HAL
open science

Contributions à l'étude de l'assemblage de logiciels

Antoine Beugnard

To cite this version:

Antoine Beugnard. Contributions à l'étude de l'assemblage de logiciels. Interface homme-machine [cs.HC]. IMT Atlantique, 2005. tel-02461971

HAL Id: tel-02461971

https://hal.science/tel-02461971v1

Submitted on 31 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Habilitation à Diriger des Recherches

présentée devant

L'université de Bretagne Sud
en sciences et technologie de l'information et de la communication

par

Antoine Beugnard

Contributions à l'étude de l'assemblage de logiciels

soutenue le 5 décembre 2005 devant le jury composé de :

Mme	Isabelle Borne	Présidente
MM	Guy Bernard	Rapporteurs
	Oscar Nierstrasz	
	Michel Riveill	
MM	Jean-Marc Geib	Examinateurs
	Yvon Kermarrec	

Remerciements

L'activité de « chercheur » est essentiellement faite de discussions. C'est en parlant, en échangeant avec de nombreux interlocuteurs, industriels, chercheurs, enseignants, étudiants qu'apparaissent des questions. Car avant tout, l'activité de recherche, c'est poser les bonnes questions ; trouver les réponses s'ensuit ...parfois. Les meilleures questions viennent parfois de regards étrangers au spécialiste, qui obnubilé par sa propre vision du monde passe parfois à côté de questions essentielles, d'hypothèses cachées - car tellement « naturelles » -. Mes travaux de recherches se sont donc alimentés de toutes ces discussions, de tous ces échanges, de la nécessité d'expliquer, de ré-expliquer, de changer de point de vue. Aussi je tiens à remercier tous les contributeurs anonymes qui m'ont permis de me poser des questions.

Je souhaite également remercier les personnalités scientifiques qui ont accepté de faire partie de mon jury : Isabelle Borne, Guy Bernard, Jean-Marc Geib, Yvon Kermarrec, Oscar Nierstrasz et Michel Riveill.

Je voudrais aussi remercier mes collègues du département informatique de l'ENST Bretagne avec lesquels j'ai, depuis des années, des discussions fructueuses et qui m'ont permis de développer ces travaux de recherche. Ces travaux doivent beaucoup à Olivier Aubert, Eric Cariou, Selma Matougui et Eveline Kaboré qui leur ont consacré quelques années à réfléchir à d'étranges questions ...

Préambule

Enseignant-Chercheur à l'ENST Bretagne depuis 1992, mon domaine de recherche a, depuis ma thèse en 1993, été consacré à la problématique de la modélisation et de la spécification, en s'appuyant sur les technologies objets. Ayant eu la responsabilité de l'enseignement informatique à l'ENST Bretagne pendant quelques années, mon activité de recherche a été mise en veille autour des années 1995. De retour à des activités de recherche, vers 1997, j'ai participé, sous la direction de Robert Rannou, puis d'Annie Gravey à la mise en place de projets de recherche au sein du département informatique de l'ENST Bretagne.

J'anime actuellement la recherche du département et suis co-responsable avec Fabien Dagnat du projet CAMA (Composants pour Architectures Mobiles Adaptables) où mes recherches se focalisent sur l'assemblage des composants logiciels.

Après un bref premier chapitre d'introduction, la première partie de ce rapport - les chapitres 2 et 3 - présente une synthèse de mes travaux de recherche, la seconde - chapitre 4 - décrit les axes de recherches que j'envisage d'approfondir.

Chapitre 1

Introduction

L'informatique évolue grâce à plusieurs moteurs. Le premier est sans conteste celui de la recherche de performance qui s'appuie sur le matériel, l'algorithmique et des traducteurs/optimiseurs qui exploitent au maximum l'architecture matérielle. Un deuxième est la recherche d'abstractions ; quelles entités inventer pour exprimer et manipuler plus facilement et plus sûrement programmes et données ? Ces deux aspects ne sont pas indépendants puisque, pour être utiles, les abstractions doivent disposer d'outils de mise en œuvre qui se traduisent par des traducteurs, des transformateurs ou des compilateurs[1].

Nos travaux ont concerné ce second moteur, selon deux axes complémentaires : les langages et les méthodes de programmation. Les langages ont évolué dans leur structure interne, mais aussi dans l'organisation de leur source. Les méthodes ont sans doute moins subi de révolutions internes, mais ont été fortement influencées par les abstractions provenant des travaux sur les langages.

Voici en 4 points un bref résumé de la chronologie de nos recherches :

1. Nos travaux commencent par une thèse [9] sur la méthode de développement JSD [38] (Jackson System Development). Nous proposons pour l'analyse d'un problème de manipuler un temps abstrait idéal, comme les langages synchrones (Esterel, Signal, ...) le proposent lors de l'implémentation. Cette abstraction temporelle se traduit par différentes stratégies de reconstruction du temps lors de la réalisation [8]. Nous proposons également d'organiser les spécifications d'un système en deux parties, l'une sémantique basée sur une description opérationnelle du système, l'autre organisationnelle pour présenter plusieurs points de vue sur le système. On peut également noter que l'approche JSD introduit deux « types de communication » (canal et inspecteur) qui ont influencé la suite de nos travaux sur les abstractions de communication (chapitre 3)...
2. Une spécification décrit des propriétés d'un système dont on peut regrouper certaines sous forme de contrat. Dans le prolongement des propositions de B. Meyer, et dans un but d'organisation des propriétés, nous avons proposé une classification des contrats en quatre niveaux [15].
3. Dans le cadre de la modélisation des systèmes complexes, nous avons étudié la notion d'abstraction de communication. Sans être absolument nouvelle, puisqu'elle apparaît comme une abstraction de certains ADL, voire de certains langages, nous avons proposé une nouvelle

[1] Ses trois mots désignent la même idée ; le *passage* d'une représentation (abstraite) à une autre (en général, mais pas forcément, plus concrète).

classification de ces abstractions et ce qui nous a conduit à proposer un nouveau type de composant [21] et une façon originale de voir l'assemblage de composant [42].

4. En complément, nous avons étudié la sémantique de la redéfinition des méthodes des langages objet. En effet, la plupart des outils de modélisation actuels reposent sur UML qui utilise le paradigme objet pour décrire des spécifications. Or, bien que s'affichant comme un langage de modélisation unifié, UML ne propose pas de sémantique de la redéfinition, ce qui pose de nombreux problèmes potentiels comme nous l'avons montré [11, 12, 22, 13].

Nous détaillons dans ce mémoire les trois derniers points. Concernant les abstractions de communication, nous explorons les mécanismes qui permettent de connecter des entités logicielles à l'aide d'autres entités logicielles en cherchant des abstractions de haut niveau. Ces abstractions requièrent pour réaliser l'assemblage de l'information décrivant les entités à relier. La notion de contrat permet de structurer et classifier toute cette information sous une forme abstraite. Enfin, nous approfondissons un cas particulier d'information présente dans les contrats, qui concerne l'étude des différentes interprétations de la redéfinition et de la surcharge des méthodes dans les langages à objets et constitue, à notre avis, un point particulièrement délicat à traiter pour garantir un assemblage correct de logiciels.

Le fil directeur de nos recherches est donc *l'étude de l'assemblage des entités logicielles* avec la volonté de trouver les bonnes abstractions qui permettent de réaliser ces assemblages et de réutiliser les nombreuses techniques déjà existantes de vérification et de validation.

Assembler du logiciel est un vieux problème ; il est naturellement complémentaire de l'approche de décomposition. Pour résoudre un problème, on le divise, on le résout par partie, puis « il ne reste plus » qu'à assembler les parties ! Aujourd'hui, l'assemblage devient une nécessité liée à la réutilisation des programmes déjà réalisés, souvent par une tierce partie. Les programmes doivent pouvoir être assemblés pour être réutilisés et ainsi faire évoluer le service global.

Les techniques d'édition de liens ont été les premières à être développées. Leur but est d'assembler des binaires pour créer un exécutable. Pour que cet exécutable soit correct, de nombreuses recherches ont été effectuées qui ont abouti aux théories des types, aux systèmes de preuves, aux techniques de *model checking*.

Toutes ces techniques sont fondamentales et ont permis une compréhension des mécanismes d'assemblage de programmes. Toutefois deux problèmes cruciaux demeurent :

1. le passage à l'échelle et la capacité de « montrer » la correction d'un assemblage de grande taille et,
2. la gestion de l'évolution des programmes ou des parties de programme.

Nos travaux s'intéressent fondamentalement à ce problème d'assemblage et tendent de répondre à deux questions :

- Quelle information est nécessaire pour assembler un composant/module/unité avec une autre ?
- Quels mécanismes doivent être mis en oeuvre pour valider/vérifier/construire l'assemblage ?

Le document est organisé de la façon suivante. Le chapitre 2 présente une synthèse de trois publications [15, 12, 13] qui concernent l'expression des interfaces des entités logicielles et le problème de la redéfinition de méthode. Le chapitre 3 réalise à partir des travaux de thèse de E. Cariou [20] et S. Matougui [41] une synthèse des études des abstractions de communication en présentant la notion de composant de communication et celle de connecteur. Enfin, dans la conclusion, nous développons des pistes de recherche qui embrassent l'ensemble de ces thèmes.

Chapitre 2

Les contrats d'assemblage

Comme nous l'avons vu en introduction, pour réaliser un assemblage de logiciels, la première question à se poser est : *Quelles informations doivent être associées à une entité pour qu'un outil d'assemblage puisse être utilisé et garantir un assemblage correct ?*

Pour répondre à cette question, il est tout d'abord nécessaire de définir ce qu'on entend par « correct ». En effet, en fonction du niveau de confiance qu'on veut obtenir sur le résultat de l'assemblage, les outils et techniques seront plus ou moins complexes. Voici quelques exemples de niveau de confiance ou de correction (les noms sont des suggestions, il n'existe à ma connaissance pas de classification) :

- compilable - tous les objets (données, méthodes) sont définis, les types compatibles.
- exécutable - il existe une exécution du résultat de la compilation qui s'exécute sans erreur inattendue (exception, interblocage, non respect de la spécification).
- sûr - toutes les exécutions du résultat de la compilation s'exécutent sans erreur inattendue.
- correct - sûr et conforme aux exigences non-fonctionnelles telles que performance, fiabilité, etc.

Cette liste n'a de but que de montrer que les objectifs d'assemblage peuvent être nombreux.

Ensuite, on peut se demander comment l'information est obtenue ? L'entité la fournit-elle explicitement ? L'outil d'assemblage est-il capable d'extraire par lui-même les informations dont il a besoin sans que celle-ci ne soit explicitement décrite par l'entité ? On peut se dire qu'idéalement la seconde possibilité est préférable puisqu'elle ne demande pas une explicitation de l'information à fournir, ce qui rend l'entité à assembler indépendante de l'outil d'assemblage qui sera utilisé. C'est à ce dernier d'aller chercher dans l'entité (binaire, source) ce dont il a besoin. Si cette approche est attirante, elle nous paraît peu réaliste. En effet, le résultat de la conception d'une entité est le résultat de nombreux choix de conceptions, de choix d'algorithmes et de représentation de données qui cachent des intentions pas toujours explicites. Alors, faute de conserver la trace de l'ensemble de décisions prises par le concepteur, les entités perdent une partie de l'information qui pourrait être nécessaire à un assemblage « correct ». Un autre argument allant dans le sens d'une explicitation de l'information d'assemblage vient d'une observation des états d'une entité. Une limite importante des outils de vérification (de type *model checker*) est l'explosion combinatoire des états. Pour résoudre

ce problème, les recherches actuelles élaborent des stratégies qui abstraient des états. Le choix de ces abstractions est complexe et son automatisation est encore difficilement envisageable.

Dans la suite de ce chapitre, nous allons présenter comment nous avons proposé d'organiser l'information d'assemblage sous la forme d'un *contrat*. Puis, nous verrons sur un point précis, comment la diversité des sémantiques de la redéfinition et de la surcharge de méthode dans les langages à objets, rend l'assemblage complexe. Pour résoudre ce problème, nous pensons que, à l'instar de la (quasi-)disparition du « *goto* », il serait nécessaire de faire des choix dans les sémantiques de la redéfinition et de la surcharge pour rendre l'assemblage plus sûr.

La partie consacrée aux contrats est basée sur les publications [15, 14], celles sur les conséquences de la redéfinition de méthode sur [12, 11, 22, 13]

2.1 La notion de contrat

Le terme de conception par contrat a été introduit pas B. Meyer en 1986 pour décrire de manière symétrique les relations entre client et serveur lors de leurs interactions. Cette approche est une application pratique des travaux de Hoare sur la spécification de fonctions à l'aide de préconditions et de postcondition. Les premières applications des transformateurs de prédicats de Hoare étaient :

- La dérivation de code qui permet de déduire un programme, une séquence d'instructions, à partir de la connaissance d'une postcondition,
- et la preuve de programme qui permet de vérifier qu'une séquence d'instructions satisfait une postcondition.

Ces approches théoriques n'ont guère diffusé le monde de l'industrie avant l'approche proposée par B. Meyer. Intégrées au langage Eiffel [47] les prédicats `require` (précondition) et `ensure` (postcondition) permettent de décrire un contrat entre l'objet client d'une fonction et l'objet qui implante cette fonction. La table 2.1 illustre la symétrie de la relation entre le client et le serveur. Le développement de la notion de composant logiciel a offert une nouvelle opportunité d'appliquer cette vision contractuelle.

	Client	Serveur
Obligation	satisfaire la précondition	satisfaire la postcondition
Gain	le résultat est correct	l'état initial du service est correct

Tab. 2.1 – Contrat : Gain et devoirs des deux parties

Cette approche est une application pratique des travaux théorique de Hoare sur les transformations de prédicat (weakest precondition - wp) qui s'appliquent aux programmes séquentiels. Autrement dit, une hypothèse implicite se cache derrière l'approche par contrat de B. Meyer : les opération s'exécutent de manière atomique. Partant de ce constat nous avons proposé avec J.-M. Jézéquel, N. Plouzeau et D. Watkins une classification des contrats selon 4 niveaux [15] :

Le niveau syntaxique où l'on décrit les propriétés de compatibilité du client et du serveur d'une opération en terme de détectable par un compilateur (noms, nombre et type des arguments compatibles.)

Le niveau sémantique où l'on décrit l'intention de l'opération en s'appuyant sur des préconditions et des postconditions.

Le niveau synchronisation où l'on s'intéresse à l'entrelacement des opérations, non plus considérées comme indépendantes les unes des autres. La concurrence d'exécution, les dépendances temporelles entre les opérations, sont ici décrites.

Le niveau quantitatif (qualité de service) où l'on décrit les propriétés mesurables souhaitées ou offertes par une opération. On précisera les temps de réponse, la gigue, les durée de reprise en cas d'erreur, etc.

Comme toute classification, il s'agit essentiellement là d'un choix de point de vue. La frontière est parfois fine entre deux niveaux. Par exemple, le type des arguments considérés dans le niveau syntaxique (1) des contrats pourrait, selon un point de vue tout à fait défendable, apparaître dans le deuxième niveau. Le choix de le placer au premier niveau est lié à l'outil capable de détecter une rupture de contrat, ici, le compilateur, le second niveau étant réservé à l'expression de l'intention de l'opération.

Cette généralisation de la notion de contrat a donné lieu à de multiples travaux. Dans le cadre du projet RNTL ACCORD [71] auquel nous avons participé, la notion de contrat est réifiée et utilisée pour exprimer les différentes offres de services ainsi que les besoins de service. La confrontation des contrats devrait pouvoir ensuite être étudiée en fonction de la nature de la relation qui relie l'offre et la demande :

Client-Serveur L'assemblage classique entre un composant qui offre une opération et un autre qui l'utilise.

Substituabilité Un composant peut être remplacé par un composant qui offre les mêmes propriétés, ou des propriétés compatibles.

Raffinement Un composant peut être remplacé par un autre qui le raffine (les mêmes propriétés plus d'autres, ou moins certaines).

Il est intéressant de noter que les règles de compatibilité de contrat sont différentes selon la nature de la relation. Pour la compatibilité des opérations au niveau syntaxique, des propriétés sont déjà mises en évidences comme celles relatives à l'invariance, la covariance ou la contravariance des redéfinitions. Toutefois leur domaine d'application n'est pas encore totalement élucidé. On sait ce que sont ces notions, mais il n'y a pas de consensus sur la façon de les utiliser. Nous en reparlerons dans la partie 2.2 page 10.

Le projet RNTL ACCORD s'appuyait sur les travaux de l'OMG qui ont abouti à la définition d'UML2.0 (Unified Language Modelling). Les outils de modélisation basés sur ce langage permettent de décrire des spécifications de manière semi-formelle. En effet, Malgré des efforts de formalisation qui tendent à rendre plus rigoureuse cette approche, la souplesse des interprétations (dites « variations sémantiques ») permet un large spectre d'applications industrielles et garantit de nombreux utilisateurs potentiels. Aussi, pour compléter les capacités de description d'UML on peut naturellement penser enrichir le méta-modèle avec le concept de contrat. C'est cette approche qui a été développée dans ACCORD.

À chaque élément d'un modèle on peut associer un contrat qui décrit les propriétés de cet élément indépendamment du contexte de son utilisation. Afin de structurer la description du contrat, les propriétés décrites peuvent être classées selon les 4 catégories proposées précédemment : Syntaxique,

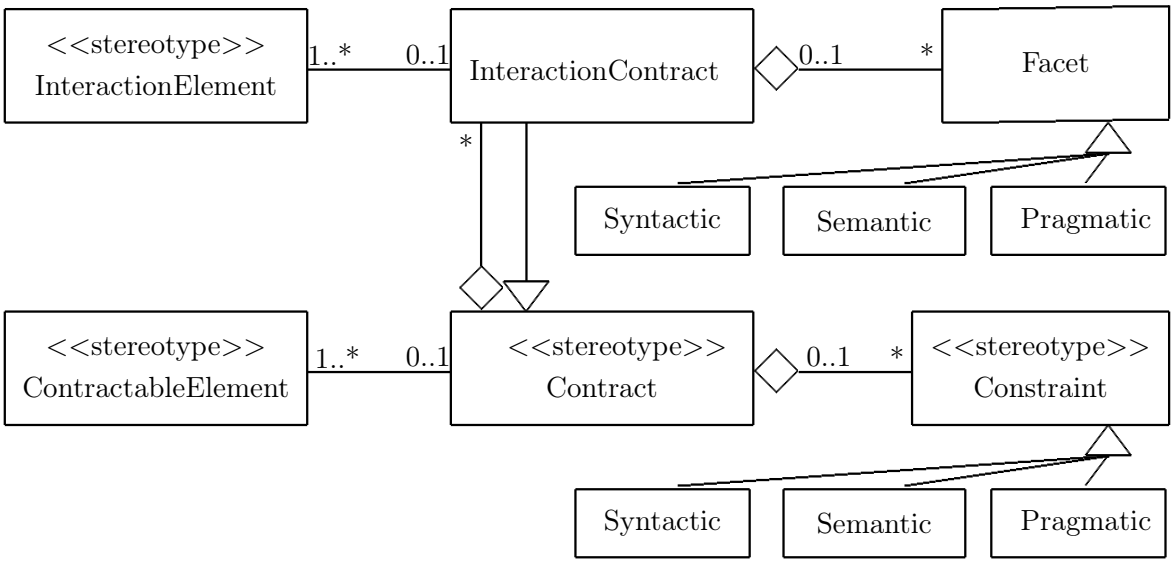


Fig. 2.1 – Méta-modèle ACCORD pour les contrats

Sémantique, Synchronisation/Concurrence, Qualité de service. Toutefois, dans le projet ACCORD, une autre classification a été adoptée, comme on peut le voir sur la figure 2.1 :

1. Syntaxique
2. Sémantique
3. Pragmatique : toutes les propriétés non liées aux *fonctions* de l'élément (temps, qualité de service, etc.)

Lors de l'utilisation d'un élément, par exemple lorsqu'il est assemblé avec, ou substitué à un autre, on confronte les contrats des éléments mis en jeu grâce à des outils adaptés. Pour les aspects syntaxiques, un compilateur vérifiera la compatibilité des types. Pour les aspects sémantiques, un « prouveur » de théorème pourra confronter les préconditions et les postconditions. Pour les aspects synchronisation, un *model checker*, par exemple, pourra exploiter les automates décrivant les interactions possibles. Pour les aspects qualité de service, les outils restent, à ma connaissance, à développer.

La confrontation des contrats conduit à l'élaboration de nouveaux contrats qui peuvent être propagés lors de l'élaboration d'un système.

Bien évidemment, l'ensemble de ces confrontations ne prouvera pas la correction absolue de l'utilisation des éléments, mais elle devrait permettre d'augmenter la confiance dans la construction du système.

Nous allons voir dans les parties suivantes à quel point, sur un détail qui peut sembler anecdotique, la complexité de l'interprétation des contrats est grande. Complexité qu'il faudra pourtant bien réduire si l'on souhaite systématiser l'assemblage de logiciels.

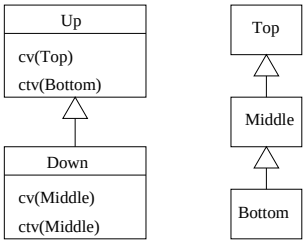


FIG. 2.2 – Comment se comportera ce modèle ?

2.2 Un cas particulier de contrat : la redéfinition de méthode

Les techniques de programmation objet sont largement disséminées dans la communauté scientifique informatique, et si des mécanismes comme l'encapsulation et l'héritage sont bien connus et compris, leur conséquence opérationnelle – la liaison dynamique – a encore beaucoup d'interprétations, faisant des langages objet une solution toujours immature, ou, du moins, une juxtaposition de nombreuses théories, présentées comme unifiées pour des raisons marchandes [39] !

Puisque nous pensons que la notion d'objet a un sens, et apporte une réelle amélioration par rapport aux langages classiques – sans liaison dynamique –, nous avons comparé sept langages objet (C++, CLOS, Dylan, Eiffel, Java, Smalltalk, OCaml) et étudié leurs différences vis-à-vis de la redéfinition et de la surcharge afin d'en proposer une sémantique « raisonnable »[1]. Nous nous sommes limités à la liaison dynamique simple, même si plusieurs langages testés (CLOS, Dylan) offrent la possibilité de réaliser la liaison dynamique sur plusieurs objets.

L'apport principal de la programmation objet est, à notre avis, la généralisation de l'accès à la liaison dynamique qui rend possible le développement de frameworks qui sont faits pour être étendus et spécialisés. Le fait que la liaison dynamique soit naturellement utilisée dans les langages objet par simple redéfinition de méthode, sans coût supplémentaire de développement (elle est intégrée dans les langages, et ne nécessite pas, comme en C, d'utiliser des pointeurs de fonction) rend les langages objet bien plus puissants que les langages classiques. Toutefois, le coût de cette liaison tardive se traduit pour le compilateur par une recherche dynamique de la méthode à appliquer (lookup), et pour l'utilisateur par la nécessité de connaître les signatures des méthodes que l'ancien code (celui du framework) s'attend à trouver. Ceci explique la courbe d'apprentissage des langages objet, défavorable comparée aux langages classiques ; pour programmer efficacement, il faut connaître les interfaces de programmation usuelles (APIs) pour les appeler, *mais surtout pour être appelé par elles.*

Quoi qu'il en soit, la liaison dynamique possède, comme nous le verrons, de nombreuses interprétations. Ceci rend les tentatives d'unification comme UML [56] pour l'analyse et la conception objet, ou .NET [51] pour l'intégration de plusieurs langages, une tâche très compliquée qui risquent de n'être cohérentes ni sémantiquement, ni opérationnellement.

Les nombreuses interprétations opérationnelles de la liaison dynamique empêchent une traduction syntaxique simple d'un langage vers un autre qui garantisse un comportement à l'exécution similaire. Par exemple, les appels aux opérations du modèle UML de la figure 2.2 après une génération de code Java ne se dérouleront pas de la même façon que pour une génération C++ ! *Alors,*

[1] Depuis la publication dans la conférence LMO [12] d'autres sémantiques raisonnables sont défendues. Aucun consensus n'étant en vue, j'invite le lecteur à consulter la page `http ://perso-info.enst-bretagne.fr/ beugnard/papiers/lb-sem.shtml` pour plus d'information.

n'y a-t-il qu'une seule théorie objet ? Et parmi celles existantes laquelle est la meilleure ? D'ailleurs, y en a-t-il une meilleure ?

Pour mettre en lumière cela, nous avons organisé cette partie comme suit. Dans la section 2.2.1 nous construisons un petit programme sans signification qui exécutera tous les appels possibles. Puis, avant toute exécution, nous présentons une sémantique « raisonnable » dans la section 2.2.2 page suivante. Nous observons et commentons en 2.2.3 page 13 les exécutions du programme écrit dans différents langages (C++, CLOS, Dylan, Eiffel, Java, Smalltalk, OCaml).

2.2.1 Description du test

Pour étudier le comportement des langages vis-à-vis de la liaison dynamique nous avons défini un petit modèle de cinq classes dont les deux principales contiennent chacune deux méthodes qui permettent d'avoir des « intentions de redéfinitions » covariantes et contravariantes. Le comportement est observé par une classe cliente qui exécute tous les appels possibles (18 au total). Afin d'éviter toute tentative d'interprétation des noms des classes et méthodes que nous allons décrire, et afin de nous concentrer sur le fonctionnement de la liaison dynamique, nous choisissons des noms sans connotation autre que mnémonique.

Nous considérons deux classes Up et Down telles que Down est une sous-classe de Up. La superclasse Up offre deux services cv et ctv. Les méthodes cv et ctv nécessitent un unique paramètre. Les paramètres sont des instances des classes Top, Middle ou Bottom où Bottom est une sous-classe de Middle qui est une sous-classe de Top (figure 2.2 page précédente). Pour tester tous les comportements covariants et contravariants nous définissons les classes Up et Down comme suit :

```
class Up
   method cv(Top t)
   method ctv(Bottom b)
class Down subclass of Up
   method cv(Middle m) - redéfinition covariante de cv
   method ctv(Middle m) - redéfinition contravariante de ctv
```

La redéfinition covariante signifie que le type de l'argument varie dans le même sens que la hiérarchie d'héritage. C'est le cas de la méthode cv puisque Down est une sous-classe de Up et Middle est une sous-classe de Top. La redéfinition contravariante signifie que le type de l'argument varie en sens opposé. C'est le cas de ctv puisque Down est une sous-classe de Up et Middle est une *super*-classe de Bottom. Une longue controverse a opposé la communauté scientifique pour décider quelle redéfinition était la bonne. Les théoriciens défendaient la contravariance puisqu'elle est sémantiquement cohérente, et *simple*. Les praticiens observent que de nombreux programmes utilisent fréquemment la covariance. Alors ? Dans [25] G. Castagna réconcilie les deux points de vue en montrant qu'ils peuvent être utilisés simultanément avec des buts différents ; la redéfinition complète ou la redéfinition partielle.

Une autre sémantique, n'utilisant ni la covariance ni la contravariance, est couramment utilisée : l'invariance. Nous aurions pu ajouter dans Up et Down le service supplémentaire inv(Middle m) qui posséderait exactement la même définition dans les deux classes. Pour simplifier l'étude nous avons écarté ce cas, car tous les langages comparés se comportent de manière identique[2].

[2]Seule leur façon de traiter les erreurs les distingue, voir les tableaux 2.11 page 18, 2.12 page 18 et 2.13 page 19.

Lorsque ni covariance ni contravariance ne sont acceptées par un langage on parle alors de surcharge ; on utilise le même nom de méthode avec des types (ou un nombre) de paramètres différents. Cette technique est critiquée par B.Meyer [48] qui affirme que si les programmeurs veulent changer la signature d'un service ils ont toujours la possibilité de changer le *nom* du service plutôt que de changer le type ou le nombre des paramètres.

procedure main
– objets récepteurs (la déclaration n'est pas nécessaire dans tous les langages)
    Up u, ud ;
    Down d ;
– arguments possibles
    Top t = new Top() ;
    Middle m = new Middle() ;
    Bottom b = new Bottom() ;

– Premier test	– Deuxième test	– Troisième test
– u est vraiment un Up	– d est vraiment un Down	– ud est *effectivement* un Down
u := new Up() ;	d := new Down() ;	ud := new Down() ; – Down → Up
u.cv(t) ;	d.cv(t) ;	ud.cv(t) ;
u.cv(m) ;	d.cv(m) ;	ud.cv(m) ;
u.cv(b) ;	d.cv(b) ;	ud.cv(b) ;
u.ctv(t) ;	d.ctv(t) ;	ud.ctv(t) ;
u.ctv(m) ;	d.ctv(m) ;	ud.ctv(m) ;
u.ctv(b) ;	d.ctv(b) ;	ud.ctv(b) ;

Tab. 2.2 – La série de trois tests

Quelque soit la position du programmeur sur la validité de la covariance, de la contravariance ou de l'invariance, les deux définitions de classe ci-dessus sont compréhensibles et se rencontrent dans des frameworks objet. Imaginons un développement où, dans une première phase, seules les classes Up, Top, Middle et Bottom sont définies et fournies dans un framework, puis dans une seconde phase, une extension Down de Up est définie. La question est « comment va se comporter le nouveau système ? »

Nous montrons que les implantations des langages objet actuels offrent de nombreuses sémantiques. Pour illustrer cela nous créons, pour chacun des langages étudié, un programme client qui instancie les objets puis essaye tous les appels possibles ; les 2 méthodes sont appelées successivement avec chacun des 3 types d'argument possibles pour les 3 récepteurs suivant : u (de type Up), d de type (Down) et ud déclaré comme Up mais effectivement un Down. Nous avons donc (2 x 3 x 3) = 18 tests. Notez que la liaison dynamique n'est effective que dans le troisième test, et uniquement sur le récepteur (ou le premier paramètre pour les langages qui utilisent une notation fonctionnelle plutôt que pointée.)

Ce code peut ne pas compiler. C'est une partie du test que d'observer quand les compilateurs refusent de compiler, puis d'observer le résultat de l'exécution. Le corps de chaque méthode cv et ctv imprime simplement la signature de la méthode effectivement exécutée.

2.2.2 Une sémantique « raisonnable »

Avant toute exécution par un langage de programmation réel, le tableau 2.3 page suivante résume une sémantique « raisonnable » en décrivant les résultats attendus. Cette sémantique accepte aussi

appels	u	d	ud
cv(t)	cv(Top) in Up	cv(Top) in Up	cv(Top) in Up
cv(m)	cv(Top) in Up	cv(Middle) in Down	cv(Middle) in Down
cv(b)	cv(Top) in Up	cv(Middle) in Down	cv(Middle) in Down
ctv(t)	Top NOT Bottom	Top NOT Middle	Top NOT Bottom
ctv(m)	Middle NOT Bottom	ctv(Middle) in Down	Middle NOT Bottom
ctv(b)	ctv(Bottom) in Up	ctv(Middle) in Down	ctv(Middle) in Down

TAB. 2.3 – Résultats attendus, i.e. une sémantique raisonnable

bien des redéfinitions de méthode covariante *et* contravariante.

Cette sémantique offre toutes les combinaisons habituellement exploitées par les langages de programmation objet, mais n'est, de manière surprenante, jamais implantée comme la prochaine section le montre. Quelques commentaires :

1. Les méthodes sont trouvées dans la classe acceptable (sans erreur de type) la plus spécialisée : dans Up pour la colonne 1, dans Down pour les colonnes 2 et 3 avec un cas particulier...
2. Un appel de la méthode redéfinie de manière covariante (cv) dans la classe Down avec un récepteur Down et un argument Top est trouvé dans Up (ligne 1, colonnes 2 and 3) puisque, dans Down, l'argument requis est trop spécialisé, mais qu'il est accepté dans Up.
3. Les colonnes 2 et 3 sont quasiment identiques. La seule différence est l'erreur qui doit être détectée par le compilateur à l'appel de `ctv(m)` – case (5,3)[3] – puisque nous imaginons que le programmeur s'attend à trouver des services déclarés dans la classe Up, même s'il sait que des objets plus spécialisés pourraient être effectivement passés en argument. Si l'on ne déclenche pas une erreur dans ce cas, cela signifie que la classe Up devra être recompilée chaque fois qu'une sous-classe redéfinit une de ses méthodes. Il est alors *impossible de construire un compilateur incrémental sûr*. Pour un compilateur incrémental, nous devons choisir entre accepter des risques d'erreurs à l'exécution ou refuser de compiler. C'est ce dernier choix que nous avons fait.

La dernière remarque est la plus intéressante. Puisque le marché de l'objet repose sur l'argument de la réutilisation, les outils les plus prometteurs construits à base d'objets sont des *frameworks*. Dans [17, page 244], Jan Bosch explique que l'inversion de contrôle (qu'il appelle le principe d'Hollywood, « *Don't call us, we call you !* ») qui résulte de l'usage intensif de la liaison dynamique rend les frameworks objets fondamentaux pour la construction de « lignes de produits », bien plus que les classiques bibliothèques de sous-programmes. Mais pour l'exploiter efficacement les programmeurs doivent étendre et spécialiser les classes du framework *sans avoir à recompiler l'ensemble du framework*. La question est : quel sera le comportement du framework étendu ? Nous verrons dans la section suivante que, pour un même modèle, *ça dépend du langage !*

2.2.3 Résultats et analyses

La sémantique « raisonnable » définie, nous allons observer les résultats obtenus par des langages de programmation objets. Nous avons utilisé C++ [68], CLOS [67], Dylan [29], Eiffel [47],

[3] (Ligne, Colonne) dans [1..6]x[1..3].

Java [7], OCaml [63] et Smalltalk [33]. Nous avons compilé le même programme, détaillé en tableau 2.2 page 12 (en adaptant la syntaxe) avec les compilateurs respectifs suivants : gcc de Cygnus cygwin beta 20 et Microsoft Visual C++ 6.0 pour C++, Allegro CL [1] pour CLOS, the Functional Developer(TM) de Functional Objects [2], le GNU smalleiffel [27] et Eiffel workbench 4.5 de ISE [49] pour Eiffel, la JDK1.3 de SUN pour Java, OCaml de l'INRIA et Squeak [37] pour Smalltalk. Les sources et les résultats des exécutions peuvent être lus sur `http ://persoinfo.enst-bretagne.fr/ beugnard/papiers/lb-sem.shtml`.

Les conventions suivantes sont utilisées pour mettre l'accent sur les différences avec la sémantique « raisonnable » utilisée comme étalon :

- Le *résultat* (italique) n'est pas le plus spécialisé, mais ne produit pas d'erreur
- Une `erreur` (courrier) est signalée alors qu'une solution sûre existe
- Une **erreur** (gras) à l'exécution peut se produire si le code se complique[4]
- Une ***erreur*** (gras, italique)à l'exécution se produit effectivement

C++

appels	u	d	ud
cv(t)	cv(Top) in Up	`Top NOT Middle`	cv(Top) in Up
cv(m)	cv(Top) in Up	cv(Middle) in Down	*cv(Top) in Up*
cv(b)	cv(Top) in Up	cv(Middle) in Down	*cv(Top) in Up*
ctv(t)	Top NOT Bottom	Top NOT Middle	Top NOT Bottom
ctv(m)	Middle NOT Bottom	ctv(Middle) in Down	Middle NOT Bottom
ctv(b)	ctv(Bottom) in Up	ctv(Middle) in Down	*ctv(Bottom) in Up*

Tab. 2.4 – résultats C++

Le tableau 2.4 synthétise les résultats obtenus après l'exécution du programme C++. Les colonnes 1 et 2 ne sont pas surprenantes. La première ligne de la colonne 2 est une classique contrainte trop forte où le compilateur ne cherche pas de solution dans la hiérarchie de Down (dans Up). La colonne 3 est plus éloignée de la sémantique attendue ; aucune liaison dynamique n'est trouvée. C'est, en fait, normal, puisque C++ utilise une sémantique invariante, en comparant les signatures exactes. Pour C++ la méthode cv(Top) n'est pas considérée comme redéfinie mais une nouvelle méthode cv(Middle) est créée.

C++ implante une sémantique invariante avec une contrainte trop forte. Il garantit l'absence d'erreur d'exécution (due à ce problème).

CLOS

Le tableau 2.5 page suivante présente les résultats obtenus avec l'exécution du programme CLOS. Ils sont très proches de la sémantique « raisonnable », mais 4 appels déclenchent des erreurs alors qu'on peut espérer que le compilateur détectera les erreurs de typage. En fait le compilateur avertit le programmeur, mais compile tout de même. La troisième colonne est identique à la deuxième

[4]Si, par exemple, au lieu d'imprimer la signature de la méthode trouvée, on utilise l'argument et on appelle un service proposé par sa classe.

appels	u	d	ud
cv(t)	cv(Top) in Up	cv(Top) in Up	cv(Top) in Up
cv(m)	cv(Top) in Up	cv(Middle) in Down	cv(Middle) in Down
cv(b)	cv(Top) in Up	cv(Middle) in Down	cv(Middle) in Down
ctv(t)	***Runtime Error***	***Runtime Error***	***Runtime Error***
ctv(m)	***Runtime Error***	ctv(Middle) in Down	*ctv(Middle) in Down*
ctv(b)	ctv(Bottom) in Up	ctv(Middle) in Down	ctv(Middle) in Down

TAB. 2.5 – résultats CLOS

puisque les variables CLOS ne sont pas déclarée (il n'y a donc pas de différence entre le type déclaré et le type effectif).

CLOS accepte des erreurs à l'exécution, mais est compatible avec la sémantique raisonnable partout ailleurs, sauf en (5,3) qui risque de poser des problèmes de compilation incrémentale.

Dylan

appels	u	d	ud
cv(t)	cv(Top) in Up	cv(Top) in Up	cv(Top) in Up
cv(m)	cv(Top) in Up	cv(Middle) in Down	cv(Middle) in Down
cv(b)	cv(Top) in Up	cv(Middle) in Down	cv(Middle) in Down
ctv(t)	***Runtime Error***	***Runtime Error***	***Runtime Error***
ctv(m)	***Runtime Error***	ctv(Middle) in Down	*ctv(Middle) in Down*
ctv(b)	ctv(Bottom) in Up	ctv(Middle) in Down	ctv(Middle) in Down

TAB. 2.6 – Résultats Dylan

Le tableau 2.6 présente les résultats obtenus avec l'exécution du programme Dylan. Ils sont très proches de la sémantique « raisonnable », mais 4 appels déclenchent des erreurs alors qu'on aurait pu espérer que le compilateur détecte les erreurs de type.

Dylan, comme CLOS, accepte des erreurs à l'exécution, mais est compatible avec la sémantique raisonnable partout ailleurs, sauf en (5,3) qui risque de poser des problèmes de compilation incrémentale.

Eiffel

Le tableau 2.7 page suivante présente les résultats obtenus avec l'exécution du programme Eiffel. Les lignes 4, 5 et 6 montrent que les redéfinitions contravariantes de la méthode ctv sont interdites ; ctv est toujours trouvé dans Up. La ligne 1, colonne 2 a la même contrainte trop forte que C++. La ligne 1, colonne 3 est plus étonnante dans un langage fortement typé. Une erreur à l'exécution est produite, par les deux compilateurs utilisés, si l'argument (qui est un Top) est utilisé comme un Middle (plus spécialisé).

Avec une sémantique covariante, qui rejette les redéfinitions contravariantes, Eiffel qui clame

appels	u	d	ud
cv(t)	cv(Top) in Up	`Top NOT Middle`	**cv(Middle) in Down**
cv(m)	cv(Top) in Up	cv(Middle) in Down	cv(Middle) in Down
cv(b)	cv(Top) in Up	cv(Middle) in Down	cv(Middle) in Down
ctv(t)	Top NOT Bottom	Top NOT Bottom	Top NOT Bottom
ctv(m)	Middle NOT Bottom	`Middle NOT Bottom`	Middle NOT Bottom
ctv(b)	ctv(Bottom) in Up	*ctv(Bottom) in Up*	*ctv(Bottom) in Up*

TAB. 2.7 – Résultats Eiffel

être sûr, ne l'est pas[5]. Eiffel est loin de la sémantique raisonnable.

Java

appels	u	d	ud
cv(t)	cv(Top) in Up	cv(Top) in Up	cv(Top) in Up
cv(m)	cv(Top) in Up	cv(Middle) in Down	*cv(Top) in Up*
cv(b)	cv(Top) in Up	cv(Middle) in Down	*cv(Top) in Up*
ctv(t)	Top NOT Bottom	Top NOT Middle	Top NOT Bottom
ctv(m)	Middle NOT Bottom	ctv(Middle) in Down	Middle NOT Bottom
ctv(b)	ctv(Bottom) in Up	`ambiguous`	*ctv(Bottom) in Up*

TAB. 2.8 – Résultats Java

Le tableau 2.8 présente les résultats obtenus avec l'exécution du programme Java. La colonne 1 est classique. La ligne 1, colonne 2 décrit le comportement souhaité en outrepassant la redéfinition. La ligne 6, colonne 2 montre que le compilateur hésite entre deux définitions acceptables de ctv : ctv(Bottom) et ctv(Middle). Il ne choisit pas le comportement le plus spécifique comme le fait C++ et refuse de compiler. La colonne 3 est, comme pour C++, plus éloignée de la sémantique attendue ; il semble n'y avoir aucune liaison dynamique. L'explication est la même que pour C++, le compilateur ne cherche une liaison dynamique que pour des méthodes de signatures identiques, et considère donc la redéfinition de cv(Top) comme une surcharge, i.e. une autre méthode.

Java utilise une sémantique invariante de la redéfinition, avec une contrainte trop forte sur la dernière ligne de la deuxième colonne, alors que C++ a cette contrainte sur la première ligne de la deuxième colonne.

Smalltalk/Squeak

Le tableau 2.9 page suivante présente les résultats obtenus avec l'exécution du programme Smalltalk. Smalltalk n'est pas statiquement typé, aussi aucune erreur de compilation n'est-elle détectée lors de la compilation. En conséquence, de nombreuses (ligne 1, colonnes 2 et 3, la ligne 4 et

[5]Une solution non encore implantée est proposée pour supprimer ce risque : la règle du CATcall. Une étude des conséquences du CATcall en s'appuyant sur les tableaux présentés ici est proposée sur http ://perso-info.enst-bretagne.fr/˜ beugnard/papiers/catcall.html.

appels	u	d	ud
cv(t)	cv(X) in Up	**cv(X) in Down**	**cv(X) in Down**
cv(m)	cv(X) in Up	cv(X) in Down	cv(X) in Down
cv(b)	cv(X) in Up	cv(X) in Down	cv(X) in Down
ctv(t)	**ctv(X) in Up**	**ctv(X) in Down**	**ctv(X) in Down**
ctv(m)	**ctv(X) in Up**	ctv(X) in Down	*ctv(X) in Down*
ctv(b)	ctv(X) in Up	ctv(X) in Down	ctv(X) in Down

TAB. 2.9 – Résultats Smalltalk/Squeak

la ligne 5, colonne 1) erreurs peuvent survenir, produisant l'apparition d'une fenêtre Smalltalk `MessageNotUnderstood`. Notez que les colonnes 2 et 3 sont identiques, ce qui rend en Smalltalk, la liaison dynamique très naturelle.

Smalltalk accepte de nombreuses erreurs de type potentielles, est assez éloigné de la sémantique « raisonnable », mais il possède une régularité intéressante.

OCaml

appels	u	d	ud
cv(t)	cv(Top) in Up	*cv(Middle) in Down*	*cv(Middle) in Down*
cv(m)	cv(Top) in Up	cv(Middle) in Down	cv(Middle) in Down
cv(b)	cv(Top) in Up	cv(Middle) in Down	cv(Middle) in Down
ctv(t)	*ctv(Bottom) in Up*	*ctv(Middle) in Down*	*ctv(Middle) in Down*
ctv(m)	*ctv(Bottom) in Up*	ctv(Middle) in Down	*ctv(Middle) in Down*
ctv(b)	ctv(Bottom) in Up	ctv(Middle) in Down	ctv(Middle) in Down

TAB. 2.10 – Résultats OCaml

Le tableau 2.10 présente les résultats obtenus avec l'exécution du programme Ocaml. OCaml se comporte exactement comme Smalltalk, mais avec la garantie de ne pas échouer lors de l'exécution. Pour le vérifier nous avons modifié la classe Middle afin de tenter de déclencher une erreur – comme pour Eiffel –, en faisant appel à un service spécifique de Middle (voir le code source sur [10].) Le compilateur détecte bien l'incohérence de type et refuse alors de compiler la classe Down.

OCaml garantit l'absence d'erreur par un mécanisme d'inférence de type sophistiqué. Toutefois, comme la surcharge est interdite, il est impossible dans les cas covariants (lignes 1 à 3) d'obtenir deux types de comportements différents, celui de la première ligne, spécialisé pour Top, et celui des deux lignes suivantes, spécialisés pour Middle. Il y a donc au minimum deux écarts par rapport à la sémantique « raisonnable » : ligne1, colonnes 2 et 3.

Comparaison

Afin de comparer les différences nous regroupons dans trois tableaux, les comportements de chaque langage dans chacun des tests. Le tableau 2.11 page suivante montre déjà des différences dans le cas le plus simple. Toutefois ces différences sont plus liées à la philosophie de la compilation

et du développement qu'à la nature de la programmation objet. Les langages d'origine fonctionnelle acceptent de compiler un code comportant des erreurs de type.

Nous faisons apparaître trois lignes supplémentaires dans les tableaux 2.11, 2.12 et 2.13 page suivante pour montrer comment les langages se comportent dans le cas de redéfinition invariante. Une méthode inv(Middle) est définie avec la même signature dans Up et Down. Dans ce cas « normal », tous les langages font les mêmes choix (au traitement des erreurs près).

appels	Raisonnable	C++	CLOS	Dylan	Eiffel	Java	Smalltalk	OCaml
cv(t)	Up	Up	Up	Up	Up	Up	Up	Up
cv(m)	Up	Up	Up	Up	Up	Up	Up	Up
cv(b)	Up	Up	Up	Up	Up	Up	Up	Up
ctv(t)	Error	Error	***Error***	***Error***	Error	Error	**Up**	*Up*
ctv(m)	Error	Error	***Error***	***Error***	Error	Error	**Up**	*Up*
ctv(b)	Up	Up	Up	Up	Up	Up	Up	Up
inv(t)	Error	Error	***Error***	***Error***	Error	Error	**Up**	*Up*
inv(m)	Up	Up	Up	Up	Up	Up	Up	Up
inv(b)	Up	Up	Up	Up	Up	Up	Up	Up

Tab. 2.11 – Comparaison pour le récepteur u (Up)

Le tableau 2.12 commence à montrer la variabilité de l'interprétation. Le sous-classage de Up par Down ne se traduit pas par les mêmes contraintes en C++ et Java. Eiffel, par principe, interdit la contravariance. Les langages d'origine fonctionnelle prennent toujours des risques.

appels	Raison.	C++	CLOS	Dylan	Eiffel	Java	Smalltalk	OCaml
cv(t)	Up	`Error`	Up	Up	`Error`	Up	**Down**	*Down*
cv(m)	Down	Down	Down	Down	Down	Down	Down	Down
cv(b)	Down	Down	Down	Down	Down	Down	Down	Down
ctv(t)	Error	Error	***Error***	***Error***	Error	Error	**Down**	*Down*
ctv(m)	Down	Down	Down	Down	`Error`	Down	Down	Down
ctv(b)	Down	Down	Down	Down	*Up*	`Error`	Down	Down
inv(t)	Error	Error	***Error***	***Error***	Error	Error	**Down**	*Down*
inv(m)	Down	Down	Down	Down	Down	Down	Down	Down
inv(b)	Down	Down	Down	Down	Down	Down	Down	Down

Tab. 2.12 – Comparaison pour le récepteur d (Down)

C'est dans le tableau 2.13 page suivante que s'exprime la plus grande différence, en n'ayant pourtant exploité qu'une liaison dynamique simple, celle sur l'objet récepteur.

La variabilité des interprétations de la liaison dynamique est surprenante et montre le besoin d'en définir une sémantique opérationnelle commune. Trop souvent, la sémantique est définie par le compilateur, et c'est alors aux concepteurs et programmeurs de la découvrir et de s'y adapter.

Nous allons voir dans la partie suivante les conséquences de la variabilité des sémantiques des langages à objets dans le cadre de composants logiciels qu'on souhaite assembler.

appels	Raison.	C++	CLOS	Dylan	Eiffel	Java	Smalltalk	OCaml
cv(t)	Up	Up	Up	Up	**Down**	Up	**Down**	*Down*
cv(m)	Down	*Up*	Down	Down	Down	*Up*	Down	Down
cv(b)	Down	*Up*	Down	Down	Down	*Up*	Down	Down
ctv(t)	Error	Error	***Error***	***Error***	Error	Error	**Down**	*Down*
ctv(m)	Error	Error	*Down*	*Down*	Error	Error	*Down*	*Down*
ctv(b)	Down	*Up*	Down	Down	*Up*	*Up*	Down	Down
inv(t)	Error	Error	***Error***	***Error***	Error	Error	**Down**	*Down*
inv(m)	Down	Down	Down	Down	Down	Down	Down	Down
inv(b)	Down	Down	Down	Down	Down	Down	Down	Down

TAB. 2.13 – Comparaison pour le récepteur ud (Down déclaré Up)

2.3 Peut-on réaliser des composants avec un langage à objets ?

Comme nous l'avons vu dans la partie précédente les interprétations de la redéfinition et de la surcharge dans les langages à objets sont nombreuses.

Aujourd'hui, les besoins d'industrialisation du logiciel mènent à une organisation (architecture) du code sous la forme de composants. L'idée de construire des logiciels, comme on le fait pour des systèmes électroniques, par assemblage de composants est ancienne. De nombreux modèles de composants sont proposés (Fractal, .NET, EJB, CCM), tous implantés avec des langages à objets.

Nous démontrons que la différence des comportements des langages à objets vis-à-vis de la redéfinition et de la surcharge se marie mal avec la caractéristique essentielle des composants qu'est l'encapsulation. Pour ce faire nous montrons par une expérience comment l'assemblage de composants ne peut conduire à un comportement prévisible que si le contrat du composant est précis au point de remettre en cause l'encapsulation.

2.3.1 Composant et contrat

Un composant est défini comme une entité logicielle indépendante, déployable et pouvant être composée avec d'autres [69]. Pour parvenir à cela, de nombreux modèles ont été proposés. Tous s'appuient sur une caractéristique qui permet de manipuler des objets complexes sans en montrer tous les détails : l'encapsulation. Pour ce faire, l'information utilisée pour l'assemblage des composants n'est pas le composant lui-même, alors considéré comme une boîte blanche, mais une représentation du composant, son interface, afin de ne voir du composant qu'une boîte noire. Comme l'ont constaté de nombreux auteurs, il est parfois indispensable de connaître certains détails de l'implantation du composant pour pouvoir l'assembler correctement [18]. On parle alors de boîte grise. Pour évaluer le degré d'opacité d'un composant, nous avons proposé dans [15] une organisation de l'interface des composants en quatre niveaux de contrat (voir 2.1 page 7) : Syntaxique, sémantique, synchronisation, qualité de service.

La spécification de ces quatre niveaux de contrat devrait permettre l'assemblage de composants. C'est-à-dire non seulement la possibilité de réaliser l'assemblage par des mécanismes technologiques adaptés, mais également de prévoir le comportement résultant de cet assemblage.

Or, comme nous allons le voir, pour prévoir le comportement de certains assemblages, il faut connaître le langage d'implantation du composant mais également celui de ses clients !

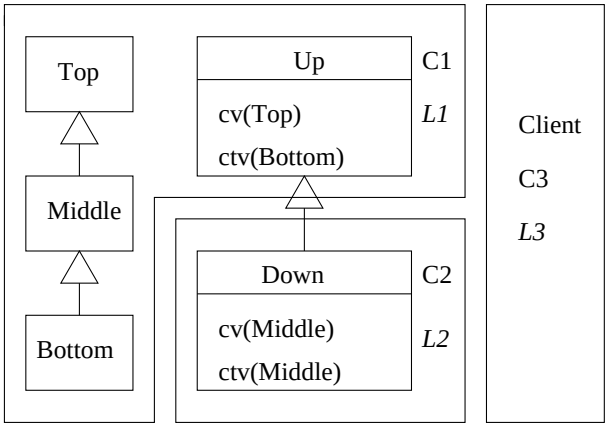


FIG. 2.3 – Comment se comportera cet assemblage ?

2.3.2 Assemblage de composants

Le modèle de la figure 2.2 page 10 peut servir de support à une expérience d'assemblage de composants. Imaginons, comme dans la figure 2.3, que les classes Up, Top, Middle et Bottom définissent un cadriciel C1 élaboré dans un langage *L1*. Par la suite, dans le cadre d'une évolution de C1, la classe Down, écrite dans un langage *L2*, étend Up pour former un composant C2. Enfin, un client C3 écrit avec le langage *L3* réalise les différents appels.

Expérience avec .NET

Nous avons réalisé cette expérience avec l'environnement .NET [51] qui annonce une interopérabilité de langage. Ainsi, les langages C#, Visual Basic et C++ qui forment le trio de langage de base de .NET ont été utilisés pour réaliser les différents composants et assemblages. Nous obtenons donc 27 signatures différentes.

Résultats

Pour simplifier la lecture, les résultats sont organisés en 3 tableaux de 9 signatures. Chaque tableau correspond à un langage client (respectivement C#, Visual Basic et C++). Le langage de programmation du cadriciel de référence (L1) apparaît par colonne, celui de l'extension (L2) par ligne, et chaque case est la signature observée.

Le tableau 2.14 page suivante montre les résultats obtenus pour un client toujours écrit en C#. On retrouve presque partout la signature de C# sauf dans la dernière colonne où l'on retrouve un comportement « à la C++ ». La dernière colonne correspond au cas où le cadriciel C1 est écrit en C++.

Le tableau 2.15 page suivante montre les résultats obtenus pour un client toujours écrit en Visual Basic. On retrouve presque partout la signature de Visual Basic sauf dans la dernière colonne où l'on retrouve un comportement « à la C++ ». La dernière colonne correspond encore au cas où le cadriciel C1 est écrit en C++. Une autre différence apparaît dans le cas où le cadriciel est écrit en Visual Basic et l'extension en C++ ; le comportement observé est celui de C# (ligne 3, colonne 2) ! La dernière ligne est intéressante car elle fait apparaître les trois signatures : Visual Basic, C# puis C++ !

L2/L1	C#	VB	C++
C#	C#	C#	C++
VB	C#	C#	C++
C++	C#	C#	C++

TAB. 2.14 – Résultats pour un client C# (L3 = C#)

L2/L1	C#	VB	C++
C#	VB	VB	C++
VB	VB	VB	C++
C++	VB	C#	C++

TAB. 2.15 – Résultats pour un client Visual Basic (L3 = Visual Basic)

La tableau 2.16 montre que, pour un client C++, le comportement observé est celui de C++ sauf si C2, l'extension du cadriciel C1, est écrit en C# ce qui fait apparaître un comportement qui mélange celui de C++ et de Visual Basic et ne correspond à aucune signature identifiée !

L2/L1	C#	VB	C++
C#	(C++/VB)	(C++/VB)	(C++/VB)
VB	C++	C++	C++
C++	C++	C++	C++

TAB. 2.16 – Résultats pour un client C++ (L3 = C++)

Le tableau 2.17 page suivante montre les résultats détaillés pour la première ligne du tableau précédent (D pour Down, U pour Up et E pour Erreur). On retrouve presque partout la signature de C++ sauf dans la dernière ligne (ctv(B)) où l'on retrouve un comportement « à la Visual Basic », même quand Visual Basic n'est pas utilisé !

Interprétation

L'expérience menée ne met en œuvre que le niveau « syntaxique » des contrats. Les niveaux « sémantique », « synchronisation » et « qualité de service » n'apporteraient *a priori* aucune information. On constate que dans la grande majorité des cas, le comportement observé est proche de celui attendu par le client. Les cas de C# et C++ sont les plus stables avec 6/9 « bons » résultats (en détail, 3 « mauvais » *slots*[6]) tandis que Visual Basic obtient 5/9 « bons » comportements (en détail, 7 « mauvais » *slots*). Les causes des « mauvais » comportements sont essentiellement dues à C++ lorsque les clients sont C# ou Visual Basic. La raison principale est probablement l'exception à l'héritage que fait C++ dans le slot ligne 1, colonne 2 de la table 2.4 page 14, où le cv(T) défini dans Up est masqué par la surcharge cv(M) définie dans Down.

[6]Un slot est une case du tableau détaillé.

C#C#	u	d	ud	VBC#	u	d	ud	C++C#	u	d	ud
cv(T)	U	E	U	cv(T)	U	E	U	cv(T)	U	E	U
cv(M)	U	D	U	cv(M)	U	D	U	cv(M)	U	D	U
cv(B)	U	D	U	cv(B)	U	D	U	cv(B)	U	D	U
ctv(T)	E	E	E	ctv(T)	E	E	E	ctv(T)	E	E	E
ctv(M)	E	D	E	ctv(M)	E	D	E	ctv(M)	E	D	E
ctv(B)	*U*	***U***	*U*	*ctv(B)*	*U*	***U***	*U*	*ctv(B)*	*U*	***U***	*U*

TAB. 2.17 – Résultats détaillés pour un client C++ et une extension C#

Si l'on essaye de décrire le contrat d'interface du composant C2, on serait amené à quelque chose du style de ce qui est précisé dans le tableau 2.18. C3 est le client.

Services	Retourne le résultat de la méthode trouvée dans
cv(Top)	Up Erreur si le récepteur est un Down déclaré Down et que le C1 est en C++ ou si C3 est écrit en C++.
ctv(Bottom)	Up généralement, mais Down lorsque (1) C3 est écrit en C# et que le récepteur est un Down déclaré Down, ou (2) C3 est écrit en Visual Basic et que C1 est écrit en C++ ou que (C1 est écrit en Visual Basic et C2 en C++), ou (3) C3 est écrit en C++ et que C2 est écrit en Visual Basic ou C++.
cv(Middle)	Up généralement, mais Down lorsque le récepteur est un Down déclaré Down.
ctv(Middle)	Down Erreur lorsque le récepteur n'est pas un Down déclaré Down.

TAB. 2.18 – Tentative de contrat du composant C2

Comme on le voit, le contrat fait référence à des détails d'implantation et à des propriétés des clients du composant. On a moins de propriété d'encapsulation et de localité. La boîte noire s'éclaircit et ses frontières se désagrègent ...

Les références explicites au langage pourraient peut-être remplacées par des meta-informations relatives aux concepts choisis par les langages en suivant l'approche proposée dans [19]. Toutefois, l'expérience que nous avons sur la variabilité des langages et la finesse des différences entre interprétation, nous convainc, pour le moment, que de telles meta-informations seraient plus complexes à décrire que la simple référence au langage.

Au-delà de la compréhension fine des résultats, cette expérience montre que, pour prévoir le comportement d'un assemblage, le client d'un composant doit avoir des informations sur l'interprétation de la surcharge qui est utilisée dans le composant auquel il souhaite s'assembler. Cette information peut être précisée en explicitant *les langages* ayant servi à implanter le composant, mais également en exposant la structure interne du composant comme les relations d'héritage réelles, les redéfinitions et surcharges réalisées, etc. Mais dans ce cas, qu'en est-il de l'encapsulation ?

2.4 Conclusion sur les contrats d'assemblage

Les études théoriques sur les assemblages de composants se focalisent sur les aspects statiques [30, 6, 5], même lorsqu'ils traitent de l'assemblage dynamique. La liaison dynamique au sens de la programmation à objets est passée sous silence ; seule la « compilabilité » est étudiée pour garantir l'absence d'erreur à l'exécution sans se soucier du comportement attendu. Les approches pragmatiques proposent des implantations de modèle de composant dans un milieu homogène (Java par exemple), ou laissent à la responsabilité des programmeurs l'implantation des composants (CORBA et CCM) ce qui les dédouane de l'apparition de comportements inattendus.

L'expérience décrite dans la partie 2.3 page 19 se limite à l'environnement .NET et à des langages avec redéfinition invariante et surcharge autorisée. Déjà dans ce cadre, la variabilité est grande. Nous sommes convaincus, mais les expériences restent à faire, que par exemple l'introduction du langage Eiffel dans l'expérience multiplierait les points de variabilité, puisqu'Eiffel interdit la surcharge et autorise des redéfinitions covariantes.

Or, il nous semble que pour acquérir toute son utilité le paradigme des composants doit pouvoir être utilisable en milieu ouvert multi-langages et garantir le comportement des assemblages à partir des comportements et contrats des composants.

Pour atteindre cet objectif, voici quelques suggestions :

- Étudier la combinatoire des interactions entre langages pour gérer la multiplicité des interprétations. Mais comme nous l'avons montré avec .NET pour 3 langages pas tellement différents, toutes les combinaisons sont à considérer. La combinatoire est exponentielle et l'opération de « combinaison » des signatures n'est pas une loi interne, comme la première ligne du cas d'un client C++ l'illustre dans le tableau 2.16 page 21. Devant la prolifération des langages et des interprétations, cette approche nous semble peu applicable. Mais pourra-t-on faire autrement ?
- Enrichir le niveau de contrat sémantique pour exprimer les interprétations retenues. Mais ce cas est complémentaire de la première suggestion. L'exploitation des contrats s'appuiera sur la compréhension des interactions de langage.
- Contraindre l'usage de la redéfinition aux cas invariants et interdire la surcharge. Tous les langages à objets ont alors le même comportement, et l'assemblage est simple. Cela induit des modifications dans les compilateurs actuels pour interdire effectivement les cas problématiques. Il faudrait des mesures pour compter le nombre de cas d'utilisation des interactions difficiles pour évaluer quel serait l'impact sur les codes existants.

La dernière solution nous semble la plus prometteuse. L'histoire des langages de programmation nous apprend que l'évolution des concepts de langages suit un double mouvement ; tout d'abord une exploration d'idées et de concepts, puis une sélection de ces concepts qui se traduit par des contraintes sur les langages. Peut-être est-il temps pour les langages à objets de faire des choix et de contraindre les règles d'utilisation de la redéfinition et de la surcharge ?

Les dangers de la surcharge ont déjà été mis en évidence [50, 31] et nos expériences peuvent être considérées comme un argument supplémentaire en faveur de sa suppression. Le choix de la covariance, contravariance ou l'invariance de la redéfinition semble plus ouvert, mais dans la pratique, l'invariance est largement dominante et sa mise en œuvre par la liaison dynamique est relativement simple. De plus, dans ce cas, si le besoin s'en fait sentir, la covariance peut être simplement simulée pourvu que le langage dispose de la capacité de tester le type effectif d'un objet (en l'occurrence un argument). Il serait intéressant de mesurer l'impact de ces choix sur des applications et des composants existants.

Pour conclure ce chapitre, il nous semble que l'étude de l'interprétation de la surcharge et de

la redéfinition dans les langages à objet et de ses conséquences sur la programmation à objets ne progresse pas beaucoup. Dans un contexte où la transformation de modèles (MDA [55]), l'interopérabilité de langages et l'assemblage de composants sont considérés comme cruciaux pour l'industrialisation du logiciel, c'est très étonnant ! Si l'on souhaite construire des systèmes par assemblage de composant un compromis devra être trouvé entre (1) limiter les possibilités des langages pour simplifier la vérification des outils d'assemblages et (2) construire des vérificateurs d'assemblage très complexes[7]...

[7] À court terme, l'industrie de l'informatique préférera sans doute la seconde option pour d'évidente raison de marché !

Chapitre 3

Les abstractions de communication

Le chapitre précédent était consacré aux informations nécessaires à l'assemblage des entités logicielles. Nous allons dans ce chapitre étudier des techniques d'assemblage. Les approches abstraites d'assemblage ont été introduites par les architectures logicielles et les langages d'architecture logicielle (ADL) [44, 40, 53, 66, 34, 4][1].

L'architecture logicielle est une étape importante dans le cycle de développement des systèmes. Elle est, la plupart du temps, définie à partir de composants et de connecteurs et d'une configuration qui décrit leur assemblage. Une architecture est décrite par un assemblage de boites (les composants) et de traits (les connecteurs).

Comme nous l'avons déjà vu, il existe de nombreux modèles de composants et plusieurs implantations (EJB, CCM, .NET). Les modèles de connecteurs sont plus rares [72, 73].

Nous allons montrer que derrière le terme de connecteur plusieurs concepts logiciels se cachent. En effet, les implantations des connecteurs prennent différentes formes ; parfois celle de protocole de communication (http, tcp), parfois celle de langage (SQL), parfois celle de composant (linda, jini) et parfois de mécanisme d'intergiciels comme les appels de procédures (ou de méthodes) à distance (Remote Procedure Call - RPC de CORBA, RMI ou .NET).

Dans la suite de ce chapitre nous allons établir dans la partie 3.1 la différence que nous faisons entre deux types d'abstractions de communication : les connecteurs et les composants de communication (aussi appelés médiums). Puis, dans la partie 3.2 page 28, nous détaillons les travaux que nous avons réalisés pour spécifier et concevoir des composants de communication. Enfin, la partie 3.3 page 59 illustre la notion de connecteur et une implantation d'un connecteur spécialisé dans l'équilibrage de charge.

La partie consacrée aux composants de communication est basée sur les publications [22, 24] et celle consacrée aux connecteurs sur [43, 42]

3.1 Deux types d'abstraction de communication

L'architecture logicielle consiste à décrire un modèle abstrait d'un système qui précise quelles interdépendances existent entre les « constituants » du système. En pratique, les architectes présentent les applications comme un ensemble de composants interagissant. Ainsi, deux entités principales sont manipulées dans la description de l'architecture d'une application : les composants,

[1] Pour un état de l'art sur les langages d'architecture, on pourra lire le rapport du projet ACCORD accessible sur `http ://www.infres.enst.fr/projets/accord/lot1/etat_art.html`.

qui représentent les principales unités de calcul du système, et leurs connexions (souvent désignées comme connecteurs), qui représentent des abstractions de communication de haut niveau. Pour simplifier la communication entre architectes et développeurs, les architectures d'application sont décrites informellement en utilisant une collection de boites et de traits [3], représentant respectivement les composants et les connecteurs. Cette combinaison constitue l'organisation générale de l'application ou sa configuration. Dans cette partie, nous montrons pourquoi la représentation des connexions par des traits est insuffisante et ambiguë. Nous expliquons ensuite comment nous résolvons ce problème en introduisant une différence entre composant de communication et connecteur puis entre une connexion et un connecteur.

3.1.1 Représentation des connexions par des traits

Nous distinguons entre deux types de connexions dans la communication entre deux composants : simples et complexes. Les connexions simples sont utilisées dans un environnement local, elles y sont réalisées comme des appels de procédure. Ce sont les connexions les plus utilisées dans les langages de programmation, elles y sont primitives. L'appel de procédure est réalisé par un branchement direct depuis l'appelant vers l'appelé. La communication est fiable, lorsqu'un appel est lancé, sa réception est toujours garantie par l'autre composant. La sémantique d'une connexion simple est claire, elle est représentée par un trait entre deux boites nommées (Figure 3.1).

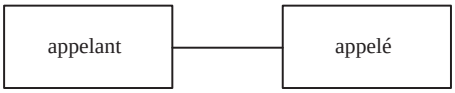


Fig. 3.1 – Connexion simple.

Une connexion complexe peut refléter soit une communication avec des propriétés comme la (non) fiabilité ou l'asynchronisme, soit une communication impliquant plus de deux participants.

Dans les sections suivantes nous montrons comment ce connecteur-trait est insuffisant pour décrire la notion de connecteur pour deux raisons ; la pauvreté et l'ambiguïté de la relation point-à-point du trait et l'imprécision de la nature des extrémités.

Ambiguïté du « connecteur-trait »

En architecture logicielle, une connexion simple entre deux composants dans un environnement local, comme représenté dans la Figure 3.1, est facile à interpréter : l'appelant requiert un service offert par l'appelé. La situation se complique dans une communication complexe qui, par exemple, fait intervenir plusieurs composants.

La Figure 3.2 page suivante (a) représente une connexion complexe impliquant plusieurs appelants et un seul appelé. A priori, cette figure ne soulève aucune interrogation, il est naturel qu'un composant (l'appelé) puisse fournir un même service à plusieurs autres composants en même temps. On peut tout de même se demander si les composants appelants requièrent le service en exclusion mutuelle ou non ?

Le Figure 3.2 page suivante (b) montre une autre connexion complexe, cette fois-ci impliquant un seul composant appelant et plusieurs appelés. Une interprétation intuitive de cette figure serait que le composant appelant requiert plusieurs services de différents autres composants. Mais est-ce

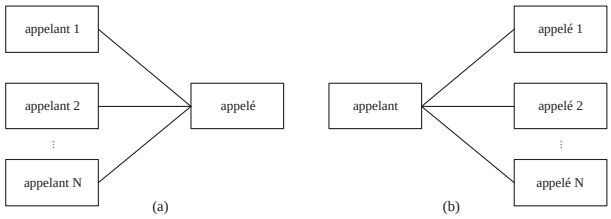


FIG. 3.2 – Connexion complexe. (a) plusieurs appelants - un appelé, (b) un appelant - plusieurs appelés

vraiment la bonne intention ? Il est possible que le composant appelant utilise les autres composants pour équilibrer la charge, ou bien les utilise pour assurer la fiabilité à l'aide d'un consensus [32] par exemple.

Cette description d'architecture devrait être complétée par une documentation bien détaillée. Sa représentation graphique, avec des boites et des traits, est destinée à faciliter sa compréhension et à simplifier la communication entre les concepteurs et les développeurs. Cependant, comme représentée dans la Figure 3.2, elle manque de précision et mène à plusieurs interprétations.

Pour résoudre cette ambiguïté nous proposons de représenter les abstractions de communication par des ellipses. Les ellipses sont reliées aux composants qu'elles assemblent par des traits. Ces traits ont une sémantique de connexion simple (localité, synchronisme de type appel de procédure).

Nature des extrémités

Les « connecteur-trait », s'ils permettent de donner une bonne idée de l'architecture générale d'un système, ne permettent pas de distinguer la nature différente des « connexions » en fonction des interfaces des composants à relier. La figure 3.3 illustre deux cas de figure : les composants à assembler ont des interfaces faites pour être relier et les composants à assembler ont des interfaces faites pour communiquer *au travers du* « connecteur-trait ».

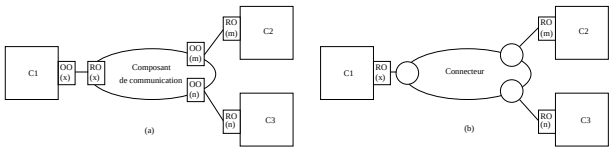


FIG. 3.3 – Deux types d'abstraction de communication. (a) composant de communication, (b) connecteur

Pour distinguer les deux sortes de « connecteur-trait » nous introduisons dans la section suivante deux types d'abstraction de communication complexe.

3.1.2 Composant de communication et connecteur

Nous distinguons les deux types d'abstraction de communication suivants :

Les composants de communication qui sont des abstractions de communication qui possèdent des points d'attache explicites appelés ports (représentés par des carrés autour de l'ellipse).

Les ports implémentent les interfaces du composant, les signatures des méthodes requises et offertes sont ainsi bien spécifiées. Dans la Figure 3.3 page précédente (a), le composant de communication requiert l'opération offerte $OO(x)$ du composant standard $C1$, et offre deux opérations $OO(m)$ et $OO(n)$. Ces dernières sont requises par les opérations $OR(m)$ et $OR(n)$ des composants standards $C2$ et $C3$. Les opérations x, m et n de $C1$, $C2$ et $C3$ sont complètement différentes, elles ne se connaissent pas. Elles interagissent avec les opérations x, m et n du composant de communication à travers une connexion simple (un appel de procédure).

Des connecteurs qui sont des abstractions de communication qui possèdent des points d'attache implicites appelés prises (représentés par des ronds autour de l'ellipse). Les prises reflètent des interfaces implicites, ainsi elles ne spécifient pas directement des opérations offertes ou requises, elles donnent seulement des contraintes sur des interfaces explicites futures. Ces interfaces sont adaptables aux interfaces de composants standards connectés à l'assemblage. Dans la Figure 3.3 page précédente (b), l'opération offerte $OO(x)$ du composant $C1$ est requise par les opérations $OR(x)$ des composants $C2$ et $C3$ mais les appels passent à travers le connecteur pour définir la sémantique de la communication. L'opération x offerte par $C1$ est la même qui est requise par $C2$ et $C3$, ces composants ne voient pas le connecteur, la communication, si elle est distante, est faite de manière *transparente*.

La nature de ces deux types d'abstraction de communication sont fondamentalement différentes. Leurs interfaces sont très différentes, mais leur façon d'être développées aussi, ainsi que la façon de les utiliser. Les sections suivantes (3.2, 3.3 page 59) vont le détailler, mais pour synthétiser, les grandes lignes sont les suivantes :

- Les **composants de communication** sont réalisés par un processus de développement classique. Le résultat est un logiciel qui peut, comme tout composant, être placé sur étagère. Pour être utilisé, il devra être assemblé, à d'autres composants via un connecteur.
- Les **connecteur** sont réalisé par un processus de développement d'un compilateur. Le résultat est un générateur qui est placé sur étagère. Pour être utilisé, il devra être relié à plusieurs composants (qui lui fourniront leurs interfaces) puis le générateur produira les éléments de la connexion.

La partie suivante est consacrée aux composants de communication qui sont structurellement proches des composants. Ils sont dédiés à la communication et sont distincts des composants classiques par la nature répartie de leurs interfaces. Puis la partie 3.3 page 59 présentera les connecteurs.

3.2 Composant de communication

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, dans la section 3.2.1 page suivante, 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 dans 3.2.2 page 36 une définition des médiums. Puis, nous détaillons en 3.2.3 page 37 la méthodologie de spécification en UML de ces composants. Nous donnons un exemple complet de spécification de médiums qui suit cette méthodologie. La spécification est faite à un niveau abstrait, indépendamment de tout contexte d'utilisation ou de choix d'implémentation. Ensuite nous décrivons succinctement comment

ont été mis en œuvre ces idées dans un *framework* de déploiement de médiums (3.2.5 page 46). Enfin, pour clore cette partie (3.2.6 page 49), nous présenteront le processus de transformation de modèle qui permet, à partir de la spécification abstraite du médium, d'obtenir des implémentation déployables.

3.2.1 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.

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 et 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.

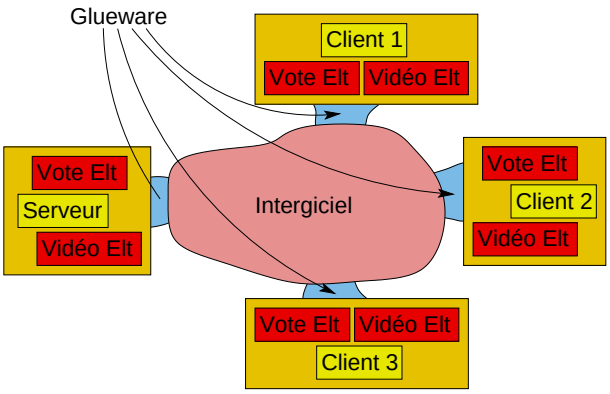


FIG. 3.4 – Détail de l'architecture de l'application de vidéo interactive

Architecture de l'application de vidéo interactive L'architecture de cette application est représentée sur la figure 3.4, 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.

Afin de résoudre ces problèmes et limitations, nous proposons 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 3.5 page suivante. 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

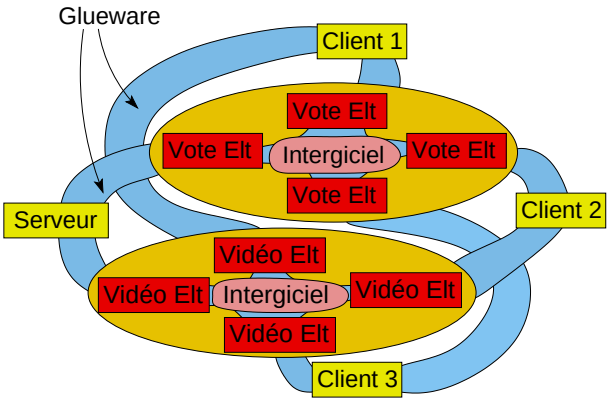


FIG. 3.5 – Réorganisation des éléments d'interaction dans l'application de vidéo interactive

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 forment 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'interactions*. Afin d'éviter toute ambiguïté entre un composant « fonctionnel » et un composant d'interaction, ce dernier type de composant sera aussi appelé *médium*. Un médium est donc la réification d'une *abstraction d'interaction* sous la forme d'un composant logiciel.

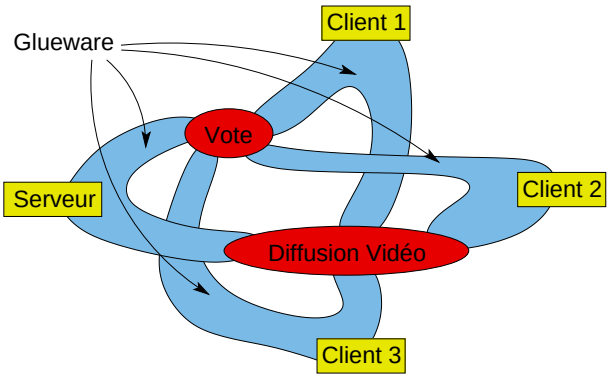


FIG. 3.6 – Nouvelle architecture de l'application de vidéo interactive

En considérant les systèmes d'interaction comme des médiums, l'architecture de notre application de vidéo interactive devient celle de la figure 3.6. 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, contrairement à la première approche. De plus, les abstractions d'interaction sont désormais placées à un niveau équivalent aux aspects fonctionnels.

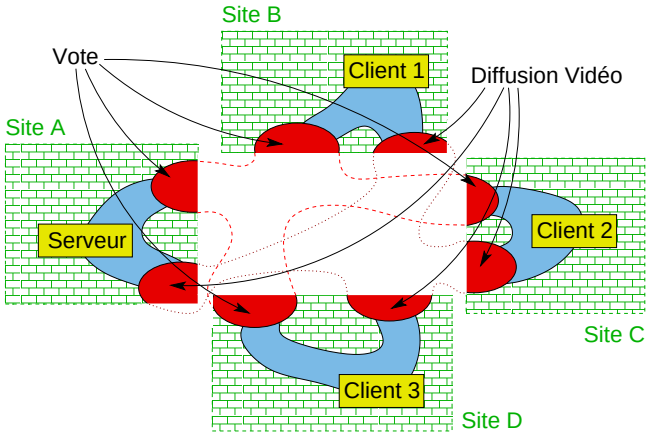


FIG. 3.7 – Architecture de déploiement de l'application de vidéo interactive

Le déploiement de la nouvelle application est représenté sur la figure 3.7. 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 3.4 page 29 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.

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.

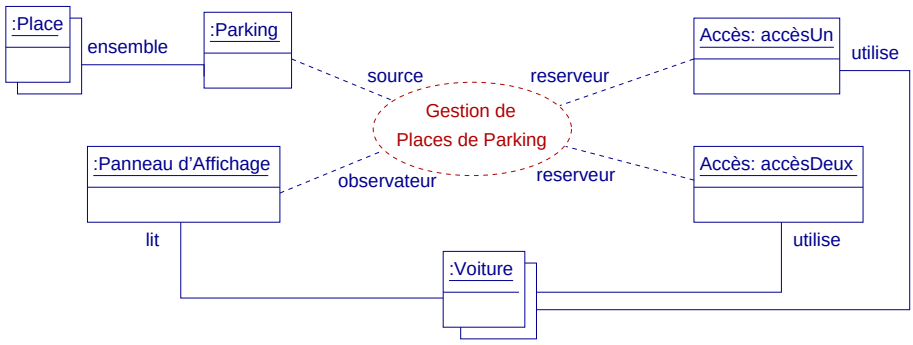


FIG. 3.8 – Application de gestion de places de parking

La figure 3.8 est un diagramme d'instance UML représentant cette application. On y retrouve les deux accès, le panneau d'affichage ainsi que le composant 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 connait 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 3.9 page suivante). 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

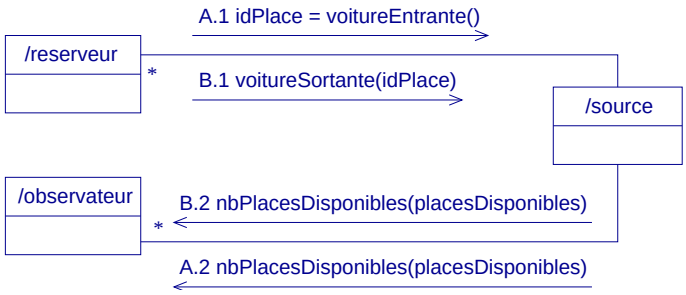


FIG. 3.9 – Une première spécification de la collaboration de gestion de parking

informés du nouveau nombre de places disponibles par l'appel de `nbPlacesDisponibles`, messages A.2 et B.2).

Pour plus de précision sur ce que font ces méthodes, des pré et postconditions en OCL pourraient être définies afin de spécifier complétement 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 précisement, 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 méthodes 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 voyages jouent les rôles de réserveur et d'observateur afin de réserver des places pour leurs clients.

Le contexte de cette application est complétement 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.

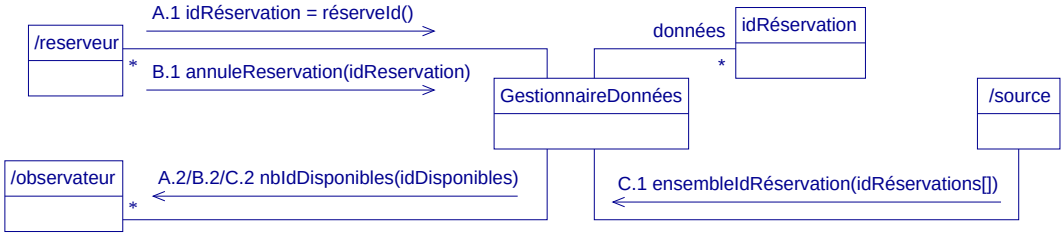


FIG. 3.10 – Collaboration générique de gestion de réservation d'identificateurs

Pourquoi alors ne pas concevoir un sytème générique de gestion d'identificateurs et adaptable à tout type de donnée ? Ce système serait alors systèmatiquement 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 3.10 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 prévénir les composants observateurs des changements du nombre d'identificateurs disponibles (en appelant le service `nbIdDisponibles`, messages A.2, B.2 et C.2).

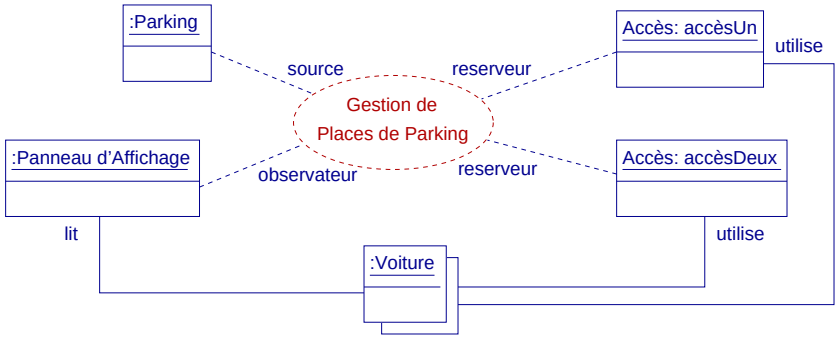


FIG. 3.11 – Nouvelle description de l'application de gestion de places de parking

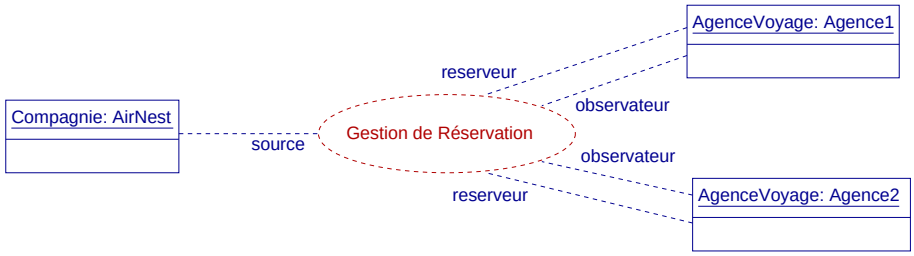


FIG. 3.12 – Description de l'application de réservation aérienne

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 3.11 pour l'application de gestion de places de parking et par la figure 3.12 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 « autosuffisante », 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.

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 précisement cette notion de médium.

3.2.2 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 appellons 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 pahses 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 systémes 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.

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 médium.

3.2.3 Méthodologie de spécification de médiums en UML

Comme pour les composants classiques, les composants de communication requièrent une spécification précise afin de pouvoir être utiliser 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 [16, 45, 46]. 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 [57]). Une abstraction de communication peut être représentée par une collaboration UML comme nous l'avons vu dans la section 3.2.1 page 32. 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 différentes :

Un diagramme de collaboration pour décrire l'aspect structurel du médium. Des messages pourront 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 la séquentialité 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'état. 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'état 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 ne sera 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'expressions des gardes de transition dans les diagrammes d'états ou d'envoi de messages dans les diagrammes de collaboration. L'utilisation d'OCL nous permet d'avoir un lien logique entre les différentes vues utilisées pour la spécification d'un médium.

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.

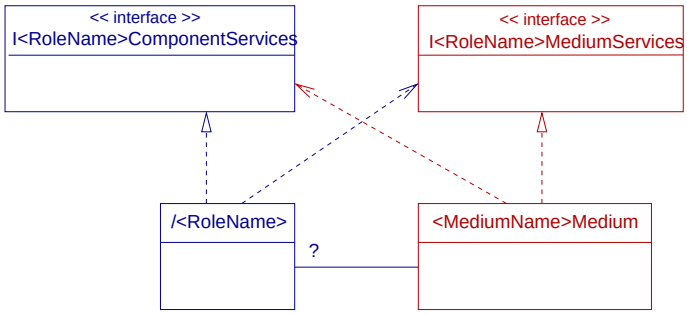


Fig. 3.13 – Relation générique entre un rôle et le médium

La figure 3.13 page précédente 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`[2]. Pour chaque rôle « <RoleName> » de composant, il existe deux 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 attributs de la classe `<MediumName>Medium` représentent les propriétés globales du médium. 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.

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. 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 appele 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 appararaître une référence sur l'appelant dans les signatures des services offerts et requis. Surtout que, comme nous le verrons

[2]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 `« medium »`.

dans le chapitre suivant, en fonction du niveau de spécification auquel nous nous plaçons, pour une même abstraction, 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 losrsque 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 similiaire 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.

Gestion de la dynamicité et du cycle de vie du 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

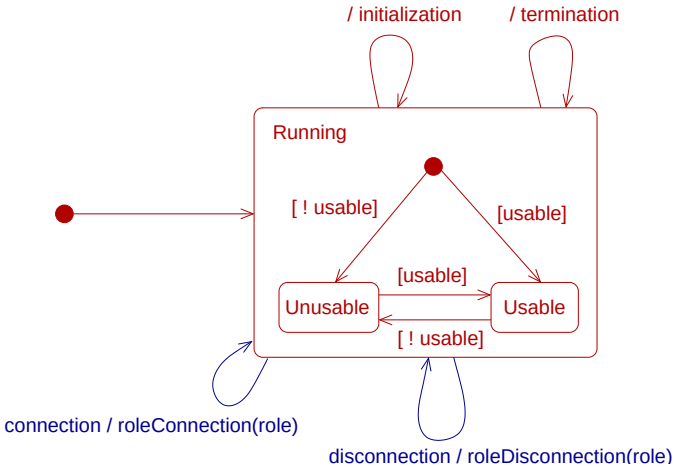


FIG. 3.14 – Diagramme d'état générique d'un médium

diagramme d'états de la figure 3.14. 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 trouvera 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 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 : `initialisation` et `termination`. Pour spécifier les actions, il suffit alors de définir la sémantique de ces opérations.

Le diagramme d'état 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`.

3.2.4 Un exemples de spécification de médium : un système de réservation

Nous spécifions un médium afin d'illustrer l'application de notre méthodologie de spécification de médiums. Nous avons choisi de spécifier un médium intégrant un système de vote (tel que celui utilisé dans l'application de vidéo interactive de la section 3.2.1 page 29). Ce médium, relativement complexe, permet d'étudier une spécification 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. Pour d'autres exemples, on pourra se reporter à [20, 21].

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 3.2.1 page 32. 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). 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. La paramètre 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 Le diagramme de collaboration du médium de réservation est représenté sur la figure 3.15 page suivante. 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`)

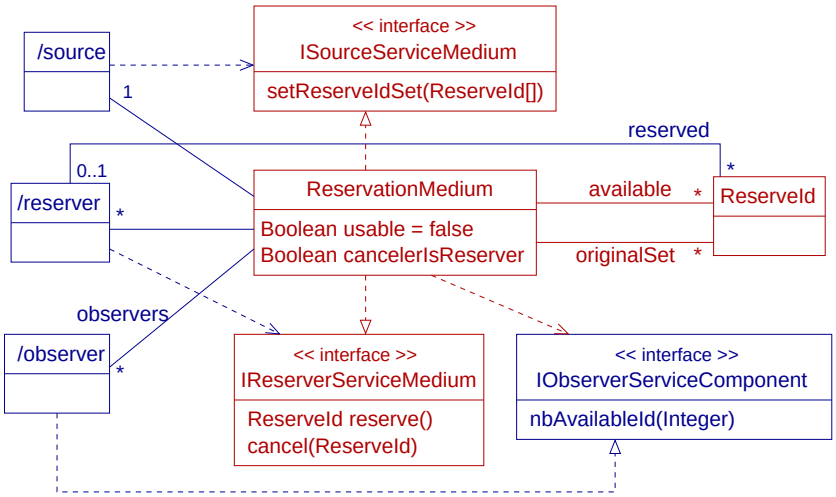


FIG. 3.15 – Diagramme de collaboration du médium de réservation

et l'interface requise (`IObserverComponentServices`) chez le rôle `observer`. Le nombre de rôles `observer` et `reserver` est quelconque (multiplicité « * ») alors que le nombre de rôle `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`.

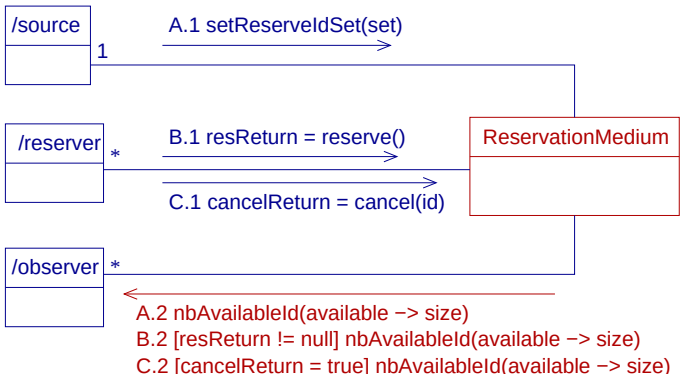


FIG. 3.16 – Vue dynamique de la collaboration du médium de réservation

La figure 3.16 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 la taille de l'ensemble des identificateurs disponibles.

Les trois services offerts peuvent amener à l'appel de cette opération. Plus précisement, 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 est appelée quand :

- L'initialisation de l'ensemble est fait par l'appel de `setReserveIdSet` (message A.2).
- Une réservation est faite par l'appel de `reserve` . 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`. 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 effctuées. 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 depart 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 reserver 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 teste la négation de ces conditions car cela concerne le cas ou l'opération retourne `false`.

Diagrammes d'état Nous allons avoir besoin d'utiliser le diagramme d'état générique du médium (voir la section 3.2.3 page 40). 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 3.16 page 43). 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'état 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.

Cet exemple complet de spécification de médiums montre la plupart des point importants de notre méthodologie : 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'état 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.

3.2.5 Architecture de déploiement d'un médium

Les gestionnaires : des *proxies* intelligents

Un médium réifie ou implémente un systéme 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 plus directement, ils passent par ces opérations. Une communication directe entre deux composants distants n'est jamais nécessaire si 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 3.2.1 page 29. Afin de réaliser l'interaction réifiée dans le médium, ces gestionnaires communiquent entre eux via un intergiciel ou toute autre technologie.

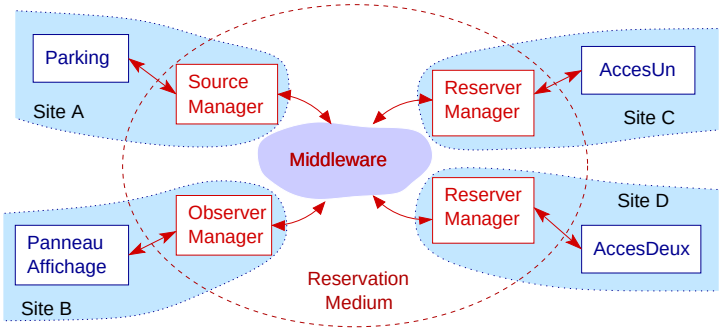


FIG. 3.17 – Architecture de déploiement du médium de réservation

La figure 3.17 montre l'architecture de déploiement du médium de réservation dans le cadre de l'application de gestion de places de parking (voir la section 3.2.1 page 32). Ce médium offre rôles différents donc trois gestionnaire 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 nécessaires sur ce rôle. Les gestionnaires peuvent être aussi complexes que cela est nécesssaire.

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 instantié en une seule fois mais un ensemble de gestionnaires distribués et logiquement cohérents. Si la connexion et la déconnexion dynamique 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 évoluer dynamiquement, en ce qui concerne pour ce qui est de la présence et du 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)[3]. 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. 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.

Gestion de la dynamicité et du cycle de vie des gestionnaires

Le diagramme d'état de la figure 3.18 page suivante représente le cycle de vie d'un gestionnaire, depuis son instantiation jusqu'à sa terminaison. Ce diagramme d'état est une évolution du diagramme d'état générique d'un médium (voir la section 3.2.3 page 40) 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 est instantié, il ne pourra pas faire partie du médium[4]. C'est pour cela que le gestionnaire est d'abord dans un état d'instantiation

[3]Dans la méthodologie de spécification, l'utilisabilté du médium est définie en fonction de l'état du médium et des rôles. 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.

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

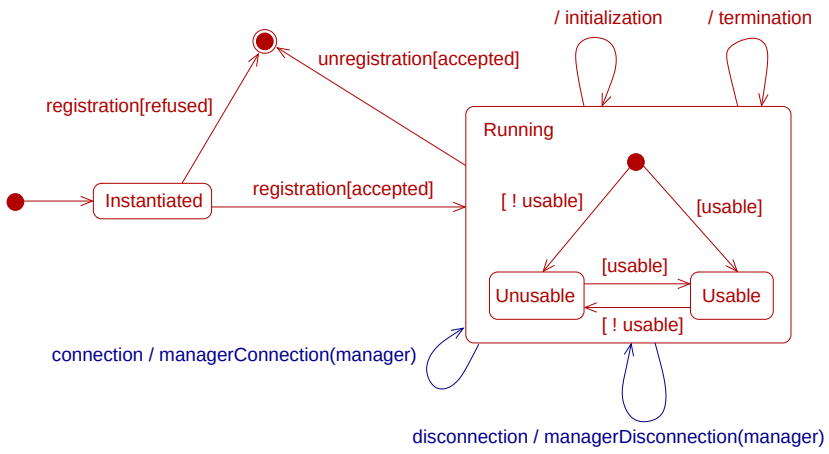


FIG. 3.18 – Diagramme d'état générique d'un gestionnaire de rôle

(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 inutisable, 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 deconnectés. 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'état (opérations `managerConnection` et `managerDisconnection` qui prennent en paramètre la référence sur le gestionnaire connecté ou déconnecté).

Ce diagramme d'état 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 la prochaine partie qui présente un processus de raffinement.

Un framework de développement de médium

Dans le cadre de sa thèse, E. Cariou à développé un framework en Java qui sert de plateforme d'expérimentation. Des services de communication de base sur TCP/IP et UDP/IP ont été développés ainsi que des classes de bases qu'il est relativement simple de spécialiser pour décrire de nouveaux médium.

composant associé est refusée

L'organisation du framework permet à plusieurs implantations du même médium de coexister, permettant ainsi de choisir différentes variantes algorithmiques ou non-fonctionnelles.

Le framework est sur `http ://www-info.enst-bretagne.fr/medium/framework/`

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

Le processus de raffinement que nous présentons dans ce chapitre à 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. Les étapes du processus sont les suivantes, dans l'ordre :

- Faire apparaître la classe représentant le médium comme l'aggrégation des gestionnaires de rôle.
- Définir des spécifications d'implémentations. 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.

Le processus de raffinement que nous présentons n'est pas complétement formel, au contraire de [54] 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. Néanmoins, 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.

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 3.2.4 page 42). 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* [58] de l'OMG car notre processus s'inscrit dans cette approche en en étant un exemple d'application.

Comme nous l'avons dit en introduction de cette partie, 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 en fonction de 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, 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.

Première étape : introduction des gestionnaires de rôle

Objet et principes de la première étape La première phase du processus de raffinement est de représenter le médium comme une agrégation de gestionnaires de rôle. Auparavant, dans la version abstraite, le médium était représenté par une unique entité. Or, 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 posséde 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 certain type 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 propriété 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 3.19 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.

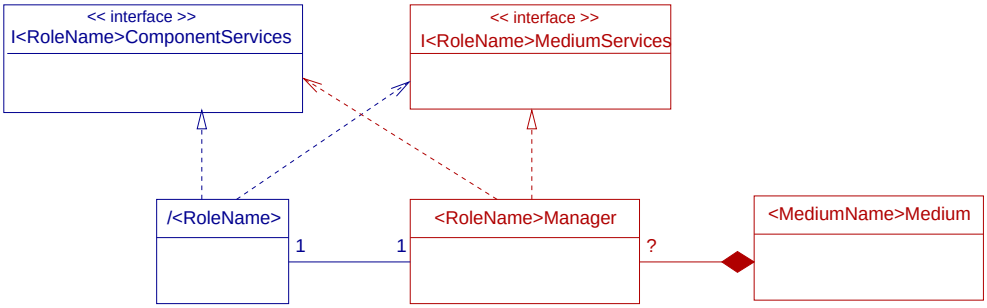


FIG. 3.19 – Relation générique entre un rôle, un gestionnaire et le médium

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'état.

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 consiste en 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 soit exprimées dans un autre contexte (celui du gestionnaire adéquat). Dans la spécification d'un service offert, si la référence vers un objet 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'état, il y a deux changements qui peuvent intervenir. Tout d'abord, la navigation et le contexte des contraintes OCL qui seraient utilisées dans un diagramme d'état. Elles doivent prendre en compte l'ajout des nouvelles classes comme nous venons de le voir. Ensuite, si le diagramme d'état générique d'un médium est utilisé, il est conservé en ce qui concerne la gestion de l'utilisabilité et de l'intialisation 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'état générique d'un gestionnaire (ce diagramme d'état est associé à la classe représentant le médium). Si un diagramme d'état était associé au médium, il reste associé au médium. Si un diagramme d'état 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'état 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 spé-

cifiant 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'état 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'état 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.

Deuxième étape : choix de conception

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 complétement 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 [26], 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 possible 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 [23], 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 spécifications : le diagramme de collaboration, les contraintes et spécifications OCL et les diagrammes d'état. La classe représentant le médium disparaît, les diagrammes d'état qui lui était associés doivent maintenant être modifiés et associés à des gestionnaires. Toutes ces modification 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 effectués 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 [35, 61]. Elles permettent de s'assurer que la spécification définie à un comportement similaire ou respectant celui de la spécification abstraite.

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 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. De même 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 moins si l'on se place dans un contexte où les communications entre les sites sont parfaites (elles aboutissent à chaque fois et se réalise 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.

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.

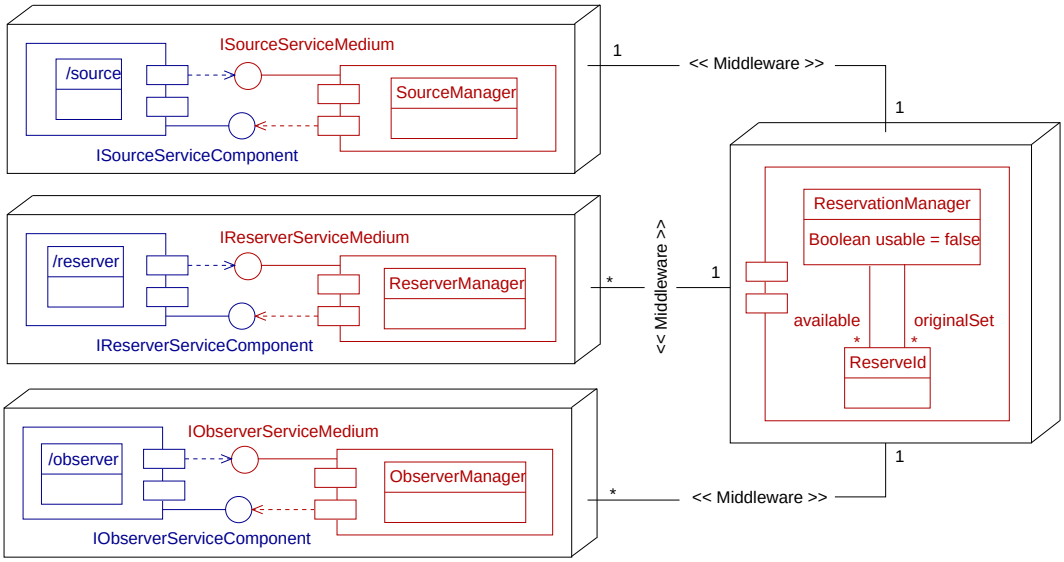


FIG. 3.20 – Déploiement de la version centralisée, avec le gestionnaire de réservation autonome

La figure 3.20 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 quatres types de sites qui peuvent être instantiés. Nous pouvons déterminer également les contraintes sur le nombre de ce 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 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 fait par l'intermédiaire de dépendances sur les interfaces de services offerts et requis par un gestionnaire[5].

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

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.

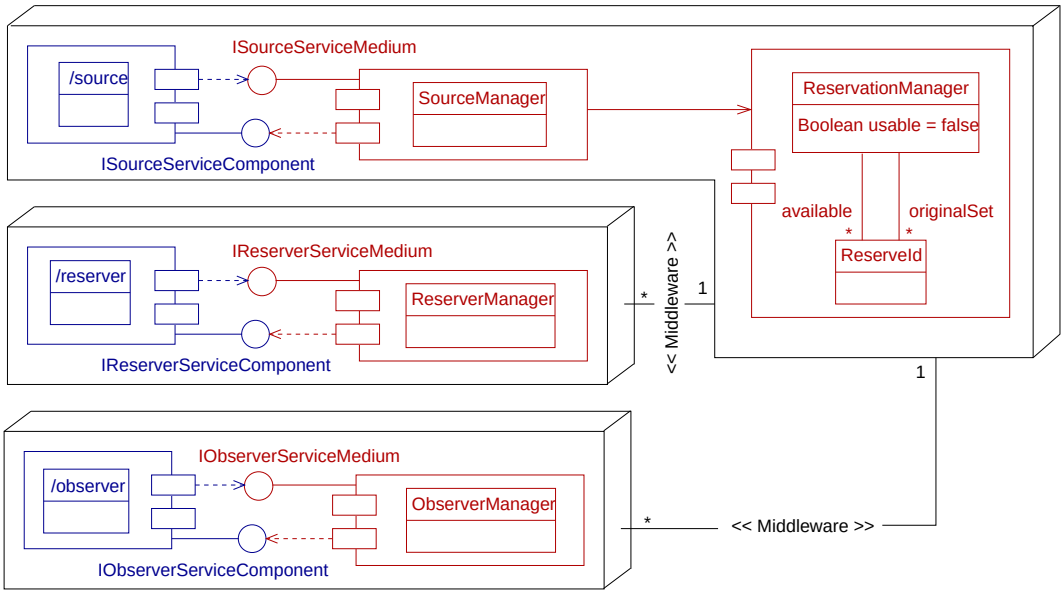


FIG. 3.21 – Déploiement de la version centralisée, avec le gestionnaire de réservation associé au gestionnaire de source

La figure 3.21 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.

Déploiement pour la gestion distribuée de la mémoire 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 3.22 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.

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

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

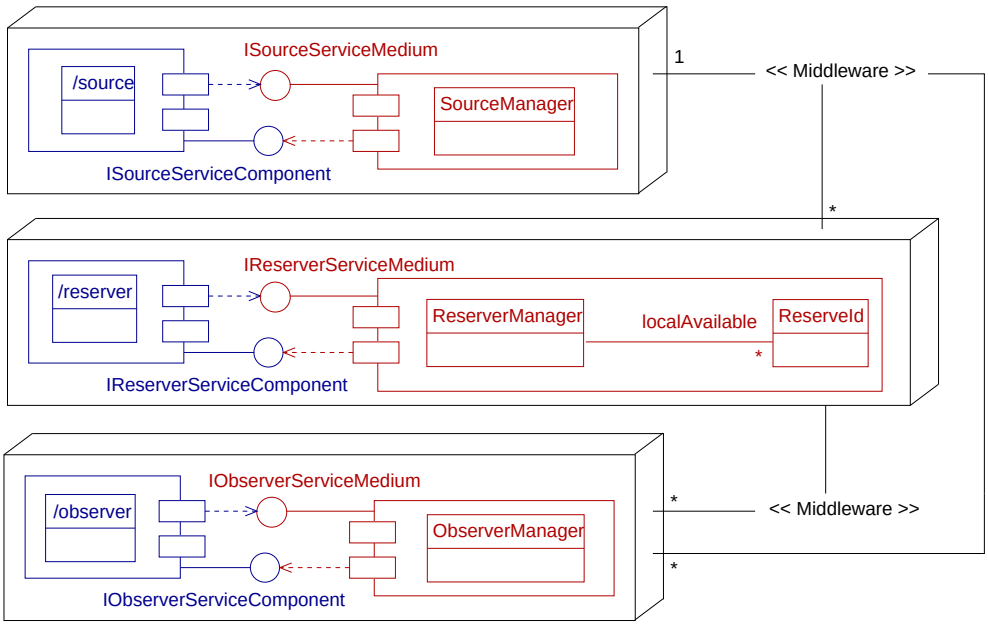


FIG. 3.22 – Déploiement de la version distribuée

médium qui correspond à la spécification abstraite, doit être respecté à tous les niveaux, autant que cela est 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 processsus consiste à faire apparaître, sur le diagramme de collaboration, la classe représentant le médium dans son ensemble comme l'aggrégation 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 dépendant de la plate-forme. Dans cette deuxième étape, nous avons donc affaire à une spécification de niveau PSM[6]. 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 complétement 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

[6]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.

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.

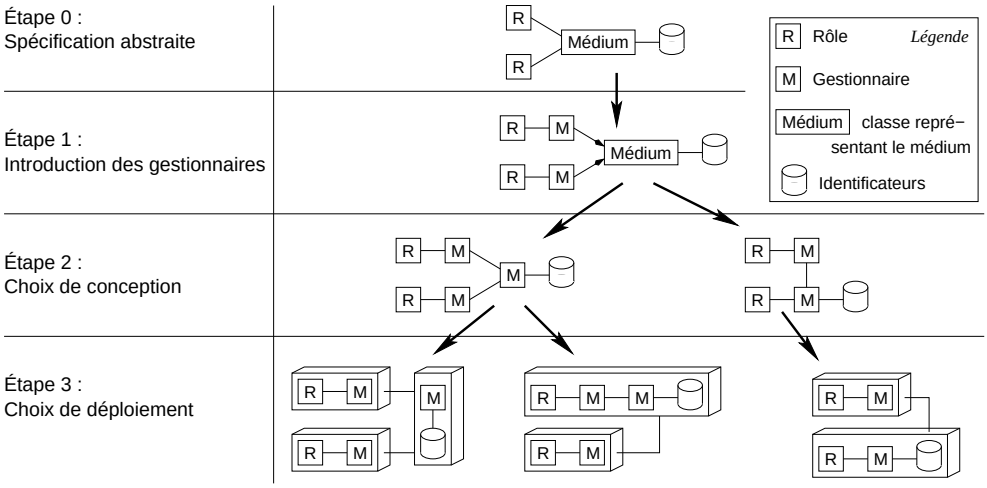


FIG. 3.23 – Les étapes du processus de raffinement pour le médium de réservation

La figure 3.23 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'aggrégation 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 d'implémentation et trois spécifications de déploiement. Pour toutes ses spécification et à tous les niveaux, les services offerts et requis par le médium n'ont pas changé, ni leur sémantique. La cohérence et l'unité de l'abstraction de communication réifiée par le médium a donc été conservée pendant 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 spécification 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.

3.2.7 Conclusion sur les médiums

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 cette partie, 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 composant (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.

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. La description de ce processus à été entamée dans la thèse d'E. Cariou [20] et se poursuit dans la thèse en cours de E. Kaboré. Ce processus s'appuie sur une démarche de type ingénierie des modèles (IDM) proposée par l'OMG. Nous y reviendrons dans la conclusion de ce mémoire (chapitre 4 page 70).

3.3 Connecteurs

3.3.1 Définition

Un connecteur est une réification d'un système d'interaction, de communication ou de coordination, il existe pour servir les besoins de communication des composants interagissant. Il décrit des interactions entre composants en se basant sur leur propre spécification d'interface. Il fournit des mécanismes d'interaction extra fonctionnels, indépendants de l'application. Le connecteur offre un moyen d'organisation pour décrire des connexions complexes, ceci rend leur interprétation plus claire et sans aucune ambiguïté. Deux points distinguent un connecteur d'un composant :

- C'est une entité autonome au niveau conception mais non déployable et non compilable (pas d'existence dans un langage de programmation conventionnel). Le connecteur doit être connecté pour être compilé puis déployé. Un composant, par contre, existe sous forme exécutable, et n'attend qu'à être déployé.
- Il ne spécifie pas de services offerts ou requis mais des patterns d'interaction. Ses extrémités (ou points d'accès) sont appelées prises. Ces prises représentent des interfaces d'application génériques qui lors de l'assemblage avec d'autres composants, seront transformés en proxies et s'adapteront aux types des ports de ces composants.

Nous distinguons quatre formes dans le cycle de vie d'un connecteur. Le tableau 3.1 les résume et les figures 3.24 et 3.25 page suivante les illustrent. Le connecteur évolue au fil du temps et nous le distinguons à chaque fois par une entité différente. Nous donnons une représentation particulière aux éléments du connecteur dans chacune des phases, avec à chaque fois des parties abstraites (partie pointillées) et d'autre concrètes (parties pleines).

Niveau État	Architecture (abstrait)	Implémentation (concret)
Isolé (sur étagère)	*Connecteur*	*Générateurs*
Lié	*Connexion*	*Composant de liaison*

Tab. 3.1 – Noms d'un connecteur en fonction du niveau d'abstraction et de son état

Fig. 3.24 – Le connecteur isolé : (a) abstrait, (b) implémenté

Dans son état le plus abstrait, figure 3.24 (a), cette entité est appelée un connecteur. La seule description concrète est la propriété à assurer par le connecteur ainsi que le nombre d'intervenants dans l'interactions envisagée (nombre de prises différentes) et leur multiplicité (le nombre de prises de même nature). Ce sont en effet les seuls invariant qui sont présents dans tout le cycle de vie. Les parties abstraites, à spécifier ultérieurement dans le cycle, sont les protocoles des plateformes sous-jacentes sur lesquels sera implémenté le connecteur, et les interfaces des composants interagissant (tous les deux en gris dans la figure).

La concrétisation d'un connecteur isolé est réalisée par une famille de générateurs sur étagère. Ces générateurs sont réalisés au-dessus de différents protocoles de communication ou plateformes.

Ils servent à générer les proxies (ou à transformer les prises en proxies) au travers desquels la connexion effective (physique) sera effectuée. La figure 3.24 page précédente (b) montre la concrétisation du connecteur. Dans cet état, la plateforme sous-jacente initialement abstraite se concrétise indépendamment des interfaces des composants à connecter qui restent abstraites.

FIG. 3.25 – Le connecteur assemblé : (a) abstrait, (b) implémenté

En restant abstrait, il est possible de relier un connecteur à des composants. Le connecteur n'est plus isolé, il est lié à d'autres composants et forme ainsi une *connexion*. À ce stade, les interfaces abstraites du connecteur abstrait (les prises) se concrétisent et s'adaptent aux interfaces applicatives (APIs) des composants interagissant. Dans la figure 3.25 (a), les cercles des prises se concrétisent car désormais leur type est spécifié. Ainsi, le connecteur qui exprime, à travers la propriété qui le caractérise, l'intention de la connexion, sert à relier et à connecter les composants sans se soucier de la plate-forme sous-jacente sur laquelle se feront les communications.

La forme la plus concrète d'un connecteur est représentée figure 3.25 (b). Le protocole de mise en œuvre est connu (par le générateur utilisé) et le connecteur est lié aux composants qui exposent leurs interfaces. Les proxies peuvent être générés pour satisfaire les exigences des composants par rapport à une propriété de communication sur une plateforme donnée. Cette forme est appelée composant de liaison en référence à ODP [62] où l'on parle plutôt d'objet de liaison.

Avec cette terminologie, la technologie CORBA met en œuvre un connecteur RPC, à l'aide de divers générateur (idl2java, idl2c, etc.) qui s'appuient sur le protocole IIOP. Les proxies générés permettent l'installation des objets de liaisons qui réalisent effectivement la connexion.

3.3.2 Application à l'équilibrage de charge

Certaines applications, comme les systèmes de commerce électronique, peuvent traiter en concurrence un très grand nombre de requêtes émises par un très grand nombre de clients. Un moyen efficace pour gérer cette grosse demande est d'augmenter le nombre de serveurs — des copies offrants le même service — et leur appliquer la technique de l'équilibrage de charge, où des mécanismes logiciels et matériels déterminent quel serveur devrait exécuter chacune des requêtes des clients. Le mécanisme d'équilibrage de charge distribue la charge de travail d'un client équitablement parmi n serveurs pour améliorer la vitesse de réponse globale du système. Ces mécanismes peuvent suivre l'une des deux stratégies : non adaptative ou adaptative.

Un exemple de stratégie non adaptative est la politique du Round Robin. Cette politique ne prend pas en compte la charge de travail du serveur, elle exécute simplement un algorithme qui permet de passer à un serveur suivant à chaque requête de client. On peut citer comme exemple de stratégie adaptative la politique du moins chargé qui, elle, prend en considération la charge de travail des serveurs suivant des métriques différentes. Elle peut, par exemple, prendre des informations d'exécution — comme le taux d'utilisation du CPU des serveurs — pour sélectionner le serveur qui traitera la requête. Il existe différents niveaux d'abstraction pour l'implantation du service d'équilibrage de charge dans le cadre des systèmes répartis. Nous pouvons les citer en partant du

plus bas niveau :

Niveau réseau : À ce niveau, l'équilibrage de charge est fourni par les routeurs IP et le système de nom de domaine (Domain Name Service) [36] qui permettent de servir un ensemble de machines hôtes.

Niveau système d'exploitation (OS) : L'équilibrage de charge est prise en charge par des systèmes d'exploitation distribués avec, éventuellement, des mécanismes de migration de processus [28].

Niveau Middleware (Intergiciel) : À ce niveau, un service de nommage peut être ajouté pour équilibrer la charge [70]. Une entité spéciale, appelée *load balancer* (équilibreur de charge) est ajouté entre les clients et les serveurs pour gérer les communications [59].

Niveau application : À ce niveau, les mécanismes d'équilibrage de charge sont mélangés avec le code des applications.

La réalisation de l'équilibrage de charge au niveau OS et réseau manque de flexibilité car il n'est pas possible d'utiliser des métriques liées à l'application au moment de l'exécution, lors de la prise de décisions sur l'équilibrage de charge [60]. Par exemple, il n'est pas possible de prendre en considération le nombre et le contenu des requêtes.

Réalisé au dernier niveau, le service d'équilibrage de charge permet plus de flexibilité. Cependant, il est explicite et constitue une partie intégrante de l'application ; son code est complètement mélangé avec le code fonctionnel des composants. Sa conception fait partie de la conception de l'application et il est ainsi difficile de réutiliser les composants sans ce service, ou bien de réutiliser ce service bien qu'il soit répétitif.

Afin de découpler l'application de l'équilibrage de charge, nous allons voir, dans la suite de cette section, la conception et la réalisation du service d'équilibrage de charge selon les deux abstractions présentées précédemment, les composants de communication et les connecteurs. Dans ces exemples, nous considérons une application répartie où un client a besoin d'un service bien défini offert par un serveur. Le client envoie un très grand nombre de requêtes auxquelles n serveurs identiques répondent.

Équilibrage de charge avec un composant de communication

La première méthode consiste à réaliser le service d'équilibrage de charge, en dehors de l'application, comme un protocole sous forme d'un composant de communication, comme décrit en 3.2 page 28. Le protocole possède des interfaces bien définies destinées à être utilisées explicitement par le client et les serveurs. Connaissant ces interfaces, il n'est pas utile de connaître son fonctionnement intérieur. On ne se préoccupe pas de la manière dont est implémenté le protocole et donc de la façon dont est réalisée la politique d'équilibrage de charge.

Pour cette solution, le composant de communication (LBMedium pour load balancing Medium) offre un service qui renvoie la référence du serveur élu, suivant la politique choisie, pour satisfaire les requêtes du client. D'un autre côté, il communique en permanence avec les serveurs (soit en mode *pull* ou *push*) pour récupérer leur état ou leur charge dans le cas d'une politique d'équilibrage de charge adaptative.

Le diagramme de séquence, figure 3.26 page suivante, montre les différentes interactions entre les composants « conventionnels » — le client et les serveurs — et le LBMedium. Chaque interaction représentée dans la figure 3.26 page suivante est décrite ci-dessous.

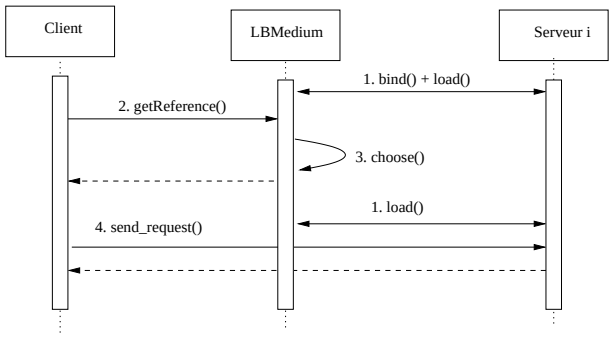


FIG. 3.26 – Equilibrage de charge avec un composant de communication

1. Les serveurs communiquent explicitement avec le LBMedium pour s'enregistrer et lui communiquer leur charge en mode *pull* ou *push*,
2. Le client appelle explicitement la méthode getReference() offerte par le composant de communication pour obtenir la référence du serveur qui traitera ses requêtes,
3. Connaissant tous les serveurs impliqués dans l'application, le LBMedium choisit la référence à transmettre suivant la politique et la renvoie au client.
4. Le client lance alors l'appel effectif relatif à la requête. Il établit une session avec le serveur choisi et lui envoie ses requêtes.

Cette approche préserve les avantages de l'équilibrage de charge niveau application. Elle permet d'utiliser des métriques qui sont liées à cette dernière mais, en plus, elle assure la séparation des responsabilités. Ceci rend la réutilisation, la maintenance et l'évolution des programmes plus faciles, la modification de la politique n'affecte ni le client ni le serveur. Cependant, il reste quelques inconvénients :

- Les services du composant de communication étant explicites, le service d'équilibrage de charge n'est transparent ni du côté du client ni du côté du serveur. L'interaction entre le protocole et l'application est totalement explicite, ainsi, le remplacement du LBMedium avec un autre composant présentant des interfaces différentes affecte les autres composants.
- Cette approche est efficace dans un mode de connexion par session, c'est-à-dire que le client envoie toutes ses requêtes au premier serveur dont il a reçu la référence par le composant de communication. Elle est moins adaptée au mode de connexion par requête, où le client peut changer de serveur lorsqu'il y a changement de charge de travail.

Équilibrage de charge avec connecteur

Une deuxième solution consiste à réaliser le service d'équilibrage de charge sous forme d'un connecteur associé à un générateur comme décrit dans la section 3.3.1 page 59. Cette approche permet d'assurer une meilleure transparence. Le connecteur permet de cacher toute la mécanique de l'équilibrage et même de cacher toute existence d'un tel service. Il existe juste pour transférer les requêtes du client aux serveurs suivant une politique particulière.

Le connecteur étant sur étagère, il ne spécifie que les types des intervenants (les prises) et leur multiplicité. Ici, on considère deux prises : un client et un serveur distants, le serveur étant de multiplicité n. Dans la phase implémentation, le générateur implémente, en plus des propriétés

de distribution comme RPC, les propriétés de l'équilibrage de charge. Une fois assemblé avec les composants client et serveurs, les prises sont transformées en proxies, PxC du côté du client et PxS du côté des serveurs. Ces derniers sont chargés de cacher les détails de distribution, de choix du serveur qui traitera les requêtes, et de la récupération de la charge. En effet, la surveillance de la charge de travail n'est plus de la responsabilité du serveur mais du proxy qui lui est attaché. De plus, les prises du connecteur étant anonymes au début, elles s'adaptent aux interfaces (ports) des composants interagissant.

Dans la figure 3.27, le diagramme de séquence montre l'enchaînement des événements au déploiement, après la génération des proxies décrits plus en détails en section 3.3.3.

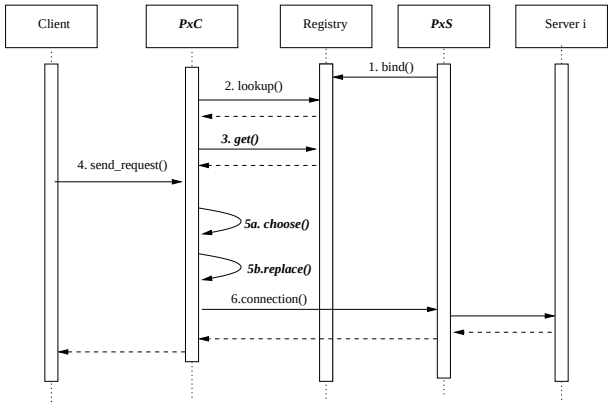


FIG. 3.27 – Equilibrage de charge avec un connecteur

Ainsi, utiliser le protocole d'équilibrage de charge dans un connecteur permet au client d'envoyer directement sa requête à un serveur pour obtenir le service souhaité sans se préoccuper de la politique d'équilibrage de charge. Les composants de l'application sont déchargés de tout code superflu. N'ayant pas une relation directe avec le service d'équilibrage de charge, leur maintenance, réutilisation et évolution sont plus aisées. La transparence est ainsi totale. Les composants ignorent l'existence d'un service d'équilibrage de charge et l'adaptation des services se fait automatiquement à l'assemblage, avec la génération des proxies.

3.3.3 Implémentation du connecteur d'équilibrage de charge

Nous avons réalisé le connecteur d'équilibrage de charge en spécifiant ses propriétés ainsi que les générateurs associés à déposer sur étagère. Les générateurs peuvent être implémentés en utilisant différentes technologies. On peut les créer à partir d'une extension d'un connecteur existant, comme RMI, ou bien en utilisant un autre composant. Dans la suite, nous allons voir les deux implémentations du connecteur d'équilibrage de charge, ainsi que les résultats d'une expérimentation.

Implémentation

La première implémentation est celle de la politique Round Robin illustrée en figure 3.27. Pour cette politique non adaptative, il n'est pas nécessaire de surveiller la charge de travail des serveurs. Nous avons réalisé le générateur comme une extension du générateur RMI. Le code de la politique d'équilibrage de charge est injecté dans le proxy généré (PxC). Ce dernier est chargé de récupérer

tous les serveurs impliqués dans l'application présents dans le registre de RMI. Le client envoie ses requêtes aux serveurs et le proxy les intercepte localement. Lorsque le proxy reçoit une requête du client, il exécute les mêmes opérations de codage et de décodage initialement prévues par le connecteur RMI. De plus, il a la responsabilité d'appliquer la politique de Round Robin. Il fait suivre chaque requête à la prochaine copie de serveur dans le groupe de serveurs enregistré dans le registre de RMI. Ceci se faisant en remplaçant la référence courante à chaque requête.

Nous avons fait d'autres expériences qui sont rapportées un stage de mastere [64]. Par exemple, un connecteur d'équilibrage de charge avec cette fois une politique d'équilibrage de charge adaptative *du moins chargé* a été implémenté. Dans cette politique, les requêtes des clients sont transmises aux serveurs en fonction de leur charge de travail. Pour l'implémenter et réaliser le générateur associé, nous avons utilisé un composant qui suit le modèle *publish/subscribe*. Ce composant reçoit les charges de travail des différents serveurs (les future proxies à générer (PxSi) possèdent cette responsabilité). Il implémente l'algorithme de la politique pour choisir le serveur le moins chargé qui fournira le service, et assigne la référence de ce serveur au proxy PxC du côté client qui établira la connexion. Nous combinons ce composant avec le connecteur RMI pour réaliser le générateur du connecteur. Le composant *Publish/Subscribe*, les proxies, et registre représentent désormais l'objet de liaison : le connecteur d'équilibrage de charge à l'exécution.

Les résultats de l'expérimentation du connecteur d'équilibrage de Charge

Dans le but d'évaluer tout le cycle de vie du connecteur, nous avons implémenté un connecteur d'équilibrage de charge dans le projet open source de Jonathan [52]. À la base, ce *framework* offre un connecteur RMI et CORBA. Nous avons ajouté des générateurs qui implémentent le connecteur d'équilibrage de charge avec deux variantes de stratégie : le *Round Robin* et le serveur le moins chargé.

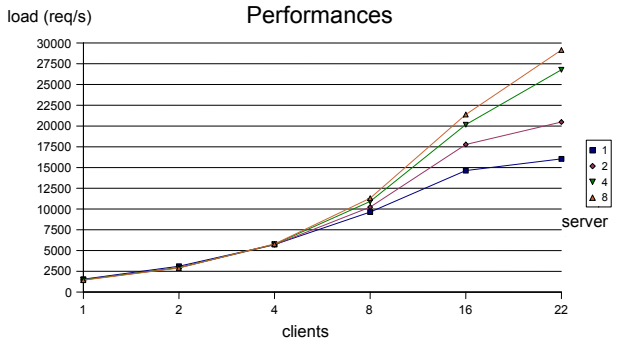


Fig. 3.28 – Performance de l'équilibrage de charge (en requête par seconde)

Afin d'évaluer la viabilité de cette approche, nous avons évalué les performances du nouveau connecteur sur un réseau de 32 machines avec 1 à 8 serveurs et 1 à 22 clients[7]. Les résultats de la Figure 3.28 montrent une très petite surcharge due à des changements de référence mais montrent une bonne *scalabilité*. D'autres résultats peuvent être consultés dans [64]. Il est à noter que le générateur de Jonathan peut toujours être utilisé pour réaliser des connexions point à point RMI classique. De ce fait, l'introduction de l'équilibrage de charge, ou le changement de la stratégie

[7]Une machine a été réservée au registre et une autre pour le contrôle du jeu de tests.

d'équilbrage de charge ne demande qu'une simple re-génération de proxies à partir de la même description d'interfaces.

3.3.4 Exemples de connecteurs

Pour illustrer ce que la notion de connecteur pourrait apporter nous allons dresser une liste de connecteurs qu'il pourrait être intéressant d'implanter sous la forme de familles de générateurs. Nous n'en proposons pas de spécification formelle, mais décrivons une *intention*.

Voici quelques propositions de connecteurs :

- Appel de Procédure (SOAP, CORBA, RMI, ...)
- Équilibrage de charge (Appel de procédure efficace)
- Consensus (Appel de procédure fiable) on peut l'imaginer basé sur une algorithme de consensus, ou sur des algorithmes plus simples du style « au moins 2 réponses identiques ».
- Adaptateur qui réalise des « transformations de type » (entier vers flottant, Fahrenheit vers Celsius, etc.)
- Coordinateur (Met en œuvre une coordination décrite par exemple par un langage de coordination de type Flo/C)

Pour chacun d'eux, de nombreuses variantes peuvent être envisagées, tant fonctionnellement en utilisant des variantes algorithmiques, qu'en fonction des cibles de déploiement.

3.3.5 Conclusion sur les connecteurs

Un connecteur est donc une sorte de compilateur capable de générer des composants et de les interconnecter à l'aide de primitives (comme les appels de procédures) ou de protocoles. Cette génération s'effectue à l'aide des descriptions des composants qui sont interconnectés et sous réserves que les propriétés d'« assemblabilité » soient respectées.

Pour définir une architecture logicielle on dispose donc de deux types d'entités logicielles : les composants (de communication ou non) et les connecteurs.

Le terme de connecteur est utilisé dans de nombreux contextes. Notre travail tente de clarifier son usage en proposant une définition précise liée à la structure et à la nature de cette entité, non pas à son usage dans une architecture. De plus, pour chacune des étapes du cycle de vie des connecteurs nous proposons des termes qui sont souvent employés les uns pour les autres (connecteur, connexion, objet de liaison).

Un connecteur n'est pas un protocole. Un protocole fait parti des composants de communications ; il offre des interfaces (API) explicites à ses clients. Un connecteur s'appuie sur un (ou des) protocole pour réaliser les interconnexions entre les composants que son générateur produit.

Un connecteur, au sens où nous l'avons défini, n'est pas spécifié par des interfaces qu'on appelle des rôles comme dans de nombreux ADL (Wright, C2, ACME, ...). La notion de prise recouvre parfois celle de rôle, mais est bien plus générale. Un rôle est la description d'une interface « générique » qu'on attache à l'interface explicite d'un composant. La notion de prise que nous défendons est décrite par une grammaire. Cette grammaire, peut, bien évidemment se réduire à la description de rôles, mais peut-être beaucoup plus générale que cela. Par exemple, les prise du connecteur CORBA sont décrites par la grammaire du langage IDL. Un connecteur est donc un compilateur qui exploite les grammaires décrivant les prises.

Le but de la mise en évidence de la façon d'implanter un connecteur sous la forme d'un générateur est d'inviter les concepteurs de logiciel à développer des générateurs variés de connexions. Il

nous semble qu'actuellement pour assembler un système les concepteurs n'ont à leur disposition, et donc n'utilise, que le connecteur RPC. En conséquence, pour réaliser des systèmes aux propriétés non fonctionnelles complexes, les solutions consistent à développer « à la main » l'équivalent des composants de liaison. Nous pensons que le développement de générateur devrait permettre de capitaliser du savoir faire sous la forme d'outils de générations de code qui, s'ils existent parfois, ne sont pas identifiés comme des connecteurs, ni utilisés comme tel.

3.4 Conclusion sur les abstractions de communication

L'assemblage de logiciels est un vieux problème. Les techniques d'édition de liens ont été développées très tôt avec les compilateurs. Parmi les problèmes difficiles on peut citer :

- La capacité à compiler de manière séparée. Ceci induit pour le concepteur de choisir les informations (données, fonctions) qu'il n'a pas à décrire sachant qu'elles seront fournies par ailleurs. Mais, ce choix doit être détectable pour que lors de l'assemblage, toutes les informations manquantes soient trouvées. En général, le compilateur produit des tables (de symboles) qui seront exploitées par l'éditeur de lien.
- La correction du système assemblé. Une fois les informations regroupées, l'éditeur de lien produit soit des liens physiques statiques lorsqu'il dispose d'une vision globale des logiciels, soit des procédures qui permettent de trouver l'information dynamiquement au cours de l'exécution. L'objectif est la « compilabilité » du système. La sûreté et la correction, au sens introduit en 2 page 6, sont en général laissés à d'autres outils (*model checker*, *theorem prover*, plateforme de tests, etc.). En effet, le code source contient rarement les informations suffisantes pour atteindre ce niveau de contrat ; pas ou peu d'assertions, pas ou peu de descriptions des propriétés temporelles et non-fonctionnelles souhaitées.

Nous faisons ce parallèle avec les compilateurs pour introduire deux réflexions sur l'architecture logicielle et l'assemblage de composant. Premièrement, le choix des frontières est un processus critique de l'architecture logicielle. Deuxièmement, la solution du traitement des systèmes complexes réside dans la découverte des bonnes abstractions, même si, à la fin, nous n'obtenons que des 0 et des 1 !

3.4.1 Bien choisir les frontières

Choisir les frontières entre entités logicielles, c'est définir les interfaces et les liens entre ces entités. Des règles de bons sens telle que « faible couplage » entre entités et « forte cohérence » d'une entité sont de précieux guides. Ces idées ont conduit à la notion d'objet. Un objet est cohérent et responsable de ses données. Un objet est couplé aux autres et offre et demande des services. Toutefois, découvrir les « bons » objets reste un problème difficile, et nombre de « bons » objets ont un couplage fort avec certains de leurs collègues.

Prenons l'exemple classique des philosophes chinois. Quelle doit être l'interface des philosophes ? Voici quelques variantes (pour simplifier, on suppose que les actions de manger et penser ont une durée aléatoire) :

1. Ils pensent, ils mangent,
2. ils pensent, ils prennent les deux baguettes, ils mangent, ils relâchent les deux baguettes,
3. ils pensent, ils prennent une baguette (x2), ils mangent, ils relâchent une baguette (x2),

4. ils pensent, ils prennent la baguette de droite, ils prennent la baguette de gauche, ils mangent, ils relâchent la baguette de droite, ils relâchent la baguette de gauche,

Le choix n'est pas simple. Et tous peuvent mener à une solution qui fonctionne. Pourtant, certaines solutions sont meilleures que d'autres. Par exemple, est-il normal que chaque philosophe soit responsable de l'algorithme de choix des baguettes ?

Si, dans un soucis d'anthropomorphisme compréhensible, on répond oui, le système construit risque de créer des famines et des interblocages ! En effet, chacun utilisera sa stratégie et rien ne garantit la correction du système. Par contre, si l'on délègue la responsabilité de l'algorithme à une autre entité, chargée d'assurer la correction du système, et qu'on dispense les philosophes de réfléchir à l'algorithme d'utilisation des baguettes, la solution est beaucoup plus élégante. On peut, par exemple, remplacer un philosophe par un autre sans risque pour le système.

Dans la première variante, la correction est immédiate ; pourvu que la fonction mange *fournie* soit correcte. Dans la deuxième, il faut s'assurer que l'appel aux fonctions respecte un automate simple. Dans les autres variantes, les preuves sont très complexes et pourraient devoir être refaites à chaque arrivée ou départ d'un nouveau philosophe : situation incontrôlable dans le cas d'un système ouvert.

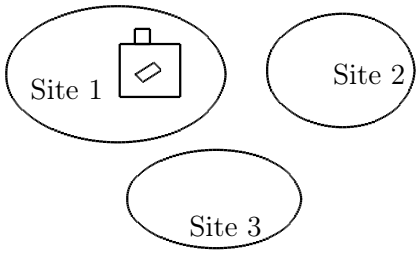
Les composants de communication permettent d'imaginer une solution simple au problème d'exclusion mutuelle des philosophes en regroupant toute la logique dans le médium, laissant aux philosophe le temps de réfléchir aux choses vraiment importantes.

Le modèle de composant que nous proposons n'a pas d'équivalent actuellement. La plupart des implantations proposent des composants dont tous le code est localisé en un unique site (cas a) de la Figure 3.29 page suivante, ou alors s'ils permettent une implantation répartie, imposent de n'avoir qu'une interface accessible à partir d'un unique site (cas b). Des travaux de F. Guidec au VALORIA proposent de construire pour des composants Fractal une solution qui rend accessible l'interface du composant sur plusieurs sites (cas c). Le modèle des médiums permet une implantation distribuée avec plusieurs interfaces (rôles) accessibles de divers sites (cas d). Les implantations des services peuvent être très différentes. Par exemple, le médium solution du problème des philosophe pourrait exister avec différents algorithmes ; la nature même du médium - et son implantation - forcent tous les philosophes (autour de la même table) à utiliser le même algorithme.

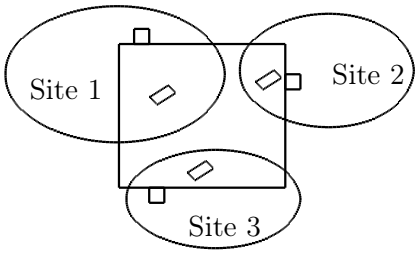
Les composants de communication permettent de s'abstraire des frontières physiques de l'infrastructure du système. Un composant abstrait n'est contraint par aucune frontière physique. Mais son implantation, et son développement, sont plus complexes pour atteindre cet objectif.

3.4.2 Pourquoi des connecteurs ? tout est composant !

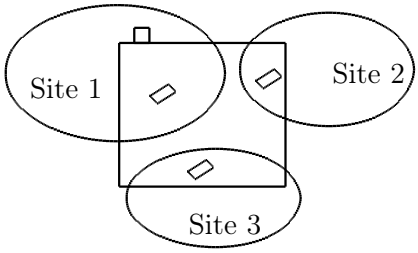
Quand on observe le résultat d'une compilation, d'autant plus si celle-ci est optimisée, la structure initiale du code source n'est plus forcément visible ; il ne reste que des instructions d'assembleur, des suites de 0 et de 1. Est-ce pour autant que les structures abstraites telles que les procédures, les types de données sont inutiles ? Bien évidemment, non. Dans le cas d'un système à base de composants, le phénomène est identique. Quand on observe un assemblage, on voit des composants qui se rendent mutuellement des services. Tout est composant. Tous les composants sont reliés par



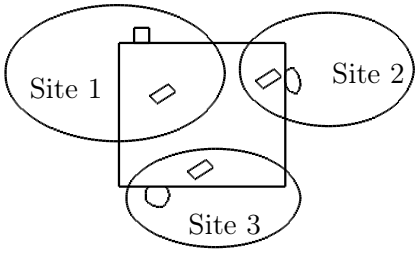
a) composant mono-site, interface simple



c) composant multi-site, interface simple répartie



b) composant multi-site, interface simple



d) composant multi-site, interface répartie

FIG. 3.29 – 4 modèles de composants

des appels de procédures (à distance ou non). Pourtant les façons dont ces composants ont été obtenus peuvent-être très différentes. Certains ont été réutilisés tels quels, d'autres ont été fabriqués, générés, pour ce système spécifique au moment de l'assemblage. Les talons d'une architecture à base d'appel de procédure à distance sont des exemples de « composants générés ».

Nous pensons que l'identification des connecteurs comme outil d'assemblage spécifique des architectures logicielles permet de mettre en évidence ce besoin spécifique d'assemblage. Elle ne permet bien évidemment pas de faire plus qu'avant ; nous obtenons après tout toujours que des machines de Turing, mais le processus de construction, de programmation de la machine, est différent.

Le but de la mise en évidence de la façon d'implanter un connecteur sous la forme d'un générateur est d'inviter les concepteurs de logiciel à développer des générateurs de connexions variés. Il nous semble qu'actuellement, pour assembler un système, les concepteurs n'ont à leur disposition, et donc n'utilise, que le connecteur RPC. En conséquence, pour réaliser des systèmes aux propriétés non fonctionnelles complexes, les solutions consistent à développer « à la main » l'équivalent des composants de liaison.

Le rôle du générateur qui met en œuvre le mécanisme de liaison est donc multiple :

- Vérifier la correction de l'assemblage. En fonction de l'information qu'il peut collecter, ou qu'on lui fournit, et des objectifs qu'on lui assigne. Il vérifie la « connectabilité » des intervenants. On peut s'appuyer sur les niveaux de contrat pour classifier l'ambition de la vérification.
- Générer la mécanique d'assemblage. En fonction de la nature de la liaison qui doit être réalisée, le générateur produit l'ensemble des composants nécessaires au fonctionnement du système. Pour une même nature de liaison, il peut y avoir plusieurs architectures ou algorithmes solutions.

Le processus de production des systèmes de composants assemblés doit s'appuyer sur des éditeurs de liens abstraits que nous appelons des connecteurs.

À partir de ces travaux sur l'expression des contrats de composants et leur exploitation par des connecteurs, nous proposons dans le chapitre final une synthèse et des perspectives de travaux de recherche.

Chapitre 4

Conclusion : des pistes de recherche autour de l'assemblage

Les travaux présentés dans ce mémoire tentent d'élaborer une vision générale et abstraite du problème de l'assemblage de logiciels. La démarche de recherche consiste à chercher de bonnes abstractions et à confronter leur utilisation à certains aspects ou points délicats. Nous avons travaillé sur deux problèmes en particulier.

Le premier est un problème qui paraît simple et réglé depuis longtemps comme l'appariement de méthodes entre un objet client et un objet serveur. Nous avons montré que la diversité des interprétations actuelles de la surcharge et de la redéfinition rendait, dans l'état actuel des technologies, l'assemblage de composants à objet très difficilement prévisible quand les langages utilisés sont différents. Nous pensons que pour résoudre ce problème, il faudra sans doute accepter de réduire la capacité d'expression des contrats syntaxiques en limitant les utilisations de la redéfinition à des cas invariants et en supprimant la surcharge[1].

Le second problème concerne la nature et la définition des connecteurs utilisés en architecture logicielle. Nous avons mis en évidence deux entités logicielles de nature différente mais souvent confondues et utilisées comme « *connecteur* ». Nous avons montré en quoi ces objets sont différents structurellement, dans la manière de les concevoir et de les utiliser. Le modèle de composants que nous présentons généralise ceux habituellement développés en permettant non seulement une implantation répartie, mais également des interfaces réparties par rôle. Enfin, l'identification de la notion de connecteur comme l'entité logicielle abstraite chargée de vérifier l'assemblage et de produire le code nécessaire d'assemblage, permet d'envisager de nouveaux travaux.

4.1 Sur la notion de contrat

La notion de contrat est maintenant largement répandue. Si les contrats syntaxiques sont utilisés par nécessité, les contrats des niveaux supérieurs sont encore peu exploités. Il existe pourtant de nombreux outils de spécification et d'exploitation de ces spécifications. Étant utilisés à des niveaux d'abstraction et à des phases du développement différents, les efforts faits pour s'assurer de la validité des spécifications peuvent être perdus lorsque, suite à des décisions de conception, le système est

[1] Sauf peut-être dans certains cas particuliers comme les constructeur Java qui sont déjà des cas particulier de méthodes.

raffiné. Pour remédier à cela, une piste qui nous semble intéressante est d'étudier la traçabilité des contrats dans le processus de conception d'un système.

La thèse en cours de C. Kaboré aborde ce type de problème en s'appuyant sur la notion de transformation de modèle introduite par l'OMG. Toutefois, si les principes semblent prometteurs, la compréhension de la notion de transformation en est à ses balbutiements, ainsi que leurs techniques de définition et de mise en œuvre. Tout cela dans le contexte de modélisation UML qui, comme il est pudiquement et officiellement dit, possède de nombreux points de variations sémantiques qui ne simplifient pas les choses (et en particulier la détection d'hypothèses cachées sur le sens de telle ou telle notation.)

Pour réaliser ces études, nous pensons qu'il est intéressant de restreindre le cadre de ces transformations à des objets particuliers comme les composants de communication. En effet, comme on l'a vu, la spécification de ces objets est très contrainte ainsi que la cible visée (un ensemble de gestionnaires communicants.) De plus de nombreux phénomènes caractéristiques des systèmes distribués viennent enrichir la réflexion comme les problèmes de synchronisation, de simultanéité, de fiabilité, de réplication, etc.

Ce travail devrait donc permettre de mieux comprendre comment exprimer un contrat, ce qu'est une transformation ainsi que ses conséquences sur les contrats de spécification.

4.2 Sur la notion de composant de communication

4.2.1 Spécification

La spécification des abstractions de communication s'appuie actuellement sur une description UML. Ceci n'est que partiellement satisfaisant. La sémantique imprécise d'UML ne permet pas une description rigoureuse. De plus, les propriétés spécifiques de distribution (localisation, causalité, a/synchronicité, ...) ne peuvent pas être simplement exprimées. Une des pistes abordées dans le cadre du projet ACCORD est l'utilisation de contrats de spécifications qui étendent l'utilisation standard du langage de contrainte OCL et des règles de conformité qui leur seraient associées. On envisage aussi d'utiliser des techniques de spécification basées sur une logique épistémique [65] pour traiter la répartition des données.

Nous pensons qu'une partie du raffinement de la spécification abstraite d'un composant de communication pourrait se traduire par la transformation de formules de logique épistémique. Cette approche pourrait permettre de *vérifier des transformations* de modèle liées à la distribution des données d'un composant de communication.

4.2.2 Plateforme cible

La plateforme cible actuelle est un framework écrit en Java qui repose sur le mécanisme de communication IP. Elle a servi à démontrer la faisabilité de l'approche et son extensibilité. Faute de ressources, nous n'avons pas pu la faire évoluer. Voici deux améliorations importantes que nous avons envisagées :

- Un point faible de l'implantation actuelle est l'existence d'un « registry » qui centralise toutes les informations concernant l'application distribuée et les règles d'assemblage de ses composants. Ce registry pourrait être réalisé comme un composant de communication, réparti et robuste.

– Des outils de supervisions pourraient être développés pour observer les composants de communication d'une application : leur création, leur destruction, l'arrivée et le départ des gestionnaires, etc. Des alertes pourraient être levées en cas de violation de contrat...

4.2.3 Transformations

Ayant une spécification et une cible, dans un cadre bien délimité, l'étude des transformations (raffinements) de la spécification vers la cible nous semble intéressante. Les premières étapes ont été esquissées par E. Cariou dans sa thèse. À partir de cette base, nous prévoyons des transformations qui permettent de distribuer plus ou moins les données ou d'implanter des algorithmes distribués dédiés à la sécurité, la robustesse, la gestion de la causalité, des reprises en cas de panne, etc. en fonction des choix de conception.

Pour étudier les transformations et sélectionner les plus pertinentes dans le cadre de la distribution, nous envisageons de partir de la spécification de types abstraits de données, en les considérant comme des données partagées élémentaires qui permettent à des processus ou des composants répartis de communiquer par leur intermédiaire. Les types concernés pourraient être simples (booléen, nombre) ou complexes (pile, liste, arbre, tableau, etc.) Il s'agirait ensuite d'étudier leur répartition en respectant l'architecture cible définie dans nos médiums de communication. Les variantes de répartition étudiées pourraient couvrir les cas :

Centralisé La donnée est effectivement gérée par un seul contrôleur du composant, les autres contrôleurs n'agissent alors que comme des proxies vers la donnée. Cette transformation est a priori la plus simple à mettre en œuvre.

Réparti La donnée (complexe) est répartie physiquement sur plusieurs contrôleurs (avec éventuellement des variantes de répartition dépendantes de la nature de l'objet). Les contrôleurs garantissent le respect des propriétés du type abstrait et agissent comme des proxies pour la partie de donnée non locale.

Répliqué La donnée est copiée sur tous les contrôleurs et ceux-ci coopèrent pour garantir la cohérence de la donnée.

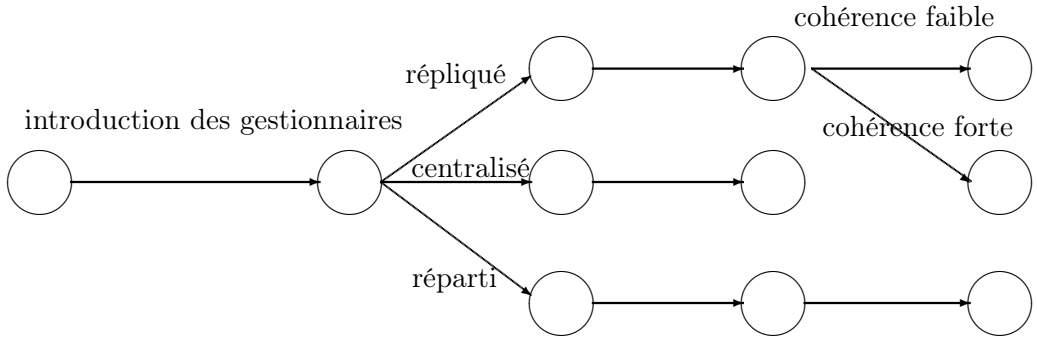


Fig. 4.1 – Un arbre de transformations appliquées à un médium.

Les règles de transformation s'inspireront certainement d'algorithmes existants de répartition de

données ou de gestion de la cohérence. L'utilisation de ces algorithmes comme règle de transformation de modèles permettrait leur réutilisation au sein d'atelier de modélisation[2], et pourrait automatiser la création de variantes d'implémentation d'abstractions de communication. La figure 4.2.3 page précédente montre comment à partir d'une même transformation on peut aboutir à différentes variantes d'implantations, toutes fonctionnellement équivalentes et donc substituables les unes aux autres, mais avec des propriétés non-fonctionnelles différentes.

Le cadre des médiums de communication nous semble particulièrement adapté à l'étude de transformation de modèles alors que ces techniques sont encore émergentes. En effet, le point de départ des transformations (la spécification abstraite) et l'architecture de la cible (l'ensemble des contrôleurs répartis communicants entre eux) délimitent fortement les transformations possibles.

4.3 Sur la notion de connecteur

4.3.1 Spécification

Un travail important reste à faire sur la spécification des connecteurs et des prises. Si un connecteur peut être implanté comme un compilateur et un éditeur de lien, les langages de description des prises restent à étudier ; quelles sont les bonnes abstractions ? quelles sont les bonnes structures ? Les quelques exemples actuels (CORBA, CCM, RMI, WSDL, BPEL4WS) peuvent donner des orientations, mais ne sont pas vus comme les connecteurs que nous proposons. Des langages dédiés sont à inventer qui intégreraient plusieurs niveaux de contrats (syntaxique, sémantique, pragmatique) et seraient exploités par des outils qui s'assureraient de la compatibilité de ces contrats. Le projet RNTL ACCORD ouvre des perspectives dans ce sens.

Des travaux parallèles concernant les langages de coordination comme (FLO/c) pourraient également s'intégrer à cette réflexion. En effet, on peut considérer qu'un connecteur a pour but de synthétiser une coordination entre les interfaces des composants qu'il assemble. Un générateur « universel » pourrait être un compilateur qui analyse l'assemblage puis produit les composants de connexion qui correspondent à une interaction décrite dans un langage de coordination et passée en paramètre au générateur « universel ».

4.3.2 Implémentation

Les objets produits par un générateur associé à un connecteur peuvent l'être selon différentes stratégies. Il est probable que, comme pour les composants de communication, les schémas de transformations (génération) soient différents en fonction des propriétés non-fonctionnelles recherchées. Ce niveau de variation supplémentaire des générateurs s'apparente à une séparation des préoccupations et pourrait s'enrichir des recherches sur les aspects ; chaque générateur « universel » choisissant une stratégie de génération fonction de propriétés non-fonctionnelles recherchées.

La spécification des connecteurs et l'implantation de leurs incarnations sous forme de générateurs nous semblent un champ de recherche prometteur.

[2]Comme OpenTool de TNI-valiosys, Objecteering de Softeam, MTL de l'équipe Trikell de l'IRISA ou tout autre atelier permettant la manipulation de modèles UML.

4.4 À la croisée des chemins

4.4.1 Méthode, démarche, architecture

Les médiums doivent s'insérer dans une démarche de conception par assemblage de composants et compte tenu de leur architecture ils forment un élément clé pour la description de la distribution. Ils pourraient permettre de construire des unités de déploiement et donc être à la base d'un langage de déploiement qui reste à créer.

Dans l'étude des démarches à base de composant, un des problèmes majeur demeure l'identification des « frontières » et en conséquence des « bons » composants. Montrer que les composants ne sont pas contraints par la répartition de l'architecture physique est une étape vers l'acceptation de nouvelles frontières dans les architectures logicielles. Un axe de recherche en génie logiciel pourrait être de trouver des règles qui caractérisent les bons composants, et qui permettent de définir, par exemple, des métriques de composabilité ou de substituabilité. Dans le cadre de composants à base d'objet, l'utilisation de la redéfinition ou de la surcharge a, comme on l'a vu, un impact certain sur ces deux qualités de composants.

4.4.2 Déploiement

À la croisée des problèmes de spécification, d'architecture et de plateformes cible, le déploiement des « abstractions de communication » est un point sans doute crucial. En effet, si l'on sait déployer du code à distance - code mobile, réseaux actifs, aspects dynamiques - la gestion de la cohérence globale des codes interagissant est généralement traitée manuellement. Il nous semble intéressant de chercher, en s'appuyant sur des abstractions de communication, des moyens d'exprimer des contraintes de déploiement, au travers sans doute de langages ou de contrats de déploiement. Une esquisse de tels contrats a été expérimentée dans le cadriciel médium réalisé par E. Cariou lors de sa thèse à l'aide de « contrats de médium » qui décrivent des conditions d'utilisation du composant réparti.

4.4.3 Intégrer générateur et contrat

Le lieu naturel d'intégration de nombreuses vérifications sémantiques des assemblages de logiciels est le connecteur. Il est tentant d'intégrer les techniques de vérification de systèmes concurrents dans les analyses et vérification que le compilateur/générateur peut faire. En fonction du niveau des contrats décrits, en fonction des contraintes exprimées et en fonction de la nature des connecteurs, le générateur pourrait fabriquer un nouveau composant dont le contrat serait lui même résultat de la compilation. Cette forme de « propagation » de contrat pourrait ainsi permettre de construire, de proche en proche, des assemblages de composants dont le niveau de sûreté serait calculé.

Nous pensons qu'attacher des techniques de vérification aux connecteurs permettrait d'*organiser* les vérifications des systèmes de grande taille. On pourrait être amené à choisir un connecteur en fonction du type de propriété que l'on souhaite vérifier ou mettre en place. L'ensemble d'un système étant le résultat d'un assemblage de composants via de multiples connecteurs, ceux-ci offrent une organisation de la stratégie de vérification.

4.4.4 Liaison dynamique des langages à objets

Comme nous l'avons montré, l'interprétation « naturelle » de la liaison dynamique est loin de faire consensus. La recherche d'un langage objet consensuel vis-à-vis de la liaison dynamique pourrait être entamée, dans le cadre d'UML et de l'Action Semantics, par exemple. Mais le volume de code existant et les enjeux économiques sont tels que le succès d'une telle entreprise est peu probable.

Mais dans tous les cas, faire des transformations de modèle (ou de source) requiert une définition précise de la liaison dynamique (choisie parmi celles possibles). D'autres expériences d'assemblages de composants dans des contextes plus hétérogènes que .NET pourraient être menées pour espérer mieux comprendre les interactions de langages.

L'industrie du logiciel se trouve devant un conflit d'intérêt. Doit-elle imposer des restrictions sur l'usage de certains concepts de langage pour pouvoir fabriquer des outils de vérification plus puissants et plus fiables, ou doit-elle conserver la liberté des programmeurs au risque de ne pas arriver à construire d'outils de vérification réellement utilisables ? L'abandon progressif de l'assembleur (qui reste utile dans certains cas très spécifiques) ou du GOTO ont montré que la résistance au changement est grande, mais pas impossible !

4.5 Conclusion de la conclusion

Pour conclure, notre travail est un retour aux sources. En cherchant à s'abstraire des contraintes physiques des architectures matérielles et en raisonnant sur des architectures logicielles indépendantes de l'infrastructure, nous avons mis en évidence de nouveaux types de composants, et proposé une définition claire de ce que sont des connecteurs et quels sont leurs rôles dans l'assemblage des composants. L'identification de ces entités informatiques nous ramène au vieux problème de l'édition de lien, de la construction de tables de symboles, et de la vérification de la « composabilité » des binaires. Dans les architectures logicielles, les tables de symboles sont des contrats, plus riches, plus abstraits. Ils sont aussi plus complexes à exploiter. L'éditeur de lien est, quant à lui, remplacé par le connecteur et ses générateurs qui embarquent tous les outils nécessaires à la vérification des contrats et à la génération de la « connectique logicielle ».

Nous pensons qu'il y a deux grandes activités dans la production de logicielle : le processus de développement de composants (dont les composants de communication) et le processus de développement d'éditeur de liens/générateurs. La nature des entités logicielles, leur rôle, leurs façons d'être utilisées en font des objets d'étude très différents ... mais complémentaires ! Jusqu'à présent, les efforts se sont essentiellement portés sur les premiers, nous espérons avoir contribué à l'étude des seconds.

Les pistes de recherche que nous envisageons couvrent un spectre large. Nous nous plaçons dans une perspective d'intégration de nombreux travaux : algorithmique distribuée pour les transformations de composants de communication, techniques de vérification au travers des contrats et de leur exploitation par les connecteurs, transformation de modèles et sémantique des langages. Si faire avancer la recherche c'est souvent « creuser un sillon », comme me le disait l'un de mes mentors, mettre en perspective des travaux nous semble aussi une façon de faire de la recherche en informatique.

Table des matières

1 Introduction **3**

2 Les contrats d'assemblage **6**
2.1 La notion de contrat 7
2.2 Un cas particulier de contrat : la redéfinition de méthode 10
2.2.1 Description du test 11
2.2.2 Une sémantique « raisonnable » 12
2.2.3 Résultats et analyses 13
2.3 Peut-on réaliser des composants avec un langage à objets ? 19
2.3.1 Composant et contrat 19
2.3.2 Assemblage de composants 20
2.4 Conclusion sur les contrats d'assemblage 23

3 Les abstractions de communication **25**
3.1 Deux types d'abstraction de communication 25
3.1.1 Représentation des connexions par des traits 26
3.1.2 Composant de communication et connecteur 27
3.2 Composant de communication 28
3.2.1 Vers une meilleure utilisation des abstractions de communication 29
3.2.2 Définition des composants de communication 36
3.2.3 Méthodologie de spécification de médiums en UML 37
3.2.4 Un exemples de spécification de médium : un système de réservation 41
3.2.5 Architecture de déploiement d'un médium 46
3.2.6 De la spécification abstraite à l'implémentation : définition d'un processus de raffinement 49
3.2.7 Conclusion sur les médiums 58
3.3 Connecteurs 59
3.3.1 Définition 59
3.3.2 Application à l'équilibrage de charge 60
3.3.3 Implémentation du connecteur d'équilibrage de charge 63
3.3.4 Exemples de connecteurs 65
3.3.5 Conclusion sur les connecteurs 65
3.4 Conclusion sur les abstractions de communication 66
3.4.1 Bien choisir les frontières 66
3.4.2 Pourquoi des connecteurs ? tout est composant ! 67

4 Conclusion : des pistes de recherche autour de l'assemblage **70**
4.1 Sur la notion de contrat 70
4.2 Sur la notion de composant de communication 71
4.2.1 Spécification 71
4.2.2 Plateforme cible 71
4.2.3 Transformations 72
4.3 Sur la notion de connecteur 73
4.3.1 Spécification 73
4.3.2 Implémentation 73
4.4 À la croisée des chemins 74
4.4.1 Méthode, démarche, architecture 74
4.4.2 Déploiement 74
4.4.3 Intégrer générateur et contrat 74
4.4.4 Liaison dynamique des langages à objets 75
4.5 Conclusion de la conclusion 75

Table des figures

2.1 Méta-modèle ACCORD pour les contrats 9
2.2 Comment se comportera ce modèle ? 10
2.3 Comment se comportera cet assemblage ? 20

3.1 Connexion simple. 26
3.2 Connexion complexe. (a) plusieurs appelants - un appelé, (b) un appelant - plusieurs appelés 27
3.3 Deux types d'abstraction de communication. (a) composant de communication, (b) connecteur 27
3.4 Détail de l'architecture de l'application de vidéo interactive 29
3.5 Réorganisation des éléments d'interaction dans l'application de vidéo interactive . . . 31
3.6 Nouvelle architecture de l'application de vidéo interactive 31
3.7 Architecture de déploiement de l'application de vidéo interactive 32
3.8 Application de gestion de places de parking 33
3.9 Une première spécification de la collaboration de gestion de parking 34
3.10 Collaboration générique de gestion de réservation d'identificateurs 34
3.11 Nouvelle description de l'application de gestion de places de parking 35
3.12 Description de l'application de réservation aérienne 35
3.13 Relation générique entre un rôle et le médium 38
3.14 Diagramme d'état générique d'un médium 41
3.15 Diagramme de collaboration du médium de réservation 43
3.16 Vue dynamique de la collaboration du médium de réservation 43
3.17 Architecture de déploiement du médium de réservation 46
3.18 Diagramme d'état générique d'un gestionnaire de rôle 48
3.19 Relation générique entre un rôle, un gestionnaire et le médium 51
3.20 Déploiement de la version centralisée, avec le gestionnaire de réservation autonome . 54
3.21 Déploiement de la version centralisée, avec le gestionnaire de réservation associé au gestionnaire de source 55
3.22 Déploiement de la version distribuée 56
3.23 Les étapes du processus de raffinement pour le médium de réservation 57
3.24 Le connecteur isolé : (a) abstrait, (b) implémenté 59
3.25 Le connecteur assemblé : (a) abstrait, (b) implémenté 60
3.26 Equilibrage de charge avec un composant de communication 62
3.27 Equilibrage de charge avec un connecteur 63
3.28 Performance de l'équilibrage de charge (en requête par seconde) 64
3.29 4 modèles de composants 68

4.1 Un arbre de transformations appliquées à un médium. 72

Bibliographie

[1] Franz inc., 2001. http ://www.franz.com.

[2] Functional objects, inc. http ://www.fun-o.com/, 2001.

[3] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM Software Engineering Notes*, 18(5) :9–20, 1993.

[4] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Software Eng. and Methodology*, 6(3) :213–249, July 1997.

[5] D. Ancona, S. Fagorzi, and E. Zucca. A calculus for dynamic linking. In Springer, editor, *ICTCS'03 - Italian Conference on Theoretical Computer Science*, Lecture Notes in Computer Science 2841, 2003.

[6] D. Ancona and E. Zucca. Sound and complete inter-checking (the very essence of principal typings). Technical report, Universita di Genova, 2004.

[7] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, 2000.

[8] Antoine Beugnard. Compiler, c'est reconstruire le temps. *Bigre*, (78) :95 – 113, mars 1992.

[9] Antoine Beugnard. *ArMor : Une méthode de spécification opérationnelle*. Thèse de doctorat, Université de Rennes 1, mars 1993.

[10] Antoine Beugnard. Comparison of various oo languages relatively to their late-binding semantics. Web page : http ://perso-info.enst-bretagne.fr/ beugnard/papiers/lb-sem.shtml, 2000-2005.

[11] Antoine Beugnard. OO languages late-binding signature. In *The Ninth International Workshop on Foundations of Object-Oriented Languages*, Portland, Oregon, January 19 2002.

[12] Antoine Beugnard. Une comparaison de langages objet relative au traitement de la redéfinition de méthode et à la liaison dynamique. In *Langages et Modèles à Objets, LMO 2002*, pages 99 – 113. Hermes, 2002.

[13] Antoine Beugnard. Peut-on réaliser des composants avec un langage à objets ? In *Langages et Modèles à Objets, LMO 2005*, pages 47 – 60. Hermes, 2005.

[14] Antoine Beugnard, Olivier Caron, Jean-Philippe Thibault, and Bruno Traverson. Assemblage de composants par contrats, le modèle de composants accord. *L'Objet*, To appear 2005/6.

[15] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, pages 38 – 45, July 1999.

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

[17] Jan Bosch. *Design & Use of Software Architecture. Adopting and Evolving a Product-Line Approach.* Addison Wesley, 2000.

[18] Martin Büchi and Wolfgang Weck. The greybox approach : When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, August 1999. http ://www.abo.fi/~mbuechi/publications/TR297.html.

[19] Adeline Capouillez, Pierre Crescenzo, and Philippe Lahire. Le modèle ofl au service du méta-programmeur – application à java. *L'objet, LMO'2002*, pages 11 – 24, 8 2002.

[20] Eric Cariou. *Contribution à un Processus de Réification d'Abstraction de Communication.* Thèse de doctorat, Université de Rennes 1, juin 2003.

[21] Eric Cariou and Antoine Beugnard. *Ingénierie des Composants : Principes et Fondements*, chapter 10 : Les Composants de Communication. Vuibert, 2005.

[22] Eric Cariou and Antoine Beugnard. The specification of UML collaboration as interaction component. In J.M. Jézéquel, H. Hussmann, S. Cook, editor, *«UML» 2002 – The Unified Modeling Language*, volume 2640 of *LNCS*, pages 352 – 367, Dresden, Germany, September 30 - October 4, 2002. Springer Verlag.

[23] 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)*, 2002.

[24] Eric Cariou, Antoine Beugnard, and Jean-Marc Jézéquel. An archictecture and a process for implementing distributed collaborations. In *The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2002)*, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland, September 17 - 20 2002.

[25] Guiseppe Castagna. Covariance and contravariance : Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3) :431–447, March 1995.

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

[27] Dominique Colnet and Suzanne Collin. SmallEiffel the GNU eiffel compiler. http ://www.loria.fr/projets/SmallEiffel/, 2001.

[28] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems : Concepts and Design.* Harlow, England : Pearson Education Limited, 2001.

[29] Iain D. Craig. *Programming in Dylan.* Springer, 1996.

[30] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A fragment calculus towards a model of separate compilation, linking and binary compatibility. In *Logic in Computer Science*, pages 147–156, 1999.

[31] Roland Ducournau. Spécialisation et sous-typage : thème et variations. *Technique et Science Informatique*, 21(10) :1305–1342, 2002.

[32] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Fundamentals of Computation Theory*, pages 127–140, 1983.

[33] Adele Goldberg and David Robson. *Smalltalk-80 : The Language and Its Implementation.* Addison-Wesley, 1983.

[34] M.M. Gorlick and R.R. Razouk. Using weaves for software construction and analysis. In *13th Int'l Conf. Software Eng. (ICSE13)*, pages 23–34, May 1991.

[35] 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.

[36] Cisco Systems Inc. High availability web services, 2000. `http ://www.cisco.com/warp/public/cc/so/neso/ibso/ibm/s390/mnibm_wp.htm`.

[37] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future : The story of squeak, A practical smalltalk written in itself. In *Conference Proceedings of OOPSLA '97, Atlanta*, volume 32(10) of *ACM SIGPLAN Notices*, pages 318–326. ACM, October 1997.

[38] M. Jackson. *System Development*. Prentice Hall, Englewood Cliffs, 1983.

[39] B. Jacobs. Object-oriented programming oversold ! OOP criticism and OOP problems, the emperor has no clothes ! reality check 101, 2001. http ://www.geocities.com/SiliconValley/Lab/6888/oopbad.htm.

[40] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Trans. Software Eng.*, 21(4) :336–355, April 1995.

[41] Selma Matougui. *Contribution à un la définition de la notion de connecteur (titre provisoire)*. Thèse de doctorat, Université de Rennes 1, prévu fin 2005.

[42] Selma Matougui and Antoine Beugnard. How to implement software connectors ? a reusable, abstract and adaptable proposal. In *5th IFIP international conference on Distributed Applications and Interoperable Systems*, Athens, Greece.

[43] Selma Matougui and Antoine Beugnard. Two ways of implementing software connections among distributed components. In *International Symposium on Distributed Objects and Applications*, Agia Napa, Cyprus, Oct 31 - Nov 2 2005.

[44] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor. A language and environment for architecture-based software development and evolution. In *21st Int'l Conf. Software Eng. (ICSE '99)*, pages 44–53, May 1999.

[45] B. Meyer. Applying "design by contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10) :40–52, oct 1992.

[46] B. Meyer. *Object-Oriented Software Construction, second edition*. Prentice-Hall, 1997.

[47] Bertrand Meyer. *Eiffel : The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[48] Bertrand Meyer. *Object Oriented Software Construction, Second Edition*. Prentice Hall, 1997.

[49] Bertrand Meyer. Interactive software engineering. http ://eiffel.com/, 2001.

[50] Bertrand Meyer. Overloading vs. object technology. *Journal of Object-Oriented Programming*, pages 3 – 7, October/November 2001.

[51] Microsoft. Microsoft .NET, 2001. http ://www.microsoft.com/net.

[52] ObjectWeb Open Source Middleware. Jonathan : an Open Distributed Objects Platform. `http ://jonathan.objectweb.org/`.

[53] M. Moriconi and R.A. Riemenschneider. Introduction to sadl 1.0 : A language for specifying software architecture hierarchies. Technical Report SRI-CSL-97-01, SRI Int'l, Mars 1997.

[54] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4) :356–372, April 1995.

[55] OMG. site mda. http ://www.omg.org/mda.

[56] OMG. OMG Unified Modeling Language Specification, version 1.3. `www.omg.org`, June 1999.

[57] OMG. OMG Unified Modeling Language Specification, version 1.3. `http ://www.omg.org/cgi-bin/doc ?ad/99-06-08`, June 1999.

[58] OMG. Model Driven Architecture. `http ://www.omg.org/mda`, December 2002.

[59] O. Othman, C. O'Ryan, and D. Schmidt. The design of an adaptive corba load balancing service. *IEEE Distributed Systems Online*, 2(4), April 2001.

[60] O. Othman, C. O'Ryan, and D. Schmidt. An efficient adaptive load balancing service for corba. *IEEE Distributed Systems Online*, 2(3), March 2001.

[61] 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.

[62] J.R. Putman. *Architecting with RM-ODP*. Prentice Hall, 2001.

[63] D. Remy and J. Vouillon. Objective ML : An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1) :27–50, 1998.

[64] Francisco Javier Iranzo Sanchez. Calcul de performance sur une application de répartition de charge. rapport de mastère, ENST-Bretagne, Brest, France, Août 2004.

[65] Lionel Seinturier. *Conception d'algorithmes répartis et de protocoles réseaux en approche objet*. PhD thesis, CNAM-CEDRIC, Décembre 1997.

[66] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Software Eng.*, 21(4) :314–335, April 1995.

[67] G.L. Steele. *Common Lisp - The Language*. Digital Press, 1990.

[68] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[69] Clemens Szyperski. *Component Software*. Addison-Wesley, 1999.

[70] IONA Technologies. Orbix 2000. `http ://www.iona.com/products/orbix2000_home.htm`.

[71] Bruno Traverson, Bernard Rougeot, Philippe Desfray, Jean marc Jezequel, Gérard Florin, Laurence Duchien, Elie Najm, and Antoine Beugnard. Rapports du projet ACCORD. Délivrables, RNTL, 2002-2003.

[72] Stephen S. Yau and Fariaz Karim. Integration of object-oriented software components for distributed application software development. In *FTDCS*, pages 111–118, 1999.

[73] Young Ran Yu, Soo Dong Kim, and Dong Kwan Kim. Connector modeling method for component extraction. In *APSEC*, pages 46–53, 1999.