé comme tel par leurs éditeurs. Certes, l'imprécision d'UML et les lacunes dans sa sémantique obligent les éditeurs à faire des choix et à proposer leur propre interprétation de certaines parties d'UML. En ce sens, aucun outil ne peut être totalement compatible par principe avec la norme. Mais pour les diagrammes de collaboration au niveau spécification, il s'agit tout de même d'un type complet de diagramme qui n'est pas supporté.

Pour définir une interaction, il faut préciser les envois de messages entre les rôles et les classes ainsi que leur ordonnancement. Les messages sont représentés par des flèches. Ici, l'interaction représentée correspond uniquement à une opération de débit sur le compte bancaire d'un client. Il est bien sûr possible de définir toutes les autres opérations, comme le crédit, le création et la destruction de comptes avec des interactions. Un message correspond en fait à un appel de méthode. Par défaut, l'appel est synchrone et si plusieurs objets sont destinataires, il se fait sur tous les objets. Il est possible de préciser des sous-ensembles d'objets destinataires ou de réaliser l'appel en mode asynchrone. Notre interaction débute avec le message 1 qui appelle la méthode `debit` sur la classe `GestionCompte`. Cet appel est demandé par le rôle agence. Les paramètres sont l'identificateur du compte et la somme à débiter. Le message 1.1 est ensuite envoyé par la classe `GestionCompte` à la classe `Compte`. L'envoi du message contient une garde exprimée entre crochets. Cette garde précise que le message ne doit être envoyé qu'au compte dont l'identificateur est celui passé en paramètre (l'écriture de cette garde ne suit pas de formalisme particulier). Il est possible également dans les gardes de préciser des conditions d'appel des méthodes, par exemple, du type « `[x > 5]` ». Si la variable `x` contient une valeur supérieure à 5 alors la méthode est appelée sinon elle ne l'est pas. Enfin, le message 1.2 définit que la méthode `enregistrer` est appelée sur la classe `Enregistrements`. Elle sert à mémoriser que le débit a été effectué. Les messages 1.1 et 1.2 s'exécutent l'un après l'autre comme leur numérotation le précise. Elle définit également que ce sont des appels imbriqués de la méthode appelée par le message 1 (car numérotés 1.X).

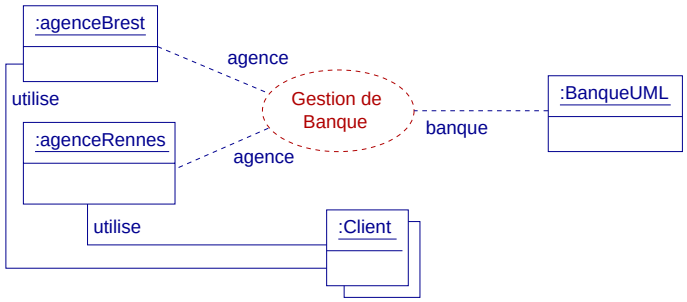

Fig. 1.2: Exemple d'utilisation de collaboration UML

La figure 1.2 est un diagramme d'instance dans lequel la collaboration que nous venons de définir est utilisée. L'ellipse nommée `Gestion de banque` représente cette collaboration. Lors de l'utilisation d'une collaboration, il faut lier ses rôles à des objets ou des classes du diagramme dans lequel elle est utilisée. Dans notre cas, les deux objets `agenceBrest` et `agenceRennes` jouent le rôle d'agence (noté sur le lien entre la collaboration et l'objet). L'objet `banqueUML` joue le rôle de banque. Ce diagramme d'instance spécifie donc que ces trois objets interagissent pour gérer des comptes bancaires (en créant et supprimant des comptes pour la banque et en accédant aux comptes pour les agences).

Dans un diagramme de classe ou d'instance, l'application d'une collaboration revient à intégrer tous les éléments définis par cette collaboration dans le diagramme où elle est utilisée. Notre diagramme d'instance serait donc équivalent à un diagramme d'instance dans lequel les deux objets agences seraient connectés à une instance d'une classe `GestionCompte` qui gérerait un ensemble de comptes bancaires et d'enregistrements d'actions.

À partir de cette collaboration, nous pouvons montrer un exemple simple d'utilisation d'OCL. Pour un compte bancaire, la spécification de l'opération `debit` est la suivante :

context Compte::débit(Float somme)
pre: somme > 0
post: solde = solde@pre - somme

La précondition (définie par `pre:`) précise que la somme passée en paramètre doit être positive pour que l'opération puisse être appelée. Cette contrainte doit être assurée par le client, par l'objet qui appelle l'opération. La postcondition (définie par `post:`) indique qu'à la fin de l'exécution de l'opération, l'attribut `solde` doit avoir une valeur égale à sa valeur avant l'exécution de l'opération (référencée par `@pre`) moins la somme passée en paramètre. Le respect de la postcondition doit être assurée par l'objet qui exécute l'opération.

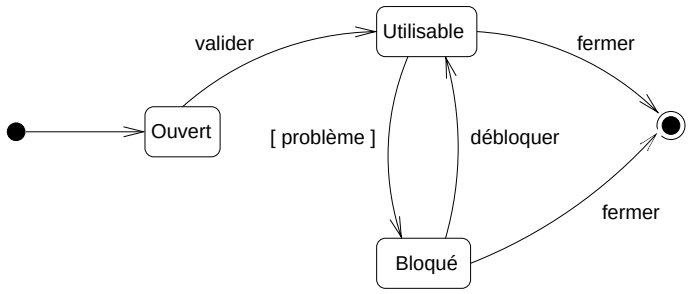

FIG. 1.3: Exemple de diagramme d'états UML

La figure 1.3 est un diagramme d'états représentant l'évolution d'un compte bancaire. Un diagramme d'états est constitué de deux types éléments : les états et les transitions permettant de passer d'un état à un autre. Les transitions sont déclenchées lorsque des événements ou des actions se produisent. Il est possible de mettre des gardes sur les transitions de ces états. Pour un compte en banque, dès sa création (représenté par le pseudo-état correspondant à un gros point), il passe dans un état `Ouvert`. Ensuite, lorsque il est validé, il passe dans un état `Utilisable`. Dès qu'un problème survient (représenté par la transition avec la garde `[ problème ]`), il passe dans un état `bloqué`. Il redevient utilisable si l'action `débloquer` est réalisée. Enfin, lorsqu'il est fermé, il passe dans le pseudo-état de terminaison (correspondant à un gros point entouré d'un cercle).

Pourquoi UML ?

Nous avons choisi d'utiliser UML pour spécifier nos composants de communication car UML s'est imposé comme la référence en modélisation et génie logiciel. Ce standard de fait est très utilisé dans l'industrie du logiciel et de nombreux outils permettant de créer, éditer et manipuler des diagrammes UML existent actuellement sur le marché. Ce choix n'est donc pas lié au fait qu'UML nous semblait être le meilleur langage de spécification dans notre contexte. Au contraire, UML souffre de nombreuses lacunes. Certains éléments ont une sémantique incomplète, ambiguë voire inexistante. Outre ces problèmes de sémantique, UML pose des problèmes de spécification lorsque l'on veut faire apparaître des aspects temporels. UML n'est donc pas le meilleur langage mais le

plus utilisé et celui autour duquel un consensus s'est instauré. Voulant nous intégrer dans une démarche de génie logiciel et de modélisation autour des standards actuels, le choix d'UML nous a donc semblé être le plus opportun.

Néanmoins, nous avons essayé de combler certaines lacunes d'UML dans nos spécifications. Notamment en généralisant l'utilisation d'OCL, afin de lier le plus formellement possible les différentes vues et diagrammes d'une spécification de médium. Nous reviendrons sur ce point dans le prochain chapitre.

Les diagrammes de collaboration sont par nature adaptés à la spécification d'abstractions de communication ou d'interaction. Cet outil est donc intéressant dans un processus de réification d'abstractions de communication basé sur UML comme le notre. Ainsi, les collaborations UML sont un élément central de la spécification de médiums comme nous le verrons dans le chapitre suivant.

1.2.3 La méthodologie Catalysis

Catalysis [40] est une méthodologie de conception d'applications à base de composants logiciels. Catalysis définit à la fois des techniques de modélisation et un processus complet de développement d'applications qui commence par la découverte des concepts métiers et va jusqu'à l'implémentation. Les modèles de Catalysis sont basés sur UML mais s'éloignent tout de même du standard. Les auteurs de Catalysis ont participé à la définition de la norme UML 1.0.

Principes fondamentaux de Catalysis

Catalysis s'appuie sur trois outils de modélisation :

- Les collaborations entre un groupe d'objets. Elles permettent de spécifier des interactions et le comportement d'un groupe d'objets.
- Les types qui servent à définir le comportement d'un objet d'un point de vue externe et qui spécifient la sémantique de l'objet en décrivant un modèle abstrait, sans s'intéresser à son implémentation.
- Le raffinement qui est un mécanisme permettant de passer d'un niveau de description à un autre niveau de description moins abstrait, c'est-à-dire en apportant plus de précisions et de détails qui étaient cachés au niveau précédent.

Ces trois outils de modélisation sont à la base de la définition des différents modèles que Catalysis propose de spécifier. En plus de ces outils, Catalysis met en avant l'utilisation de *frameworks* (ou canevas d'application en français[4]). Un framework est un modèle « incomplet » qui présente un type ou patron d'interaction entre un ensemble d'objets. Le but est de pouvoir réutiliser ces patrons de modèles et de conception dans différents contextes. L'utilisation d'un framework se fait en liant certains éléments ou rôles du framework à des éléments du modèle dans lequel il est utilisé. Dans ce modèle, tous les éléments du framework sont donc intégrés (Catalysis parle de « dépliage » du framework) de manière logique, même si visuellement cela n'est pas visible sur le modèle (où seul le symbole représentant le framework avec ses liens vers des éléments du modèle apparaît). L'application d'un framework se fait de manière similaire à l'utilisation des collaborations UML comme nous l'avons vu dans la section précédente.

[4]Le terme de canevas d'application ne nous semble pas très approprié. Nous préférons donc utiliser le terme anglais framework, qui est plus « parlant ».

En ce qui concerne le processus de développement, les modèles de Catalysis concernent trois niveaux de modélisation :

- Métier ou domaine : il s'agit de spécifier les concepts et règles métier. Pour cela, il faut notamment décrire l'usage que font les utilisateurs du système à construire.
- La spécification des composants : il faut définir les composants qui forment l'application, les services qu'ils offrent et requièrent et leur sémantique. Il s'agit de définir les frontières des éléments qui forment l'application mais sans décrire comment chaque élément est construit.
- L'implémentation des composants : pour chaque composant, il faut définir son fonctionnement interne, les objets qui le composent et comment son implémentation est réalisée. C'est l'intérieur du composant qui est donc traité.

Pour le processus de développement en lui-même, de manière générale, le développement complet d'une application comporte 5 phases, qui recoupent les 3 niveaux de modélisation que nous venons de citer :

- Modélisation du métier : description des concepts et règles métiers.
- Spécification des besoins : définition de ce que le logiciel doit faire.
- Conception des composants : définition à haut niveau des composants et de leurs collaborations et interactions.
- Conception des objets : description du fonctionnement interne des composants à un niveau d'implémentation.
- Architecture de jeu de composants : description d'éléments communs à une famille de composants, afin notamment d'aider à la réalisation de composants d'une même ligne de produit.

Pour terminer, Catalysis met en avant trois grands principes lors de l'application de la méthodologie et du processus de développement :

- L'abstraction qui consiste à cacher des éléments, des parties ou des détails de conception qui n'ont pas d'intérêt à un niveau donné afin de se focaliser sur les aspects principaux de ce niveau. L'abstraction est essentielle pour pouvoir bien gérer la complexité. L'abstraction va de paire avec le raffinement.
- La précision : il s'agit d'éviter de se retrouver avec des modèles sémantiquement incomplets ou non-entièrement définis. Ou du moins, si la sémantique ne peut être clairement définie, d'en être conscient et de bien faire apparaître l'inconsistance d'un modèle. La précision n'est pas l'opposée de l'abstraction dans le sens où l'abstraction ne correspond pas à quelque chose de flou mais à quelque chose dont tout n'est pas montré. L'abstraction et la précision ne sont donc pas contradictoires.
- Des éléments connectables : des éléments logiciels de tous niveaux (conception, implémentation, etc.) qui sont conçus pour être assemblés entre eux. Le but est de favoriser la réutilisabilité de ces éléments et de forcer à spécifier précisément les interfaces d'utilisation de ces éléments afin de bien ou de mieux pouvoir les utiliser.

Les collaborations et les connecteurs dans Catalysis

Catalysis est une méthodologie orientée composants qui propose de définir l'architecture d'une application à l'aide de composants et de connecteurs. Un connecteur est un élément logiciel qui intègre un protocole de communication ou un schéma d'interaction entre un ensemble de rôles. Dans Catalysis, un connecteur connecte des ports de composants afin d'unir ces composants comme un composant plus complexe ou un produit

logiciel. Un port est une interface qui permet d'accéder aux services d'un composant. Un connecteur impose des contraintes sur les rôles joués par les ports qu'il relie. Un connecteur peut être le raffinement ou la réalisation d'une interaction qui est spécifiée par un framework de collaboration. Un connecteur peut avoir plusieurs implémentations, notamment pour réaliser la même interaction, mais qui manipule plusieurs types de données différents (comme des messages ou des flux), ce qui nécessite de réaliser une implémentation spécifique à chaque type de données même si l'interaction est dans son principe la même.

Une collaboration décrit les interactions entre un ensemble d'objets ou de composants. Elle décrit les échanges de messages entre ce groupe d'objets pour réaliser l'interaction ou la fonction désirée. Les frameworks dans Catalysis sont des pièces de conception réutilisables. Un framework de collaboration est donc une collaboration conçue pour être réutilisée et permet de réutiliser une interaction lors de la conception de différentes applications.

Analyse de cette approche

Catalysis se focalise beaucoup sur la spécification et la description des interactions entre les objets ou composants lors de la spécification d'une application. La définition des collaborations entre les objets est un élément clé de cette approche. Pour cela, Catalysis définit des diagrammes de collaboration et des frameworks de collaboration. Ces derniers servent à la spécification de connecteurs. Le raffinement est également un des points essentiels de cette approche. Pour finir, grâce aux frameworks, Catalysis offre un outil puissant pour réutiliser des éléments de conception de tout type, notamment des spécifications de collaboration ou d'interaction. Au delà de ces aspects de conception et de méthodologie, Catalysis définit un processus complet de développement d'application qui part de la découverte des concepts métier et qui va jusqu'à l'implémentation. Catalysis semble donc pouvoir offrir tous les outils et techniques pour réifier des abstractions de communication pendant tout le cycle de développement du logiciel.

Nous pouvons tout de même noter deux limitations. La première est que Catalysis définit les collaborations ainsi qu'un processus de raffinement dans un contexte global et général de spécification d'applications à base de composants. À ce titre, la manière exacte de réifier, de manipuler, de spécifier et d'implémenter les abstractions de communication est laissée à la discrétion et aux choix du concepteur. Si Catalysis offre les outils nécessaires à ces opérations, il ne propose pas un processus bien précis qui aiderait de manière automatique ou guidée le concepteur à raffiner une spécification abstraite en une ou plusieurs spécifications d'implémentation. Nous proposons de lever cette limitation en définissant un processus de spécification spécialisé dont le but est de réifier des abstractions de communication dans des composants logiciels (des médiums). Nous proposons également une architecture de déploiement de médium. À partir de là, notre processus de raffinement est spécialisé pour cette méthodologie et le but final de ce processus est bien connu, à savoir spécifier une ou plusieurs implémentations d'un médium en respectant notre architecture de déploiement. Les étapes et les transformations de notre processus de raffinement sont en conséquence pour certaines complètement automatisables. Pour celles qui ne le sont pas, nous précisons clairement le but et l'objectif de ces transformations. Notez que comme Catalysis est très général et offre tous les outils et mécanismes nécessaires, notre méthodologie de spécification et notre processus de raffinement peuvent

être entièrement réalisés en utilisant Catalysis.

La deuxième limitation concerne l'implémentation. Si les frameworks de collaboration de Catalysis permettent de spécifier n'importe quel connecteur, indépendamment de sa complexité, Catalysis parle très peu de leur implémentation. Dans la partie décrivant le processus de développement, Catalysis consacre un chapitre sur l'implémentation des composants (chapitre 16). Ce chapitre contient des guides, des recommandations ou des patrons pour passer de la spécification à l'implémentation. Dans ce chapitre, il n'est jamais directement question de connecteurs. En fait, dès que l'aspect distribution ou communication à distance est évoqué, Catalysis parle d'intergiciels (comme RMI ou CORBA) ou de requêtes SQL pour définir le mode d'interaction entre deux ou plusieurs composants. Ainsi, à l'implémentation, Catalysis ne consacre pas un patron où quelques recommandations sur les connecteurs réifiant une interaction complexe (plus que du RPC ou une requête SQL qui d'un point de vue abstrait sont de relativement bas niveau). Il ne nous semble donc pas que Catalysis insiste sur la nécessité d'avoir des connecteurs complexes à l'implémentation. Un des points importants de notre approche est au contraire d'offrir la possibilité d'utiliser et de réaliser des abstractions d'interaction complexes et de toute nature au niveau de l'implémentation.

1.2.4 La méthodologie UML Components

UML Components [36] est un processus de spécification d'applications à base de composants logiciels qui part de la définition des règles, besoins et informations métiers pour identifier et spécifier les composants qui forment une application. Leur processus de développement est dérivé de celui de Catalysis en étant à la fois plus simple et moins généraliste. UML Components peut être vu comme une simplification et une spécialisation de Catalysis. En ce sens, il définit un processus qui est « plus applicable » que celui de Catalysis grâce à des règles plus claires et précises, là ou Catalysis ne décrit plutôt que des patrons ou de simples recommandations. L'idée d'UML Components pour identifier les composants est de partir des informations métiers et d'y trouver les concepts essentiels qui sont utilisés pour définir des types. Un composant réalise alors un ou plusieurs types. Dans un deuxième temps, l'étude des interactions entre les composants permet de définir précisément les interfaces de services offerts et requis ainsi que la sémantique des composants, c'est-à-dire la responsabilité et la frontière de chaque composant. Enfin, à partir de cette sémantique, la dernière étape consiste à définir une spécification d'implémentation pour les composants.

Malheureusement, les interactions décrites entre les composants ne concernent que les appels de services entre ces composants. UML Components ne parle jamais d'abstractions de communication plus complexes, ni de connecteurs. Si le processus est intéressant car facilement applicable et se plaçant dans un contexte de composants logiciels, il a tout de même laissé de côté tout ce qui pouvait concerner les interactions de haut niveau entre composants et leur manipulation en tant qu'entité de premier ordre.

1.3 La vision langages et modèles

Dans cette section, nous étudions les langages ou modèles spécialisés dans la représentation et la spécification de la communication, de l'interaction ou de la coordination.

Tout d'abord nous détaillons les langages de description d'architecture en nous intéressant plus particulièrement à la notion de connecteur. Ensuite nous décrivons les langages et modèles de coordination puis les systèmes multi-agents.

1.3.1 Les langages de description d'architecture et les connecteurs

Il n'existe pas vraiment de consensus sur la définition des langages de description d'architecture – ou ADL (pour *Architecture Description Language* en anglais) – ou sur l'architecture logicielle de manière plus générale. De nombreux ADLs existent et bien qu'ayant des éléments et des problématiques en commun, ils n'adressent pas tous les mêmes problèmes et ne les traitent pas de la même manière. Malgré tout, [74] donne une définition des ADLs à partir de la définition de l'architecture logicielle telle qu'on la trouve dans [109] :

> *L'architecture logicielle est une phase de conception qui implique la description des éléments à partir desquels des systèmes sont construits, la description des interactions entre ces éléments, la description de patrons qui guident leur composition et la description de contraintes sur ces patrons.*

Les ADLs sont alors des langages utilisés pour modéliser l'architecture logicielle d'un système. Cette modélisation est faite à un niveau de conception. Le but des ADLs est de raisonner à un niveau plus abstrait que l'implémentation et de définir les principaux éléments d'un système et leurs interactions. Cette abstraction permet au concepteur d'une application de mieux appréhender la complexité de celle-ci, de mieux concevoir le fonctionnement et le découpage de son application ainsi que les inter-connexions entre les différents éléments la constituant. Un des principaux buts des ADLs est en effet de bien séparer les aspects de spécification et d'implémentation.

[74] est une étude approfondie sur les principaux ADLs. Le but de cette étude est de définir les éléments récurrents et essentiels pour qu'un langage puisse être considéré comme un ADL. Cette étude définit donc un certain nombre de concepts principaux et décrit comment ils sont définis ou utilisés dans chacun des ADLs. La description d'une architecture fait intervenir trois éléments primordiaux :

Les composants : un composant est une unité de calcul ou de stockage de données (voire les deux). À un composant est associée une sémantique définissant le fonctionnement du composant, ce que fait ce composant. Un composant est typé et possède des ports permettant d'interagir avec lui.

Les connecteurs : un connecteur est une unité qui modélise des interactions entre des composants. Un connecteur est défini également par une sémantique. Pour [106], la sémantique d'un connecteur correspond à un protocole. Un connecteur est aussi typé et possède également des ports pour interagir avec lui.

Les configurations : les configurations d'architecture ou topologies sont des graphes connectés de composants et de connecteurs et servent à décrire l'architecture globale d'un système.

Définir l'architecture d'un système consiste donc à définir les différents composants et connecteurs utilisés dans ce système ainsi que la topologie de leurs inter-connexions. La spécification de la sémantique d'un composant ou d'un connecteur peut beaucoup varier selon l'ADL utilisé. Certains se basent sur des langages formels tel que CSP [55] alors que d'autres ne définissent ou ne préconisent aucun langage particulier. Les buts

recherchés par les ADLs ne sont pas non plus toujours les mêmes, certains s'intéressent plus particulièrement à la sémantique des composants et connecteurs alors que d'autres s'attachent plutôt à définir les inter-connexions de composants et de connecteurs.

Les ADLs ne se contentent pas de représenter l'architecture d'un système à un niveau abstrait ou conceptuel. Via des outils, beaucoup d'ADLs intègrent des mécanismes de raffinement et de génération de code. Certains ADLs disposent même d'un support d'exécution. Là encore, chaque ADL à ses propres techniques et certains intègrent peu voire pas de fonctionnalités de ce type.

Dans les ADLs, la réification d'abstractions d'interaction se fait naturellement via les connecteurs. Nous parlons plus en détail de ces connecteurs dans la section suivante. En terme d'abstractions d'interaction, les connecteurs permettent a priori de réifier tout type d'interaction indépendamment de sa complexité. Un connecteur ne se limite donc pas seulement aux classiques appels de procédure à distance ou diffusion de message. Par exemple, il est possible de définir des connecteurs pour de la diffusion asynchrone d'événements, des transactions avec une base de données, de la diffusion de flux, etc.

Les connecteurs dans les ADLs

En architecture logicielle, de nombreux travaux tel que [106] insistent depuis longtemps sur la nécessité et l'importance de la représentation des interactions entre les composants via des connecteurs. Malgré tout, le traitement et l'intérêt portés aux connecteurs varient beaucoup selon les ADLs.

La définition suivante des connecteurs est tirée de [109] :

> *Les connecteurs gèrent les interactions entre les composants, ils établissent les règles qui gouvernent les interactions entre les composants et spécifient tout mécanisme supplémentaire requis dans ce but.*

D'après [74], un connecteur peut être défini par six caractéristiques ou propriétés :

- Une interface afin de définir les points d'interactions entre les composants et le connecteur et les services que doivent lui offrir les composants. Ces interfaces sont généralement des rôles joués par des composants participant à l'interaction.
- Un ou plusieurs types qui définissent des primitives d'interaction ou de communication entre des composants (telles que de l'appel de procédure à distance ou la diffusion par exemple).
- Une sémantique qui spécifie le protocole ou l'abstraction d'interaction réifiée dans le connecteur.
- Des contraintes sur le connecteur et sur ses conditions de fonctionnement (par exemple, des contraintes sur la multiplicité de connexion des composants).
- Des capacités d'évolution permettant de modifier les propriétés du connecteur, comme ses interfaces ou sa sémantique pour, par exemple, prendre en compte et s'adapter aux modifications de composants.
- Des contraintes non-fonctionnelles qui ne sont pas dérivables de la sémantique du connecteur et qui sont utiles dans des buts de simulation ou d'analyse.

Sans vouloir être exhaustif, nous allons nous intéresser à la définition et à l'utilisation de la notion de connecteur dans quelques ADLs représentatifs. Le lecteur trouvera une description de ces ADLs dans [97].

Dans Unicon [119, 120, 107, 108] (qui signifie *Universal Connector Support*), un connecteur est défini par un protocole qui spécifie les inter-connexions entre des com-

posants et des rôles qui sont joués par ces composants. Le protocole est défini par un type de connecteur. Malheureusement, Unicon fournit un ensemble de types de connecteurs prédéfini et non-extensible ce qui limite la possibilité de spécifier n'importe quelle forme d'interaction entre composants. Il est cependant possible de composer des connecteurs entre eux pour former un connecteur composite. Unicon permet de générer le code correspondant à un connecteur mais uniquement dans le cas d'un connecteur primitif en utilisant le code prédéfini par cet ADL.

Aesop [47, 48] utilise des connecteurs assez proches de Unicon avec là aussi la notion de rôles joués par des composants dans une interaction définie. Les connecteurs peuvent également être le résultat de la composition de connecteurs. Le problème est que s'il est possible de définir de nouveaux connecteurs, il n'existe pas de langage de spécification pour définir leur sémantique de manière précise.

Darwin [65, 66] et Rapide [63, 64, 51] sont deux ADLs qui ne proposent pas de connecteurs explicites comme dans Unicon ou Aesop. Ces ADLs permettent de définir précisément comment un groupe de composants inter-agissent et les règles de connexion entre ces composants. Mais ils n'offrent pas la possibilité d'intégrer ces règles et contraintes dans un élément typé et aux interfaces définies comme un connecteur. En fait, ces deux approches permettent de définir des instances de connecteur mais pas des types de connecteur. Il n'est pas donc possible de réutiliser ces connecteurs dans différentes architectures d'application. C'est en ce sens que ces approches ne proposent pas de vrais connecteurs. Néanmoins, ces deux approches proposent de réaliser des connexions complexes dans des « composants connecteurs ».

Wright [4, 5] est un ADL qui intègre la notion de connecteur explicite. Un connecteur est composé de deux parties : les rôles et une glu. Un rôle représente un composant qui participe à l'interaction définie par le connecteur. Wright utilise le langage CSP [55] pour spécifier un rôle. Ensuite, la glu définit les liens et les interactions entre les rôles. Wright est uniquement un langage de spécification et n'offre pas de support d'exécution comme Darwin, ni de générateur de code.

Dans C2 [71, 72, 73], les connecteurs permettent de gérer les échanges de messages entre les composants. Un connecteur définit des règles de transmission et de filtrage de messages entre les composants émetteurs et les composants récepteurs. Ce type de connecteur ne permet donc pas de définir n'importe quelle abstraction d'interaction (par exemple celles ayant un état ou un espace de données partagées).

Olan [18, 16, 17] permet d'utiliser différents types de connecteur mais ne fournit pas de formalisme pour les spécifier. Les connecteurs sont accessibles à l'implémentation lors de l'utilisation du support d'exécution d'Olan.

Résumé des propriétés des connecteurs

La notion de connecteur n'apparaît donc pas dans tous les ADLs même s'ils sont nombreux à l'intégrer. Parmi ceux-là, les connecteurs prennent souvent la même forme avec un protocole qui définit une interaction entre un ensemble de rôles de composants. La spécification de la sémantique du connecteur se fait soit grâce à un langage formel ou dans un autre formalisme moins formel, voire sans formalisme prédéfini.

Le support du raffinement dans les ADLs est très souvent limité. En général, il consiste à générer directement le code applicatif à partir de la spécification d'une architecture et des ses éléments. Quelques ADLs offrent par contre des fonctionnalités plus évoluées de

raffinement et de traçabilité autorisant plusieurs niveaux d'abstraction et de spécification.

Les connecteurs dans les ADLs permettent donc de définir une abstraction de communication, principalement pendant l'étape de spécification abstraite mais aussi à l'implémentation. Un des buts des ADLs est de séparer clairement le niveau de spécification de celui de l'implémentation, que cela concerne les composants ou les connecteurs.

Analyse de cette approche

Si les connecteurs se présentent donc a priori comme une bonne solution pour réifier des abstractions d'interaction, en pratique cela est à modérer, notamment en ce qui concerne la phase d'implémentation. [75] explique en effet que dans certains ADLs, un connecteur n'existe qu'au niveau de la conception et a disparu au niveau de l'implémentation. Dans ce cas, il est moins naturel de vouloir spécifier des connecteurs au niveau de la spécification si l'on sait que l'on ne pourra plus les manipuler à l'implémentation.

Certains ADLs restreignent le nombre ou le type de connecteurs. Dans ce cas, le concepteur doit se contenter d'un ensemble prédéfini et figé de connecteurs. Malgré tout, certains ADLs de ce type autorisent la composition de connecteurs. Mais globalement, ces ADLs ne permettent pas via les connecteurs de réifier facilement n'importe quelle abstraction d'interaction.

Enfin, la présentation de type « *boxes and lines* », souvent donnée pour introduire l'architecture logicielle, pose un problème plus général. Dans cette vision, les boîtes décrivent les composants et les lignes les connecteurs. La liaison des boîtes par des lignes permet de décrire l'architecture du système considéré. Si abstraitement et intuitivement une boîte peut a priori contenir plein de choses, ça n'est pas le cas d'une ligne. Cette vision ou représentation du connecteur en tant qu'élément simple (pour ne pas dire simpliste) de liaison point-à-point va peut-être amener inconsciemment les concepteurs à ne pas chercher à construire des connecteurs beaucoup plus complexes que, par exemple, de l'appel de procédure à distance. En restant dans ce type de description graphique d'architecture, nous proposons de passer à un style *« boxes, lines and ellipses »*. Les ellipses sont nos médiums et sont donc intuitivement potentiellement plus complexes, par rapport à un simple trait.

Pour résumer, [75] précise que le support des connecteurs dans les ADLs est bien moins important que celui des composants. Nous pensons que les connecteurs ou les entités réifiant des abstractions de communication doivent être traitées à un niveau équivalent à celui des composants. À ce titre, nous réifions ces abstractions dans des composants logiciels. Ces derniers ont une architecture particulière afin d'être bien adaptés à la réalisation de tout type d'abstraction de communication.

Concernant le raffinement, rares sont les ADLs offrant de vraies fonctionnalités de ce type. Bien souvent le raffinement se limite à générer l'implémentation d'une architecture à partir du niveau conceptuel. Le problème de cette approche est de ne pas offrir la possibilité de faire des spécifications de niveaux intermédiaires et de forcer l'architecture de conception à ressembler à l'architecture d'implémentation. Cela revient en pratique à limiter le niveau d'abstraction lors de la conception, car il faut raisonner en pensant directement à l'architecture d'implémentation. Cela interdit aussi la possibilité de définir plusieurs variantes d'implémentation pour une même spécification. Nous proposons au contraire plusieurs niveaux intermédiaires à partir d'une spécification abstraite. De plus, un des points essentiels de notre approche est qu'il est possible d'obtenir plusieurs spéci-

fications d'implémentation et plusieurs implémentations à partir d'une seule spécification abstraite.

1.3.2 Les langages et modèles de coordination

La notion de coordination regroupe un nombre important de travaux. [92] est une étude qui décrit et compare les langages ou modèles de coordination les plus courants. Tout comme pour les ADLs, la coordination concerne elle aussi des outils et langages différents et ne s'intéressant pas forcément aux mêmes problématiques ou domaines. De ce fait, il n'existe pas un langage ou modèle unique qui couvrirait tous les aspects de la coordination. Et là encore, il n'existe pas de définition précise de ce qu'est la coordination. Néanmoins, [92] reprend la définition de [34] comme étant la plus largement répandue :

> *La coordination est le processus de création de programmes en collant ensemble des éléments actifs.*

Un modèle de coordination est alors la spécification de la glu qui relie des activités séparées. Un langage de coordination est un langage qui permet de définir formellement cette glu et qui offre des primitives ou opérations de synchronisation et de communication entre des activités, ainsi que de création et de destruction de celles-ci. Pour rejoindre ce que nous avons vu avec les ADLs, le langage de coordination peut être vu comme un moyen de définir des configurations ou des assemblage d'activités (pour les ADLs, il s'agit d'assemblage de composants). Ainsi, là encore nous retrouvons une séparation des composants et de leurs interactions.

[12] précise également qu'un des buts des langages et modèles de coordination est de diminuer le fossé entre le modèle de coopération défini au niveau de la conception et le modèle de communication de bas niveau utilisé à l'implémentation.

Les différents langages et modèles de coordination

Globalement, les langages et modèles de coordination peuvent être classés en deux grandes familles :

- Les modèles orientés données : la coordination fait intervenir en plus des processus ou activités, les données qui sont utilisées et manipulées par ces processus.
- Les modèles orientés contrôle : la coordination concerne principalement l'ordonnancement ou la synchronisation des processus entre eux. Il y a donc une séparation complète des processus et de leur coordination.

Des descriptions des principaux modèles et langages de coordination se trouvent dans [92, 96]. Nous allons en étudier succinctement quelques-uns.

Les langages et modèles orientés données. Parmi les langages et modèles orientés données, Linda [1, 33] est un des premiers et des plus célèbres modèles de coordination. De nombreux travaux ont porté sur ce modèle et en ont défini de nombreuses variantes. Le modèle de base de Linda est très simple. Il consiste en une mémoire partagée et trois primitives d'accès aux données de cette mémoire. Celle-ci contient des tuples qui peuvent avoir une arité et un contenu quelconques. La mémoire partagée est donc également appelée espace des tuples. Un tuple est une séquence de champs, qui peuvent soit contenir une valeur, soit contenir une variable. Le mécanisme d'interaction de base est l'appariement (*pattern-matching* en anglais) avec un tuple de l'espace des tuples. Un tuple T1

est apparié avec un tuple T2 si T1 possède les mêmes valeurs que T2 et aux mêmes emplacements. Les autres champs de T1 sont des variables. Les 3 primitives d'accès aux tuples sont les suivantes :

- `in(t)` : le processus évalue le tuple `t` (c'est à dire calcule les valeurs contenues dans le tuple) puis cherche un tuple `t'` dans l'espace des tuples pour lequel `t` est apparié avec `t'`. Si il en trouve un, il l'enlève de l'espace et affecte aux variables de `t` les valeurs de `t'` se trouvant aux mêmes emplacements. Sinon le processus reste bloqué tant qu'il n'y a pas de tuple disponible que `t` apparie.
- `read(t)` : possède le même comportement que `in` sauf qu'il ne supprime pas le tuple de l'espace.
- `out(t)` : évalue `t` puis l'ajoute à l'espace. Cette primitive est bloquante tant que le tuple n'a pas été ajouté.

Il existe une quatrième primitive `eval()` permettant de lancer un processus.

De nombreux travaux ont porté sur Linda et des extensions ou des variantes de son modèle de coordination. Par exemple, Bauhaus Linda [35] permet l'utilisation de multiples espaces de tuples. Ce modèle définit la notion de multi-ensemble qui peut contenir des tuples ou d'autres multi-ensembles. Ainsi, il est possible d'avoir une hiérarchie d'espaces de tuples. Objective Linda [60] supporte aussi les hiérarchies d'espaces de tuples. Cette approche définit une variante de Linda dans un contexte objet et de système distribué ouvert. Par exemple, la primitive `in` a la signature suivante : `multiset in(oil_object *o, int min, int max, double timeout)`. L'appel de cette primitive essaye de retirer de l'ensemble des objets appariés avec le patron référencé par l'objet `o`. Pour que l'appel soit valide, il faut qu'au moins `min` objets soient retirés en moins d'un temps `timeout`. Au plus, `max` objets sont retirés. Si l'appel n'est pas valide, aucun objet n'est retiré et la primitive retourne `null` au lieu d'un ensemble d'objets. Ces contraintes de temps de recherche et de taille d'ensemble sont utiles dans le cadre des systèmes distribués ouverts pour lesquels aucune hypothèse sur les vitesses d'exécutions et les temps de communication ne peut être faite.

Le modèle conceptuel de Linda et ses extensions ont été implémentés dans plusieurs langages. Généralement, cela se fait en rendant les primitives du modèle accessibles à un programmeur lorsqu'il développe dans un certain langage. Certaines extensions de Linda ont été définies pour pouvoir offrir de meilleures performances en terme de vitesse d'exécution (c'est le cas de Bonita [102] par exemple) ou gérer différents contextes d'utilisation (comme Objective Linda avec les systèmes distribués ouverts).

Il existe de nombreux autres modèles orientés données, qu'ils soient inspirés de Linda ou non. La caractéristique commune de la grande majorité de ces modèles est un espace de données partagées par les processus ou composants qui sont coordonnés.

Les langages et modèles orientés contrôle. Dans les langages ou modèles orientés contrôle, chaque processus est vu comme une boîte noire qui précise ses interfaces d'interaction avec les autres processus en décrivant ses ports d'entrées et de sorties. La coordination consiste à définir les règles et propriétés d'inter-connexions de ces ports, c'est-à-dire d'inter-connexions des processus. Les connexions entre ports sont généralement de type point-à-point (à travers un canal d'événements ou des flux) et fonctionnent en mode producteur/consommateur. Dans certains modèles, la diffusion est également disponible.

Parmi ces langages orientés contrôle, un bon nombre d'entre eux sont des langages dits « de configuration ». Chaque processus décrit ses ports d'entrées et de sorties et la coordination consiste à décrire la configuration des connexions entre les différents ports à l'aide du langage ou du formalisme offert. Les processus et leur coordination sont donc clairement séparés. Néanmoins, la coordination se limite uniquement à l'expression des connexions entre les processus, c'est-à-dire à lier un service d'un port requis à celui d'un port offert, sans traitement supplémentaire. Il est rarement possible de réifier des abstractions de coordination pouvant avoir un état ou gérer des données. Beaucoup de ces modèles offrent des fonctionnalités de reconfiguration dynamique et d'instantiation de processus ou composants. Ces modèles et langages sont souvent assez proches de l'implémentation, la coordination n'est pas très abstraite. D'ailleurs beaucoup de ces modèles sont utilisés via des extensions de langages courants de programmation comme C, C++ ou Ada. Dans cette catégorie de langages et modèles de configuration, nous pouvons citer notamment PCL [111], Conic [67, 61] ou Durra [13, 14]. Notons également que [92] considère que les ADLs Darwin et Rapide (dont nous avons parlé dans la section précédente) font partie de cette catégorie de langages de configuration.

Manifold [10, 11, 9] est un modèle de coordination qui est basé sur quatres notions : les processus, les ports, les événements et les canaux. Les canaux connectent des ports de processus et servent à faire transiter des événements. Une des particularités de ce modèle est de ne pas distinguer les processus de calcul et de coordination. Ainsi, un processus de coordination peut coordonner d'autres processus de coordination en plus de processus de calculs. Il est donc possible de réaliser une forme de méta-coordination, ce qui est rarement possible avec d'autres approches.

[7, 8] est une approche qui propose d'associer des contrats aux interactions (appels de services) entre objets. Le principe est qu'en fonction du contexte, on modifie la sémantique et les règles d'appel d'un service à l'aide d'un contrat associé à ce service. Cette approche est dans le principe assez proche de celle de la programmation par aspect interactionnelle dont nous avons parlé dans la section 1.1.4 page 23.

La plupart des langages de coordination orientés contrôle définissent la coordination en tant que règles et propriétés d'inter-connexions et de traitement des événements pour tous les processus ou composants d'un système complet. Une des évolutions importantes des langages orientés contrôle est de confiner la coordination d'un système dans plusieurs éléments. Chaque élément exprime la coordination d'un sous-ensemble des composants ou processus et non pas du système global. Cela permet aussi de réutiliser ces modes de coordination dans d'autres systèmes. Dans cette catégorie de modèles, nous pouvons citer par exemple Micado [19, 20] et son langage ISL. Cette approche définit un modèle où la coordination est réifiée dans des gestionnaires d'interaction qui décrivent des règles d'interaction. Un gestionnaire peut être raffiné pour construire d'autres gestionnaires, grâce à l'ajout ou la modification de règles. Cela facilite la réutilisation et l'adaptation des gestionnaires et donc d'abstractions d'interaction. FLO/C [52, 41] est un autre modèle qui met également en avant la réification de règles de coordination dans des connecteurs explicites. Il est donc possible d'utiliser plusieurs connecteurs dans une application afin de coordonner des groupes d'objets actifs. Il permet également de gérer la dynamique de création de connecteurs et de connexion ou déconnexion d'objets sur ces connecteurs.

Analyse de cette approche

Les langages de coordination peuvent donc également servir à réifier des abstractions d'interaction. Mais il ne sont pas tous très bien adaptés pour cela. Tout d'abord, les langages et modèles orientés données ne conviennent pas pour ce genre de tâches. Si l'on prend un modèle comme Linda par exemple, il ne permet pas de définir tout type d'interaction entre composants mais représente en fait une abstraction d'interaction particulière. Chaque modèle orienté données est donc généralement la réification d'une et une seule abstraction d'interaction. Les modèles et langages orientés contrôle n'ont pas cette restriction et peuvent a priori servir à réifier n'importe quelle abstraction. Mais ils n'offrent pas tous de notion de connecteur qui intègre et confine une abstraction donnée. En fait, beaucoup d'entre eux servent à décrire la configuration globale d'une application ou d'un système. Certains modèles de coordination tels que Darwin et Rapide sont également considérés comme des ADLs. Ce qu'il est intéressant de noter, c'est que ces deux ADLs font partie des rares ADLs n'ayant pas de notion de connecteur explicite. Les langages orientés contrôle sont donc principalement utilisés pour définir des configurations de composants ou de processus mais sans se soucier d'intégrer ces configurations dans des éléments réutilisables tels que des connecteurs. En ce sens, la réification d'abstractions d'interaction avec ces langages ne nous semble pas être très intéressante. Malgré tout, certains langages récents, comme Micado ou FLO/C par exemple, font apparaître explicitement des unités réutilisables de coordination entre composants. Dans ces cas là, cela rejoint l'approche connecteur des ADLs. Ces unités peuvent également posséder un état et gérer des données, choses que l'on ne retrouve pas dans beaucoup de langages orientés contrôle ce qui rend bien sûr la réification de certaines interactions difficile, par exemple pour un modèle comme Linda où la mémoire partagée fait partie de la coordination à part entière. Il est bien sûr possible de donner la responsabilité de la gestion de cette mémoire à un composant particulier mais cela brise l'abstraction (nous reviendrons sur ce point dans la section 2.1.2 page 51).

Pour ce qui est du processus de raffinement, les langages de coordination permettent pour beaucoup d'entre eux de passer de la spécification à l'implémentation et offrent même bien souvent un support d'exécution pour la coordination (qui peut notamment prendre la forme d'extensions de langages de programmation existants ou de plates-formes de type intergiciel). Mais il n'existe pas vraiment de raffinement, comme nous le proposons dans notre approche, qui permet de définir une interaction à un niveau abstrait, de la raffiner pour obtenir plusieurs spécifications d'implémentation et enfin plusieurs implémentations. Dans la majorité des langages et modèles de coordination orientés contrôle, le raffinement consiste à générer directement ou écrire le code correspondant à un système, ce qui comme nous l'avons vu pour les ADLs, force à manipuler des interactions moins abstraites car très proches de l'implémentation. Beaucoup de modèles sont implémentés comme des extensions ou des bibliothèques pour des langages de programmation courants. Les modèles orientés données reposent moins souvent sur un support d'exécution et il est possible d'implémenter un modèle pour différents langages de programmation ou contextes d'utilisation (dans un cadre distribué ou centralisé par exemple). Mais il n'y a pas de lien de raffinement ou de génération de code automatique pour passer du modèle à une implémentation particulière.

En résumé, la grande majorité des langages ou modèles de coordination ne permet pas de réifier une abstraction de communication tel que nous l'entendons. Les principaux

manques concernent la réification d'abstractions (pouvant avoir un état) dans des éléments réutilisables ainsi que le raffinement qui ne consiste bien souvent qu'a générer le code de la coordination sur une plate-forme d'exécution bien particulière. Certains langages permettent tout de même de réifier des interactions dans des éléments réutilisables, dans ces cas là, la solution se rapproche des connecteurs des ADLs.

1.3.3 Les systèmes multi-agents

Les systèmes multi-agents [43, 82, 118] sont un paradigme de réalisation d'applications distribuées qui s'inspire de la sociologie en représentant des entités informatiques comme des personnes vivant dans un environnement social. Cette approche a également en partie pour origine des travaux sur l'intelligence artificielle. D'après [43], voici la définition d'un agent :

> *Un agent est une entité réelle ou virtuelle, évoluant dans un environnement, capable de le percevoir et d'agir dessus, qui peut communiquer avec d'autres agents, qui exhibe un comportement autonome, lequel peut être vu comme la conséquence de ses connaissances, de ses interactions avec d'autres agents et des buts qu'il poursuit.*

En ce qui concerne la phase d'implémentation, il existe de nombreuses plate-formes permettant de réaliser des applications distribuées à l'aide d'agents.

Dans l'approche multi-agents, un point important concerne les interactions entre les agents. La définition de règles de dialogue ou politiques de communication entre des agents est un domaine de recherche assez vaste, de nombreux travaux portent sur ce sujet. Néanmoins, très peu d'entre eux s'intéressent à réifier ces politiques dans des entités utilisables à l'exécution, ni même à les rendre réutilisables lors de la phase de spécification (néanmoins, parmi les quelques travaux réifiant les interactions entre agents, nous pouvons notamment citer [70]). De manière générale, la définition des règles de dialogue se fait à un niveau très abstrait, comme pour un protocole, sans qu'un lien ne soit fait vers la mise en œuvre de ces règles.

1.4 Conclusion

Nous avons présenté dans ce chapitre, des techniques, outils et méthodologies permettant de réifier et de manipuler des abstractions de communication ou d'interaction. Cela forme un domaine très vaste et très varié qu'il n'est pas très simple de classifier. Nous avons appuyé notre étude sur les niveaux du cycle de développement du logiciel que couvrent ces méthodes. Notre approche, qui consiste à réifier une abstraction d'interaction dans un composant logiciel – un médium –, permet en effet de manipuler cette abstraction de la spécification abstraite jusqu'à l'implémentation et le déploiement. Nous proposons une méthodologie de spécification pour ce niveau abstrait et un processus de raffinement permettant de transformer cette spécification abstraite en une ou plusieurs spécifications d'implémentation qui prennent en compte l'architecture de déploiement de médium que nous avons définie. La plate-forme que nous avons développée aide ensuite à l'implémentation de ces médiums (même si nous ne faisons pas encore de génération automatique de code).

Approche	**Phase de spécification**	**Phase d'implémentation**	**Raffinement ou transformation de la spécification**	**Commentaires**
Protocoles	Spécification abstraite			Uniquement au niveau spécification
Plateformes de composition de protocoles	Spécification de réalisation du protocole	Plate-forme d'exécution	Génération de code	Spécification proche de l'implémentation, de la composition
Intergiciels et plate-formes de composants		Quelques abstractions disponibles		Peu prévu pour étendre les abstractions et en ajouter
Programmation orientée aspect interactionnelle	Spécification abstraite	Plate-forme et langage		Ne permet pas de réifier toutes les abstractions
Patrons de conception	Abstrait, abstraction générale			Pas de passage d'un patron à sa réalisation
Collaborations UML	De tout niveau, au choix du concepteur			Uniquement au niveau spécification
Catalysis	Plusieurs niveaux, de l'abstrait à la spécification d'implémentation	Guide, patrons d'implémentations	Processus de raffinement général, processus de transformation de spécifications	Peu de prise en compte d'abstractions de communication à l'implémentation, processus de transformation très général
UML Components	Plusieurs niveaux, de l'abstrait à la spécification d'implémentation		Processus de transformation de spécifications détaillé et précis	Aucune prise en compte des abstractions de communication
Langages de description d'architecture (ADLs)	Spécification relativement abstraite	Plate-forme d'exécution	Génération de code	Spécification proche de l'implémentation, pas de niveaux de spécifications intermédiaires
Langage de coordination (orienté contrôle)	Spécification relativement abstraite	Plate-forme d'exécution ou extension de langages de programmation	Génération de code	Spécification proche de l'implémentation, pas de niveaux de spécifications intermédiaire
Systèmes multi-agents	Spécification abstraite	Plate-forme d'exécution, d'implémentation		Uniquement au niveau spécification

FIG. 1.4: Principales caractéristiques des différents travaux traitant de la réification d'abstractions de communication

Tout d'abord, en ce qui concerne la réification d'abstractions d'interaction à un niveau abstrait, nous pouvons constater que de très nombreuses méthodes existent. À ce titre, notre approche et notre méthodologie n'ont rien de révolutionnaires. Il s'agit uniquement d'une méthodologie de plus et dans le contexte d'UML. Beaucoup d'approches dont notamment les ADLs définissent et utilisent la notion de connecteur. Un connecteur est un élément logiciel permettant de spécifier des interactions entre composants. Ces connecteurs peuvent bien souvent réifier n'importe quelle abstraction d'interaction et permettent de la réutiliser dans différentes spécifications d'applications ; ce que permettent également nos médiums. Certains langages de coordination définissent aussi des éléments équivalents et ayant des propriétés proches de ces connecteurs. Dans la méthodologie Catalysis, les connecteurs existent aussi et sont spécifiés à l'aide de frameworks de collaboration, assez proches des collaborations UML. La plupart des modèles de connecteurs ou des langages de coordination servent à concevoir des « modes » d'interaction en définissant des connecteurs qui n'ont pas forcément de services offerts ou requis bien spécifiés. Par exemple, un connecteur peut servir à faire de la diffusion. La spécification de ce connecteur définit la sémantique de la diffusion mais sans se préoccuper de la nature des données à diffuser. La diffusion est définie dans un contexte global entre des composants qui jouent soit un rôle d'émetteur, soit un rôle de récepteur. Ensuite, dans un contexte particulier, ces rôles sont liés aux composants désirés et le connecteur est utilisé pour un type particulier de données. Nos médiums ne peuvent pas fonctionner ainsi. En effet, ils offrent forcément une ou plusieurs interfaces de services offerts et requis. Ces services ont chacun une signature bien précise et ils sont typés. Il n'est donc pas possible de définir un médium de diffusion de manière générale, il faut en définir un en particulier pour chaque type de données considéré. En ce sens, nos médiums sont donc plus proches des composants (aux interfaces de services typés) que des connecteurs (ou du moins que de la plupart des types de connecteurs car certains travaux se basent également sur des connecteurs aux interfaces de services typés).

Pour ce qui est de l'implémentation, les choses sont assez différentes. A priori, les abstractions définies au niveau de la spécification abstraite devraient pouvoir se retrouver au niveau de l'implémentation. Si bien souvent, il n'y a aucune contrainte ou restriction à l'implémentation de ces abstractions dans beaucoup d'ADLs (certains posent tout de même des limitations de ce point de vue avec leur support d'implémentation ou d'exécution), le problème se situe plutôt au niveau de l'usage de ces abstractions. En effet, en pratique, les abstractions utilisées ne sont plus aussi complexes et le concepteur a tendance à ne considérer que des abstractions de plus bas niveau comme les classiques appels de procédure à distance, requêtes HTTP ou diffusion d'événements. L'abstraction complexe au niveau de la conception a été diluée ou dispersée entre les différents composants[5] pendant le raffinement menant à l'implémentation. À ce niveau, les connecteurs sont en effet souvent assez proches de l'intergiciel utilisé pour gérer la distribution et la communication entre les composants distants. Cela peut se comprendre car l'une des principales difficultés dans une application distribuée est la gestion de la répartition et donc l'utilisation de ces intergiciels. Cela peut avoir une conséquence plus générale sur les connecteurs : l'usage de connecteurs relativement simples à l'implémentation peut

[5]Les actions ou tâches réalisées par un connecteur et correspondant à une abstraction d'interaction au niveau abstrait, vont être déplacées à l'implémentation vers les composants qui s'acquittent alors de ces actions. En d'autres termes, un connecteur n'existe que conceptuellement et à l'implémentation ce sont les composants qui ont la charge de gérer eux-mêmes leurs communications.

pousser les concepteurs à ne pas spécifier de connecteurs complexes ou plus abstraits à la spécification, et par la même occasion, les amener à ne pas considérer les connecteurs et donc les abstractions de communication comme des éléments essentiels dans l'architecture d'une application, au même titre que les composants. Nous pensons au contraire qu'il est utile et souhaitable d'utiliser des connecteurs de haut niveau et de toute complexité.

Enfin, pour le processus de raffinement, les ADLs ou les langages de coordination sont généralement assez limités de ce point de vue. Généralement, le raffinement consiste à générer du code source à partir de la spécification de la coordination ou du connecteur. Cette approche limite le niveau d'abstraction en forçant l'architecture abstraite à ressembler à l'architecture d'implémentation. De plus, cela ne permet pas d'avoir des niveaux intermédiaires de spécification, ni de pouvoir spécifier des variantes d'implémentation. Les méthodologies et processus de développement comme Catalysis peuvent par contre offrir un vrai processus de raffinement de spécifications. La différence avec notre approche est que ces méthodologies servent à spécifier toutes applications à base de composants (et aussi de connecteurs) et le processus de raffinement qu'elles offrent est donc très général. Le notre est beaucoup plus spécialisé car il est utilisé dans un cadre bien précis et connu et est donc en partie automatisable comme nous le verrons par la suite.

Ainsi, nous pensons que l'originalité de notre approche concerne le fait d'avoir la possibilité et la volonté de réifier une abstraction d'interaction à tous les niveaux du cycle de développement du logiciel, y compris à l'implémentation. Notre architecture de déploiement – bien que très simple – permet a priori d'implémenter tout type d'abstraction et même plusieurs versions d'implémentation de cette abstraction. Cela peut être un apport intéressant pour que les concepteurs « pensent » à utiliser et réutiliser des abstractions de communication ou d'interaction complexes, y compris à l'implémentation. Notre processus de raffinement est spécialisé et adapté à ce contexte particulier et permet donc de faciliter la transition de la spécification abstraite à l'implémentation en gardant la cohérence de l'abstraction pendant tout le déroulement du processus et en offrant la possibilité d'avoir différentes implémentations d'une même abstraction.

Chapitre 2

Les composants de communication ou médiums

Dans ce chapitre nous définissons la notion de composants de communication ou d'interaction que nous appelons aussi *médiums*. Un médium est la réification d'une abstraction de communication ou d'interaction, à tous les niveaux du cycle de vie du logiciel, de la spécification abstraite à l'implémentation. Pour commencer, nous montrons à partir d'exemples simples, qu'il est aisé de manipuler ces abstractions de communication aussi bien à l'implémentation qu'à la spécification d'une application.

Après cette introduction, nous donnons une définition des médiums. Nous détaillons ensuite la méthodologie de spécification en UML de ces composants. Nous donnons trois exemples complets de spécification de médiums qui suivent cette méthodologie. La spécification est faite à un niveau abstrait, indépendamment de tout contexte d'utilisation ou de choix d'implémentation.

Enfin, pour clore ce chapitre, nous définissons l'architecture générale de déploiement des médiums.

2.1 Vers une meilleure utilisation des abstractions de communication

Dans le cadre de la conception et de l'implémentation d'applications à base de composants, un des principes fondamentaux et reconnu est que la communication entre les composants est un élément clé, et cela plus particulièrement dans un contexte distribué. Il n'est cependant pas courant de rencontrer des implémentations de systèmes complexes de communication entre composants. Dans cette section, nous montrons par l'intermédiaire de deux exemples, qu'il est aisé et intéressant en terme d'architecture logicielle d'implémenter et de spécifier des abstractions de communication ou d'interaction de haut niveau. Ces interactions pouvant être réifiées dans des composants de communication, nos médiums. Nous nous intéressons d'abord au niveau de l'implémentation puis traitons celui de la spécification.

2.1.1 Étude de l'implémentation des interactions inter-composants

Nous étudions ici une application de vidéo interactive et nous nous intéressons plus particulièrement à l'implémentation des interactions entre les composants formant cette application.

Dans cette application, un unique composant serveur gère une liste de films. Un film est diffusé par ce serveur et est visualisé par plusieurs composants clients. À la fin de chaque film, le serveur lance une session de vote pour demander aux clients le prochain film qu'ils souhaitent visionner. À la fin du vote (quand tous les clients ont voté ou quand la durée du vote s'est écoulée), le serveur diffuse le film correspondant au choix majoritaire.

Architecture de l'application de vidéo interactive

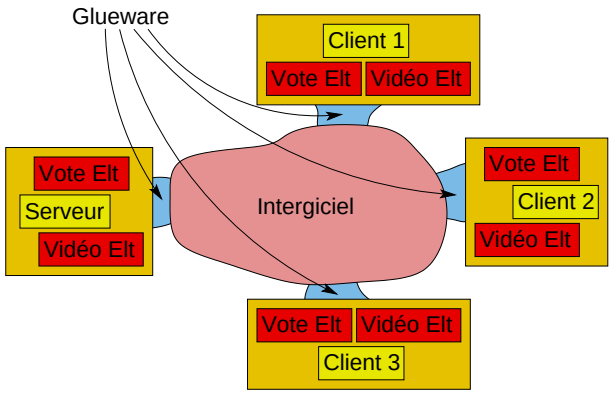

Fig. 2.1: Détail de l'architecture de l'application de vidéo interactive

L'architecture de cette application est représentée sur la figure 2.1, dans le cas où elle est composée de trois clients en plus de l'unique serveur. Les quatre composants communiquent entre eux par l'intermédiaire d'un intergiciel (comme CORBA par exemple) permettant à un composant d'appeler à distance des opérations sur d'autres composants. De la « glu » est nécessaire pour adapter le composant afin qu'il puisse utiliser l'intergiciel. Dans le cas de CORBA, la glu correspond aux talons et aux squelettes qui doivent être générés pour appeler un service à distance. L'ensemble de la glu forme ce que l'on appelle la « glueware ».

La communication entre les composants concerne deux types d'interactions : la diffusion et la réception du flux vidéo ainsi que la gestion des votes. La réalisation de ces deux interactions nécessite des communications entre les composants ainsi que l'exécution de certaines tâches comme l'encodage et le décodage d'un flux vidéo ou la gestion de la durée du vote. Pour cela, chaque composant intègre des éléments pour gérer chacune des interactions (`Vidéo Elt` pour la diffusion de flux et `Vote Elt` pour le vote). Ces éléments sont associés à la partie fonctionnelle ou métier qui constitue le reste du composant.

Étude de cette architecture

Cette manière classique de concevoir des composants dans un contexte distribué pose quelques problèmes. Tout d'abord, les interactions de vote et de diffusion de flux ne sont pas spécifiques à notre application. En effet, de la diffusion de flux vidéo peut aussi être utilisée dans une autre application (comme de la vidéo conférence par exemple). Or, si les concepteurs et les développeurs du système de diffusion de flux vidéo ne l'ont pas conçu dès le départ en terme de système réutilisable et indépendant, il sera très difficile de le réutiliser dans un autre contexte. Il sera fortement dépendant de notre application et le code gérant l'interaction de diffusion sera en partie mélangé avec le code de la partie fonctionnelle du composant.

De plus, comme les codes fonctionnel et « interactionnel » sont mal découplés, une modification du code d'une interaction risque d'avoir une influence sur le code fonctionnel. Ces deux éléments, a priori distincts, sont donc plus ou moins inter-dépendants, ce qui peut amener à des problèmes concernant l'évolutivité et la maintenabilité de l'application. Par exemple, supposons que notre application fonctionne avec un petit nombre de composants et se base sur un réseau local à haut débit et fiable. Il n'est pas trivial de faire évoluer cette application pour qu'elle gère plus de clients en se basant sur un réseau à faible débit et sujet à des problèmes de transmission. Il faut pour cela revoir en grande partie le code correspondant aux communications et aux interactions pour y gérer la tolérance aux fautes et s'adapter à une baisse du débit de transmission. Ce problème de débit est essentiel pour la diffusion du flux vidéo, et la tolérance aux fautes doit assurer que les demandes de vote sont bien envoyées à tous les clients et que les votes des clients sont bien reçus par le serveur. Si le code des interactions n'est pas bien découplé et séparé du code fonctionnel des composants, cette tâche d'adaptation s'en trouve complexifiée car ce code fonctionnel va être également en partie modifié. De plus, il est plus difficile de savoir où intervenir dans le code et les conséquences que peut avoir une modification.

Une meilleure structuration des interactions

La première architecture que nous avons décrite présente donc le désavantage majeur de mélanger la partie fonctionnelle de la partie interactionnelle du composant rendant cette dernière difficilement réutilisable.

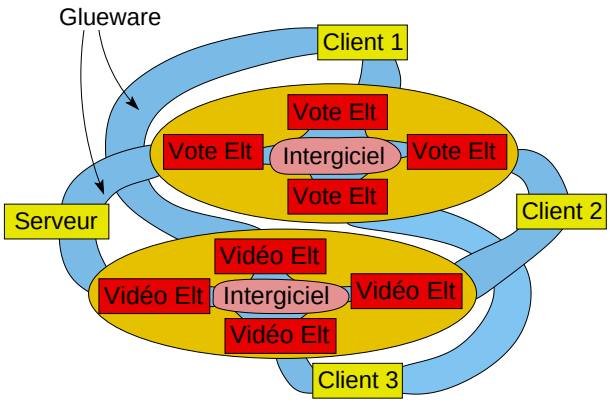

FIG. 2.2: Réorganisation des éléments d'interaction dans l'application de vidéo interactive

Afin de résoudre ces problèmes et limitations, nous proposons de bien découpler et d'extraire des composants tous les éléments et le code concernant une même interaction et de les regrouper de manière logique dans une entité à part. Cette nouvelle architecture est décrite sur la figure 2.2 page précédente. Tous les éléments gérant une même interaction sont regroupés au sein d'une même entité logique. Ces éléments communiquent toujours entre eux via un intergiciel. La nouvelle conception d'un système d'interaction permet de le rendre indépendant d'une application donnée et donc réutilisable. Elle permet aussi de bien séparer la partie fonctionnelle de la partie interactionnelle. Un composant serveur ou client (qui est maintenant réduit uniquement à sa partie fonctionnelle) possède un lien sur chacune des deux entités d'interaction (vote et diffusion de flux vidéo). Comme ces interactions sont maintenant conçues pour être réutilisées, les services qu'elles offrent sont génériques et doivent être adaptés à chaque contexte particulier, d'où la présence de glu entre les composants et les interactions.

Nous considérons que l'ensemble des éléments correspondant à une interaction forme de manière logique un composant logiciel à part entière. Ce type de composants est un peu particulier car ils sont dédiés aux communications ou aux interactions inter-composants. C'est pourquoi nous les appelons *composants d'interaction ou de communication*. Afin d'éviter toute ambiguïté entre un composant « fonctionnel » et un composant d'interaction, ce dernier type de composant est aussi appelé *médium*. Un médium est donc la réification d'une *abstraction d'interaction ou de communication* sous la forme d'un composant logiciel.

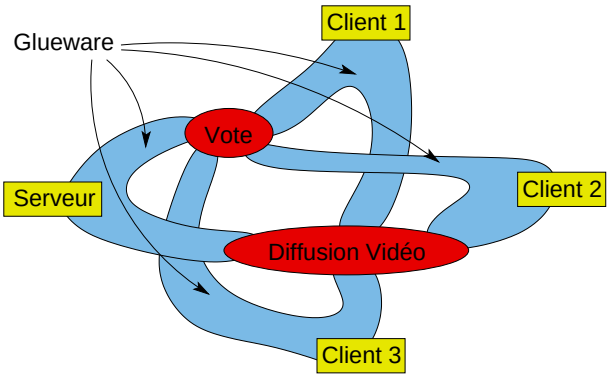

Fig. 2.3: Nouvelle architecture de l'application de vidéo interactive

En considérant les systèmes d'interaction comme des médiums, l'architecture de notre application de vidéo interactive devient celle de la figure 2.3. Elle contient deux composants d'interaction : l'un pour gérer la diffusion de flux vidéo et l'autre pour gérer le vote. Les quatre composants fonctionnels (le serveur et les trois clients) sont bien sûr toujours présents. On peut noter que ces composants fonctionnels n'ont plus aucun lien direct entre eux. En fait, pour communiquer, ils vont appeler les services offerts par les médiums. La glu entre un composant et un médium permet d'adapter les services offerts par un médium au besoin du composant les utilisant.

Cette nouvelle architecture met en avant les aspects d'interaction et de communication en les découplant clairement des aspects fonctionnels, contrairement à la première approche. De plus, les abstractions d'interaction sont désormais placées à un niveau équi-

valent aux aspects fonctionnels.

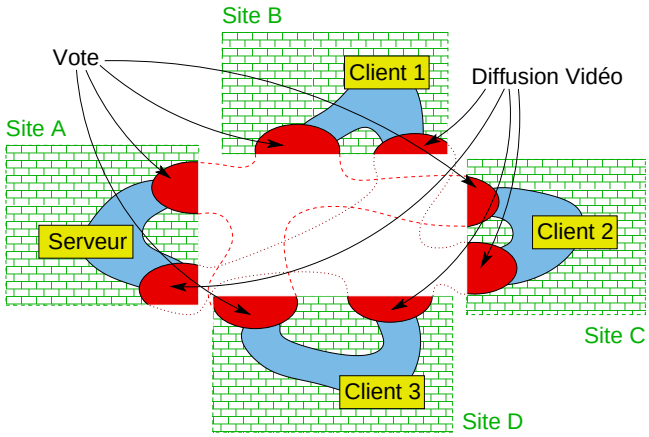

FIG. 2.4: Architecture de déploiement de l'application de vidéo interactive

Le déploiement de la nouvelle application est représenté sur la figure 2.4. Chaque composant (serveur ou client) est instantié sur un site différent. On peut constater que les médiums sont éclatés en plusieurs parties et qu'un morceau de chacun d'entre eux est déployé sur le même site qu'un composant (nous retrouvons alors la même architecture que celle de la figure 2.1 page 48 mais avec cette fois une séparation nette des aspects fonctionnels et interactionnels). Un composant a ainsi un lien local sur les médiums qu'il utilise pour réaliser ses communications avec les autres composants. Il n'a donc besoin d'appeler des services que localement sur les médiums. Ce sont ces derniers qui ont la charge de gérer les problèmes de localisation des composants dans le cadre d'un environnement distribué.

Ainsi, au niveau implémentation, en séparant clairement les aspects d'interactions des aspects fonctionnels, nous pouvons identifier, utiliser et réutiliser des composants spécialisés dans les interactions ou la communication : des médiums ou composants de communication.

2.1.2 Étude de la spécification des abstractions de communication

Après cette introduction aux médiums du point de vue de l'implémentation, nous nous intéressons dans cette section à leur spécification et plus précisément nous étudions l'intérêt des abstractions de communication de « haut niveau ». Pour cela, nous étudions la spécification d'une application de gestion de places de parking.

Spécification de l'application de gestion de parking

Un parking est composé d'un ensemble de places. Chaque place possède un identificateur unique. Une voiture garée dans le parking occupe une place précise. Une voiture peut entrer et sortir via l'un des deux accès du parking. Lorsqu'une voiture entre dans le parking, le système lui assigne la place qu'elle doit occuper. À l'extérieur du parking, un panneau d'affichage indique le nombre de places disponibles dans le parking.

La figure 2.5 page suivante est un diagramme d'instances UML représentant cette application. On y retrouve les deux accès, le panneau d'affichage ainsi que le composant

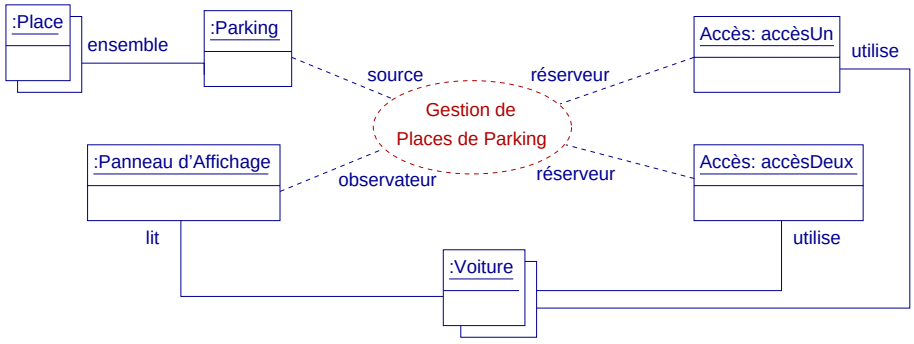

Fig. 2.5: Application de gestion de places de parking

parking gérant l'ensemble des places du parking physique. Ces quatre composants forment le système de gestion du parking. Les voitures sont des composants externes qui utilisent ce système.

Les quatre composants du système interagissent de la manière suivante :

- Quand une voiture veut entrer dans le parking, l'accès qu'elle utilise envoie une requête au composant parking pour obtenir un identificateur de place. Si le parking est plein, une valeur particulière est retournée.
- Quand une voiture quitte le parking, l'accès qu'elle utilise informe le composant parking que la place qu'elle occupait est de nouveau disponible à la réservation.
- Quand une voiture entre dans le parking ou en sort, le nombre de places disponibles dans le parking change et l'affichage du panneau est mis à jour en conséquence.

Ces quatre composants interagissent donc via la collaboration UML « Gestion de places de parking ». Dans cette collaboration, un accès joue un rôle de réserveur car il peut réserver des places pour des voitures (et également annuler ces réservations). Le panneau d'affichage joue le rôle d'observateur car il est informé en permanence du nombre de places disponibles dans le parking. Enfin, le composant parking joue le rôle de source car c'est lui qui connaît et gère la liste des places du parking.

Quelle est la complexité de la collaboration ?

La complexité de la collaboration utilisée dépend de ce que fait le composant parking et de comment la collaboration a été définie. Comme nous allons le voir, selon la responsabilité de ce composant, la collaboration prend différentes formes.

Une première spécification. Dans notre spécification de l'application de gestion de parking, le composant parking a la responsabilité de gérer lui même l'ensemble des places. Il doit savoir quelles sont les places occupées et celles qui sont disponibles. La collaboration est donc simple (voir figure 2.6 page ci-contre). Elle se contente de décrire les messages échangés entre les rôles et leurs dépendances. Par exemple, à chaque fois qu'une place est réservée (appel de `voitureEntrante`, message A.1) ou de nouveau rendue disponible (appel de `voitureSortante`, message B.1), les rôles observateurs sont informés du nouveau nombre de places disponibles par l'appel de `nbPlacesDisponibles` (messages A.2 et B.2).

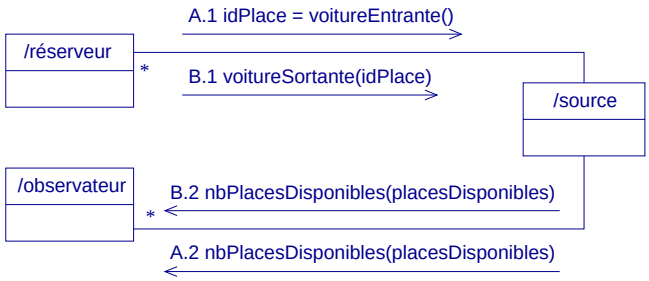

FIG. 2.6: Une première spécification de la collaboration de gestion de parking

Pour plus de précision sur ce que font ces méthodes, des pré et postconditions en OCL pourraient être définies afin de spécifier complètement leur sémantique.

Une spécification plus intéressante. Le patron d'interaction décrit par la collaboration n'est pas spécifique à notre application. Il peut être utilisé – ou plus précisément, réutilisé – dans d'autres contextes. Par exemple, si au lieu de réserver des places de parking, il s'agit de réserver des sièges dans un avion, la collaboration, après quelques changements mineurs (comme du renommage de méthode par exemple), peut être réutilisée dans un contexte de réservation aérienne. Le composant jouant le rôle de source est alors une compagnie aérienne voulant vendre des places sur un de ses vols. Des agences de voyage jouent les rôles de réserveur et d'observateur afin de réserver des places pour leurs clients.

Le contexte de cette application est complètement différent mais la structure de l'interaction est la même que dans la gestion de parking. La collaboration utilisée est donc très proche de la première. Mais comme pour le composant parking, c'est le composant correspondant à la compagnie aérienne qui a la responsabilité de gérer les places.

Cette gestion peut bien sûr être calquée sur celle faite par le composant parking de l'application de gestion de places de parking. La façon de gérer l'ensemble des identificateurs est en effet très similaire, les différences entre les deux applications au niveau de la manipulation des données étant minimes. Réutiliser la collaboration revient donc à concevoir un système de gestion d'identificateurs (au niveau du composant jouant le rôle de source) dont les différentes variantes – en fonction du contexte – sont très proches.

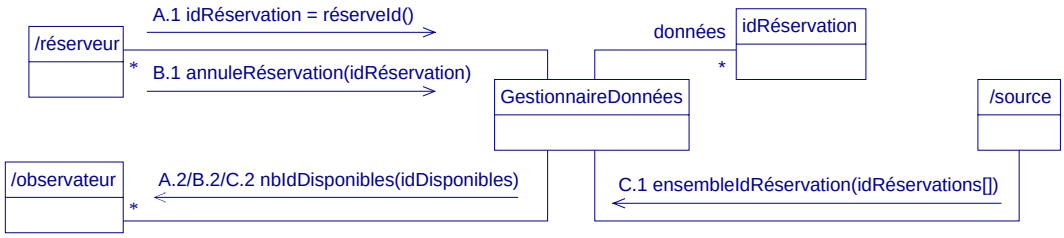

FIG. 2.7: Collaboration générique de gestion de réservation d'identificateurs

Pourquoi alors ne pas concevoir un système générique de gestion d'identificateurs et adaptable à tout type de données ? Ce système serait systématiquement utilisé à chaque

réutilisation de la collaboration, évitant ainsi de concevoir un nouveau système à chaque fois. Pour cela, nous proposons de placer ce système de gestion des identificateurs – et donc également l'ensemble des identificateurs – à l'intérieur de la collaboration. Cette nouvelle collaboration est représentée sur la figure 2.7 page précédente. Les identificateurs sont des objets de type `idRéservation` et ils sont gérés par le composant nommé `GestionnaireDonnées`. Le composant jouant le rôle source n'a plus à gérer ces données directement mais uniquement à préciser à ce gestionnaire quel est l'ensemble à manipuler via l'appel du service `ensembleIdRéservation` (message C.1). Les composants réserveurs font désormais leurs requêtes (appels des services `réserveId` (message A.1) pour réserver et `annuleRéservation` (message B.1) pour annuler une réservation) auprès du gestionnaire de données. C'est ce dernier qui a également la charge de prévenir les composants observateurs des changements du nombre d'identificateurs disponibles (en appelant le service `nbIdDisponibles`, messages A.2, B.2 et C.2).

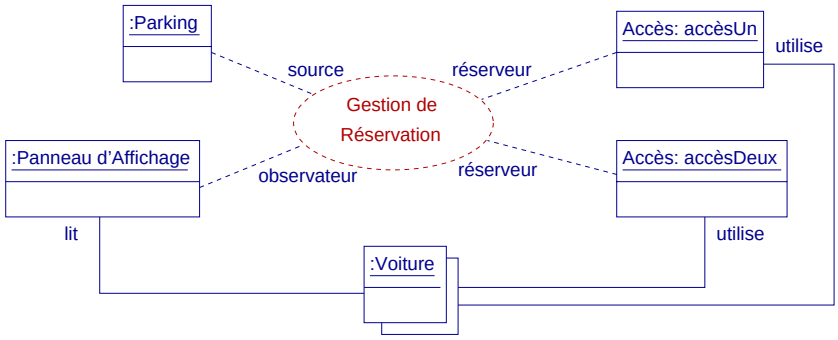

Fig. 2.8: Nouvelle description de l'application de gestion de places de parking

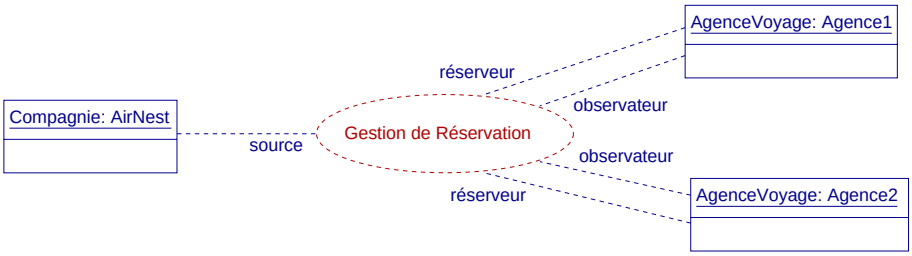

Fig. 2.9: Description de l'application de réservation aérienne

Les deux applications que nous avons décrites peuvent donc utiliser cette nouvelle collaboration (nommée « Gestion de Réservation »). Leurs nouvelles architectures sont décrites par la figure 2.8 pour l'application de gestion de places de parking et par la figure 2.9 pour l'application de réservation aérienne. Dans ces deux contextes, la collaboration utilisée est la même. Nous pouvons constater que les ensembles d'identificateurs n'apparaissent plus sur les figures car ils sont en effet maintenant gérés en interne dans la collaboration.

Comparaison des deux collaborations. En terme de réutilisabilité, la première collaboration présente très peu d'intérêt. Elle se contente uniquement de décrire les appels

de services et leurs imbrications entre les différents rôles. Ce patron d'interaction est trop simple et spécifique pour appartenir à un catalogue d'interactions. La deuxième collaboration est par contre beaucoup plus intéressante. Elle intègre directement les identificateurs et leur gestion, elle est donc « auto-suffisante », spécifiée indépendemment d'un contexte particulier et peut être facilement utilisée dans la spécification de différentes applications.

La deuxième collaboration, de par sa complexité, et enrichie de la responsabilité de gestion des identificateurs, est plus intéressante car en plus d'être réutilisable, elle permet de manipuler et de spécifier une *abstraction d'interaction* de haut niveau. Ici, l'abstraction concerne une interaction via une mémoire partagée contenant un ensemble d'identificateurs.

Une abstraction de communication et d'interaction est spécifiée par une collaboration UML comme nous venons de le voir. Cette abstraction peut être implémentée par un médium comme nous l'avons vu dans la section précédente. Ainsi, il est possible de manipuler et de réutiliser une même abstraction de communication aux niveaux de la spécification et de l'implémentation. De plus, une collaboration UML peut être implémentée par un médium.

2.1.3 Conclusion

À travers deux exemples, nous avons vu l'intérêt que pouvaient avoir les abstractions d'interaction et de communication de haut niveau pendant les phases de spécification et d'implémentation. Ces abstractions peuvent être réifiées sous forme de composants de communication ou médiums. Cela permet de manipuler une même abstraction de communication pendant tout le cycle de développement du logiciel, de la spécification la plus abstraite à l'implémentation. Cela permet également de réutiliser une abstraction de communication lors de la spécification ou la réalisation de différentes applications. Dans la prochaine section, nous définissons plus précisément cette notion de médium.

2.2 Définition des composants de communication

Un composant de communication ou d'interaction est un composant logiciel qui réifie une abstraction de communication. Une application est construite en interconnectant des composants de communication et des composants métiers ou fonctionnels. Afin de différencier ces deux types de composant, nous appelons les composants de communication des *médiums*.

Un médium réifie donc une abstraction de communication ou d'interaction. Ces abstractions peuvent être des protocoles, des services de communication, des systèmes d'interaction ou de collaboration de tout type, de tout niveau et de toute complexité. Nous ne différencions pas les abstractions « simples » des abstractions « complexes ». Nous considérons qu'elles sont de même niveau, qu'elles sont manipulables de la même manière et ce, à la fois pendant les phases de spécification et d'implémentation. Par exemple, un système de diffusion de flux vidéo est beaucoup plus complexe à implémenter qu'un système de diffusion d'événements. Mais ces deux systèmes peuvent être manipulées de manière identique sous forme de médium. En effet, abstraitement, ces deux systèmes sont très similaires, seule la nature de l'information à diffuser change. Il n'y a donc a priori pas de raison de les traiter différemment.

Sans exhaustivité, voici quelques exemples d'abstractions de communication ou d'interaction que l'on peut utiliser sous forme de médiums :

- Un système de diffusion d'événements,
- Un système de diffusion de flux vidéo,
- Une coordination à la Linda à travers une mémoire partagée,
- Un système de vote,
- Un protocole de consensus,
- Un protocole de diffusion atomique.

Un médium, comme tout composant logiciel, prend plusieurs « formes » selon le niveau auquel il est considéré. Il existe sous la forme d'une spécification permettant de décrire l'abstraction de communication qu'il réifie, mais aussi au niveau de l'implémentation et du déploiement. Il est donc possible de manipuler une abstraction de communication de haut niveau pendant tout le cycle de développement du logiciel.

Dans la section suivante, nous détaillons la méthodologie de spécification des médiums en UML puis donnons quelques exemples de spécifications de médiums.

2.3 Méthodologie de spécification de médiums en UML

Comme pour les composants classiques, les composants de communication requièrent une spécification précise afin de pouvoir être utilisés correctement. Cette spécification décrit toutes les informations nécessaires sur comment utiliser un médium mais également sur ce qu'il réalise (en se plaçant du point de vue du client du médium). Tout cela peut être encapsulé dans un contrat, comme décrit dans [21, 76, 77]. Ce contrat doit inclure la liste des services offerts et requis par un médium mais cela ne suffit pas. Il doit aussi spécifier la sémantique et le comportement dynamique de chacun des services et du médium. Notre méthodologie définit ces contrats au niveau de la spécification abstraite, indépendamment de tout choix d'implémentation.

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 un 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 fonction du rôle qu'ils jouent du point de vue des services du médium qu'ils utilisent. Pour le médium de diffusion par exemple, il y a un rôle d'émetteur et un rôle de récepteur.

Notre méthodologie de spécification est basée sur UML (dans sa version 1.3 [83]). Une abstraction de communication peut être représentée par une collaboration UML comme nous l'avons vu dans la section 2.1.2 page 51. Une collaboration UML est donc une base logique pour la spécification d'un médium. Les rôles des composants utilisant un médium pour leurs communications correspondent aux rôles utilisés dans les collaborations UML.

Comme tout composant, le médium offre des services qui seront utilisés par les composants connectés au médium. Le médium peut aussi avoir besoin d'appeler des services sur ces composants. Les services requis et offerts sont regroupés en interfaces offertes et requises. À chaque rôle de composant peuvent être associées une interface de services offerts (par le médium aux composants jouant ce rôle) et une interface de services requis (par le médium sur les composants jouant ce rôle).

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

Un diagramme de collaboration pour décrire l'aspect structurel du médium. Des

messages peuvent être ajoutés afin de décrire les appels d'opérations réalisés dans le cadre de l'exécution d'un service (plusieurs interactions, correspondant chacune à un service, sont décrites au besoin). La numérotation des messages permet de spécifier l'ordonnancement de ces appels.

Des contraintes OCL qui permettent de spécifier les propriétés du médium (voire des rôles si nécessaire) ainsi que la sémantique statique des services offerts et requis (pré et postconditions sur les services).

Des diagrammes d'états associés au médium ou à ses services afin de gérer les contraintes temporelles et de synchronisation et de spécifier la sémantique dynamique des services (en plus des interactions précisées sur le diagramme de collaboration).

Les autres types de diagrammes ou outils UML, tels que les diagrammes de séquence, peuvent bien sûr être utilisés. Mais nous pensons que les trois types que nous avons cités sont suffisants dans la majorité des cas.

La collaboration représentant le médium est définie au niveau spécification. En effet, il faut spécifier de manière « générale » le médium et ses services. Une collaboration au niveau instance ne permet pas de faire cela. Elle ne représente qu'un cas particulier d'une interaction et ne permet pas de généraliser le fonctionnement d'une interaction.

Nous généralisons l'utilisation d'OCL partout où cela est possible afin de lier le plus formellement possible les différentes vues et d'assurer au mieux leur cohérence. En particulier, nous écrivons en OCL les expressions booléennes des gardes d'émission des messages des collaborations ainsi que celles des gardes de transition entre états des diagrammes d'états. 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 ou une de ses opérations, le contexte est cette classe. Dans un diagramme de collaboration au niveau spécification, il est possible de spécifier qu'un message n'est envoyé qu'à un certain nombre d'instances d'une certaine classe ou rôle (par défaut, il est envoyé à toutes les instances). Pour cela, nous proposons d'utiliser une expression OCL `select` permettant de sélectionner ce sous-ensemble. La norme 1.3 d'UML sur laquelle nous nous basons, ne préconise pas de langage particulier pour l'expression des gardes de transition dans les diagrammes d'états ou d'envoi de messages dans les diagrammes de collaboration.

La structure de la collaboration doit suivre une forme particulière, afin de prendre en compte les caractéristiques des composants en général et des médiums en particulier. À l'intérieur de la collaboration, une classe représente le médium dans son ensemble. Les interfaces de services offerts et requis doivent être présentes ainsi que tous les rôles de composant pouvant se connecter au médium. Dans le diagramme de collaboration représentant le médium, un rôle correspond à un composant (jouant un rôle donné) connecté au médium.

La figure 2.10 page suivante montre la structure générique de la relation entre un rôle de composant générique et un médium. Un médium nommé « <MediumName> » est représenté dans le diagramme de collaboration par une classe nommée `<MediumName>Medium`[1]. Pour chaque rôle « <RoleName> » de composant, il existe deux

[1] Nous avons choisi d'utiliser des conventions de nommage pour identifier des éléments particuliers de nos spécifications mais nous aurions pu utiliser les mécanismes standards d'extensions d'UML comme les stéréotypes. Par exemple, la classe représentant le médium aurait pu être marquée avec le stéréotype

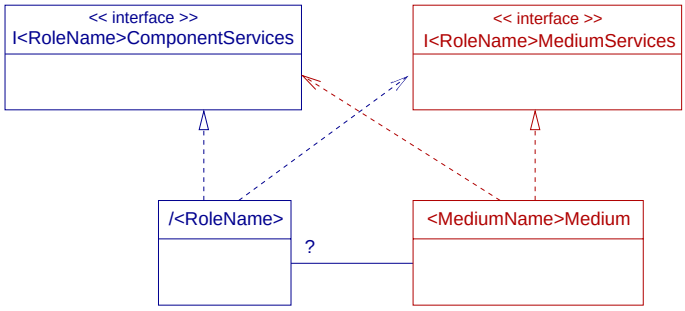

Fig. 2.10: Relation générique entre un rôle et un médium

interfaces :

- `I<RoleName>MediumServices` qui est l'interface regroupant les services offerts par le médium aux composants jouant le rôle « <RoleName> ». C'est-à-dire les services utilisés par ces composants. Ce rôle a donc une dépendance sur cette interface.
- `I<RoleName>ComponentServices` qui est l'interface regroupant les services qui doivent être réalisés par le composant jouant le rôle « <RoleName> » et qui sont appelés par le médium. Le médium a donc un lien de dépendance sur cette interface.

Si certaines interfaces sont vides, il n'est pas nécessaire de les faire apparaître.

Le « ? » de l'association entre le rôle `/<RoleName>` et la classe `<MediumName>Medium` est remplacé dans une vraie description par le nombre de composants de ce rôle qui peuvent se connecter à ce médium.

D'autres éléments (classes, attributs, interfaces, etc.) peuvent être en relation avec la classe `<MediumName>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>Medium`. Les propriétés locales à une liaison composant/médium le sont dans une classe d'association de l'association entre les classes `<MediumName>Medium` et le rôle adéquat.

Une propriété un peu particulière peut être définie : la propriété d'utilisabilité du médium. Elle spécifie dans quelles conditions les services du médium sont utilisables. Ces conditions peuvent être par exemple l'obligation de présence d'au moins un composant d'un certain rôle ou que le médium ou un rôle soit dans un état particulier. Cette propriété du médium s'appelle `usable` et est un attribut de type booléen de la classe `<MediumName>Medium`. Sa valeur est fixée par une expression OCL exprimée dans le contexte de cette classe.

2.3.1 Extensions d'OCL

Nous avons rencontré quelques problèmes d'expressivité en OCL. Bien que ne voulant pas nous éloigner de la norme, il nous a tout de même semblé pertinent de rajouter deux extensions au langage OCL. La première sert à référencer l'appelant d'une opération et la seconde à exprimer qu'une opération pouvant avoir des effets de bords a été appelée.

« `medium` ».

Afin de bien distinguer nos extensions de l'OCL standard, leurs utilisations sont notées en *italique*.

Le pseudo attribut *caller*

Le pseudo attribut *caller* permet d'obtenir une référence sur l'instance qui a appelée une opération. Il n'est donc utilisable que dans la définition des pré et des postconditions d'une opération. Il s'utilise de la même façon que l'attribut `self`.

Au niveau implémentation et programmation, il est également souvent utile de préciser à l'appelé quel est l'appelant d'une méthode. Pour cela, un des paramètres de la méthode appelée contient la référence de l'appelant qui la passe lorsqu'il appelle la méthode. Il serait possible d'utiliser la même technique dans nos spécifications. Mais nous ne trouvons pas élégant dans nos spécifications abstraites, réalisées indépendamment de toute implémentation, de faire apparaître une référence sur l'appelant dans les signatures des services offerts et requis. Surtout que, comme nous le verrons dans le chapitre suivant, en fonction du niveau de spécification auquel nous nous plaçons, pour un même service, la référence sur l'appelant pourra être indispensable ou inutile. Il n'est donc pas possible de la faire apparaître dans la signature des services car cela implique de modifier la signature de ceux-ci lors de l'application du processus de raffinement. Ce qui est en contradiction avec un des principes de notre approche qui est que l'abstraction réifiée – et donc les services offerts et requis – ne change pas, quelque soit le niveau de manipulation considéré (pour l'implémentation, cela n'est néanmoins pas toujours possible à assurer).

L'intégration du `caller` pourrait également se faire assez facilement dans les langages de programmation, les intergiciels ou les plates-formes de composants. En effet, lors de l'appel d'une méthode, la référence de l'appelant est stockée dans la pile d'exécution du programme lorsque l'on est en centralisé. Pour un appel à distance, il est indispensable de connaître l'appelant pour lui renvoyer le résultat de l'exécution de la méthode. La référence sur l'appelant serait donc simple à récupérer.

La primitive *oclCallOperation*

Parfois, il est utile de spécifier dans une postcondition qu'une opération a été appelée, bien que souvent ce genre d'information est plutôt spécifié sur le diagramme de collaboration. Néanmoins, si cette opération retourne un résultat indispensable à la spécification de la postcondition d'une opération en OCL, il faut pouvoir référencer ce résultat. C'est pour cela que nous avons rajouté la primitive `oclCallOperation` dont la signature est :

object.*oclCallOperation(opName [,param]*)*

Elle est utilisable uniquement dans une postcondition et permet de préciser qu'une opération `opName` a été appelée sur l'objet `object` avec les paramètres `([, param]*)`. Cet appel a eu 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. L'opération appelée peut avoir des effets de bords (OCL permet seulement de faire apparaître l'appel d'une opération qui n'en a pas).

Voici un exemple d'utilisation de cette primitive :

```
context MaClasse::op1()
post:
    let res = obj.oclCallOperation(op2, true) in
    obj2.values -> includes(res)
```

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` et que le résultat retourné est disponible dans l'attribut `res`. Ensuite, ce résultat doit appartenir à l'ensemble nommé `values` de l'objet `obj2`.

Dans la nouvelle version 2.0 d'OCL, qui sera intégrée à la future norme 2.0 du langage UML, une primitive similaire à notre extension a été ajoutée. Il sera donc possible en UML 2.0 de spécifier en respectant la norme UML ce que nous faisons actuellement avec une extension personnelle.

2.3.2 Gestion de la dynamicité et du cycle de vie du médium

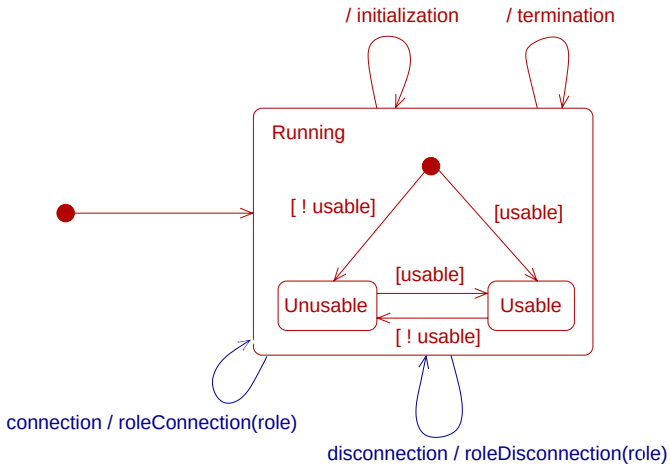

Fig. 2.11: Diagramme d'états générique d'un médium

Dans les spécifications de médiums, il peut parfois être indispensable de gérer la dynamicité de la connexion ou de la déconnexion d'un rôle au médium. Par exemple, spécifier qu'une opération donnée doit être appelée sur un rôle dès que celui-ci se connecte au médium. C'est le but du diagramme d'états de la figure 2.11. Il représente le cycle de vie générique d'un médium. Dès sa création, il passe dans un état appelé `Running`. Ensuite, en fonction de son état d'utilisabilité défini dans l'attribut `usable`, il se trouve dans un des deux états `Usable` ou `Unusable`.

À la connexion d'un nouveau rôle, l'événement `connection` est levé et l'opération `roleConnection` est appelée avec en paramètre la référence sur le nouveau rôle. Si dans une spécification il est nécessaire de spécifier ce qu'il se passe quand un nouveau rôle se connecte, il suffit soit de modifier l'opération appelée, soit de spécifier la sémantique de cette opération `roleConnection` (en OCL) dans ce contexte bien particulier. La déconnexion d'un rôle est traitée de manière similaire avec la génération de l'événement

`disconnection` qui a pour conséquence l'appel de l'opération `roleDisconnection` avec la référence sur le rôle qui s'est déconnecté.

Enfin, il est possible également d'attacher des actions à l'initialisation et à la terminaison du médium. Deux opérations sont définies dans ce but : `initialization` et `termination`. Pour spécifier les actions, il suffit de définir la sémantique de ces opérations.

Le diagramme d'états est associé à la classe représentant le médium. Les expressions OCL sont donc à définir dans ce contexte. Dans ces spécifications, il est possible de vérifier le type du rôle avec la fonction OCL `isKindOf`. Il peut aussi être utile de savoir si le médium est dans un état utilisable ou pas en vérifiant via la fonction OCL `oclInState` si l'on se trouve dans le sous-état `Usable` ou `Unusable`.

2.4 Exemples de spécification de médiums

Nous spécifions trois médiums différents afin d'illustrer l'application de notre méthodologie de spécification de médiums. Tout d'abord nous commençons par un exemple simple : un médium d'envoi asynchrone de messages. Ensuite, nous spécifions un médium intégrant un système de vote (tel que celui utilisé dans l'application de vidéo interactive de la section 2.1.1 page 48). Ce médium, plus complexe, permet d'étudier une spécification plus intéressante en intégrant des diagrammes d'états et en montrant des exemples d'utilisation de nos extensions d'OCL ainsi que de la généralisation de l'usage de ce dernier. Enfin, nous spécifions le médium de réservation d'identificateurs. Ce médium intègre l'abstraction de réservation utilisée par les applications de gestion de parking et de compagnie aérienne (voir la section 2.1.2 page 51). La collaboration représentant ce médium est cette fois structurée selon les règles que nous avons définies (contrairement à la collaboration de la figure 2.7 page 53 qui ne tenait compte d'aucune contrainte particulière). La spécification de ce médium nous sert de base à l'application du processus de raffinement que nous présentons dans le prochain chapitre.

2.4.1 Un médium simple d'envoi asynchrone de messages

Description du médium et de ses services

Ce médium permet d'envoyer des messages de manière asynchrone à un composant connecté au même médium. 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 messages sont ordonnés, ils sont lus dans l'ordre où ils sont envoyés. L'émetteur et le récepteur doivent être deux composants différents et le médium n'est utilisable que si les deux composants sont présents.

Le médium définit donc deux rôles de composants : émetteur (ou `sender`) et récepteur (ou `receiver`). Les services offerts par le médium sont les suivants :

- Pour le rôle émetteur : `void send(Message msg)` pour envoyer un message `msg` au récepteur.
- Pour le rôle récepteur : `Message receive()` pour lire le prochain message non lu. Si aucun message n'est disponible, la valeur retournée est `null`.

Le diagramme de collaboration du médium

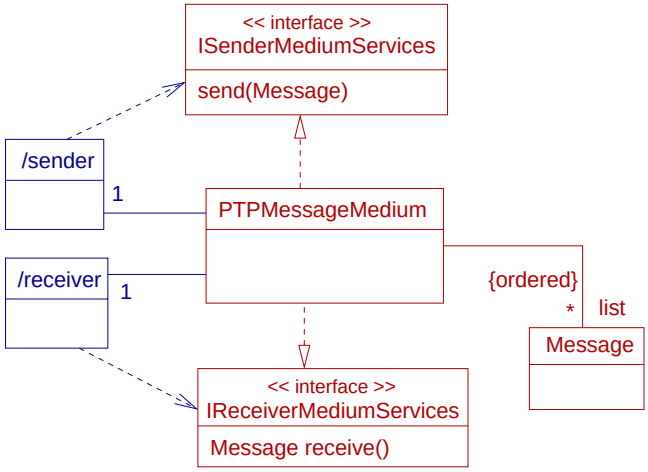

Fig. 2.12: Le médium d'envoi de message asynchrone point à point

La figure 2.12 représente la collaboration décrivant le médium. On y retrouve les deux rôles `sender` et `receiver`. Les interfaces `ISenderMediumServices` et `IReceiverMediumServices` contiennent les signatures des services offerts par le médium. Chaque rôle de composant est dépendant d'une de ces deux interfaces en fonction des services dont il a besoin. Le rôle `sender` est dépendant de l'interface `ISenderMediumServices` définissant le service `send` et le rôle `receiver` est dépendant de l'interface `IReceiverMediumServices` définissant le service `receive`.

Le médium est représenté par la classe `PTPMessageMedium`. Il implémente donc ces deux interfaces. Comme aucun des rôles n'a besoin d'implémenter de services qui seront appelés par le médium, les interfaces `ISenderComponentServices` et `IReceiverComponentServices` sont vides et elles ne sont donc pas représentées sur le diagramme de collaboration.

Une file de messages de type « FIFO » représentée par l'association ordonnée entre les classes `Message` et `PTPMessageMedium` et nommée `list` est associée au médium. Elle contient les messages envoyés par le rôle émetteur et qui seront lus par le rôle récepteur.

La multiplicité de l'association entre la classe `PTPMessageMedium` et chacun des rôles, du côté du rôle est « 1 » dans les deux cas. Cela signifie qu'il ne peut y avoir qu'un seul composant jouant le rôle émetteur et qu'un seul composant jouant le rôle récepteur connectés à ce médium.

Spécifications OCL

Les deux composants émetteur et récepteur doivent être différents. Cela peut s'exprimer par l'écriture en OCL d'un invariant pour la classe représentant le médium :

context PTPMessageMedium
inv: self.sender <> self.receiver

Le médium n'est utilisable que si les deux composants sont connectés. La propriété `usable` qui décrit cette condition peut-être exprimée de la manière suivante en OCL[2] :

```
context PTPMessageMedium inv:
usable = (self.sender -> notEmpty and self.receiver -> notEmpty)
```

Les services `send` et `receive` doivent aussi être spécifiés en OCL. Pour ces deux services, ils ne peuvent être appelés que si le médium est dans un état utilisable (propriété `usable` à vrai). Cette contrainte apparaît donc dans les préconditions.

L'appel de `send` ajoute le message passé en paramètre à la fin de la liste :

```
context PTPMessageMedium::send(Message msg)
pre: usable = true
post: list = list@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 PTPMessageMedium::receive() : Message
pre: usable = true
post:
    if list -> isEmpty
    then result = null
    else
        list = list@pre -> subSequence(2,list@pre->size) and
        result = list@pre -> first
    endif
```

Le comportement du médium est entièrement spécifié en OCL. L'utilisation de diagrammes d'états n'est donc pas utile dans le cas de ce médium. Il n'est pas non plus nécessaire de faire apparaître les appels de services sur le diagramme de collaboration en y rajoutant les messages associés.

Exemple d'utilisation du médium d'envoi de messages asynchrone

FIG. 2.13: Exemple d'utilisation du médium d'envoi de messages asynchrone point à point

La figure 2.13 décrit l'architecture d'une application utilisant le médium d'envoi de

[2]La structure de la collaboration impose la présence d'un et d'un seul rôle de chaque type. L'expression de cette condition peut sembler alors redondante mais il est utile d'exprimer clairement cette propriété. Car comme nous le verrons dans le chapitre 4, grâce à notre plate-forme d'implémentation, elle peut en effet être vérifiée lors du déploiement et de l'instantiation.

messages point-à-point asynchrone. Dans cette application, un capteur de température fait des mesures à des intervalles de temps réguliers. Ces mesures sont transmises à un analyseur qui – lorsque son activité le lui permet – récupère les valeurs de ces mesures et les traite. Le médium sert donc à la transmission des mesures entre le capteur qui joue le rôle d'émetteur et l'analyseur qui joue le rôle de récepteur.

2.4.2 Un médium de vote

Description informelle et listes des services

Le médium de vote réifie une abstraction de vote comme son nom l'indique. Il permet à deux types de composants différents d'interagir. Certains composants sont initiateurs de votes, ils proposent une liste de valeurs dans laquelle les autres composants qui sont des votants choisissent chacun un élément. L'initiateur de vote précise la durée maximale du vote pendant laquelle les composants votants peuvent voter. Le vote n'est pas obligatoire. Le service de lancement de vote 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.

Les deux rôles de composants sont les suivants :
- `voteProposer` pour les composants initiateurs de votes.
- `voter` pour les composants votants.

Pour le rôle `voteProposer`, les services sont les suivants :
- Offert par le médium :
  `Value[] newVoteSession(Value[] prop, Integer timeout)` : lance un nouveau vote et demande à tous les votants présents au moment de l'appel de ce service d'y participer. `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 votés. Ce service est bloquant tant que le vote n'est pas terminé.

Pour le rôle `voter`, les services sont les suivants :
- À implémenter par le composant :
  `void voteRequest(Value prop[], Integer timeout, Integer voteId)` : le médium informe le composant votant qu'une nouvelle session de vote a été lancée et demande au composant d'y participer. `prop` est la liste des valeurs parmi lesquelles le composant peut en choisir une. `timeout` est le temps maximum qui lui est alloué pour voter et `voteId` est l'identificateur du vote permettant de différencier les différents votes en cours.
- Offert par le médium :
  `void vote(Value choice, Integer voteId)` : permet à un composant de donner son choix pour un vote donné. `voteId` est l'identificateur du vote auquel participe le composant et `choice` est la valeur qu'il a choisie.

Il peut y avoir autant de composants jouant le rôle votant ou initiateur de votes que voulu. Le médium est dans un état utilisable si au moins un composant initiateur de votes et un composant votant sont présents.

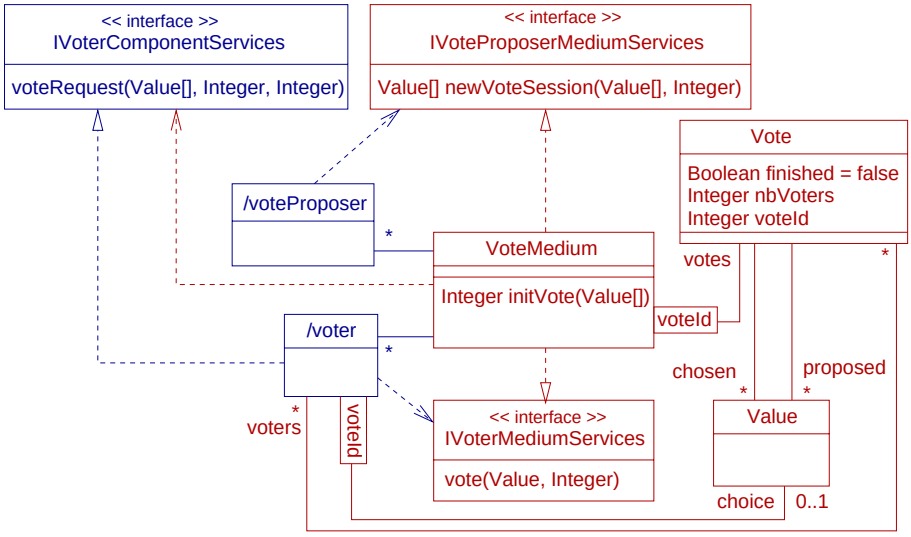

FIG. 2.14: Diagramme de collaboration du médium de vote

Diagramme de collaboration du médium de vote

Le diagramme de collaboration du médium de vote est décrit par la figure 2.14. On y retrouve les deux rôles ainsi que la classe `VoteMedium` représentant le médium de vote. Ce médium implémente deux interfaces : `IVoteProposerMediumServices` et `IVoterMediumServices` spécifiants les services offerts aux composants jouant respectivement les rôles d'initiateur de votes et de votant. L'interface `IVoterComponentServices` spécifie les services que doivent implémenter les composants jouant le rôle votant et qui sont appelés par le médium. On y retrouve le service `voteRequest` qui est appelé pour demander au composant jouant un rôle votant de participer à un nouveau vote qui vient d'être lancé.

La classe `Vote` représente un vote. Un vote est identifié par la valeur de `voteId` dans l'association qualifiée entre les classes `VoteMedium` et `Vote`. L'attribut booléen `finished` permet de savoir si le vote est terminé (valeur vraie) ou non (valeur fausse). L'attribut `nbVoters` contient le nombre de votants à qui la demande de vote a été envoyée. L'attribut `voteId` contient l'identificateur du vote (que l'on retrouve sur les associations qualifiées). 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'association entre un rôle `voter` et un vote, nommée `voters` du côté du rôle, permet de référencer la liste des votants à qui la demande de vote a été envoyée pour chaque vote.

L'opération `initVote(Value prop[])` de la classe `VoteMedium` sert à initier un vote (création d'un objet `Vote` et de son association `proposed` ainsi que d'un identificateur de vote, c'est-à-dire de la valeur de l'attribut `voteId` pour ce vote) avant de demander à tous les votants de voter.

La figure 2.15 page suivante représente un sous-ensemble du diagramme de collaboration du médium de vote (voir la figure 2.14) sur lequel sont ajoutés les messages constituant l'interaction correspondant à l'appel du service `newVoteSession`. Les seuls éléments qui ont été conservés sont les deux rôles et la classe représentant le médium.

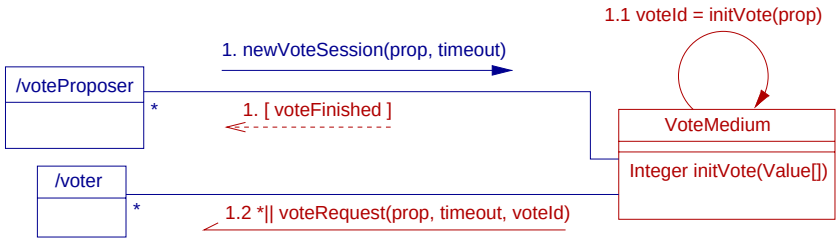

Fig. 2.15: Vue dynamique de la collaboration du médium de vote

L'interaction est la suivante :

1. L'opération `newVoteSession` est appelée (message 1). `prop` est la liste des valeurs parmi lesquelles choisir et `timeout` est le temps maximum de vote.
2. Le vote est initialisé par l'appel de l'opération `initVote(prop)` (message 1.1) qui retourne l'identificateur du vote courant (attribut `voteId`).
3. La demande de vote est envoyée ensuite en parallèle à tous les composants jouant le rôle `voter`[3] (message 1.2)
4. Certains composants `voter` votent mais pas obligatoirement tous[4].
5. L'opération `newVoteSession` 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 `voteFinished` en OCL.

Spécifications en OCL

Nous allons maintenant spécifier les deux services offerts par le médium ainsi que l'opération `initVote` à l'aide d'OCL. Le service `voteRequest` qui doit être implémenté par un rôle votant n'a pas à être spécifié car la demande de vote est interprétée par le composant comme il le désire. Aucune contrainte de traitement de cette opération ne lui est imposée (il n'est même pas obligé de voter).

La contrainte d'utilisabilité du médium est exprimée de la façon suivante :

```
context VoteMedium inv:
usable = (self.voteProposer -> notEmpty and self.voter -> notEmpty)
```

Cet invariant exprime le fait que les services du médium sont utilisables si et seulement si au moins un composant initiateur de vote et au moins un composant votant sont connectés au médium.

[3] La demande est envoyée plus précisément à tous les composants votants connectés au médium au moment de l'appel de l'opération pour lancer un nouveau vote. Si des composants votants se connectent plus tard au médium, ils ne participeront pas au vote.

[4] C'est pour cela qu'il n'est pas possible d'ajouter un message indiquant l'appel de l'opération `vote` sur la classe `VoteMedium`.

Opération Integer initVote(Value[] prop). Cette opération initialise un vote à l'aide de la liste des valeurs (paramètre `prop`) parmi lesquelles les votants pourront en choisir une. Elle 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). La spécification en OCL de cette opération est la suivante :

```
context VoteMedium::initVote(Value[] prop) : Integer
pre: usable = true
post:
    votes[result]@pre -> isEmpty and                    -- il n'existait pas de vote ayant
                                                        -- result comme identificateur
    votes[result].proposed = prop and
    votes[result].voters = voter and
    votes[result].nbVoters = voter -> size and
    votes[result].voteId = result and
    votes[result].oclIsNew                              -- nouveau vote initialisé
```

`result` contient l'identificateur du nouveau vote et c'est donc la valeur retournée lors de l'appel de ce service. Le nouveau vote est initialisé en fonction des rôles `voter` présents : l'ensemble `voters` contient la liste des votants et l'attribut `nbVoters` est égal à leur nombre. L'identificateur du vote est également mémorisé dans le vote dans l'attribut `voteId`.

Opération newVoteSession(Value[] prop, Integer timeout). 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. Voici la spécification de cette opération :

```
context VoteMedium::newVoteSession(Value prop[], Integer timeout) : Value[]
pre: usable = true
post:
    let voteId = self.oclCallOperation(initVote, prop) in
        voter.oclCallOperation(voteRequest, prop, timeout, voteId) and
        votes[voteId].finished = true and
        voteFinished = true and
        result = votes[voteId].chosen
```

À 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 2.15 page précédente. Il est aussi précisé que l'opération `initVote` qui retourne la valeur de `voteId` utilisée dans cette spécification OCL[5] a été appelée pendant l'exécution de l'opération. Pendant cette exécution, le service `voteRequest` a été également appelé chez tous les votants pour leur demander de voter (cela a en fait été aussi spécifié dans la vue dynamique du diagramme de collaboration). Par contre, l'ordre d'appel de ces deux opérations

[5] L'appel de l'opération `initVote` est indispensable à la spécification OCL du service `newVoteSession` car elle retourne l'identificateur du vote courant. Comme cette opération n'est pas sans effet de bord, il n'est pas possible d'utiliser l'appel classique d'opération en OCL. Il faut donc passer par notre extension `oclCallOperation`. Ici, nous sommes en présence d'un exemple typique d'utilisation de cette extension.

n'est pas déterminable. Or il est essentiel que `initVote` soit appelée avant `voteRequest`. Cet ordre est défini dans le diagramme d'interaction de la figure 2.15 page 66.

Opération vote(Value choice, Integer voteId). Cette opération permet à un composant votant de répondre à une demande de vote. `choice` est le choix effectué par ce composant pour le vote identifié par `voteId`. Le vote est 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 de gestion 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 n'est pas pris en compte puisque le résultat du vote aura déjà été renvoyé au composant ayant initié ce vote. La spécification de cette opération est la suivante :

```
context VoteMedium::vote(Value choice, Integer voteId)
pre:
    usable = true
    votes[voteId] -> notEmpty and                                  -- le vote existe bien
    votes[voteId].proposed -> includes(choice) and                 -- le choix est correct
    caller.choice[voteId] -> isEmpty          -- le composant n'a pas déjà participé à ce vote
post:
    if votes[voteId].finished = false                              -- si le vote est non terminé
    then
        votes[voteId].chosen = votes[voteId].chosen@pre -> including(choice) and
                                                  -- le choix est ajouté à la liste des votes
        caller.choice[voteId] = choice              -- il est mémorisé que le votant a voté
    else                                                           -- si le vote est terminé
        votes[voteId].chosen = votes[voteId].chosen@pre
                                          -- pas de changement car le vote est terminé
        caller.choice[voteId] = choice               -- le choix du votant est tout de même
                                          -- mémorisé pour qu'il ne puisse pas revoter
    endif
```

Nous pouvons noter dans cette spécification l'utilisation du pseudo attribut *caller*. Il est ici indispensable de l'utiliser car la mise à jour des différentes valeurs relatives à un vote dépend d'informations relatives au votant. Il faut donc être capable de l'identifier et la seule manière de le faire est d'utiliser notre extension d'OCL.

Diagrammes d'états

La figure 2.16 page suivante représente le diagramme d'états associé à l'opération `newVoteSession` (appelée par un composant jouant le rôle d'initiateur de votes). Il est relativement simple et sert à préciser quand l'opération se termine, c'est-à-dire quand le vote est terminé.

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

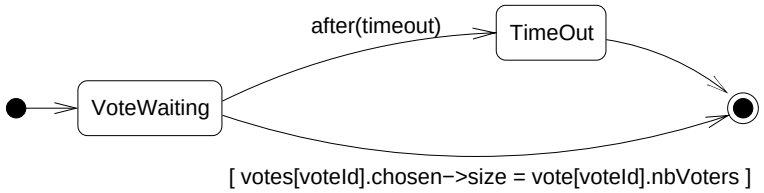

FIG. 2.16: Diagramme d'états associé au service `newVoteSession`

- tous les votants ont répondu. Comme il ne peuvent voter chacun qu'une seule fois, si le nombre de réponses est égal au nombre de votants à qui la demande de vote a été envoyée, ils auront tous votés. Cela est exprimé par la contrainte OCL suivante :
  votes[voteId].nbVoters = votes[voteId].chosen -> size
  Avec voteId ayant pour valeur celle de la spécification OCL de l'opération `newVoteSession` 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'états explicité, il est possible de définir l'expression OCL `voteFinished` utilisée précédemment :

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

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

Nous avons également besoin de détecter et de gérer la déconnexion des votants. En effet, si un votant se déconnecte et qu'il n'a pas voté à un ou plusieurs votes, il faut décrémenter le nombre de votants participant à ces votes. Sinon, la condition qui vérifie que le nombre de votes est égal au nombre de votants à qui la demande a été envoyée ne pourra jamais être vérifiée pour ces votes. Pour cela, nous utilisons le diagramme d'états générique du médium (voir la section 2.3.2 page 60). Ici, nous devons définir en OCL la sémantique de l'opération `roleDisconnection` :

```
context VoteMedium::roleDisconnection(Role role)
post:
    role.isKindOf(voter)) implies votes -> forAll ( v |
        (v.voters -> includes(role) and role.choice[v.voteId] -> isEmpty) implies
            v.nbVoters = v.nbVoters@pre - 1)
```

Si le composant qui se déconnecte est un votant, alors il faut vérifier pour chacun des votes auxquels on lui a demandé de participer, si il a voté ou non. Si il n'a pas voté, alors il faut décrémenter le nombre de votants participant à ce vote. Si il a déjà voté, rien ne change.

Exemples d'utilisation du médium de vote

Le médium de vote est celui qui est utilisé dans le cadre de l'application de vidéo interactive décrite dans la section 2.1.1 page 48.

2.4.3 Le médium de réservation des applications de gestion de parking et de réservation aérienne

Ce médium réifie l'abstraction d'interaction via un système de réservation d'identificateurs que nous avons utilisée dans la section 2.1.2 page 51. La collaboration UML décrivant cette interaction avait été décrite de manière intuitive. Nous allons maintenant la spécifier en suivant les règles de notre méthodologie de spécification de médiums.

Description informelle et listes des services

Le médium de réservation gère un ensemble d'identificateurs de réservation (qui correspondent à des places dans un parking ou des numéros de siège dans un avion dans le cas de nos deux exemples précédents). Ces identificateurs peuvent être réservés par des composants jouant le rôle réserveur (ou `reserver`). Ces mêmes composants peuvent aussi annuler ces réservations. Des composants jouant le rôle observateur (ou `observer`) sont informés du nombre d'identificateurs disponibles à chaque fois que celui-ci change (après chaque réservation ou annulation notamment). Un composant unique particulier jouant le rôle source (ou `source`) sert à l'initialisation du médium, c'est lui qui indique quel est l'ensemble des identificateurs à manipuler.

Les services offerts par le médium sont les suivants :

- Pour le rôle `source` :
  `void setReserveIdSet(ReserveId setId[], Boolean cancelerIsReserver)` : positionne l'ensemble des identificateurs. `setId` est l'ensemble des identificateurs et `cancelerIsReserver` précise si le composant qui annule une réservation doit être celui qui a fait cette réservation (valeur « vrai ») ou peut être n'importe quel composant jouant un rôle `reserver` (valeur « faux »).
- Pour le rôle `reserver` :
  `ReserveId reserve()` : retourne un identificateur disponible et l'en retire de l'ensemble des identificateurs disponibles à la réservation. Si aucun identificateur n'est actuellement disponible, retourne `null`.
  `Boolean cancel(ReserveId)` : annule une réservation précédemment faite. Le paramètre du service permet de préciser l'identificateur qui avait été réservé. L'annulation est validée si les trois conditions suivantes sont toutes vérifiées :
  - L'identificateur appartient à l'ensemble initial.
  - L'identificateur est réservé (c'est-à-dire qu'il n'est pas disponible à la réservation et n'appartient donc pas à l'ensemble des identificateurs actuellement disponibles).
  - Si `cancelerIsReserver` est à vrai, le composant qui fait l'annulation doit être celui qui avait fait la réservation.

  Si l'annulation est validée, le service retourne vrai et l'identificateur passé en paramètre est à nouveau disponible à la réservation. Sinon le service renvoie faux.

Les services requis par le médium sont les suivants :

– Les composants jouant le rôle `observer` doivent implémenter ce service : `void nbAvailableId(Integer newValue)` : ce service est appelé par le médium à chaque fois que le nombre d'identificateurs disponibles change. Le paramètre précise ce nouveau nombre.

Diagramme de collaboration

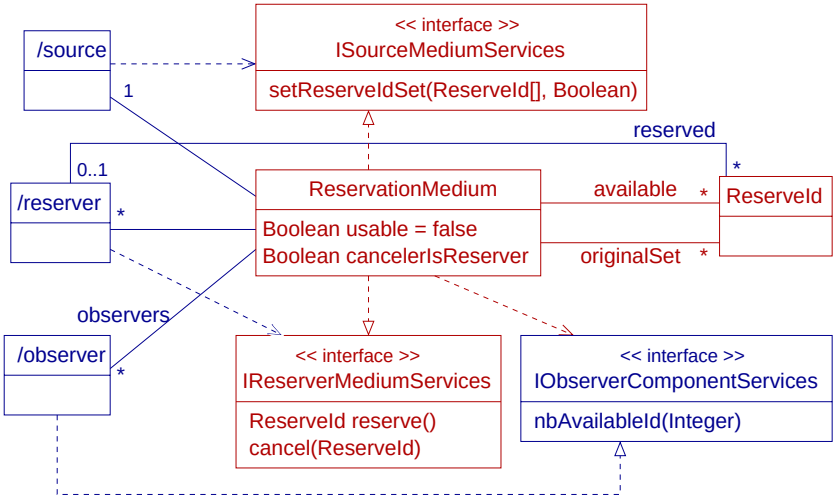

FIG. 2.17: Diagramme de collaboration du médium de réservation

Le diagramme de collaboration du médium de réservation est représenté sur la figure 2.17. La classe `ReservationMedium` représente le médium et les trois rôles sont présents. On y retrouve aussi les trois interfaces de services : les deux offertes (`ISourceMediumServices` pour le rôle `source` et `IReserverMediumServices` pour le rôle `reserver`) et l'interface requise (`IObserverComponentServices`) chez le rôle `observer`. Le nombre de rôles `observer` et `reserver` est quelconque (multiplicité « * ») alors que le nombre de rôles `source` est de un et uniquement un.

Un identificateur est une instance de la classe `ReserveId`. Le médium gère deux ensembles d'identificateurs : `originalSet` qui permet de garder une référence sur l'ensemble initial et `available` qui contient les identificateurs disponibles à la réservation (c'est un sous-ensemble de l'ensemble initial). Chaque rôle `reserver` a une référence sur l'ensemble des identificateurs qu'il a réservé via le lien `reserved`.

La figure 2.18 page suivante est la vue dynamique du médium de réservation. Elle montre les relations entre les différents services en ce qui concerne la notification aux rôles `observer` du fait que le nombre d'identificateurs disponibles a changé. Cela se fait par l'appel du service `nbAvailableId` dont le paramètre est à chaque fois `available -> size` qui est une expression OCL exprimée dans le contexte de la classe `ReservationMedium` (car c'est de cette classe que part le message). Elle renvoie la taille de l'ensemble `available`, c'est-à-dire le nombre d'identificateurs disponibles.

Les trois services offerts peuvent amener à l'appel de cette opération. Plus précisément, si l'appel de ces opérations se fait dans certaines conditions, l'opération est appelée. C'est pour cela qu'il y a des gardes devant les trois messages A.2, B.2 et C.2. L'opération

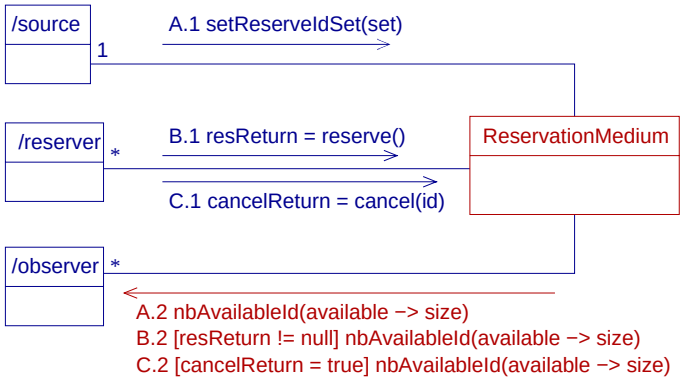

Fig. 2.18: Vue dynamique de la collaboration du médium de réservation

est appelée quand :

- L'initialisation de l'ensemble est fait par l'appel de `setReserveIdSet` (message A.1). Il faut donc informer tous les observateurs du nombre d'identificateurs disponibles (message A.2).
- Une réservation est faite par l'appel de `reserve` (message B.1). Si cette réservation est effective, c'est-à-dire si l'appel de l'opération ne renvoie pas `null`, alors le nombre d'identificateurs disponibles a changé (message B.2).
- Une annulation de réservation est faite par l'appel de `cancel` (message C.1). Si cette annulation est valide (l'opération retourne `true`) alors le nombre d'identificateurs disponibles a changé (message C.2).

Spécifications OCL

Les spécifications OCL des services offerts par le médium sont les suivantes.

Opération setReserveIdSet(ReserveId[ ], Boolean). Ce service sert à initialiser le médium. Une fois qu'il a été appelé, les services du médium (offerts aux rôles `reserver`) deviennent utilisables car les réservations peuvent être effectuées. Il ne peut être appelé qu'une seule fois. L'ensemble des identificateurs passés en paramètre sert à initialiser les ensembles `originalSet` et `available` du médium. Voici la spécification de ce service :

```
context ReservationMedium::setReserveIdSet(Set idSet, Boolean cancel)
pre: usable = false                                  -- le service n'a pas encore été appelé
post:
    originalSet = idSet
    and available = idSet
    and cancelerIsReserver = cancel
    and reserver -> forAll( r | r.reserved -> isEmpty )  -- aucun identificateur n'est réservé
    and usable = true                                    -- le médium devient utilisable
```

Opération ReserveId reserve(). Cette opération retourne un identificateur de l'ensemble des identificateurs disponibles et l'en retire. Si aucun identificateur n'est disponible, la valeur `null` est renvoyée. L'appel de cette opération ne peut se faire que

si le médium est dans un état utilisable. Voici la spécification de ce service :

```
context ReservationMedium::reserve() : ReserveId
pre: usable = true                                  -- le médium est dans un état utilisable
post:
    if available -> isEmpty                         -- il n'y a plus d'identificateur disponible
    then result = null
    else
        originalSet -> includes(result)
        and available@pre -> includes(result)       -- l'identificateur était disponible
        and available -> excludes(result)                          -- et ne l'est plus
        and caller.reserved -> includes(result)     -- marque que le composant
                                -- appelant l'opération a résérvé l'identificateur retourné
    endif
```

Opération cancel(ReserveId). Cette opération annule la réservation d'un identificateur (qui est passé en paramètre de l'opération) faite précédemment. Elle renvoie vrai si l'annulation est valide (voir la description des services) et dans ce cas, l'identificateur est à nouveau disponible à la réservation. Sinon, elle retourne faux. Voici la spécification de ce service :

```
context ReservationMedium::cancel(ReserveId id) : Boolean
pre: usable = true
post:
    if ( originalSet -> excludes(id) or available@pre -> includes(id) or
                                -- id appartient à l'ensemble de départ et est déjà réservé
        (cancelerIsReserver and caller.reserved@pre -> excludes(id)) )
                -- si cancelerIsReserver=vrai, le composant appelant est celui qui a réservé id
    then result = false
    else
        available = available@pre -> including(id)          -- id est à nouveau réservable
        and reserver -> forAll( r | r.reserved -> excludes(id))
                                                    -- aucun réserveur ne réserve plus id
        and result = true
    endif
```

Le premier test est en « opposition » par rapport aux commentaires. Les commentaires parlent de ce qui doit être vérifié alors que le test vérifie la négation de ces conditions car cela concerne le cas ou l'opération retourne `false`.

Diagrammes d'états

Nous allons avoir besoin d'utiliser le diagramme d'états générique du médium (voir la section 2.3.2 page 60). Il faut en effet, quand le médium est dans un état utilisable, que nous informions chaque nouveau rôle observateur du nombre d'identificateurs disponibles dès qu'il se connecte au médium.

Il n'est par contre pas nécessaire de s'intéresser au cas où des rôles observateurs se connectent au médium alors que les services de celui-ci ne sont pas encore utilisables. Ce

cas est pris en compte dans l'interaction décrite par la vue dynamique de la collaboration (voir la figure 2.18 page 72). Quand le service `setReserveIdSet` est appelé, le service `nbAvailableId` est appelé chez tous les composants observateurs alors présents.

La spécification de l'opération `roleConnection` du diagramme d'états générique est donc la suivante :

context ReservationMedium::roleConnection(Role role)
post: (oclInState(Running::Usable) **and** role.isKindOf(observer))
implies role.*oclCallOperation(nbAvailbleId, available –> size)*

Cette spécification décrit que si le médium est dans un état utilisable et si le composant qui se connecte est de rôle observateur, alors cela implique que la méthode `nbAvailableId` doit être appelée sur ce rôle avec en paramètre la taille de l'ensemble des identificateurs disponibles.

2.4.4 Conclusion sur ces exemples de spécification

Ces trois exemples complets de spécification de médiums ont pour but de montrer comment notre méthodologie de spécification s'applique. Nous avons montré la plupart des points importants de notre méthodologie dans ces exemples : la structure de la collaboration avec les interfaces de services offerts et requis, la généralisation de l'utilisation d'OCL (dans les gardes de messages, des transitions d'états et pour les paramètres des services), l'utilisation de nos deux extensions d'OCL et la gestion de la connexion et de la déconnexion dynamiques de composants au médium.

2.5 Déploiement d'un composant de communication

Dans cette section nous décrivons l'architecture de déploiement de médiums que nous avons définie. Ensuite, nous décrivons le diagramme d'états générique du médium adapté en fonction de cette architecture de déploiement.

2.5.1 Architecture de déploiement d'un médium

Un médium réifie ou implémente un système d'interaction ou un protocole de communication de tout niveau et de toute complexité. L'utilisation de ce système ou de ce protocole se fait via l'appel sur le médium d'opérations regroupées en interfaces de services offerts (et requis quand le médium appelle des opérations sur un composant). Les composants qui communiquent entre eux ne le font pas directement, ils passent par ces opérations. Une communication directe entre deux ou plusieurs composants distants n'est jamais nécessaire s'ils sont interconnectés via des médiums. Un composant n'a donc pas besoin de connaître la localisation exacte des composants avec qui il communique (voire même dans certains cas, de savoir combien ils sont). C'est donc au médium de gérer tous ces problèmes de localisation des composants distants et des communications « physiques » entre eux. Le composant connecté à un médium se contente d'appeler localement un service sur ce dernier pour réaliser une communication ou une interaction avec les autres composants connectés au même médium que lui.

Pour être capable d'offrir un service local à un composant et de gérer la distribution de ces composants, le médium doit être composé de plusieurs éléments distribués ; chacun de

ses éléments étant localement associé à un composant connecté au médium. La fonction de l'élément dépend du rôle joué par le composant connecté ; de ce rôle vont dépendre les services implémentés par cet élément. Ces éléments sont pour cela appelés gestionnaires de rôle.

Un médium définit un gestionnaire de rôle différent pour chacun des rôles de composant possibles. Un médium déployé est le regroupement logique d'un ensemble de gestionnaires distribués. À chaque composant connecté est associé localement un gestionnaire. La connexion d'un composant à un médium correspond à l'instantiation du gestionnaire associé à ce composant. Cette architecture peut être vue comme le résultat du regroupement des morceaux de code spécifiques à une interaction comme nous l'avons expliqué dans la section 2.1.1 page 48. Afin de réaliser l'interaction réifiée dans le médium, ces gestionnaires communiquent entre eux via un intergiciel ou toute autre technologie.

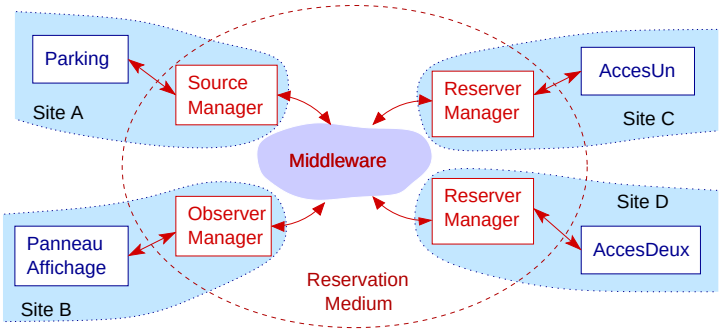

FIG. 2.19: Architecture de déploiement du médium de réservation

La figure 2.19 montre l'architecture de déploiement du médium de réservation dans le cadre de l'application de gestion de places de parking (voir la section 2.1.2 page 51). Ce médium accepte trois rôles différents donc trois types de gestionnaires de rôle différents forment le médium :

- Un gestionnaire de source (nommé `SourceManager`) est associé au composant jouant le rôle source, ici le composant `Parking`.
- Un gestionnaire de réserveur (nommé `ReserverManager`) pour les composants jouant le rôle réserveur, ici les deux accès.
- Un gestionnaire d'observateur (nommé `ObserverManager`) pour les composants jouant le rôle observateur, ici le composant `PanneauAffichage`.

Chaque gestionnaire a la responsabilité d'implémenter les services offerts à son rôle associé et également d'appeler les services requis sur ce rôle. Les gestionnaires peuvent être aussi complexes que cela est nécessaire.

Sur chaque site, le composant est déployé avec son gestionnaire associé. Un médium est donc différent du point de vue du déploiement d'un composant classique. Ce n'est pas un composant monolocalisé et instantié en une seule fois mais un ensemble de gestionnaires distribués et logiquement cohérents. Si la connexion et la déconnexion dynamiques de composants au médium est possible, alors de nouveaux gestionnaires peuvent être ajoutés ou supprimés du médium de manière dynamique, pendant l'exécution d'une application. La structure interne d'un médium peut donc évoluer dynamiquement, en ce qui concerne la présence et le nombre de gestionnaires présents.

C'est pour cela que la notion d'utilisabilité d'un médium est importante. Comme nous

sommes dans un contexte distribué, tous les gestionnaires formant le médium ne peuvent pas être instantiés simultanément de manière atomique. Un médium n'est utilisable que si certains gestionnaires indispensables à son fonctionnement sont présents (et correctement initialisés)[6]. Par exemple, dans le cadre de l'application de gestion de places de parking, la réservation d'une place de parking (par un composant réserveur) n'a de sens que si l'ensemble des identificateurs a été initialisé, ce qui implique la présence du gestionnaire de source dans le médium et son initialisation (par l'appel de `setReserveIdSet`).

Cette architecture présente donc plusieurs avantages. Tout d'abord elle permet de conserver l'unité et la cohérence de l'abstraction de communication réifiée par le médium. Les services appelés par un composant et requis par le médium sont les mêmes que ceux définis lors de la spécification abstraite. Ensuite, il n'y a aucune contrainte sur la réalisation des gestionnaires. Ils peuvent être de toute nature ou de toute complexité, tant que les services offerts et requis sont gérés et respectent la sémantique définie lors de la spécification. Ainsi, il est possible de réaliser plusieurs implémentations différentes d'une même abstraction de communication.

2.5.2 Gestion de la dynamicité et du cycle de vie des gestionnaires

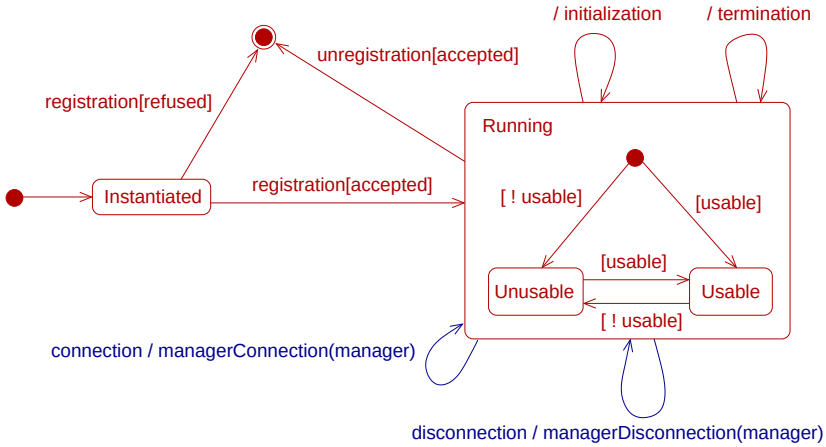

Fig. 2.20: Diagramme d'états générique d'un gestionnaire de rôle

Le diagramme d'états de la figure 2.20 représente le cycle de vie d'un gestionnaire, depuis son instantiation jusqu'à sa terminaison. Ce diagramme d'états est une évolution du diagramme d'états générique d'un médium (voir la section 2.3.2 page 60) en prenant en compte les spécificités dues aux gestionnaires de rôle.

Un gestionnaire peut voir son appartenance à un médium refusée en fonction du contrat du médium. Typiquement, si une seule connexion d'un rôle de composant et donc d'un gestionnaire donné est autorisée, quand un deuxième gestionnaire de ce même type

[6] Dans la méthodologie de spécification, l'utilisabilité du médium est définie en fonction de l'état du médium (au niveau de la classe représentant le médium dans sa globalité). Dans le chapitre suivant, nous introduisons des niveaux de spécification où les gestionnaires remplacent le médium. Dans ces cas là, l'utilisabilité du médium est définie en fonction de l'état des gestionnaires. Cependant, à tous les niveaux, l'utilisabilité du médium représente toujours les mêmes contraintes, seul le contexte de l'expression de ces contraintes change.

est instantié, il ne pourra pas faire partie du médium[7]. C'est pour cela que le gestionnaire est d'abord dans un état d'instantiation (état `Instantiated`) avant de passer dans un état de fonctionnement normal (état `Running`). Il passe dans cet état si sa demande d'appartenance au médium (via l'appel de `registration`) est acceptée. Sinon il passe directement dans un état de terminaison.

Dans l'état de fonctionnement normal, le médium peut être dans un état utilisable (c'est-à-dire que les services offerts par les gestionnaires peuvent être appelés par les composants connectés au médium) ou inutilisable, qui correspondent respectivement aux états `Usable` et `Unusable`. Le passage d'un état à un autre dépend de la valeur de la propriété `usable` du médium.

Un gestionnaire peut être initialisé au besoin (opération `initialization`) et peut effectuer quelques actions avant de se terminer (opération `termination`). Pour préciser cela, il suffit de définir la sémantique associée à ces opérations.

Un gestionnaire peut avoir besoin de savoir que de nouveaux gestionnaires font partie du médium ou qu'ils en sont déconnectés. Il en est informé lorsque les événements `connection` et `disconnection` sont envoyés. Là encore, il faut, pour gérer ces événements, définir des actions qui leur sont associées. Il est aussi possible de définir en OCL la sémantique des opérations correspondant à ces événements sur le diagramme d'états (opérations `managerConnection` et `managerDisconnection` qui prennent en paramètre la référence sur le gestionnaire connecté ou déconnecté).

Ce diagramme d'états est générique et adaptable en fonction des besoins des médiums pour déclencher localement dans un gestionnaire les actions nécessaires en fonction des occurrences de certains événements. Il est utile pour les spécifications où l'élément représentant globalement le médium est remplacé par l'ensemble des gestionnaires. Nous verrons cela dans le prochain chapitre qui traite du processus de raffinement.

2.6 Conclusion

Un composant de communication ou médium est donc la réification d'une abstraction de communication ou d'interaction dans un composant logiciel. Comme un composant logiciel prend plusieurs formes en fonction du niveau auquel on le manipule (spécification abstraite, spécification d'implémentation, implémentation, déploiement, exécution, etc.), un composant de communication permet de manipuler une même abstraction de communication à tous ces niveaux.

Dans ce chapitre, nous nous sommes principalement intéressés à deux niveaux. Tout d'abord, celui de la spécification abstraite. Le but est de définir le contrat d'usage du médium. C'est-à-dire de spécifier ce que fait le médium (mais sans dire comment) du point du vue de son utilisation. Pour cela, nous avons défini une méthodologie de spécification en UML basée principalement sur les collaborations UML. Une collaboration sert de base à la spécification d'un médium. La structure de cette collaboration suit certaines règles afin de s'adapter au contexte de la spécification de composants (les interfaces de services offerts et requis par le médium doivent par exemple être présentes). Afin de définir complètement le contrat de ce médium, nous utilisons également des contraintes OCL ainsi que des diagrammes d'états.

[7] Si l'appartenance d'un gestionnaire au médium est refusée, cela signifie que la connexion au médium de son composant associé est refusée.

Le deuxième aspect de ce chapitre concerne le niveau le plus bas, à savoir l'instantiation et le déploiement d'un médium. Nous proposons une architecture de déploiement basée sur un groupe d'objets distribués formant un ensemble cohérent. Cette architecture permet d'implémenter facilement un médium ou différentes variantes de ce médium.

Le problème qui se pose alors est de savoir comment faire le lien entre cette spécification abstraite et l'architecture de déploiement. C'est-à-dire comment implémenter une abstraction de communication à partir de sa spécification. Pour cela, nous avons défini un processus de raffinement transformant cette spécification abstraite en une ou plusieurs spécifications d'implémentation prenant en compte notre architecture de déploiement. C'est ce processus de raffinement que nous détaillons dans le prochain chapitre.

Chapitre 3

De la spécification abstraite à l'implémentation : définition d'un processus de raffinement

Le processus de raffinement que nous présentons dans ce chapitre a pour but de passer de la spécification abstraite d'un composant de communication à une ou plusieurs spécifications d'implémentation en fonction de choix de conception ou pour gérer différentes contraintes non-fonctionnelles. Pendant la phase de déploiement, un médium est constitué d'un ensemble de gestionnaires distribués. La spécification d'implémentation doit se baser sur cette architecture. Le but du processus est donc globalement de transformer la classe unique représentant le médium au niveau de la spécification abstraite en un ensemble de gestionnaires. Le processus est constitué de plusieurs étapes, chacune transformant la spécification de l'étape précédente en une nouvelle spécification. Dans l'ordre, les étapes du processus sont les suivantes :

- Faire apparaître la classe représentant le médium comme l'agrégation des gestionnaires de rôle.
- Définir des spécifications d'implémentation. Pour cela, il s'agit de faire disparaître la classe représentant le médium et ainsi spécifier le médium uniquement à l'aide des gestionnaires. Cette étape implique la prise de choix de conception et d'implémentation. Plusieurs spécifications de ce niveau peuvent être définies pour un même médium.
- Pour chaque spécification d'implémentation, définir un ou plusieurs choix de déploiement.

Lors de l'application du processus, le contrat du médium, c'est-à-dire la spécification au niveau abstrait, doit – autant que faire se peut – être respecté à chaque étape du processus. La sémantique des services offerts et du médium doit rester la même pendant toute l'application du processus. Il est tout de même possible lors d'une étape, de rajouter des contraintes pour certaines parties du contrat, notamment lors de l'application de choix de conception ou de déploiement. Il n'est pas possible de prouver formellement que le contrat est respecté pendant toute l'application du processus. Le processus de raffinement que nous présentons n'est pas complètement formel, au contraire de [80] par exemple. En effet, UML est un langage que l'on peut considérer comme uniquement semi-formel de par ses ambiguïtés et sa sémantique parfois floue ou indéfinie.

Afin de montrer en pratique l'utilisation du processus de raffinement et de ses différentes étapes, nous l'appliquons sur la spécification du médium de réservation d'identificateurs (voir la section 2.4.3 page 70). Ce médium gère un ensemble de données. Le principal problème est de gérer ces données dans un contexte distribué lors de l'implémentation. Parmi les nombreux choix de gestion possibles, nous en définissons deux : le premier avec une gestion centralisée des données et le second avec une gestion complètement distribuée. Une spécification d'implémentation est définie pour chacun de ces choix.

Avant de présenter chacune des étapes du processus de raffinement, nous parlons de l'approche *Model-Driven Architecture* [88] de l'OMG car notre processus s'inscrit dans cette approche en en étant un exemple d'application.

3.1 L'approche Model-Driven Architecture de l'OMG

L'approche Model-Driven Architecture [89, 28, 95] – ou MDA en abrégé – a pour but de définir une nouvelle façon de spécifier et de concevoir des applications en recentrant la problématique sur la spécification des fonctionnalités métiers plutôt que sur les techniques d'implémentation. Cette approche est le nouveau fer de lance de l'Object Management Group (OMG) qui travaille depuis plus de dix ans à la définition de standards basés sur les technologies objets.

3.1.1 Les origines du MDA

Une des contributions majeures de l'OMG à l'industrie du logiciel est la norme CORBA (pour *Common Object Request Broker Architecture*) [86, 110, 49]. Cette norme définit un standard d'interopérabilité entre objets distribués, indépendamment du système d'exploitation ou du langage de programmation utilisé. De nombreux intergiciels basés sur cette norme existent aujourd'hui et CORBA est devenu un standard industriel en constante évolution. Récemment, l'OMG a défini un modèle de composant basé sur cet intergiciel, le CORBA Component Model (CCM) [85, 68].

Mais si ce succès est indéniable, force est de constater que CORBA n'est pas l'unique intergiciel utilisé et que CCM est loin d'être encore le standard des plates-formes de composants. En effet, d'autres plates-formes telles que les EJB de Sun [114, 3] ainsi que le nouveau .Net de Microsoft [79, 78, 112] sont largement utilisées dans l'industrie. Ainsi, en permanence, de nouveaux standards industriels pour l'implémentation d'applications distribuées arrivent, s'imposent et sont à leur tour petit à petit remplacés par de nouveaux arrivants. La plate-forme unique et universelle risque donc de ne jamais exister.

En partant de ce constat et en remarquant que, finalement, la définition des fonctionnalités métiers est plus importante que les problématiques d'implémentation, l'OMG s'est tourné vers l'approche MDA. Le but de cette approche est d'apporter une nouvelle manière de concevoir les applications en se basant sur une spécification abstraite indépendante de tout système, technologie ou plate-forme. Cette spécification est ensuite successivement transformée en de multiples autres spécifications jusqu'à la génération de l'application pour une plate-forme ou technologie donnée comme CORBA ou EJB par exemple. Ainsi, il s'agit de se focaliser sur la partie métier d'une application en faisant passer au second plan le choix de la technologie qui sera mise en œuvre.

3.1.2 Les grands principes du MDA

Le principe fondamental du MDA est de s'intéresser à la spécification d'une application à un niveau indépendant de toute implémentation ou technologie. On parle alors de modèle au niveau PIM pour *Platform Independent Model*. Ensuite, cette spécification PIM est transformée en d'autres spécifications pour aboutir notamment à une ou plusieurs spécifications dépendantes de choix technologiques ou d'architecture. On parle alors de spécification au niveau PSM pour *Platform Specific Model*. Pour terminer, ces spécifications PSM servent de base à la génération de code correspondant à une technologie donnée pour cette application.

L'intérêt de cette approche est donc clair : toute la complexité de l'application est dans la définition de la spécification PIM, c'est-à-dire dans les règles et les concepts métiers. Ensuite, à l'aide de transformations de modèles et de génération de code, cette spécification PIM est dérivable pour plusieurs plates-formes ou technologies comme EJB ou .Net par exemple. De cette façon, la dépendance vis-à-vis d'une technologie est bien moins importante que précédemment car le savoir faire est recentré sur la spécification PIM au lieu de la technologie d'implémentation. Cela permet également de réutiliser les concepts fondamentaux d'une application et aussi de faciliter la transition entre deux technologies car seules les transformations vers les niveaux PSM sont à revoir, le niveau PIM ne variant pas.

Les transformations de modèles sont de plusieurs niveaux :

- D'un modèle PIM à un autre modèle PIM : il s'agit d'un raffinement de la spécification mais en restant à un niveau abstrait et indépendant de toute plate-forme.
- D'un modèle PIM à un modèle PSM : c'est la projection du modèle abstrait vers un modèle respectant une architecture ou une technologie donnée.
- D'un modèle PSM à un modèle PSM : il s'agit du raffinement de spécifications dépendantes d'une certaine plate-forme. Dans cette catégorie entre aussi la génération de code à partir d'un modèle PSM.

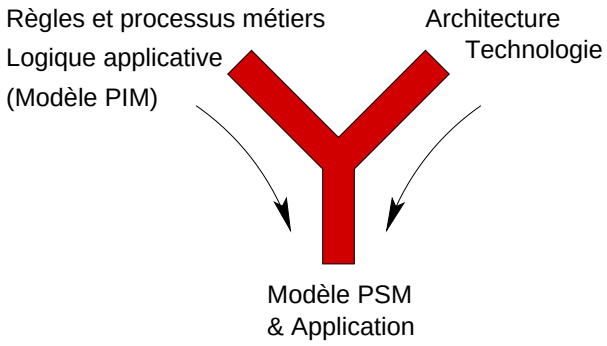

FIG. 3.1: Cycle en Y de l'approche MDA

Au niveau du cycle de vie et de développement du logiciel, l'utilisation du MDA se fait généralement en suivant un cycle dit « en Y », comme le montre la figure 3.1. En entrée du cycle, deux spécifications sont définies : la spécification PIM de l'application et celle de l'architecture et des technologies retenues pour une des implémentations de cette application. Ces deux spécifications sont ensuite « fusionnées », ce qui mène à la création d'une spécification de niveau PSM. La transformation de PIM vers PSM est donc faite

en tenant compte des choix d'architecture et de technologies définis dans la deuxième branche du Y.

3.1.3 Les éléments techniques du MDA

Afin de rendre le MDA utilisable en pratique, sa mise en œuvre se base sur certains standards de l'OMG et sur d'autres normes qui sont actuellement en cours de définition.

Pour les spécifications PIM et PSM, le langage utilisé pour définir les différents modèles de chaque niveau est principalement UML. UML est donc un des éléments fondamentaux de la mise en œuvre du MDA. Au niveau PSM, pour une plate-forme ou une technologie donnée (comme CORBA ou EJB par exemple), des profils particuliers peuvent être définis pour bien prendre en compte les concepts et caractéristiques de chaque plate-forme. Un profil UML définit un ensemble de règles et de contraintes pour un domaine donné. Un profil CORBA décrit par exemple comment représenter et manipuler en UML les concepts de CORBA.

Le deuxième outil essentiel du MDA est le MOF (pour *Meta Object Facilities*) [87]. Le MOF est un standard de méta-modélisation. Il définit quatre niveaux de modélisation, correspondant au méta-méta-modèle, aux méta-modèles, aux modèles et aux instances de modèles. Le méta-modèle de UML est par exemple une instance du méta-méta-modèle défini dans le MOF. Un des buts du MOF est de permettre le passage entre différents méta-modèles et ainsi l'interopérabilité entre différents langages de modélisation.

Afin de pouvoir en pratique passer d'une spécification à une autre, ou d'un méta-modèle à un autre, le MOF et le MDA s'appuie sur XMI (pour *XML Metadata Interchange*) [90]. XMI est le langage utilisé par les outils pour représenter tout type de modèle et ainsi pouvoir les échanger.

L'OMG travaille aussi à la définition de services définis à un niveau abstrait. En effet, lorsque vous utilisez CORBA, vous pouvez vous appuyez sur des services tels que la persistance, les transactions, la sécurité, les événements ou le service de nommage. Des profils UML permettront d'utiliser ces services lors de la spécification d'une application mais à un niveau PIM (et non pas à un niveau PSM comme cela serait le cas avec les services CORBA), c'est-à-dire sans s'appuyer sur une implémentation particulière de ces services.

3.1.4 L'approche MDA dans notre processus de raffinement

Comme nous l'avons dit en introduction de ce chapitre, notre processus de raffinement sert à transformer une spécification abstraite de médium en une ou plusieurs spécifications d'implémentation prenant en compte des choix de conception et notre architecture de déploiement de médium. Le parallèle avec l'approche MDA est aisé : la spécification abstraite d'un médium correspond à une spécification de niveau PIM et une spécification d'implémentation à une spécification de niveau PSM. Nous retrouvons également le cycle en Y avec en entrée la spécification abstraite du médium comme spécification PIM et notre architecture de déploiement de médium comme le choix de plate-forme et d'architecture.

Dans la suite de ce chapitre, nous détaillons une à une les étapes de notre processus de raffinement. Afin de montrer en pratique ce que fait chaque étape, nous appliquons le processus à la spécification du médium de réservation précédemment définie.

3.2 Première étape : introduction des gestionnaires de rôle

3.2.1 Objet et principes de la première étape

La première phase du processus de raffinement consiste à représenter le médium comme une agrégation de gestionnaires de rôle. Auparavant, dans la version abstraite, le médium était représenté par une unique entité. Or, comme nous l'avons vu dans la section 2.5.1 page 74, un médium n'est jamais déployé sous cette forme. La première étape consiste donc à faire apparaître les types de gestionnaires qui forment le médium lors de son implémentation et de son déploiement.

Le but de cette étape n'est pas de faire disparaître complètement la classe représentant le médium sous forme unique mais de la montrer comme étant le résultat de l'agrégation de plusieurs gestionnaires. Cette modification est effectuée sur la vue structurelle du diagramme de collaboration. Elle entraîne alors des répercutions sur les autres diagrammes et contraintes OCL utilisés pour spécifier le médium.

Les modifications de la spécification abstraite

Les modifications structurelles de la collaboration sont les suivantes :

- Pour chaque rôle « <Role> », ajout d'une classe représentant le gestionnaire dont le nom est « <Role>Manager ». Elle possède un lien vers son rôle associé et vers le médium. Le lien vers son rôle associé est d'une multiplicité un vers un. Le lien entre ce gestionnaire et la classe représentant le médium est un lien d'agrégation. Le médium est donc composé de l'ensemble des gestionnaires. La multiplicité de l'ancien lien entre le médium et un rôle (du côté du rôle) est reportée sur le nouveau lien entre son gestionnaire et le médium (du côté du gestionnaire). Cela permet de conserver les informations sur le nombre de connexions autorisées pour certains types de composant. Les liens entre les rôles et le médium sont supprimés. Chaque gestionnaire dispose d'un attribut nommé `component` qui est la référence sur son composant associé.
- Au niveau des interfaces de services offerts aux rôles et requis sur les rôles, une interface de services offerts par un médium dont dépendait un rôle est maintenant implémentée par le gestionnaire associé à ce rôle à la place du médium. Si le médium avait une dépendance sur une interface implémentée par un rôle, cette dépendance est déplacée sur le gestionnaire associé à ce rôle.
- Les références qu'avaient les rôles sur des objets, des données « internes » au médium (et gérées par lui) sont déplacées sur le gestionnaire associé au rôle. Les propriétés locales à une liaison composant/médium sont déplacées dans le gestionnaire en tant que propriétés du gestionnaire. Un rôle est donc bien découplé du reste du médium, il n'a plus de dépendances sur le médium que via les interfaces de services.

La relation générique entre un rôle, un gestionnaire et le médium est représentée sur la figure 3.2 page suivante. Le « ? » de l'association entre le gestionnaire `<RoleName>Manager` et la classe `<MediumName>Medium` est remplacé dans une vraie description par le nombre de composants du rôle `<RoleName>` qui peuvent se connecter au médium. Il doit être identique à la cardinalité notée du coté du rôle sur l'association entre le rôle et le médium pour la collaboration de la spécification abstraite.

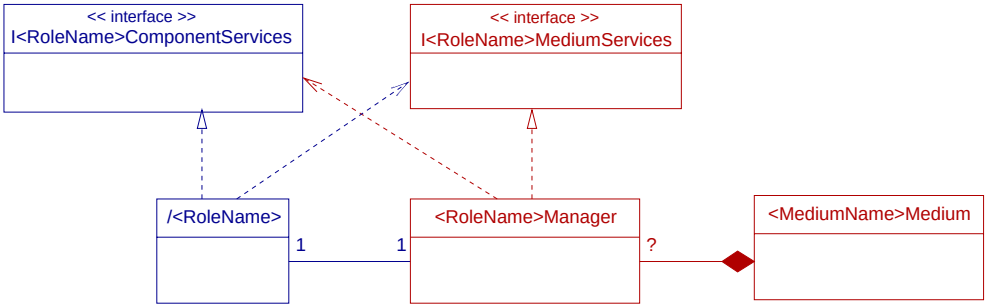

Fig. 3.2: Relation générique entre un rôle, un gestionnaire et un médium

Cette modification de la structure de la collaboration va bien entendu impliquer des changements sur la vue dynamique de la collaboration, sur les contraintes OCL et les diagrammes d'états.

Pour les contraintes et spécifications des services en OCL, le changement consiste à modifier la navigation à travers les instances. En effet, la modification apportée est l'ajout d'une classe intermédiaire entre la classe représentant le médium et chacun des rôles. La navigation doit donc prendre en compte ces nouvelles classes. Comme les interfaces de services offerts et requis ne sont plus implémentées par la même classe, il faut que les contraintes OCL spécifiant les services offerts par le médium soient exprimées dans un autre contexte (celui du gestionnaire adéquat). Dans la spécification d'un service offert, si la référence vers un objet géré par le médium était directe dans l'expression, elle se fait maintenant à travers l'objet représentant le médium. Si la navigation débutait par le *caller*, celui-ci est supprimé. Si la navigation dans le contexte de la classe représentant le médium passait par un rôle, ce dernier est remplacé par son gestionnaire associé. Enfin, s'il faut accéder aux données manipulées par la classe représentant le médium, alors la navigation passe par cette classe.

Pour les diagrammes d'états, il y a deux changements qui peuvent intervenir. Tout d'abord, la navigation et le contexte des contraintes OCL qui sont utilisées dans un diagramme d'états. Elles doivent prendre en compte l'ajout des nouvelles classes comme nous venons de le voir. Ensuite, si le diagramme d'états générique d'un médium est utilisé, il est conservé en ce qui concerne la gestion de l'utilisabilité et de l'initialisation et la terminaison du médium. Si la connexion et la déconnexion de rôles de composant étaient gérées, il faut désormais s'intéresser à la connexion et à la déconnexion des gestionnaires associés à ces rôles, en utilisant le diagramme d'états générique d'un gestionnaire (ce diagramme d'états est associé à la classe représentant le médium). Si un diagramme d'états était associé au médium, il reste associé au médium. Si un diagramme d'états était associé à un service offert du médium, alors il reste associé à ce service mais dans le contexte du gestionnaire qui réalise ce service. Si un diagramme d'états était associé à un rôle ou un service requis sur ce rôle, il reste associé à ce rôle.

Pour la vue dynamique du diagramme de collaboration, les modifications concernent là encore les contraintes OCL et également l'émission des messages. Les messages émis par un rôle et à destination du médium sont maintenant à destination du gestionnaire associé à ce rôle. Les messages émis par le médium et à destination d'un rôle sont désormais émis par le gestionnaire associé au rôle.

Cette première étape est complètement automatisable. La modification de la structure de la collaboration se fait toujours de la même manière, sur tous les diagrammes de collaboration spécifiant un médium. De plus, la modification des contraintes et expressions OCL, ainsi que des diagrammes d'états, est elle aussi automatisable. En effet, la modification du diagramme de collaboration n'engendre que des modifications de la navigation dans ces contraintes et expressions OCL ou la modification des éléments auxquels sont rattachés les diagrammes d'états et les messages des collaborations.

Le respect du contrat du médium

La sémantique et le contrat du médium sont totalement préservés lors de cette première étape. La modification apportée est uniquement structurelle et la prise en compte de cette modification ne change que la navigation dans les contraintes OCL ou le contexte de ces contraintes, d'un diagramme d'états ou des messages du diagramme de collaboration. Cela ne modifie pas intrinsèquement la spécification des services, du moins pas du point de vue de leur sémantique.

3.2.2 Exemple d'application de la première étape

Afin de montrer un exemple d'application de cette première étape, nous étudions ici la transformation pour le médium de réservation. La spécification abstraite de ce médium qui sert de base à notre processus se trouve section 2.4.3 page 70.

Diagramme de collaboration

Le diagramme de collaboration transformé par l'introduction des gestionnaires est représenté sur la figure 3.3.

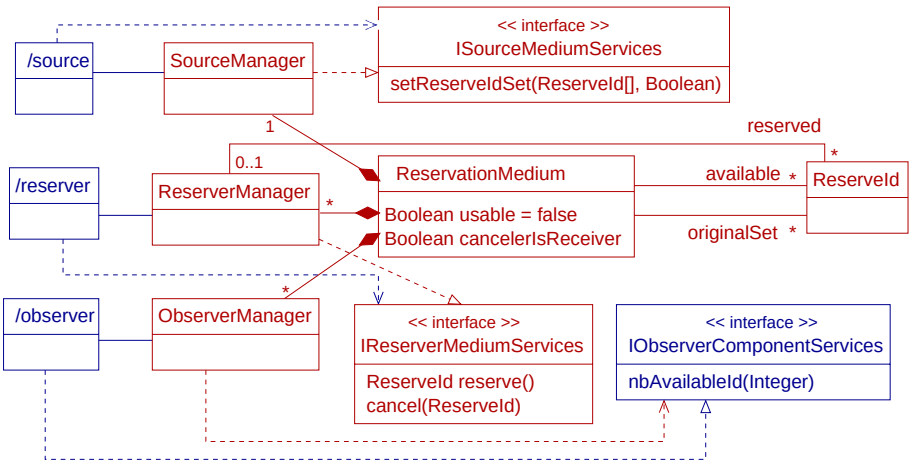

FIG. 3.3: Introduction des gestionnaires sur le diagramme de collaboration

Trois nouvelles classes sont donc rajoutées car le médium accepte trois rôles de composant. Ces classes sont `SourceManager`, `ReserverManager` et `ObserverManager` associées respectivement aux rôles `source`, `reserver` et `observer`. La classe `ReservationMedium` n'implémente plus d'interface et n'est plus dépendante d'aucune interface. Ce sont les

différents gestionnaires qui ont la responsabilité d'implémenter les services offerts à leur rôle ainsi que d'appeler chez eux les services requis. Les cardinalités entre les rôles et le médium sont désormais précisées sur le lien entre chaque gestionnaire et le médium.

Le lien `reserved` qui permettait de savoir quels étaient les identificateurs réservés par un rôle réserveur est maintenant associé au gestionnaire de réserveur. Ainsi, les seules dépendances entre un composant (un rôle) et un médium se situent au niveau de l'appel des services offerts et requis via les gestionnaires. Après leur introduction, les attributs et données gérés par le médium le sont entièrement « dans le médium ». Auparavant, pour spécifier le médium, l'association `reserved` impliquait que la connaissance des identificateurs réservés se trouve chez le rôle. Maintenant, cette connaissance est déportée vers le gestionnaire de réserveur et il n'y a plus aucun lien de ce type entre un rôle et un élément « interne » à la spécification du médium.

Le fonctionnement du médium peut ainsi être complètement spécifié sans contrainte ou relation sur les composants utilisant le médium (excepté bien sûr les appels des services offerts et requis). Le découplage entre les composants et le médium est donc complet. La spécification du médium devient « auto-contenue ».

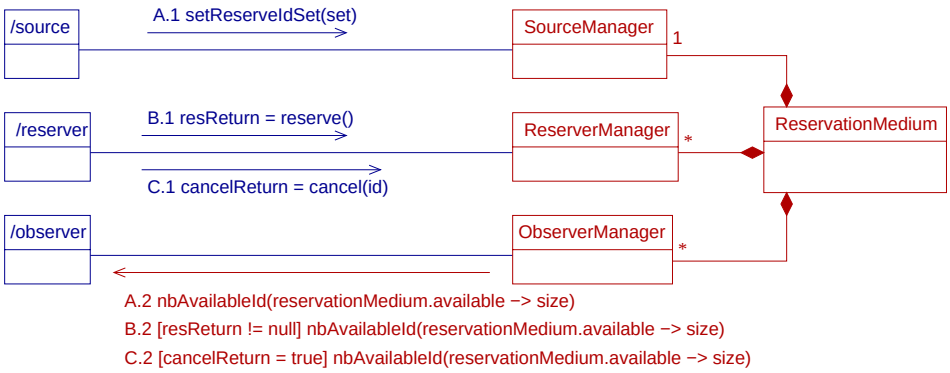

Fig. 3.4: Vue dynamique après l'introduction des gestionnaires

La partie dynamique de la collaboration est modifiée en conséquence comme le montre la figure 3.4. Une chose intéressante à noter est qu'il y a une « rupture » dans la chaîne d'appel. Par exemple, si le rôle réserveur appelle la méthode `reserve` sur la classe `ReserverManager` cela implique l'appel de `nbAvailableId` par les classes `ObserverManager` chez tous les rôles observateurs. Si l'on regarde les liens structurels, cet appel doit transiter obligatoirement via l'objet `reservationMedium`. Cela implique d'ajouter deux appels de méthodes supplémentaires. Mais ces appels ne sont pas utiles pour spécifier l'interaction entre les rôles, c'est pourquoi ils n'apparaissent pas.

Pour expliquer cela d'une autre manière, le lien d'agrégation entre les gestionnaires et la classe représentant le médium font que toutes ces objets n'en forment qu'un et que la classe `ReservationMedium` est uniquement un gestionnaire de données.

Nous pouvons noter la modification de l'expression OCL utilisée dans les messages A.2, B,2 et C.2 pour déterminer la taille de l'ensemble des identificateurs. L'expression référence cet ensemble en « naviguant jusqu'à » la classe `ReservationMedium`. En effet, le contexte de l'expression est la classe `ObserverManager` et non plus la classe `ReservationMedium` comme précédemment.

Spécifications OCL

La spécification du service `setReserveId` est – comme le montre la spécification ci-dessous – identique à celle faite au niveau le plus abstrait, à la différence de l'accès aux différents instances ou données qui se fait maintenant via l'objet `reservationMedium`.

```
context SourceManager::setReserveIdSet(Set idSet, Boolean cancel)
pre:
    reservationMedium.usable = false                    -- le service n'a pas encore été appelé
post:
    reservationMedium.originalSet = idSet
    and reservationMedium.available = idSet
    and reservationMedium.usable = true
    and reservationMedium.cancelerIsReserver = cancel
    and reservationMedium.reserverManager -> forAll( r | r.reserved -> isEmpty )
```

À part la navigation qui change pour accéder aux différents objets et attributs, les spécifications des services `reserve` et `cancel` sont identiques à celles de la spécification au niveau abstrait, avant l'introduction des classes représentant les gestionnaires de rôle.

Voici la spécification du service `reserve` :

```
context ReserverManager:reserve() : ReserveId
pre:
    reservationMedium.usable = true                              -- le médium est utilisable
post:
    if reservationMedium.available -> isEmpty    -- il n'y a plus d'identificateurs disponibles
    then result = null
    else
        reservationMedium.originalSet -> includes(result)
        and reservationMedium.available@pre -> includes(result)
        and reservationMedium.available -> excludes(result)
        and reserved -> includes(result)
    endif
```

Ce que l'on peut noter dans la spécification de ce service, c'est que maintenant il n'y a plus de référence au *caller* pour l'accès à l'ensemble `reserved`. Cet attribut est maintenant géré par le gestionnaire de réserveur associé au rôle qui demande à effectuer des réservations.

Voici la spécification du service `cancel` :

```
context reserverManager::cancel(ReserveId id) : Boolean
pre:
    reservationMedium.usable = true
post:
    if ( ReservationMedium.originalSet -> excludes(id) or
        reservationMedium.available@pre -> includes(id) or
        (reservationMedium.cancelerIsReserver and reserved@pre -> excludes(id)) )
    then result = false
    else
        reservationMedium.available =
            reservationMedium.available@pre -> including(id)
        and reservationMedium.reserverManager -> forAll( r |
            r.reserved = -> excludes(id))
        and result = true
    endif
```

Diagrammes d'états

Pour les diagrammes d'états, l'introduction des gestionnaires entraîne là encore peu de modifications. Pour notre médium de réservation, le principe quant à lui ne change pas : il s'agit d'informer les composants observateurs qui se connectent du nombre d'identificateurs disponibles lorsque le médium est utilisable. Mais maintenant, il ne s'agit pas de gérer la connexion du composant directement, mais du gestionnaire qui lui est associé. L'arrivée d'un nouveau gestionnaire dans le médium correspond en effet à la connexion d'un nouveau composant. Nous allons donc nous baser sur le diagramme d'états générique des gestionnaires de rôle (voir section 2.5.2 page 76) au lieu de celui des médiums (celui de la section 2.3.2 page 60). Il faut maintenant spécifier la sémantique de l'opération `managerConnection()` à la place de `roleConnection()`. Le diagramme d'états est toujours associé à la classe représentant la médium. Voici cette spécification :

```
context ReservationMedium::managerConnection(Manager manager)
post: (oclInState(Running::Usable) and manager.isKindOf(ObserverManager))
            implies manager.component.oclCallOperation(nbAvailableId, available -> size)
```

Dans cette spécification, le composant n'est plus référencé par l'attribut `role` passé en paramètre comme dans la spécification abstraite, mais par l'attribut `component` du gestionnaire (l'attribut `manager`) passé en paramètre.

3.3 Deuxième étape : choix de conception

3.3.1 Objet et principes de la deuxième étape

La seconde étape du processus consiste, à partir de la spécification obtenue lors de la première étape, à faire disparaître la classe représentant le médium dans son ensemble. Le but est maintenant de spécifier le médium uniquement à travers les gestionnaires de rôle. Cela afin de pouvoir définir une spécification respectant complètement l'architecture de déploiement du médium. Il s'agit donc de définir une spécification d'implémentation.

Les contrats de réalisation

Si nous reprenons la terminologie de UML Components [36], il s'agit de spécifier un *contrat de réalisation* du médium. UML Components définit en effet deux niveaux de contrats :

- Contrat d'usage : le contrat entre un composant et ses clients.
- Contrat de réalisation : le contrat entre la spécification d'un composant et son implémentation.

Le contrat d'usage spécifie le composant ou le médium du point de vue de son utilisation. Il correspond dans notre processus de raffinement à la spécification abstraite du médium. Le contrat de réalisation spécifie l'implémentation ou du moins exprime des contraintes que doit respecter l'implémentation. La spécification réalisée lors de cette deuxième étape n'est donc pas la définition exacte d'une implémentation et de son fonctionnement mais c'est une spécification qui impose des contraintes que doit respecter l'implémentation.

La deuxième étape du processus de raffinement consiste donc à définir un contrat de réalisation, ou plus exactement, un ou plusieurs contrats de réalisation. Pour une même spécification abstraite d'une interaction, il est utile de définir plusieurs spécifications d'implémentation. En effet, en fonction de choix de conception, d'implémentation ou de contraintes non fonctionnelles que l'on veut gérer, il est possible de définir plusieurs spécifications d'implémentation. Dans [32], nous détaillons le cas d'un médium de réservation (mais gérant plusieurs ensembles d'identificateurs et non pas un seul comme dans notre exemple). Nous expliquons que la montée en charge en terme de nombre de données à traiter et de composants effectuant des réservations impose de modifier l'implémentation du médium. Une implémentation conçue pour un petit nombre de composants et de données peut fonctionner avec l'ensemble des données sur un seul site. Lorsque le nombre de données et de composants augmentent, ce site unique devient un goulot d'étranglement et il faut passer à une autre implémentation. Ainsi, même si les services de réservation et d'annulation des réservations ne changent pas, il faut pouvoir disposer pour ce médium de plusieurs variantes d'implémentation en fonction du contexte d'utilisation.

Contrairement à la première étape du processus de raffinement, cette étape ne pourra pas être réalisée de manière automatique dans la majorité des cas, du fait notamment des choix de conception qui doivent être faits. C'est au concepteur que revient la responsabilité de déterminer un contrat de réalisation. C'est à lui que reviennent les décisions à prendre. À partir de la spécification obtenue à la première étape, le concepteur doit modifier et adapter toutes les éléments de spécification : le diagramme de collaboration, les contraintes et spécifications OCL et les diagrammes d'états. La classe représentant le médium disparaît, les diagrammes d'états qui lui était associés doivent maintenant être modifiés et associés à des gestionnaires. Toutes ces modifications ne sont pas automatisables à moins d'appliquer des algorithmes ou des politiques pré-définies.

Le respect du contrat du médium

Le contrat de réalisation doit respecter le contrat d'usage. La sémantique des services offerts par le médium ne doit pas varier lors de la définition d'un contrat de réalisation. Vérifier le respect de la sémantique n'est pas simple. Si lors de la première étape cela ne posait aucun problème, ici il n'y a pas de moyens pour le vérifier. Les modifications et

choix réalisés par le concepteur peuvent être de toute nature et il n'est pas possible de s'assurer du respect de la sémantique des services, du moins lorsque les modifications ne sont pas effectuées automatiquement ou suivant des règles bien définies. Dans certains cas, pour certains choix de conception, il faut même rajouter des contraintes qui n'existaient pas à l'étape précédente.

S'il n'est donc pas possible de vérifier formellement que le contrat est respecté, il est tout de même possible d'utiliser des techniques de tests et de simulation [50, 94]. Elles permettent de s'assurer que la spécification définie à un comportement similaire ou respectant celui de la spécification abstraite.

Les choix de conception pour notre exemple

Dans le cas du médium de réservation, le principal problème est de déterminer comment sont gérés les identificateurs de réservation. Cela revient à définir comment un ensemble de données est géré dans un contexte distribué (les gestionnaires de réserveur et de source qui accèdent aux données sont en effet distribués lors du déploiement du médium). Il existe de nombreux choix pour résoudre ce problème. Nous détaillons ici deux solutions particulières :

- Les identificateurs sont entièrement gérés de manière centralisée par un seul gestionnaire.
- L'ensemble des identificateurs est complètement distribué à l'ensemble des gestionnaires de réserveur, chacun d'entre eux possède un sous-ensemble de l'ensemble global des identificateurs.

Comme nous le verrons, les deux spécifications résultant de ces deux choix sont très différentes mais elles doivent toutes les deux respecter le contrat d'usage. Par exemple, si au moins un identificateur est disponible, il faut lors de l'appel du service `reserve` que ce dernier retourne un identificateur. La sémantique de ce service reste la même indépendamment de la gestion des identificateurs qui a été choisie.

3.3.2 Exemple d'application, premier choix de conception : gestion centralisée

Le premier choix que nous détaillons est de gérer l'ensemble des identificateurs de manière centralisée. Pour cela, nous décidons de créer un nouveau type de gestionnaire : le gestionnaire de réservation. C'est ce gestionnaire qui gère l'ensemble des identificateurs. Les autres gestionnaires se contentent alors de relayer les requêtes de leur rôle associé sur ce nouveau gestionnaire qui a la charge d'exécuter ces requêtes. Ce gestionnaire particulier n'est instantié qu'en un seul exemplaire et n'est associé à aucun composant.

Diagramme de collaboration

La nouvelle structure de la collaboration est représentée sur la figure 3.5 page suivante. Nous y retrouvons le nouveau gestionnaire : la classe `ReservationManager`. C'est lui qui gère entièrement les identificateurs (c'est pourquoi il implémente également les interfaces `ISourceServiceMedium` et `IReserverServiceMedium` car elles contiennent les signatures des opérations d'accès aux identificateurs). Les classes `SourceManager` et `ReserverManager` ne sont plus que des proxies des services qu'elles implémentent. Ces derniers sont réellement rendus par la classe `ReservationManager`. Les ensembles

`originalSet`, `available` et `reserved` sont identiques à ceux de la spécification de l'étape précédente. Ils représentent donc, respectivement, l'ensemble des identificateurs gérés par le médium, l'ensemble des identificateurs disponibles et l'ensemble des identificateurs réservés par un gestionnaire.

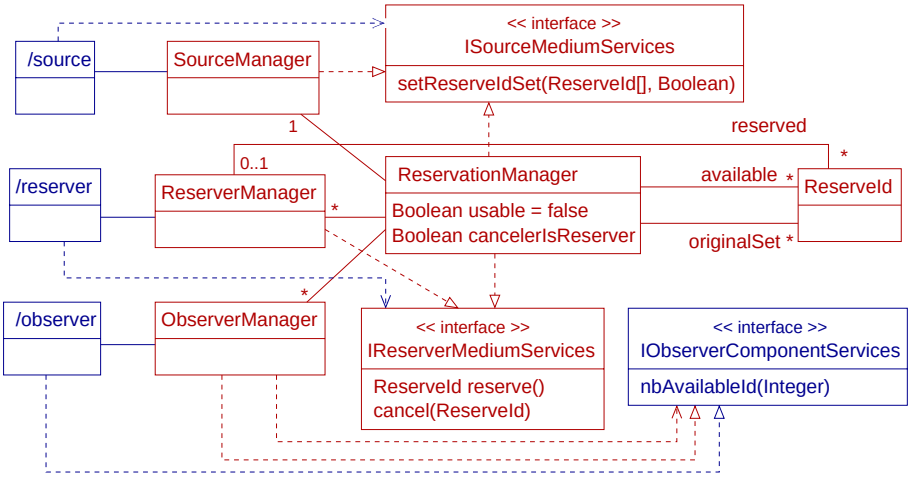

FIG. 3.5: Premier choix de conception : gestion centralisée

La vue dynamique de cette collaboration est représentée sur la figure 3.6. Les services appelés sur les gestionnaires par les rôles sont rappelés avec les mêmes paramètres sur la classe `ReservationManager`. Les gestionnaires ne font que faire suivre les demandes de service au gestionnaire de réservation. Les rôles observateurs sont notifiés du changement du nombre d'identificateurs disponibles par le gestionnaire de réservation via leur gestionnaire d'observateur associé.

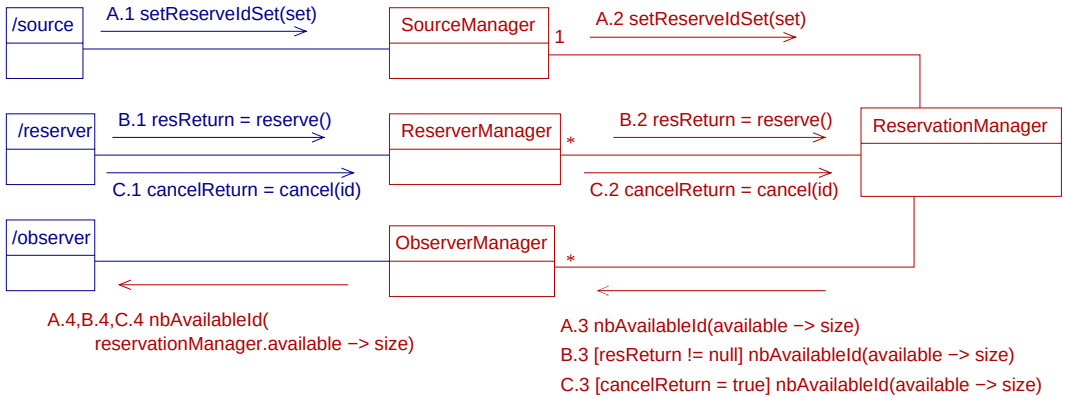

FIG. 3.6: Vue dynamique de la version centralisée

Spécifications OCL

Tout d'abord, nous pouvons nous intéresser aux spécifications des services de la classe `ReservationManager`. Elles sont en fait identiques à celles des services de la spécification

abstraite, au contexte d'expression près. Ces spécifications sont les suivantes :

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

Le service `setReserveIdSet` est maintenant défini dans le contexte de la classe `ReservationManager`. Ce service modifie la valeur de l'état d'utilisabilité du médium. Il faut tout de même noter qu'il y a une contrainte implicite pour que le médium soit utilisable : il faut que le gestionnaire de réservation soit présent. Sinon, le service `setReserveIdSet` ne peut bien sûr pas être appelé. Bien entendu, il serait possible voire intéressant de faire ressortir explicitement cette contrainte dans la définition de l'état d'utilisabilité du médium. Cette contrainte d'obligation de la présence de ce gestionnaire est spécifique à ce choix d'implémentation et à sa spécification. Ainsi, en fonction de la spécification d'implémentation, des contraintes supplémentaires pourront être ajoutées par rapport au contrat d'usage du médium.

Les spécifications des services `reserve` et `cancel` du gestionnaire de réservation sont décrites ci-dessous. Ici aussi, les spécifications sont identiques à celles faites à l'étape de spécification abstraite, au contexte d'expression près.

Voici la spécification du service `reserve` :

```
context ReservationManager::reserve() : ReserveId
pre:
    usable = true
post:
    if available -> isEmpty
    then result = null
    else
        originalSet -> includes(result)
        and available@pre -> includes(result)
        and available -> excludes(result)
        and caller.reserved -> includes(result)
    endif
```

Nous pouvons noter que nous avons à nouveau besoin dans cette spécification d'utiliser notre pseudo-attribut *caller*. En effet, le gestionnaire de réservation doit référencer l'ensemble des identificateurs réservés par le composant ayant demandé à réserver un nouvel identificateur. Cet ensemble est géré par son gestionnaire de réserveur qui a relayé cette requête sur le gestionnaire de réservation (et donc appelé le service `reserve` sur ce dernier).

Voici la spécification du service `cancel` :

```
context ReservationManager::cancel(ReserveId id) : Boolean
pre:
    usable = true
post:
    if ( originalSet -> excludes(id) or available@pre -> includes(id) or
        (cancelerIsReserver and caller.reserved@pre -> excludes(id)) )
    then result = false
    else
        available = available@pre -> including(id)
        and reserverManager -> forAll ( r | r.reserved -> excludes(id) )
        and result = true
    endif
```

Les spécifications OCL des services offerts aux composants par les gestionnaires de réserveur et de source sont inutiles. En effet, la vue dynamique de la collaboration exprime que les services appelés par les rôles sur les gestionnaires sont rappelés sur la classe `ReservationManager` avec les mêmes paramètres. Les valeurs retournées par les services de la classe `ReservationManager` sont renvoyées par les services de ces gestionnaires. À part cette redirection d'appel, les services ne font rien d'autre. Les gestionnaires de source et de réserveur, ainsi que les gestionnaires d'observateur ne sont donc que des proxies qui relayent les requêtes venant des composants ou du gestionnaire de réservation.

Diagrammes d'états

La spécification de l'opération `managerConnection` que nous vu dans l'étape d'introduction des gestionnaires change très peu : le contexte de définition est maintenant la classe `ReservationManager` et la méthode `nbAvailableId` doit être appelée sur le gestionnaire d'observateur et non pas directement sur le rôle. En effet, dans une implémentation de médium, le gestionnaire de réservation n'a pas de référence directe sur un composant observateur, il doit passer par son gestionnaire associé. Voici cette spécification :

```
context ReservationManager::newConnection(RoleManager manager)
post:
    (oclInState(Running::Usable) and manager.isKindOf(ObserverManager))
    implies manager.oclCallOperation(nbAvailableId, available -> size)
```

Nous devons maintenant définir la méthode `nbAvailableId` dans le contexte d'un gestionnaire d'observateur. Cette méthode appelle la méthode du même nom avec les mêmes paramètres sur le rôle associé à ce gestionnaire :

```
context ObserverManager::nbAvailableId(int available)
post: component.oclCallOperation(nbAvailableId, available)
```

La spécification de cette méthode peut sembler redondante par rapport au diagramme de collaboration, où il est exprimé que la méthode `nbAvailableId` est aussi appelée par un gestionnaire d'observateur sur son rôle. En fait, le diagramme de collaboration ne prend pas en compte le cas de la connexion d'un nouveau gestionnaire d'observateur. C'est ce cas que nous gérons ici, même si a priori la spécification de la méthode `nbAvailableId`

dans le contexte du gestionnaire d'observateur est valide dans tous les cas. Notons tout de même que la sémantique ne change pas entre cette spécification OCL et le diagramme de collaboration : la méthode est toujours appelée avec en paramètre le nombre d'identificateurs disponibles. Il n'y a donc pas de contradiction ou d'ambiguïté dans la sémantique de cette méthode, mais juste une redondance dans certains cas.

3.3.3 Exemple d'application, deuxième choix de conception : gestion distribuée

Le deuxième choix de conception que nous faisons est de gérer l'ensemble des identificateurs de manière totalement distribuée. Chaque gestionnaire de réserveur possède une partie de l'ensemble des identificateurs disponibles. Le but est, pour un gestionnaire de réserveur, d'effectuer les réservations demandées par son rôle dans cet ensemble local. La contrainte que nous ajoutons est que l'ensemble des identificateurs disponibles doit être équitablement partagé entre tous les gestionnaires de réserveur.

Le problème principal de cette conception d'implémentation concerne la distribution de l'ensemble. À l'initialisation, il peut être équitablement distribué entre tous les gestionnaires de réserveur présents. Se posent alors trois problèmes :

- Comment s'assurer que lorsqu'au moins un identificateur est disponible chez un des gestionnaires de réserveur, l'appel à un service `reserve` retourne un identificateur même si l'ensemble des identificateurs du gestionnaire de réserveur sur qui cet appel a été fait est vide ?
- Que faire lorsqu'un nouveau gestionnaire de réserveur est lancé ? Doit-il également obtenir une partie de l'ensemble des identificateurs qui a déjà été entièrement distribué ?
- Que faire lorsqu'un gestionnaire de réserveur quitte le médium ? Il possède en effet une partie de l'ensemble des identificateurs qui doivent être redistribués entre tous les gestionnaires de réserveur restant.

Les réponses et les solutions à ces questions sont loin d'être triviales. Ce qu'il faut en fait, c'est définir ou utiliser un algorithme de partage et de redistribution de ressources distribuées. Si cela est indispensable pour réaliser l'implémentation du médium, cela ne l'est pas forcément au niveau de spécification auquel nous nous plaçons. La spécification que nous devons faire doit tenir compte à la fois de l'architecture de déploiement des gestionnaires et du fait que les identificateurs soient distribués. À partir de cela, il faut redéfinir les spécifications des services offerts par le médium pour qu'elles continuent à respecter la sémantique définie lors de la spécification abstraite, mais sans pour autant décrire comment est gérée en pratique cette distribution.

Il est bien sûr possible de raffiner la spécification incluant ce choix de conception distribuée pour faire apparaître le choix exact et le fonctionnement de l'algorithme de gestion de distribution des données. Mais dans notre exemple, nous ne descendons pas jusqu'à ce niveau de précision.

Diagramme de collaboration

Le diagramme de collaboration de cette conception distribuée est représenté sur la figure 3.7 page suivante. Le lien `localAvailable` représente l'ensemble des identificateurs disponibles alloués à un gestionnaire de réserveur. L'ensemble des identificateurs gérés

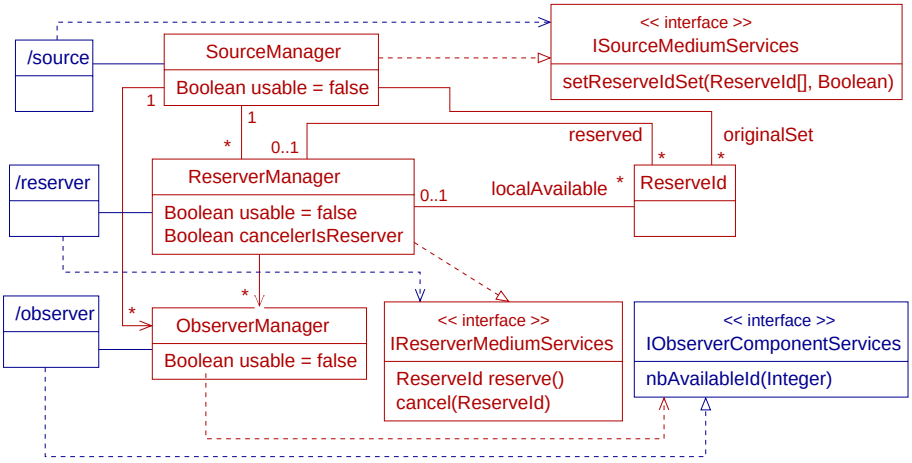

FIG. 3.7: Second choix de conception : gestion distribuée

par le médium est quant à lui représenté par l'association `originalSet`. Comme dans la spécification précédente, les identificateurs actuellement réservés par un composant réserveur sont représentés par l'ensemble `reserved`, associé à chaque gestionnaire de réserveur. Chaque gestionnaire possède également une copie locale de l'attribut `usable` qui lui permet de connaître l'état d'utilisabilité du médium. Par défaut, le médium n'est pas utilisable.

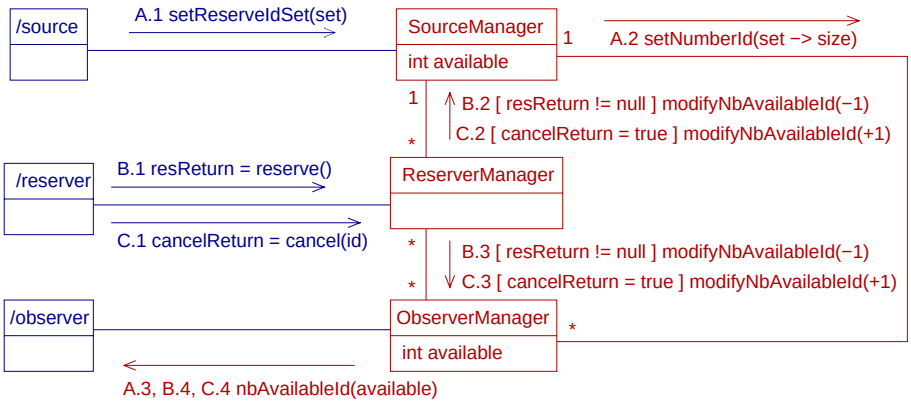

FIG. 3.8: Vue dynamique de la version distribuée

La vue dynamique du diagramme de collaboration correspond à la figure 3.8. Dans les spécifications précédentes, la vue dynamique nous avait principalement permis de spécifier quand notifier les composants observateurs du changement du nombre d'identificateurs disponibles. Ici aussi nous allons spécifier cela mais d'une manière différente, car contrairement aux autres cas, il n'y a plus un seul élément qui centralise tous les identificateurs et qui connaît donc le nombre exact d'identificateurs disponibles à tout moment. Le principe que nous allons utiliser est le suivant :

– Quand l'ensemble des identificateurs est initialisé lors de l'appel de `setReserveIdSet` sur le gestionnaire de source (message A.1), tous les gestionnaires d'observateur

présents à ce moment sont informés du nombre d'identificateurs via la méthode `setNumberId` (message A.2). Les gestionnaires d'observateurs appellent alors le service `nbAvailableId` sur leur rôle observateur associé (message A.3) avec la valeur transmise par le gestionnaire de source (qui est stockée au passage dans l'attribut `available`).

– Lorsqu'une réservation ou une annulation de réservation est effectuée (via l'appel des services `reserve` (message B.1) ou `cancel` (message C.1) quand ils aboutissent), les gestionnaires d'observateur sont informés de la variation du nombre d'identificateurs disponibles via la méthode `modifyNbAvailableId`. Une réservation d'identificateur correspond à une modification du nombre d'identificateurs de moins un (message B.3) et une annulation de réservation de plus un (message C.3). Avec cette information, un gestionnaire d'observateur peut informer son rôle associé du nouveau nombre d'identificateurs disponibles (message B.4 ou C.4). Cette variation est également envoyée au gestionnaire de source pour que lui aussi puisse connaître le nombre d'identificateurs disponibles (message B.2 ou C.2). Il doit en effet envoyer cette information à tous les gestionnaires d'observateurs qui se connectent au médium après l'initialisation de l'ensemble (nous verrons ultérieurement comment cela est spécifié).

Nous allons détailler les spécifications OCL des méthodes spécifiques à cette vue dynamique. La méthode `setNumberId` est utilisée à l'initialisation pour indiquer le nombre d'éléments dans l'ensemble des identificateurs. Localement, un gestionnaire d'observateur stocke le nombre d'identificateurs disponibles dans un attribut nommé `available`. Il s'agit donc ici d'initialiser cet attribut :

context ObserverManager::setNumberId(int nb)
post:
available = nb **and**
observer.*oclCallOperation(nbAvailableId, available)*

La dernière contrainte exprime que le service `nbAvailableId` est appelée sur le composant associé au gestionnaire. Si cela est redondant par rapport à la vue dynamique du diagramme de collaboration, ça ne l'est pas lorsqu'un nouveau gestionnaire d'observateur se connecte (comme nous le verrons dans la partie traitant des diagrammes d'états pour cette version du médium).

La méthode `modifyNbAvailableId` sert à prévenir un gestionnaire (de source ou d'observateur) que le nombre d'identificateurs a changé. Le paramètre indique la valeur de cette variation (qui peut être positive ou négative). L'attribut `available` doit donc être mis à jour en fonction de cette variation.

Voici la spécification de cette méthode dans le contexte du gestionnaire de réservation :

context ObserverManager::modifyNbAvailableId(int diff)
post:
available = available@pre + diff **and**
observer.*oclCallOperation(nbAvailableId, available)*

La dernière contrainte exprime que le service `nbAvailableId` est appelé sur le composant associé au gestionnaire. Si cela est redondant par rapport à la vue dynamique du diagramme de collaboration, ça ne l'est pas lorsqu'un nouveau gestionnaire d'observateur

se connecte (comme nous le verrons également dans la partie traitant des diagrammes d'états pour cette version du médium).

La spécification de la méthode `modifyNbAvailableId` dans le contexte du gestionnaire de source est la suivante :

```
context SourceManager::modifyNbAvailableId(int diff)
post: available = available@pre + diff
```

Là aussi, le gestionnaire de source utilise un attribut nommé `available` pour mémoriser le nombre d'identificateurs disponibles.

La manière d'informer les composants observateurs du nombre d'identificateurs disponibles dans le médium est donc très différente de celle de la spécification abstraite ou de la version centralisée. Elle met donc en avant la difficulté que l'on peut rencontrer pour s'assurer que le contrat est respecté lors d'un choix de conception. Pour vérifier que la sémantique de la notification aux observateurs est respectée, il faudrait montrer que les deux spécifications (celle de la version distribuée et celle de la spécification abstraite) sont équivalentes.

Spécifications OCL

Avant de définir les spécifications OCL des services offerts, nous allons nous attacher à définir un invariant qui définit que l'ensemble des identificateurs disponibles est bien partagé par tous les gestionnaires de réserveur présents dans le médium. Voici cet invariant :

```
context SourceManager inv:
usable implies (
    reserverManager -> forAll(r1, r2 | r1 <> r2 implies
        r1.localAvailable.intersection(r2.localAvailable) -> isEmpty and
        abs( (r1.localAvailable -> size) - (r2.localAvailable -> size) ) <= 1 ) and
    reserverManager -> forAll(
        originalSet -> includesAll(localAvailable) and
        originalSet -> includesAll(reserved) ) )
```

Cet invariant exprime la distribution de l'ensemble des identificateurs disponibles. Chaque gestionnaire de réserveur possède un ensemble local d'identificateurs disponibles (référencé par `localAvailable`). Lorsque le médium est utilisable, les contraintes sur ces ensembles locaux sont les suivantes :

- Deux ensembles locaux de deux gestionnaires différents ne comportent aucun élément en commun.
- Le nombre d'identificateurs dans deux ensembles locaux ne diffère au maximum que d'un seul identificateur.
- Un ensemble local d'identificateurs disponibles est un sous-ensemble de l'ensemble original.
- L'ensemble des identificateurs réservés chez un gestionnaire est également un sous-ensemble de l'ensemble original.

Cet invariant exprime donc que l'ensemble des identificateurs disponibles est équitablement réparti entre tous les gestionnaires présents. Cela implique notamment que

lorsqu'un nouveau gestionnaire se connecte, les identificateurs doivent être redistribués pour que tous les gestionnaires – y compris le nouveau – possèdent une partie de l'ensemble. De même, lorsqu'un gestionnaire se déconnecte, son ensemble d'identificateurs disponibles est partagé entre tous les gestionnaires de réserveur restants.

La spécification de cet invariant montre bien le niveau de spécification auquel nous nous plaçons. Cet invariant exprime que les identificateurs doivent être distribués mais ne décrit pas comment la mise en œuvre exacte de cette distribution doit être faite. Il impose des contraintes sur l'implémentation mais sans préciser quels sont les algorithmes et techniques utilisés pour assurer cet invariant en pratique dans une implémentation.

Opération SourceManager::setReserveIdSet(Set, Boolean). Le but du service `setReserveIdSet` est d'initialiser le médium en positionnant l'ensemble des identificateurs de réservation. Lors de cette initialisation, cet ensemble doit être partagé entre tous les gestionnaires de réserveur présents. Voici la spécification de ce service :

```
context SourceManager::setReserveIdSet(Set idSet, Boolean cancel)
pre:
    usable = false                                    -- le médium n'est pas utilisable
    and reserverManager -> forAll( r | r.usable = false)
    and observerManager -> forAll( o | o.usable = false)
post:
    originalSet = idSet
    and available = idSet -> size
    and usable = true                                  -- le médium devient utilisable
    and observerManager -> forAll( o | o.usable = true)
    and reserverManager -> forAll( r |
        r.usable = true
        and r.cancelerIsReserver = cancel
        and r.reserved -> isEmpty )
    and if reserverManager -> isEmpty
        then localAvailable = idSet          -- pas de réserveur : le source garde tous les id.
        else reserverManager -> collect (localAvailable) = idSet
                                  -- il y a des réserveurs, ils se partagent les identificateurs
        endif
```

La précondition impose que le médium soit inutilisable pour tous les gestionnaires. Dans la version centralisée, l'utilisabilité est gérée de manière centralisée (dans le gestionnaire de réservation). Ici, dans un contexte distribué, chaque gestionnaire possède une copie locale de cet attribut. Chez tous les gestionnaires, sa valeur doit donc être égale à faux avant l'appel du service. Elle passe à vrai à la fin de l'appel, comme spécifié dans la postcondition.

La valeur de l'attribut `cancelerIsReserver` passé en paramètre du service doit être diffusée à tous les gestionnaires de réserveur. Chez tous ces gestionnaires, aucun identificateur ne doit être réservé (l'ensemble `reserved` est vide).

La gestion de l'ensemble des identificateurs passé en paramètre du service se fait de la manière suivante :

– L'ensemble `originalSet` est une copie de cet ensemble.

- S'il y a des gestionnaires de réserveur présents, alors l'ensemble est partagé par tous ces gestionnaires. Cela s'exprime par le fait que l'union de tous les ensembles locaux d'identificateurs disponibles doit être égal à l'ensemble passé en paramètre.
- Si aucun gestionnaire de réserveur n'est présent lors de l'initialisation, alors le gestionnaire de source conserve localement l'ensemble dans son ensemble `localAvailable`. À la connexion du premier gestionnaire de réserveur, le gestionnaire de source lui passera cet ensemble (nous verrons cela dans la partie concernant les diagrammes d'états).

Opération ReserverManager::reserve() : ReserveId. Pour chaque gestionnaire, la réservation d'un identificateur se fait si possible dans l'ensemble local des identificateurs disponibles. Si cet ensemble est vide, alors l'identificateur doit être retiré de l'ensemble d'un autre gestionnaire. Ici, nous ne détaillons pas comment cette dernière opération doit être réalisée. Nous exprimons uniquement une contrainte qui dit que l'identificateur retourné appartient à l'ensemble local d'un autre gestionnaire. Voici la spécification de l'opération `reserve` :

```
context ReserverManager::reserve() : ReserveId
pre:
    usable = true
post:
    if localAvailable -> notEmpty
    then                          -- l'ensemble local n'est pas vide : la réservation y est faite
        localAvailable@pre -> includes(result) and
        localAvailable -> excludes (result) and
        reserved -> includes (result)
    else                          -- ensemble local vide, on cherche chez les autres gestionnaires
        let globalAvailable =
                          sourceManager.reserverManager -> collect(localAvailable@pre) in
        if globalAvailable -> isEmpty          -- aucun id disponible dans tout le médium
        then result = null
        else                      -- l'id retourné est récupéré chez un des autres gestionnaires
            globalAvailable -> includes (result) and
            reserverManager -> forAll ( localAvailable -> excludes (result) ) and
            reserved -> includes (result)
        endif
    endif
```

Dans la postcondition, le premier test vérifie si l'ensemble local est vide ou pas. Si il ne l'est pas, alors la réservation se fait dans cet ensemble. Cela est spécifié comme pour la version centralisée et la spécification abstraite : l'identificateur retourné appartient à l'ensemble des identificateurs disponibles avant l'appel du service mais plus après. La réservation de cet identificateur est mémorisée en l'ajoutant dans l'ensemble `reserved`.

Si cet ensemble local est vide, alors la réservation doit se faire dans l'ensemble d'un autre gestionnaire. Pour spécifier cela, un pseudo ensemble nommé `globalAvailable` contient l'ensemble des identificateurs disponibles chez tous les gestionnaires de réserveur avant l'appel de l'opération. Si cet ensemble est vide, alors il n'y a aucun identifica-

teur disponible dans le médium et donc le service doit retourner la valeur `null`. Sinon, l'identificateur retourné appartient à cet ensemble. La spécification précise aussi que plus aucun gestionnaire ne doit posséder cet identificateur dans son ensemble local (puisque cet identificateur a été réservé).

La spécification de ce service montre bien là aussi le niveau de spécification auquel nous nous plaçons. Comme nous l'avons dit, le but est de définir des contraintes que l'implémentation doit respecter et non pas décrire comment l'implémentation doit être réalisée. Ici, nous précisons par exemple que si un ensemble local est vide, alors la réservation doit se faire dans l'ensemble d'un autre gestionnaire. C'est une contrainte sur l'implémentation. Mais nous ne précisons pas comment l'implémentation doit être réalisée pour satisfaire cette contrainte. Pour cela, il faudrait définir un algorithme qui spécifierait précisément comment un gestionnaire interagirait avec les autres gestionnaires pour leur demander s'ils possèdent des identificateurs disponibles et comment le transfert d'un identificateur entre deux gestionnaires se ferait.

Opération ReserverManager::cancel(ReserveId) : Boolean. L'annulation de la réservation d'un identificateur doit suivre les mêmes contraintes que pour la spécification abstraite. Pour cela, il faut notamment vérifier que l'identificateur dont la réservation est à annuler n'est pas disponible chez aucun des gestionnaires de réserveur. Ainsi, la validité de l'annulation doit être vérifieé à partir d'informations non locales pour le gestionnaire qui doit effectuer la réservation. La spécification du service `cancel` est la suivante :

```
context ReserverManager::cancel(ReserveId id) : Boolean
pre: usable = true
post:
    let globalAvailable = sourceManager.reserverManager -> collect(localAvailable@pre) in
    if ( originalSet -> excludes(id) or globalAvailable -> includes(id) or
                        -- si id n'appartient pas à l'ensemble de départ ou n'est pas réservé
        (cancelerIsReserver and reserved@pre -> excludes(id)) )
                            -- ou si cancelerIsReserver=vrai et que le composant appelant
                                                    -- n'est pas celui qui a réservé id
    then result = false                             -- alors l'annulation n'est pas valide
    else                                               -- sinon l'annulation est valide
        localAvailable = localAvailable@pre -> including(id) -- id est à nouveau réservable
        and sourceManager.reserverManager -> forAll( r | r.reserved -> excludes(id))
                                                    -- aucun réserveur ne réserve plus id
        and result = true
    endif
```

Cette spécification est dans le principe identique à celle de la spécification abstraite du médium. La différence principale est que l'ensemble des identificateurs disponibles est distribué. Pour s'assurer que l'identificateur n'est disponible chez aucun gestionnaire, nous utilisons un pseudo ensemble (nommé `globalAvailable`) qui est l'union de tous les ensembles d'identificateurs disponibles de tous les gestionnaires. La vérification se fait donc dans cet ensemble. Si l'annulation est validée, l'identificateur doit être rajouté dans

l'ensemble local du gestionnaire[1].

Diagrammes d'états

Un des problèmes que nous avons à résoudre est le même que dans la version centralisée : informer les gestionnaires d'observateur du nombre d'identificateurs disponibles dans l'ensemble dès qu'ils se connectent au médium, si les services de celui-ci sont utilisables. Nous allons pour cela utiliser le diagramme d'états générique d'un gestionnaire, comme dans le cas de la spécification de la version centralisée, en spécifiant l'opération `managerConnection` dans le contexte du gestionnaire de source. De plus, dans le cas de la connexion d'un gestionnaire de réserveur alors que le médium est utilisable, il faut l'informer de la valeur de l'attribut `cancelerIsReserver`. Ensuite, dans tous les cas, quelque soit l'état d'utilisabilité du médium, il faut informer tout nouveau gestionnaire de la valeur de cet état (qui est connu par le gestionnaire de source car c'est lui qui le modifie). Enfin, il faut aussi gérer la connexion du premier gestionnaire de réserveur pour lui attribuer l'ensemble des identificateurs disponibles. Tout cela est accompli par le gestionnaire de source comme le montre la spécification suivante :

```
context SourceManager::managerConnection(Manager manager)
post:
    (manager.oclIsKindOf(ObserverManager) and oclInState(Running::Usable))
        implies manager.oclCallOperation(setNumberId, available)
    and (manager.oclIsKindOf(ReserverManager) and oclInState(Running::Usable))
        implies manager.cancelerIsReserver = cancelerIsReserver
    and manager.usable = usable
                                        -- il faut gérer la première connexion d'un réserveur
    and (manager.oclIsKindOf(ReserverManager) and oclInState(Running::Usable)
            and (reserverManager -> size = 1) ) implies (
        manager.localAvailable = localAvailable@pre and
        localAvailable -> isEmpty )
```

La première contrainte précise que dans le cas où le gestionnaire qui se connecte est un gestionnaire d'observateur et que le médium est utilisable, alors ce gestionnaire doit être informé du nombre d'identificateurs disponibles. La seconde contrainte précise que dans le cas où le gestionnaire qui se connecte est un gestionnaire de réserveur et que le médium est utilisable, alors la valeur locale de l'attribut `cancelerIsReserver` chez ce gestionnaire doit être égale à la valeur de ce même attribut chez le gestionnaire de source. La troisième contrainte précise qu'à chaque nouvelle connexion de gestionnaire, la valeur de l'état d'utilisabilité chez ce gestionnaire doit être égale à la valeur de ce même attribut chez le gestionnaire de source. Enfin, la dernière contrainte gère la connexion d'un gestionnaire de réserveur lorsque le médium est utilisable et que ce gestionnaire est le seul gestionnaire de réserveur dans le médium. Dans ce cas, tous les identificateurs disponibles doivent lui être associés. Lorsqu'aucun gestionnaire de réserveur n'est présent, c'est le gestionnaire de source qui possède l'ensemble des identificateurs disponibles à la réservation[2]. Ensuite,

[1] L'invariant sur la distribution des identificateurs du médium impose ensuite, que si cet ensemble contient trop d'éléments par rapport aux autres ensembles locaux, que des éléments de l'ensemble local soient déplacés vers d'autres ensembles.

[2] Même si aucun gestionnaire de réserveur n'est présent, l'ensemble des identificateurs disponibles n'est

quand d'autres gestionnaires de réserveur se connecteront, l'invariant que nous avons défini précédemment impose que les identificateurs disponibles soient partagés entre tous les gestionnaires de réserveur.

Il faut également gérer la déconnexion des gestionnaires de réserveur : lorsque le dernier d'entre eux se déconnecte, il faut que l'ensemble de ses identificateurs disponibles (c'est-à-dire également l'ensemble des identificateurs disponibles dans tout le médium) soit attribué au gestionnaire de source, en attendant que de nouveaux gestionnaires de réserveur se connectent. Cela se fait en spécifiant dans le contexte du gestionnaire de source, le sémantique de l'opération `managerDisconnection` comme suit :

```
context SourceManager::managerDisconnection(Manager manager)
post:
    (manager.oclIsKindOf(ReserverManager) and oclInState(Running::Usable))
            and (reserverManager -> size = 0) ) implies (
        localAvailable = manager.localAvailable@pre and
        manager.localAvailable -> isEmpty )
```

Si le médium est dans un état utilisable et que le gestionnaire de réserveur qui se déconnecte est le dernier (c'est-à-dire quand l'ensemble des gestionnaires de ce type est vide à la fin de l'exécution de l'opération `managerDisconnection`), alors le gestionnaire de source doit récupérer, dans son ensemble `localAvailable`, l'ensemble des identificateurs disponibles de ce gestionnaire (et donc du médium).

3.4 Troisième étape : choix de déploiement

La dernière étape du processus, avant de s'attacher à la phase d'implémentation, consiste à définir l'architecture de déploiement ou plus exactement des contraintes sur le déploiement des gestionnaires. Nous savons qu'un gestionnaire de rôle est déployé sur le même site que le composant lui étant associé mais nous avons vu que certains gestionnaires ne sont pas forcément associé à un composant. Pour ces gestionnaires, il faut décider de leur déploiement. Par exemple, s'ils sont déployés chacun seul sur un site ou regroupés à plusieurs sur un site donné. Il est également possible de définir toute contrainte sur les déploiements de certains gestionnaires.

Pour cela, il s'agit de définir un diagramme de déploiement UML pour chaque type de déploiement défini, et cela à partir d'une spécification de conception donnée. Il est possible de définir plusieurs déploiements pour une même spécification d'implémentation.

Respect du contrat du médium

Comme lors de toutes les étapes, il faut s'assurer que le contrat du médium est respecté. En fait, dans cette étape nous ne décrivons que des choix de déploiement de spécifications d'implémentation. Ce choix n'a aucune influence sur la sémantique des services du médium. Du moins si l'on se place dans un contexte où les communications

pas forcément égal à l'ensemble des identificateurs originaux. En effet, il est possible que des composants réserveurs se soient connectés, aient fait des réservation sans les annuler et se soient ensuite déconnectés du médium. Dans ce cas, les identificateurs réservés le restent et donc l'ensemble des identificateurs disponibles n'est pas égal à l'ensemble original.

entre les sites sont parfaites (elles aboutissent à chaque fois et se réalisent en un temps nul). Dans la pratique, il n'est pas toujours possible de se baser sur ce genre d'hypothèses. Lors de l'implémentation il faut donc en tenir compte. Cela peut aboutir à l'ajout de contraintes sur le contrat du médium où à l'impossibilité de vérifier certaines contraintes.

Lors de cette étape de choix de déploiement, la sémantique du médium est donc la même que celle définie lors de l'étape précédente de choix de conception et d'implémentation.

3.4.1 Déploiement pour la gestion centralisée des identificateurs

Comme nous l'avons vu dans la spécification de conception de la version centralisée, il y a un gestionnaire non associé à un composant qui est utilisé : le gestionnaire de réservation. Le diagramme de déploiement doit donc faire apparaître ce gestionnaire et préciser le déploiement choisi.

Plusieurs variantes sont possibles : déployer ce gestionnaire seul sur un site à part où le déployer sur le même site qu'un autre gestionnaire. À chacun de ces choix correspond un diagramme de déploiement décrivant le déploiement choisi.

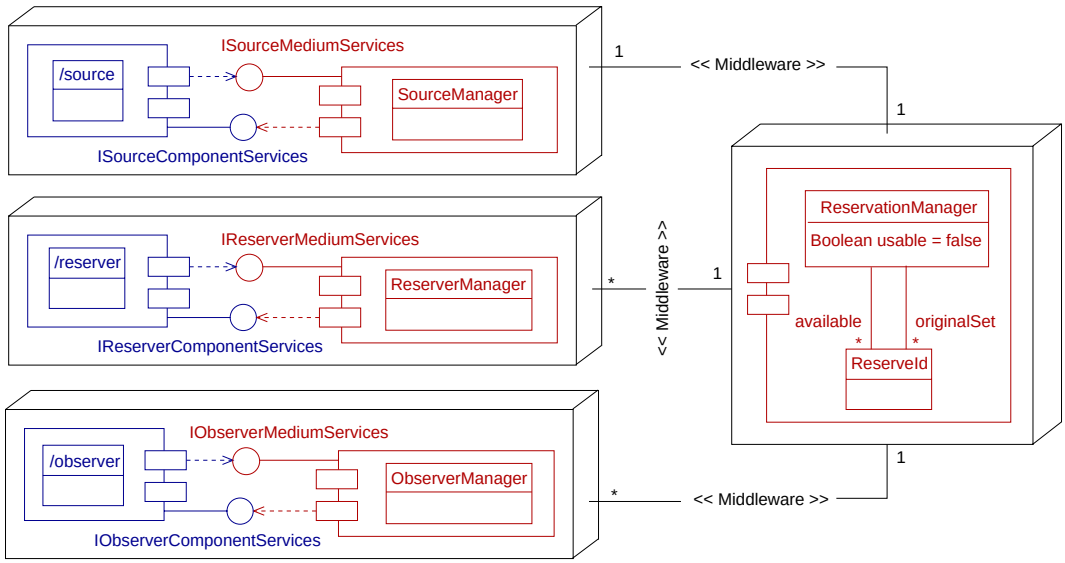

FIG. 3.9: Déploiement de la version centralisée, avec le gestionnaire de réservation autonome

La figure 3.9 représente le diagramme de déploiement dans le cas où le gestionnaire de réservation est déployé seul sur un site distinct. Sur cette figure, on distingue les quatre types de sites qui peuvent être instantiés. Nous pouvons déterminer également les contraintes sur le nombre de chaque site qui peut être déployé en notant les cardinalités des liaisons entre ces sites. Cela donne en détail :

- Un composant source et son gestionnaire de source associé, qui ne peuvent être déployés qu'une unique fois.
- Un composant réserveur et son gestionnaire de réserveur associé, qui peuvent être déployés de multiples fois.

– Un composant observateur et son gestionnaire d'observateur associé, qui peuvent être déployés de multiples fois.
– Un gestionnaire de réservation seul, qui ne peut être déployé qu'une unique fois.

Un composant ou un gestionnaire est représenté par un composant graphique UML. Sur chaque site où se trouvent un composant et son gestionnaire associé, les liens entre ces deux composants UML se font par l'intermédiaire de dépendances sur les interfaces de services offerts et requis par un gestionnaire[3].

En ce qui concerne le gestionnaire de réservation, nous pouvons constater que c'est lui qui gère l'ensemble des identificateurs. Le diagramme de déploiement fait donc apparaître de manière claire le choix de conception qui avait été fait, à savoir la gestion centralisée des identificateurs.

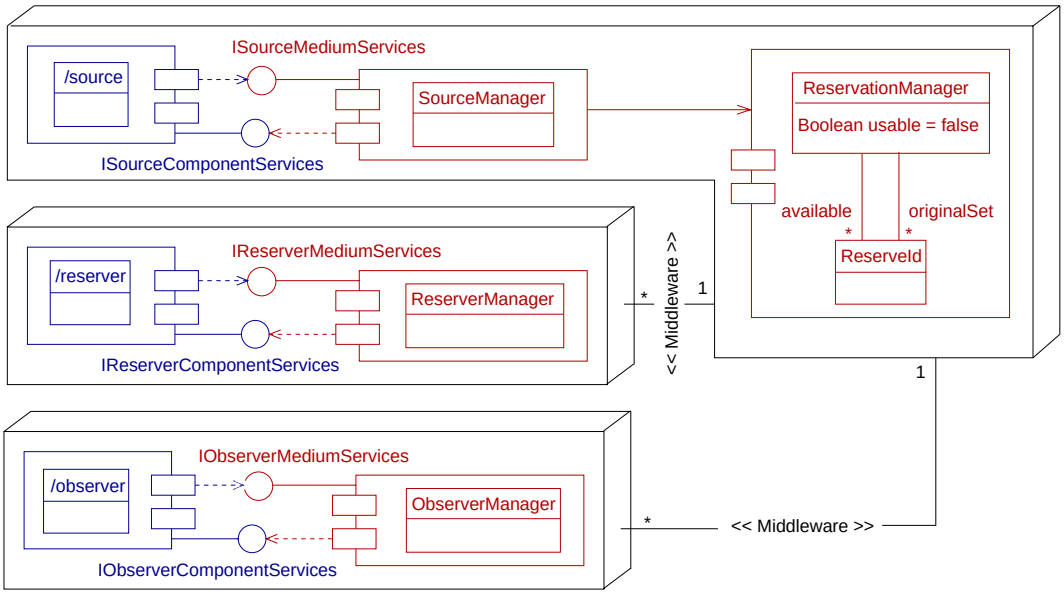

Fig. 3.10: Déploiement de la version centralisée, avec le gestionnaire de réservation associé au gestionnaire de source

La figure 3.10 représente le diagramme de déploiement dans le cas où le gestionnaire de réservation est déployé sur le même site que le gestionnaire de source. Il s'agit d'un deuxième choix de déploiement pour la spécification d'implémentation de la gestion centralisée des identificateurs.

3.4.2 Déploiement pour la gestion distribuée des identificateurs

Pour ce choix de conception, nous ne définissons qu'un seul choix de déploiement : chaque composant avec son gestionnaire associé sur le même site. La figure 3.11 page suivante montre le diagramme de déploiement correspondant à ce choix. Il apparaît clairement sur ce diagramme que les identificateurs sont entièrement distribués et leur ensemble partagé, chacun des gestionnaires de réserveur en possédant une partie.

[3]Sur le diagramme de déploiement, toutes les interfaces de services sont représentées, y compris celles qui sont vides.

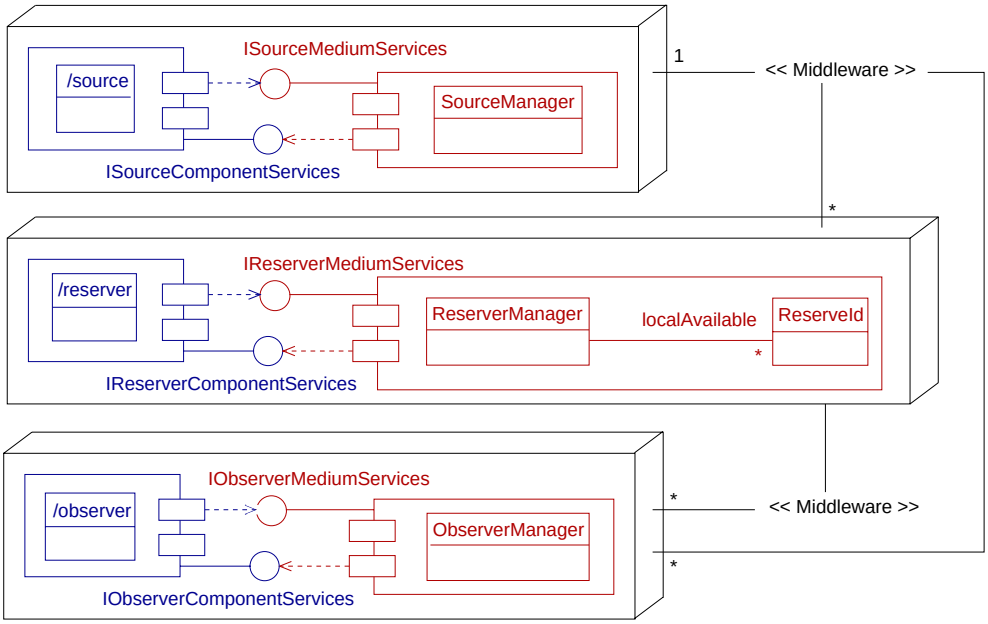

FIG. 3.11: Déploiement de la version distribuée

Au niveau des cardinalités de connexion de sites physiques, nous pouvons noter que le site contenant le composant source avec son gestionnaire associé n'est déployé qu'une seule fois alors que les deux autres types de site peuvent être déployés un nombre quelconque de fois.

3.5 Résumé du processus de raffinement

Avec ce processus de raffinement, nous disposons d'un outil pour passer de la spécification abstraite d'un médium à une spécification de plus bas niveau tenant compte de l'architecture de déploiement des médiums et de choix de conception ou d'implémentation. Ainsi, il est assez aisé de manipuler une même abstraction de communication à tous les niveaux de spécification et de pouvoir définir des variantes de conception ou d'implémentation pour une même abstraction. Le contrat du médium qui correspond à la spécification abstraite, doit être respecté à tous les niveaux, autant que cela soit possible.

Le processus de raffinement est constitué de différentes étapes qui transforment la spécification de l'étape précédente en une nouvelle spécification. Voici ces différentes étapes et leurs transformations, en les replaçant dans le contexte du MDA que nous avons présenté en début de chapitre :

- La donnée en entrée du processus de raffinement est la spécification abstraite du médium, c'est-à-dire la spécification du contrat d'usage du médium, indépendamment de tout choix ou contrainte de conception ou d'implémentation. Cette spécification correspond au niveau PIM comme il est défini dans le MDA.
- La première étape du processus consiste à faire apparaître, sur le diagramme de collaboration, la classe représentant le médium dans son ensemble comme l'agrégation de gestionnaires de rôle. Un gestionnaire est associé à chaque rôle. Cette

étape est entièrement automatisable dans un outil. La spécification correspondant à cette étape introduit l'architecture de déploiement des médiums, c'est-à-dire une information dépendante de la plate-forme. Dans cette deuxième étape, nous avons donc affaire à une spécification de niveau PSM[4]. La transformation qui a été effectuée est de type PIM vers PSM. Le contrat du médium est totalement respecté lors de l'application de cette étape.

- La deuxième étape consiste à faire disparaître complètement cette classe représentant le médium dans son ensemble. Le problème est alors de définir comment les services offerts par le médium sont spécifiés dans ce nouveau contexte et comment gérer les différentes données manipulées par le médium uniquement à partir des gestionnaires. Il s'agit donc de définir un contrat de réalisation qui prend en compte l'architecture de déploiement d'un médium. À ce stade, il faut également faire des choix de conception ou d'implémentation. Il est possible de définir plusieurs spécifications de conception en fonction de différents choix ou contraintes à gérer. Contrairement à la précédente étape, celle-ci est difficilement automatisable car elle fait apparaître des choix qui a priori sont à faire par le concepteur du médium et qui peuvent amener à des modifications complexes des différentes spécifications tout en étant difficilement dérivables de la spécification de l'étape précédente. La spécification à cette étape est elle aussi de niveau PSM et la transformation, qui correspond en fait à un raffinement, est de type PSM vers PSM. Le respect du contrat du médium est difficile à vérifier. Dans la plupart des cas, cela ne peut pas être fait de manière formelle. Il est néanmoins possible d'utiliser des techniques de tests et de simulation.
- Dans la troisième et dernière étape, pour chacune des variantes de spécification de conception obtenues à l'étape précédente, il s'agit de définir un ou plusieurs diagrammes de déploiement pour décrire comment les différents gestionnaires et les composants sont distribués et regroupés lors du déploiement d'un médium. Là encore, nous avons une spécification de niveau PSM et la transformation permettant d'arriver à cette étape est de type PSM vers PSM. Cette étape ne modifie pas la spécification des services et du médium. La sémantique du médium est identique à celle de l'étape précédente.

La figure 3.12 page ci-contre répertorie l'ensemble des spécifications que nous avons définies lors du raffinement pour le médium de réservation. L'étape 0 correspond à la spécification abstraite, au contrat d'usage du médium. Cette première spécification est raffinée par l'ajout des gestionnaires associés aux rôles et dont l'agrégation correspond à la classe représentant le médium dans son ensemble. Le résultat de cette opération donne l'étape 1. À partir de cette spécification, nous avons défini dans l'étape 2, deux spécifications d'implémentation ou deux contrats de réalisation : la première où l'ensemble des identificateurs est géré de manière centralisée par un gestionnaire particulier et la seconde où cet ensemble est distribué entre tous les gestionnaires de réserveur. Finalement, lors de la dernière étape (étape 3) nous avons défini trois types de déploiement : deux pour la version centralisée (avec le nouveau gestionnaire seul sur un site ou sur le même site que le gestionnaire de source) et un pour la version distribuée. À partir d'une unique spécification abstraite d'une abstraction d'interaction, nous avons donc défini deux spécifications

[4]Néanmoins, la spécification reste abstraite, aucun choix d'implémentation n'est réellement pris. Il est donc possible de considérer ce niveau comme étant encore de niveau PIM.

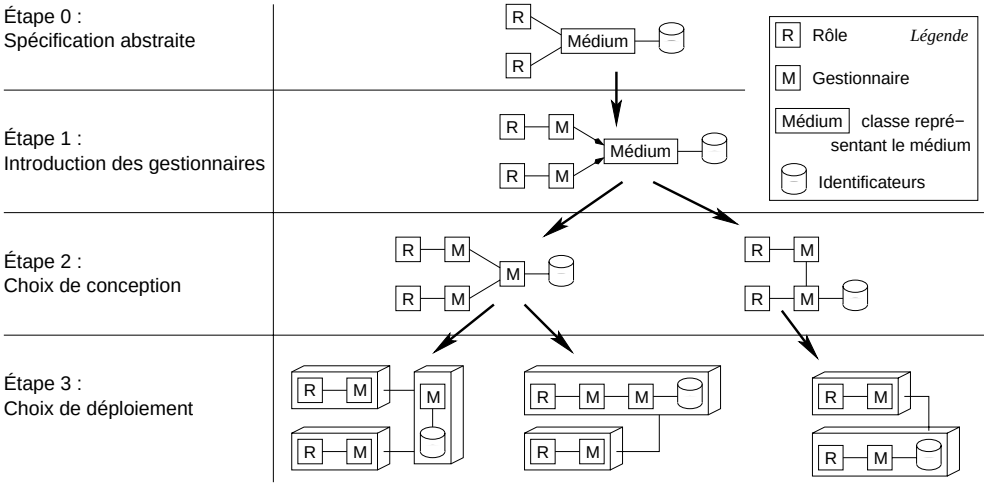

FIG. 3.12: Les étapes du processus de raffinement pour le médium de réservation

d'implémentation et trois spécifications de déploiement. Pour toutes ces spécifications et à tous les niveaux, les services offerts et requis par le médium ainsi que leur sémantique n'ont pas changé. La cohérence et l'unité de l'abstraction de communication réifiée par le médium a donc été conservée pendant toute l'application de ce processus.

En ce qui concerne la mise en œuvre du processus de raffinement et de l'approche MDA dans ce contexte, nous n'avons actuellement pas vraiment travaillé sur ce problème. Il faudrait pour cela créer des profils UML pour définir et marquer à l'aide de stéréotypes (en remplacement de nos conventions de nommage actuelles) les éléments particuliers de nos spécifications (tels que la classe représentant le médium, les gestionnaires, les interfaces de services requis ou offerts, etc.). Ensuite, il faudrait définir les différentes transformations pour passer d'une étape à une autre (du moins dans la mesure du possible). Malgré tout, nous nous heurtons actuellement à un problème critique : à notre connaissance, il n'existe aucun outil de spécifications UML ou d'atelier de génie logiciel permettant de définir des diagrammes de collaboration au niveau spécification. Or ces diagrammes sont à la base de nos spécifications et du processus de raffinement.

La dernière étape dont nous n'avons pas parlé est bien sûr l'implémentation de ces différentes variantes de spécification de conception, pour une même spécification abstraite. Nous traitons cette étape dans le prochain chapitre en décrivant la plate-forme que nous avons développée pour aider à implémenter des médiums.

Chapitre 4

Une plate-forme d'implémentation de médiums

Nous avons développé une plate-forme pour faciliter l'implémentation et l'utilisation de composants de communication. Cette plate-forme est un support possible pour développer des médiums mais il est possible d'utiliser tout autre type d'implémentation via d'autres outils, langages ou architecture, tant que les concepts sous-jacents aux médiums sont respectés.

Les buts principaux de cette plate-forme sont principalement d'aider à la création d'un ensemble de gestionnaires et d'assurer leur cohérence au sein d'un même médium lors de l'instantiation et du déploiement d'un médium.

Cette plate-forme est développée en Java 1.3 (version du JDK de Sun utilisée). Elle est distribuée sous forme de logiciel libre avec une licence GNU GPL[1]. Les sources sont accessibles et téléchargeables sur la plate-forme de développement collaboratif Picolibre de l'ENST Bretagne à cette adresse : `http://picolibre.enst-bretagne.fr` (projet *Medium Framework*).

Dans ce chapitre, nous décrivons d'abord succinctement la plate-forme puis nous nous attardons sur quelques points comme les gestionnaires de rôle et les contrats. Ensuite, nous détaillons un exemple complet de l'implémentation d'un médium et de son utilisation à l'aide de notre plate-forme. Enfin, nous concluons sur des retours d'expérience de l'implémentation et de l'utilisation de médiums à l'aide de notre plate-forme.

4.1 Vue générale de la plate-forme

Le but de la plate-forme est double. Tout d'abord il s'agit d'offrir un support pour aider à l'implémentation de médiums. Ensuite, la plate-forme est également un support d'exécution facilitant l'instantiation et la gestion de médiums. Comme nous l'avons vu, un médium déployé est constitué d'un ensemble de gestionnaires distribués qui communiquent entre eux pour réaliser le service ou le protocole d'interaction réifié par le médium. Chaque gestionnaire est localement associé à un composant connecté au médium (certains gestionnaires particuliers peuvent être indépendants, ils ne sont associés à aucun composant). La plate-forme permet de développer et d'utiliser des gestionnaires respectant ces contraintes.

[1] http://www.gnu.org/licenses/gpl.html

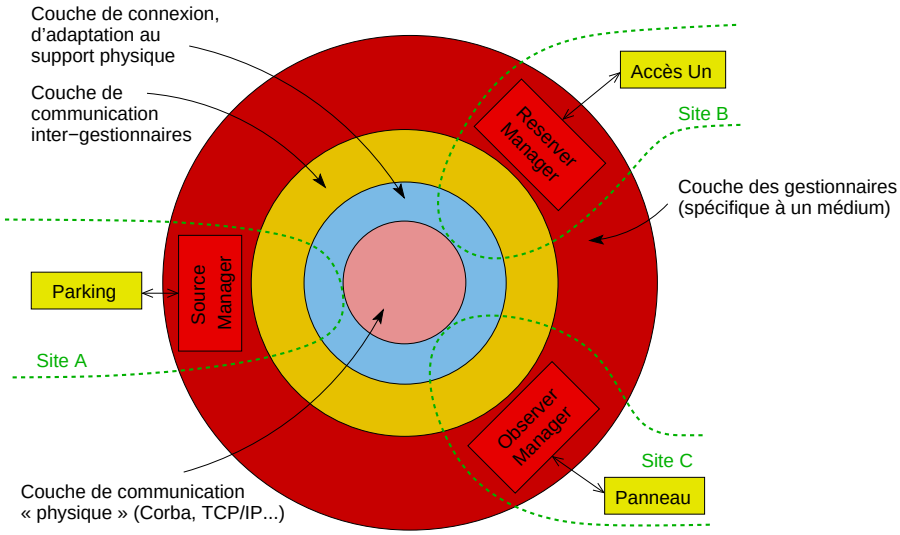

Fig. 4.1: Architecture en couche de la plate-forme d'implémentation de médiums

La plate-forme a une architecture en couche. La figure 4.1 représente cette architecture dans le contexte du médium de réservation et de l'application de gestion de parking (voir la section 2.1.2 page 51). Nous y retrouvons trois composants connectés au médium : le composant parking jouant le rôle de source, le composant panneau jouant le rôle d'observateur et un des deux accès (le composant nommé « Accès Un ») jouant le rôle de réserveur.

Les différentes couches de la plate-forme sont les suivantes :

- La couche externe est composée des gestionnaires de rôle qui forment le médium. Ces gestionnaires ont la responsabilité d'implémenter les services de communication ou d'interaction offerts aux composants en fonction du rôle qu'ils jouent dans l'interaction réifiée par le médium. Ce sont eux qui gèrent l'interaction du point du vue du rôle que joue le composant. À chaque composant connecté à un médium est associé un gestionnaire. Dans notre exemple, les composants connectés peuvent jouer trois rôles différents (source, observateur et réserveur), il y a donc trois gestionnaires différents, que nous retrouvons sur la figure. Ces gestionnaires correspondent aux gestionnaires de l'architecture de déploiement des médiums décrite dans la section 2.5.1 page 74. Cette couche est spécifique à un médium donné car les types et les fonctions des gestionnaires dépendent de l'interaction réifiée par un médium.
- La couche centrale est la couche de communication inter-gestionnaires. Comme son nom l'indique, cette couche est utilisée par les gestionnaires pour réaliser leurs communications. Elle offre une série de primitives permettant à un gestionnaire d'appeler une opération à distance sur un autre gestionnaire ou sur tous les gestionnaires d'un ou plusieurs types. Cet appel d'opération en multicast peut se faire en mode synchrone ou asynchrone.
  C'est dans cette couche que sont stockées toutes les informations concernant la présence et les coordonnées des autres gestionnaires appartenant au même médium. Ces informations sont mises à jour par le guichet dont nous reparlons juste après.

– La dernière couche est utilisée pour l'adaptation au support de communication « de bas niveau ». Cette couche est appelée couche de connexion car elle concerne la gestion des connexions point-à-point entre deux sites (un site correspondant soit à un gestionnaire, soit au guichet). Nous avons réalisé deux versions de cette couche : une s'appuyant sur des sockets TCP et l'autre s'appuyant sur IIOP de CORBA.

4.1.1 Le guichet et les contrats de médium

Un élément spécial de notre plate-forme est le « *medium registry* », que nous appellerons le guichet. Il est utilisé pour gérer la présence et la localisation de tous les gestionnaires de tous les médiums utilisés dans une application donnée. Le guichet est également l'élément qui va gérer les *contrats de médiums*.

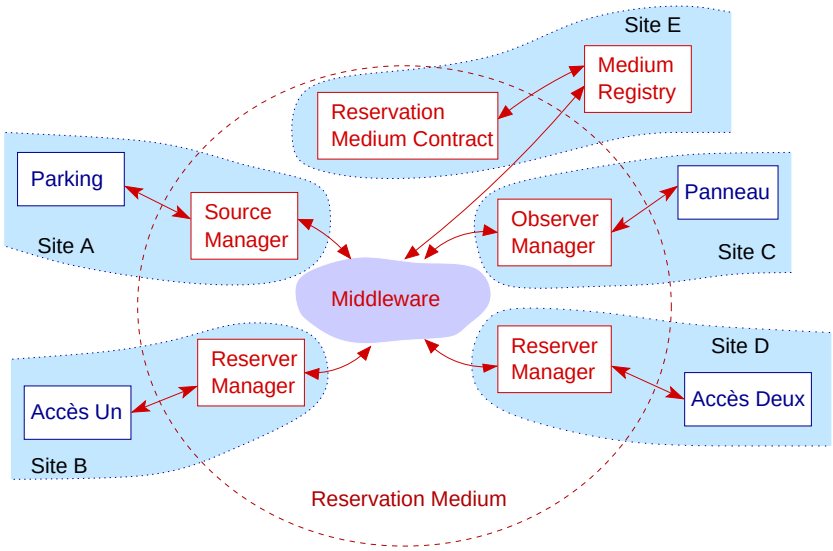

FIG. 4.2: Les gestionnaires, le guichet et le contrat de médium

En conservant le même exemple de l'application de gestion de parking, la figure 4.2 montre le guichet (nommé `Medium Registry`) et le contrat du médium de réservation (nommé `Reservation Medium Contract`)[2]. À chaque gestionnaire est associé un élément de la couche de communication inter-gestionnaires et un élément de la couche de connexion comme nous venons de le voir. Ces éléments ne sont pas représentés sur cette figure. Le guichet est déployé sur un site à part comme le montre la figure. Il est représenté en dehors du médium car il n'appartient pas au médium. En fait, un seul guichet est instantié pour toute une application et c'est cet unique guichet qui gère tous les médiums de l'application. Pour chacun de ces médiums, le guichet gère un contrat. Dans notre exemple, ce contrat concerne le médium de réservation. Il est donc représenté à « l'intérieur » de ce médium car il lui est associé.

Il existe un autre type de contrat qui se place au niveau de l'application et non plus des médiums. Il est lui aussi géré par le guichet et son but est d'autoriser la création

[2]Cette figure est identique à celle de la figure 2.19 page 75, mais avec l'ajout du guichet et du contrat pour le médium de réservation.

de médiums et de définir quel est le contrat associé à chaque médium. Nous reparlons succinctement de ce contrat d'application dans la section décrivant un exemple complet d'utilisation de la plate-forme.

Enregistrement d'un gestionnaire auprès du guichet

La connexion d'un composant au médium se fait par l'instantiation de son gestionnaire. Quand un gestionnaire est instantié, la première chose qu'il fait est de s'enregistrer auprès du guichet. Celui-ci vérifie en fonction du contrat du médium auquel le gestionnaire veut appartenir, s'il est autorisé à faire partie de ce médium (par exemple, dans le cas du médium de réservation, il ne peut y avoir qu'un seul composant de rôle source donc un seul gestionnaire de source ; dans le cas où un gestionnaire de source serait déjà présent, si un deuxième gestionnaire de ce type demande à se connecter, alors sa demande doit être rejetée). Si c'est le cas, il informe alors les gestionnaires de ce même médium de cette nouvelle connexion d'un gestionnaire en leur envoyant ses coordonnées. Ces gestionnaires peuvent alors entrer en communication avec lui. Ainsi, le guichet est l'élément qui centralise toutes les informations sur les gestionnaires des médiums et qui les diffuse au besoin à l'intérieur de chaque médium. Le guichet est donc lancé avant tout gestionnaire. De plus, ses coordonnées doivent être connues et être transmises à l'instantiation des gestionnaires pour qu'ils puissent communiquer avec lui.

Gestion de l'instantiation et de l'utilisabilité du médium

Le guichet, combiné au contrat, sert à gérer le problème de l'instantiation et de l'initialisation d'un médium. En effet, dans un contexte distribué, l'instantiation et le déploiement d'une application composée de plusieurs composants distribués n'est pas une tâche triviale. Il est impossible par exemple de déployer une application ou un médium de manière complètement atomique, comme un seul bloc. Certaines parties seront instantiées avant d'autres et la connexion de ces parties à travers le réseau peut prendre un certain temps. Quand un composant veut communiquer avec un autre composant, il doit vérifier avant toute chose que ce composant est bien présent. Si ces composants communiquent via un ou plusieurs médiums, c'est de la responsabilité des médiums de faire cette vérification. Un médium étant composé d'éléments distribués, il est souvent indispensable pour qu'il soit dans un état de fonctionnement correct, que certains de ces éléments soient présents et dans un état particulier. Dans le cas du médium de réservation par exemple, cela n'a pas vraiment de sens de demander à réserver un identificateur alors que l'ensemble des identificateurs gérés par le médium n'a pas été initialisé. Ainsi, l'utilisabilité de ce médium se définit par le fait que le gestionnaire de source est présent et qu'il a été initialisé[3]. Le guichet est le seul élément capable de connaître ces informations car il a la connaissance des gestionnaires présents dans le médium. Il collabore avec le contrat pour déterminer si le médium est utilisable ou pas. Les contraintes définissant l'état d'utilisabilité sont décrites dans le contrat. La plate-forme offre des fonctionnalités aux gestionnaires pour qu'ils puissent envoyer des informations sur leur état au guichet. Lorsque l'état d'utilisabilité du médium change, le guichet en informe tous les gestionnaires appartenant à ce médium.

[3]Cette utilisabilité doit être celle qui est définie lors de la spécification d'implémentation du médium.

Gestion de la dynamique de connexion de composants

Le guichet permet aussi de gérer la dynamique d'un système et de la connexion de composants à des médiums. Certains médiums fonctionnent avec un nombre quelconque de gestionnaires de certains types et autorisent l'ajout et la suppression dynamiques de ces gestionnaires dans un médium (ce qui correspond à la connexion et la déconnexion dynamiques de composants au médium). Comme le guichet est un élément fixe dans une application et qu'il sert de centralisateur de connaissance sur les gestionnaires, il est très facile de connecter et déconnecter des gestionnaires dynamiquement. Dès qu'ils sont instantiés, ils s'enregistrent auprès du guichet qui diffuse les coordonnées du nouveau gestionnaire aux autres gestionnaires du même médium. C'est la couche de communication inter-gestionnaires qui a la responsabilité d'ouvrir et de gérer les connexions avec les autres gestionnaires, en s'appuyant sur la couche de connexion. La connexion et la déconnexion de gestionnaires est donc transparente pour un gestionnaire donné. Elle offre également, comme nous le verrons plus tard, des mécanismes de communication anonymes avec tous les gestionnaires de certains types. Cela évite pour un gestionnaire de devoir explicitement connaître les coordonnées de tous les gestionnaires avec qui il souhaite communiquer. Tout cela facilite la connexion et la déconnexion dynamiques de composants à un médium.

Même si la couche de communication inter-gestionnaires a la responsabilité de gérer les références sur les gestionnaires du même médium, un gestionnaire peut être informé de la connexion et de la déconnexion de gestionnaires. Cela est utile, par exemple, dans le cas où il aurait besoin de connaître l'identification exacte de certains gestionnaires. Pour cela, deux services peuvent être redéfinis dans un gestionnaire, l'un concernant la connexion de gestionnaires et l'autre leurs déconnexions.

Conclusion sur le guichet

Le guichet est un élément centralisateur d'informations sur la présence des gestionnaires des médiums d'une application. C'est lui qui redistribue au besoin ces informations aux gestionnaires. Il est également responsable de la gestion des contrats de médiums.

La gestion de la dynamique de connexion et d'état d'utilisabilité du médium est décrite par le cycle de vie du gestionnaire dont nous avons parlé dans la section 2.5.2 page 76 avec notamment le diagramme d'états de la figure 2.20 page 76. La plate-forme permet de gérer, au niveau d'un gestionnaire, le passage entre les états d'utilisabilité et d'inutilisabilité et de l'informer de la connexion ou de la déconnexion d'un gestionnaire distant et de réagir en conséquence.

4.2 Détails de la plate-forme

Dans cette section, nous détaillons certaines parties ou éléments de la plate-forme, sans pour autant aborder des aspects trop techniques. Pour implémenter un médium, ou une de ses versions, il faut réaliser les gestionnaires de rôle correspondant à une spécification d'implémentation ainsi que définir un ou plusieurs contrats. Nous parlons dans cette section un peu plus précisément de ces éléments et comment le lien se fait avec l'étape de spécification d'implémentation, qui est la dernière étape du processus de raffinement (avec celle du choix de déploiement).

4.2.1 Les gestionnaires de rôle

Un médium définit un ou plusieurs gestionnaires de rôle en fonction de l'interaction qu'il réifie. La réalisation d'un médium passe donc par l'implémentation des classes correspondant aux différents gestionnaires de rôle. À l'aide de notre plate-forme, la réalisation d'un gestionnaire se fait en créant une classe qui est une spécialisation (par héritage) du gestionnaire générique de notre plate-forme. Ce gestionnaire générique est le squelette d'un gestionnaire et contient tous les mécanismes et méthodes pour utiliser les couches de communication inter-gestionnaires et de connexion. Il convient alors pour un gestionnaire spécialisé de rajouter le code correspondant à l'interaction réifiée par le médium en s'appuyant sur ce squelette.

Les conventions de nommage et l'organisation des classes de gestionnaire

Nous avons vu dans la méthodologie de spécification de médium (voir la section 2.3 page 56) que certaines conventions de nommage devaient être suivies. Elles concernent principalement les noms des interfaces de services offerts et de services requis. Le nom d'une interface commence par la lettre « I » en majuscule puis est suivi par le nom d'un rôle et enfin de « MediumServices » pour les services offerts par le médium ou de « ComponentServices » pour les services requis par le médium sur le composant. Pour utiliser notre plate-forme, pour chaque rôle de chaque médium, deux interfaces Java sont définies, l'une pour les services offerts et l'autre pour les services requis. Le nom de ces interfaces suivent ces mêmes contraintes de nommage que lors de la phase de spécification. Si une interface est vide, il n'est pas nécessaire de la définir. À chaque service contenu dans une interface au niveau spécification correspond un service de même signature au niveau implémentation. Une interface de services offerts doit étendre l'interface de services offerts générique de notre plate-forme. Celle-ci définit des services de gestion de l'utilisabilité du médium qui sont implémentés par le gestionnaire de rôle générique et donc accessibles par les composants pour tous les gestionnaires de tous les médiums.

Les gestionnaires de rôle suivent eux aussi des conventions de nommage. Une classe représentant un gestionnaire doit avoir un nom qui est l'agrégation du nom du rôle (avec une première lettre majuscule) et de « Manager ». Si un médium définit un rôle intitulé *voter*, alors le gestionnaire de ce rôle est une classe intitulée `VoterManager` qui implémente une interface de services offerts `IVoterMediumServices` et qui appelle des services déclarés dans une interface `IVoterComponentServices` qui est implémentée par un composant jouant un rôle *voter*.

Au niveau de l'organisation des classes, là aussi des règles sont définies. Tout d'abord, un médium correspond à un paquetage Java qui a le même nom que celui du médium. À l'intérieur de ce paquetage se trouvent toutes les interfaces du médium (offertes et requises) ainsi que les classes correspondant aux gestionnaires. Des sous-paquetages peuvent aussi être créés. Ils servent à définir des versions du médium. Nous avons vu lors du processus de raffinement, qu'à partir d'une spécification abstraite d'un médium, il est possible de définir plusieurs spécifications d'implémentation et de déploiement. À chacune de ces spécifications correspond un sous-paquetage. Dans chacun d'entre eux, l'ensemble des gestionnaires doit être défini. Les gestionnaires se trouvant dans le paquetage de base du médium correspondent à ce que nous appelons la version par défaut du médium. Pour certains médiums, il n'y a qu'une seule version d'implémentation. Dans ce cas, il n'est

pas utile de faire apparaître des sous-paquetages. Un médium peut donc avoir plusieurs implémentations dont une par défaut (qui n'est pas obligatoire si d'autres versions sont implémentées). Il est possible d'imbriquer plusieurs niveaux de sous-paquetages pour définir les versions.

Ces conventions de nommage et d'organisation servent donc à faire le lien avec les spécifications d'implémentation de médiums en permettant la gestion des différentes versions des médiums et en définissant des interfaces Java pour chacune des interfaces du niveau spécification, ainsi qu'une classe Java pour chaque gestionnaire de rôle défini à ce niveau.

L'identification d'un gestionnaire dans un médium

Dans un médium, un gestionnaire est identifié par cinq champs :
- Le type du médium auquel il appartient.
- L'identificateur donné à ce médium.
- La version de ce médium.
- Le type du rôle du composant auquel il est connecté (autrement dit, le type de gestionnaire).
- L'adresse physique permettant de communiquer avec lui. Le format de cette adresse dépend de la couche de connexion utilisée.

Le type du médium doit correspondre au nom du paquetage du médium. La version du médium est le nom d'un sous-paquetage du médium et le type du gestionnaire est associé au nom de la classe implémentant un gestionnaire donné. À l'instantiation d'un gestionnaire, ces trois paramètres sont donc utilisés pour déterminer la classe Java qui doit être instantiée. L'identificateur du médium est par contre à définir par le composant qui instantie le gestionnaire afin de préciser à quelle instance d'un médium en particulier il veut se connecter. Le dernier champ, l'adresse physique, est automatiquement généré par la couche de connexion. Notez que l'adresse physique du guichet doit être connue pour pouvoir entrer en communication avec lui. Le guichet doit donc être lancé avec une adresse connue.

La communication entre les gestionnaires

La couche de communication inter-gestionnaires, en s'appuyant sur la couche de connexion, s'occupe, à l'aide des informations envoyées par le guichet, d'ouvrir des canaux de communication entre les différents gestionnaires d'un même médium. Cette tâche n'est donc pas du ressort des gestionnaires. Pour leur communication, la couche de communication inter-gestionnaires offre aux gestionnaires un ensemble de primitives d'appel de méthodes à distance. Il s'agit de primitives permettant d'invoquer des méthodes, non pas sur un seul gestionnaire distant comme cela est le cas avec les RPC, mais sur un ensemble de gestionnaires en même temps.

Sans rentrer dans les détails (nous verrons un exemple simple d'utilisation de ces primitives plus tard), voici les grands principes des primitives de communication offertes en ce qui concerne ce multicast d'appel de méthodes à distance. Nous parlons de multicast car il s'agit d'appeler une méthode sur un sous-ensemble de tous les gestionnaires du médium. Ce sous-ensemble est défini de trois façons :

- Tous les gestionnaires d'un certain type (tous les gestionnaires d'émetteur par exemple).
- Tous les gestionnaires de plusieurs types (émetteur et récepteur par exemple).
- Un gestionnaire unique en particulier (en précisant ses coordonnées).

L'appel d'une méthode à distance peut renvoyer, selon la méthode, une valeur de retour ou peut lever une exception en cas de problème. Lorsque du multicast d'appel est effectué, il ne faut plus gérer une valeur de retour ou une exception mais un ensemble de valeurs de retour et d'exceptions. Nos primitives ont été conçues pour cela. De plus, l'appel des méthodes à distance peut se faire de manière synchrone ou asynchrone. Selon le mode choisi, voici comment sont gérées les valeurs de retour et les exceptions :

- Si l'appel de la méthode se fait en mode synchrone, l'utilisation de la primitive est bloquante tant que l'exécution de la méthode appelée n'est pas terminée chez tous les appelés. Si une valeur de retour est renvoyée par la méthode, alors la primitive renvoie la liste de toutes les valeurs retournées par tous les appels. Si un ou plusieurs gestionnaires lèvent une exception pendant l'appel de la méthode, lorsque l'exécution de la méthode est terminée chez tous les appelés, la primitive lève une exception. Celle-ci contient la liste des exceptions levées ainsi que la liste des valeurs retournées (pour les gestionnaires dont l'exécution s'est bien déroulée) si la méthode retourne une valeur.
- Si l'appel se fait en mode asynchrone, l'utilisation de la primitive est non bloquante, elle « retourne » immédiatement. Si la méthode appelée retourne une valeur, alors la primitive renvoie un objet particulier qui permet ultérieurement d'aller lire une par une les valeurs retournées. La lecture d'une valeur renvoie soit une valeur pour une exécution correcte sur un gestionnaire, soit lève une exception en cas de problème lors de l'exécution de la méthode sur un gestionnaire (ce qui avait occasionné la levée de cette exception). Si la méthode ne retourne aucune valeur, alors la primitive ne renvoie pas d'objet permettant d'accéder aux valeurs retournées. Dans ce cas, il n'est pas non plus possible de récupérer les éventuelles exceptions et donc d'être informé de problèmes d'exécution chez certains gestionnaires.

Il y a un ensemble de primitives pour gérer les différents modes d'appel en fonction de la méthode appelée (synchronicité, valeur retournée, définition de l'ensemble des appelés).

Ces primitives offrent une souplesse intéressante dans l'appel de méthodes à distance entre les gestionnaires. Dans beaucoup de médiums, l'appel d'une méthode sur tous les gestionnaires d'un certain type est très courant et cela sans avoir besoin de connaître le nombre de gestionnaires de ce type présents. Nous avons vu dans le chapitre sur le processus de raffinement, que dans le cadre du médium de réservation, ce genre de communication de type « un vers un ensemble de gestionnaires du même type » est utilisé. Si nous avions détaillé les spécifications d'implémentation des autres médiums que nous avons spécifiés (voir la section 2.4 page 61) nous aurions utilisé également ce même principe de communication entre les gestionnaires. De manière générale, vu l'architecture interne des médiums, ce patron de communication se retrouve dans la très grande majorité des spécifications d'implémentation de médiums. Au niveau de l'implémentation, il est donc simple avec notre plate-forme de retrouver et d'utiliser ce patron de communication entre gestionnaires.

De plus, même si un gestionnaire n'est instantié qu'en un seul exemplaire dans un médium, ces primitives de multicast d'appel peuvent être utilisées et cela permet de communiquer avec ce gestionnaire unique sans avoir besoin de connaître ses coordonnées.

Malgré tout, de l'appel de méthode à distance de type classique (un vers un) est également possible à réaliser en connaissant les coordonnées du gestionnaire distant.

La connexion entre un composant et un gestionnaire

La connexion d'un composant au médium se fait en instantiant un gestionnaire. Comme le gestionnaire peut avoir besoin d'appeler des services sur son composant associé, il doit posséder une référence sur celui-ci. Celle-ci est positionnée à l'instantiation. Notre plate-forme offre une fabrique de gestionnaire qui permet d'instantier et d'initialiser un gestionnaire (nous décrivons l'utilisation de cette fabrique dans l'exemple d'utilisation de médium). Un des paramètres à passer à cette fabrique est la référence sur le composant qui instantie le gestionnaire. La fabrique appelle chez ce dernier un service pour lui passer la référence sur le composant. Ce service peut être redéfini dans les gestionnaires, notamment pour vérifier que le composant implémente bien l'interface de services requis. Si ce n'est pas le cas, une exception particulière doit être levée et la connexion du composant au médium échoue.

4.2.2 Les contrats associés à un médium

Notre plate-forme permet d'utiliser des contrats de médium au niveau de l'implémentation. Nous avons précédemment parlé des contrats d'usage et de réalisation lors des différentes spécifications des médiums et du processus de raffinement. Ils permettent de spécifier trois types de contraintes et informations. Tout d'abord, ils spécifient la sémantique du médium et de ses services. Ensuite, ils précisent les contraintes sur la cardinalité de connexion de composants au médium (ces contraintes sont représentées sur le diagramme de collaboration). Enfin, les contrats définissent aussi l'utilisabilité du médium en fonction de son état.

Au niveau de l'implémentation, les contrats utilisables avec notre plate-forme ne vont pas être aussi complets. En effet, vérifier la validité de la sémantique des services au niveau de l'implémentation revient à vérifier que le code des différents gestionnaires est valide au regard de la spécification d'implémentation du médium. Cela consiste à faire de la preuve de programme dans un contexte distribué, ce qui est hors du cadre de cette thèse. Nos contrats concernent donc uniquement la vérification des cardinalités de connexion de composants au médium et également la gestion de l'utilisabilité des services du médium.

Pour chaque médium instantié dans une application, un contrat lui est associé. Il est géré par le guichet car c'est lui qui est le seul élément à connaître tous les gestionnaires d'un médium et tous les médiums. Ce contrat fait deux types de vérification :

La topologie du médium : il s'agit d'autoriser ou de refuser la connexion (ou la déconnexion) de composants au médium. C'est-à-dire d'accepter ou de refuser qu'un gestionnaire appartienne ou quitte le médium. Ainsi, il est par exemple possible de spécifier que l'on veut que seulement un seul gestionnaire d'un certain type appartienne au médium. Si un deuxième gestionnaire du même type demande à faire partie du médium, sa requête sera refusée.

De même, le contrat autorise la communication entre certains types de gestionnaires uniquement. Pour cela, des gestionnaires demandent, en s'abonnant, à recevoir les

coordonnées de certains types de gestionnaires. Certains abonnements peuvent être refusés par le contrat.

L'utilisabilité du médium : dans les spécifications de médiums, nous avons vu qu'il y a un attribut qui définit l'utilisabilité des services du médium. C'est le contrat qui est chargé de définir l'état d'utilisabilité du médium en fonction de la présence et de l'état des gestionnaires. Cet état d'utilisabilité est diffusé à tous les gestionnaires et ils peuvent ainsi gérer l'appel des services qu'ils offrent en fonction de cet état.

Dans la section suivante, où nous détaillons un exemple de médium, nous décrivons un contrat complet et revenons sur la gestion de la topologie et de l'utilisabilité du médium.

La gestion du contrat concerne donc seulement une sous-partie du contrat plus global comme il est défini dans les spécifications. Malgré tout, il permet d'offrir des fonctionnalités intéressantes. Notamment, il est très utile pour gérer l'instantiation d'un médium comme nous l'avons dit précédemment. Un médium étant composé de gestionnaires physiquement distribués et qui ne sont par nature pas instantiables simultanément, le contrat permet de préciser quels sont les gestionnaires indispensables au bon fonctionnement du médium et de gérer cela lors de l'implémentation.

4.3 Exemple d'implémentation de médium et d'utilisation de la plate-forme

Afin de montrer en pratique comment utiliser notre plate-forme, voici un exemple simple mais complet. Nous détaillons la réalisation d'un médium de diffusion de messages (avec un fonctionnement de type boîte aux lettres). Ce médium accepte deux rôles de composant avec ces services offerts par le médium :

- Le rôle émetteur avec le service de signature « `void send(Object message)` » qui permet d'envoyer un message à tous les composants récepteurs connectés au médium au moment de l'appel de ce service.
- Le rôle récepteur avec le service de signature « `Object receive()` » qui retourne le premier message reçu et non lu par le composant. Si aucun message n'a été envoyé ou qu'ils ont tous été lus, alors le service retourne la valeur `null`.

Aucun service n'est requis par le médium sur les composants. L'utilisabilité des services du médium est défini par la présence du composant jouant le rôle émetteur. S'il est présent, alors le médium est utilisable, sinon il ne l'est pas. Il ne peut y avoir qu'un seul composant de ce type connecté au médium. Il n'y a par contre aucune contrainte sur le nombre de connexions de composants jouant le rôle de récepteur.

Dans cette section, nous étudions d'abord l'implémentation des différents gestionnaires de rôle de ce médium et définissons un contrat pour ce médium. Ensuite, nous réalisons une application afin de montrer un exemple de son utilisation et étudions une trace de l'exécution de cette application.

4.3.1 L'implémentation du médium de diffusion

Toutes les classes et interfaces relatives à ce médium se trouvent dans un paquetage nommé fr.enstb.medium.messageBroadcast. Le nom ou type de ce médium est donc « messageBroadcast ». L'implémentation que nous détaillons ici est la version par défaut

du médium. Toutes les classes et interfaces de notre plate-forme sont définies dans un paquetage fr.enstb.medium.kernel ou un sous-paquetage de ce dernier.

Les interfaces de services offerts

Dans notre médium, il y a deux rôles de composant (émetteur et récepteur) à qui deux services sont offerts mais sur lesquels aucun service n'est requis. Il n'y a donc que deux interfaces de services à définir.

```
package fr.enstb.medium.messageBroadcast;                          1

import fr.enstb.medium.kernel.IMediumServices;                     2

public interface ISenderMediumServices extends IMediumServices     3
{
    public void send(Object message);                              4
}
```

FIG. 4.3: Interface `ISenderMediumServices`

Pour le rôle émetteur, l'interface de services est l'interface `ISenderMediumServices` de la figure 4.3. Une interface de services offerts est une spécialisation de l'interface générique de services offerts de notre plate-forme (ligne 3). Ici, nous avons juste à définir la signature du service `send` (ligne 4). L'interface est définie comme appartenant au paquetage du médium (ligne 1).

```
package fr.enstb.medium.messageBroadcast;                          1

import fr.enstb.medium.kernel.IMediumServices;                     2
import fr.enstb.medium.kernel.exception.UnusableServiceException;  3

public interface IReceiverMediumServices extends IMediumServices   4
{
    public Object receive() throws UnusableServiceException;       5
}
```

FIG. 4.4: Interface `IReceiverMediumServices`

Pour le rôle récepteur, l'interface des services qu'on lui offre est l'interface `IReceiver-MediumServices` de la figure 4.4. L'unique service offert est le service `receive` défini à la ligne 5. Il est intéressant de noter que ce service peut lever l'exception `UnusableService-Exception`. Cela a lieu lorsque le service est appelé alors que le médium n'est pas dans un état utilisable.

Le gestionnaire d'émetteur

Le code de la classe réalisant le gestionnaire d'émetteur est représenté sur la figure 4.5 page suivante. Le fonctionnement de ce gestionnaire est assez simple. Lorsque le service

```
package fr.enstb.medium.messageBroadcast;

import java.util.Vector;

import fr.enstb.medium.kernel.RoleManager;
import fr.enstb.medium.kernel.communication.CommunicationManager;

import fr.enstb.medium.kernel.exception.MediumException;

public class SenderManager extends RoleManager implements ISenderMediumServices
{
    public void send(Object msg)
    {
        Object param[] = new Object[1];
        param[0]=msg;
        try
        {
            communicationManager.
                synchronousOperationCall("newMessage",param,"receiver");
        }
        catch(MediumException e)
        {
            System.err.println("Warning: "+e);
        }
    }

    public SenderManager()
    {
        addSubscribedRoleName("receiver");
    }
}
```

Fig. 4.5: Classe `SenderManager`

`send` est appelé par son composant associé, alors il appelle une méthode chez tous les gestionnaires de récepteur afin de leur envoyer le message passé en paramètre du service `send`.

Un gestionnaire de rôle est une sous-classe du gestionnaire de rôle générique de notre plate-forme (classe `RoleManager`). De plus, un gestionnaire doit implémenter son interface de services offerts. La ligne 6 fait apparaître ces contraintes dans la déclaration de la classe `SenderManager`. Nous pouvons aussi noter que cette classe est définie directement dans le paquetage du médium (ligne 1), ce qui correspond à la version par défaut du médium comme nous l'avons vu.

La définition de l'appel du service `send` se fait à partir de la ligne 7. Pour appeler une méthode à distance sur un ou plusieurs gestionnaires, il faut utiliser une des primitives fournies par l'objet `communicationManager` qui correspond à la couche de communication inter-gestionnaires dont nous avons parlé précédemment. Dans notre cas, la primitive que nous utilisons permet d'appeler de manière synchrone une méthode qui ne retourne pas de valeur sur tous les gestionnaires de récepteur (ligne 11). Les paramètres de cette primitive `synchronousOperationCall` sont, dans l'ordre :

- Le nom de la méthode à appeler : `newMessage`.
- La liste des paramètres à utiliser lors de l'appel de cette méthode sous forme d'un tableau d'objets : ici il contient uniquement l'attribut `msg` qui est passé en paramètre de `send`.
- La spécification de la liste des gestionnaires de rôle sur lesquels la méthode est appelée : tous les gestionnaires de récepteur (ce qui est précisé par la chaîne `receiver`).

Si la méthode avait retourné une valeur, il aurait fallu utiliser une autre primitive équivalente permettant en plus de spécifier le type du paramètre de retour attendu.

Pour plusieurs raisons, l'appel de cette méthode `newMessage` peut mal se passer (pas de gestionnaires de récepteur présents, la méthode n'est pas trouvée, sur certains gestionnaires un problème est rencontré, etc.) et donc une exception peut être levée. Dans notre cas, afin de ne pas compliquer l'exemple, ces exceptions ne sont pas explicitement gérées, la ligne 13 se contente d'afficher un message d'erreur.

Nous pouvons donc constater que la communication entre gestionnaires distants est relativement simple et souple. À part le fait que la spécification de la méthode à appeler et de ses paramètres soit un peu particulière et peu pratique[4], cela permet d'appeler facilement une méthode sur un ensemble de gestionnaires sans avoir à en spécifier la liste explicitement. Dans notre exemple, la méthode `newMessage` est appelée sur tous les gestionnaires de récepteur présents au moment de l'appel du service `send`. Le gestionnaire d'émetteur qui demande cet appel n'a pas besoin de connaître chacun de ces gestionnaires ni de savoir combien ils sont ce qui offre une souplesse dans la communication.

Enfin, dans le constructeur de la classe, la ligne 15 précise que ce gestionnaire s'abonne auprès du guichet afin de recevoir les coordonnées des gestionnaires de récepteur. Son fonctionnement lui impose en effet d'appeler des méthodes sur ces gestionnaires, il doit donc les connaître. La gestion des coordonnées des gestionnaires de récepteur est faite automatiquement dans la couche de communication inter-gestionnaires à partir des informations envoyées par le guichet. Malgré tout, comme nous l'avons précisé précédemment,

[4]Nous nous sommes inspirés de l'invocation dynamique de méthode en Java. Les paramètres à utiliser sont – à l'exception du type de retour que nous gérons en plus – les mêmes que pour une invocation dynamique. D'ailleurs, c'est ce mécanisme que nous utilisons pour réaliser l'appel de la méthode.

des mécanismes sont prévus pour qu'un gestionnaire soit informé de la connexion et de la déconnexion de gestionnaires appartenant à des types pour lesquels il s'est abonné auprès du guichet. Comme ici aucun gestionnaire n'a besoin de connaître explicitement d'autres gestionnaires, ces mécanismes ne sont pas utilisés dans notre exemple. Pour les utiliser il suffit de redéfinir deux méthodes particulières (l'une pour la connexion de gestionnaire et l'autre pour leur déconnexion) qui ont pour paramètre les coordonnées d'un gestionnaire. Ces méthodes sont systématiquement appelées lors d'une connexion ou déconnexion de gestionnaire, mais ne font rien par défaut.

Le gestionnaire n'a pas besoin d'appeler de services sur son composant. Il n'a donc pas besoin de gérer explicitement la référence sur celui-ci. Si cela avait été le cas, il aurait fallu redéfinir une méthode particulière dans le gestionnaire, afin de vérifier notamment que le composant implémentait bien l'interface de services requis. Si cela n'avait pas été le cas, une exception aurait été levée. Par défaut, cette méthode ne fait que positionner la référence sur le composant dans un attribut, sans faire de vérification concernant son type et les interfaces qu'il implémente.

Le gestionnaire de récepteur

Le code du gestionnaire de récepteur est représenté sur la figure 4.6 page suivante. Le fonctionnement de ce gestionnaire est également assez simple. Il gère une liste de messages (attribut `messageList` déclaré à la ligne 6) qui contient les messages envoyés par le gestionnaire d'émetteur et qui sont retirés par le composant associé au gestionnaire lors de l'appel du service `receive` défini dans l'interface de services offerts aux composants récepteurs et implémentée par le gestionnaire de récepteur.

Ce service est défini à la ligne 7. Son principe est le suivant :

- Si un message est disponible dans la liste, alors le premier message de la liste est retiré de celle-ci et retourné par la méthode.
- Si aucun message n'est disponible et que les services du médium sont utilisables, alors la méthode retourne la valeur `null`. Sinon, si les services du médium ne sont pas utilisables, la méthode lève une exception précisant que les services du médium ne sont pas utilisables (exception `UnusableServiceException`).

Pour connaître et gérer l'état d'utilisabilité du médium, un gestionnaire dispose de deux attributs : le booléen `mediumUsability` qui précise si les services du médium sont utilisables ou pas et la chaîne de caractères `mediumUnusabilityReason` qui donne une information sur la non-utilisabilité du médium. Ces deux attributs sont disponibles à la lecture par tous les gestionnaires de rôle et leurs valeurs sont automatiquement mises à jour par les informations envoyées par le guichet en fonction de ce que le contrat du médium lui a précisé (c'est le contrat qui détermine si les services sont utilisables ou pas, en fonction de la présence et de l'état des différents gestionnaires présents à un instant donné).

Dans notre cas, si le médium n'est pas utilisable (et si tous les messages ont été lus) une exception est levée. Cette exception doit être gérée par le composant associé au gestionnaire. Il peut pour cela appeler un service pour se bloquer jusqu'à ce que le médium devienne utilisable comme nous le verrons plus tard. Un gestionnaire dispose également de cette fonctionnalité. Cela permet donc de gérer l'attente d'utilisabilité soit du coté du composant, soit dans le gestionnaire, en fonction des besoins et des choix. L'appel d'un service du gestionnaire peut donc donc être bloquant dans certains cas tant

```
package fr.enstb.medium.messageBroadcast;

import java.util.Vector;

import fr.enstb.medium.kernel.RoleManager;

import fr.enstb.medium.kernel.exception.UnusableServiceException;

public class ReceiverManager extends RoleManager implements IReceiverMediumServices
{
    protected Vector messageList = new Vector();

    public Object receive() throws UnusableServiceException
    {
        Object res = null;

        if (!mediumUsability && messageList.isEmpty())
        throw new UnusableServiceException(" Medium services unusable: "+
            mediumUnusabilityReason);

        if (messageList.isEmpty()) return null;
        else
        {
            res = messageList.elementAt(0);
            messageList.removeElementAt(0);
            return res;
        }
    }

    public void newMessage(Object message)
    {
        messageList.add(message);
    }
}
```

FIG. 4.6: Classe `ReceiverManager`

que le médium n'est pas utilisable. En plus de cette utilisabilité des services du médium, il est possible de gérer l'utilisabilité interne d'un gestionnaire. Pour cela, un gestionnaire dispose aussi d'attributs et de fonctionnalités identiques à ceux qui gèrent l'utilisabilité des services du médium. Dans notre exemple, dès son instantiation, un gestionnaire est localement utilisable donc nous n'avons pas besoin ici d'utiliser ces mécanismes.

La méthode `newMessage` est définie à la ligne 16. Cette méthode se contente de rajouter à la fin de la liste le message passé en paramètre. C'est elle qui est appelée à distance par le gestionnaire d'émetteur. Il s'agit d'une méthode classique Java, qui ne doit pas suivre de règles particulières. Lorsqu'une méthode appelée à distance retourne une valeur, elle la retourne normalement via l'opérateur `return`. Lorsqu'elle lève une exception ou que son exécution lève une exception pour une quelconque raison, cela se fait aussi de manière normale. C'est la couche de communication inter-gestionnaires qui s'occupe d'envoyer la valeur retournée ou l'exception levée au gestionnaire distant qui avait fait la demande d'appel de cette méthode.

Pour terminer, nous pouvons constater qu'aucun constructeur n'est défini. En effet, le gestionnaire de récepteur n'a pas besoin de définir d'abonnements. Il est passif dans le sens où il n'appelle pas de méthodes à distance sur d'autres gestionnaires, il n'a donc pas besoin d'ouvrir une connexion avec les autres gestionnaires du médium.

Un contrat associé au médium

Le contrat que nous détaillons ici est un exemple de contrat qui peut être associé au médium. Bien entendu, la définition de plusieurs contrats différents est possible. Celui que nous utilisons a les caractéristiques suivantes :

- Le médium n'accepte que des gestionnaires de rôle nommés `sender` ou `receiver`.
- Le nombre de gestionnaires de récepteur est quelconque. Il y a au plus un seul gestionnaire d'émetteur.
- Le médium est dans un état utilisable si et seulement si le gestionnaire d'émetteur est présent.

Les variantes possibles du contrat peuvent concerner le nombre de composants récepteurs ou émetteurs qui ont le droit de se connecter au médium ainsi que la gestion de l'utilisabilité du médium (par exemple imposer qu'il y ait une instance de gestionnaire de chaque rôle émetteur et récepteur qui soit présente pour que le médium soit utilisable). Le contrat est celui défini lors de la phase de spécification du médium (la topologie l'est au niveau de spécification abstrait et l'utilisabilité également mais avec parfois des contraintes supplémentaires au niveau d'une spécification d'implémentation). Les variantes du contrat correspondent donc à des variantes de spécification. Néanmoins, il est possible de réutiliser les mêmes gestionnaires pour différents contrats, notamment quand la différence ne concerne que des contraintes topologiques. Si les gestionnaires communiquent entre eux par des appels de méthode sur tous les gestionnaires d'un même type, le nombre de ces gestionnaires n'a pas d'influence sur la réalisation du médium. Certains gestionnaires peuvent aussi être adaptés à différentes spécifications de l'utilisabilité.

Le contrat est assez complexe à implémenter et il ne nous semble pas très pertinent de le détailler ici. Néanmoins, voici tout de même quelques informations sur son fonctionnement. Le contrat associé à un médium est instantié par le guichet lorsque le premier gestionnaire d'un médium est lancé. Chaque médium est géré par son propre contrat (s'il

y a plusieurs médiums du même type utilisés dans une application, alors chacun d'entre eux a son propre contrat même s'il s'agit d'instances du même contrat). Le guichet, en appelant des méthodes sur le contrat, peut informer ce dernier de l'occurrence de quatre événements :

- Un nouveau gestionnaire demande à faire partie du médium (demande d'enregistrement ou de connexion).
- Un gestionnaire demande à quitter le médium.
- Un gestionnaire demande à s'abonner à un ou plusieurs types de gestionnaire.
- Un gestionnaire demande à se désabonner à un ou plusieurs types de gestionnaire.

Pour spécifier le comportement du contrat vis-à-vis de ces quatre événements, il suffit dans la classe représentant le contrat de redéfinir les méthodes adéquates. Un contrat doit obligatoirement hériter du contrat générique défini par notre plate-forme. Par défaut, les méthodes du contrat générique acceptent toutes les connexions et déconnexions ainsi que tous les abonnements et désabonnements. La valeur de retour ou la levée d'une exception par une méthode permet de spécifier si le contrat accepte ou non la requête du gestionnaire. Le guichet se charge alors de lui transmettre cette information. Dans le cas où la connexion est acceptée, le guichet transmet également au gestionnaire la valeur de l'état d'utilisabilité du médium. Il diffuse aussi ses coordonnées à tous les gestionnaires s'étant abonnés à ce type de gestionnaire.

Pour son fonctionnement, le contrat peut interroger le guichet pour lui demander des informations sur l'état du médium qu'il gère. Par exemple, il peut demander à connaître le nombre de gestionnaires d'un certain type. De plus, il dispose d'un service que lui offre le guichet pour diffuser à tous les gestionnaires l'état d'utilisabilité du médium. Cela est utile lorsque cet état a changé et qu'il faut alors diffuser à tous les gestionnaires sa nouvelle valeur. Lorsqu'il la détecte, le guichet informe également le contrat de la déconnexion d'un gestionnaire (et cela indépendamment de la demande de déconnexion, ce qui permet notamment de gérer les cas où un gestionnaire a disparu suite à un problème technique ou toute autre raison). Enfin, les gestionnaires ont la possibilité d'envoyer au contrat des informations sur leur état. Pour les recevoir, le contrat doit redéfinir un service du contrat générique. Ce service est appelé sur le contrat par le guichet. Du coté du gestionnaire, ces informations sont envoyées via un service offert par la couche de communication inter-gestionnaires. Dans notre exemple de contrat et de médium, ces mécanismes de communication entre les gestionnaires et le contrat par envoi d'informations ne sont pas utilisés. Avec tout cela, le contrat peut donc gérer l'état d'utilisabilité du médium et les contraintes sur la connexion et déconnexion de certains gestionnaires.

Pour notre exemple de contrat, voila ce que globalement il va exécuter :

Connexion d'un gestionnaire : refus de la connexion des gestionnaires qui ne sont pas de type émetteur ou récepteur. Si le gestionnaire qui se connecte est un émetteur et qu'il en existe déjà un de ce type, alors la connexion est refusée. Sinon, si c'est le premier gestionnaire d'émetteur, alors le contrat passe l'état d'utilisabilité du médium à vrai et diffuse ce nouvel état à tous les gestionnaires. Il n'y a aucune contrainte sur la connexion des gestionnaires de récepteur.

Demande de déconnexion d'un gestionnaire : acceptée pour les gestionnaires de récepteur mais pas pour les gestionnaires d'émetteur.

Abonnements : le seul abonnement autorisé est celui des gestionnaires d'émetteur pour recevoir les coordonnées des gestionnaires de récepteur.

Désabonnements : aucun désabonnement n'est autorisé.

Disparation ou déconnexion d'un gestionnaire : s'il s'agit de la disparition du gestionnaire d'émetteur, alors l'état d'utilisabilité du médium passe à faux et le contrat diffuse cette nouvelle valeur à tous les gestionnaires.

```
public void roleManagerSubscriptionsRequest(RoleIdentifier roleId, Vector roleNameList)
    throws ManagerSubscriptionsRefusedException
{
    ManagerSubscriptionsRefusedException exception = null;
    String roleName = roleId.getRoleName();
    for (int i = 0; i < roleNameList.size(); i++)
    {
        if (!(roleName.equals("sender") &&
            ((String) roleNameList.elementAt(i)).equals("receiver")))
        {
        if (exception == null) exception =
            new ManagerSubscriptionsRefusedException("Subscriptions not allowed.");
        exception.addRefusedSubscription((String) roleNameList.elementAt(i),"not allowed");
        }
    }
    if (exception!= null)
        throw exception;
}
```

Fig. 4.7: Gestion des demandes d'abonnements par le contrat du médium

Afin de simplifier la réalisation du contrat, l'idéal serait de développer un langage pour définir les contrats et ainsi faciliter la définition des contraintes. Car comme le montre la figure 4.7 qui correspond à l'implémentation du service de demande d'abonnements (nommé `roleManagerSubscriptionsRequest`), l'écriture du contrat est assez laborieuse. Sans rentrer dans une étude complète et détaillée de cette méthode, nous pouvons décrire succinctement ce qu'elle fait. Cette méthode lève une exception si certains abonnements ne sont pas autorisés. Sinon, la méthode se termine normalement si tout est accepté. La méthode liste toutes les demandes d'abonnements (d'où la boucle) et dans le cas où elle rencontre un abonnement différent de celui des émetteurs pour les récepteurs, elle enregistre cet abonnement dans l'exception (en la créant si nécessaire). L'écriture de ce genre de méthode est assez fastidieuse et se fait toujours a priori de la même façon. Il serait donc intéressant de disposer d'un langage spécialisé pour écrire la majorité des contraintes du contrat dans un formalisme plus simple et plus expressif. Il suffirait ensuite de générer automatiquement la classe Java correspondante.

En reprenant l'exemple de la méthode précédente, cela pourrait donner :

```
Subscriptions::
allowed: Sender -> Receiver
rejected: others withMessage ("Subscriptions not allowed.")
```

Cette définition de la contrainte sur les abonnements est bien plus facile à écrire et à comprendre que sous la forme de méthode de la classe du contrat comme sur la figure 4.7.

4.3.2 L'utilisation du médium de diffusion

Après avoir vu comment implémenter le médium de diffusion, nous allons voir comment l'utiliser. Pour cela, nous allons créer deux types de composant : un composant émetteur et un composant récepteur. Nous allons ensuite instantier plusieurs fois ces composants et les connecter entre eux via le même médium. Le composant émetteur envoie une dizaine de messages, à des intervalles de temps aléatoires. Les composants récepteurs vérifient toutes les deux secondes qu'ils n'ont pas reçu de messages.

L'instantiation des gestionnaires

Pour commencer, afin de pouvoir se connecter au médium, il faut qu'un composant instantie son gestionnaire. Cela se fait en utilisant la fabrique de gestionnaires offerte par notre plate-forme. Son but est non seulement d'instantier le gestionnaire mais aussi de l'initialiser en demandant l'enregistrement du gestionnaire et la validation de ses abonnements auprès du guichet et du contrat.

Pour créer un gestionnaire, il faut disposer de plusieurs paramètres :

- Le nom du paquetage Java dans lequel se trouve le paquetage du médium.
- Le type du médium.
- La version du médium (facultatif, dans ce cas, la version par défaut est sélectionnée).
- Le type de gestionnaire à instantier.
- L'identificateur du médium.
- Les coordonnées du guichet (dont le contenu et le type dépendent de la couche de connexion utilisée).

Le nom de la classe correspondant au gestionnaire à instantier est définie en concaténant les quatre premiers paramètres. Si l'instantiation dynamique de cette classe se passe mal, cela signifie que le gestionnaire désiré n'existe pas ou que les paramètres sont erronés. La concaténation des paramètres se fait de la manière suivante :

> *Nom de la classe = paquetage + nom du médium + version du médium + type du gestionnaire (avec une majuscule et suivi de « Manager »)*

```
ISenderMediumServices manager = null;                                          1
try                                                                            2
{
    MediumManagerBuilder builder =
        new MediumManagerBuilder("fr.enstb.medium", "messageBroadcast",
                                 "monMedium", "sender", loc);                  3
    manager = (ISenderMediumServices) builder.instantiateAndRegisterManager(); 4
}
catch (Exception e)                                                            5
{ // Treat the exception
}
```

FIG. 4.8: Instantiation et initialisation du gestionnaire d'émetteur

Dans le cas du gestionnaire d'émetteur, la figure 4.8 définit le code permettant de l'instantier et de l'initialiser. Le gestionnaire est déclaré à la ligne 1 (attribut `manager`).

Il est intéressant de noter qu'il est typé par l'interface de services offerts par le médium pour les composants de rôle émetteur. En effet, il s'agit ici d'instantier un gestionnaire offrant ces services qui sont les seuls points d'accès et d'interaction du composant avec le médium. De plus, comme plusieurs implémentations du même gestionnaire peuvent être définies, le typage par l'interface de services offerts permet de s'abstraire du besoin de connaître la version (et donc la classe exacte correspondante) du médium utilisée en dehors de la phase d'instantiation.

La ligne 3 définit et instantie une fabrique de gestionnaire (une instance de la classe `MediumManagerBuilder`). Nous utilisons ici le constructeur pour utiliser la version par défaut du médium car notre médium n'est réalisé qu'en une seule et unique version. Un autre constructeur permettant de définir en plus la version du médium à utiliser existe également. Les paramètres sont les suivants, dans l'ordre :

- Le paquetage de base où se trouve le médium (`fr.enstb.medium`).
- Le type du médium (`messageBroadcast`).
- L'identificateur du médium (`monMedium`).
- Le type du gestionnaire (`sender`).
- L'adresse du guichet (`loc`).

La concaténation des paramètres pour définir la classe à instantier donne dans ce cas précis la classe suivante : fr.enstb.medium.messageBroadcast.SenderManager.

Une fois cette fabrique instantiée, le gestionnaire est à son tour instantié et initialisé (ligne 4). Il est possible avant de faire cela, de configurer la fabrique pour qu'elle utilise d'autres couches de communication inter-gestionnaires ou de connexion que celles qui sont définies par défaut. C'est également lors de la configuration de la fabrique qu'il est possible de passer la référence sur le composant instantiant le gestionnaire, en appelant le service `setComponent` sur la fabrique. Par défaut, si ce service n'est pas appelé (comme dans notre exemple), la référence du composant est égale à `null`. Cela ne pose pas de problème dans notre médium, car aucun gestionnaire n'appelle de service sur son composant associé.

L'instantiation de la fabrique ou du gestionnaire peut poser problème pour diverses raisons (classe du gestionnaire non trouvée, communication avec le guichet impossible, connexion au médium refusée[5], etc.) ce qui se traduit par la levée d'une exception qu'il faudra traiter (ligne 5). Si aucune exception n'est levée, le gestionnaire est correctement instantié et initialisé. Il est alors possible d'appeler les services (définis dans son interface de services offerts) qu'il implémente.

L'instantiation du gestionnaire de récepteur se fait de manière similaire. Nous ne détaillons donc pas cette partie. La différence concerne bien sûr le type du gestionnaire à utiliser (récepteur au lieu d'émetteur) et le typage à utiliser (l'interface de services offerts par le médium aux composants jouant le rôle de récepteur).

```
Random r = new Random();                                                    1
String message = "message ";                                                2

for (int i = 0; i < 10; i++)                                                3
{
    System.out.println(message + i);                                        4
    manager.send(message + i);                                              5
    try { Thread.sleep(1000 + r.nextInt(3000)); } catch (Exception e) { }   6
}
```

FIG. 4.9: Utilisation du médium de diffusion par le composant émetteur

L'utilisation des gestionnaires

L'utilisation du médium par le composant émetteur est décrit sur la figure 4.9. Ce composant envoie une dizaine de messages (ligne 3). Entre la diffusion de chaque message, il fait une pause comprise entre une et quatre secondes (ligne 6) . Cette diffusion se fait bien entendu en utilisant le service d'émission offert par le médium. La référence sur le médium est la référence sur le gestionnaire dont nous venons de décrire l'instantiation. Il suffit alors d'appeler le service `send` sur ce gestionnaire pour diffuser un message (ligne 5). L'utilisation des services du médium est donc extrêmement simple et nous pouvons constater qu'il n'est pas nécessaire de s'occuper d'aucune tâche ou gestion relative à la communication. Ici, c'est le médium qui est chargé de diffuser le message à tous les composants récepteurs sans que le composant émetteur n'ait besoin ni de les connaître explicitement ni même de savoir combien ils sont.

La figure 4.10 page suivante décrit le code utilisé par un composant récepteur pour utiliser le médium. Le composant tente de lire un message une dizaine de fois (ligne 1). Entre chaque lecture, il fait une pause de deux secondes (ligne 4). La réception d'un message se fait là aussi de manière simple, en appelant le service `receive` (ligne 3) sur l'instance du gestionnaire de récepteur du médium qui a été retournée par la fabrique (l'attribut `manager`). Comme ce service peut lever une exception en cas d'inutilisabilité des services du médium, il faut gérer cette éventualité (ligne 5). Ce cas se produit lorsque le composant demande à retirer un message alors qu'aucun composant émetteur n'est présent. La gestion de l'exception consiste à se bloquer en attendant que le médium devienne utilisable (ligne 7), c'est-à-dire à attendre qu'un composant émetteur se connecte au médium d'après ce que nous avons vu dans la définition du contrat.

Le contrat d'application

Il reste une dernière chose à implémenter : le contrat d'application. Le contrat d'application a pour principe de gérer des contraintes sur les médiums, à l'image du contrat

[5] Dans le cas où certains abonnements seraient refusés par le contrat du médium, aucune exception ne sera levée ici. Nous considérons en effet que cela permet tout de même au gestionnaire de fonctionner. Il est néanmoins possible d'avoir un contrôle plus précis des abonnements. Dans ce cas, il faut à la place de la méthode `instantiateAndRegisterManager` qui fait toutes les opérations (instantiation, connexion et abonnements), appeler une méthode pour instantier et demander la connexion auprès du guichet puis appeler une deuxième méthode pour lancer la requête concernant les abonnements. Cette dernière méthode renvoie alors une exception en cas d'abonnements refusés.

```
for (int i = 0; i < 10; i++)                                          1
{
    try                                                               2
    {
        System.out.println("Receive: " + manager.receive());          3
        try { Thread.sleep(2000); } catch (Exception e) { }           4
    }
    catch (UnusableServiceException e)                                5
    {
        System.out.println(" The medium is unusable: " + e);          6
        manager.waitForMediumUsability();                             7
        System.out.println(" The medium is ok now.");                 8
    }
}
```

Fig. 4.10: Utilisation du médium de diffusion par le composant récepteur

de médium qui gère des contraintes au niveau des gestionnaires. Nous n'avons pas vraiment travaillé pour le moment sur ce contrat d'application. Actuellement, le but de ce contrat est uniquement d'autoriser ou de refuser la création d'un médium (un médium existe à partir du moment où au moins un gestionnaire de ce médium est instantié). Si elle est autorisée, alors il faut définir le contrat de médium qui est associé à un nouveau médium. Ce contrat d'application est lui aussi géré par le guichet. C'est d'ailleurs un paramètre indispensable à son lancement. Il faut spécifier un contrat d'application lorsque l'on l'instantie. Si ce n'est pas fait, le guichet utilise un contrat d'application par défaut qui autorise toute création de médium et qui associe à chaque médium un contrat générique qui accepte toutes les demandes de connexion, de déconnexion, d'abonnements et de désabonnements.

```
public MediumContract mediumCreationRequest(String mediumType,
                                    String mediumVersion, String mediumId)   1
    throws MediumCreationRefusedException                                    2
{
    if (! (mediumType.equals("messageBroadcast") ))                          3
        throw new MediumCreationRefusedException("The medium is"+
                                                 "not a messageBroadcast");  4
    return new MessageBroadcastMediumContract(registry);                     5
}
```

Fig. 4.11: Gestion de la création du médium et de son contrat

La figure 4.11 définit le service de demande de création du médium (ligne 1) dans le cas de notre application nous servant d'exemple. Ce service est appelé quand le premier gestionnaire appartenant au médium demande à s'enregistrer auprès du guichet. Si le médium qui va être créé n'est pas un médium de diffusion (ligne 3) alors une exception est levée (ligne 4) afin de refuser la création de ce nouveau médium, et par la même occasion, refuser la connexion du gestionnaire à ce médium (et pour cause, le médium n'existera

pas). Si la création est acceptée, alors le contrat instantie et retourne un contrat de médium qui est celui que nous avons défini précédemment (ligne 5). Dans cet exemple, le contrat d'application n'autorise dans l'application que la création de médiums de diffusion mais sans restriction sur leur nombre.

4.3.3 Traces d'exécution

Afin de comprendre plus clairement le fonctionnement de ce médium, de son contrat et des composants émetteurs et récepteurs que nous venons de décrire, nous allons étudier une trace d'exécution.

La figure 4.12 page suivante représente une trace d'exécution où interviennent quatre machines différentes. Deux d'entre elles sont utilisées pour lancer des composants émetteurs et les deux autres des composants récepteurs. Le temps correspond à un temps logique et est utilisé pour décrire l'enchaînement des événements et des actions. Il n'est pas lié à un quelconque temps physique ou aux temps de pause effectués par les composants pendant leur exécution. Au temps 1, un émetteur est lancé sur la machine A et un récepteur est lancé sur la machine C. Au temps 2, l'émetteur émet un message. Au même moment, le récepteur lit un message mais le message envoyé par l'émetteur ne lui est pas encore arrivé : il lit donc la valeur `null`. Le message envoyé est lu au temps suivant (temps 3). Si on lance un second émetteur sur la machine D (temps 4), il reçoit également les messages envoyés par l'émetteur (le message 2 lu au temps 6).

Avec ces traces d'exécution, nous pouvons étudier la gestion du contrat concernant la topologie de connexion de composants et l'utilisabilité du médium. Tout d'abord, si l'on tente de lancer un second émetteur (temps 7), sa connexion est refusée par le contrat car un émetteur est déjà présent sur la machine B (temps 8). Ensuite, si le composant émetteur est arrêté (temps 10), le contrat passe le médium dans un état inutilisable. Cela provoque le blocage du récepteur de la machine C lorsqu'il veut lire un message (temps 10). Pour l'autre récepteur (machine D), il n'est pas tout de suite bloqué car il n'avait pas encore lu le message 3, ce qu'il fait au temps 10. Ensuite, à sa lecture suivante, il est lui aussi bloqué (temps 11). Lorsqu'un nouvel émetteur est lancé sur la machine B, les deux récepteurs qui étaient bloqués en attente de l'utilisabilité du médium se retrouvent de nouveau actifs (temps 12). Ils recommencent à lire des messages, qui sont cette fois envoyés par ce nouvel émetteur (temps 13 et suivants).

4.4 Conclusion sur la plate-forme d'implémentation de médium

4.4.1 Retour sur le but et l'état de la plate-forme

La plate-forme que nous avons développée a pour but d'aider à la création et à l'utilisation de médiums. Pour cela, elle est conçue pour s'inscrire comme la dernière étape du processus de raffinement, en se plaçant naturellement après les étapes de spécification d'implémentation et de déploiement. La création d'un médium se fait en respectant les contraintes de nommage sur les gestionnaires et interfaces qui étaient définies lors de la spécification. La plate-forme permet de réaliser plusieurs versions d'un même médium, car le processus de raffinement permet de spécifier plusieurs variantes d'implémentation et de déploiement. Au niveau de l'implémentation, les interfaces de services sont les mêmes

Temps	Machine A	Machine B	Machine C	Machine D
1	> **java Emetteur**		> **java Recepteur**	
2	message 1		Receive: null	
3			Receive: message 1	
4	message 2			> **java Recepteur**
5			Receive: message 2	
6				Receive: message 2
7	message 3	> **Java Emetteur**		
8		Error : already a sender manager connected		
9			Receive: message 3	
10	**^C**		The medium is unusable	Receive: message 3
11				The medium is unusable
12		> **java Emetteur**	The medium is ok now	The medium is ok now
13			Receive: null	Receive: null
14		message 1		
15			Receive: message 1	Receive: message 1
17		message 2		
18		message 3		
19			Receive: message 2	Receive: message 2

Fig. 4.12: Traces d'exécution lors de l'utilisation du médium de diffusion

que lors de la spécification. Cela permet de conserver la cohérence du médium, même à l'implémentation. De plus, la plate-forme permet de gérer l'utilisabilité d'un médium et les contraintes sur les cardinalités de connexion de composants (comme cela se fait dans la spécification d'un médium, avec le cycle de vie d'un gestionnaire et les cardinalités sur le diagramme de collaboration). Enfin, elle offre des patrons de communication inter-gestionnaires qui sont identiques à ceux utilisés dans les spécifications d'implémentation de médiums, c'est-à-dire des communications du type « un gestionnaire vers tous les gestionnaires d'un certain type ».

Globalement, cette plate-forme peut être vue comme un médium qui ne réifie aucune interaction. En effet, les gestionnaires de rôle génériques de cette plate-forme sont opérationnels mais ne font rien. La création d'un médium réifiant une interaction donnée se fait en spécialisant ces gestionnaires génériques pour qu'ils réalisent l'interaction voulue.

Notre plate-forme offre un support d'exécution et d'implémentation de ces gestionnaires. Nous avons axé notre développement dans deux directions principales, afin d'assurer le lien avec les étapes de spécification et de déploiement. La première est la communication entre les gestionnaires. Notre plate-forme offre des services d'appels de méthodes à distance de type multicast, en permettant d'appeler une même méthode sur un sous-ensemble de tous les gestionnaires d'un médium (tous les gestionnaires du même type ou de plusieurs types). Cette communication est relativement souple car elle se fait sans qu'un gestionnaire n'ait besoin de connaître explicitement les autres gestionnaires avec qui il veut communiquer ni même combien ils sont. Cela facilite la gestion de la distribution et de la communication lors de l'implémentation de médiums.

Le deuxième point concerne la dynamicité de la connexion et de la déconnexion des gestionnaires couplée avec la gestion d'un contrat par médium. En effet, dans un contexte distribué, il est impossible d'instantier tous les éléments (ici les gestionnaires et les composants) d'une application simultanément, de manière atomique. De plus, certains médiums acceptent des connexions et des déconnexions dynamiques de composant. Notre plate-forme intègre des mécanismes permettant de gérer cette dynamicité et de permettre aux gestionnaires de se connecter entre eux. Le contrat associé à un médium permet de contraindre la connexion et la déconnexion de certains types de gestionnaire et également de déterminer quand les services du médium sont utilisables en fonction de la présence des gestionnaires et de leurs états.

Enfin, les couches de communication inter-gestionnaires et de connexion sont remplaçables, ce qui est un point important de notre plate-forme. Pour la couche de connexion, il est actuellement possible d'utiliser une couche basée sur les sockets TCP ou une couche s'appuyant sur CORBA. Seules ces deux couches sont pour le moment disponibles mais de nouvelles couches utilisant d'autres supports (comme RMI par exemple) peuvent être définies.

Notre plate-forme n'est pas un outil industriel mais doit plutôt être vue comme un prototype. À ce titre, elle n'est pas parfaite et nous pouvons citer quelques uns de ses défauts. Pour commencer, en terme de performance, il y aurait sans doute des améliorations à apporter. La plate-forme est multi-threadée, il y a beaucoup d'introspection, les méthodes sur les gestionnaires sont à chaque fois appelées par invocation dynamique dans un nouveau thread créé dans ce but. Si ces techniques sont nécessaires afin d'assurer une bonne flexibilité, il serait néanmoins intéressant d'améliorer leur utilisation, en exploitant par exemple des « pools » de threads spécialisés. Ensuite, nous avons entièrement développé toutes les fonctionnalités dont nous avions besoin. Mais nous pouvons remarquer

que le guichet remplit un rôle qu'un service de nommage CORBA pourrait jouer en partie. Il aurait sans doute été pertinent de réutiliser ce genre d'outils déjà existants. Enfin, comme nous l'avons vu, l'expression des contrats est assez pénible. Un langage spécialisé pour décrire les contraintes du contrat permettrait de les définir bien plus facilement.

4.4.2 Expérimentations et intérêts des abstractions d'interaction à l'implémentation

Cette plate-forme nous a permis de faire quelques expérimentations intéressantes. Par exemple, en implémentant des médiums complexes comme le médium de diffusion vidéo (celui de l'application de vidéo interactive de la section 2.1.1 page 48), nous avons pu voir l'intérêt de la réification d'abstractions d'interaction « sophistiquées ». Pour ce médium, il faut en effet implémenter des tâches non triviales comme l'encodage et le décodage de flux ou l'émission en multicast de paquets de données multimédia. Mais après avoir réalisé cette interaction de diffusion de flux vidéo, nous nous sommes rendu compte qu'elle était très simple à utiliser. En effet, pour diffuser un flux vidéo, un composant émetteur appelle uniquement un simple service prenant en paramètre le nom du fichier contenant le film à diffuser. Ensuite, le médium s'occupe de toutes les tâches complexes concernant la diffusion du flux vidéo. Du coté des récepteurs, la réception du flux est également simple : ils récupèrent, via un service du médium, une référence sur un panneau graphique Java dans lequel le film est automatiquement joué. Ils n'ont alors qu'à ajouter ce panneau dans leur interface graphique. En résumé, toute la complexité de l'interaction (que cela concerne l'abstraction ou son implémentation) est entièrement localisée dans le médium, les composants utilisant ce médium n'ayant absolument pas besoin d'accomplir une quelconque tâche liée à l'interaction (à part celle bien sûr de demander au médium de diffuser un film). De plus, la réutilisation de ce médium est très simple. Ainsi, la séparation des préoccupations fonctionnelles et interactionnelles à l'implémentation est complète.

Le médium de réservation que nous avons utilisé comme exemple dans le chapitre traitant du processus de raffinement a été implémenté en deux versions différentes. Une version centralisée et une version distribuée comme elles avaient été spécifiées pendant l'application de ce processus. Hormis une différence d'initialisation (avec le gestionnaire de réservation qui n'existe que dans la version centralisée), l'utilisation de ces deux versions se fait exactement de la même manière. Du point de vue des composants réserveurs par exemple, il y a une complète transparence en ce qui concerne la version utilisée. Pour ces composants, il est donc possible d'utiliser une version plutôt qu'une autre en fonction des besoins, sans avoir à modifier en quoi que soit les composants ou leur façon d'utiliser les services de réservation. Cela permet de faire évoluer relativement facilement une application en fonction de contraintes non-fonctionnelles, comme nous en avons discuté dans la section 3.3.1 page 89 et [32]. Ainsi, au niveau de l'implémentation et du déploiement, la gestion des versions de médiums ne pose pas de problème particulier.

Enfin, en implémentant l'application de vidéo interactive que nous avons décrite dans la section 2.1.1 page 48, nous nous sommes rendus compte de l'intérêt d'avoir des interactions complexes à l'implémentation. Cette application contient deux médiums : un médium de vote et un médium de diffusion vidéo. Les deux types de composant utilisant ces médiums (le serveur et le client) sont très simples. En fait, toute la complexité de cette application réside dans les interactions entre le serveur et les clients. Le fonctionne-

ment du serveur est basique : diffuser un film puis lancer un vote à la fin de chaque film. Le client est lui aussi peu complexe, il se contente de répondre aux demandes de vote et d'afficher le film (en fait d'intégrer un panneau graphique dans son interface). Chacun de ces composants est programmé en seulement quelques dizaines de lignes de Java. Par contre, les médiums sont bien plus complexes, notamment le médium de diffusion vidéo. Avoir ces médiums permet donc de simplifier les applications les utilisant (dans ce cas précis en tout cas) et aussi de pouvoir facilement utiliser des abstractions d'interaction complexes. Malgré tout, même si notre plate-forme offre un support d'implémentation de médiums et si elle permet de gagner du temps dans leur implémentation, cette tâche de réalisation de médium peut s'avérer longue et compliquée. Mais elle n'est à effectuer qu'une seule fois et à la prochaine utilisation de la même abstraction d'interaction dans une autre application, aucun travail concernant l'implémentation de cette abstraction n'aura à être fourni de nouveau.

Conclusion

Contribution de la thèse

Dans ce document, nous avons présenté un processus de réification d'abstractions de communication ou d'interaction sous forme de composants logiciels (que nous appelons des médiums). Ce processus est composé de plusieurs parties :

- Une méthodologie de spécification de médiums en UML. Cette spécification est réalisée à un niveau abstrait, indépendamment de toute implémentation. Dans la terminologie du MDA, il s'agit d'une spécification de niveau PIM. Cette spécification est le contrat du médium qui doit ensuite être respecté indépendamment du niveau de manipulation considéré (de cette spécification abstraite à l'implémentation).
- Une architecture de déploiement de médiums qui offre la flexibilité nécessaire pour implémenter a priori n'importe quelle abstraction d'interaction.
- Un processus de raffinement qui permet de transformer une spécification abstraite de médium en une ou plusieurs spécifications d'implémentation, en prenant en compte l'architecture de déploiement et différents choix de conception ou d'implémentation. Dans la terminologie du MDA, le processus de raffinement sert à transformer un modèle PIM en un ou plusieurs modèles PSM.

En complément de ce processus, nous avons développé une plate-forme d'implémentation de médiums. Nous l'avons utilisée pour réaliser plusieurs médiums et des applications les utilisant. Cela afin de vérifier en pratique l'intérêt et la faisabilité des médiums au niveau de l'implémentation. La plate-forme s'inscrit dans la continuité du processus de raffinement. Elle permet de réaliser différentes versions d'un médium et de gérer une sous-partie de son contrat à l'exécution. Cette plate-forme n'est pas la seule qui peut être utilisée pour développer des médiums. Il est bien entendu possible d'utiliser d'autres intergiciels ou technologies, tant que les concepts sous-jacents aux médiums sont respectés.

La réification d'abstractions de communication sous forme de composants logiciels offre des propriétés intéressantes pour leur manipulation. Tout d'abord, cela permet d'encapsuler l'abstraction dans une entité clairement définie en terme d'interfaces d'interaction avec les composants l'utilisant (grâce aux interfaces de services offerts et requis). Ensuite, cela permet de bien séparer l'aspect spécification de l'aspect d'implémentation, ce qui offre la possibilité d'avoir plusieurs implémentations d'une même abstraction, ses interfaces de services et sa sémantique ne changeant pas. Il est donc possible dans une application, de remplacer une version de médium par une autre version de ce médium correspondant à une implémentation différente. Enfin, cela permet de réutiliser une abstraction d'interaction dans différents contextes ou applications.

Le but de notre processus de réification d'abstractions de communication est d'offrir un cadre précis pour manipuler la communication ou l'interaction sous forme de composant logiciel et ainsi offrir les propriétés que nous venons de citer. Le processus facilite le passage d'un niveau à un autre et permet de conserver la cohérence et l'unité d'une abstraction tout au long du cycle de développement du logiciel. Le médium conserve exactement les mêmes interfaces de services offerts et requis, y compris à l'implémentation. Que l'on soit à un niveau de conception abstraite, de conception d'implémentation ou de déploiement, l'abstraction réifiée est toujours la même. Le processus permet de définir plusieurs variantes d'implémentation de la même abstraction afin de pouvoir adapter l'utilisation d'une abstraction à un contexte particulier et ainsi gérer différents types de contraintes non-fonctionnelles ou de choix d'implémentation.

Limites de notre approche

Notre approche présente quelques limitations. En ce qui concerne la méthodologie de spécification de médiums à un niveau abstrait, la méthodologie en elle-même n'est pas uniquement en cause, mais les limitations viennent également d'UML. Ce langage de modélisation présente en effet des lacunes sémantiques. À ce titre, il ne sera pas toujours possible de spécifier complètement ou très formellement toutes les abstractions de communication. De plus, bien qu'ayant essayé de respecter au maximum la norme UML dans sa version 1.3, nous avons du rajouter deux extensions au langage OCL. Il ne nous semblait pas possible de se passer de ces deux extensions dans de nombreuses spécifications et donc cela pourra poser quelques problèmes lors de l'utilisation d'un outil de gestion et d'édition de diagrammes UML qui ne saura pas manipuler ces deux extensions.

Le processus de raffinement que nous avons défini est sans doute trop simple dans certains cas. Comme nous l'avons vu, une des étapes consiste à définir une spécification d'implémentation avec un ensemble de contraintes que l'implémentation doit respecter. Mais son but n'est pas de décrire comment l'implémentation le fait. Cette étape supplémentaire pourrait être intéressante à définir. De même, des étapes intermédiaires pourrait être utiles à différents niveaux. Si nous n'en avons pas parlé, il est néanmoins possible d'introduire ces étapes et modèles dans le processus de raffinement.

Lors de l'instantiation et du déploiement d'une application distribuée, le problème de la gestion de la localisation et de la distribution des composants formant l'application n'est pas trivial. Dans une application utilisant des médiums, toutes les interactions et communications entre les composants passent par ces médiums. C'est naturellement dans ces derniers que cette gestion se fait. Notre plate-forme d'implémentation est donc logiquement conçue pour gérer cela. Mais dans le cas où d'autres technologies seraient utilisées pour réaliser des médiums, le développeur devra s'acquitter de cette tâche si elle n'est pas rendue par le support d'exécution choisi. Cela pourra grandement compliquer l'implémentation d'un médium. Néanmoins, quoiqu'il arrive, cette tâche se doit d'être réalisée. Habituellement, ce sont les composants qui s'en chargent en s'appuyant sur des services fournis par l'intergiciel ou la plate-forme de composants utilisé. Déplacer cette problématique dans les médiums complexifie d'un côté leur développement mais offre également un cadre pour gérer ce problème. Ce qui est plutôt intéressant et supprime cette responsabilité des composants.

Perspectives

Notre approche couvre le processus de développement du logiciel, de la conception abstraite à l'implémentation. Une phase que nous ne traitons pas est l'analyse. Nous partons d'une abstraction de communication connue et nous la réifions, mais nous n'avons pas parlé de comment découvrir cette abstraction. Ce problème n'est pas spécifique aux médiums mais est général à tous les composants logiciels. Résoudre ce problème revient à répondre à la question suivante : quelle est la responsabilité du composant, quelle est sa frontière ? Nous avons commencé à travailler sur ce sujet et nous avons donné quelques idées sur la manière de répondre à cette question dans [31], où le lecteur intéressé par ce sujet trouvera plus d'informations. Notre méthodologie est basée sur celle de UML Components [36] (voir la section 1.2.4 page 33 pour plus de détails sur UML Components). Nous avons appliqué à nos médiums la notion de type telle qu'elle est définie dans UML Components. Cela a abouti à la définition de quelques règles formant un processus de définition de la frontière et de la responsabilité d'un médium. Dans la section 2.1.2 page 51, nous avons vu qu'il était préférable que le médium de réservation gère en interne la liste des identificateurs. De manière intuitive, nous avons décidé qu'il était de la responsabilité du médium de réaliser cette tâche. Cela a une influence sur les interfaces des services offerts (et donc la frontière du médium), car nous devons dans ce cas avoir une interface et un service permettant de spécifier la liste des identificateurs à manipuler par le médium. Notre processus de recherche de la frontière du médium permet d'aboutir à la même conclusion mais cette fois à partir de l'application de règles et non plus de manière intuitive. Ce processus n'est encore qu'une ébauche, mais en définir une version plus formelle et plus complète pour la définition et la recherche de la responsabilité et la frontière d'un médium serait un complément très intéressant pour notre approche de réification d'abstractions de communication.

En ce qui concerne le processus de raffinement que nous avons défini dans le chapitre 3, un axe de recherche potentiel concerne la définition de patrons de distribution. En effet, lors du passage de la spécification abstraite à une spécification d'implémentation, la gestion des données dans le médium passe toujours d'une version centralisée (le médium dans sa globalité) à une version complètement distribuée (l'ensemble des gestionnaires). Dans l'exemple du raffinement du médium de réservation, il s'agit de gérer une liste d'identificateurs. Nous avons défini deux spécifications d'implémentation : une première où les identificateurs sont gérés par un seul gestionnaire et une deuxième où ils sont entièrement partagés entre tous les gestionnaires de réserveur du médium. La spécification des services offerts et des autres éléments du médium doit être modifiée en conséquence, en fonction du choix fait. Il est fort probable que ces deux choix (ainsi que bien d'autres) de distribution de données se retrouvent dans beaucoup de médiums. Il serait donc intéressant de définir des patrons de distribution qui faciliteraient le raffinement en proposant des variantes de distribution des données et qui aideraient de manière plus ou moins automatique à la modification des diagrammes et des expressions OCL spécifiant le médium. Ces patrons seraient alors des transformations ou des patrons de transformations de modèles tels que l'approche MDA les définit. Pour terminer à propos du processus de raffinement, un autre complément intéressant serait, à partir d'une spécification d'implémentation, de générer le code ou une partie du code correspondant et ayant pour cible notre plate-forme.

Bibliographie

[1] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and Friends. *IEEE Computer*, 19 (8), 1986.

[2] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1977.

[3] S. Allamaraju, K. Avedal, R. Browett, J. Diamond, J. Griffin, M. Holden, A. Hoskinson, R. Johnson, T. Karsjens, L. Kim, A. Longshaw, T. Myers, A. Nakimovsky, D. O'Connor, S. Tyagi, G. Van Damme, G. Van Huizen, M. Wilcos, and S. Zeiger. *Programmation avec Java 2 Enterprise Edition : Conteneurs J2EE, servlets, JSP, EJB*. Eyrolles, 2000.

[4] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. CMU Technical Report CMU-CS-97-144.

[5] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[6] Egil P. Andersen and Trygve Reenskaug. System Design by Composing Structures of Interacting Objects. In Ole Lehrmann Madsen, editor, *ECOOP'92, European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 133–152. Springer-Verlag, 1992.

[7] Luís Filipe Andrade and José Luiz Fiadeiro. Interconnecting Objects via Contracts. In Robert France, Bernhard Rumpe, editor, *UML '99 (The Second International Conference on The Unified Modeling Language)*, volume 1723 of *LNCS*. Springer-Verlag, 1999.

[8] Luís Filipe Andrade and José Luiz Fiadeiro. Contracts: Supporting Architecture-Based Evolution. `http://www.fiadeiro.org/jose/papers/contracts.pdf`.

[9] F. Arbab. Manifold Version 2: Language Reference Manual. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1996. `http://www.cwi.nl/ftp/manifold/refman.ps.Z`.

[10] F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, 5(1):23–70, February 1993.

[11] F. Arbab and E. Rutten. Manifold: a Programming Model for Massive Parallelism. In *the IEEE Working Conference on Massively Parallel Programming Models*, 1993.

[12] Farhad Arbab. Coordination and its Relevance. In H.R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*. CSREA Press, 1999.

[13] M. R. Barbacci, C. B. Weinstock, D. L. Doubleday, M. J. Gardner, and R. W. Lichota. Durra: A Structure Description Language for Developing Distributed Applications. *IEE Software Engineering Journal*, 8(2):83–94, March 1996.

[14] Mario R. Barbacci and Jeannette M. Wing. A Language for Distributed Applications. In *International Conference on Computer Languages (ICCL'90)*, pages 59–68. IEEE Press, 1990.

[15] David Bellin and Susan Suchman Simone. *The CRC Card Book*. Addison Wesley, 1997.

[16] L. Bellissard, S. B. Atallah, A. Kerbrat, and M. Riveill. Component-Based Programming and Application Management with Olan. In *Proceedings of the Workshop on Object-Based Parallel and Distributed Computation (OBPDC '95)*, volume 1107 of *LNCS*. Springer Verlag, 1995.

[17] L. Bellissard, F. Boyer, and M.Riveill. Construction and Management of Cooperative Distributed Applications. In *International Workshop on Object Orientation in Operating Systems (IWOOOS'95)*, pages 149–152. IEEE, 1995.

[18] Luc Bellissard. *Construction et Configuration d'Application Reparties*. Ph.D. thesis, Institut National Polytechnique de Grenoble, 1997.

[19] L. Berger, A.-M. Pinna-Dery, and M. Blay-Fornarino. Interactions between Objects : an Aspect of Object-Oriented Languages. In *ICSE'98, Workshop on Aspect-Oriented Programming*, 1998.

[20] Laurent Berger. *Mise en Œuvre des Interactions en Environnements Distribués, Compilés et Fortement Typés : le Modèle MICADO*. PhD thesis, Université de Nice-Sophia Antipolis, 2000.

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

[22] Antoine Beugnard and Robert Ogor. Encapsulation of Protocols and Services in Medium Components to build Distributed Applications. In *Engineering Distributed Objects (EDO '99), ICSE 99 Workshop*, 1999.

[23] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16(4), November 1998.

[24] G. Booch. *Object-Oriented Analysis and Design with Applications, Second Edition*. Benjamin/Cummings, 1993.

[25] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1998.

[26] E. Bruneton, T.Coupaye, and J.B. Stefani. The Fractal Composition Framework. `http://www.objectweb.org/fractal/current/Fractal1.0-0.pdf`, 2002.

[27] Franck Buschmann, Regine Meunier, Hans Rohnert, Peter Somerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons Ltd., 1996.

[28] Jean Bézivin. From Object Composition to Model Transformation with the MDA. In *TOOLS'USA*, volume TOOLS-39, August 2001.

[29] Eric Cariou. Spécification de Composants de Communication en UML. In *Objets, Composants, Modèles (OCM'2000), Nantes, France*, May 2000.

[30] Eric Cariou and Antoine Beugnard. Specification of Communication Components in UML. In H.R. 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.

[31] Eric Cariou and Antoine Beugnard. The Specification of UML Collaborations as Interaction Components. In J.M. Jézéquel, H. Hussmann, S. Cook, editor, *«UML» 2002 – The Unified Modeling Language*, volume 2640 of *LNCS*, pages 352–367. Springer Verlag, 2002.

[32] Eric Cariou, Antoine Beugnard, and Jean-Marc Jézéquel. An Architecture and a Process for Implementing Distributed Collaborations. In *The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2002)*. IEEE Computer Society, 2002.

[33] Nicholas Carriero and David Gelernter. Linda in Context. *Communication of the ACM*, 35 (2):444–458, 1989.

[34] Nicholas Carriero and David Gelernter. Coordination Languages and their Significance. *Communication of the ACM*, 35 (2), 1992.

[35] Nicholas Carriero, David Gelernter, and L. Zuck. Bauhaus Linda. In O. Nierstrasz P. Ciancarini and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 66–76. Springer Verlag, 1994.

[36] J. Cheesman and J. Daniels. *UML Components - A Simple Process for Specifying Component-Based Software.* Addison-Wesley, 2000.

[37] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design.* Addison-Wesley, Reading, MA, 1995.

[38] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Failure Detection Lower Bounds on Registers and Consensus. In *DISC: International Symposium on Distributed Computing*, volume 2508 of *LNCS*, pages 237–251. Springer Verlag, 2002.

[39] Troy Bryan Downing. *Java RMI: Remote Method Invocation.* IDG Books, 1998.

[40] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML - The Catalysis Approach.* Object Technology Series. Addison Wesley, 1998.

[41] Stéphane Ducasse and Manuel Günter. Coordination of Active Objects by Means of Explicit Connectors. In *Proceedings of the DEXA workshops*, pages 572–577. IEEE Computer Society Press, 1998.

[42] Wolfgang Emmerich. *Engineering Distributed Objects.* John Wiley & Sons, LTD, 2000.

[43] Jacques Ferber. *Les Systèmes Multi-Agents.* InterEditions, 1995.

[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[45] Benoît Garbinato. *Protocol Objects & Patterns for Structuring Reliable Distributed Systems.* PhD thesis, Swiss Federal Institute of Technology, Lausanne (EPFL), 1998.

[46] Benoît Garbinato and Rachid Guerraoui. Flexible Protocol Composition in Bast. In *the 18th International Conference on Distributed Computing Systems ICDCS-18*, 1998.

[47] David Garlan. An Introduction to the Aesop System. `http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps`, 1995.

[48] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994.

[49] Jean-Marc Geib, Christophe Gransart, and Philippe Merle. *CORBA : des Concepts à la Pratique.* Dunod, seconde edition, 1999.

[50] Alain Le Guennec. *Génie Logiciel et Méthodes Formelles avec UML : Spécification, Validation et Génération de Tests.* PhD thesis, école doctorale MATISSE, Université de Rennes 1, 2001.

[51] Guide to the Rapide 1.0 Language Reference Manuals. Rapide design team. Program Analysis and Verification Group, Computer Systems Lab, Standford University, 1997.

[52] Manuel Günter. Explicit Connectors for Coordination of Active Objects. Master thesis, Faculty of Science, University of Bern, 1998.

[53] Mark Hayden. *The Ensemble System.* PhD thesis, Cornell University, 1998.

[54] Matti A. Hiltunen and Richard D. Schlichting. An Approach to Constructing Modular Fault-Tolerant Protocols. In *the 12th IEEE Symposium on Reliable Distributed System*, pages 105–114, 1993.

[55] C.A.R Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985.

[56] Michel Hurfin and Michel Raynal. A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector. *Distributed Computing*, 12(4):209–223, 1999.

[57] Ivar Jacobson, Magnus Christenson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Publishing Company, 1992.

[58] Alan C. Kay. The Early History of Smalltalk. In *The second ACM SIGPLAN conference on History of programming languages*, pages 69–95. ACM Press, 1993.

[59] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, 1997.

[60] T. Kilmann. Designing a Coordination Model for Open Systems. In P. Ciancarini and O. Nierstrasz, editors, *First International Conference on Coordination Models, Languages and Application (Coordination '96)*, volume 1061 of *LNCS*, pages 267–284. Springer Verlag, 1996.

[61] J. Kramer, J. Magee, and A. Finkelstein. A Constructive Approach to the Design of Distributed Systems. In *the 10th International Conference on Distributed Computing Systems (ICDCS'90)*, pages 580–597. IEEE Press, 1990.

[62] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise Modeling of Design Patterns. In *UML'2000*, volume 1939 of *Lecture Notes in Computer Science*, pages 482–496. Springer Verlag, 2000.

[63] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[64] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[65] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architecture. In *the Fifth European Software Engineering Conference (ESEC'95)*, 1995.

[66] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.

[67] J. Magee, J. Kramer, and M. Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[68] Raphaël Marvie, Philippe Merle, Jean-Marc Geib, and Mathieu Vadet. OpenCCM: une Plate-forme pour Composants CORBA. In *Seconde Conférence Française sur les Systèmes d'Exploitation (CFSE-2)*, 2001.

[69] Raphaël Marvie and Marie-Claude Pellegrini. Modèles de Composants, un État de l'Art. *Coopération dans les Systèmes à Objets, numéro spécial de L'Objet*, 8(3), September 2002.

[70] P. Mathieu, JC. Routier, and Y. Secq. RIO : Roles, Interactions and Organizations. In M. Pchouek V. Maik, J. Müller, editor, *Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003*, volume 2691 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 2003.

[71] N. Medvidovic and R. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, 144(5-6):237–248, 1997.

[72] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21(6) of *ACM Software Engineering Notes*, pages 24–32. ACM Press, 1996.

[73] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 44–53. ACM Press, 1999.

[74] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *Software Engineering*, 26(1):70–93, 2000.

[75] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. In *22nd International Conference on Software Engineering (ICSE 2000)*, 2000.

[76] B. Meyer. Applying "Design by Contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, October 1992.

[77] B. Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice-Hall, 1997.

[78] Bertrand Meyer. .NET is Coming. *IEEE Computer*, 34(8), August 2001.

[79] Microsoft. La Plate-forme .Net. `http://www.microsoft.com/net/`, 2003.

[80] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[81] Pierre-Alain Muller. *Modélisation Objet avec UML*. Eyrolles, 1997.

[82] G. M. P. O'Hare and N. R. Jennings, editors. *Foundations of Distributed Artificial Intelligence*. John Wiley and Sons, New York, 1997.

[83] OMG. OMG Unified Modeling Language Specification, version 1.3. `http://www.omg.org/cgi-bin/doc?ad/99-06-08`, June 1999.

[84] OMG. OMG Unified Modeling Language Specification, version 1.4. `http://www.omg.org/cgi-bin/doc?formal/01-09-67`, 2001.

[85] OMG. CORBA Component Model. `http://www.omg.org/technology/documents/formal/components.htm`, 2002.

[86] OMG. CORBA/IIOP Specification, version 3.02. `http://www.omg.org/technology/documents/formal/corba_iiop.htm`, 2002.

[87] OMG. Meta Object Facilities Specification, version 1.4. `http://www.omg.org/technology/documents/formal/mof.htm`, 2002.

[88] OMG. Model Driven Architecture. `http://www.omg.org/mda`, December 2002.

[89] OMG. Model Driven Architecture FAQ. `http://www.omg.org/mda/faq_mda.htm`, December 2002.

[90] OMG. XML Metadata Interchange, version 1.2. `http://www.omg.org/technology/documents/formal/xmi.htm`, 2002.

[91] OpenCCM: the Open CORBA Component Model Platform. `http://www.objectweb.org/openccm/`.

[92] George A. Papadopoulos and Farhad Arbab. Coordination Models and Languages. *Advances in Computers, Academic Press*, 46, 1998.

[93] Renaud Pawlak. *La Programmation Orientée Aspect Interactionnelle pour la Construction d'Applications à Préoccupations Multiples*. PhD thesis, Conservatoire National des Arts et Métiers, 2002.

[94] S. Pickin, C. Jard, Y. Le Traon, T. Jéron, J.-M. Jézéquel, and A. Le Guennec. System Test Synthesis from UML Models of Distributed Software. In D. Peled and M. Vardi, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2002*, volume 2460 of *LNCS*, 2002.

[95] Projet RNTL ACCORD. État de l'Art sur la Démarche MDA. `http://www.infres.enst.fr/projets/accord/lot1/lot_1.1-5.pdf`, 2002.

[96] Projet RNTL ACCORD. État de l'Art sur Les langages de Coordination. `http://www.infres.enst.fr/projets/accord/lot1/lot_1.1-3.pdf`, 2002.

[97] Projet RNTL ACCORD. État de l'Art sur les Langages de Description d'Architecture (ADLs). `http://www.infres.enst.fr/projets/accord/lot1/lot_1.1-2.pdf`, 2002.

[98] Michel Raynal. *Algorithmes Distribués et Protocoles.* Eyrolles, 1985.

[99] Michel Raynal. *Gestion des Données Réparties : Problèmes et Protocoles.* Eyrolles, 1992.

[100] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OORam Software Engineering Method.* Prentice-Hall, 1995.

[101] *Le Petit Robert, Dictionnaire de la Langue Française.* Dictionnaires le Robert - Paris, 2002.

[102] A. Rowstron and A. Wood. Bonita: a Set of Tuple Space Primitives for Distributed Coordination. In *30th Hawaii International Conference on Systems Sciences (HICSS-30)*, pages 379–388. IEEE Press, 1997.

[103] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley, 1997.

[104] James Rumbaugh, W. Lorenson, and M. Blaha. *Object-Oriented Modeling and Design.* Prentice-Hall, 1990.

[105] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2 – Networked and Concurrent Objects.* John Wiley and Sons, 2000.

[106] Mary Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *ICSE Workshop on Studies of Software Design*, volume 1078 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.

[107] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Software Engineering*, 21(4):314–335, 1995.

[108] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and Implementations for Architectural Connections. In *the 3rd International Conference on Configurable Distributed Systems.* IEEE Press, 1996.

[109] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, April 1996.

[110] Jon Siegel. *CORBA: Fundamentals and Programming.* John Wiley & Sons Inc., 1996.

[111] I. Sommerville and G. Dean. PCL: A Language for Modelling Evolving System Architectures. *Software Engineering Journal, IEE*, March 1996.

[112] Nigel Stanley. *Microsoft .NET, Jumpstart for Systems Administrators and Developers.* Digital Press, 2003.

[113] Sun Microsystems. Java Remote Method Invocation. `http://java.sun.com/products/jdk/rmi/`.

[114] Sun Microsystems. Entreprise Java Beans Technology. `http://java.sun.com/products/ejb/`, 2002.

[115] G. Sunyé, A. Le Guennec, and J.-M. Jézéquel. Design Pattern Application in UML. In E. Bertino, editor, *ECOOP'2000*, volume 1850 of *Lecture Notes in Computer Science*, pages 44–62. Springer Verlag, 2000.

[116] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley ACM Press, 1999.

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

[118] M. Woolridge and N. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[119] G. Zelesnik. *The Unicon Language Reference Manual.* School of Computer Science Carnegie Mellon University, 1996.

[120] G. Zelesnik. *The Unicon Language User Manual.* School of Computer Science Carnegie Mellon University, 1996.

Résumé

Lors de la conception et du développement d'applications distribuées, la communication et les interactions entre les composants répartis représentent un point crucial. La spécification d'abstractions d'interaction entre ces composants est donc un élément primordial lors de la définition de l'architecture d'une application. Si bien souvent des abstractions d'interaction de haut niveau sont définies au niveau de la spécification, à celui de l'implémentation il est par contre plus courant de ne trouver que des abstractions bien plus simples (comme des appels de procédure à distance ou de la diffusion d'événements). Pendant le raffinement qui mène à l'implémentation, ces abstractions de haut niveau ont été diluées, dispersées à travers les spécifications et implémentations des composants de l'application.

Nous proposons un processus de réification d'abstractions de communication ou d'interaction sous forme de composants logiciels. Ce processus permet de conserver l'unité et la cohérence d'une abstraction d'interaction pendant tout le cycle de développement d'une application. Que l'on soit à un niveau de conception abstraite, de conception d'implémentation, d'implémentation ou de déploiement, l'abstraction réifiée et manipulée est toujours la même. Ces composants logiciels sont aussi appelés médiums pour les différencier des autres composants d'une application.

Le processus est composé de plusieurs éléments. Le premier est une méthodologie de spécification de médiums en UML. Cette spécification est réalisée à un niveau abstrait, indépendamment de toute implémentation. Dans la terminologie du MDA (Model-Driven Architecture) de l'OMG, il s'agit d'une spécification de niveau PIM (Platform Independent Model). Cette spécification est le contrat du médium qui doit ensuite être respecté quel que soit le niveau de manipulation considéré. Le deuxième élément est une architecture de déploiement de médiums qui offre la flexibilité nécessaire pour implémenter a priori n'importe quelle abstraction d'interaction. Enfin, le troisième élément est un processus de raffinement qui permet de transformer une spécification abstraite de médium en une ou plusieurs spécifications d'implémentation prenant en compte l'architecture de déploiement et différents choix de conception ou d'implémentation. Dans la terminologie du MDA, le processus de raffinement sert à transformer une spécification de niveau PIM en une ou plusieurs spécifications de niveau PSM (Platform Specific Model).

Mots-clés : architecture et composants logiciels, modélisation, abstractions de communication, collaborations UML, processus de raffinement.