

RIO : Rôles, Interactions et Organisations

une méthodologie pour les systèmes multi-agents ouverts

THÈSE

présentée et soutenue publiquement le 2 Décembre 2003

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité Informatique)

par

Yann Secq (Yann.Secq@lifl.fr)

Composition du jury

Président :	Jean-Marc Geib, LIFL, Université des Sciences et Technologies de Lille
Rapporteurs :	Yves Demazeau, LEIBNIZ, Institut IMAG, Université de Grenoble Jacques Ferber, LIRMM, Université de Montpellier II
Examinateurs :	François Bourdon, GREYC, Université de Caen René Mandiau, LAMIH, Université de Valenciennes
Directeur :	Philippe Mathieu, LIFL, Université des Sciences et Technologies de Lille
Co-Directeur :	Jean-Christophe Routier, LIFL, Université des Sciences et Technologies de Lille

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Laboratoire d'Informatique Fondamentale de Lille — UMR 8022
U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 20 43 47 24 – Télécopie : +33 (0)3 20 43 65 66 – email : `direction@lifl.fr`

Vous pouvez trouver la version la plus récente de ce document à l'adresse suivante : http://www.lifl.fr/~secq. Ce document a été mis en page et composé avec LaTeX et la classe thlifl.

Remerciements

Il est toujours délicat de remercier l'ensemble des personnes ayant contribué à cet achèvement que représente l'aboutissement d'un travail de thèse. Je vais pourtant m'y essayer en demandant aux personnes qui auront été oubliées de ne m'en tenir trop rigueur.

Je tiens à remercier Yves Demazeau et Jacques Ferber d'avoir accepté de rapporter mon document de thèse et de m'avoir apporté leur regard sur mon travail. Je tiens aussi à remercier François Bourdon et René Mandiau d'avoir bien voulu examiner ce document, ainsi que Jean-Marc Geib d'avoir accepté la présidence du jury de thèse.

Je tiens aussi à remercier l'équipe Smac au sein de laquelle j'ai trouvé un environnement intellectuel et humain d'une rare richesse. Plus particulièrement, je tiens à remercier Philippe Mathieu, mon directeur de thèse, et Jean-Christophe Routier, mon co-directeur, pour l'ensemble du travail de formation et d'accompagnement qu'ils ont effectué tout au long de ces quatre années. Que cela soit sur les aspects liés à la recherche, les aspects pédagogiques, ou encore les aspects politiques et humains, leurs expériences et leur enseignements m'ont été très importants. Je tiens aussi à remercier les autres membres de l'équipe, Bruno Beaufils et Sébastien Picault, avec qui j'ai pu partager de nombreuses discussions, toujours chaleureuses et délicieusement mouvementées. Je remercie aussi Jean-Paul Delahaye, qui avait très tôt cultivé ma curiosité à travers ses passionnants articles dans "Pour la science", puis il a persévéré lors de son option sur "L'histoire de l'Informatique" en Deug et surtout lors de nos premiers contacts avec un projet de Maîtrise consacré à l'étude des jeux de taquins.

Je tiens à remercier l'ensemble du personnel du Lifl et de l'Iut qui m'a accueilli et permis de m'épanouir dans mon travail de recherche, d'enseignement et qui m'a permis de m'impliquer au sein de ces différentes organisations. Je tiens notamment à souligner le travail de toutes les personnes qui chacune à leur niveau participent à la vie de ces établissements à la fois pour les étudiants, mais aussi pour les personnels.

Bien entendu je n'aurais pu atteindre cet objectif sans l'aide continuelle des enseignants qui, au long de mon apprentissage, m'ont transmis la passion de leur matière, et sûrement aussi le bonheur de partager ses connaissances grâce à l'enseignement. Je remercie aussi les étudiants à qui j'ai dispensé des cours, et surtout ceux dont j'ai eu l'occasion d'encadrer les projets, qui je l'espère ont été aussi enrichissants pour eux qu'ils l'ont été pour moi. Je voudrais par ailleurs souligner le rôle fondamental et actuellement totalement mésestimé du Cies, à qui revient la lourde tâche de transmettre les rudiments de la pédagogie en à peine 20 jours de formation.

Cependant, je n'aurais pu réussir mes études sans ma famille, et je tiens ici à remercier mes parents qui depuis mon plus jeune âge ont toujours fait leur maximum, en consacrant temps et argent, pour m'éveiller et m'encourager dans mes passions. Merci Maman de m'avoir transmis la passion de la lecture, que je considère comme la clé de la connaissance, et merci Papa de m'avoir toujours poussé dans mes intérêts scientifiques. Cet environnement a très sûrement guidé et favorisé ma progression et mon épanouissement. Néanmoins, il n'aurait peut être pas suffi à m'amener vers les métiers de la recherche sans l'aide d'un ami proche de la famille, Djimédo Kondo, qui pendant de nombreuses années m'a aidé, suivi et encouragé à travailler les mathématiques. Avec le recul, je crois

bien que, plus encore que les nombreuses heures passées ensemble à m'exercer à résoudre des problèmes, ce sont les multiples discussions sur la science et ses développement plus récents tels que la théorie du chaos, qui ont constitué mon éveil et mon attrait pour la recherche scientifique.

Finalement, je voudrai terminer cette liste qui ne se veut en aucun cas exhaustive, par ma compagne, Nathalie, qui depuis le DEUG m'a suivi tout au long de ce périple, m'a soutenu et compris dans des choix pas toujours faciles. Qu'elle soit ici remerciée de sa patience, et de sa compréhension, car il n'est pas toujours aisé de vivre avec une personne totalement investie dans son travail, et n'arrivant pas toujours à faire la part des choses.

Bien heureusement, la période la plus difficile est maintenant passée et que tout ceux qui m'ont permis de la surmonter soient ici remerciés.

Lille, le 11 Novembre 2003.

"Sitôt que j'eus achevé tout ce cours d'études au bout duquel on a coutume d'être reçu au rang des doctes, je changeai entièrement d'opinion ; car je me trouvais embarrassé de tant de doutes et d'erreurs qu'il me semblait n'avoir fait autre profit, en tâchant de m'instruire, sinon que j'avais découvert de plus en plus mon ignorance"

Discours de la méthode, Première partie, Descartes.

Résumé

L'objectif que nous avons poursuivi tout au long de nos travaux est d'identifier les concepts fondamentaux des systèmes multi-agents distribués à gros grain, et de proposer des modèles génériques pour ces concepts. Plus précisément, nous avons répondu à la diversité des propositions en terme de modèles cognitifs d'agent et de modèles organisationnels. Pour cela, nous proposons d'aborder le problème de l'interopérabilité selon deux axes : l'identification et la définition d'une infrastructure minimale et générique facilitant le développement de différents modèles d'agents, et la proposition d'un modèle de spécification de protocoles d'interaction s'appuyant sur le principe de la coordination par standardisation[86]. Ces deux axes tendent à uniformiser les infrastructures des systèmes multi-agents au niveau des fonctionnalités et des garanties que les concepteurs peuvent en attendre.

Notre travail s'est ainsi articulé autour des quatre points suivants :

- un modèle d'agent minimal générique,
- une réification des protocoles d'interactions entre agents,
- une réification de la notion d'organisation au sein des systèmes multi-agents,
- une proposition de méthodologie d'analyse et de conception de systèmes multi--agents ouverts.

Ces différents aspects ont été abordés à la fois dans le but d'identifier les notions fondamentales qui constituent les systèmes multi-agents, mais aussi dans un souci d'ingénierie de ces systèmes. Ainsi, en plus de l'interopérabilité qui constitue le pivot de nos travaux, l'ingénierie des systèmes produits a été une préoccupation continue lors de nos phases de réflexion et d'implémentation.

Ces travaux ont montré que l'utilisation de notre modèle d'agent minimal générique facilite le développement de systèmes multi-agents et apporte d'intéressantes fonctionnalités aussi bien au niveau système qu'au niveau applicatif. En outre, le modèle de spécifications exécutables de protocoles d'interaction aide le concepteur à gérer et à structurer les interactions d'un système, en lui permettant de décrire les interactions dans leur globalité. Finalement, notre démarche méthodologique se base sur ce modèle d'interaction pour proposer une approche de conception incrémentale de systèmes multi-agents ouverts.

Mots-clés: Conception orientée agents, ingénierie des protocoles d'interaction, méthodologie basée sur les rôles, les interactions et les organisations

Abstract

The aim that we have followed during our work is the identification of fundamental concepts for distributed coarse grained multi-agent systems, and to propose generic models for these concepts. More precisely, we tried to bring an answer to the diversity of agent and organizational models. We propose to tackle the interoperability problem through two aspects : the identification and the definition of a minimal generic infrastructure to ease the development of different agent models, and the proposition of a model for runnable specifications of interaction protocols based on the principle of coordination by standardization[86]. These two aspects lead to a uniformization of multi-agent systems infrastructures at the functional level and on the guarantees that designers can expect.

Our work relies on the following propositions:

- a minimal generic agent model,
- the reification of interaction protocols between agents,
- the reification of the notion of organizations within multi-agent systems,
- a proposition of a methodology for the analysis and design of open multi-agent systems.

These aspects have been studied to identify the fundamental notions of multi-agent systems, but also to improve the engineering of these systems. Thus, if interoperability constitutes the heart of our work, the engineering of produced systems has always been in mind during our reflexions and implementations.

These works have shown that the use of our minimal generic agent model eases multi-agent systems development, and brings interesting features both on the system and applicative level. Moreover, the model of runnable specifications of interaction protocols helps the designer to manage and structure the interactions of a system, while enabling him to describe a bird's-eye view of interactions. Finally, our methodology relies on this model of interaction to propose an incremental design of multi-agent systems.

Keywords: Multi-Agent systems modeling, interaction protocol engineering, minimal agent model, agents organizations

Table des matières

Table des figures

Introduction

Problématique

L'évolution des réseaux informatiques, suite au développement du marché des ordinateurs personnels et d'Internet, a amené de profondes modifications quant à la manière de concevoir et de développer des applications. La notion d'application est passée de celle d'un programme s'exécutant sur un ordinateur, à un ensemble de programmes répartis sur un réseau de machines. Ainsi, l'environnement logiciel qui s'apparentait à un ensemble de *programmes autistes* ne pouvant communiquer entre eux, tend à devenir un ensemble de *systèmes ouverts* en *interaction* les uns avec les autres. Concevoir de tels systèmes distribués pose alors le problème de la gestion des interactions. Cela revient à définir comment les différentes composantes du système peuvent s'identifier, à décrire leurs fonctionnalités, à spécifier les interactions possibles entre ces entités, et à préciser les modalités leur permettant d'entrer en contact les unes avec les autres.

Les systèmes multi-agents sont des systèmes constitués d'un ensemble d'entités autonomes appelées agents. Ils peuvent apporter des réponses qui aident à maîtriser la complexité des systèmes distribués. Ce domaine est né à l'intersection de domaines tels que l'intelligence artificielle distribuée, la représentation des connaissances, la théorie des actes de langages ou encore la vie artificielle. Et bien que la notion de système multi-agents soit assez ancienne, elle n'a connu un fort développement scientifique que depuis la fin des années 1980. L'avènement d'Internet, et plus encore l'apparition du commerce électronique, a amené les industriels à s'intéresser plus particulièrement à cette approche. D'un autre côté, la recherche publique a aussi produit de nombreux systèmes, sans qu'aucune définition consensuelle de "*ce qu'est un agent*" apparaisse.

Enjeux et contributions

Le domaine des systèmes multi-agents, bien qu'établi maintenant, réussi difficilement à percer au niveau industriel. Plusieurs raisons peuvent expliquer la situation actuelle, comme la jeunesse du domaine, mais aussi et surtout sa diversité et son étendue. Un des défis majeurs de la discipline est sûrement de réussir à s'accorder et à préciser un corpus de connaissances caractéristiques des systèmes multi-agents. En effet, que cela soit au niveau des modèles cognitifs d'agents, des plateformes de développement, ou encore des concepts fondamentaux du domaine, il est difficile d'avoir un consensus qui aboutisse sur des définitions formelles. Nous nous intéresserons dans ce document aux systèmes

multi-agents physiquement distribués, constitués d'agents à gros grain.

Notre principal objectif est de remédier à la diversité des propositions en terme de modèles cognitifs d'agent et de modèles organisationnels. Pour cela, nous proposons d'aborder ce problème de l'interopérabilité selon deux axes : l'identification et la définition d'une infrastructure minimale et générique facilitant le développement de différents modèles d'agents d'une part, et la proposition d'un modèle de spécification de protocoles d'interaction s'appuyant sur le principe de la coordination par standardisation[86] d'autre part. Ces deux axes tendent à uniformiser les infrastructures des systèmes multi-agents au niveau des fonctionnalités et des garanties que les concepteurs peuvent en attendre.

Notre travail s'articule autour de quatre points, qui ont été successivement étudiés :

- un modèle d'agent minimal générique (section 2.3),
- une réification des protocoles d'interactions entre agents (section 3.1.2),
- une réification de la notion d'organisation au sein des systèmes multi-agents (section 3.2),
- une proposition de méthodologie d'analyse et de conception de systèmes multi--agents ouverts (section 3.3).

Ces différents aspects sont abordés à la fois dans le but d'identifier les notions fondamentales qui constituent les systèmes multi-agents, mais aussi dans un souci d'ingénierie de ces systèmes. Ainsi, en plus de l'interopérabilité qui constitue le pivot de nos travaux, l'ingénierie des systèmes produits a été une préoccupation continue lors de nos phases de réflexion et d'implémentation. Avant de détailler chacun de ces aspects, dans le chapitre 2 pour le modèle d'agent, et dans le chapitre 3 pour le modèle d'interaction et la méthodologie, nous allons présenter succinctement chacun de ces axes de travail.

Mes travaux ont débuté lors du stage de DEA sur le sujet suivant : "*La notion de service dans les systèmes multi-agents*". Après une étude bibliographique et technologique, nous avons défini la notion de service et l'avons intégrée au sein de MAGIQUE, la plateforme multi-agents développée au sein de l'équipe[9]. Cette plateforme initialement développée en PROLOG avait été réécrite par Jean-Christophe Routier avec le langage JAVA. Cette deuxième version reposait sur l'extension de classes existantes pour spécialiser des agents, dont le comportement était implémenté sous forme de méthodes JAVA (se reporter à la section 4.1 pour une description plus approfondie de cette version de MAGIQUE).

L'intégration de la notion de service au sein de MAGIQUE a entraîné une réécriture des classes fondamentales, particulièrement celles implémentant le modèle d'agent, et a impliqué un changement de paradigme lors de la conception de systèmes multi-agents. Le premier point a débouché sur notre modèle d'agent minimal générique présenté dans le chapitre 2, tandis que le second point nous a amené à nous interroger sur une conception orientée services des systèmes distribués. En effet, au lieu de créer un ensemble d'agents représentés chacun par une classe, nous proposons maintenant de définir un ensemble de compétences (ou services), qu'il faut ensuite attribuer aux agents. Cette approche componentielle facilite une conception incrémentale du système et favorise une meilleure réutilisation.

Cependant, cette approche pose de nouvelles problématiques, liées à la description des services dans le système d'une part, et à la gestion des dépendances entre les services du système d'autre part. J'ai donc étudié les travaux existants et j'en ai conclu que les

recherches effectuées autour du Web Sémantique pourraient nous fournir un cadre et des outils de description des services. Suite au développement d'applications avec MAGIQUE (présentées dans le chapitre 5), je me suis intéressé aux travaux initiés par Munindar Singh[76] sur la notion de programmation orientée interactions, pour faciliter la gestion des dépendances entre services. C'est dans ce contexte que nous avons développé notre modèle de spécifications exécutables de protocole d'interaction. Finalement, les expériences en terme d'implémentation d'applications et d'infrastructures pour les systèmes multi-agents, nous ont permis d'initier une démarche méthodologique pour la conception de systèmes ouverts.

Un modèle d'agent minimal générique

Notre modèle d'agent repose sur la notion de compétence. Cette notion de compétence est assimilable à celle de composant logiciel : c'est une unité de code pouvant implémenter des comportements, et qui dans certaines limites peut être réutilisée dans différents contextes. La notion de service correspond à l'utilisation des compétences par l'agent les possédant. Ainsi, un agent qui possède une compétence gérant l'accès à une base de données, pourra proposer ce service aux autres agents du système.

Un agent étant constitué d'un ensemble de compétences, nous nous sommes interrogés sur le nombre minimal de compétences nécessaires à un agent, ainsi qu'à la nature de ces compétences. Cette recherche poursuit deux buts : proposer une définition de "*ce qu'est un agent*" dans le cadre des systèmes multi-agents physiquement distribués constitués d'agents à gros grain, et surtout proposer une infrastructure facilitant la réalisation des différents modèles d'agents existants. Ce deuxième point permet de disposer d'un "*test-bed*" facilitant l'évaluation et la comparaison de modèles existants, ce qui clarifierait la situation actuelle, où bien trop souvent le modèle d'agent se trouve *intrinsèquement* lié à la plateforme l'implémentant.

Nous avons conclu qu'un agent n'était qu'une coquille vide, possédant au moins deux compétences : une compétence de communication et une compétence gérant l'évolution de l'agent. En effet, sans la possibilité de communiquer, un agent n'existe pas, et sans la possibilité d'évoluer, l'agent ne peut implémenter de comportements complexes. Les deux fonctionnalités qui pour nous définissent ce qu'est un agent sont donc la capacité de communication, et la possibilité d'évolution (ce qui dans notre approche correspond à la gestion des compétences d'un agent).

Une conception centrée sur les interactions et leur exécution

La conception d'applications avec MAGIQUE (se reporter aux illustrations du chapitre 5) nous a permis d'identifier deux aspects implicites : les relations intervenant entre les compétences des agents et le rôle de l'organisation au sein d'un système multi-agents. Le premier point, qui correspond aux dépendances fonctionnelles existantes entre les compétences, nous a conduits à réifier la notion de protocole d'interaction, à la fois pour pouvoir l'expliciter et le manipuler, mais aussi pour fournir des garanties sur le déroulement des conversations entre agents.

Ainsi, le cœur de notre proposition est un modèle formel de description de protocoles d'interaction, et un mécanisme de transformation générant le code nécessaire à la gestion de ces protocoles. Pour que les agents d'un système en cours d'exécution puissent exploiter ces nouvelles interactions, nous nous reposons sur notre modèle d'agent générique minimal, qui autorise la construction incrémentale des agents par ajout de compétences. Notre approche se situe donc dans la continuité des travaux initiés par Singh sur la programmation orientée interaction.

Notre modélisation des interactions s'appuie sur une représentation graphique de schéma de conversations entre différents rôles. Ces schémas centralisent l'ensemble des informations qui caractérisent une conversation : les différents rôles participant, les compétences métiers nécessaires au déroulement de la conversation, la nature des différents messages échangés, et enfin le déroulement temporel de l'interaction. Dans cette approche, une conversation entre agents est considérée comme une loi sociale (au sens des travaux menés par Shoham[74]), et implique une certaine perte d'autonomie des agents. Cette perte est compensée par les garanties qu'un agent est en droit d'attendre de l'exécution d'un protocole d'interaction, ainsi que par un allégement significatif de la charge du concepteur de systèmes multi-agents. D'autre part, la nature de cette spécification offre une ouverture favorisant l'interopérabilité de systèmes hétérogènes (c'est-à-dire des plateformes de nature différentes).

La notion d'organisation : une réification des liaisons entre entités

Dans les sociétés humaines, l'organisation repose bien souvent sur des liens hiérarchiques qui indiquent le pouvoir de contrôle ou de décision de chacun des agents au sein du système. Cette notion d'organisation est généralement liée à la notion de rôle, que l'organisation peut affecter à ses agents pour garantir son bon fonctionnement. Dans une société logicielle, la notion d'organisation assouplit les relations liant les agents, et définit un cadre au sein duquel les interactions entre agents peuvent se produire.

Cependant, la situation sur les modèles d'organisation (hiérarchique, holonique, orienté groupes) ou même la définition de "*ce qu'est une organisation*" est similaire à la situation précédemment décrite à propos des modèles d'agents. Un consensus existe autour de certaines fonctionnalités des organisations, mais de nombreux modèles sont proposés et il est difficile de les évaluer conjointement. Notre objectif est donc de préciser la notion d'organisation pour les systèmes multi-agents, et de la réifier au sein de notre modèle de spécification de protocoles d'interaction.

Nous montrons dans le chapitre 3, l'importance de la réification de la notion d'organisation (ce qui a été souligné dans les travaux de Jacques Ferber[31] avec le modèle organisationnel Aalaadin[31], ou encore avec la notion d'institution introduite par John-Jules Meyer[27]), et nous détaillons l'utilisation que nous en faisons, particulièrement lors de l'exécution des protocoles d'interaction.

Une méthodologie pragmatique pour les systèmes multi-agents ouverts

L'objectif final de nos travaux est de regrouper l'ensemble des thèses précédentes pour proposer une méthodologie pragmatique de conception de systèmes multi-agents. Plus précisément, nous proposons à la fois un cadre conceptuel reposant sur notre modèle d'agent générique minimal, la réification de la notion d'organisation, ainsi que la réification des interactions, mais aussi un cadre pragmatique, en produisant une implémentation de référence d'un environnement de développement et de déploiement de systèmes multi--agents.

Il est important de noter que ce travail s'inscrit dans une logique d'ingénierie et n'impose pas de modèle d'agent, ni de modèle d'organisation. En effet, notre proposition doit être considérée comme une infrastructure n'imposant pas de modèle d'agent de haut niveau (tels que le modèle Bdi), de même la réification de la notion d'organisation signifie bien que nous fournissons les outils permettant de créer différents modèles organisationnels. La seule caractéristique imposée est la gestion des interactions, qui peut cependant être considérée comme une facilité de gestion et de suivi des conversations.

De même, du point de vue de l'implémentation, le but est de fournir une architecture modulaire reposant sur les standards actuels (pour les aspects agents avec la Fipa ou pour les positionnements technologiques tels que Owl et les modèles de composants) et dont les composants soient suffisamment découplés pour faciliter leur réutilisation (pour une intégration au sein d'autres plateformes, ou pour la création simplifiée d'agents *façade* pour des *legacy systems*).

La méthodologie Rio, pour *Rôles Interactions et Organisations*, combinée à la plateforme G que nous développons, définit une nouvelle approche de la conception de systèmes distribués. Le cycle habituel de création d'un logiciel, qui consiste à commencer par une phase d'analyse, puis de conception, enfin d'implémentation puis de maintenance, est une approche dont le but est de fournir un logiciel monolithique dont l'évolutivité est assurée par la production de versions successives (parfois non totalement compatibles). A cette approche *itérative*, nous substituons une approche *incrémentale* ou *continue*, consistant non plus à considérer une création de logiciel, mais à fournir un système flexible pouvant être adapté et enrichi au fur et à mesure de l'évolution des besoins. Ceci est rendu possible grâce à la dynamicité de notre modèle d'agent, et à une gestion fine des dépendances fonctionnelles (via la réification des interactions) et structurelles (via le modèle de composant choisi).

Organisation de ce document

Ce document est structuré en quatre chapitres. Le premier chapitre présente un état du domaine des systèmes multi-agents. Le deuxième chapitre présente le modèle d'agent minimal générique que nous proposons, tandis que le troisième chapitre introduit la notion de spécification exécutable de protocole d'interaction et précise le rôle de la notion d'organisation dans notre approche. Ce chapitre se termine sur la démarche méthodologique, Rio, que nous avons élaboré au long de nos expérimentations. Le quatrième chapitre

décrit les plateformes que nous avons développés : MAGIQUE dans un premier temps, la plateforme initiale de développement, et dans un second temps G, qui est notre nouvelle plateforme pour le développement de systèmes multi-agents. Le cinquième et dernier chapitre illustre notre approche en détaillant deux applications réalisées dans le cadre de cette thèse : une application de travail collaboratif et une infrastructure pour le calcul distribué. Nous terminons par une conclusion générale sur l'état actuel de nos travaux et sur les perspectives envisagées.

Chapitre 1

Historique et état de l'art

> "*No one wants to learn from mistakes, but we cannot learn enough from successes to go beyond the state of the art.*"
>
> Henry Petroski.

Ce chapitre introduit le contexte dans lequel notre travail s'est effectué. La présentation de ce contexte s'articule autour de trois axes : quelques repères historiques sur l'intelligence artificielle distribuée et l'émergence des systèmes multi-agents, un aperçu des théories et technologies utilisées dans le domaine, et finalement une présentation des paradigmes fondamentaux des systèmes multi-agents.

1.1 L'émergence du domaine

Nous allons présenter dans cette section un historique succinct de l'émergence du domaine récent des systèmes multi-agents. Cette discipline constitue l'une des composantes de l'intelligence artificielle distribuée (*Distributed Artificial Intelligence*, DAI), avec la résolution de problèmes distribués (*Distributed Problem Solving*, DPS), et l'intelligence artificielle parallèle (*Parallel Artificial Intelligence*, PAI). Nous étudierons dans un premier temps le passage de l'Intelligence Artificielle aux systèmes multi-agents, nous nous intéresserons dans un second temps au Génie Logiciel des systèmes distribués, avant de donner quelques repères académiques sur l'émergence du domaine.

1.1.1 De l'intelligence artificielle aux systèmes multi-agents

Le projet initial de création d'une intelligence artificielle remonte presque à la nuit des temps. La littérature illustre ce fantasme depuis les mythes grecs d'Hephaestus et du Pygmalion[1] en passant par le golem de Rabbi Loew[2] jusqu'au Frankeinstein de Mary

1. Plus d'informations disponibles sur :`http://www.pantheon.org/articles/h/hephaestus.html` et `http://www.pantheon.org/articles/p/pygmalion.html`
2. Un court résumé de la création du golem : `http://pnews.org/bio/2golem.shtml`

Shelley. Cependant, ce projet n'a acquis sa crédibilité scientifique que très récemment, principalement avec l'avènement de l'informatique et des modèles permettant de manipuler ces nouvelles *machines à tout faire*. Il est d'ailleurs très intéressant de constater que les pionniers de l'informatique qu'ont été Alan Turing et John Von Neumann, dont les modèles de calcul et d'architecture de l'ordinateur sont toujours les fondements des machines actuelles, avaient dès le début de leurs travaux cet objectif en tête.

Cet intérêt provient probablement d'un contexte dans lequel l'idée d'intelligence était pensée en terme de calcul. En effet, la machine de Pascal, ou encore celle de Babbage, étaient des calculateurs. La notion de calculabilité fut d'ailleurs formellement exprimée par Gödel dans sa thèse[42], et elle constitue le fondement logique pour l'informatique théorique. Ce concept de calcul, et plus précisément de la description ou de la spécification de calcul, amena le développement des premiers langages informatiques et avec eux les débuts de l'algorithmique. L'ensemble de ces idées (ainsi que la seconde guerre mondiale...) ont fourni un cadre théorique et pragmatique permettant la conception, la création et le développement des premiers ordinateurs.

Avec les premiers ordinateurs, le projet de création d'une intelligence artificielle a trouvé son meilleur vecteur. John Von Neumann s'intéressa aux jeux et créa le domaine de la théorie des jeux[62] en lui apportant notamment le théorème du *min-max*[62] pour les jeux à deux joueurs à information complète. Tandis, qu'Alan Turing, après avoir initié les premiers travaux en cryptologie pour casser le code ENIGMA, proposa son fameux test d'intelligence, reposant sur l'impossibilité de différenciation de la machine et de l'homme lors de dialogues[84] . Ensuite, les langages tels que LISP [3] ont donné de nouveaux outils pour modéliser le monde et ses mécanismes en terme de symboles, tout comme Prolog[19] en terme de formules logiques et d'inférences. Du projet initial d'intelligence artificielle est née une multitude de domaines de recherche, soit en terme d'objectifs :la reconnaissance vocale, la reconnaissance de formes, la compréhension des langages naturels... soit en terme d'outils : les réseaux neuronaux, la programmation génétique, la programmation logique...

Rapidement, la complexité algorithmique et conceptuelle des problèmes en intelligence artificielle a amené le développement de nouveaux travaux, tentant de maîtriser ou de répartir la puissance de calcul ou la mémoire nécessaire, mais aussi de penser la résolution de ces problèmes non plus de manière monolithique et séquentielle, mais de manière distribuée et concurrente.

De l'intelligence artificielle à l'intelligence artificielle distribuée

Au début des années 1960 des chercheurs américains eurent l'idée de faire fonctionner des machines en réseau, mais à grande échelle (et non pas au niveau d'un bâtiment, comme les recherches menées au PARC sur le protocole Ethernet [4]). Pour cela, ils ont défini

3. Issu d'une Université d'Eté à Dartmouth qui s'est tenue en 1956 et qui est restée dans les annales : `http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html`

4. "A la fin des années 1972, Meltcafe et ses collègues du PARC XEROX ont développé le premier système Ethernet expérimental pour interconnecter le XEROX ALTO, une station de travail disposant d'une interface graphique. Le système expérimental fut utilisé pour relier des ALTOs les uns aux autres, ainsi qu'à des serveurs et des imprimantes laser. La vitesse d'horloge de l'interface Ethernet expérimentale

l'infrastructure et notamment l'algorithme de routage par paquets, qui constitue un des éléments fondateurs d'Internet. C'est en 1972 qu'eut lieu la première démonstration publique d'ARPANET, et quelques mois après apparaissaient les premiers systèmes d'échange de courriers électroniques, qui furent pendant des années la principale application de ce nouveau média de communication.

Parallèlement à ces évolutions technologiques, le marché de l'informatique a fortement évolué. Après les gros systèmes (un serveur fournissant la puissance de calcul à un ensemble de terminaux[5]), l'informatique s'est démocratisée avec l'apparition des premiers PC (*Personal Computer*) d'IBM. Ce sont d'abord les entreprises qui en ont profité, avant que le marché de l'informatique personnelle ne se développe avec les PC mais aussi d'autres machines telles que l'*Apple II* , ou le *Personal Electronic Transactor* de COMMODORE ...

Ainsi, la conjonction de ces deux facteurs, la baisse du coût des machines et la généralisation des réseaux locaux et d'Internet, a entraîné les débuts de la conception de systèmes distribués, qui s'est traduit dans le domaine de l'Intelligence Artificielle par la création d'une branche dédiée : l'*Intelligence Artificielle Distribuée* (IAD).

Les problèmes de prédilection de l'IAD sont les disciplines suivantes : le *Distributed Problem Solving* (DPS), les *Multi-Agent Systems* (MAS), et la *Parallel Artificial Intelligence* (PAI). Nous ne décrirons pas ici plus en détail le domaine de la résolution de problèmes distribués, ni celui de l'intelligence artificielle parallèle, mais nous nous concentrerons sur les systèmes multi-agents (SMA) qui constituent le cœur de notre travail.

Les premiers systèmes multi-agents

Nous allons dans cette section brièvement présenter les grandes étapes ayant mené à la constitution du domaine d'étude des systèmes multi-agents[6]. Dans un premier temps, nous présenterons les travaux des *pionniers*, qui ont posé les problématiques et qui ont ouvert la voie aux travaux maintenant reconnus comme *classiques*. Dans un second temps, nous verrons l'influence de la vie artificielle sur le domaine des systèmes multi-agents, et finalement ce que Jacques Ferber appelle l'âge *moderne*.

Les premiers travaux ont eu lieu aux Etats-Unis, avec des systèmes étudiant la relation entre architecture et mode de raisonnement. Particulièrement, le système de reconnaissance vocale HERSAY-II[43] a donné lieu à la mise en place d'une architecture de partage d'information : les tableaux noirs. Cette architecture permet de résoudre un problème sans structure de contrôle *a priori*, mais grâce à l'interaction de spécialistes via une zone d'information partagée. Ces travaux ont permis l'identification d'importantes problématiques des systèmes multi-agents telles que la coopération, la coordination d'action et la négociation.

L'âge *classique* du domaine caractérise le développement de deux systèmes conséquents répondant partiellement ou précisant les thématiques précédentes, et d'un modèle de distribution de tâches qui reste l'un des plus utilisé. Le premier système, *Distributed Vehicule Monitoring Test* développé par l'équipe de Victor Lesser[15], avait pour but

était dépendante de l'horloge système de l'ALTO, ce qui entraînait un taux de transmission d'environ 2.94 Mbps.", Ethernet : The Definitive Guide, Charles E. Spurgeon, O'Reilly and Associates

5. Ce principe est similaire au fonctionnement du Minitel.

6. Cette section repose sur le chapitre 1.3 du livre "Les systèmes multi-agents", vers une intelligence collective, écrit par Jacques Ferber.

de suivre le déplacement d'une voiture dans une ville à l'aide d'un réseau de capteurs. L'intérêt de ce projet est la mise en avant des problèmes de planification et de coordination dans un environnement produisant des informations bruitées, ou même contradictoires. DVMT eu un impact important aux Etats-Unis, tandis qu'en Europe c'est le système MACE développé par Les Gasser[39] qui eu le plus d'influence. Ce projet fit le lien entre les travaux de Carl Hewitt (que nous évoquerons dans la section suivante) sur les langages d'acteurs, et l'importance de la représentation que doivent avoir les agents d'eux mêmes et des autres. En effet, l'échange de messages n'est pas suffisant, et l'organisation d'un ensemble d'entités nécessite des notions telles la représentation des compétences d'un agent pour qu'il puisse raisonner sur ses capacités propres, mais aussi sur celles de ses partenaires. Le dernier projet ayant eu un impact toujours sensible de nos jours est la notion de réseau contractuel, ou *Contract Net*, introduit par Reid Smith[79] au début des années 80. Ce système répond au problème de l'allocation de tâches par la mise en place d'un protocole d'interactions reposant sur le fonctionnement des appels d'offre dans les marchés publics. Ainsi, la nécessité d'une zone partagée (i.e. un tableau noir, ou *blackboard*), dans laquelle les agents échangent l'information, n'est plus nécessaire à la collaboration et la coordination d'un ensemble d'agents.

Une autre influence importante a été apportée par les travaux menés dans le domaine de la vie artificielle, qui se destine comme le définit Langton à "*l'étude de la vie telle qu'elle pourrait être, et non de la vie telle qu'elle est*". Ce projet ambitieux remonte aux débuts de l'informatique, avec notamment les travaux sur les automates cellulaires[63] ou les réseaux de neurones[23]. Actuellement, ce domaine regroupe plusieurs thématiques telles que la dynamique des phénomènes complexes, l'étude de populations à l'aide d'algorithmes génétique, la robotique avec la réalisation d'entités autonomes évoluant en environnement complexe et surtout l'étude de l'émergence de phénomènes collectifs et de leur liaison avec le comportement individuel. C'est avec ce dernier thème que les interactions sont les plus fortes et les échanges les plus fructueux pour les systèmes multi-agents. Comme nous le verrons dans la section 1.3 présentant différents modèles d'agent, les modèles d'agents en vie artificielle sont généralement basés sur des comportements individuels simples (type stimulus/réaction). Néanmoins, l'interaction de ces comportements produit un comportement global complexe, exhibant des propriétés ou des résultats inaccessibles à un agent seul.

Enfin, la situation actuelle est un patchwork issu de l'ensemble de ces influences et d'autres développements technologiques plus récents. Les différentes approches peuvent être caractérisées par les tendances suivantes : les recherches dans la continuité de l'IAD, la formalisation logique, les travaux sur les actes de langages, l'IAD et la théorie des jeux, l'utilisation des Réseaux de Pétri dans les systèmes multi-agents, la poursuite des travaux sur les langages d'acteurs et les modèles de systèmes réactifs.

1.1.2 Le génie logiciel des systèmes distribués

Après avoir présenté succinctement l'historique du domaine des systèmes multi-agents, nous allons dans cette section aborder brièvement et de manière non exhaustive quelques travaux qui ont une forte influence sur l'avènement des systèmes multi-agents. Dans un premier temps, nous nous intéresserons aux technologies et langages qui ont été proposés

pour maîtriser l'aspect distribué des applications, avant de nous concentrer dans un second temps sur l'apport de Carl Hewitt avec les langages d'acteurs.

Invocation de procédures distantes et mobilité du code

La multiplication des postes de travail et le développement des réseaux a entraîné le développement de nouveaux types d'application ne s'exécutant plus sur une seule machine, mais sur un ensemble plus ou moins grand de machines. Ces applications étaient initialement développées à l'aide de langages standards, auxquels étaient adjointes des librairies prenant en charge la communication entre machines distantes.

Progressivement de nouveaux outils et langages ont été créés spécifiquement pour faciliter le développement de systèmes distribués. L'un des premiers outils fut probablement le principe de l'appel de procédure distante, ou *Remote Call Procedure*, qui permet d'invoquer une procédure s'exécutant sur une machine distante. Ce principe s'est ensuite étendu aux langages orientés objets, avec des langages comme ARGUS[50] ou DISTRIBUTED SMALLTALK[8], avant d'être normalisé par l'*Object Management Group*, en 1991 pour la première version CORBA, puis standardisé en 1996 avec la version 2.0 de CORBA.

Parallèlement à ces travaux, la communauté étudiant le domaine des calculs à hautes performances s'est intéressée à des réseaux particuliers de machines, appelés *clusters*. Construire un *cluster* consiste à assembler un ensemble important de machines sur un réseau local à haut débit. On peut considérer cela comme une version économique des anciennes machines massivement multi-processeurs, qui est rendu possible grâce à l'accroissement des débits des réseaux. Sur ce genre d'environnement matériel, des langages se focalisant sur l'échange de messages sont apparus comme PVM[81], pour *Parallel Virtual Machine*, ou MPI[7], pour *Message Passing Interface*. Ces langages sont un premier pas important concernant l'indépendance du langage de programmation vis-à-vis de l'environnement d'exécution. Cependant, cette indépendance reste relative et nécessite la compilation du programme sur chacune des architectures hôtes.

Ce n'est que très récemment que des langages ont été développés pour favoriser les programmes *mobiles*. La problématique de la mobilité du code repose sur deux aspects principaux : la gestion de l'exécution et la gestion de la sécurité. Les premières expérimentations se sont principalement concentrées sur le premier aspect. Ainsi, AGENTTCL[40]et le framework VOYAGER[8] illustrent cette approche en fournissant des facilités de migration du code reposant sur des langages interprétés (TCL pour le premier et JAVA pour le second). Plus récemment, les AGLETS développés par IBM ont répondu (partiellement) au second aspect et constituent l'une des plateformes actuelles d'expérimentation sur le code mobile[9]. Il est intéressant de remarquer que la notion d'agent est naturellement utilisée dans ces travaux sur le code mobile, car elle illustre bien l'idée d'une entité autonome, libre de ses déplacements et cherchant à atteindre ses propres buts.

7. Plus d'informations sur MPI : `http://www-unix.mcs.anl.gov/mpi/`

8. Initialement développée par la société *ObjectSpace*, le lecteur intéressé pourra se reporter à l'adresse suivante pour plus de renseignements : `http://www.recursionsw.com/products/voyager/`

9. D'autant plus depuis qu'IBM a abandonné son développement et a donné le code source du projet sous une licence Open Source : `http://aglets.sourceforge.net/`

Les objets actifs et les acteurs

La gestion de l'exécution *simultanée* de programmes s'est particulièrement développée au début des années 1970 avec l'apparition des premiers systèmes d'exploitation UNIX[10]. Le principe qui consiste à allouer un certain quantum de temps à un programme avant de passer à un autre, permet de simuler l'exécution *concurrente* de programmes sur les machines monoprocesseur. Les objets actifs sont un prolongement de ce principe, et consistent à associer un processus à un objet. Un des modèles les plus représentatifs de cette approche est celui d'*acteur*[11].

Le modèle *acteur* proposé par Carl E. Hewitt[44] illustre cette approche et constitue l'un des premiers langages orienté agents. Issus des travaux en intelligence artificielle destinés à pallier les limitations des approches de type *blackboard* où une mémoire globale partagée est nécessaire, le modèle *acteur* est naturellement décentralisé et distribué. Chaque acteur ne détient que des informations locales et son comportement est indépendant des autres acteurs. D'autre part, les communications qui s'effectuent uniquement par envoi de message sont asynchrones. Voici la définition d'un *acteur*, telle que l'a proposé Hewitt : *Intuitively, an* ACTOR *is an active agent which plays a role on cue according to a script. (...) Our formalism shows how all of the modes of behavior can be defined in terms of* one *kind of behavior :* sending messages to actor.[45]

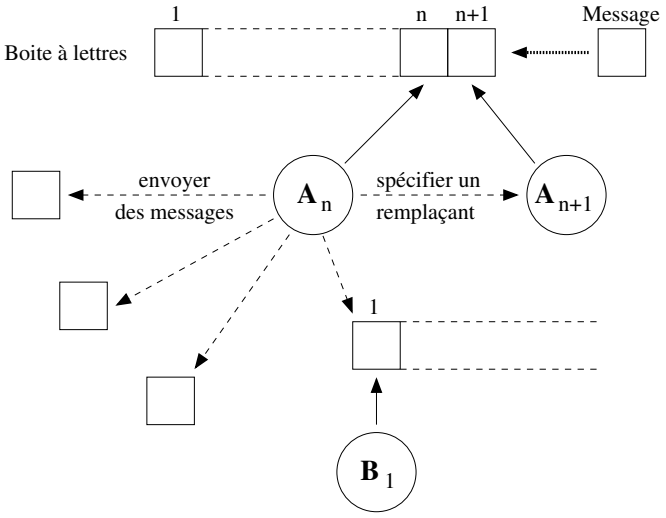

FIG. 1.1 – *Comportement d'un acteur lorsqu'il traite le nième message de sa boîte aux lettres (extrait de [2])*

La figure 1.1 illustre le fonctionnement d'un acteur. Chaque acteur dispose d'une boîte aux lettres à partir de laquelle il traite successivement les messages qu'il a reçus. Chaque message est associé à un script qui peut envoyer des messages à d'autres acteurs (ses accointances) ou créer de nouveaux acteurs. Une fois le script terminé, il spécifie

10. Source : `http://www.faqs.org/faqs/os-research/part1/section-10.html`

11. Le lecteur intéressé par l'origine des systèmes à objets concurrent pourra se reporter à la discussion relatée dans [3]

un comportement de remplacement qui devra traiter les messages suivants. Un acteur est ainsi caractérisé par un enchaînement de comportements totalement ordonnés. Nous reviendrons sur ce modèle dans le chapitre 2 lorsque nous présenterons MAGIQUE.

1.1.3 Quelques repères académiques

Le domaine des systèmes multi-agents est maintenant un domaine établi avec ses spécialistes et ses conférences nationales et internationales. Les premières rencontres académiques (i.e. *workshops* ou conférences) ont eu lieu à la fin des années 80. Le DAI *workshop* aux Etats-unis, ou encore *Modeling Autonomous Agents on Multi-Agent Worlds* en Europe et *Multi-Agent and Concurrent Programming* au Japon, se sont regroupés pour créer la première conférence internationale du domaine, *International Conference on Multi-Agent Systems*, en 1995.

Ensuite, d'autres conférences ont émergé avant de se reconcentrer en une unique conférence couvrant l'ensemble des thématiques de recherche sur les systèmes multi-agents des aspects théoriques au point de vue plus pragmatiques. Ainsi, la conférence AAMAS, *Autonomous Agents and Multi-Agent Systems* s'est tenue à Bologne en 2002 et représente la conférence internationale la plus importante du domaine. Cette concentration au niveau scientifique indique la maturité qu'a atteint le domaine.

Cette maturité est d'autant plus renforcée que des transferts technologiques vers l'industrie sont plus fréquents grâce notamment au soutien de la communauté européenne avec la création et le financement d'un réseau d'excellence européen : AGENTLINK[12]. Parmi les actions soutenues par AGENTLINK, l'organisation régulière d'Universités d'Eté, qui attirent une centaine de participants, représente une étape supplémentaire dans la structuration du domaine. Plusieurs livres d'introduction et de présentation du domaine sont aussi maintenant disponibles : "*Les systèmes multi-agents, Vers une intelligence collective*" de Jacques Ferber, "*Multi-Agent Systems*" édité par Gerhard Weiss ou encore "*An Introduction to Multi-Agent Systems*" de Mike Wooldridge.

Enfin, un dernier point mérite d'être souligné : la standardisation même partielle et incomplète que propose la FIPA[13] est une autre forme de structuration qui si elle aboutit pourrait fournir un cadre équivalent à ce que représente CORBA pour les technologies orientées objets. Cette standardisation représente un signe important à destination des entreprises, même si pour l'instant les tests d'interoperabilité ne concernent que les couches basses de communication.

12. Le site d'AGENTLINK : `http://www.agentlink.org/`
13. Le site de la FIPA, *Foundation for Intelligent Physical Agents* : `http://www.fipa.org`

1.2 Les théories et technologies pour les systèmes multi-agents

Nous présentons dans cette section les théories et les technologies qui constituent les outils de base dans le domaine des systèmes multi-agents. Cette présentation s'effectue selon deux axes distincts identifiant les apports de travaux provenant d'une part des recherches sur l'intelligence artificielle distribuée, et d'autre part des travaux issus du génie logiciel. La première partie présentera succinctement la théorie des actes de langages, qui est le fondement des langages de communication entre agents, et les théories de représentations des connaissances. La seconde partie retracera les évolutions des langages de programmation et de leurs paradigmes, puis introduira les méthodologies orientées agents que nous avons étudiées, avant de nous concentrer sur un aspect de ces méthodologies : la notion d'interaction.

1.2.1 L'héritage de l'intelligence artificielle distribuée

Dans cette première partie, nous allons présenter les principaux concepts provenant de la linguistique, de l'intelligence artificielle pour les systèmes multi-agents. Nous aborderons d'abord la théorie des actes de langages, en présentant une courte histoire de sa genèse. Ensuite, nous étudierons les travaux effectués en représentation des connaissances, qui recouvrent à la fois les problèmes de représentation, mais aussi et surtout les problèmes de manipulation ou d'interrogation de tels systèmes. Enfin, nous terminerons sur les langages de communication entre agents, qui sont d'une certaine manière un aboutissement des deux courants précédents.

La théorie des actes de langages

Cette section repose sur un article de Barry Smith[77] [14] et présente une histoire succincte de la théorie des actes de langages en présentant les principaux précurseurs et fondateurs. Cette théorie est maintenant utilisée comme un des fondements du domaine des systèmes multi-agents, et constitue l'essence des travaux sur les langages de communication entre agents (*Agent Communication Languages*, ou ACL).

Le point de départ de ces travaux est la relation subtile liant le langage et les actes qu'ils peuvent induire, et plus particulièrement l'importance du langage comme moyen d'influencer les actes d'autrui.

Bien que depuis l'Antiquité, Aristote ait déjà travaillé sur le langage, son étude se restreignait aux énoncés s'interprétant comme étant vrai ou faux (les syllogismes en sont un exemple). Cette approche très logicienne, repoussant toute interprétation sociale du langage a perduré jusqu'à la fin du XIX ème siècle. Ce n'est qu'avec les travaux du philosophe Thomas Reid (1710-1796) sur les concepts de promesse, d'avertissement ou encore de pardon, que les énoncés autres que les propositions ou les affirmations commencèrent à être étudiés. Ces concepts furent identifiés par Reid comme des *opérations sociales*, ou encore *actes sociaux*, en opposition aux *actes solitaires* tels que les jugements, les intentions,

14. L'ensemble des citations, ainsi que la trame de cette partie, provient de l'article de Barry Smith.

les délibérations ou les désirs, qui ne nécessitent pas d'interlocuteurs, ni même d'être exprimés. Ainsi, les *actes sociaux* sont intrinsèquement dirigés vers quelqu'un. Cependant, bien que Reid ait sur ce point touché un des points les plus importants des théories modernes, il n'arriva pas à exprimer la relation entre un énoncé et l'intention (ou acte de volonté) sous-jacente.

Ce n'est qu'avec les travaux du phénoménologue Adolph Reinach (1883-1917) qu'une première théorie systématique des phénomènes tels que la promesse, le questionnement, la requête, le commandement, l'accusation fut produite. Reinach a ainsi produit une taxonomie des différent actes illocutoires (*speech actions*), de leurs variations, ainsi que de leur liaison avec les obligations légales ou éthiques. Ces travaux se sont aussi étendus à la notion de délégation, c'est-à-dire lorsqu'une action de promesse, de commandement ou d'invite, est réalisée par une personne *au nom* d'une autre. Pour en arriver à ces propositions, Reinach a pu bénéficier des travaux de philosophes ou linguistes tels que Bolzano (1781-1848), Brentano (1838-1917), Frege (1848-1925) ou Husserl (1859-1938) qui ont distingué les propositions, (ou jugements), des concepts (ou idées) et des présentations. Nous allons donc présenter les idées de Brentano et Husserl qui ont contribué aux travaux de Reinach.

Leur principal apport fut la différenciation entre jugements et concepts, non seulement du point de vue de la logique, mais aussi psychologique. Ce n'est qu'ainsi qu'ont pu être identifiés le contenu (la sémantique) et la force (la pragmatique), qui ont ensuite pu devenir un sujet d'étude propre. Husserl proposa un cadre logique inspiré des travaux de Brentano, reposant sur une nouvelle distinction entre le contenu immanent (*immanent content*) d'un acte et son contenu idéal (*ideal contant*, ou *content-species*). Les deux types de contenus sont en relation de la même manière qu'un triangle dessiné dans le sable est lié à l'idée de triangle idéale et abstraite du géomètre[15]. L'apport principal de cette distinction, est qu'il était maintenant possible de concevoir des contenus propositionnels comme des ensembles de parties interchangeables, pouvant être combinées ensemble dans différents contexte.

Brentano défendait la thèse que chaque acte mental est soit une présentation d'un objet, soit est fondé sur une telle présentation. Mais surtout, pour Brentano ces énoncés sont des actions incitant l'interlocuteur à réagir :

> "*Speaking is often brought into opposition with acting. But, speaking is itself an acting. An activity, by means of which one wants to call forth certain psychic phenomena. In the request and in the command the will to do something. Questionning or addressing belong here also: the one wants to determine the will to communicate something, the other to draw the attention to something that is to be heard. (Interest) in the cry a feeling, whether of pain, whether of joy, whether of amazement. In the statement one wants to call forth a judgement, etc.*"

15. Le lecteur familier avec la conception orientée objets pourra rapprocher cette idée de la relation existante entre les classes et les instances.

La théorie des actes objectifiés de Husserl Husserl exploita cette idée d'intention et avança que chaque acte mental est soit un acte d'objectification ou repose sur un tel acte. Un acte d'objectification est un acte qui est donné comme attaché à, ou dirigé vers un objet. La théorie du langage de Husserl repose sur cette théorie des actes objectifiés. Le langage n'a de sens que dans la mesure où il existe des actes dont le sens est accordé à des expressions spécifiques dans des expériences intentionnelles spécifiques. Plus précisément, toutes les expressions sont associées soit à des *actes nominaux*, qui sont dirigés vers les objets au sens strict, soit à des *actes de jugements*, qui se rapportent à une situation (*state of affair*). Ainsi, pour Husserl toute utilisation du langage s'apparente à une utilisation référentielle (*all uses of language is approximate to referantial uses*).

Le second aspect de la thèse de Husserl est en rapport avec l'utilisation du langage pour poser des questions, produire des ordres, exprimer des avertissements ou des requêtes. Prenons l'exemple d'une question : "Est-ce que John est assis ?". Cette phrase peut se comprendre comme une formulation abrégée à propos d'un acte sous-jacent de questionnement, une formulation qui pourrait être explicitée comme "Je me demande si John est assis", ou "Ma question est de savoir si John est assis". Appliquée aux phrases exprimant des commandements ou des vœux, cette théorie affirme qu'à chaque cas est associé un acte non-linguistique (désir, espoir, volonté ...) s'effectuant en parallèle de l'utilisation de la phrase, mais survivant à l'énoncé de la phrase. Ce n'est qu'avec Anton Marty (1847-1914) et ses travaux sur l'intimidation, que les mots et les phrases du langage sont devenus des média pouvant déclencher des processus psychiques :

> "*The announcement of one's own psychic life is not the only, nor the primary, thing which is intended in deliberate speaking.That which is primarly intended is much rather a certain influencing or controlling of the alien psychic life of the hearer. Deliberate speaking is a special kind of acting, whose proper goal is to call forth certain psychic phenomena in other beings. In relation to this intention, the annoncement of processes within oneself appears merely as a side-effect.*"

Cependant, malgré les apports et éclaircissements provenant de ces travaux, Marty ne put développer une théorie unifiée du langage telle qu'en ont proposé ensuite Reinach (1883-1917), Austin (1911-1960) et Searle (1932) . Nous allons maintenant étudier plus particulièrement les deux théories complètes sur les actes de langage proposées par Reinach d'une part, et celle initiée par Austin et achevée par son étudiant John Searle. C'est d'ailleurs cette dernière théorie qui a eu le plus d'influence dans le domaine des systèmes multi-agents.

La théorie des actes sociaux d'Adolf Reinach La théorie des *actes sociaux* de Reinach est une des premières à combiner des aspects de logique, de psychologie, d'ontologie et une partie de la théorie du langage de Husserl. Une des préoccupations de Reinach, qui avait une formation en Droit, était la compréhension de l'action de promesse et notamment des obligations qui en découlent. Pour Reinach, la *promesse* ou le fait de *communiquer son intention d'effectuer quelque chose* appartiennent à la catégorie des

actes spontanés(figure 1.2), c'est-à-dire des actes qui implique un sujet exposant une partie de ses activités psychiques (ses intentions), qu'il oppose aux expériences passives telles que ressentir une douleur ou entendre une explosion. Cette classe des actes spontanés se

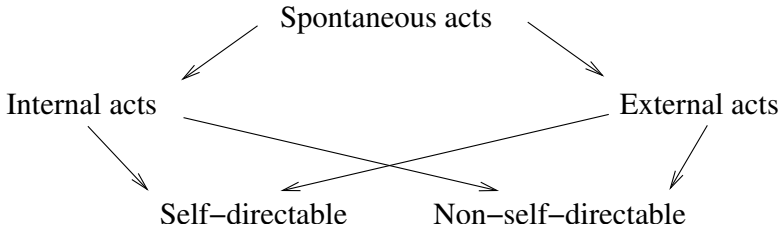

FIG. 1.2 – *Taxonomie des actes spontanés de Reinach*

divise en deux : les actes internes et les actes externes. Un acte interne, est un acte dont l'expression n'est pas essentielle, contrairement à un acte externe, qui n'existe que par son énonciation. De plus, ces classes sont de nouveau scindées en deux, entre les actes qui sont *dirigeables vers soi* (*self-directable*) et ceux qui ne le sont pas (*non-self-directable*). Une propriété particulière des actes externes non dirigeables vers soi, est que leur énoncé doit nécessairement être dirigé vers l'interlocuteur, mais plus encore, il doit être *enregistré* ou *saisi* par le sujet en question. Par exemple, un ordre doit être entendu et compris par celui devant l'effectuer (ce qui n'est pas nécessaire lors d'une excuse, ou d'un pardon). Un ordre est ainsi :

> "*an action of the subject to which is essential not only its spontaneity and its intentionality, but also being directed towards other subjects and its standing in need of being grasped by those subjects. What had been said of commands holds also for requesting, admonishing, questionning, informing, answering, and many other types of act. They are all social acts which are, in their execution, cast by him who executes them towards another subject that they may break into his mind.*"

Ce qui est important à propos de ce type d'action, est que l'énoncé fait partie intégrante de l'acte : sans l'énoncé, l'acte social ne peut se produire. Un acte social implique donc pour Reinach, des :

> "*activities of mind which do not merely find in words their accidental, supplemental expression, but which come to expression in the act of speaking itself and of which it is characteristic that they announce themselves to another by means of this or some similar external appearance.*"

Nous retrouverons dans les travaux d'Austin et de Searle de fortes similitudes avec cette définition : l'acte social est un ensemble constitué de son énoncé, des actes qu'il entraîne et des états mentaux qu'il provoque, ainsi que d'autre états tels que l'autorité des intervenants. Un autre apport de Reinach est d'avoir placé sa théorie dans un cadre plus large que celui de la logique et de la philosophie du langage. En effet, ses travaux indiquent que les actes de langages peuvent être traités *ontologiquement*, dans le cadre

d'une théorie synthétisant non seulement les aspects linguistiques et logique, mais aussi psychologiques, légaux ou relatif à l'action.

Pour Reinach, la théorie des actes de langage est une science descriptive du phénomène dans son ensemble. Il se propose donc d'étudier et d'analyser les *essences elles-mêmes*, ou les structures *a priori*. Reinach ne fait pas ici référence aux *essences* de Platon, mais à des structures *nécessaires* et *universelles*, auxquelles nous aurions accès grâce à une fonction cognitive particulière. Ainsi, pour Reinach (et il suit en cela Husserl) le monde *contient* des promesses, des obligations, des réclamations, des ordres, des relations d'autorité, de la même manière qu'il contient des instances biologiques et des espèces logiques tel que les *lion*, les *tigres* ou les *jugements* et les *inférences*. Ces objets se répartissent en deux catégories : les *espèces indépendantes*, dont l'instanciation ne nécessite pas l'instanciation d'autres espèces, et les *espèces dépendantes*, dont les instances ne peuvent exister pour elle même, mais en association avec des instances complémentaires d'autres espèces spécifiques. Un *jugement* est un exemple d'*espèce dépendante* au sens de Husserl : il n'existe qu'en tant que jugement d'un sujet donné (comme un sourire ne sourit que sur un visage). Suivant le même raisonnement, une *promesse* implique une structure caractérisée par une articulation spécifique d'un ensemble d'instances d'espèces : la *promesse* dépend de la *revendication*, de *l'obligation* et de *l'énoncé*. Il existerait ainsi des structures *a priori*, identifiant les dépendances, qu'elles soient nécessaires ou exclusives, compatibles ou possibles[16].

La théorie des actes de langages de John Searle Avant de présenter les travaux de Searle, nous allons étudier les travaux d'Austin, qui a été l'initiateur de cette théorie. Dans son livre *"How to do Things with Words"*, Austin définit sa théorie des actes de langages, et plus particulièrement le concept de performatif : dire quelque chose est faire quelque chose. En effet, lorsque j'énonce "Je promets p" (avec p, un contenu propositionnel), j'effectue un acte de promesse qui est différent de l'énoncé d'un fait qui peut être vrai ou faux. Ainsi, Austin fait une distinction entre les performatifs et les informatifs, des énoncés tentant de décrire la réalité et pouvant être jugés comme vrai ou faux. Pour Austin, lorsque l'on parle, on crée des réalités sociales dans certains contextes. Par exemple, l'utilisation d'un performatif tel que "Je vous déclare mari et femme" dans le contexte d'un mariage, est une création de réalité sociale, qui dans ce cas se rapporte au couple marié.

Austin décrit trois caractéristiques du langage, qui commencent par les briques de base que sont les mots et qui se terminent avec les effets que produisent ces mots sur l'audience. Il y a d'abord les actes locutoires (d'énonciation) :

> "*a peu près équivalent à l'énoncé d'une certaine phrase avec un certain 'sens', au sens classique*"

Ensuite viennent les actes illlocutoires tels que :

> "*informer, ordonner, avertir ..., i.e. des énoncés qui ont une certaine force (conventionnelle)*"

16. Le lecteur familié avec les systèmes multi-agents ne pourra que s'étonner du caractère innovant de ces thèses développées à la fin du XIXième siècle, qui sont si proches de problématiques actuelles en informatique.

,

Et finalement, les actes perlocutoires :

> "*ce que nous causons ou provoquons en disant quelque chose, tel que convaincre, persuader, dissuader et même surprendre ou tromper.*"

Austin s'est concentré sur les actes illocutoires, soutenant que ces actes contiennent la "force" d'un énoncé et démontrent ainsi leur nature performative. Par exemple, dire "Ne cours pas avec des ciseaux" a la force d'un avertissement dans un certain contexte. Cet énoncé pourrait être construit de manière à faire apparaître explicitement le performatif : "Je t'avertis : ne cours pas avec des ciseaux". Cet énoncé n'est ni vrai, ni faux. A la place, il crée un avertissement. En entendant cet énoncé, et en le comprenant comme un avertissement, l'auditeur est averti, ce qui ne signifie pas pour autant qu'il doit ou voudra agir d'un manière particulière vis-à-vis de cet avertissement.

John Searle affirme que l'acte illocutoire est :

> "*l'unité minimale complète de la communication linguistique humaine. Lorsque l'on se parle ou s'écrit, nous effectuons des actes illocutoires(Mind 136)*"

Mais plus encore, les actes illocutoires sont nécessairement effectués *intentionnellement*. Ceci signifie que la communication est basée sur une croyance mutuelle que lorsqu'une personne s'adresse à une autre, elle s'attend à ce que l'auditeur comprenne son intention illocutoire. Ainsi, les actes illocutoires sont nécessairement intentionnels, et n'existent que pour l'effet de perlocution qu'ils induisent.

Searle a ainsi proposé le schéma suivant :

- l'acte d'énonciation (énoncé des mots, des phrases) : la syntaxe,
- l'acte propositionnel de référence ou de prédication (déclarer des faits ou des jugements) : la sémantique,
- l'acte illocutoire (la "force" de l'énoncé : ordonner, promettre, interroger) : la pragmatique,
- l'acte perlocutoire (les actions/réactions induites) : l'effet subjectif,

et précisé que les actes d'énonciation, de proposition et d'illocution, ne sont pas des entités séparables : ils sont effectués simultanément :

> "*I am not saying, of course, that these are separate things that speakers do, as it happens, simultaneously, as one might smoke, read and scratch one's head simultaneously, but rather that in performing an illocutionary act one characteristically performs propositional acts and utterance acts.*"

Searle a ensuite défini une classification des actes illocutoires :

- informatifs : énoncés décrivant une réalité, qui peut être jugée comme vraie ou fausse,
- directifs : énoncés tentant de faire effectuer à l'auditeur les actions du contenu propositionel,
- promissifs : énoncés engageant l'orateur à effectuer les actions du contenu propositionnel,
- expressifs : énoncés exprimant un état de l'orateur,
- déclaratifs : énoncés modifiant la réalité en la représentant comme ayant changé.

Ces différentes catégories peuvent s'illustrer de la manière suivante : "X pèse 82 kilos" est informatif, "Ferme la porte !" ou "Quelle heure est-il" sont des actes directifs, "Je ferai ce travail" est promissif, "Je suis malade" est expressif, et "Je te déteste" est déclaratif.

Les langages de communication entre agents

Les travaux de Searle ont connu un écho important au sein de la communauté d'intelligence artificielle distribuée, notamment suite au projet *Knowledge Sharing Effort*, KSE. Ce projet, initié au début des années 1990, avait pour objectif la création d'un environnement où des agents logiciels pourraient effectivement communiquer, partager et échanger de l'information et des connaissances. Les deux points essentiels identifiés lors de ce projet sont la gestion du partage de connaissances et l'importance des protocoles de communication entre des modules de connaissances répartis. Ces travaux ont abouti sur deux langages : KIF et KQML. KIF, *Knowledge Interchange Format*, est un langage d'externalisation de connaissances : il facilite la transformation d'un formalisme de représentation des connaissances vers un autre. Ce langage possède une sémantique déclarative, il est logiquement complet (*comprehensive*), et propose un niveau méta pour la représentation de connaissances à propos de connaissances.

Ce problème de représentation des connaissances et surtout de leur signification est intimement lié à la notion d'ontologie, c'est-à-dire au sens à accorder aux mots d'un langage. Nous reviendrons un peu plus tard sur cette notion d'ontologie, lorsque nous nous intéresserons aux travaux actuellement mené autour de la notion de Web sémantique.

Le langage KQML , *Knowledge Query and Manipulation Language*, définit un format de message et un protocole de traitement de ces messages pour faciliter le partage dynamique de connaissances et les interactions entre agents. Ce langage a été proposé dans un premier article en 1992 [33], puis il a été plus complètement spécifié en 1993[35]. Ce n'est qu'en 1994[49], que la sémantique du langage a été définie formellement en logique propositionnelle du premier ordre. Plus précisément, KQML est inspiré des travaux de Searle sur les actes de langages et utilise la notion de performatif pour indiquer la force illocutoire des messages échangés entre agents (i.e. entre bases de connaissances).

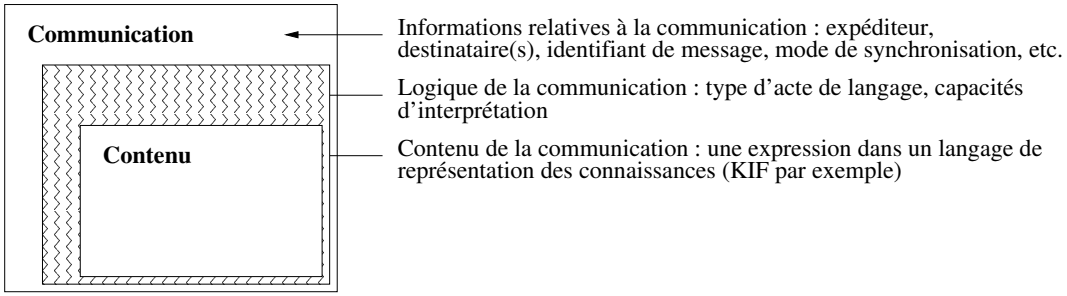

FIG. 1.3 – *Les trois niveaux d'encapsulation d'un message KQML*

La figure 1.3 illustre les trois couches qui constituent un message KQML (cette figure est une reproduction de la figure 3 de l'article [33]). Un message KQML est ainsi composé d'une expression encapsulée dans une *enveloppe de message*, elle-même encapsulée dans

une *enveloppe de communication*. La couche de *contenu* est généralement constituée d'une expression dans un langage de représentation de connaissances. Ce langage peut être KIF, INTERLINGUA, PROLOG ou tout autre langage dédié ou généraliste. Comme le contenu des messages est opaque pour KQML, la couche *message* ajoute les informations nécessaires à l'interprétation du contenu : le langage ainsi que l'ontologie utilisée, et l'acte de langage associé au message. Le dernier niveau, celui de la communication, regroupe les informations de plus bas niveau concernant l'identification de l'émetteur et du ou des récepteurs, ainsi que des paramètres propre à la couche de transport (synchrone, asynchrone, qualité de service ...).

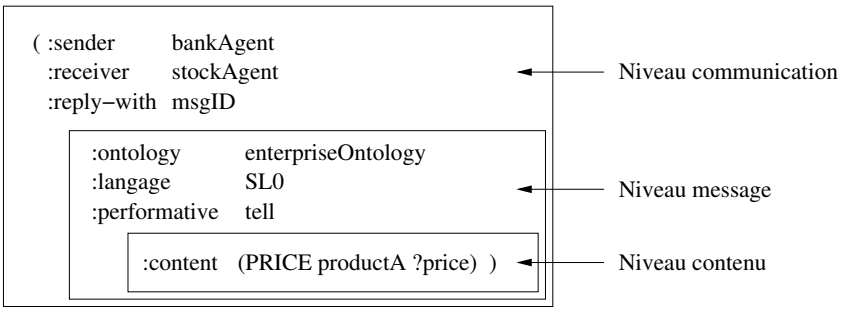

FIG. 1.4 – *Un exemple de message KQML illustrant les trois niveaux d'encapsulation*

La figure 1.4 illustre un message KQML en mettant en perspectives les trois couches :

- *Communication* : ce sont les champs gérant le routage et la conversation (expéditeur, destinataire(s), identifiant de message et de conversation)
- *Message* : les paramètres permettant l'interprétation du contenu (ontologie, langage et performatif)
- *Contenu* : dans cet exemple, une expression SL0.

Bien que de nombreuses applications aient été réalisées avec KQML, notamment au sein de KSE, le langage n'est pas vraiment devenu un standard. Il est probable que la complexité d'implémentation d'un système qui réaliserait les objectifs de KQML est un frein à son développement. De plus, le manque de consensus autour d'un outil ou d'un langage pour la gestion des ontologies ne facilite pas la tâche. Finalement, le projet initié par des universitaires sous l'égide du DARPA[17], n'a pas réussi à fédérer universitaires et industriels autour d'un projet commun. Ainsi, la plupart des applications utilisant KQML ont une architecture *ad hoc*, c'est-à-dire ne traitant qu'un sous-ensemble des fonctionnalités de KQML.

Le langage Fipa-Acl Une autre initiative importante a débuté en 1996, avec la création d'une association à but non lucratif regroupant universitaires et industriels pour promouvoir et favoriser le développement des technologies orientées agents. La FIPA, *Foundation for Intelligent Physical Agent*, propose donc un ensemble de spécifications dont le but est

17. Pour plus d'informations sur le DARPA, *Defense Advanced Research Projects Agency*, se reporter à leur site : `http://www.darpa.mil/`

de :

> "*promouvoir des technologies et des spécifications qui facilitent l'interopérabilité de systèmes d'agents intelligents dans un environnement commercial ou industriel*"

Cette association regroupe actuellement une dizaine d'universités et plus de cinquante entreprises, avec notamment d'importantes sociétés tel que Hewlett Packard, IBM, Sun, Motorola, Intel, Boeing, Fujitsu, Mitsubishi, NEC, Pioneer, Toshiba, British Telecom, France Telecom, Siemens, la SNCF... D'autre part, les différents groupes de travail sont en relation avec d'autres groupes de travail se concentrant sur les technologies orientées agents, tels que l'Omg *Agents Working Group*, le W3c *WebONT Working group*, *Agent UML* ou encore *Java Agent Services* (nous reviendrons plus tard sur ces initiatives).

Pour parvenir à ses fins, la Fipa s'organise autour de deux axes : un aspect technologique et un autre applicatif. Du point de vue technologique, la Fipa produit un ensemble de spécifications concernant les langages de communication entre agents, avec Fipa-Acl (un équivalent de Kqml), les langages de contenu (sémantique), les protocoles d'interactions (aspects conversationnels), ou encore les services minimaux que doit fournir une plateforme pour être compatible (transport des messages, systèmes de pages blanches/jaunes). Du point de vue applicatif, la Fipa définit des services pour des marchés verticaux (assistants personnels, assistants de voyage) ou horizontaux (support des applications nomades, gestion de flux multimédia, gestion du réseau).

Nom de la plateforme	Université ou entreprise créatrice	Pointeur sur leur site
Agent Development Kit	Tryllian BV	`http://www.tryllian.com/`
Zeus	Brittish Telecom	`http://193.113.209.147/projects/agents/zeus/`
Fipa-Os	Emorphia	`http://fipa-os.sourceforge.net/`
April Agent Platform	Fujitsu	`http://sf.us.agentcities.net/aap/`
Grasshopper	IV++ Technologies AV	`http://www.grasshopper.de/`
JACK Intelligent Agents	Agent Oriented Software	`http://www.agent-software.com/`
Java Agent Development Environment (Jade)	TILAB (CSELT)	`http://sharon.cselt.it/projects/jade/`
Lightweight Extensible Agent Platform (Leap)	Motorola	`http://leap.crm-paris.com/`

Fig. 1.5 – *Quelques unes des plateformes répondant aux spécifications de la* Fipa.

Ce n'est que récemment (le 6 décembre 2002), que les spécifications fondamentales, celles concernant les ACLs et les services fournis par la plateforme, ont été promus au rang de standard par le comité FIPA. C'est un pas important qui marque la stabilité et le développement de la communauté FIPA. En effet, il existe maintenant plus d'une dizaine de plateformes conformes aux spécifications, et certaines d'entre elles sont librement accessible, voire Open Source (c'est-à-dire pouvant être plus facilement réutilisées pour construire de nouveaux systèmes). Le tableau 1.5 présentent les plateformes compatible FIPA les plus connues. Parallèlement à ces efforts, un réseau de telles plateformes a été mis en place dans le cadre d'un projet européen, AGENTCITIES, pour fournir un outil d'évaluation et un environnement de démonstration. Ainsi, bien que suivant les traces de KQML, les spécifications FIPA sont en train de devenir un standard *de facto* pour l'utilisation de technologies orientées agents dans les milieux industriels et commerciaux.

La représentation des connaissances

Nous avons vu que la principale caractéristique des ACLs est l'échange et le partage de connaissances. Cependant, nous n'avons pas détaillé comment ces connaissances peuvent être modélisées et manipulées. Nous allons maintenant présenter une histoire succincte du domaine s'attachant à la représentation des connaissances, avant de mettre en perspective les travaux actuels, et l'impact qu'ils pourraient avoir.

Dans leur article intitulé *What is knowledge representation*[24], Davis, Shrobe et Szolowits affirment qu'un système de représentation des connaissances (*Knowledge Representation*, ou KR) joue cinq rôles bien distincts :

- un substitut qui est utilisé par une entité pour déterminer des conséquences en raisonnant sur une représentation plutôt qu'en agissant sur le monde.
- un ensemble d'engagements ontologiques, c'est-à-dire une réponse à la question : en quels termes devrais-je penser le monde ?
- une théorie partielle de l'intelligence rationnelle, exprimée en terme de trois composantes : (i) la représentation des concepts fondamentaux de l'intelligence rationnelle, (ii) l'ensemble des inférences que la représentation sanctionne; et (iii) l'ensemble des inférences qu'elle recommande.
- un environnement simulant le raisonnement au moyen de calculs pragmatiques et praticables. Une manière de rendre ce calcul efficace est de s'appuyer sur l'organisation de l'information imposée par la représentation dans le but de faciliter les inférences recommandées.
- un média d'expression humaine, ie. un langage dans lequel nous pouvons décrire le monde.

Selon les solutions proposées, ces cinq rôles prennent plus ou moins d'importance. Ce que nous conserverons de ces cinq critères, c'est qu'un KR sert à modéliser, représenter et raisonner sur un ensemble de connaissances. Avant de présenter les outils que nous utiliserons pour représenter les fonctions ou connaissances de nos systèmes multi-agents, nous allons brièvement présenter les principaux systèmes qui ont marqué le domaine de la représentation des connaissances. Cette présentation succincte a pour but de convaincre le lecteur de la convergence actuelle des différentes approches au sein des travaux effectués

autour de la notion de Web Sémantique.

Les différents formalismes de représentation De nombreux formalismes ont été définis pour tenter de répondre aux problématiques de la représentation des connaissances. Nous reprendrons ici la taxonomie proposée par Masini[53] et al. dans leur ouvrage "Les langages à objets"[53] : la logique, les procédures, les réseaux sémantiques et les langages de frames.

La logique a depuis longtemps servi de formalisme pour le raisonnement. Ce raisonnement s'effectuant sur des faits, il est nécessaire de pouvoir exprimer et représenter des connaissances avant de pouvoir inférer quoique ce soit. Le langage PROLOG est un bon exemple de cette approche : il repose sur la logique du premier ordre pour représenter les connaissances et raisonner dessus. Tout un ensemble de systèmes experts ont été conçus à l'aide de PROLOG, en utilisant une base de connaissances constituée de règles de production ou de clauses PROLOG en conjonction avec un moteur d'inférence. Cependant, la rigidité de la logique classique (consistance et complétude) ne s'adapte pas toujours très bien aux besoins de l'intelligence artificielle où les informations sont souvent incomplètes, voir contradictoires !

Les procédures ont été utilisées par Carl Hewitt dans son projet PLANNER, qui était un logiciel de démonstration automatique. Un programme écrit en PLANNER est constitué d'assertions, qui représentent les faits connus, et de théorèmes, qui sont des procédures permettant de déduire de nouveaux faits. Ce système a influencé de nombreux travaux en représentation des connaissances, particulièrement après avoir déclenché une importante controverse entre l'approche déclarative (à la PROLOG) et l'approche procédurale (à la PLANNER). Cette controverse a abouti notamment à l'élaboration des langages de frames.

Les réseaux sémantiques ont été parmi les premiers systèmes de représentation des connaissances basés sur des mécanismes d'héritage. Un réseau sémantique est un système à héritage simple ou multiple reposant généralement sur la théorie des prototypes[18]. Un réseau est constitué de *nœuds typés*, qui représentent les concepts, et de *liens*, qui expriment les relations entre les nœuds. Un ensemble d'opérations associées au réseau permet de l'exploiter : création/suppression de nœuds ou de liens, parcours du graphe des relations, mécanismes d'héritage et de filtrage ...

Les langages de frames initialement introduits par Marvin Minsky[61] représentent une étape importante dans le domaine de la représentation des connaissances. Dans son ouvrage *"Society of Mind"*[61], Minsky part de l'idée que les humains disposent d'un certain nombre de structures pré-existantes représentant des modèles de situation qu'ils cherchent à adapter aux nouvelles situations. Il nomme ces structures *frame*s et les définit comme un réseau hiérarchisé de nœuds et de relations. Les nœuds des niveaux supérieurs décrivent des informations toujours vraies dans le domaine considéré, tandis que les nœuds inférieurs possèdent des *attributs* (ou *slots*) auxquels sont associées des valeurs (figure 1.6).

La notion de frame a ensuite évolué, et s'est cristallisée autour des notions de *frames*,

18. Dans les langages objets à prototypes, la notion de classe n'existe pas, à la place, la notion de prototype caractérise un membre *moyen* d'une catégorie qui possède les caractéristiques les plus fréquentes. Un prototype représente ainsi une connaissance par défaut. Des langages tels que SELF (`http://research.sun.com/research/self/language.html`) illustrent l'intérêt de cette approche.

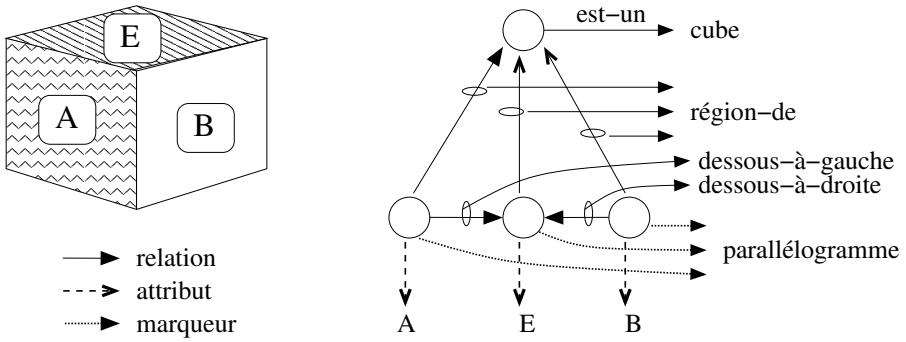

FIG. 1.6 – *Un frame représentant l'image d'un cube (extrait de[61])*

attributs et *facettes*. Un *frame* étant une entité générique constituée d'*attributs*. Ces *attributs* sont décrits par un ensemble de *facettes* possédant des valeurs. La notion de *facette* permet de représenter différents points de vue sur un même *attribut*. Dans ces langages, les *frames* sont généralement des prototypes, et leur programmation est dirigée par les accès. En effet, les *frames* sont manipulés au travers d'un nombre restreint de primitives (lecture ou écriture de la valeur d'un *attribut* et ajout ou retrait d'un *attribut*), et des procédures associées aux *attributs* (des *réflexes* ou *démons*) peuvent être déclenchées lorsque l'on y accède.

Ces différentes approches ont été poursuivies avec l'apparition de langages hybrides, qui ont tenté de marier frames et classes. Ce n'est que récemment, que la communauté travaillant sur la représentation des connaissances a connu un regain d'intérêt pour d'autres domaines avec l'émergence du concept de Web Sémantique et des concepts et technologies qui lui sont associés

D'Internet au Web Sémantique Actuellement, Internet est constitué d'un ensemble de services permettant à ses utilisateurs de rechercher des informations et de les exploiter ensuite. L'idée principale du Web Sémantique consiste à considérer que dans les années à venir, il y aura plus d'utilisateurs logiciels (ie. des programmes ou "agents intelligents") que d'utilisateurs humains. Il est donc nécessaire de structurer beaucoup plus les documents pour pouvoir leur associer un minimum de sémantique, ce qui rendra possible leur interprétation par des logiciels. Cette idée s'appuie sur plusieurs bouleversements technologiques : l'importance commerciale qu'a acquis le réseau des réseaux, l'omniprésence du langage XML dans les échanges de données structurées et l'engagement du W3C sur les technologies reposant sur XML comme RDF et plus récemment OWL.

Le but à terme du Web Sémantique est de ne plus modéliser les informations sous formes d'entités logicielles (ie. documents structurés) tel que l'on procède actuellement, mais de modéliser des mots (ie. des ressources liées par une sémantique). L'information n'étant plus décrite de manière purement syntaxique mais sémantique, sa pérennité devrait être accrue. L'impact attendu de cette technologie est donc de changer complètement l'organisation des données en transformant les interactions possibles des logiciels entre eux, ainsi que l'expressivité des interactions avec les utilisateurs. Tim Berners-Lee[10][10]

résume cela en affirmant : *"Your life is a Web. Your data is a Web"* [19].

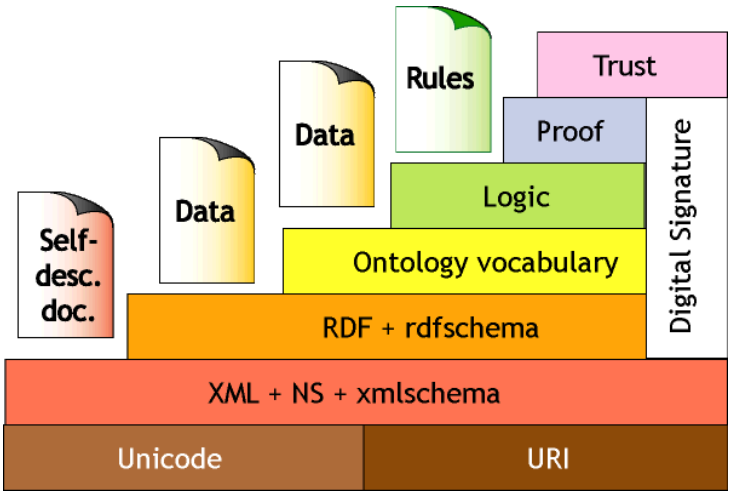

FIG. 1.7 – *Les couches* XML *constituant l'infrastructure du Web Sémantique*

Le Web Sémantique est un empilement de couches qui successivement abstraient et conceptualisent les informations qu'elles décrivent. La figure 1.7 illustre cette empilement, avec XML comme langage d'encodage (et les XML SCHÉMA pour spécifier la structure de ces documents), RDF pour ajouter la gestion des ressources et leur relations (et RDF SCHÉMA pour spécifier les propriétés ainsi que les classes de ressources RDF); et finalement OWL pour la sémantique et le raisonnement. Ce dernier langage est le plus intéressant car il introduit les possibilités de définition d'ontologies et définit les modalités d'interprétation de ces ontologies. Pour pallier aux difficultés d'implémentation des fonctionnalités de raisonnement, OWL est constitué de trois langages d'expressivité croissante [20] :

- OWL LITE permet de définir des hiérarchies de classification reposant sur des contraintes simples. Par exemple, les cardinalités sont limitées aux seules valeurs 0 ou 1.
- OWL DL fournit le maximum d'expressivité tout en garantissant la complétude (tous les théorèmes sont prouvables) et la décidabilité de l'arrêt (les calculs s'effectuent en un temps fini). Le DL est l'acronyme de *Description Logic* qui est une famille de logiques particulièrement adaptée à l'implémentation des mécanismes d'inférence sur les ontologies OWL.
- OWL FULL est le plus expressif des trois langages, mais perd en contrepartie la garantie d'arrêt des calculs ... Il est improbable que des algorithmes d'inférence puissent supporter l'ensemble des fonctionnalités d'OWL FULL.

Chacun de ces langages est une extension de son prédécesseur, à la fois sur ce qui peut être exprimé et ce qui peut être inféré. Ainsi, les relations suivantes qui expriment une compatibilité ascendante sont garanties (pas leur inverse !) :

- chaque ontologie OWL LITE est une ontologie OWL DL valide,

19. Les idées développées dans ce paragraphe proviennent d'une interview de Tim Berners-Lee (le directeur du W3C et figure de proue du Web Sémantique) : `http://www.adtmag.com/article.asp?id=7662`.
20. Source : OWL *Web Ontology Language Overview* : `http://www.w3.org/TR/owl-features/`

- chaque ontologie Owl DL est une ontologie Owl Full valide,
- chaque conclusion Owl Lite valide est une conclusion Owl DL valide,
- chaque conclusion Owl DL valide est une conclusion Owl Full valide.

Cette décomposition facilite d'une part l'implémentation progressive des fonctionnalités de raisonnement, et d'autre part la prise en main et l'apprentissage du langage. Les primitives de création d'ontologie pour OWL Lite sont les suivantes :

RDF Schema Features	(In)Equality	Property Characteristics
– Class – rdf:Property – rdfs:subClassOf – rdfs:subPropertyOf – rdfs:domain – rdfs:range – Individual	– equivalentClass – equivalentProperty – sameIndividualAs – differentFrom – allDifferent	– inverseOf – TransitiveProperty – SymmetricProperty – FunctionalProperty – InverseFunctionalProperty
Property Type Restrictions	**Restricted Cardinality**	**Header Information**
– allValuesFrom – someValuesFrom	– minCardinality (only 0 or 1) – maxCardinality (only 0 or 1) – cardinality (only 0 or 1)	– imports – priorVersion – backwardCompatibleWith – incompatibleWith
Class Intersection	**Datatypes**	
– intersectionOf		

La création d'une ontologie avec Owl consiste dans un premier à définir un ensemble de classes ainsi que leurs propriétés, puis un ensemble d'instances. Le code Xml ci-dessous présente une ontologie classique (c'est l'ontologie d'exemple développée pour Daml+Oil) qui définit quelques classes et illustre les différents opérateurs disponibles.

```
<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd ="http://www.w3.org/2000/10/XMLSchema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:exd ="http://www.w3.org/TR/@@/owl-ex-dt#"
  xmlns:dex ="http://www.w3.org/TR/@@/owl-ex#"
  xmlns     ="http://www.w3.org/TR/@@/owl-ex#"
>
<owl:Ontology rdf:about="">
  <owl:versionInfo>$Id: owl-ex.rdf,v 1.1 2002/07/29 15:33:03 $</owl:versionInfo>
```

```
  <rdfs:comment>
    An example ontology, with data types taken from XML Schema
  </rdfs:comment>
  <owl:imports rdf:resource="http://www.w3.org/2002/07/owl"/>
</owl:Ontology>
```

Ces premières déclarations permettent de définir les espaces de nommage qui seront utilisés dans le document. Ceci est très pratique, car cela permet l'inclusion d'élément provenant d'autres fichiers déclarés sur le réseau, et cela raccourcit les déclarations des éléments (`rdf` au lieu de `http://www.w3.org/1999/02/22-rdf-syntax-ns`). C'est aussi dans ce préambule que l'on indique différentes méta-données sur l'ontologie, telles que son nom, sa version et les ontologies dont elle dépend. Les espaces de nommage ne correspondent pas aux importations : ils sont utilisés par le parseur XML pour vérifier la validité du document, tandis que la balise `owl:imports` indique au moteur d'inférence qu'il lui faudra charger l'ontologie référencée.

```
<owl:Class rdf:ID="Animal">
  <rdfs:label>Animal</rdfs:label>
  <rdfs:comment>
    This class of animals is illustrative of a number of ontological idioms.
  </rdfs:comment>
</owl:Class>
```

L'exemple le plus simple de déclaration d'une classe de concept en OWL : seul un identifiant est donné. Cet identifiant est unique, et correspond à la concaténation de l'espace de nommage et de l'identifiant RDF, c'est-à-dire pour la classe `Animal` à `http://www.w3.org/TR/@@/owl-ex#Animal`. Le label peut être utilisé par les outils pour faciliter la visualisation. C'est aussi à ce niveau qu'il est possible de gérer l'internationalisation en précisant le langage utilisé dans le contenu de la balise.

```
<owl:Class rdf:ID="Male">
  <rdfs:subClassOf rdf:resource="#Animal"/>
</owl:Class>

<owl:Class rdf:ID="Female">
  <rdfs:subClassOf rdf:resource="#Animal"/>
  <owl:disjointWith rdf:resource="#Male"/>
</owl:Class>
```

Les deux classes de concepts ci-dessus illustrent l'utilisation de l'héritage dans OWL, ainsi que les opérations ensemblistes. La classe `Female` est ainsi définie comme une sous-classe de la classe `Animal` et ses instances doivent être disjointes des instances de la classe `Male`.

```
<owl:Class rdf:ID="Man">
  <rdfs:subClassOf rdf:resource="#Person"/>
  <rdfs:subClassOf rdf:resource="#Male"/>
</owl:Class>
```

```
<owl:Class rdf:ID="Woman">
  <rdfs:subClassOf rdf:resource="#Person"/>
  <rdfs:subClassOf rdf:resource="#Female"/>
</owl:Class>
```

Les deux définitions ci-dessus présentent l'héritage multiple en Owl.

```
<owl:ObjectProperty rdf:ID="hasParent">
  <rdfs:domain rdf:resource="#Animal"/>
  <rdfs:range rdf:resource="#Animal"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasFather">
  <rdfs:subPropertyOf rdf:resource="#hasParent"/>
  <rdfs:range rdf:resource="#Male"/>
</owl:ObjectProperty>
```

Les deux déclarations précédentes sont intéressantes car elles présentent deux moyens permettant de créer des propriétés de concept. Les propriétés en Owl sont définies indépendamment des classes, et ne sont reliées à ces dernières que lorsque le concepteur précise leurs domaines et co-domaines. Ainsi, la première propriété établit une relation entre la classe `Animal` et elle-même qui peut se traduire par l'assertion suivante : "*un animal a un parent qui est aussi un animal*". La seconde propriété est aussi très intéressante car elle illustre la possibilité de spécialisation des propriétés. Ainsi, la propriété `hasFather` spécialise la propriété `hasParent` en appliquant une restriction sur le co-domaine.

```
<owl:DatatypeProperty rdf:ID="shoesize">
  <rdfs:comment>
    shoesize is a DatatypeProperty whose range is xsd:decimal.
    shoesize is also a FunctionalProperty (can only have one shoesize)
  </rdfs:comment>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/10/\xml{}Schema#decimal"/>
</owl:DatatypeProperty>
```

Cette déclaration introduit le second aspect fondamental d'Owl : le lien avec les types de données définis dans les Xml Schéma. Ce lien est d'autant plus important que Wsdl les utilise aussi. D'ailleurs, une convergence s'initie entre Owl et Wsdl avec la prochaine version du standard Wsdl.

```
<Person rdf:ID="Peter">
  <rdfs:comment>
  Peter is an instance of Person. Peter has shoesize 9.5 and age 46. From the
  range restrictions we know that these are of type xsd:decimal and
  xsd:nonNegativeInteger respectively. Peter also has shirtsize 15, the type
  of which is the union type clothingsize; no discriminating type
  has been specified, so the value may be either a string or an integer.
  </rdfs:comment>
  <shoesize>9.5</shoesize>
```

```
  <age>46</age>
  <shirtsize>15</shirtsize>
</Person>
```

Cette dernière déclaration illustre la phase d'instanciation d'un concept avec Owl. Cette instanciation consiste à définir une nouvelle resource Rdf, et à valuer les différentes relations dépendantes de ce concept[21].

Pour conclure, il est important de préciser qu'Owl est passé depuis le 19 août 2003 du statut incertain de *Working Draft* à celui plus stable de *Candidate Recommandation*. Il est donc maintenant acquis que ce langage deviendra dans moins d'un an une recommandation du W3c au même titre que Xml. Ce détail du processus de standardisation du W3c a quand même son importance : les éditeurs d'outils et surtout les développeurs de moteurs d'inférence peuvent maintenant initier leur développement car peu de changements au niveau de la définition du langage se produiront avant son passage en tant que standard. De plus, l'intégration d'Owl avec Wsdl, bien qu'à un stade très préliminaire pour l'instant, est une garantie supplémentaire de diffusion rapide du langage Owl et surtout de services sémantiquement augmentés.

1.2.2 L'héritage du génie logiciel

Comme toute discipline d'ingénierie, le domaine des systèmes multi-agents repose sur un certain nombre d'avancées qui ont été effectués dans le domaine des langages et des outils de conception, de développement et de réalisation de logiciels. Nous allons donc présenter dans cette partie une brève histoire des principales familles de langages, ainsi que leurs concepts. Nous étudierons ensuite les méthodologies utilisées actuellement dans le domaine des systèmes multi-agents. Enfin, nous nous intéresserons à une approche particulière dans le domaine des systèmes multi-agents, qui constitue l'une des bases de notre proposition : la programmation orientée interaction.

Evolution des paradigmes de programmation

Depuis les débuts de l'informatique, les créateurs de langages ont toujours tentés de faciliter le travail des développeurs, en leur fournissant des abstractions les éloignant du fonctionnement interne de l'ordinateur. Bien sûr, les langages d'assemblage ne fournissaient pas énormément d'abstraction, mais les langages de plus haut niveau tel qu'Algol ou le C ont allégé la charge des développeurs en leur proposant des idiomes tels que : la définition de blocs d'instructions qui ont introduit la notion de portée de variable, la création de structure (de type) de données définissables par l'utilisateur, la notion de sous-routine et de procédure ... Ces concepts proviennent généralement d'une idée fondamentale : le paradigme du langage. Cette notion de paradigme correspond aux métaphores mises en avant par les créateurs du langage, lorsqu'ils effectuent des choix de conception et définissent les fonctionnalités de base de leur langage.

21. Il faut souligner qu'il serait plus intéressant dans cet exemple d'avoir un identifiant unique non signifiant pour un humain (une clé de hachage), et de reporter le nom de la personne dans une relation.

Des langages structurés aux objets Les premiers langages structurés reposaient sur le paradigme de "*la recette de cuisine*". Les préoccupations étaient principalement algorithmique, et les primitives proposées étaient : la notion de structure de contrôle, la notion de variable et de type ... Puis, l'apparition de la programmation modulaire a permis de faire progresser le domaine du génie logiciel, en appliquant la stratégie "*diviser pour mieux régner*". La notion de procédure a été introduite, associée à la métaphore du "*petit monsieur*" sachant traiter un problème spécifique. La notion de librairie, ou de bibliothèque, fut ensuite intégrée pour regrouper logiquement un ensemble de fonctionnalités cohérentes.

Un autre bouleversement s'est produit à la fin des années 1970 avec l'apparition des premiers langages orientés objets : d'abord grâce à MODULA, puis surtout avec SMALLTALK. Un des principes fondamental de ces langages est de regrouper les traitements et les données au sein d'une même entité : un objet. Il n'existe pas de définition formelle de "ce qu'est un objet", néanmoins voici celle proposée par Booch[11] :

> "*Un objet possède un état, un comportement et une identité; la structure et le comportement d'objets similaires sont définis dans leur classe commune; les termes instances et objets sont interchangeables[11].*"

Comme cette définition le souligne, un autre pas a été effectué vers une *agentification* de la structure d'un programme : ce dernier correspondant effectivement à un ensemble d'objets en interaction. D'autre part, la programmation orientée objets a aussi apporté d'importantes propriétés sur la gestion du cycle de vie des logiciels, en terme d'évolutivité et de réutilisation grâce à la notion d'héritage et de polymorphisme.

Des objets aux composants logiciels Plus récemment, au début des années 1990, un raffinement de la notion d'objet a été proposé pour prendre en compte la généralisation de la nature distribuée des environnements informatiques. Le développement des réseaux et de l'informatique grand public, a ouvert de nouvelles perspectives en terme de conception de logiciels *répartis*, c'est-à-dire s'exécutant sur plusieurs ordinateurs formant un réseau. Avec cette nouvelle infrastructure, les objets qui constituaient les blocs de base de programmes monolithiques, sont devenus des citoyens à part entière, pouvant offrir leur services à d'autres objets distants.

Cette notion d'appel distant, et d'autres propriétés liées à de bonnes pratiques de génie logiciel, ont amené le développement de la notion de composant. Un composant est un élément logiciel réutilisable, qui peut être combiné avec d'autres composants se trouvant sur le même ordinateur ou sur un réseau d'ordinateurs. Les composants peuvent être déployés sur les différents serveurs d'un réseau et communiquer entre eux pour former une application. Une des propriétés intéressantes des environnements à base de composants est qu'ils forcent l'utilisateur à adopter l'utilisation de bonne pratique de conception. En effet, les composants existent au sein de conteneurs qui gèrent leur cycle de vie. Ce transfert de responsabilité du composant à son hôte est une illustration du *design pattern* appelé *inversion of control*[22].

22. Des détails sur ce patron de conception peuvent être trouvés sur le site du projet Avalon : `http://jakarta.apache.org/avalon/framework/inversion-of-control.html`

Les Services Webs Depuis une dizaine d'année, cette notion de composant logiciel a connu un important écho dans le milieu industriel, dans un premier temps grâce à CORBA, et plus récemment avec l'avènement des Services Webs. Indépendamment de l'évident effet de mode, les Services Webs représentent une nouvelle avancée dans la conception de systèmes distribués. Il n'existe pas encore de définition acceptée des Services Webs, cependant plusieurs entreprises ont proposé la leur :

Sun	**IBM**	**Microsoft**
Un Service Web décrit une fonctionnalité spécifique proposée par une entreprise, généralement via Internet, dans le but de fournir un moyen pour d'autres entreprises d'utiliser ce service.	Les Services Webs sont des applications modulaires **auto-suffisantes et auto-descriptives** qui peuvent être combinées avec d'autres Services Webs pour créer des produits innovants, des processus, ou des chaînes à forte valeur ajoutée. Les Services Webs sont des applications Internet qui s'acquittent d'une tâche ou d'un ensemble de tâches spécifiques en relation avec d'autres Services Webs et créent ainsi des **workflow complexes**.	Un Service Web est un élément de logique applicative fournissant des données et des services à d'autres applications. Ces applications accèdent aux Services Webs via des protocoles Internet et des formats de données standardisés tels que HTTP, XML et SOAP, sans avoir à se préoccuper de l'implémentation de chacun des services. Les Services Webs combinent les meilleurs aspects de la conception à base de composants et d'Internet.

FIG. 1.8 – *Définition de la notion de Services Web par* SUN, IBM *et* MICROSOFT

Que pouvons nous retenir de ces définitions "commerciales" ? Les aspects fondamentaux des Services Webs sont leur auto-description (ce qui facilite les interactions entre les utilisateurs et les services existants), leur auto-suffisance (c'est-à-dire facilement déployables car reposant généralement sur une architecture à base de composants), et aussi leur implémentation qui s'appuie sur l'utilisation de protocoles standards ouverts. Ainsi, en essayant de prendre le meilleur de ces définitions trop *marketing*, on pourrait définir un Service Web comme :

> "*Un Service Web est un composant auto-descriptible et auto-suffisant, qui est accessible au travers d'une interface neutre et ouverte et qui interagit en utilisant des protocoles standards et ouverts.*"

Cette définition regroupe les principales caractéristiques des Services Webs. La majorité des technologies orientées composants possédaient déjà la plupart de ces attributs : elles prennent en compte l'aspect distribué, elles ont une interface ouverte (IDL : *Interface Description Language* pour CORBA [23]), et elles interagissent via des protocoles standardisés et ouverts (IIOP : *Internet Inter Orb Protocol* pour CORBA)

On peut se demander ce qu'il y a de si nouveau avec les Services Webs, et surtout comment ils se comparent à CORBA ? Il semblerait que les raisons techniques ne sont pas en cause, mais que des choix économiques et politiques de grosses entreprises aient ralenti le développement de CORBA en faveur des Services Webs. Les initiatives partiellement concurrentes de technologies à base de composants tels que DCOM de MICROSOFT, ou les

23. Il existe cependant quelques différences entre une spécification reposant sur IDL ou XML : IDL repose sur les concepts de la programmation procédurale (i.e. définition de structures de données et de signatures de méthodes), tandis que XML est un langage de description qui n'est pas lié à un type de programmation, mais plus à la création et à la gestion de documents.

Ejb de Sun ont aussi eu un impact significatif sur l'utilisation de Corba. Si aujourd'hui un consensus s'établit autour des Services Webs, c'est principalement parce que Microsoft et Ibm sont les initiateurs de Soap, qui constitue l'une des briques de base des Services Webs (Soap peut être vu comme un appel distant encodé en Xml).

En dépit de ces aspects économiques et politiques, il y a quelques particularités qui peuvent au moins partiellement expliquer l'attrait et le développement rapide des Services Webs. Une des raisons principales semble être l'utilisation du langage Xml aux différents niveaux du système : pour la description des interfaces de services avec Wsdl, pour l'enregistrement et la recherche de service avec Uddi, et pour le transport des messages avec Soap. Cette omniprésence de Xml fournit des représentations neutres pour toutes les entités du système. Ce choix technologique est renforcé par la disponibilité de nombreux outils pour travailler avec Xml (toutes les entreprises majeures sus-citées ainsi que de nombreux projets *open source* fournissent des analyseurs et autres Api de haut niveau pour manipuler et transformer des documents Xml), et un mouvement progressif des acteurs de l'Edi[24] vers l'intégration de Xml.

L'ensemble de ces facteurs économiques et techniques indiquent clairement que le langage Xml est devenu un standard *de facto* pour l'échange de données, non seulement dans le domaine de l'informatique, mais aussi au niveau d'autres industries.

Un autre point qu'il est important de noter concernant le choix d'une architecture à objets distribués ou à base de Services Webs : ces technologies peuvent être complémentaires et ne sont pas forcément exclusives. Les technologies telles que Corba ou les Ejb sont nécessairement orientées objets (par définition), alors que les Services Webs sont plutôt orientées documents. Cette distinction implique que ces technologies ne sont pas réellement destinées aux même domaines d'applications :

Il existe aussi une différence de granularité : les Services Webs sont a priori plus haut niveau que les composants. Bien sûr, il est possible d'implémenter un Service Web avec une technologie à base de composants, mais généralement il n'y aura pas une bijection entre un Service Web et un composant. En effet, les composants logiciels ont une granularité assez fine pour permettre leur réutilisation dans différents contexte, tandis que les Services Webs tendent à proposer des services métiers fortement intégrés (et donc beaucoup plus difficilement réutilisables).

Cette courte et non exhaustive présentation des paradigmes de programmation illustre la tendance actuelle vers une *agentification* des concepts de programmation. Cela a débuté avec l'utilisation de métaphores tels que celle du *petit homme*, avant d'être plus clairement identifié avec la notion d'objet (et plus particulièrement d'*objet actif* ou *acteur*). La notion de composant logiciel est ensuite apparue, notamment pour répondre aux besoins de gestion des aspects de distribution, et plus récemment les Services Webs ont insisté sur l'indépendance vis-à-vis des langages d'implémentation (Corba l'avait pourtant réalisé bien avant ...) et la facilité de déploiement. Les applications développées à partir de Services Webs impliquent une disparition de la notion même d'application, en faveur d'une forme d'*écosystème ouverts de services*. Sans être irréaliste, cette notion d'écosystème

24. Edi : *Electronic Data Interchange*, défini par un standard Ansi (`http://www.x12.org/`) et Edifact au travers de l'organisme Un/Ece (`http://www.unece.org/trade/untdid/welcom1.htm`)

Orienté objets	Orienté document
– *Structures de données simples* : "chapeau", "quantité", "liste de couleurs" ou "prix". – *Couplage fort* : Les résultats des invocations n'ont que peu de sens sans le contexte d'invocation. – *Communication synchrones* : A cause du fort couplage, il est nécessaire d'associer les réponses avec les requêtes. De plus, les requêtes sont souvent corrélées (la réponse d'une première requête permettra la création d'une seconde requête).	– *Données complexes structurés auto-descriptives* : un catalogue de produit complet (on sait que c'est un catalogue car le document contient une description de sa structure). – *Couplage faible* : Il n'est pas nécessaire d'associer le catalogue à la requête, car il se suffit à lui même pour ceux désirant le consulter (à condition de comprendre le schéma le décrivant). – *Communication asynchrone* : rendue possible par l'auto-description et le couplage faible.

FIG. 1.9 – *Comparaison des technologies orientées objets vs. orientées documents*

ouvert devient de plus en plus une réalité industrielle au travers de l'informatique pervasive. Le marché des assistants personnels, des téléphones portables, des réseaux sans fil sont quelques exemples démontrant que l'infrastructure est prête et le que le potentiel économique est important. Ce qui fait défaut actuellement, c'est une infrastructure logicielle cohérente et des outils de développement adaptés à ce nouvel environnement. Nous pensons que les paradigmes des systèmes multi-agents et plus précisément la conception orientée agents peut répondre à cette problématique.

La gestion des interactions de composants logiciels Les ADL, *Architecture Description Language*[60] sont des langages de description de systèmes de composants, généralement distribués. Ces langages proviennent des travaux effectués sur les MIL, *Module Interconnection Languages*, mais leur ajoutent le traitement des aspects non fonctionnels (tels que la sécurité, les transactions ...). Il n'existe pas encore de consensus sur "ce qu'est un ADL", mais la définition proposée dans [85] peut constituer un point d'entrée :

> "*An ADL software applications focuses on high-level structure of overall application rather than the implementation details of any specific source module*"

Ainsi, un ADL permet de décrire l'architecture générale d'un système, en détaillant les composants, leur interconnexion et leurs dépendances fonctionnelles. Ils permettent aussi de vérifier une partie de la sémantique de l'assemblage grâce au typage des composants en terme d'interfaces.

C'est une approche déclarative, qui permet lors de la conception de définir l'architecture globale d'une application. Cette description repose sur : les composants ou composites (composants composés de composants), les connecteurs qui définissent les protocoles de communication, de règles d'instanciation et de règles de liaisons. Si des modifications doivent être apportées, le langage de description d'architecture permet de modifier les liaisons entre composants beaucoup plus aisément que lorsque ces liaisons se trouvent *enfouies* au sein des composants.

Une autre approche récente, qui n'est pas un ADL mais qui partage le même souci de *refactoring* pour la description d'architectures logicielles, a été proposée par l'OMG. Le *Model Driven Architecture*, ou MDA, est une approche pour la spécification de systèmes et l'interopérabilité, reposant sur l'utilisation de modèles formels [69]. En MDA, des modèles indépendants de la plateforme (*Platform Independant Model*, ou PIM) sont exprimés dans un langage tel qu'UML. Ces modèles sont ensuite traduits vers un modèle spécifique de plateforme (*Platform Specific Model*, ou PSM) en projetant le PIM vers un langage d'implémentation ou une plateforme d'exécution (le langage C++ ou la plateforme JAVA par exemple). Pour la définition du modèle, le MDA utilise des standards de l'OMG, tels qu'UML (*Unified Modeling Language*), le MOF (*Meta Object Facility*), XMI (XML *Medata Interchange*) et CWM (*Common Warehouse Metamodel*).

Cette approche représente une évolution significative de l'OMG par rapport à l'intéropérabilité entre systèmes. En effet, avant MDA l'interopérabilité reposait principalement sur les standards et services CORBA, c'est-à-dire au niveau d'interfaces standards de composant. Avec MDA, les modèles formels de systèmes se trouvent au cœur de l'interopérabilité. Et surtout, à la vitesse à laquelle évoluent les technologies et plateformes de déploiement, une approche basée sur les modèles, grâce à la séparation entre la spécification et le langage d'implémentation, devrait faciliter le *refactoring* d'architecture ainsi que le redéploiement sur différentes plateformes.

Les méthodologies orientées agents

Suite au développement des systèmes multi-agents, de nombreuses méthodologies sont apparues, telles que DESIRE[12], TROPOS[16], PASSI[68], ADELPHE[52]. Bien que nous ne détaillerons pas ici l'ensemble de ces propositions, nous nous concentrerons sur trois méthodologies qui ont inspiré nos travaux : la proposition de Ferber[32], celle de Demazeau[25] puis celle de Jennings[88] et Wooldridge[88]. Il est important de remarquer que ces trois méthodologies ne font aucune supposition sur le(s) modèle(s) d'agent(s) utilisé(s) et se concentrent sur la décomposition en terme de rôles d'un système complexe.

La méthodologie Aalaadin Le modèle proposé par Ferber, appelé AALAADIN, repose sur les notions de rôle, de groupe et d'agent (figure 1.10). Un agent est une entité communicante qui joue un ou plusieurs rôles dans des groupes. Un groupe est un ensemble atomique d'agents. Chaque agent peut appartenir à différents groupes et les groupes peuvent se chevaucher. Enfin, un rôle dans AALAADIN est une représentation abstraite de la fonction, du service ou tout simplement l'identificateur d'un agent au sein d'un groupe. Chaque agent peut tenir plusieurs rôles, mais chaque rôle est local à un groupe. La communication entre les agents n'est possible qu'au travers des rôles qu'ils assument et par

conséquent, le contrôle sur les communications est effectué par le groupe.

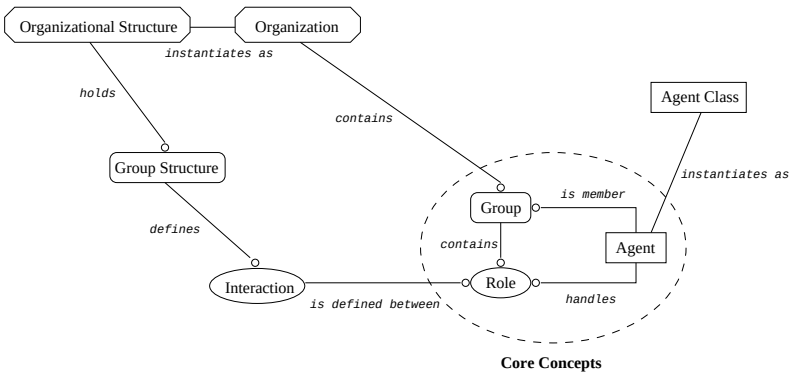

FIG. 1.10 – *Les diférentes concepts du modèle* AALAADIN

Ainsi, dans AALAADIN, une organisation est vue comme un cadre (c'est-à-dire un *framework*) pour les activités et les interactions grâce à la définition de groupes, de rôles et de leurs relations. La notion d'organisation correspond donc ici à une relation structurelle entre les rôles définis au sein des groupes. La notion d'interaction n'est pas réifiée (elle n'est pas explicitement représentée ni manipulable dans le système) et la méthode ne fournit pas de principes permettant de faciliter la décomposition d'un système complexe.

La méthodologie Voyelles La méthodologie VOYELLES développée au sein de l'équipe MAGMA du Leibniz repose sur quatre concepts, identifiés par les quatre voyelles : A pour Agents, E pour Environnements, I pour Interactions et O pour Organisations. Chaque voyelle correspond à un ensemble de modèles qui devront être combinés pour aboutir à la réalisation du système multi-agents[25] :

- Agents, qui concernent les modèles (ou les architectures) utilisés pour la partie active de l'agent, depuis un simple automate à un complexe système à base de connaissances.
- Environnements, qui sont les milieux dans lesquels sont plongés les agents. Ils sont généralement spatiaux dans la plupart des applications multi-agents.
- Interactions, qui concernent les infrastructures, les langages et les protocoles d'interactions entre agents, depuis de simples interactions physiques à des interactions langagières par actes de langage.
- Organisations, qui structurent les agents en groupes, hiérarchies, relations, etc.

De plus, la méthodologie VOYELLES est guidée par trois aspects[71] :

- déclaratif : un système est constitué d'agents, d'environnements, d'interactions et d'organisations,
- computationnel : les fonctionnalités d'un système multi-agents incluent les fonctionnalités individuelles des agents enrichies des fonctionnalités qui résultent de la valeur ajoutée par le système (parfois appelée *intelligence collective*).
- récursif : un système multi-agents peut être considéré à un niveau d'abstraction supérieur comme une entité multi-agents.

Le schéma global de conception et de développement consiste dans un premier temps à agentifier le problème grâce à la phase d'analyse, puis à l'aide de différents outils, la phase de conception permet de réaliser le système multi-agents (figure 1.11).

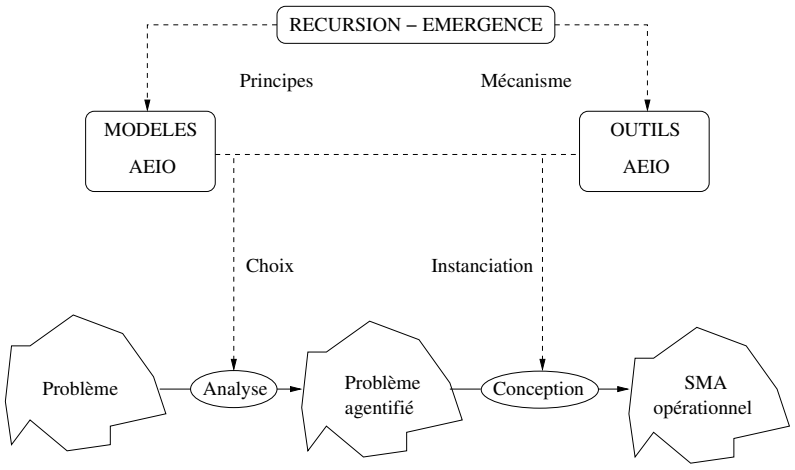

FIG. 1.11 – *Les différentes étapes de la méthodologie* VOYELLES*[65].*

C'est lors de la phase de conception que l'utilisateur spécifie le modèle d'agent, le modèle d'environnement, le modèle d'interaction et le modèle organisationnel, avant de pouvoir instancier le système.

La méthodologie Gaia La troisième méthodologie appelée GAIA, proposée par Jennings, Wooldridge et Kinny, est basée sur l'idée qu'un système multi-agents doit être vu comme une organisation "computationnelle" reposant sur les interactions entre les différents rôles présents dans le système. Le but de cette méthodologie est le même que celui d'AALAADIN : capturer la structure organisationnelle du système. Cependant, dans GAIA, cette structure est constituée d'un ensemble de rôles en relation, et qui interagissent selon des *motifs d'interaction*. Ainsi, cette méthode propose l'identification de deux modèles : le modèle de rôle et celui des interactions.

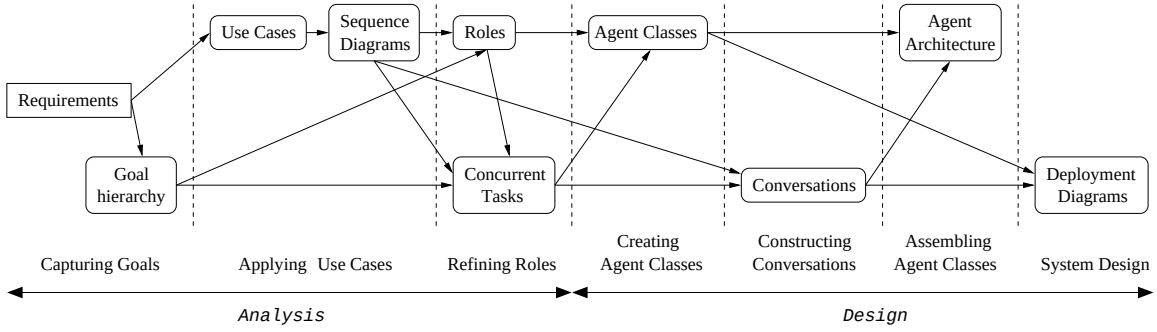

FIG. 1.12 – *Les différentes étapes de la méthodologie* GAIA

Un rôle dans GAIA est défini par quatre attributs :

– Les responsabilités : ce sont "les invariants" du rôle, qui se décomposent en deux

catégories : *liveness*, un comportement à déclencher selon certaines conditions et *safety*, qui consiste à s'assurer que certaines conditions restent valides.
- Les permissions : les droits accordés à ce rôle. Ces droits concernent principalement les droits sur l'accés à l'information.
- Les activités : qui correspondent au comportement "privé" de l'agent : c'est-à-dire des comportements pouvant être déclenchés sans que l'agent soit en interaction.
- Les protocoles : ils définissent la structure des interactions entre les rôles.

Nous nous intéressons principalement au dernier point : la représentation des protocoles. Dans GAIA, les protocoles sont des motifs d'interaction *institutionnalisés*, c'est-à-dire décrits formellement et de manière abstraite. La description d'un protocole dans GAIA est la suivante :

But	Description textuelle de la nature de l'interaction
Initiateur(s)	Le(s) rôle(s) responsable(s) de l'initiation de l'interaction.
Intervenant(s)	Le(s) rôle(s) impliqués dans l'interaction
Entrée(s)	Information(s) utilisée(s) par le rôle initiateur lors du déroulement du protocole
Sortie(s)	Information(s) fournie(s) ou produite(s) par le(s) rôle(s) non initiateur(s) lors de l'interaction
Effet(s) de bord	Description textuelle des traitements devant être effectués par l'initateur lors de l'interaction

Une interaction est représentée comme un protocole entre différents rôles, mais il n'y a pas de description précise des messages échangés ou encore de leur flux. La méthodologie proposée est intéressante sur bien des points, mais reste trop générale pour pouvoir facilement passer de la phase de conception du système à sa réalisation[67].

La programmation orientée interactions

Nous avons vu dans la section 1.2.1.0, que des langages de communication entre agents existaient. Cependant, ces langages ne permettent pas d'exprimer le déroulement des conversations. En effet, les travaux sur les langages de communication entre agents [34] identifient les différents niveaux constituants une conversation :

Sémantique	Description du contenu du message
Intention	Description de l'intention grâce aux performatifs
Interaction	Description de la coordination, la structure de la conversation

Pour les deux premiers niveaux, la sémantique et l'intention d'un message, on dispose d'un certain nombre de langages permettant de les exprimer, tel SL ou KIF pour les langages de contenu et KQML et FIPA-ACL pour les performatifs et la structure du message. Cependant, même en considérant des plateformes hétérogènes partageant la même ontologie, il reste difficile d'avoir des garanties sur le respect des protocoles d'interaction[25]. C'est pourquoi des travaux ont été menés sur la formalisation de cet aspect interactionnel avec plusieurs objectifs : décrire l'enchaînement des messages, avoir certaines garanties sur le déroulement d'une conversation et faciliter l'interopérabilité entre plateformes hétérogènes.

25. Nous faisons ici référence aux différents protocoles définis dans les spécifications de la FIPA.

L'interaction comme loi sociale Lors du développement de systèmes distribués, le concepteur doit se représenter les différentes interactions qui interviennent entre les multiples entités de son système. Ces interactions ne sont pas uniquement le reflet des communications entre les agents, mais aussi l'expression de dépendances fonctionnelles entre ces derniers[75]. Ainsi, une interaction est généralement constituée d'un ensemble d'entités, de flux de messages et d'effets de bords se produisant au sein des différentes entités.

La programmation orientée interactions, comme son nom l'indique, prend pour principal objet de son étude, la spécification des interactions. Un des principaux objectifs est de réifier cette notion pour qu'elle puisse être manipulée par le système, et aussi pour simplifier le développement des protocoles d'interaction. En effet, bien souvent, seul le concepteur conserve une vue globale des interactions se produisant dans son système, tandis que cette information concernant l'interaction se retrouve *dispersée* ou *éclatée* au sein des différentes entités.

D'autre part, la programmation orientée interactions se veut une approche pragmatique, qui tout en diminuant l'autonomie des agents impliqués (car ils doivent suivre un déroulement précis dans leurs dialogues), apporte des garanties sur le déroulement des interactions. Il est d'ailleurs intéressant de rapprocher la notion de protocole d'interaction et celle de loi sociale. Dans l'un de ses articles, Shoham[74] propose de définir ce qu'il nomme des lois sociales lors de la conception d'un système pour garantir la coordination des entités la constituant. Ces travaux ont trouvé un écho important dans ceux de Sierra[30], dans lesquels il a développé la notion d'institution et de norme.

Quelques langages orientés protocoles d'interaction Pour illustrer ces approches basées sur une formalisation des interactions, nous en avons étudié quatre : April[59], AgenTalk[48], Cool[6] et AgentUML[66]. April, *Agent PRocess Interaction Langage*, est issu des travaux sur les langages d'acteurs[44]. C'est un langage symbolique de manipulation de processus facilitant la création de protocoles. Dans April, le développeur doit concevoir un ensemble de *handlers* qui traitent chacun des messages spécifiques (la mise en relation est effectuée par *pattern matching*). Suivant les même principes, AgenTalk rajoute la possibilité de créer facilement de nouveaux protocoles par spécialisation de protocoles existants, en se reposant sur un mécanisme d'héritage. Un autre apport fondamental d'AgentTalk est la description explicite du protocole : la conversation est représentée par un ensemble d'états et un ensemble de règles de transitions. Ce même principe a été employé dans Cool, qui propose de modéliser une conversation à l'aide d'un automate à états finis. Finalement, AgentUML utilise une approche originale, consistant à capitaliser les acquis du langage Uml pour représenter les protocoles d'interactions. Cette démarche à l'avantage d'utiliser un langage maintenant largement diffusé dans la communauté objets, et pour lequel de nombreux outils sont disponibles. Cependant, AgentUML reste une notation pour l'instant, et ne fournit pas de support permettant de passer de la spécification à l'implémentation.

Dans Cool, la nécessité de l'introduction de *conventions* entre agents pour faciliter la coordination est mise en avant. Cette notion de *convention* doit être rapprochée des travaux de Shoham sur les *règles sociales*[74] et de leurs apports sur la performance globale du système. Ainsi, l'introduction de ce niveau de gestion des interactions tout

en rigidifiant d'une certaine manière les interactions possibles entre agents, apporte des garanties sur la coordination et permet de réifier ces interactions. Le tableau ci-dessous, inspiré par les travaux de Singh[76], illustre les différents niveaux d'abstractions d'un systèmes multi-agents :

Compétences applicatives	Connaissances métier
Modèles d'agents et compétences systèmes	Conception orientée agents
Gestionnaire de conversation	Conception orientée interactions
Transport des messages	Plateforme ou conteneur d'agents

Pour conclure, nous reprenons ici la définition proposée par Singh[76] de l'approche orientée interactions, qui caractérise nos objectifs :

> "*We introduce interaction-oriented programming (IOP) as an approach to orchestrate the interactions among agents. IOP is more tractable and practical than general agent programming, especially in settings such as open information environments, where the internal details of autonomously developed agents are not available.*"

1.3 Les paradigmes des systèmes multi-agents

Dans cette section, nous allons présenter les différents paradigmes caractérisant le domaine des systèmes multi-agents. Les deux premiers, la notion d'agent et d'organisation, sont incontournables et constituent la base des ces systèmes, tandis que le dernier point concerne principalement leur réalisation avec la notion de plateforme.

1.3.1 Les modèles d'agents

Nous avons vu dans la section 1.1 que différentes métaphores ont été proposé pour faciliter le développement de programmes contrôlant une machine. Ces métaphores reposent sur un modèle abstrait définissant de manière plus ou moins formelle les règles d'analyse et d'écriture de programme. Les différents langages de programmation sont l'incarnation de ces modèles abstraits et proposent à l'utilisateur (le développeur) un ensemble de structures permettant d'exprimer des traitements d'information. Des notions fondamentales telle que la notion de variable ou de fonction, ou encore la notion de structure de contrôle ou de donnée se retrouvent dans la plupart des modèles de calcul proposés. Ces notions constituent l'algorithmique mais ne suffisent pas à la décomposition d'un programme. En effet, des propriétés non fonctionnelles telle que la sécurité, la répartition physique de l'application ou la fiabilité peuvent aussi intervenir.

Les modèles de calcul reposent ainsi généralement sur un même noyau algorithmique (variables, fonctions ou procédures, arithmétique, conditionnelles et la possibilité de répétition d'un calcul), mais proposent des structures propres qui définissent une représentation singulière du monde ainsi que les manipulations possibles de cette représentation. Ainsi, en programmation orientée objets, le monde est considéré comme constitué d'un ensemble d'entités homogènes organisées de manière hiérarchique, caractérisées par les fonctionnalités qu'elles proposent (i.e. les messages qu'elles savent traiter), et échangeant des informations fortement spécifiées (c'est-à-dire fortement typées).

La programmation orientée agents considère le monde comme un ensemble d'agents interagissant au travers de conversations (c'est-à-dire des interactions pouvant faire intervenir plus de deux participants), et cherchant à résoudre un problème insoluble pour chacun d'entre eux, mais accessible à la société qu'ils forment. Un modèle d'agent défini ainsi des structures et/ou des fonctionnalités facilitant la décomposition d'un problème et sa résolution en suivant les principes de la programmation orientée agents.

Si l'on poursuit l'analogie, les langages orientés objets définissent les notions de classes (ie. ensemble de descriptions) et d'instances (des objets correspondants à ces descriptions), et favorisent l'encapsulation des données. Cette approche correspond à la phase industrielle des technologies objets dans laquelle les langages de classes (SmallTalk, C++ et Java par exemple) se sont imposés par rapport à d'autres propositions tels que les langages à prototypes (Self) ou même les langages d'acteurs.

Dans la suite de cette section, nous allons présenter les modèles d'agents existants. Contrairement aux technologies orientées objets, pour l'instant aucun modèle fédérateur ne s'est réellement imposé, et les structures et concepts mis en avant restent très diversifiés. Le panorama qui suit repose principalement sur deux articles de synthèse : celui de Nwana nwana95software et celui de Jennings et Wooldridge [89].

De la diversité des modèles existants

De nombreux modèles d'agents ont été proposés, que cela soit sous la forme de modèles théoriques ou de modèles pragmatiques. Nous introduisons cette distinction pour souligner le fait que certains modèles sont implémentés et disponibles, tandis que d'autres sont décrits formellement, mais il n'est cependant pas possible de les utiliser pour construire des applications. Cette multiplicité est une richesse qui souligne la diversité des modèles cognitifs existant pour modéliser un agent. Cependant, c'est aussi une faiblesse car il n'y a toujours pas de langage ou d'outil permettant d'exprimer ces différents modèles au sein d'un même formalisme. Ainsi, une étude comparative de modèle d'agent dans une application ne peut être effectuée simplement car bien souvent un modèle d'agent donné n'est utilisable qu'au sein d'une plateforme qui lui est dédiée. Ceci présente plusieurs inconvénients : non seulement il est impossible d'établir une taxonomie des différents modèles existants, mais il est aussi impossible de donner des critères facilitant le choix d'un modèle d'agent en fonction du domaine d'application.

Il y a donc beaucoup de modèles d'agents mais pas de taxonomie[37], ni de convergence sur la définition de ce qu'est un agent. De plus, comme nous l'avons vu précédemment, les systèmes multi-agents ont de nombreux domaines d'application ce qui entraîne des attentes parfois très différentes sur les fonctions nécessaires d'un agent. Sans vouloir établir une liste exhaustive, les critères suivants apparaissent souvent dans la littérature : autonomie, proactivité, sociabilité, négociation, coopération, mobilité, conversation de haut niveau ... Il faut remarquer qu'il n'existe pas de modèle d'agent répondant à l'ensemble de ces caractéristiques. En fonction de l'application visée, seul un sous ensemble de ces caractéristiques est nécessaire, il faut alors s'orienter vers le modèle d'agent qui semble le plus adapté.

Dans la suite de ce document, nous nous intéresserons particulièrement aux agents logiciels non situés. Nous étudierons différentes classes d'agents : les agents utilisant les outils logiques, les agents réactifs, les agents reposant sur une modélisation des croyances, des désirs et des intentions, et les agents à couches. Cette présentation reprend la structure d'un article de Mike Wooldridge publié dans le livre "*Multiagent systems, A Modern Approach to Distributed Artificial Intelligence*" édité par Gerhard Weiss.

Les agents *logiques* L'approche traditionnelle en Intelligence Artificielle consiste à utiliser une représentation symbolique de l'environnement, et à y associer des opérations syntaxiques permettant de le modifier. Les agents logiques utilisent cet outil, la représentation symbolique étant un ensemble de formules logiques, et les opérations syntaxiques, des déductions logiques ou des démonstrations automatiques de preuve. Dans ces modèles, les connaissances de l'agent sont des formules logiques, et le processus de décision de l'agent se réduit à une démonstration de preuve.

Ces modèles ont l'avantage de posséder une sémantique non ambiguë, et une forte expressivité (comme dans les langages déclaratifs type Prolog). Cependant, la complexité algorithmique de la démonstration automatique de preuve rend l'utilisation de ces modèles impossible lorsqu'il y a des contraintes de temps, et surtout, ces modèles reposent sur une hypothèse de monotonie, c'est-à-dire que les connaissances ne peuvent être remises en cause lorsque l'agent est en train de prendre sa décision. Ces deux désavantages rendent

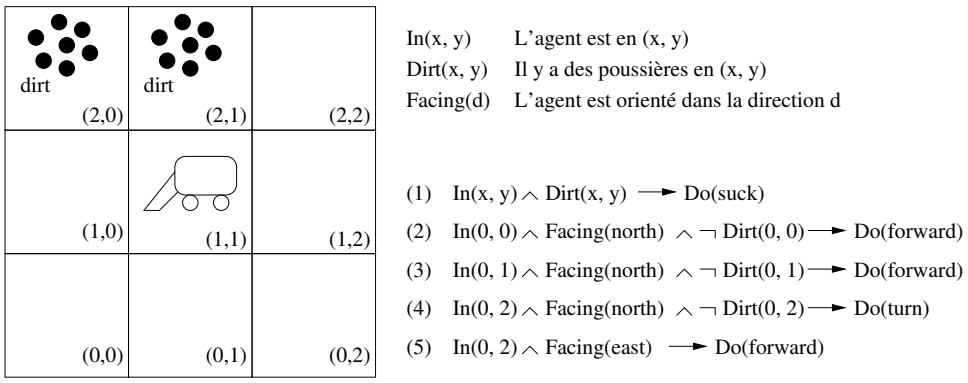

FIG. 1.13 – *L'exemple classique :* vacuum world

ces systèmes difficilement utilisables "*dans la vraie vie*".

Les agents *purement réactifs* A la fin des années 80, en réaction à l'approche basée sur les outils logiques, des travaux se développent en se utilisant des hypothèses radicalement différentes. Ces travaux rejettent la représentation basée sur la logique du monde et des actions, ainsi que l'affirmation que le comportement rationnel est le produit de l'interaction entre l'agent et son environnement. Au lieu de cela, ces modèles d'agent reposent sur l'idée que les comportements *intelligents* émergent des interactions se produisant entre des comportements plus simples.

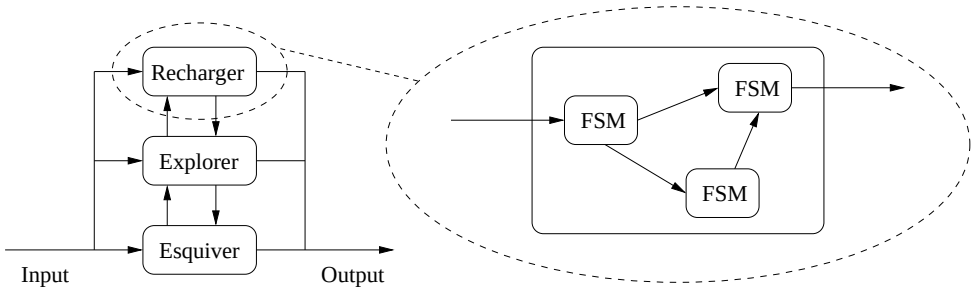

FIG. 1.14 – *L'architecture de subsumption de Brooks*

Cette approche est souvent caractérisée par les trois critères suivants :

- *approche comportementale* : consistant à développer et combiner un ensemble de comportements,
- *agents situés* : c'est-à-dire percevant et réagissant à son environnement,
- *agents réactifs* : car leur comportement semble n'être qu'une réaction à l'environnement.

La figure 1.14 illustre le fonctionnement d'un des modèles les plus connus de cette approche, l'architecture de subsumption de Brooks[14]. Dans ce modèle, un agent est constitué d'un ensemble de modules identifiant une situation donnée et produisant une action. Plusieurs modules pouvant se déclencher simultanément, il est nécessaire de mettre en place un mécanisme de sélection pour *choisir* l'action à effectuer. Brooks proposa une

hiérarchisation des modules, dans laquelle les modules les plus bas peuvent inhiber les modules situés au dessus d'eux. L'idée sous-jacente est que plus les modules sont haut dans la hiérarchie, plus ils représentent des comportement abstraits et/ou complexes, et plus faible est leur priorité. Ainsi, la figure 1.14 illustre le comportement d'un robot mobile, dont le premier objectif est d'éviter les collisions, puis d'explorer son environnement et finalement de recharger ses batteries.

Ce type d'architecture purement réactive présente certains avantages comme sa simplicité, sa faible complexité algorithmique, sa tolérance aux pannes ou encore son élégance. Cependant, il est certains points difficiles à aborder avec ce type d'agent : la prise en compte d'informations non locales à l'agent, la mise en place de mécanismes d'apprentissage, et surtout la conception de l'émergence du comportement global en fonction des comportements locaux.

Les agents *Belief-Desire-Intention* Les agents Bdi, pour *Beliefs Desires and Intentions*, sont inspirés des travaux sur le raisonnement pragmatique, c'est-à-dire le processus de décision permettant de sélectionner les actions à effectuer pour atteindre ses objectifs. Dans ces modèles, le processus se décompose en deux phases : se fixer certains buts et définir la manière de les atteindre. La première phase est appelée phase de délibération, tandis que la seconde correspond à une phase de planification.

La figure 1.15 illustre le fonctionnement d'un agent Bdi. L'agent est caractérisé par :

- ses *croyances*, les connaissances qu'il a du monde,
- ses *désirs*, les objectifs accessibles en fonction de ses *croyances* et de ses *intentions*,
- ses *intentions*, qui caractérisent les objectifs courants ou en cours de réalisation de l'agent.

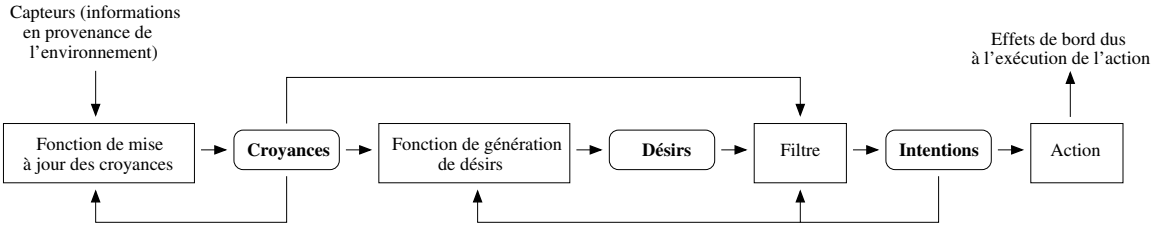

Fig. 1.15 – *Une architecture schématique d'agent* Bdi *(extrait de Gerhard Weiss, p58)*

Dans ce modèle, les *intentions* jouent un rôle central : elles orientent la planification, elles limitent les délibérations futures, elles sont persistantes et elles influencent les croyances futures sur lesquelles se base le raisonnement. L'équilibre entre ces différents aspects représente l'une des clés de la conception d'agents Bdi. Particulièrement, le compromis entre la poursuite d'une *intention* ou son abandon est une fonction particulièrement difficile à concevoir. D'autre part, les expérimentations menées par Kinny et GeorgeffGeorgeff[83] ont montré que cette fonction est fortement dépendante de la dynamicité de l'environnement de l'agent.

Le modèle Bdi est intéressant non seulement car il est intuitif, mais aussi car il fournit une bonne décomposition fonctionnelle, qui identifie clairement les sous-systèmes

nécessaires à la conception d'un agent. Néanmoins, la principale difficulté reste toujours l'implémentation efficace de ces sous-systèmes.

Les agents à *couches* Les architectures à couches consistent à décomposer le comportement d'un agent en deux parties : la partie réactive et la partie proactive. Il y a donc au minimum deux couches, mais il est possible d'avoir une hiérarchie complexe de couches qui interagissent entre elles. La figure 1.16 illustre les deux grandes classes d'architecture à couches, qui se distinguent principalement par la diffusion de l'information au sein des couches et la gestion du flux de contrôle entre les couches. La première classe est caractérisée par des couches horizontales, qui reçoivent toutes la même information et qui proposent des actions en réaction à ces informations. Dans la deuxième classe, les couches sont organisées verticalement et une couche au plus reçoit les informations et/ou produit les actions.

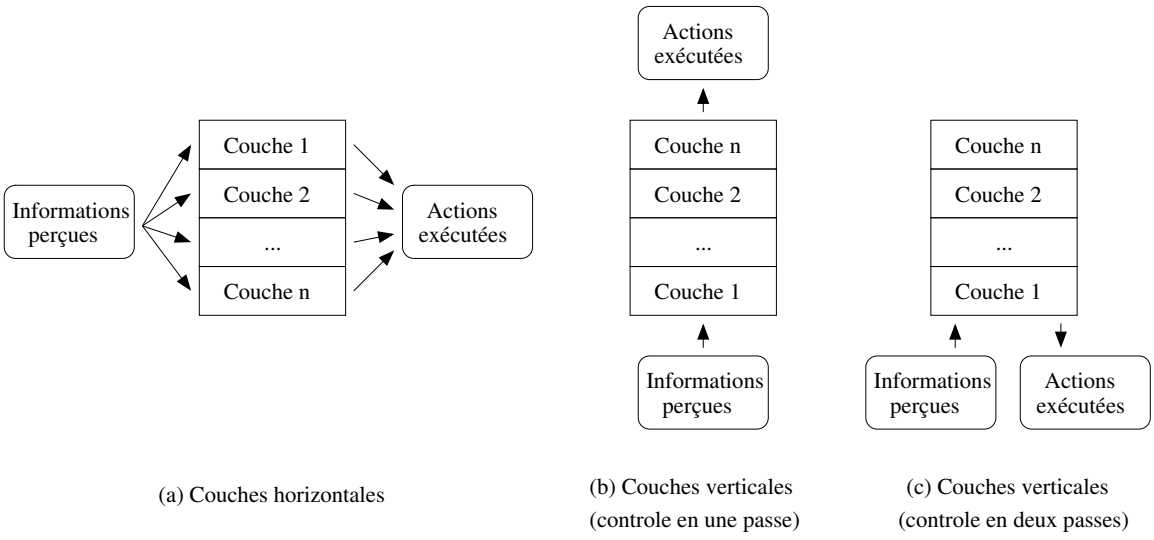

FIG. 1.16 – *Architectures et flux de contrôle dans les agents à couches (extrait de [90])*

L'avantage principal des couches organisées horizontalement est qu'il suffit d'ajouter des couches à l'agent pour qu'il exhibe un nouveau comportement. Mais comme toutes les couches proposent des actions, il est indispensable d'introduire un médiateur qui définira à chaque moment quelle est la couche prioritaire. C'est ce point de centralisation qui est particulièrement gênant dans cette architecture : pour que le comportement soit cohérent il faut que le concepteur appréhende l'ensemble des interactions entre les couches (ce qui est rapidement irréalisable lorsque le nombre de couches augmente).

Ce problème est justement partiellement résolu avec les architectures à couches verticales. Dans ces modèles, l'information traverse les couches séquentiellement, réduisant ainsi drastiquement les interactions entre les différentes couches. En fonction du mécanisme de contrôle, l'information ne traverse qu'une fois chacune des couches (figure 1.16 (b)), ou deux fois, lors de la remontée des informations et de la descente des actions (figure 1.16 (c)).

Les architectures à couches sont actuellement les plus utilisées. Les couches représentent

naturellement les différentes fonctionnalités de l'agent : réactivité, proactivité et comportement social. Le principal avantage de cette approche est aussi son principal inconvénient : c'est une approche pragmatique, c'est-à-dire facilement implémentable et praticable, qui manque de clarté conceptuelle et sémantique. D'autre part, la gestion des interactions entre les différentes couches, bien que simplifiée dans le cas des architectures à couches verticales, reste difficile à maîtriser.

1.3.2 Les modèles organisationnels

Les différentes interprétation de la notion d'organisation

A partir du moment où des d'entités sont regroupées et ont à co-exister, des structurations émergent et des organisations apparaissent. Que cela soit dans le cadre de sociétés humaines ou logicielles, le besoin d'organisation est nécessaire lorsque des objectifs communs à l'ensemble des entités doivent être atteints. Dans le cadre de la sociologie, le domaine de la théorie des organisations se concentrent sur cette problématique et étudie le comportement et les caractéristiques des organisations humaines. Même si il est difficile d'établir le lien et surtout l'utilisation qu'il serait possible de faire de ces théories dans le cadre informatique, certains aspects sont très intéressants.

Les origines de la théorie des organisations[26] proviennent du besoin de rationalisation qu'ont éprouvés certains entrepreneurs au début du XXème siècle. Son initiateur le plus médiatique est Taylor, dont l'approche a parfaitement été illustrée dans le célèbre film de Chaplin. Taylor s'est principalement concentré sur le travail des ouvriers et sur leur environnement. Sa préoccupation était d'optimiser au maximum la productivité d'un ouvrier et par extension d'une chaîne de production. La contribution principale de Taylor à la théorie des organisations est qu'il a identifié une nouvelle fonction de l'entreprise dédiée à l'étude et l'optimisation des méthodes de travail en vue de gains de productivité.

Un second contributeur important fut Henri Fayol, un ingénieur français qui a prêché le développement d'une *science de l'administration*, en publiant à la fin de sa carrière un ensemble de principes dans un ouvrage intitulé "Administration Industrielle et Générale" (1916). Fayol identifie dans son livre, cinq fonctions fondamentales du *management* :

- Prévoir et planifier : examiner les opportunités et établir des plans d'action,
- Organiser : construire l'infrastructure matérielle et humaine,
- Diriger : maintenir l'activité des personnels,
- Coordonner : rassembler, unifier et harmoniser les activités,
- Contrôler : vérifier que tout se déroule en conformité avec les règles et la pratique.

Fayol identifia aussi le rôle fondamental de la notion d'autorité, et plus précisément, de son inconcevabilité sans responsabilités. Ce point est important, car Fayol insiste sur la nécessité d'introduire des mécanismes de récompense ou de pénalité, et ces principes se retrouvent dans de nombreux travaux actuels, notamment ceux de Dignum et ses collègues[20].

26. Il n'y a volontairement pas de référence dans cette sous-section car l'ensemble des informations proviennent d'une seule source : l'Encyclopaedia Universalis.

D'autre travaux ont ensuite exploré l'importance du facteur humain au sein des organisations, mais ces travaux sont assez éloignés de nos problématiques. Nous préférons donc détailler des développements plus récents de la théorie des organisations, notamment avec les travaux de Friedberg et Crozier. La théorie actionniste des organisations considère que tout système social peut être compris à partir de l'action des différents agents qui le composent. C'est Chester Bannard qui est le premier à avoir formalisé cette idée et qui a établi cette intéressante distinction entre systèmes coopératifs et organisations :

> "*dans celui-là* (le système coopératif)*, la fin de l'action collective est fixée par les différents acteurs, et ne dépend que d'eux, tandis que, dans celle-ci* (organisation)*, elle est pré-établie à l'avance et en dehors des agents dont la seule tâche est de la réaliser. Toute organisation est un système coopératif, mais l'inverse n'est pas vrai : tout système coopératif n'est pas une organisation.*"

Cette définition est très intéressante sur plusieurs aspects : d'abord elle établit la distinction entre les systèmes coopératifs dans laquelle il y a émergence de l'organisation de ceux où elle est pré-établie, mais aussi elle insiste sur le fait que tout système coopératif ne peut être considéré comme une organisation. Cela sous-entend que l'organisation est en elle même une entité à part entière qui dépasse le cadre de la coopération entre les entités constitutives du système. Nous verrons dans la suite de ce document que nous nous inscrivons dans cette logique avec notre méthodologie de conception de systèmes multi-agents.

Les modèles organisationnels de systèmes multi-agents

Nous avons vu certains aspects de la notion d'organisation telle qu'elle est perçue en sociologie dans le cadre de la théorie des organisations. Nous allons maintenant présenter quelques travaux effectués dans le domaine des systèmes multi-agents autour de cette même notion. Nous étudierons dans un premier temps les différentes formes topologiques des organisations, puis nous identifierons quelques rôles dévolus à l'organisation.

Les différentes topologies Un premier critère caractérisant une organisation est sa structure topologique, c'est-à-dire la nature des liaisons entre les entités. La figure 1.17 illustre cet aspect. Parmi les modèles les plus répandus, il y a le modèle hiérarchique (utilisé dans MAGIQUE[9]), le modèle orienté groupes (élément fondamental d'AALAADIN[31]), le modèle holonique (une variante du modèle hiérarchique, dans lequel les fils d'un nœud sont totalement connectés[17]), le modèle peer-to-peer (à la base de la majorité des systèmes de partage de fichiers comme GNUTELLA)[1] et le modèle de diffusion (*broadcast*).

Il est important aussi d'indiquer la sémantique des liaisons entre les entités. Effectivement, les liaisons peuvent représenter un lien de communication entre deux entités, ou par exemple un lien de contrôle. Il n'est ainsi pas aberrant d'imaginer une topologie "physique" de type peer-to-peer pour représenter les liens de communication entre plateformes, au dessus de laquelle est instanciée une topologie "logique" utilisant une autre structure, dépendante de l'application.

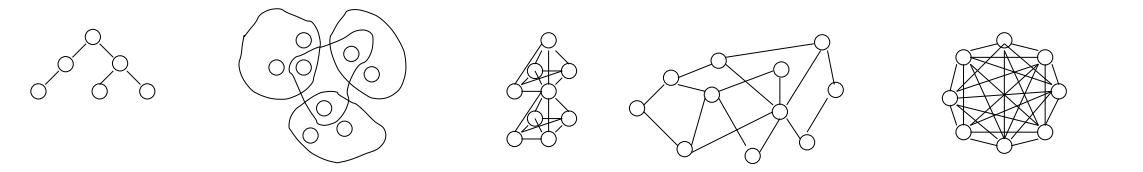

FIG. 1.17 – *Les différentes topologies de communication*

Ce dernier point, à propos des liaisons de communication au niveau physique, est crucial car il met en évidence la dépendance qui existe entre la topologie du graphe des entités et les mécanismes de contrôle associés à cette topologie. Ces mécanismes peuvent recouvrir la gestion d'abonnement et désabonnement des entités, la gestion de la recherche d'accointance ou de services, la gestion du routage des messages, l'optimisation du flux d'information au sein de l'organisation.

Quelques fonctionnalités fondamentales des organisations Indépendamment de la topologie de l'organisation, un certain nombre de fonctionnalités peuvent lui être ajoutées. Pour cela, certains rôles génériques ont été identifiés : les *facilitators*, les *brokers* et les *mediators*[72]. Un *facilitator* est responsable de la communication entre un groupe d'agents locaux et des agents distants. Cette tâche est effectuée grâce à deux services : le routage des messages sortants vers les agents distants et la transformation des messages entrants pour les agents locaux.

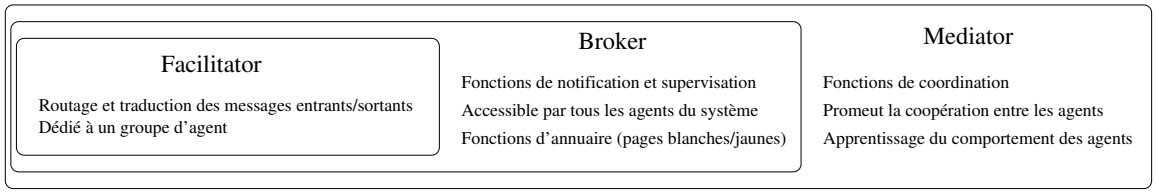

FIG. 1.18 – *Les fonctions des* facilitator*,* broker *et* mediator

Un *broker* est un *facilitator* possédant en plus des fonctionnalités de notification et de supervision. D'autre part, le *facilitator* est dédié à un groupe particulier d'agents, tandis que le *broker* peut être contacté par n'importe quel agent du système pour trouver un service ou un agent particulier. Enfin, le *mediator* reprend les fonctions du *broker*, mais assume en plus un rôle de coordination en promouvant la coopération entre agents et en "apprenant" par observation le comportement des agents. La figure 1.18 illustre l'agrégats de fonctionnalités du *facilitator* au *mediator*.

Un autre rôle fondamental, identifié particulièrement dans AALAADIN, est celui gérant les droits d'accès à l'organisation (au groupe dans le cas d'AALAADIN) : un agent de l'organisation doit constituer le point d'entrée pour les agents désirant intégrer l'organisation. Pour cela différentes politiques peuvent être instaurées, mais cet agent devra gérer les abonnements/désabonnements d'agents

1.3.3 Sur la notion de plateforme

La notion de plateforme est liée à l'implémentation des systèmes multi-agents : elle constitue un réceptacle au sein duquel les agents peuvent évoluer. Les plateformes sont un environnement permettant de gérer le cycle de vie des agents et dans lequel les agents ont accès à certains services. Nous allons dans un premier temps nous intéresser à la proposition de la Fipa avant de donner un aperçu de quelques plateformes existantes

L'initiative de la Fipa

La mission que s'est fixée la Fipa, *Foundation for Intelligent Physical Agents*, est de faciliter l'interopérabilité des agents et des systèmes multi-agents provenant de différents fournisseurs. Ainsi, la Fipa a produit depuis 1997 un ensemble de spécifications qui s'étendent des langages de communications (*Agent Communication Languages*) aux langages de contenu (*Content Language*) ainsi qu'aux protocoles d'interaction. Le leitmotiv de la Fipa est que l'utilisation des actes de langage, de la logique des prédicats et de la définition d'ontologies permet de garantir une interprétation non ambigue, respectant la sémantique des messages échangés.

Cet objectif particulièrement ambitieux est loin d'être atteint. Cependant, les premières expérimentations d'interopérabilité ont eu lieu et sont prometteuses. Même si dans cette première phase, les tests se concentrent principalement sur les couches de communications et l'interprétation de messages de gestion des agents (abonnement et recherche d'agent au sein d'une plateforme).

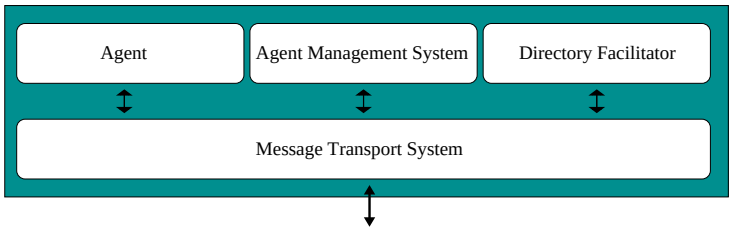

Fig. 1.19 – *L'architecture de référence de la plateforme* Fipa

La figure 1.19 illustre l'approche de la Fipa en ce qui concerne la plateforme, qui se compose de trois composants principaux : la couche de communication (*Message Transport System*), la gestion des agents (*Agent Management System*) et la gestion de l'annuaire (*Directory Facilitator*). Ce sont ces composants qui sont évalués lors des tests d'interopérabilité. Le protocole utilisé est Iiop et la structure des messages et clairement définie. La figure 1.20 détaille la structure d'un message Fipa ainsi que l'envoi d'un message entre deux agents situés sur des plateformes distinctes. Le message est construit dans un premier temps par l'agent A, avec le corps du message représentant la sémantique de l'échange et l'enveloppe qui regroupe les informations de transport (encodage, protocole, ...). L'agent A délègue ensuite l'expédition du message au Mts qui en fonction du protocole utilisé (Iiop, Http, Rmi, ...) sélectionne un *Message Transport Provider* et effectue la communication avec la plateforme hébergeant l'agent B.

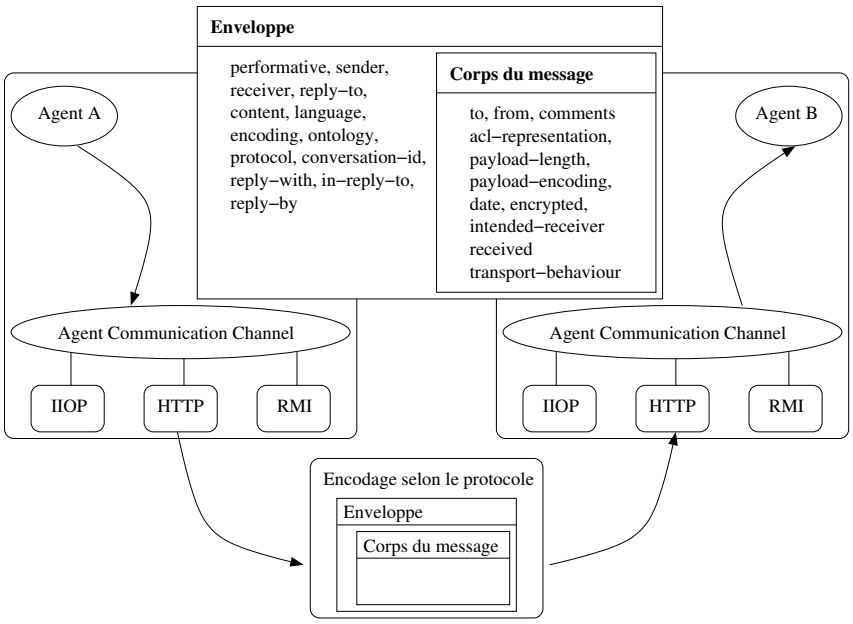

FIG. 1.20 – *Envoi de message entre deux agents* FIPA

La notion de plateforme est utilisée comme une infrastructure de communication et de gestion des agents. Ces fonctionnalités sont apportées au travers de services de plateformes accessibles soit aux agents de la plateforme, soit aux autres plateformes.

Panorama de quelques plateformes existantes

Il existe un grand nombre de systèmes multi-agents dédiés à différent modèles d'agent. Nous ne prétendons pas donner ici une liste exhaustive de l'ensemble des plateformes disponibles actuellement [27]. Nous avons en effet choisi de sélectionner les plateformes les plus significatives pour le développement d'agents logiciels, et d'en donner une description succincte.

Les trois premières (GRASSHOPPER, VOYAGER et AGLET) sont dédiées aux agents mobiles. Les deux suivantes (AGENTBUILDER et JACK) sont particulièrement adaptées à l'implémentation d'agents de type BDI. ZEUS et JADE sont des plateformes conformes aux spécifications de la FIPA. Les trois dernières plateformes illustrent deux points spécifiques des systèmes multi-agents : l'importance de l'organisation du système avec MADKIT et VOLCANO, et la simulation avec SWARM qui est l'environnement de prédilection pour des expérimentations en vie artificielle. Dans la suite de ce document, nous nous concentrerons sur les systèmes multi-agents à *gros grains*, c'est-à-dire des agents logiciels relativement complexes et des sociétés ne dépassant une centaine d'agents.

Grasshoper de Ikv++ [28] :

La plateforme GRASSHOPER est une sur-couche au dessus de la JVM fournissant la

27. Pour une liste exhaustive, le lecteur pourra se reporter à : `http://www.agentbuilder.com`
28. GRASSHOPER-2 *Agent Development Platform* : `http://www.grasshopper.de`

gestion de la migration d'agent et la continuité des communications lors de ces migrations. Les services fournies par la plateforme sont la gestion du cycle de vie des agents, la gestion d'annuaires et la migration. GRASSHOPER est l'une des rares plateformes à implémenter la spécification MASIF de l'OMG concernant la migration d'agent.

Voyager de Reticular Systems [29] :

L'environnement VOYAGER peut être considéré comme un ORB, *Object Request Broker*, orienté agents mobiles. En plus de la gestion de l'invocation à distance des méthodes d'un agent mobile, VOYAGER propose des mécanismes de persistance des messages, de gestion de la sécurité ou encore la gestion des communications entre agents utilisant différents protocoles (RMI et IIOP par exemple).

Aglet d'Ibm [30] :

L'environnement initialement proposé par IBM définit un cadre de développement d'agent mobiles. Dans cette optique, la plateforme prend en charge les services nécessaires à la migration d'agent et à la gestion des communications. Une *aglet* est une classe JAVA étendant une classe fournie par le noyau et redéfinissant les fonctions à activer lors des sérialisations/désérialisations des agents.

AgentBuilder de Reticular Systems [31] :

La plateforme AGENTBUILDER est une implémentation proche du langage AGENT-0 proposé par . Les agents sont décrits avec le langage RADL, *Reticular Agent Definition Language*, qui permet de définir les règles du comportement de l'agent. Les règles se déclenchent en fonction de certaines conditions et sont associées à des actions. Les conditions portent sur les messages reçus par l'agent tandis que les actions correspondent à l'invocation de méthodes JAVA. Il est aussi possible de décrire des protocoles définissant les messages acceptés et émis par l'agent. AGENTBUILDER est probablement la plateforme commerciale la plus aboutie (et la plus rentable).

Jack Intelligent Agents d'Agent Oriented Software [32] :

JACK INTELLIGENT AGENTS est destiné au développement d'agent de type BDI, tout en fournissant une API orientée objets. Pour cela, un langage proche de la programmation logique est proposé au développeur, qui peut ensuite le compiler pour le transformer en classes JAVA. Les classes générées étendent des classes fournies par l'API du noyau de JACK : la classe `Agent`, la classe `Event` ou encore la classe `Plan`. Le noyau de JACK gère aussi la concurrence des tâches entre les agents, la réaction aux événements et l'infrastructure de communication.

29. VOYAGER : `http://www.reticularsystems.coms/voyager`
30. AGLET, `http://www.aglets.org` ou `http://www.trl.ibm.com/aglets`
31. AGENTBUILDER : `http://www.agentbuilder.com/`
32. JACK INTELLIGENT AGENTS : `http://www.agent-software.com.au/`

Zeus de British Telecom [33] :

ZEUS fournit un environnement de développement d'agent grâce à un ensemble de bibliothèques JAVA que les développeurs peuvent réutiliser pour créer leurs agents. ZEUS propose aussi un ensemble d'agents utilitaires (serveur de nommage et facilitateur) pour faciliter la recherche d'agents. Un agent dans ZEUS est constitué en trois couches : la couche de définition, qui contient les capacités de raisonnement et des algorithmes d'apprentissage, la couche organisationnelle, qui contient la base de connaissances des accointances de l'agent, et la couche de coordination, qui définit les interactions avec les autres agents. L'architecture de ZEUS se prête bien aux applications de planification ou d'ordonnancement, ce qui la rend particulièrement intéressante pour la réalisation de simulation ou d'applications de supervision. Cependant, chaque agent fonctionne grâce à une machine virtuelle JAVA (JVM, *Java Virtual Machine*), ce qui devient rapidement difficile à gérer lorsque le nombre d'agents dans le système augmente.

Jade du Tilab [34] :

JADE, pour *Java Agent DEvelopment framework*, est un *middleware* écrit en JAVA et se conformant aux spécifications de la FIPA. Cet environnement simplifie le développement d'agent en fournissant les services de base définis par la FIPA, ainsi qu'un ensemble d'outils pour le déverminage et le déploiement.

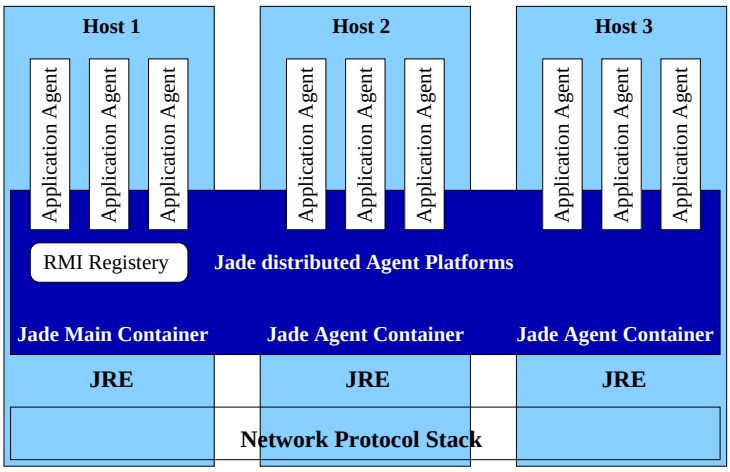

FIG. 1.21 – *Une plateforme* JADE *distribuée sur plusieurs conteneurs*

Une plateforme JADE peut être répartie sur un ensemble de machines (figure 1.21) et configurée à distance. La configuration du système peut être altérée dynamiquement, grâce à un mécanisme de migration d'agent au sein de la même plateforme.

MadKit du Lirmm [35] :

33. ZEUS AGENT BUILDING TOOLKIT : `http://www.labs.bt.com/projects/agents/zeus/index.htm`
34. JADE : `http://sharon.cselt.it/projects/jade/`
35. MADKIT : `http://www.madkit.org/`

La plateforme MadKit développée à l'Université de Montpellier II, est basée sur la notion d'agent, de groupe et de rôle. MadKit fournit une Api permettant la construction d'agent en spécialisant une classe d'agent abstraite. Chaque agent peut tenir différents rôles au sein de différents groupes. Les agents sont lancés par le noyau de MadKit, qui propose notamment les services de gestion des groupes et de communication. Il est ainsi possible d'échanger des messages directement à un agent ou à l'ensemble d'un groupe. Cette plateforme est surtout intéressante pour l'approche organisationnelle qu'elle met en avant lors de l'analyse et de la conception d'un système multi-agents.

Volcano de Magma La plateforme Volcano développée par Pierre-Michel Ricordel dans le cadre de sa thèse, est basée sur la méthodologie Voyelles et une approche componentielle. Un langage de description d'architecture de composants (Madel) est défini pour déclarer les dépendances entre les différentes briques constituant le comportement de l'agent. Le noyau de l'agent est basé sur un mécanisme événementiel polarisé selon trois interfaces :

- les événements concernant l'environnement de l'agent,
- les événements concernant les interactions de l'agent,
- les événements concernant les organisations auxquels l'agent appartient.

Cette plateforme est intéressante car les problématiques de génie logiciel liées à la réutilisabilité sont abordés et favorisés par l'utilisation de composants logiciels.

Swarm du Swarm Development Group [36] :

Swarm est un environnement de simulation de société d'agents à grande échelle (ie. en nombre d'agents) initialement développé au Sante Fe Institute. Dans Swarm une simulation est constituée d'un ensemble d'agents et d'un calendrier indiquant ce qu'un agent peut envoyer comme message et à quel moment. L'association des agents au calendrier définissant la dynamique des échanges constitue un modèle dans Swarm. Nous citons cette plateforme car c'est la plus reconnue dans le domaine de la vie artificielle. Il faut cependant bien noter que ce type de plateforme ne s'attaque pas aux mêmes problématiques que celles que nous étudions dans ce document.

36. Swarm du Santa Fe Institute : `http://www.swarm.org/`

Chapitre 2

Un modèle d'agent minimal générique

"*On devrait tout rendre aussi simple que possible, mais pas plus.*"

Albert Einstein, Autobiographical notes

Bien que de nombreux de modèles d'agents (Agent0 [73], BDI[70]) aient été proposés, il n'existe pas de définition formelle de "*ce qu'est un agent*"[37]. Même si un consensus existe autour de certaines caractéristiques dont doit faire preuve un agent (section 1.3), chaque modèle proposé n'insiste souvent que sur quelques unes de ces capacités. Le manque d'infrastructure commune empêche bien souvent la comparaison des différents modèles d'agents. Notre proposition part de cette constatation, et a pour but de fournir un modèle d'agent minimal et générique, ou plutôt une infrastructure logicielle facilitant le développement de différents modèles cognitifs d'agents.

Après avoir précisé ce que nous entendons par modèle d'agent, nous introduisons notre modèle d'agent minimal générique. Le fondement de ce modèle est d'une part une recherche sur les fonctionnalités fondamentales d'un agent, et d'autre part la création *interactive* d'agent. Cette première phase se concentre sur l'identification des fonctions suffisantes et nécessaires à la réalisation des différents modèles cognitifs d'agent existants. Il faut préciser que nous ne nous intéressons pas à la description du comportement individuel des agents, mais bien aux fonctionnalités communes aux différentes modèles existants. La deuxième phase consiste par contre à proposer un modèle d'agent (ou une infrastructure) dans lequel les caractéristiques intrinsèques à la notion d'agent sont réifiées : la gestion des interactions, la gestion des connaissances et la gestion des organisations.

2.1 Qu'est-ce qu'un modèle d'agent ?

Avant de présenter notre infrastructure d'agent, nous allons préciser ce que nous entendons par modèle cognitif/comportemental d'agent. Nous avons vu dans le chapitre 1.3,

qu'un agent était une entité autonome possédant un comportement. La définition de ce comportement dépend des fonctionnalités attendues de l'agent. Dans le cadre de la robotique, les agents réactifs sont préférables car des comportements relativement réalistes peuvent être obtenus grâce à une combinaison de comportements plus simples (architecture de subsumption de Brooks, figure 1.14 page 43). Par contre, dans le cadre d'agents de négociation, les modèles sont généralement situés à un niveau d'abstraction supérieur et reposent sur des modèles cognitifs plus complexes.

Ainsi, un *modèle d'agent* définit le cadre dans lequel le concepteur pourra exprimer les comportements de l'agent. Nous avons vu qu'il existait un nombre important de modèles d'agents. Cependant lorsqu'on s'intéresse aux plateformes les implémentant, on constate qu'ils se répartissent principalement en deux groupes : les modèles *ad'hoc* et les modèles de type BDI. Les modèles de type BDI constituent la classe la plus homogène, et il n'est pas trop difficile de passer d'un modèle à l'autre. Les modèles *ad'hoc* sont par contre généralement totalement différents : concevoir un agent dans JADE, à base de comportements dont on doit spécifier l'ordonnancement, ne correspond pas du tout à la création d'un agent dans GRASSHOPPER pour lequel il est nécessaire de surcharger la méthode `live()`. De plus, le comportement d'un agent est caractérisé par les principes qu'il utilise pour répondre à un message. Les modèles d'agents ont ainsi à répondre aux interrogations suivantes :

- comment traiter un message ?
- comment associer le traitement d'un message au programme réalisant ce traitement ?
- comment décider de la prochaine action à effectuer (du prochain message à émettre) ?

Un modèle d'agent devra répondre à ces différentes questions et proposer des concepts ou des outils facilitant le développement d'agents.

2.2 Le modèle d'agent de Magique

MAGIQUE proposait initialement un modèle d'agent réactif, proche d'un système d'objets concurrents répartis[37]. Le développeur héritait d'une classe `Agent`, et les méthodes publiques définissaient les services que l'agent proposait au système. De plus, un mécanisme de délégation de tâches s'appuyant sur une organisation hiérarchique des agents permettait d'invoquer de manière transparente des méthodes quelle que soit leur localisation. Par ailleurs, chaque agent pouvait surcharger une méthode `action()` définissant le comportement proactif de l'agent. Le cycle de vie de l'agent consistait alors à exécuter l'algorithme suivant :

- A la réception d'une requête, exprimée sous la forme d'un nom de méthode et d'une liste de paramètres, l'agent vérifiait si sa classe définissait la méthode demandée.
- Si l'agent possédait cette méthode, il l'invoquait et retournait le résultat à l'agent ayant émis la requête,
- Sinon, il recherchait dans son équipe si l'un de ses subalternes pouvait répondre à la requête.

37. Pour une description plus complète de MAGIQUE se reporter au chapitre 4.

– Si c'était le cas, il lui demandait de traiter la requête, sinon il déléguait à son supérieur hiérarchique la gestion de cette requête.

Cet algorithme étant exécuté par tous les agents, pour peu que la méthode demandée était définie dans l'une des classes d'agent et qu'au moins une instance de cette classe était présente dans le système, la requête était satisfaite. Dans le cas contraire, la requête restait bloquée en attente à la racine de la hiérarchie jusqu'à ce qu'un agent possédant la méthode rejoigne le système.

Ainsi, le développement d'un système multi-agents avec MAGIQUE consistait à définir dans un premier temps un ensemble de classes JAVA caractérisant les différentes classes d'agent. Dans un deuxième temps, l'organisation de ces agents était définie au travers d'une hiérarchie. Ensuite, les agents surchargeant leur méthode `action()` initiaient les interactions au sein du système. Cette approche du développement d'un système multi-agents est très proche du développement de logiciel classique : après une phase d'analyse, le système est implémenté et ensuite testé.

Bien que nous ayons la possibilité d'intervenir dynamiquement sur le système, via l'envoi de messages, il n'est pas possible de modifier le comportement des agents.

2.2.1 L'introduction de la notion de compétence dans Magique

A partir du moment où l'on abstrait une tâche, on caractérise une compétence. C'est-à-dire une suite cohérente d'opérations ayant pour but d'effectuer une opération précise. Une fois la compétence définie, un agent peut l'acquérir et ensuite proposer ce service au système. Cette définition de la notion de compétence est similaire à celle de la notion de composant. En effet, les composants se distinguent des objets par leur suffisance[38] et l'interface qui les définit (section 1.3, page 1.2.2.0.0). Les compétences doivent aussi posséder cette suffisance, c'est-à-dire qu'elles doivent pouvoir être transférées facilement, et n'être utilisables qu'au travers de leur interface.

La première modification de MAGIQUE consista à introduire la notion de compétence comme une classe JAVA que l'on pouvait ajouter à un agent. L'ensemble des méthodes publiques de cette classe devenait ainsi accessible à l'agent, et au travers de l'agent au système multi-agents. Nous nous sommes ensuite naturellement intéressés à l'échange de compétences entre agents. En effet, disposer de l'échange de compétences facilite la construction incrémentale d'un système et ouvre de nouvelles possibilités telles que la gestion d'accès à certaines fonctions au cours de la vie d'un agent (se reporter à la première application *diapo-conférence* du chapitre 5 pour un exemple).

2.2.2 L'échange de compétences dans Magique

La principale problématique liée à l'échange de compétences réside dans le fait que MAGIQUE est une plateforme distribuée. Ainsi, lorsque le concepteur définit une hiérarchie d'agents, il peut ensuite répartir chacun des agents sur une station de travail, ou encore tous les agents sur une seule machine. Dans ce dernier cas, l'échange de compétences

38. C'est-à-dire qu'un composant sera généralement livré sous la forme d'une archive contenant l'ensemble des ressources nécessaires à son fonctionnement.

ne pose pas de problème; par contre, lorsque des agents se trouvant sur différentes machines décident d'échanger une compétence, la gestion du transfert de code nécessaire à l'exécution de cette compétence se pose.

Pour gérer l'échange de compétences, nous avons dans un premier temps utilisé les propriétés du langage JAVA. En effet, dans ce langage orienté objets, les classes sont stockées sous forme binaire dans des fichiers appelés *bytecode*. Ces fichiers de *bytecode* correspondent à l'assembleur qui est interprété par les machines virtuelles JAVA (ou JVM pour *Java Virtual Machine*). C'est d'ailleurs ce code intermédiaire, qui peut être considéré comme un langage d'assemblage générique, qui offre à ce langage son indépendance vis-à-vis du système d'exploitation et de l'architecture matérielle sous-jacente.

La problématique de l'échange de compétences correspond à celle du code mobile : il est nécessaire de pouvoir charger du code dynamiquement dans l'environnement (ie. la JVM) et de déterminer l'ensemble des classes devant être transférées. Le premier obstacle est résolu grâce au mécanisme de `ClassLoader` proposé en JAVA. Cette classe définit la manière dont une JVM recherche une classe et la charge en mémoire. Il est ainsi possible d'étendre cette classe pour, par exemple, rechercher les classes sur le réseau au lieu du système de fichier local de la machine. Le second obstacle est un peu plus ardu, notamment à cause du mécanisme d'instanciation tardif qu'utilise le langage. Ce mécanisme a pour objectif de n'instancier que les objets nécessaires au fonctionnement d'un programme. Ainsi, lorsqu'une nouvelle classe est chargée en mémoire, le chargement des classes qui lui sont liées, ne s'opère que lors de l'instanciation de ces objets.

Imaginons par exemple une classe `A` qui possède un attribut de classe `B` (figure 2.1). Lors du chargement de la classe `A`, c'est-à-dire lors de la première instanciation d'une variable de classe `A`, la *Jvm* chargera le fichier `A.class` contenant le bytecode de la classe et définira cette nouvelle classe dans l'environnement. La classe `B` sera aussi chargée, car elle est utilisée dans le constructeur. Par contre, la classe `C` ne sera chargée que lors de la première invocation de la méthode `m()`.

```
public class A {

    protected B b = null;

    public A() {
        b = new B();
    }

    public void m() {
        C c = new C();
    }

}
```

FIG. 2.1 – *Exemple d'une classe illustrant l'instanciation paresseuse*

Ce mécanisme, bien qu'intéressant au niveau de la gestion de la mémoire, empêche l'utilisation du `ClassLoader` seul pour calculer les dépendances d'une classe. Pour ob-

tenir ces informations, il faut étudier plus précisément la structure d'un fichier de *bytecode*. La figure 2.2 détaille la structure d'un fichier de *bytecode*. On trouve tout d'abord une constante appelée *magic number*[39], identifiant le début d'un fichier de bytecode, un numéro de version, un pool de constantes, des drapeaux d'accès, ainsi que les informations concernant la classe étendue, les interfaces implémentées, les attributs et les méthodes.

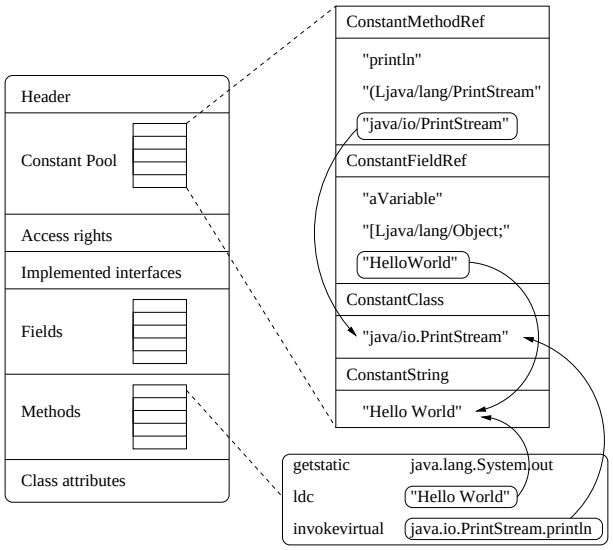

FIG. 2.2 – *Structure d'un fichier de* bytecode *d'une classe* JAVA

Pour ce qui nous intéresse, c'est-à-dire la liste des classes utilisées dans la classe courante, nous pouvons trouver l'information dans le pool de constantes. En effet, c'est dans ce pool que réside l'ensemble des constantes définies au sein de la classe. Chaque constante est caractérisée par un drapeau indiquant la nature de la constante (nom d'attribut, de méthode, de classe ...) suivie de sa valeur.

Le calcul du graphe de dépendance consiste donc à extraire les constantes de classe, et à réitérer le processus sur les classes obtenues jusqu'à saturation. On obtient ainsi la clôture du graphe de dépendance de la classe. Ensuite, nous avons profité de l'intéressante caractéristique fournie par le mécanisme de sérialisation défini en JAVA - à savoir la sauvegarde implicite du graphe de dépendance de l'objet que l'on sérialise - pour établir un mappage de notre graphe de dépendance avec le graphe des objets à sérialiser. Ainsi, lorsque l'on demande la sérialisation d'une classe, on obtient automatiquement la clôture des classes dépendantes. Il suffisait pour cela d'établir la correspondance entre nos objets contenant les archives des classes (ie. leur *bytecode*) et les dépendances structurelles entre les classes JAVA.

La figure 2.3 illustre le processus d'échange de compétence : un agent A demande à un agent B se trouvant sur une machine distante de lui transmettre la compétence C. L'agent B n'envoie que la classe C, mais la plateforme P2 grâce au mécanisme de sérialisation rajoute les classes dont la classe C dépend. Lorsque la plateforme P1 reçoit l'archive, elle l'intégre à l'ensemble des classes pouvant être instanciées, et l'agent A peut ensuite créer une instance de la compétence C. L'échange est donc effectif : si l'agent B

39. Pour information, la valeur identifiant un fichier de bytecode est : 0xCAFEBABE.

quitte le système, l'agent A pourra toujours utiliser la compétence qu'il a acquise. D'un autre côté, dans le cas d'un utilisateur nomade, l'acquisition d'une compétence par un agent sur son portable lui permettra de continuer ses tâches tout en étant déconnecté du système.

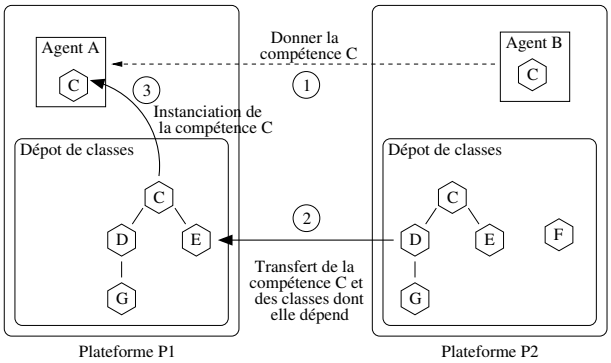

FIG. 2.3 – *Echange de compétence et transfert du* bytecode *entre deux plateformes.*

Cette approche, bien qu'intéressante par l'automatisation et la transparence qu'elle fournit, pose certains problèmes lorsque le concepteur désire utiliser des librairies tierces. En effet, une compétence ayant à manipuler des documents XML devra analyser ces documents et nécessitera donc l'utilisation d'une librairie manipulant de tels documents. La compétence référencera donc une ou plusieurs classes d'une librairie tierce, ce qui impliquera une dépendance vis-à-vis de ces classes. Lorsque cette compétence devra être échangée, l'ensemble (tout au moins la majeure partie) de la librairie sera inclus dans l'archive (et si l'on utilise par exemple le parseur XERCES, cela représente plusieurs centaines de classes).

Pour éviter ce désagrément, une option de la plateforme MAGIQUE permet d'exclure des classes du mécanisme de calcul de dépendances en spécifiant les paquetages ne devant être pris en compte. Il y a bien évidemment les classes fournies par la plateforme JAVA (`java.*`, `javax.*` ...), et les éventuelles librairies tierces. Malheureusement, il devient alors nécessaire pour le concepteur de s'assurer que les librairies tierces sont installées sur chacune des plateformes.

Pour pallier plus sérieusement à cette importante limitation, nous avons du réimplémenter complètement la plateforme MAGIQUE, en nous inspirant des travaux menés dans la communauté objets autour de la notion de composant logiciel. Ceci nous a mené dans un premier temps à définir un modèle d'agent minimal générique, puis dans un second temps à un modèle d'agent générique facilitant le développement d'agents se conformant aux spécifications de protocoles d'interaction que nous utilisons dans notre méthodologie RIO (chapitre 3).

2.3 Notre modèle d'agent générique minimal

Le modèle que nous proposons dans cette section a pour but d'unifier les différentes approches existantes en définissant un noyau minimal de compétences d'une part, et de

factoriser les développements de différents modèles d'agent en proposant une infrastructure commune d'autre part.

2.3.1 Une construction par spécialisation incrémentale

Notre modèle se base sur une construction incrémentale des agents à partir d'un agent élémentaire (ou atomique) par enrichissement dynamique de compétences. Une compétence est un "*ensemble cohérent de capacités*". Du point de vue de l'implémentation, une compétence peut être perçue comme un composant logiciel regroupant un ensemble cohérent de fonctionnalités. Les compétences peuvent donc être développées indépendamment de tout agent et être réutilisées dans différents contextes.

Nous affirmons que deux compétences initiales sont nécessaires et suffisantes à l'*agent atomique* pour que celui-ci puisse évoluer pour réaliser le modèle d'agent désiré. Nous définissons un *agent atomique* comme une entité douée de deux compétences : pour la première, une compétence de communication, et pour la seconde, une compétence qui permet à l'agent d'apprendre de nouvelles compétence. Un *agent* est ainsi un *agent atomique* qui a appris des compétences suite aux interactions auxquelles il a participé.

Ces compétences sont *nécessaires* car sans la "*compétence d'acquisition*", un tel agent ne serait qu'une coquille vide incapable de faire quoi que ce soit, et sans la "*compétence de communication*", un agent est isolé et perd de ce fait tout intérêt. De plus la communication est nécessaire à l'acquisition de nouvelles compétences. Elles sont *suffisantes* puisqu'il suffit à un agent d'utiliser sa compétence de communication pour entrer en contact avec un agent compétent et d'utiliser sa compétence d'acquisition pour apprendre de nouvelles compétences auprès de celui-ci.

2.3.2 Retour sur la notion de compétence

Nous avons vu qu'une compétence représentait un ensemble cohérent de fonctionnalités, et pouvait être implémentée sous la forme de composants logiciel. Ce qui est intéressant avec une approche componentielle est que le concepteur délègue la gestion du cycle de vie de son composant au conteneur. En fonction du modèle de composant choisi, des propriétés telles que la persistance, la gestion des transactions ou de la sécurité, sont mutualisées au sein du containeur. En outre, par définition les composants sont consistants et destinés à être réutilisés dans différents contextes.

Une compétence est constituée de deux parties : son *interface* et son *implémentation*. L'interface spécifie les messages entrants et sortants, tandis que l'implémentation réalise les traitements de ces messages. Cette séparation découple la spécification de la réalisation, et permet ainsi d'avoir plusieurs implémentations pour une interface donnée. Ainsi, lors du déploiement, l'implémentation peut être choisie en fonction du contexte d'exécution.

L'interface de compétence est définie par un ensemble de motifs de messages qu'elle accepte et produit. Ces messages devant être discriminés, il est nécessaire de les typer.

```
interface := ((m_in)*, (m_out)*)+

    m_x = motif de message
```

Le typage des motifs de messages peut prendre plusieurs formes : soit un typage fort, qui a l'avantage de spécifier complètement les interfaces, soit un typage faible qui offre plus de souplesse en ce qui concerne l'évolution de l'interface.

Si l'on applique cela à MAGIQUE, les messages ont la forme :

`réponse méthode(paramètres)`,

où `réponse` est une classe (la valeur de retour de la méthode) et `paramètres` une liste de classes (les paramètres de la méthode). Ainsi, les interfaces de service correspondantes sont : `((méthode(paramètres)), (réponse))`. Nous avons dans ce cas un typage fort des messages, qui correspond au typage des objets du langage JAVA.

Si maintenant, nous décidons d'utiliser WSDL pour spécifier les interfaces de service (ce qui revient à écrire des Services Webs), le typage sera fort et reposera sur les outils de validation XML. D'ailleurs, dans ce cas, la seule différence avec MAGIQUE est principalement l'encodage utilisé, puisque les informations représentées sont les mêmes (invocation de méthode à distance).

Si finalement, nous voulons utiliser des messages KQML, avec par exemple le contenu des messages exprimé en KIF, la granularité du typage dépendra du niveau de vérification effectué. Si l'on vérifie le performatif et la racine du contenu, le typage sera faible. Par contre, si l'on vérifie le performatif et l'intégralité du contenu du message, le typage sera fort.

Cette variance au niveau de la précision du typage est une propriété intéressante car elle facilite le prototypage, tout en laissant la possibilité de préciser ou de figer ultérieurement l'interface. Cette propriété est d'autant plus aisée à mettre en place lorsqu'on passe d'une approche orientée objets à une approche orientée documents (cf. figure 1.9 de la section 1.3, page 34).

2.3.3 Les différentes implémentations possibles de la notion de compétence

Nous avons vu au début de ce chapitre, que la réalisation initiale des compétences dans MAGIQUE reposait sur le principe des interfaces et classes JAVA. Nous allons dans un premier temps détailler un exemple complet d'application pour illustrer la création, l'instanciation et l'utilisation des compétences dans MAGIQUE. Comme nous l'avons vu précédemment, pour pouvoir échanger les compétences, un mécanisme de calcul de dépendance est nécessaire et ne résout que partiellement le problème, notamment à cause des compétences utilisants des librairies tierces. Nous présenterons donc dans un second temps, la solution que nous avons choisi pour améliorer la gestion des dépendances entre les compétences en nous reposant sur la notion de composant logiciel.

Un exemple complet d'implémentation avec Magique. L'exemple que nous allons étudier sera volontairement très rudimentaire (on pourrait même peut être contester son caractère vraiment agent). L'intérêt de sa simplicité est de permettre de montrer l'ensemble du code et de présenter ainsi les principaux concepts de programmation de systèmes multi-agents avec MAGIQUE. Nous allons montrer ici que, pour toute personne

connaissant le langage JAVA, le surcoût de l'utilisation de MAGIQUE est quasiment nul tout en permettant une distribution facile de l'application sur un réseau[40].

Le problème

Il s'agit d'un calcul de vérification de la conjecture dite de Collatz ou de Syracuse ou encore le problème $3x + 1$.Le problème étudié est le suivant :

Considérons la fonction f qui a tout entier n associe :

$$f(n) = \begin{cases} \frac{n}{2} & \textit{si } n \textit{ est pair} \\ 3n+1 & \textit{si } n \textit{ est impair} \end{cases}$$

La conjecture affirme qu'à partir de tout n, on peut toujours atteindre 1 après suffisamment d'itérations de la fonction f :

$$\forall n \in \mathbb{N}, \exists k \in \mathbb{N}, \underbrace{f(...(f}_{k \textit{ fois}}(n)..) = 1$$

Il s'agit donc de disposer d'un système multi-agents qui pour un entier n donné produit la suite des valeurs des itérations de la fonction jusqu'à 1.

Les compétences

Le compétences nécessaires à la résolution de ce problème peuvent être facilement identifiées :

`ParitySkill :` le test de la parité,

`MultSkill :` multiplication de deux entiers,

`AddSkill :` addition de deux entiers,

`DividerSkill :` division de deux entiers,

`CollatzSkill :` calcul de la conjecture.

Il faut donc écrire le code de chacune de ces compétences. Pour cela, il suffit de créer une classe qui hérite directement ou indirectement de l'interface `Skill` du paquetage `fr.lifl.magique.skill`. Chacune des méthodes publiques de cette classe pourra ête utilisée par tout agent qui possèdera cette compétence.

Pour les quatre premières compétences, rien de particulier à ajouter par rapport à un code JAVA classique. La figure 2.4 montre le code complet de la compétence `Parityskill`, le code des trois autres est immédiat.

La compétence `CollatzSkill` nécessite un peu plus d'explications. En effet, pour accomplir sa tâche, elle doit faire appel aux autres compétences. C'est ici qu'interviennent les primitives MAGIQUE d'invocation de compétences mentionnées précédemment. Dans ce cas particulier, puisqu'il s'agit de calcul nécessitant que la réponse à une requête soient

40. Le code source complet de cet exemple peut être récupéré (et donc testé) sur le site : `http://www.lifl.fr/MAGIQUE`.

```
public class ParitySkill implements fr.lifl.magique.Skill {

    public Boolean isEven(Integer x) {
        return new Boolean( (x.intValue()%2) == 0 );
    }

} // ParitySkill
```

FIG. 2.4 – *Le code source complet de la compétence* ***ParitySkill***

connues avant de poursuivre, c'est la méthode `askNow` qui doit être utilisée. Ainsi, pour invoquer la méthode `isEven` de la compétence `ParitySkill` sur l'`Integer x`, il faut écrire :

```
Boolean value = (Boolean) askNow("isEven",x);
```

On obtient pour la compétence `CollatzSkill` le code complet donné à la figure 2.5.

On peut noter aussi l'invocation de la compétence `display`, grâce à la primitive `perform`, car ici il n'y a pas de réponse attendue. Il s'agit de déléguer à la compétence qui convient (par défaut la compétence `DisplaySkill` présente dans tout agent MAGIQUE gère cette invocation) la réalisation de l'affichage demandé.

Voici donc pour les compétences impliquées. On voit que MAGIQUE n'apporte que peu de particularités en comparaison de classe et méthodes JAVA. La seule vraie différence est que pour certains appels, là où en objet on écrit :

```
objet.method(args...);
```

avec MAGIQUE, on a :

```
perform("method",args...);  // ou ask/askNow
```

Une différence notable est que le destinataire n'a pas à être nommé (à la différence de la programation objet). Un des intérêts évident est que la compétence pourra facilement être réutilisée dans différents contextes (cela est facilement imaginable pour les quatre premières compétences) et avec différents agents. On peut d'ailleurs remarquer que nous avons étudié les compétences sans même encore avoir discuté des agents qui seront impliqués dans le systèmes multi-agents. C'est ce que nous allons faire maintenant.

La hiérarchie et les agents

Il s'agit de répartir les compétences parmi les agents, puis d'organiser ces agents dans une hiérarchie. Nous allons imposer arbitrairement la structure initiale et la répartition des compétences. Nous travaillerons avec sept agents : un pour chacune des compétences précédemment citées, un agent qui supervisera les quatre agents possédant les compétences arithmétiques de base et un superviseur global du système multi-agents. D'autres choix auraient pu être faits.

On dispose donc d'une organisation hiérarchique correspondant à la structure présentée à la figure 2.6. Rappelons que l'existence de cette structure hiérarchique permet la gestion automatique de la gestion des invocations des compétences. Ainsi, lorsque la compétence `CollatzSkill` requiert la compétence `ParitySkill`, l'agent `collatz` n'a pas besoin de

```
import fr.lifl.magique.*;
import fr.lifl.magique.skill.*;
public class CollatzSkill extends MagiqueDefaultSkill {
   private Integer x;
   public CollatzSkill(Agent myAgent, Integer x) {
     super(myAgent);
     this.x = x;
   }
   private Boolean testParity(Integer x) {
     return (Boolean) askNow("isEven",x);
   }
   private Integer xByTwo(Integer x) {
     return (Integer) askNow("quotient",x,new Integer(2));
   }
   private Integer threeTimesPlus1(Integer x) {
     Integer y = (Integer) askNow("mult",x,new Integer(3));
     return (Integer) askNow("add",y,new Integer(1));
   }
   public void conjecture(Integer x) {
     this.x = x;
     conjecture();
   }
   public void conjecture() {
     while (x.intValue() != 1) {
       perform("display",new Object[]{"x  = "+x.intValue()});
       if (testParity(x).booleanValue()) {
           x = xByTwo(x);
       }
       else {
           x = threeTimesPlus1(x);
       }
     }
     perform("display",new Object[]{"x = 1 ** finished"});
   }
} // CollatzActionSkill
```

FIG. 2.5 – *Le code source complet de la compétence* ***CollatzSkill****.*

connaître explicitement l'agent `parity`, MAGIQUE gère l'acheminement de la requête. En fait, la seule chose qui importe est de savoir que la compétence `ParitySkill` est présente dans le système.

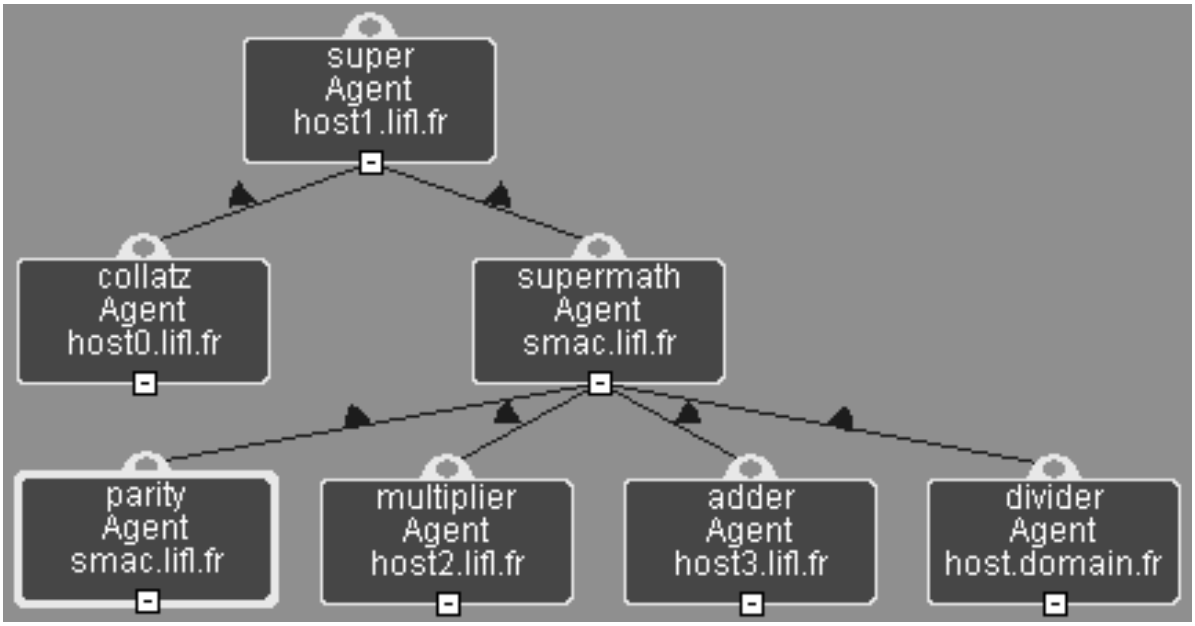

FIG. 2.6 – *La structure hiérarchique choisie*

Venons-en à la création de nos agents. MAGIQUE se base sur la notion de plateforme, qui est en fait l'équivalent de la JVM pour MAGIQUE. Cette plateforme prend en charge les problèmes liés à la distribution qui est ainsi masquée pour le développeur. *A priori*, on ne trouve qu'une plateforme par machine, même si cela n'est pas une obligation. Un agent est donc créé par une plateforme et il faut ensuite faire son éducation en lui enseignant les compétences que l'on souhaite lui donner.

La figure 2.7 présente la création de l'agent possédant la seule compétence `ParitySkill`. Cet agent doit se rattacher dans la hiérarchie à son superviseur. C'est ce qui est fait par l'appel à la méthode `connectToBoss`. Ici, on suppose que ce superviseur se trouve dans la même plateforme (ce superviseur n'a besoin d'aucune compétence particulière, il n'est là que pour chapeauter les agents mathématiques) et qu'il doit lui-même être connecté au superviseur global appelé `"super"` et placé dans une autre plateforme sur la machine `host1.lifl.fr`. Cette simple commande de connexion va suffir pour gérer les communications distantes entre les agents.

Pour les autres agents, il en va de même. Il faut juste savoir quels agents évoluent sur la même machine et donc dans la même plateforme. Cependant, pour que le calcul se fasse, il faut invoquer la compétence `conjecture()` ; cela peut être fait par exemple après la création de l'agent `collatz` (cf. figure 2.8). Mais avec le mécanisme de délégation, cette invocation aurait pu être faite par n'importe quel autre agent.

Si l'on dispose en plus des programmes, `ParityImp`, `CollatzImp`, des autres `AdderImp`, `MultiplierImp`, `DividerImp`, `SuperImp` écrits sur le même principe (en supposant donc

```
import fr.lifl.magique.*;
public class ParityImp {
  public static void main(String[] args) {
    // création et lancement de la plate-forme
    Platform p = new Platform();
    // création du superviseur "mathématique"
    Agent supermath = p.createAgent("supermath");
    // connexion à son superviseur
    supermath.connectToBoss("super@host1.lifl.fr:444");
    // création de l'agent pour la parité
    Agent parity = p.createAgent("parity");
    // enseignement de la compétence de parité
    parity.addSkill("ParitySkill");
    // connexion à son superviseur
    parity.connectToBoss("supermath");
  }
}
```

FIG. 2.7 – *Le code source complet de création du superviseur mathématique et de l'agent possédant la compétence* `ParitySkill`

```
import fr.lifl.magique.*;
public class CollatzImp {
  public static void main(String[] args) {
    Platform p = new Platform();
    Agent colAgent = p.createAgent("collatz");
    colAgent.connectToBoss("super@host1.lifl.fr:444");
    colAgent.addSkill("CollatzSkill", colAgent, 17);
    // calcul de la conjecture avec x=17
    colAgent.perform("conjecture");
  }
}
```

FIG. 2.8 – *Le code source complet de création du superviseur mathématique et de l'agent possédant la compétence* `ParitySkill`

Architectures orientées clients	Architectures orientées serveurs
– OLE, *Object Linking and Embedding*, – COM, *Component Object Model*, – ACTIVEX, successeur des OCX, *Ole Controls eXtension*), – JAVABEANS, composants légers,	– COM+, une extension de COM, – AVALON, composants dédiés serveurs, – EJB, *Enterprise Java Beans*, – CCM, *Corba Component Model*, – OSGI, composants embarqués

FIG. 2.9 – *Les deux familles d'architectures orientées composants*

que tous ces agents se trouvent sur des plateformes (donc *a priori* machines différentes), on peut maintenant, en exécutant ces programmes sur leurs machines respectives, exécuter ce système multi-agents.

De Magique à G Nous avons vu dans l'exemple précédent que les interactions qui se produisent entre les agents, ne sont pas définies au niveau de l'agent, mais au sein de ses compétences. Il n'est pas possible dans MAGIQUE de savoir quelles sont ces dépendances, sans avoir à analyser le code source (ou le bytecode) des différentes compétences. D'autre part, cela signifie aussi que la gestion des communications, et plus précisément des conversations, doit s'effectuer au sein des compétences.

D'autre part, nous avons vu que la gestion de la communication, et donc de l'invocation de compétences, était possible grâce à la hiérarchie. Cette organisation facilite le routage des messages et permet la mise en place du système de délégation de compétences utilisé lors de l'appel à la cantonnade. Cependant, dans MAGIQUE il n'y a qu'une hiérarchie, qui concentre différentes fonctionnalités. Il serait possible de généraliser cette approche en autorisant de multiples organisations (hiérarchiques ou d'autres topologies) au sein du même système multi-agents.

Ce sont ces observations qui nous ont amené à approfondir ces différentes problématiques :

– la réification de la notion d'interaction,

– la réification de la notion d'organisation,

– l'implémentation de la notion de compétence.

Les deux premiers points seront détaillés dans le chapitre **??**, par contre nous allons présenter les solutions que nous avons envisagées concernant le changement du modèle de compétence reposant sur la notion *simple* de classe JAVA, vers la notion plus complexe, mais plus riche de composant logiciel.

Nous nous sommes donc intéressés aux différentes propositions faites dans le cadre des technologies orientées objets et plus particulièrement sur l'approche orientée composants (se reporter à la section 1.3). Il existe actuellement beaucoup de modèles de composants différents, bien souvent adaptés à des domaines d'applications spécifiques. Principalement, deux grandes familles se détachent : les composants orientés *clients* et ceux orientés *serveurs* (figure 2.9).

Les principales différences entre les deux familles de composants résident généralement

au niveau de la rigueur des règles d'écriture des composants (pour les JavaBeans, une simple convention syntaxique, alors que les Ejb nécessitent l'implémentation de plusieurs interfaces) et les services proposés par les conteneurs (configuration et adaptation pour les JavaBeans, gestion de la recherche, de la persistance et des transactions pour les Ejb). Une autre différence importante se trouve au niveau de la gestion des interactions entre les composants. Dans les cas des JavaBeans, l'utilisation de classes intermédiaires faisant le lien entre deux composants est préconisée, notamment à cause de la vocation de développement graphique des JavaBeans (malheureusement, ce type d'environnement n'a pas vraiment fonctionné), ainsi que l'utilisation d'un bus à événements[41].

Quel que soit le modèle choisi, les étapes de développement et de déploiement sont proches et consistent à :

- décrire le composant (c'est-à-dire son interface),
- implémenter le composant,
- créer le paquetage contenant le composant,
- expliciter l'assemblage des différents composants,
- déployer les composants et leur assemblage,
- administrer le conteneur, qui gère ensuite le cycle de vie des composants, ainsi que certains aspects non fonctionnels (sécurité, transaction, persistence ...).

La figure 2.10 illustre le cycle de développement de systèmes à base de composants, et identifie les métiers caractérisant les différentes phases de conception, déploiement et maintenance du système.

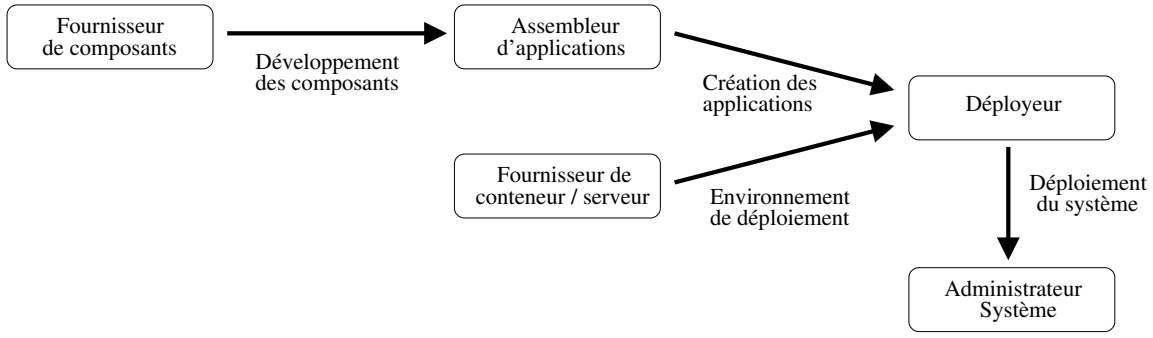

Fig. 2.10 – *Le cycle de développement et de déploiement d'un système à base de composants*

Dans le cas de Magique, l'utilisation d'un modèle de composant pour réaliser les compétences des agents est directe. En effet, la plupart des modèles existants définissent les interfaces de composants en terme d'interface Java. Une première approche consiste donc à associer un conteneur à chacun des agents. Malheureusement, avec des modèles de composants *lourds* comme les Ejb, cette approche n'est pas viable, et il devient préférable de n'avoir qu'un conteneur de composant partagé par l'ensemble des agents. L'intérêt de disposer d'une plateforme (se reporter à la section 1.3.3) mutualisant les fonctionnalités communes devient d'autant plus cohérent.

41. InfoBus : `http://java.sun.com/products/javabeans/infobus/`

Cependant, ne voulant pas nous lier à un modèle de composant particulier, nous avons défini une architecture abstraite dans laquelle les services sont définis en terme de messages dont le contenu est exprimé en OWL et dont l'implémentation peut être réalisée avec différents modèles de composant. D'une certaine manière, nous appliquons le même principe que celui défini dans le *Model Driven Architecture* de l'OMG1.3 en ce qui concerne la distinction entre PIM, *Platform Independant Model*, et PSM, *Platform Specific Model*.

2.4 Vers une infrastructure d'agent générique

Le modèle d'agent minimal générique est intéressant d'un point de vue formel. Cependant il ne définit pas de modèle de comportement, il est nécessaire de l'enrichir pour disposer d'une infrastructure plus intéressante. Nous allons donc dans cette section présenter la structuration interne que nous proposons comme modèle d'agent générique en nous appuyant sur le modèle minimal.

2.4.1 Les différents niveaux d'abstraction des compétences

Nous avons identifié quatre couches qui sont caractérisées par différents niveaux d'abstraction des fonctionnalités proposées (figure 2.11). Le premier niveau correspond aux compétences "systèmes", c'est-à-dire les fonctionnalités minimales permettant de *bootstraper* un agent : la communication (émission/réception de messages) et la gestion de compétences (acquisition/retrait dynamique de compétences)[56]. Le deuxième niveau identifie les compétences *agents* : la base de connaissances, un media d'interaction pour les compétences et le lieu de représentation des connaissances de l'agent, la gestion des protocoles d'interaction et la gestion des organisations. Le troisième niveau correspond aux compétences définissant le modèle d'agent (réactif, BDI ...), tandis que le dernier niveau représente les compétences purement applicatives.

Plus que les compétences réalisant ces différents niveaux, ce sont les fonctionnalités qu'elles représentent qui sont fondamentales : la gestion des communications, tout comme la base de connaissances peuvent être implémentées de différentes manières, par contre, il est nécessaire de disposer de ces fonctions au sein de l'agent. Nous allons maintenant détailler les différents niveaux identifiés.

2.4.2 Compétences systèmes : le noyau minimal

Cette première couche contient les compétences minimales nécessaires à la construction incrémentale de l'agent. On y retrouve donc la capacité de communication, c'est-à-dire le point d'entrée de l'agent pour émettre et recevoir des messages, et la fonction de gestion des compétences, c'est-à-dire l'ajout et le retrait dynamique de compétences. Pour que ce premier niveau soit fonctionnel, il est nécessaire d'ajouter à ces deux compétences un interpréteur de message *système*. Ainsi, l'agent minimal générique ne peut traiter que les messages *système* associés aux fonctionnalités de gestion de compétences.

L'interprète se réduit à un processus vérifiant s'il y a de nouveaux messages dans la boîte de réception de l'agent, qui est gérée par la compétence de communication, et à

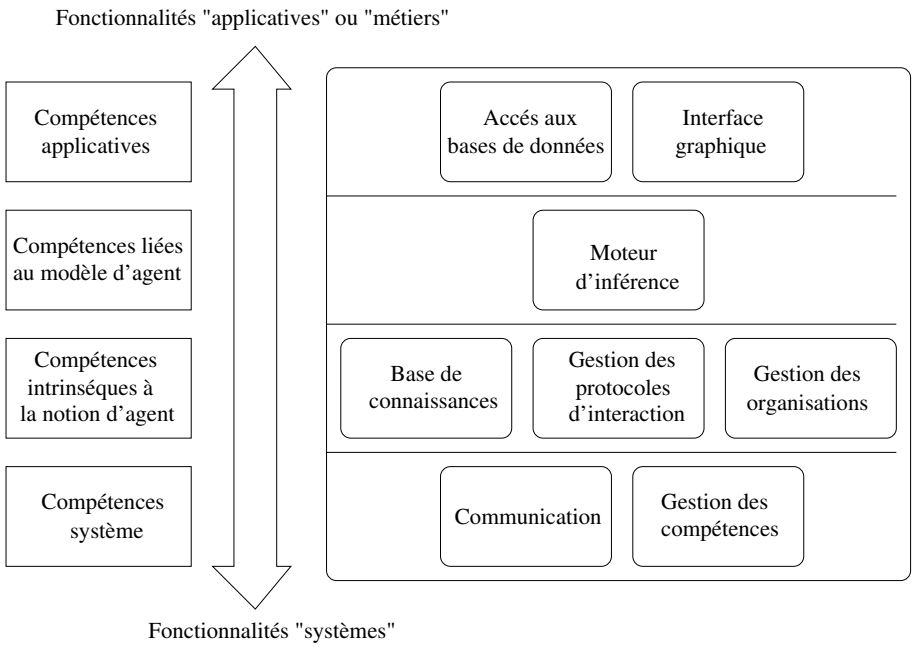

FIG. 2.11 – *Les quatre niveaux d'abstractions de notre modèle d'agent*

traiter les messages *système* d'ajout ou de retrait de compétence. Cette implémentation se rapproche du système des *handlers* dans les langages d'acteurs.

2.4.3 Compétences agents : les fonctions intrinsèques

Ce deuxième niveau introduit des fonctionnalités propre à la notion d'agent. On retrouve donc à ce niveau les compétences pouvant être utilisées par les différents modèles d'agent existants : la base de connaissance, la gestion des organisations et la gestion des protocoles d'interaction. Le lecteur pourrait objecter que tous les modèles d'agent n'utilisent pas la notion d'organisation, ni même de protocole d'interaction. Cependant, ils utilisent tous un mécanisme permettant de rechercher des accointances. De même, pour des interactions dépassant le stade de l'invocation de méthode, il est nécessaire pour l'agent de gérer des conversations (et particulièrement leur état).

Donc, même si tous les modèles d'agent ne définissent pas explicitement ces fonctions, ils les intègrent implicitement dans le comportement de leurs agents. L'objectif de ce second niveau est de réifier ces fonctions, en les extrayant du comportement de l'agent et en les proposant à un niveau inférieur à celui de la définition du modèle d'agent.

2.4.4 Compétences liées au modèle cognitif de l'agent

Ce troisième niveau spécialise les agents en définissant leur modèle cognitif, c'est-à-dire la manière dont ils traitent les messages qu'ils reçoivent et ainsi l'association avec les services utilisés pour leur traitement. Le troisième niveau regroupe les différentes compétences nécessaires au fonctionnement du modèle d'agent. Les niveaux précédents

peuvent être considérés comme fournissant les services systèmes facilitant et orientant l'implémentation des compétences du modèle cognitif de l'agent.

Ainsi, la compétence représentant un automate à états finis dans le cas d'agents réactifs, ou un moteur d'inférence dans le cas d'agents BDI, se retrouvera à ce niveau d'abstraction. Le modèle cognitif se repose sur la couche précédente pour exprimer la gestion des connaissances, des accointances et des interactions, mais impose aussi au niveau suivant, celui des compétences applicatives, le contrat que les services doivent respecter pour être utilisés par le modèle d'agent.

2.4.5 Compétences applicatives

Ce dernier niveau regroupe les compétences développées par le concepteur utilisant le *framework*. A ce niveau, le concepteur se repose sur les couches précédentes pour implémenter les différents services utilisés pour définir le comportement de l'agent. Les compétences développées pour ce niveau sont généralement moins réutilisables car fortement liées à l'application réalisée.

Mise à part les compétences liées au métier, les compétences réalisant les interfaces avec l'utilisateur sont définies à ce niveau. Ces dernières sont celles qui sont le plus facilement réutilisables : un navigateur (interprète de langage HTML) ou un encodeur/décodeur de flux audio/vidéo, peuvent facilement être intégrés dans différentes applications. Ce sont aussi ces compétences qui pourront être échangées dynamiquement par les agents pour modifier et faire évoluer le système au niveau applicatif. Il est aussi possible de changer des compétences des niveaux inférieurs, mais il s'agira alors de modification *système* telle que le changement de protocoles de transport bas niveau (encryption, qualité de service ...).

2.4.6 Implémentation du modèle générique d'agent

De plus en plus de systèmes multi-agents sont disponibles et ré-implémentent généralement des fonctionnalités présentes dans les autres systèmes. Il n'y a que très peu de mutualisation au niveau des développements, même sur les aspects de bas niveau telle que la couche de communication. L'effort initié autour de JAS[42], *Java Agent Services* était important de ce point de vue, mais ne semble malheureusement pas aboutir. En effet, la proposition de spécification et l'implémentation de référence sont disponibles, mais la standardisation n'est pas encore effectuée. Nous pensons que l'acceptation par les milieux industriels ne se produira pas tant qu'une plateforme commune, ou au moins une spécification à laquelle se conformerait les différentes implémentations, ne sera disponible.

Les efforts menés par la FIPA sont importants, notamment sur l'aspect "standardisation". Malheureusement, il est difficile de comprendre la raison pour laquelle la FIPA ne fournit pas d'implémentation de référence des différentes spécifications qu'elle propose.

Il y a plusieurs avantages à diffuser une implémentation de référence conjointement à une spécification. Le premier et le plus important des avantages est de démontrer la

42. Plus d'information sur JAS : `http://www.jcp.org/aboutJava/communityprocess/review/jsr087/`

faisabilité. Le deuxième avantage est de fournir à la communauté des développeurs un moyen d'expérimenter facilement les nouvelles spécifications sans avoir à les implémenter. Un troisième avantage, plus contestable mais indéniable en pratique, consiste à lever *de manière pragmatique* les ambiguïtés de la spécification. Spécifier totalement une Api n'est pas aisé, et l'implémentation de référence peut servir de comportement *par défaut* à appliquer lorsque la spécification n'est pas assez précise.

Un des objectifs que nous poursuivons dans l'implémentation de notre *framework* est de se reposer sur les librairies existantes et, dans la mesure du possible, de découpler les différentes fonctionnalités liées à la gestion des agents au sein d'une plateforme. L'objectif à terme étant de distribuer la plateforme, soit pour une utilisation *directe* (développement purement applicatif), soit pour une utilisation *indirecte* en réutilisant différents composants du *framework* pour créer une nouvelle plateforme. Nous détaillerons l'implémentation de notre plateforme dans le chapitre 4.

2.5 Avantages de notre modèle d'agent minimal générique

Indépendamment de l'implémentation retenue pour représenter la notion de compétence, ce paradigme de construction d'agents par enrichissement dynamique de compétences, procure de nombreux avantages tant au niveau de la conception qu'au niveau de l'exécution :

le développement est facilité. Construire un agent, c'est lui enseigner des compétences. La programmation d'un agent est donc "réduite" à celle de compétences, et une fois qu'une compétence a été développée, elle peut être réutilisée dans différents contextes. Les compétences peuvent être vues comme des *composants logiciels*, avec tous les avantages liés à cette notion : modularité, réutilisabilité, etc. ;

efficacité. Un agent peut décider de déléguer l'accomplissement de telle ou telle tâche à un autre agent (si celui-ci l'accepte). Mais cela a un coût (dû à la communication par exemple) et dépend du bon vouloir de l'autre. C'est pourquoi dans certains cas, s'il a à accomplir souvent cette tâche par exemple, un agent peut "préférer" apprendre une compétence et supprimer ainsi la nécessité de la déléguer. D'un autre point de vue, si l'agent "se sent" submergé par des requêtes des autres pour exploiter une de ses compétences, il peut choisir de l'enseigner à d'autres agents (qu'il aurait d'ailleurs pu créer et éduquer dans ce but spécifique) afin d'alléger sa charge ;

robustesse. Si pour quelque raison un agent doit disparaître du SMA et qu'il est le détenteur d'une compétence critique, il peut l'enseigner à un autre agent du SMA et ainsi garantir la pérennité et la cohérence de l'ensemble ;

autonomie et évolutivité. Au cours de sa "vie", une compétence donnée d'un agent peut évoluer et être améliorée et de nouvelles compétences peuvent être ajoutées. Ainsi, l'agent accroît ses capacités et son autonomie ;

évolution dynamique. Quand une compétence d'un agent doit être changée (*a priori* pour l'améliorer), il n'est plus nécessaire d'accomplir le cycle classique (et pénible) : "l'arrêter, modifier le source, compiler et redémarrer". La nouvelle compétence peut être dynamiquement enseignée à l'agent (qui doit oublier l'ancienne). Cela peut être

particulièrement important pour un agent dont la durée de vie est longue et qui est dédié à un rôle qui ne peut tolérer la moindre interruption ;

adaptation à l'environnement d'exécution. Si vous considérez un agent disposant d'une faible capacité mémoire (situé sur un téléphone portable ou un assistant personnel), vous pouvez choisir de charger votre agent avec uniquement les compétences requises à un moment donné (et le décharger des autres).

Chapitre 3

RIO : Rôles, Interactions et Organisations

> "*Interaction is more than an exchange of messages. Issues associated with it, are : models of agents (beliefs, goals, representation and reasoning), interaction protocols (an interaction regime that guides the agents) and interaction languages (languages that introduce standard message types that all agents interpret identically).*"
>
> A semantic approach for KQML, Yannis Labrou and Tim Finin[49].

Dans ce chapitre, nous présentons l'approche que nous préconisons pour la réalisation de systèmes ouverts constitués d'entités en interaction. Pour cela, nous mettons en avant la notion de spécification exécutable de protocole d'interaction et la notion d'organisation. Nous décrirons donc dans un premier temps le modèle d'interaction que nous proposons, puis nous préciserons l'utilisation que nous faisons de la notion d'organisation lors de l'exécution de ces protocoles. Nous terminerons par la description de notre méthodologie Rio, acronyme pour *Rôles, Interactions et Organisations*, qui se base sur les spécifications exécutables de protocole d'interaction pour faciliter la conception, le déploiement et la maintenance de systèmes multi-agents ouverts.

3.1 Un modèle de spécifications exécutables d'interactions multi-partites

Dans cette section, nous introduisons dans un premier temps la problématique à laquelle nous nous intéressons, à savoir la modélisation de protocoles d'interaction faisant intervenir plusieurs agents. Cette problématique se situe dans le cadre plus général de

Coordination process	Computer Science	Economics and Operations Research	Organization Theory
Managing shared resources (including **task assignments**)	techniques for processor scheduling and memory resource allocation	analyses of markets and other resource allocation mechanisms : scheduling algorithms and other optimization techniques	analyses of different organizational structures : budgetting processes, organizationnal powers, and **resource** dependance
Managing producer / consumer relationships (including prerequisites and usability constraints)	data **flow** and Petri net analyses	PERT charts, critical path methods : scheduling techniques	Participatory design : market research
Managing simultaneity constraints	synchronization techniques, mutual exclusion	scheduling techniques	meeting scheduling; certain kinds of process modeling
Managing **task** / subtask relationship	modularization techniques in programming; planning in artificial intelligence	economies of scale and scope	strategic planning; management by objectives; methods of grouping people into units

FIG. 3.1 – *Différentes interprétations de la notion de coordination selon les domaines d'application (extrait de [51])*

la coordination d'un ensemble d'entités. Le tableau 3.1 ci-dessous, illustre les différentes significations que peut prendre la notion de coordination en fonction du domaine.

Ce tableau est intéressant car il montre bien les recoupements existants entre la théorie des organisations, les théorie économiques et l'informatique. Il identifie ainsi les similarités entre différents concepts provenant de plusieurs disciplines et suggère les passerelles envisageables où la collaboration entre informaticiens et autres spécialistes pourrait être fructueuse. Nous allons dans un premier temps préciser la problématique de la coordination dans le cadre des protocoles d'interaction. Puis, nous décrirons la notion de spécification exécutable de protocole d'interaction, avant de mettre en perspective les apports du point de vue de l'ingénierie de systèmes distribués ouverts.

3.1.1 Problématique des protocoles d'interaction

La conception de systèmes informatiques pose le problème de la décomposition d'une tâche en un ensemble de structures et de fonctions interagissant selon des principes bien définis. Que l'on soit dans le cadre de logiciels mono-machine ou de logiciels répartis, la gestion des dépendances entre les différentes entités du système est cruciale et a de profondes implications sur les propriétés de stabilité, d'évolutivité et de maintenance du système. Cette gestion des dépendances peut être caractérisée par une métrique : le degré de couplage d'un système. Cette métrique permet de quantifier les dépendances existantes d'un système en comptabilisant les liaisons entre les différentes entités.

Plus un système est couplé, plus il est difficile à maintenir et à faire évoluer. A l'inverse, un système peu couplé ne sera que peu affecté par une modification sur l'une de ses entités. Ceci est très important car cela implique que certaines modifications sur le système n'entraîneront pas de propagation dans différentes parties du logiciel. Plus précisément, lorsque le concepteur ne rajoute ou ne retire pas de nouvelles entités dans son système, et que ces modifications ont lieu au sein d'une entité, il y a une isolation évitant la

propagation de modifications multiples à différents endroits du système.

Pour diminuer le couplage au sein d'un système, différentes approches ont été expérimentés dans les langages de programmation. L'une des plus répandues consiste à séparer l'interface d'une fonctionnalité de son implémentation. Cela correspond à la notion de spécification et de corps en ADA, ou à la notion d'interface et de classe réalisant une interface en JAVA. Nous appellerons ce type de découplage, découplage de l'implémentation. D'autre part, le couplage peut aussi être diminué en utilisant un typage agnostique vis-à-vis du langage d'implémentation sur les informations échangées par les interfaces. C'est l'approche qui a été adoptée avec IDL pour spécifier les interfaces de composants CORBA, ou avec WSDL pour spécifier les interfaces de Services Webs. Ce principe est similaire au concept du langage naturel *esperanto*, qui consiste à proposer un langage unique, neutre vis-à-vis des langages existants. Ceci facilite les interactions puisqu'il suffit d'apprendre un seul langage pour pouvoir communiquer avec quiconque. Nous appellerons ce type de découplage, découplage de langage, que l'on peut d'ailleurs considérer comme une généralisation du découplage d'implémentation.

Cependant, à la différence des langages naturels, les langages informatiques ont une sémantique faible. Ainsi, WSDL ne constitue qu'un artefact syntaxique et ne doit son succès qu'à l'adoption massive qu'il suscite (ce qui n'a pas été le cas pour l'*esperanto*). Aller plus loin dans le découplage pose le problème de la description de la fonction rendue par un service, ce qui revient à la sémantique même du service. Actuellement cette information n'est exprimée qu'en langage naturel, bien souvent au sein du code source même (la JAVADOC pour le langage JAVA). L'interface elle même est décrite en terme d'entrées/sortie associées à un symbole représentant le nom du traitement à effectuer. Les entrées et la sortie sont décrites par des structures de données ou des objets (ce qui pose ici le problème de transfert des classes associées aux paramètres transmis comme nous l'avons souligné dans la section 2.2.2).

Nous pensons qu'une amélioration de la situation actuelle peut provenir de la définition d'ontologies décrivant les structures de données échangées entre les entités communicantes du système. Le problème consiste ainsi à passer d'une vérification syntaxique à une vérification sémantique. Plus précisément, l'utilisation d'ontologies permet d'une certaine manière de se libérer de la forme (ie. la structure) et d'établir des liaisons (ie. des relations) sémantiquement plus riches que les liens d'héritage et d'agrégation dans les langages orientés objets.

Malheureusement, il est difficile de mesurer l'impact exact d'une telle modification, car la relation entre la structure de données représentée et le réseau sémantique la décrivant pose le problème de la séparation entre syntaxe et sémantique. De plus, par rapport à l'évolution actuelle provenant des langages orientés objets, et particulièrement avec la notion de composant logiciel, l'utilisation d'ontologies revient à séparer les structures de données des traitements les manipulant. Toutefois, cette séparation implique une nouvelle conception des applications, qui ne se reposent plus sur des informations identifiées par leur structure, mais par un ensemble de ressources (les données) accessibles selon différentes descriptions grâce aux mécanismes d'inférence.

Une vision schématique de tels systèmes, consiste à avoir des services consommant et produisant des informations sémantiquement enrichies, dont les interactions sont formalisées et interprétables par le système lui même. L'avantage est que le couplage est ainsi

identifié par l'objet formalisant l'interaction et manipulable par le système. Un agent peut ainsi connaître les interactions dans lesquelles il est engagé, et peut les gérer en refusant d'initier de nouvelles interactions ou suspendre des interactions en cours. Nous considérons ces protocoles d'interactions comme des lois sociales au sens de Shoham[74] , cela implique que les agents perdent une partie de leur autonomie, mais en contrepartie le système gagne en déterminisme et en fiabilité. Il est intéressant de remarquer que la même approche a lieu dans le domaine de la modélisation des systèmes de *workflow* avec le consortium WORFLOW MANAGEMENT COALITION[43], ou encore dans le domaine des Services Webs, avec les spécifications telles que WSCI[44], BPEL4WS[45], et WSFL[46] visant à *chorégraphier* un ensemble d'invocations de Services Webs

3.1.2 La notion de spécification exécutable de protocole d'interaction

De nombreux travaux ont été effectués pour spécifier des protocoles d'interaction. Récemment, AGENTUML[66] a été défini comme une extension d'UML, pour spécifier les conversations entre agents, notamment en spécialisant les diagrammes de séquence. Cependant, ces spécifications nécessitent l'interprétation de développeurs, qui doivent ensuite les traduire dans leur propre système. Les travaux de Labrou et Finin[21] explorent l'utilisation des Réseaux de Pétri Colorés[47] (RdPC) pour modéliser les conversations entre agents. Dans [58], la même approche est utilisée, mais la notion de Réseau de Pétri Colorés Récursifs est introduite pour faciliter la composition de conversations.

Nos travaux suivent le même principe : représenter l'interaction de manière globale, et utiliser un formalisme reconnu et établi. Cependant, contrairement aux travaux précédents, notre objectif est de produire une *spécification exécutable*. C'est-à-dire, une description du protocole d'interaction qui peut être ensuite directement intégrée dans un système s'exécutant, suite à la génération du code gérant l'interaction pour chacun des participants. De plus, les RdPC sont indéniablement adaptés à la modélisation de processus concurrents, et fournissent un formalisme graphique intéressant, mais ils ne s'utilisent malheureusement pas très aisément. C'est pourquoi il nous paraît préférable de définir un langage adapté à la modélisation des protocoles d'interactions, et d'utiliser ensuite un mécanisme de transformation permettant de traduire ce langage vers des RdPC (dans le même esprit que les travaux de Huget[46]).

Notre modèle a pour but de faciliter la spécification et le déploiement de protocoles d'interactions au sein de systèmes multi-agents. Le concepteur a pour tâche de réaliser la spécification, les autres phases étant *automatisées*. Pour cela, nous avons défini un formalisme représentant la vue globale d'un protocole d'interaction, et un mécanisme de projection qui transforme cette vue globale en un ensemble de vues locales dédiées à chaque rôle. Nous décrirons dans un premier temps la spécification de la vue globale,

43. Site de l'organisme WFMC : `http://www.wfmc.org/`
44. Web Service Choreography Interface : `http://www.w3.org/TR/2002/NOTE-wsci-20020808/`
45. Business Process Execution Language for Web Services : `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`
46. Web Services Flow Language : `http://www4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`

avant de présenter le mécanisme de projection qui génère les vues locales qu'ont les agents de l'interaction lors de l'exécution du protocole d'interaction

Le formalisme de description de protocole d'interaction

Notre modèle repose sur la notion de compétence, de micro-rôle et de graphe d'état de conversation. Un protocole d'interaction spécifie formellement le déroulement d'une conversation entre différentes entités, c'est-à-dire la nature des messages échangés, le flux de ces messages et les compétences que doivent mettre en œuvre les entités à chaque étape de la conversation. Ces entités correspondent à des micro-rôles, et sont caractérisées par leur nom, leur représentation graphique et leurs compétences. Un micro-rôle est donc une abstraction caractérisant un ou des participants de la conversation ayant la même fonction dans l'interaction. On utilise un graphe pour représenter le déroulement de la conversation : les nœuds représentent les états de chacun des micro-rôles au cours de la conversation, et les arcs correspondent à des envois de messages entre deux micro-rôles qui peuvent être associés à un moment donné de l'interaction à une interface de compétence ou à l'initiation d'un nouveau protocole, et les arcs correspondent à des envois de messages entre deux micro-rôles, typés par des motifs de message.

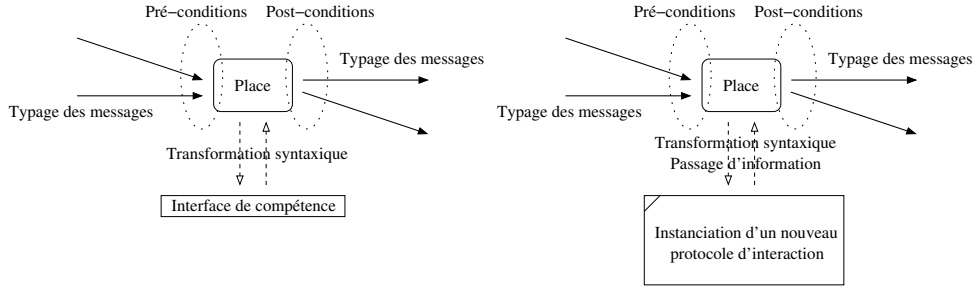

FIG. 3.2 – *Structure des nœuds constituant un protocole d'interaction*

Le bloc de base d'un graphe décrivant un protocole d'interaction est constitué d'un nœud, recevant un ensemble de messages en entrée, invoquant une compétence ou instanciant un nouveau protocole et émettant un ensemble de messages en sortie. A un instant donné de la conversation, un bloc tel que celui de la figure 3.2 représente un micro-rôle en attente de messages spécifiques en entrée, du traitement à effectuer (soit une invocation de compétence, soit l'initiation d'un nouveau protocole d'interaction), et des messages qui seront émis suite à ce traitement. Il est important de souligner que l'invocation de compétence et l'instanciation du protocole d'interaction sont effectuées de manière synchrone, c'est-à-dire que le flux du protocole courant est en attente tant que la compétence ou le protocole instancié ne terminent pas leur exécution.

La figure 3.3 présente la notation utilisée pour représenter graphiquement les spécifications de protocoles d'interaction : les figures géométriques (ronds, carrés, losanges) représentent les symboles des micro-rôles intervenant dans l'interaction, les annotations sur les arcs correspondent aux motifs de message (`mi`) ou aux insertions/extractions d'information. Lorsqu'un symbole de micro-rôle est en pointillé, cela indique un état initial du protocol si aucun message n'arrive sur cet état, et un état final sinon.

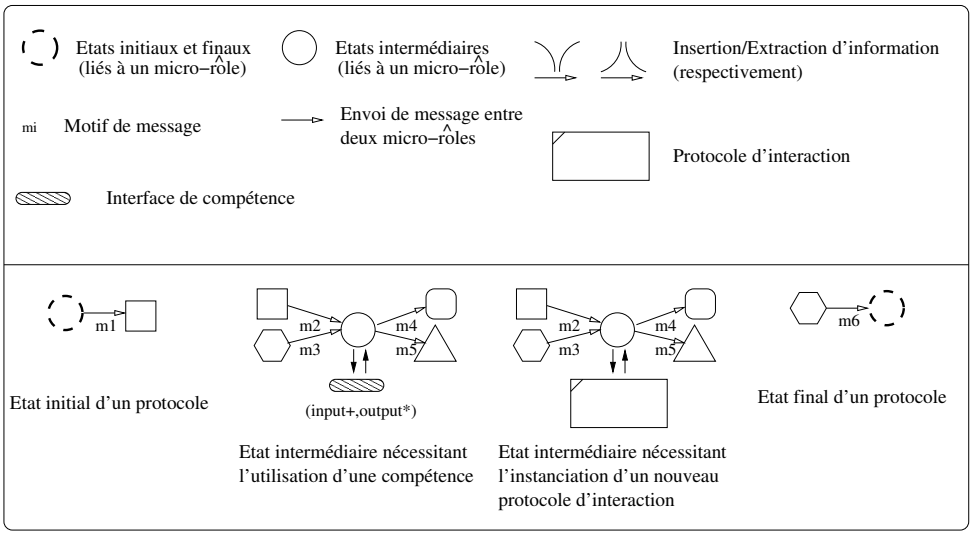

FIG. 3.3 – *Définition des éléments syntaxiques constituant un protocole d'interaction*

La figure 3.4 illustre le cartouche d'une spécification de protocole d'interaction. On constate que l'ensemble des informations se retrouvent soit dans l'entête du cartouche, soit en tant qu'annotation sur le graphe d'interaction :

- le nom du protocole d'interaction,
- une description textuelle sommaire du but de ce protocole,
- la liste des micro-rôles intervenant dans l'interaction et le symbole géométrique qui leur est associé,
- les ontologies des messages échangés,
- une liste d'informations insérées ou extraites des messages,
- un graphe d'interaction regroupant les informations comme le déroulement temporel de la conversation, la synchronisation et la nature des messages échangés, les compétences nécessaires aux différents micro-rôles.

Ce qu'il est important de comprendre c'est que le concepteur dispose ainsi d'une vue globale de l'interaction : le flux des messages, leur nature (le typage de ces messages correspond aux annotations attachées aux arcs), les compétences nécessaires et les informations utilisées ou produites.

D'autre part, toute l'information nécessaire à la gestion du protocole d'interaction est ici centralisée et peut être utilisée pour effectuer la génération de code nécessaire à la gestion de cette interaction pour chacun des micro-rôles. Le concepteur n'a donc qu'à définir le protocole d'interaction en utilisant un outil graphique, le mécanisme de projection se charge de la génération des descriptions pour chacun des micro-rôles, et le modèle générique d'agent peut ensuite exécuter ces descriptions.

Implémentations possibles des motifs de message

Une première implémentation possible de ce modèle consiste à conserver la structure des messages MAGIQUE (section 2.3.3) et à associer les interfaces de compétence aux inter-

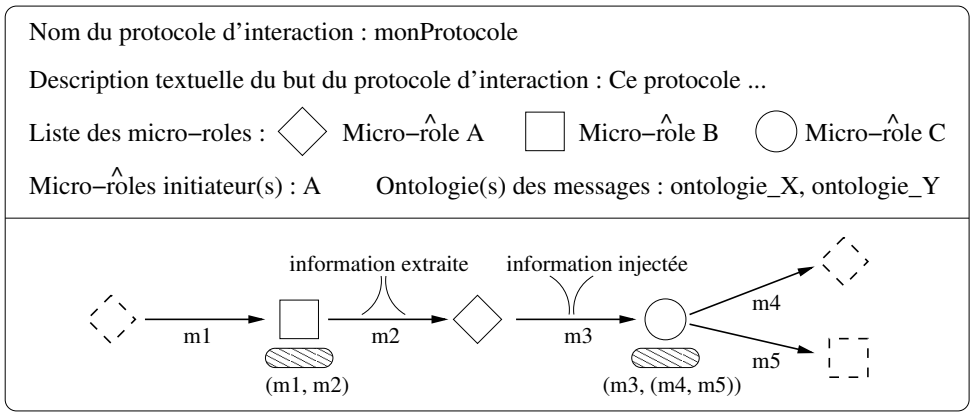

FIG. 3.4 – *Cartouche spécificiant les protocoles d'interactions*

faces JAVA. Cependant, cette approche rend le modèle spécifique à la plateforme MAGIQUE, or c'est ce que nous voulons éviter puisque nous voulons favoriser l'interopérabilité entre systèmes. De meilleurs choix technologiques reposent sur l'utilisation de langages XML, ce qui permet de devenir agnostique vis-à-vis des langages d'implémentation. Cette dernière approche permet de bénéficier à la fois du découplage d'implémentation et de langage.

Les messages peuvent ainsi être typés selon différentes technologies, telles que les XML SCHÉMA, RELAX ou encore de simples expressions XPATH. XSLT est aussi un outil parfaitement adapté pour la phase de transformation syntaxique et éventuellement pour l'expression de pré-conditions simples. Cette phase de transformation syntaxique permet de faciliter la réutilisation d'interfaces de compétence existantes. D'un point de vue plus prospectif, cette phase de transformation pourrait devenir sémantique, en procédant à des associations entre l'ontologie utilisée dans le protocole d'interaction et les interfaces de services liées.

En ce qui concerne la grammaire du langage XML représentant les messages, plusieurs approches sont de nouveaux possibles : l'utilisation de WSDL représente la transition la plus simple vis-à-vis de MAGIQUE, malheureusement la sémantique reste celle de l'invocation de méthode à distance. Le second langage qui nous est apparu comme le plus pertinent est DAML+OIL, et son successeur OWL.

Une dernière alternative intéressante est le langage DAML-S, qui définit une ontologie pour les services. Ce langage est souvent qualifié de langage de description de services internet sémantiques. Il a été récemment mis à jour et devrait à terme se baser sur OWL. La figure 3.5, extraite de [4], présente les principaux concepts de DAML-S : une ressource fournit un certain nombre de services. Un service présente un profil (*ServiceProfile*), est décrit par un modèle de service (*ServiceModel*), et peut être utilisé au travers de ses différentes implémentations (*ServiceGrounding*).

Le profil de service définit ce *que fait le service*, au travers de la description des entrées/sorties, des pré-conditions et des mécanismes d'invocation. C'est le profil de service qui sera utilisé pour effectuer la recherche de service (*matchmaking*). Le modèle de service décrit *comment le service fonctionne*, ceci est particulièrement significatif lorsque le service est composé, c'est-à-dire lorsqu'il nécessite l'invocation d'autres services. Il faut remarquer

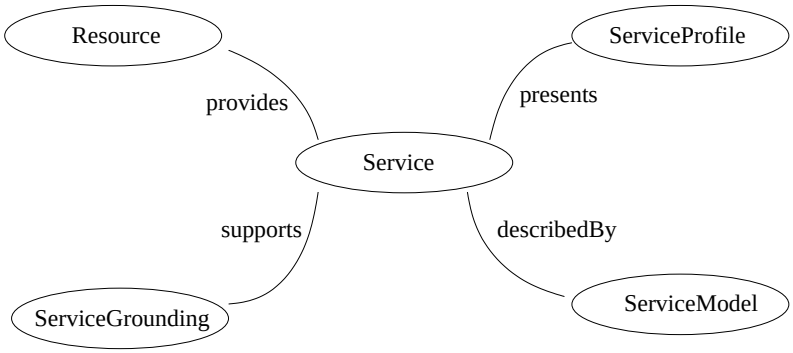

FIG. 3.5 – *Les concepts de base définissant* DAML-S

que dans l'utilisation que nous pourrions faire de DAML-S il faudrait empêcher l'utilisation de services composés, à moins qu'ils ne soient instanciés au sein d'un seul agent. Une autre possibilité consisterait à utiliser notre modèle pour spécifier ces dépendances et générer le modèle de service correspondant. Enfin, le dernier élément, le *ServiceGrounding* détaille le mécanisme d'invocation du service. Les informations définies à ce niveau concerne le modèle de composant et les protocoles de transports ainsi que les points d'accès physique au service.

Une spécification exécutable de protocoles d'interaction

La section précédente a décrit le formalisme représentant les protocoles d'interactions, nous allons maintenant expliquer la transformation permettant de rendre cette spécification *exécutable*. La spécification des protocoles d'interaction donne au concepteur une vue globale de l'interaction. Notre objectif est de générer pour chacun des micro-rôles une vue locale à partir de cette vue globale, celle-ci pourra être ensuite distribuée dynamiquement aux agents du système.

Le mécanisme de projection prend donc le fichier de spécification du protocole d'interaction [47], et génère la vue locale de l'interaction pour chacun des micro-rôles. Cette vue locale, qui correspond à ce que perçoit un micro-rôle de son point de vue, est ensuite transformée en Réseau de Pétri Coloré. Ce RDPC est d'abord exprimé en XML, puis traduit en code source JAVA, avant d'être compilé en fichier de bytecode. La figure 3.6 reprend ces différentes transformations en distinguant les étapes dépendantes et indépendantes de l'implémentation actuelle.

En effet, l'implémentation actuelle de transformation de la vue globale en vue locale et la transformation de cette vue locale en RDPC est écrite en JAVA. Une implémentation plus générique devrait reposer sur une feuille de style XSLT, qui permettrait de rendre l'algorithme indépendant de tout langage jusqu'à la phase d'interprétation des réseaux. Le fait d'utiliser une représentation en XML des RDPC permet à d'autres plateformes multi-agents d'écrire leur propre interprète. Dans notre cas, nous avons utilisé la librairie JFERN [48] pour générer les classes JAVA correspondantes.

47. Nous ne disposons pas encore d'éditeur graphique pour les protocoles d'interactions, la spécification est donc représentée sous la forme d'un fichier XML

48. Plus d'informations sur JFERN : `http://sourceforge.net/projects/jfern`

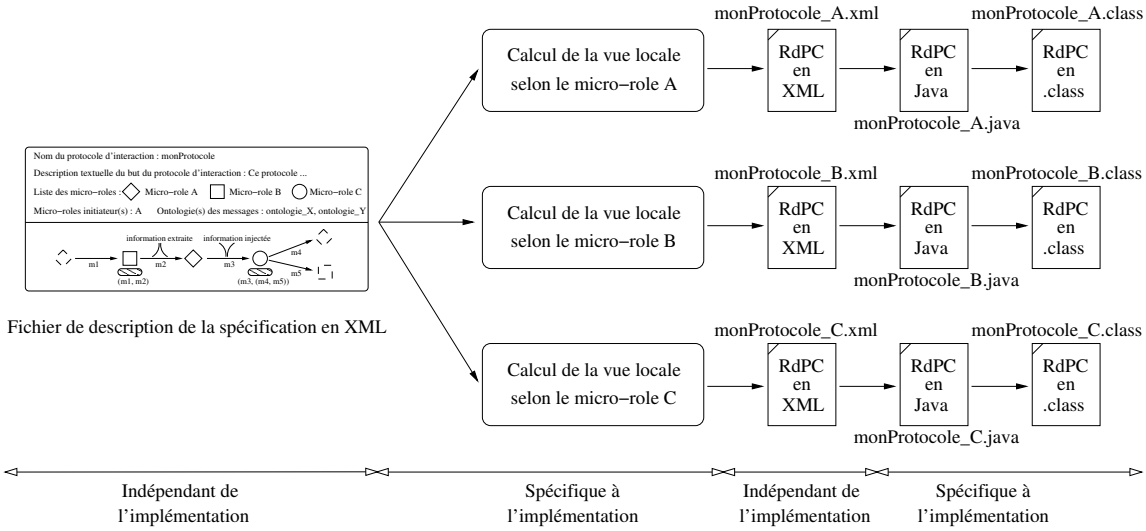

FIG. 3.6 – *Les différentes étapes de transformation de la spécification vers les fichiers exécutables*

Le principe de l'algorithme de transformation, consiste à générer les automates pour chacun des micro-rôles participant à l'interaction, à partir du fichier XML de description de la vue globale du protocole d'interaction. Pour cela, nous appliquons l'algorithme suivant :

```
1ère étape: création des graphes partiels des différents micro-rôles
   - récupération des places correspondant à un micro-rôle donné,
   - ordonnancement de ces places temporellement,
   - création des noeuds représentant les places et des arcs représentant
     les consommation et production de messages,
   - association des interfaces de compétence ou des initiations de
     nouveaux protocoles aux noeuds concernés

2ème étape : génération du fichier XML décrivant le réseau de Pétri pour
             chacun des graphes partiels
   - on parcourt chacun des graphes et l'on génére le RdPC correspondant
   - implémentation dépendante du fichier de description de JFern.
```

Pour illustrer cette phase de transformation, nous allons étudier un exemple de réalisation avec le protocole d'interaction *Contract Net*[78]. Ce protocole repose sur un mécanisme similaire à celui des appels d'offres dans les marchés publics. Ce protocole définit deux micro-rôles : un initiateur et des participants. L'initiateur cherche à trouver un interlocuteur pouvant répondre à la réalisation d'une tâche selon certaines contraintes. Il publie donc un *Call For Proposal* (appel à propositions), et attend les réponses des différents participants. Les participants doivent posséder une compétence *BID* leur permettant de répondre ou non à l'offre. L'initiateur sélectionne ensuite, grâce à sa compétence *CHOOSE*, une ou plusieurs offres le satisfaisant et demande à leurs ou à son émetteur d'effectuer la tâche en utilisant leur compétence *DO*. Les participants sélectionnés notifient ensuite l'initiateur de l'échec ou de la réussite de leur traitement.

La figure 3.7 représente un cartouche possible pour la spécification de ce protocole d'interaction. Le typage est constitué d'expressions XPATH et il y a trois interfaces de compétences définies.

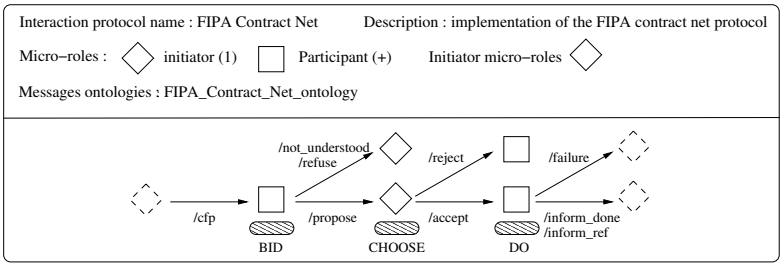

FIG. 3.7 – *La spécification du protocole d'interaction "FIPA Contract Net"*

Lors de la première phase de transformation, un RDPC est généré pour chacun des micro-rôles. La figure 3.8 illustre cette première phase, qui consiste a générer les vues partielles de l'interaction pour chacun des micro-rôles. Ensuite, ces vues partielles sont transformées en un automate qui gère le déroulement du protocole : la cohérence du protocole (ordonnancement des messages), le typage des messages et les effets de bords (invocation de compétences). Ce RDPC qui est représenté sous la forme d'un fichier XML peut ensuite être traduit en classe JAVA grâce à JFERN. Cette classe pourra ensuite être utilisé par la compétence de gestion des interactions pour participer à des interactions du type *Contract Net*.

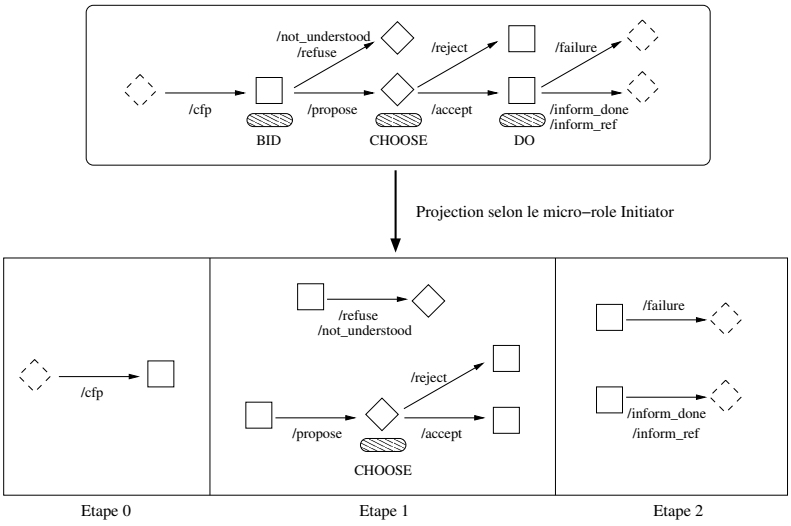

FIG. 3.8 – *La projection de la représentation globale selon le micro-rôle* ***`Initiator`***

La figure 3.9 détaille la structure des RDPC générés pour chacun des micro-rôles du protocole d'interaction. La structure est toujours similaire : les places `Inbox` et `Outbox` identifient le point d'entrée et de sortie de l'automate, la place `Terminal` représente la fin de l'interaction et les places situées entre les transitions garantissent l'aspect temporel du protocole.

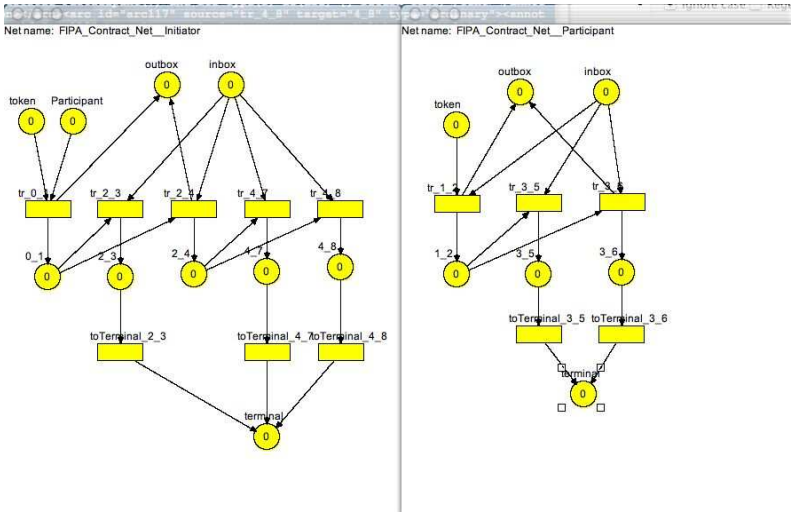

FIG. 3.9 – *Les réseaux de Pétri Colorés de l'initiateur et des participants du protocole Contract Net*

La figure 3.10 détaille le RdPC généré pour le micro-rôle *Initiator*. La place *token* contient le jeton *temps* qui garanti le déroulement temporel de la conversation, tandis que la place *Participant* contiendra les jetons des agents *Participant*s présents dans le système lors de l'initiation du protocole (cette information provient de la compétence organisationnelle). La première transition déclenche l'émission du message */cfp* à l'ensemble des *Participant*s et dépose autant de jetons *temps* dans la place *0_1* qu'il y a de *Participant*s. Ensuite, deux transitions sont déclenchables : *tr_2_3* ou *tr_2_4*. La première transition permet de traiter les refus, et mène rapidement à la fin du protocole. La seconde transition utilise la compétence *Choose* pour sélectionner la ou les propositions les plus intéressantes envoyées par les *Participant*s. Plus précisément, à chaque réception d'un message */propose* la compétence est invoquée[49], la réponse d'acceptation ou de refus est envoyé au *Participant* concerné et un jeton *temps* est ajouté à la place *2_4*.

L'automate est alors en attente du traitement de la (les) requête(s) par le(s) participant(s) : que cela soit un message de réussite (*/inform_done* ou */inform_ref*) ou d'échec (*/failure*), le protocole arrive en phase terminale. Pour éviter les problèmes d'attente infinie, lorsque par exemple un des participants quitte le système en cours d'interaction, nous utilisons un mécanisme de *timeout*. Ce mécanisme est pour l'instant global, et ne permet de spécifier un *timeout* que sur l'exécution globale du protocole d'interaction. Nous envisageons d'améliorer ce mécanisme, en ajoutant la possibilité de spécifier des *timeout* à un niveau plus fin : sur chacune des places.

Une fois la phase de transformation achevée, c'est-à-dire lorsque le réseau de Pétri est généré, il est possible de déployer l'interaction dans le système multi-agents. Il est important de préciser la neutralité des réseaux générés vis-à-vis du langage d'implémentation, car dans un premier temps, la description du réseau de Pétri est effectué au sein d'un fichier XML. Il est donc possible pour une autre plateforme d'implémenter l'interprétation

49. Nous ne disposons pas pour l'instant de concept permettant de représenter et de gérer le traitement "groupé" de l'ensemble des propositions par l'iniateur.

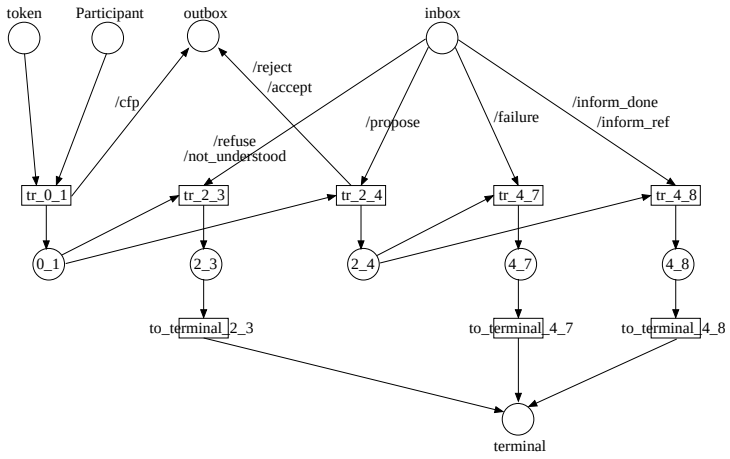

FIG. 3.10 – *Le réseau de Pétri Coloré généré pour le micro-rôle* Initiator

de ce fichier pour respecter le protocole et participer à l'interaction. Cependant, cette ouverture représente aussi une forte limitation sur les informations pouvant être exprimées dans le protocole d'interaction. Ceci explique les efforts que nous avons fait dans les choix technologiques, pour nous reposer sur les langages tels que XML, XPATH et XSLT pour être agnostique vis-à-vis des langages d'implémentation.

La phase de déploiement consiste à envoyer les descriptions aux agents concernés. Plus précisément, le déployeur devra indiquer des couples `(AgentConcret X, Micro-rôle M dans le Protocole d'Interaction P)`. L'agent `X` recevra ainsi la description du réseau généré pour le micro-rôle `M` du protocole d'interaction `P`. Dans notre implémentation, cette description est transformée en classe JAVA représentant l'automate, grâce à la librairie JFERN. A la réception d'un nouveau message, la compétence de gestion des interactions vérifiera l'identifiant de message pour associer le message à un protocole d'interaction et déléguera sa gestion à l'automate concerné. Si aucun automate ne peut traiter le message, cela signifie qu'il ne correspond pas à un message défini au sein des protocoles d'interaction connus par l'agent, et dans ce cas le traitement de ce message sera délégué au modèle d'agent.

L'intérêt de cette approche est que le concepteur spécifie graphiquement la vue globale de l'interaction, le mécanisme de projection génère ensuite les automates nécessaires à la gestion de cette interaction. De plus, grâce à l'acquisition dynamique de compétence, il est possible d'ajouter de nouveaux protocoles d'interaction aux agents du système alors que ce dernier est en cours d'exécution. C'est cette dynamicité qui établit un glissement de la conception *monolithique* d'application à une conception *incrémentale* de système. Bien évidemment, les interactions se produisant au sein du modèle d'agent entre les différentes informations produites par les compétences lors des protocoles d'interaction doivent toujours être interprétées et liées par le concepteur. Nous n'avons ni formalismes ni outils actuellement permettant de spécifier ces interactions *implicites*.

3.2 Le rôle de la notion d'organisation dans l'exécution des interactions

La gestion des connaissances et la gestion des organisations font aussi partie des fonctionnalités nécessaires et ne devant pas être liées au modèle d'agent. La gestion des connaissances regroupe à la fois leur représentation, le stockage des informations, le moyen d'y accéder et de les manipuler. L'implémentation de ces fonctionnalités est fortement dépendante du modèle d'agent utilisé : un agent réactif n'aura généralement pas de représentation de son environnement et ne nécessitera donc pas une base de connaissances aussi puissante qu'un agent de type Bdi.

Nous ne détaillerons pas les aspects liés à la base de connaissances, mais nous pensons qu'il est préférable d'utiliser les mêmes outils que ceux exprimant la sémantique des messages du système pour garder une certaine homogénéité. Particulièrement, si les messages sont représentés avec un langage tel que Owl, les informations et faits peuvent être aisément représentés dans ce langage et il est alors possible de bénéficier des mécanismes d'inférences d'Owl.

La notion d'organisation joue un rôle fondamental au sein des systèmes multi-agents : c'est un moyen permettant d'organiser logiquement les agents, c'est un réseau de communication par défaut et c'est un média de recherche d'agent, de rôle ou de compétence. De plus, sa réification fournit un point d'entrée dans le système permettant de visualiser et d'améliorer les interactions entre agents grâce à son évolution dynamique. Cette dynamicité de l'organisation facilite la tâche du concepteur, et améliore la fiabilité du système. Nous allons dans cette section aborder ces différents aspects et argumenter que les organisations tout comme les interactions doivent être des objets de premier ordre dans les systèmes multi-agents.

La notion d'organisation est nécessaire pour structurer les interactions qui interviennent entre les différentes entités du système. Chaque organisation est constituée d'un ensemble de rôles (les rôles abstraits joués par les agents) et dispose d'un rôle particulier qui sert de point d'entrée dans l'organisation. C'est ce rôle qui encapsule les fonctionnalités d'authentification et d'autorisation. Ce rôle a été identifié dans le modèle Aalaadin de Jacques Ferber[31].

Dans Aalaadin, les agents ne peuvent communiquer qu'au travers des groupes auxquels ils participent. Nous adhérons totalement à cette approche qui renforce le contrôle sur les communications entre les agents et qui formalise les responsabilités des agents au sein de l'organisation. De ce point de vue, l'organisation doit être considérée comme une entité en elle-même, indépendante des agents la constituant : les agents peuvent changer au sein de l'organisation, mais pas les rôles qui la définissent.

L'organisation, puisqu'elle gère les rôles, peut aussi constituer un mécanisme intéressant de recherche d'accointances. Utiliser l'organisation de cette manière, permet de disposer d'un système de pages jaunes *applicatif* distribué. Au sein de chaque système, il peut y avoir plusieurs organisations de différentes natures, que cela soit au niveau de la topologie ou des règles de gestion (admission / démission, algorithme de recherche de rôle, etc.).

Les organisations peuvent être créées *ex-nihilo*, ou être des spécialisations d'organisations génériques [38]. Une approche similaire à la notion de *design pattern* mais appliquée

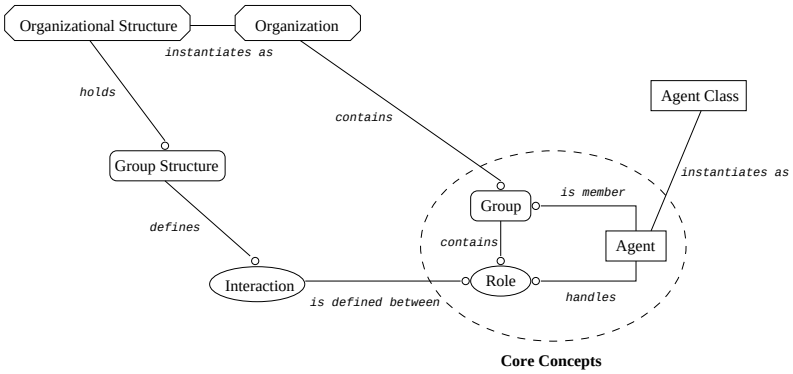

FIG. 3.11 – *Le modèle organisationnel d'*AALAADIN

à des modèles organisationnels est envisageable et permettrait de proposer au concepteur un ensemble de patron, qu'il pourrait instancier et spécialiser pour son application. Ceci permettrait de proposer aux concepteurs de systèmes multi-agents des *design pattern* organisationnels, par exemple en utilisant le formalisme i^*[91] qui se développe maintenant au sein du projet TROPOS[50] .

De la même manière que la gestion des organisations doit échapper au modèle d'agent, nous pensons que le principe de la délégation de service devrait être une fonctionnalité détenue par les organisations et ne devrait pas être liée au modèle d'agent utilisé. Bien sûr, ceci suppose qu'une ontologie partagée par l'ensemble des agents intervenant dans le système soit définie, pour représenter la notion de service et les notions liées telles que la qualité de service (performance, délai, coût ...). Nous avons cependant vu que le langage DAML-S pourrait devenir un tel langage, et cela ouvrirait d'intéressantes possibilités de délégation automatique de services. Le mécanisme utilisé dans MAGIQUE, a démontré son intérêt à la fois en terme de facilités fournies lors de la conception, mais aussi en terme d'amélioration des performances lors de l'exécution.

Effectivement, l'organisation peut constituer un média de communication initial pour les agents. Cependant, en ajoutant la gestion de la délégation de service à l'organisation, cette dernière peut ensuite optimiser les liens de communications de ses agents. Nous avons mené quelques expérimentations dans ce domaine de l'optimisation dynamique des performances d'un système, à la fois grâce à la création dynamique d'accointances par l'organisation, mais aussi par l'utilisation du mécanisme de transfert de compétence pour alléger la charge d'un agent trop sollicité [57].

3.3 RIO : vers une méthodologie de conception de systèmes multi-agents ouverts.

Dans cette section, nous introduisons la démarche méthodologique que nous employons pour la conception de systèmes multi-agents ouverts. Cette méthodologie repose principalement sur les notions détaillées dans les sections précédentes : le modèle générique

50. Des informations sur i^* et TROPOS sont disponibles sur le site :`http://www.troposproject.org/`

d'agent basé sur la notion de compétence, la réification des protocoles d'interactions et des organisations.

La figure 3.12 représente les principales étapes et les différents métiers intervenant lors de la conception, le déploiement et la maintenance de systèmes ouverts. Cette illustration est à rapprocher de la figure 2.10, mais s'en distingue en rajoutant la modélisation des interactions, qui était effectuée implicitement lors de la création de l'application par assemblage de composants.

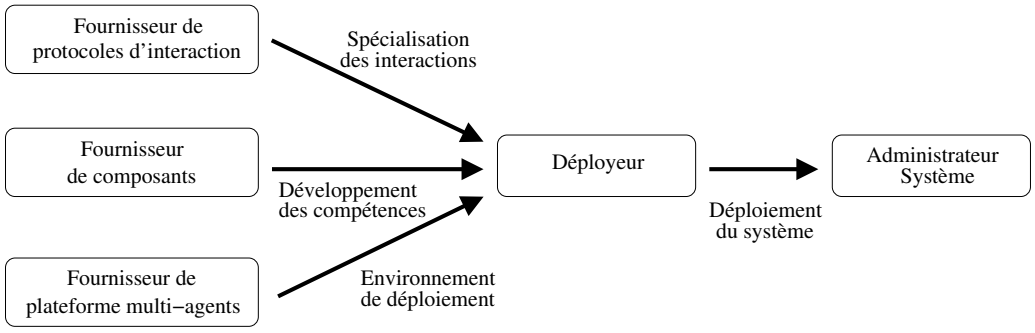

FIG. 3.12 – *Illustration synthétique des différentes étapes de la méthodologie* RIO

L'objectif de cette approche est de bien distinguer les différents métiers intervenant lors de la conception d'un système multi-agents : le créateur de composants, le créateur de protocoles d'interaction, le fournisseur de la plateforme, le déployeur et l'administrateur du système. Dans un premier temps cette décomposition peut paraître un peu exagérée, mais lorsque l'on étudie par exemple ce qui se passe effectivement au niveau de l'architecture J2EE en entreprise, on constate que ces séparations entre les différents métiers sont réelles et répondent à des besoins de gestion de la complexité et de la productivité.

3.3.1 L'objectif et le domaine d'applications de RIO

Nous allons présenter la méthodologie que nous développons, et qui repose sur la notion de spécification *exécutable* présentée précédemment. Cette méthodologie s'inscrit dans la lignée de GAIA[88], et vise donc les mêmes domaines d'applications. Cependant, GAIA reste trop générale pour pouvoir facilement passer de la phase de conception du système à sa réalisation[67]. Notre proposition a pour but de faciliter cette transition.

3.3.2 Les quatre étapes de RIO

La méthodologie RIO repose sur quatre phases, les deux premières représentent des spécifications réutilisables, tandis que les deux dernières sont singulières à l'application réalisée (figure 3.13). D'ailleurs, nous ne parlerons pas d'application, mais de société d'agents. En effet, la méthodologie RIO propose une construction incrémentale et interactive desystèmes multi-agents. Par analogie avec notre modèle minimal générique d'agent, où un agent est un réceptacle pouvant accueillir des compétences, nous voyons un système multi-agents comme un réceptacle devant être enrichi par des interactions. Nous allons détailler cette approche en étudiant les quatre phases de notre démarche méthodologique.

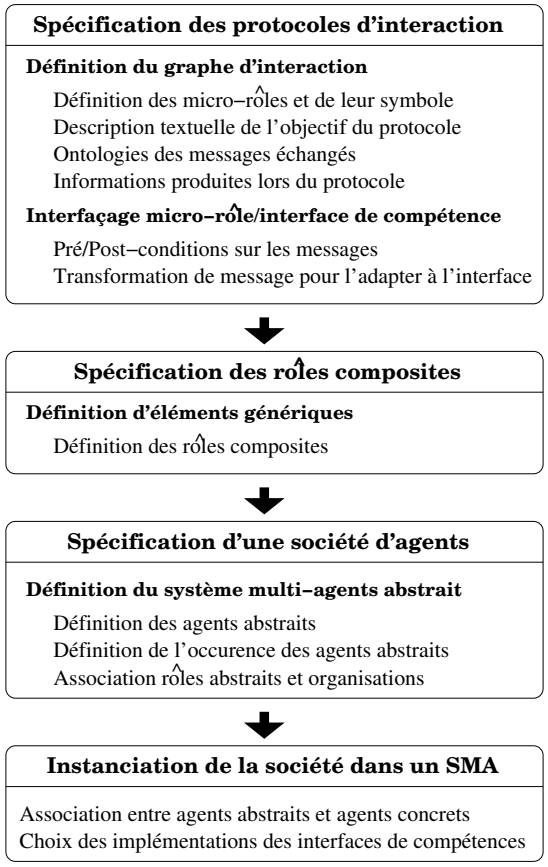

FIG. 3.13 – *Les différentes étapes de la méthodologie RIO*

La spécification des protocoles interactions

La première phase consiste à identifier les interactions et les différents rôles intervenant dans le système. Ensuite, il faut déterminer la granularité de ces interactions. En effet, pour des raisons de réutilisation de protocoles existants, il est important de trouver un équilibre entre des protocoles faisant intervenir un trop grand nombre de rôles, ces protocoles devenant ainsi trop spécifiques, et des protocoles où il n'y a que deux rôles, dans ce cas la vue de l'interaction n'est plus globale.

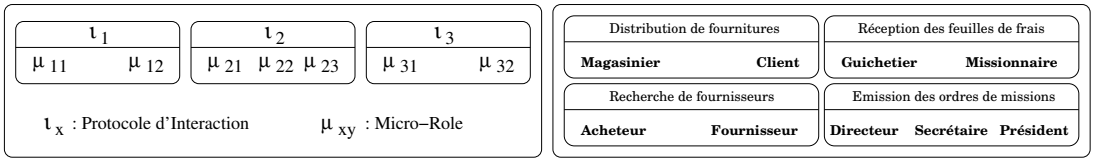

FIG. 3.14 – *Première étape : les spécifications de protocoles d'interaction*

La spécification des protocoles d'interaction peut ensuite être effectuée de différentes manières : soit *ex-nihilo*, soit par spécialisation, soit par composition. La création *ex-nihilo* consiste à spécifier le protocole d'interaction en détaillant son cartouche. La spécialisation permet d'annoter un cartouche existant. Ainsi, il est possible de spécifier le cartouche

d'un protocole d'interaction tel que FIPA ContractNet, sa spécialisation consistera à changer les noms des micro-rôles pour les adapter à l'application, à affiner le typage des motifs de messages, et éventuellement à modifier les insertions/extractions d'informations. Finalement, la composition consiste à assembler des protocoles existants en spécifiant les associations de micro-rôles et les transferts d'informations.

A la fin de cette phase, le concepteur dispose d'un ensemble de protocoles d'interaction (figure 3.14). Il peut ensuite passer à la description des rôles composites, qui va permettre l'agrégation des micro-rôles intervenant dans des interactions de même nature.

Spécification des rôles composites

Cette deuxième phase spécifie des modèles de rôle. Les rôles composites correspondent à un regroupement logique de micro-rôles (figure 3.15). En effet, un rôle se compose généralement d'un ensemble de tâches pouvant être, ou devant être effectuées par l'agent jouant ce rôle. Chacune de ces tâches peut elle même être décomposée et correspondre à un ensemble d'interactions avec d'autres rôles. La notion de rôle composite sert donc à donner une cohérence logique entre les micro-rôle représentant les différentes facettes d'un rôle.

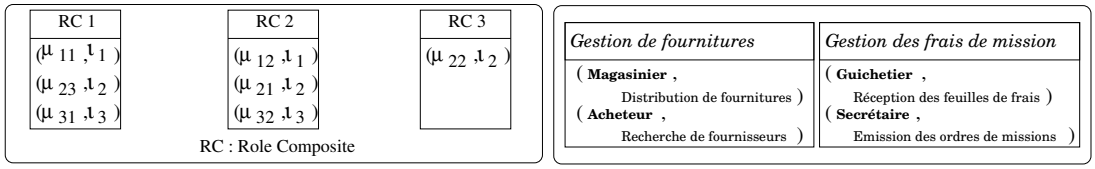

Fig. 3.15 – *Deuxième étape : agrégation de micro-rôles au sein de rôles composites*

Par exemple (figure 3.15), un rôle composite Gestion de fournitures regroupera les micro-rôle Acheteur du protocole d'interaction Recherche de fournisseurs et le micro-rôle Magasinier de l'interaction Distribution de fournitures. Ceci correspond aux deux interactions que doit maîtriser le gestionnaire de fournitures : dans un premier temps, il doit pouvoir s'approvisionner auprès des fournisseurs (micro-rôle Acheteur) et ensuite distribuer les fournitures au sein de son organisation (micro-rôle Magasinier).

Cet exemple est intéressant car il illustre une dépendance *implicite* entre deux interactions. Nous ne disposons pas de formalisme pour l'exprimer actuellement, et nous pensons qu'un travail devrait être fait au niveau des interfaces de compétences pour que l'agent puisse connaître les informations susceptibles d'être produites par la compétence dans la base de connaissance de l'agent. Ainsi, il serait possible d'aller plus loin au niveau de la vérification, en comparant les informations produites par un micro-rôle et pouvoir les corréler avec les informations consommées par d'autres micro-rôles.

Ainsi, les rôles composites peuvent être vus comme des patrons définissant des rôles *abstraits*, qui regroupent un ensemble de micro-rôles cohérents. Ces modèles sont des descriptions abstraites pouvant être réutilisées. une fois de plus la granularité des agrégats influencera la réutilisation : plus un rôle composite contient de micro-rôles, plus il sera difficile de le réutiliser dans différents contextes.

Spécification d'une société abstraite d'agents

Si l'on poursuit l'exemple précédent, le rôle composite Gestion de fournitures pourrait être associé au rôle composite de Gestion frais de mission, pour caractériser un agent abstrait Secrétaire (figure 3.16). On remarquera que l'agrégation effectuée ici est d'une granularité plus élevée que dans l'étape précédente.

Cette troisième phase peut être considérée comme une spécification d'une société abstraite d'agents, c'est-à-dire une description des agents abstraits et de leur occurrence, ainsi que la liaison des rôles composites avec les organisations. Ainsi, une fois l'ensemble des rôles composites créé, il est possible de définir les agents abstraits (des patrons d'agent au sens d'*agent template*), qui sont constitués d'un agrégat de rôles composites (figure 3.16). Ces agents abstraits décrivent des modèles d'agent spécifiques, car ils introduisent des dépendances fortes entre les rôles composites.

Une fois les agents abstraits définis, il faut préciser leur occurrence dans le système. Cela revient pour chaque agent abstrait à préciser le nombre *d'instances* qui pourront être créées dans le système (exactement n agents, 1 ou plus, *, ou encore [m..n]). Cette information n'est qu'une déclaration d'intention car il est impossible de garantir cette propriété dans le cadre d'un systèmes ouverts faisant intervenir des plateformes de différentes nature.

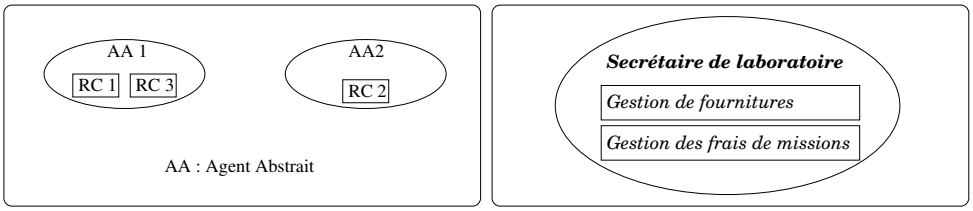

Fig. 3.16 – *Troisième étape : agrégation des rôles composites au sein d'agents abstraits*

La deuxième partie de cette phase consiste à préciser pour chacun des agents abstraits, et éventuellement pour chacun des rôles composites, quelle organisation utiliser pour trouver leurs accointances. En effet, lorsque les agents s'exécutent, pour qu'ils puissent initier des protocoles d'interactions, ils ont dans un premier temps à trouver leurs interlocuteurs. Comme nous l'avons vu dans la section 3.2, l'organisation peut servir de média à cette recherche. Cette association permet d'utiliser différentes organisations pour chacun des protocoles d'interaction éventuellement. Nous avions en effet identifier dans Magique la rigidité du modèle organisationnel car un seul plan de communication était disponible. Nous offrons donc plus de liberté et de souplesse dans ce modèle en permettant l'association d'une organisation à un agent abstrait, à un rôle composite ou encore à un protocole d'interaction au niveau le plus fin.

Instanciation d'une société dans un systèmes multi-agents

Cette dernière phase spécifie les règles de déploiement des rôles abstraits et des organisations sur les agents d'un système en cours d'exécution. Arrivés à ce point de notre démarche, nous disposons d'une spécification complète de la société d'agents, qu'il faut

maintenant projeter sur les agents concrets du système multi-agents. Pour cela, il faut indiquer les affectations des agents abstraits aux agents concrets. C'est lors de cette phase, que la liaison entre une interface de compétence et son implémentation se réalise. Le concepteur doit en effet, en fonction de critères propres à la plateforme d'accueil ou sur des critères applicatifs, lier les implémentation avec les interfaces de compétences avant que le déploiement ne puisse s'effectuer.

C'est lors du déploiement que le modèle générique d'agent se justifie pleinement grâce aux possibilités d'ajout dynamique des nouvelles compétences liées à l'interaction. En effet, l'interaction, une fois transformée par le mécanisme de projection, est représentée pour chacun des rôles par un Réseau de Pétri Coloré et des compétences associées. L'ensemble de ces informations est envoyé à l'agent, qui ajoute les compétences applicatives et délègue au gestionnaire de protocole le RdPC.

Synthèse des apports de notre démarche

La principale problématique sur laquelle nous nous sommes concentrés concerne l'ingénierie des systèmes distribués et plus spécifiquement la gestion des dépendances entre les entités en interaction dans un système. Pour cela, nous avons adopté les principes de la programmation orientée interactions et nous avons proposé un modèle de spécifications exécutables de protocoles d'interaction. Le principal apport de de ce modèle est de donner au concepteur une vue globale d'une interaction se produisant entre différentes entités, tout en laissant le système projeter cette vue globale en un ensemble de vues locales dédiées à chaque entité. Ces vues locales peuvent ensuite être utilisées par les agents pour participer aux protocoles d'interaction décrits. Nous considérons ces protocoles d'interaction comme des *motifs de conversation institutionalisés*, et rapprochons cette idée de la notion de loi sociale. Cependant, contrairement aux travaux de Dignum[27], nous considérons que les agents du système se conforment à ces règles et ne peuvent en déroger. Cela revient à limiter fortement l'autonomie des agents mais offre en contrepartie des garanties sur le déroulement des interactions, et par conséquent sur le comportement du système.

La démarche méthodologique qui consistait lors de nos premiers travaux à une intuition sur la conception de systèmes multi-agents physiquement distribués, s'est progressivement affinée particulièrment lors de l'intégration des spécifications exécutables. La figure 3.17 reprend de manière synthétique les quatre étapes présentées dans la figure 3.13 : la définition des protocoles d'interation, le groupement de ces protocoles au sein de rôles composites, l'agrégation de ces rôles composites en agents abstraits ainsi que la liaison avec les organisations, et finalement, le déploiement effectif dans le système multi-agents.

Il reste encore de nombreuses questions en suspens, comme la gestion du partage des compétences entre agents, ou encore la gestion des conversations simultanées lorsque les compétences utilisées produisent des effets de bords dans la base de connaissance. Il faudrait aussi pouvoir décrire plus précisément le formalisme de représentation des protocoles d'interaction, et approfondir la question des productions et consommations d'information des compétences dans la base de connaissances.

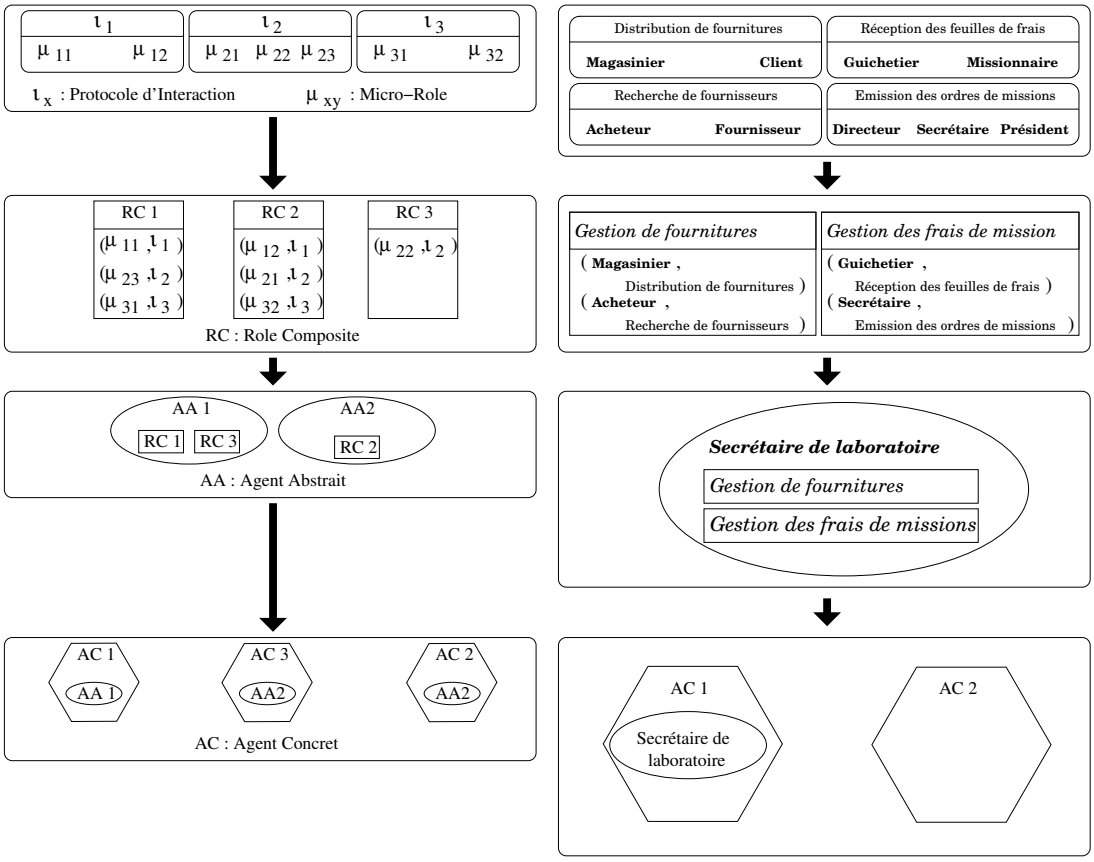

FIG. 3.17 – *Illustration synthétique des différentes étapes de RIO*

Chapitre 4

Magique et G

> "*Nothing tends so much to the advancement of knowledge as the application of a new instrument. The native intellectual powers of men in different times are not so much the causes of the different success of their labours, as the peculiar nature of the means and artificial resources in their possession.*"
>
> Sir Humphry Davy

Ce chapitre présente les logiciels que nous avons développés, ainsi que leur utilisation.MAGIQUE et G sont des *frameworks*, c'est-à-dire un ensemble de classes facilitant la conception et le déploiement de système multi-agents. Nous présenterons dans un premier temps MAGIQUE, qui est la plateforme d'expérimentation ayant précédé G, mais qui présentait un certain nombre de limitations. Dans un second temps, nous détaillerons la plateforme G, qui est en cours de développement, et qui implémente et améliore le modèle d'agent ainsi que le mécanisme de gestion des interactions.

4.1 La plateforme Magique : le précurseur de G

Cette section introduit le *framework* MAGIQUE, qui facilite la réalisation de systèmes multi-agents en proposant un modèle basé sur une organisation (par défaut) hiérarchique et sur la notion de compétence. Ce modèle est appelé MAGIQUE, ce qui signifie "MULTI-AGent hiérarchIQUE" [51]. MAGIQUE est à la base un modèle d'organisation d'agents qui propose une *organisation hiérarchique* [9]. Cette structure permet de proposer un mécanisme de délégation de compétences entre agents, facilitant ainsi le développement de systèmes distribués.

Plus concrètement, MAGIQUE existe sous la forme d'une API JAVA permettant le développement de systèmes multi-agents hiérarchiques constitués d'agents tels que ceux

51. MAGIQUE est disponible à :`http://www.lifl.fr/SMAC` rubrique MAGIQUE

décrits ci-dessus. Dans ce cas, MAGIQUE correspond à un *framework* pour le développement de système multi-agents. Le concepteur de SMA peut donc s'appuyer sur les fonctionnalités offertes par MAGIQUE telles que la communication entre agents, la distribution de l'application, l'évolution dynamique, etc. Il a la charge de développer les *compétences* applicatives dont il a besoin [54].

MAGIQUE peut être utilisée pour la réalisation de diverses applications multi-agents[52]. Citons quelques exemples non limitatifs :

- un système de vente aux enchères avec des agents acheteurs à stratégies différentes et un agent commissaire-priseur,
- le parcours d'un territoire par des agents explorateurs,
- des jeux multi-joueurs avec un agent arbitre-ordonnanceur et des agents joueurs,
- etc.

4.1.1 Les agents et leurs compétences

Dans MAGIQUE, un agent est une entité possédant un certain nombre de compétences. Ces compétences permettent à un agent de tenir un *rôle* dans une application multi-agents. Les compétences d'un agent peuvent évoluer dynamiquement (par échange entre agents) au cours de l'existence de celui-ci, ce qui implique que les rôles qu'il peut jouer (et donc son statut) peuvent évoluer au sein du SMA. Un agent est construit dynamiquement à partir d'un agent élémentaire "vide", par enrichissement/acquisition de ses compétences.

Du point de vue de l'implémentation, une compétence peut être perçue comme un composant logiciel regroupant un ensemble cohérent de fonctionnalités. Les compétences peuvent donc être développées indépendamment de tout agent et donc réutilisées dans différents contextes. Une fois ces compétences créées, la construction d'un agent MAGIQUE se fait très simplement par un simple mécanisme d'enrichissement de ces compétences par l'agent à construire (il s'agit de "l'éducation de l'agent"). L'ajout de nouvelles compétences (et donc fonctionnalités) dans une application SMA est donc facile.

La figure 4.1 décrit l'architecture générale de MAGIQUE en présentant les classes principales :

- la classe abstraite `AbstractAgent`,
- la classe `AtomicAgent`,
- la classe `Agent`,
- l'interface `Skill`,
- l'interface `Message` et la classe `Request`.

Ces classes et interfaces constituent le noyau du modèle MAGIQUE. La classe abstraite `AbstractAgent` gère un `Agenda` qui contient les accointances de l'agent et une `QuestionTable` contenant les réponses de requêtes (utilisée pour l'exécution de requêtes asynchrones). La classe `AtomicAgent` étend la classe `AbstractAgent` et rajoute une liste de messages, une table des compétences, la plateforme à laquelle est rattaché l'agent et une politique de gestion des requêtes concurrentes (gérant le traitement des réponses aux requêtes concurrentes déclenchées via l'invocation de la méthode `concurrentAsk`). Enfin,

52. voir pour différents petits exemples `http://www.lifl.fr/MAGIQUE` rubrique TOY PROBLEMS.

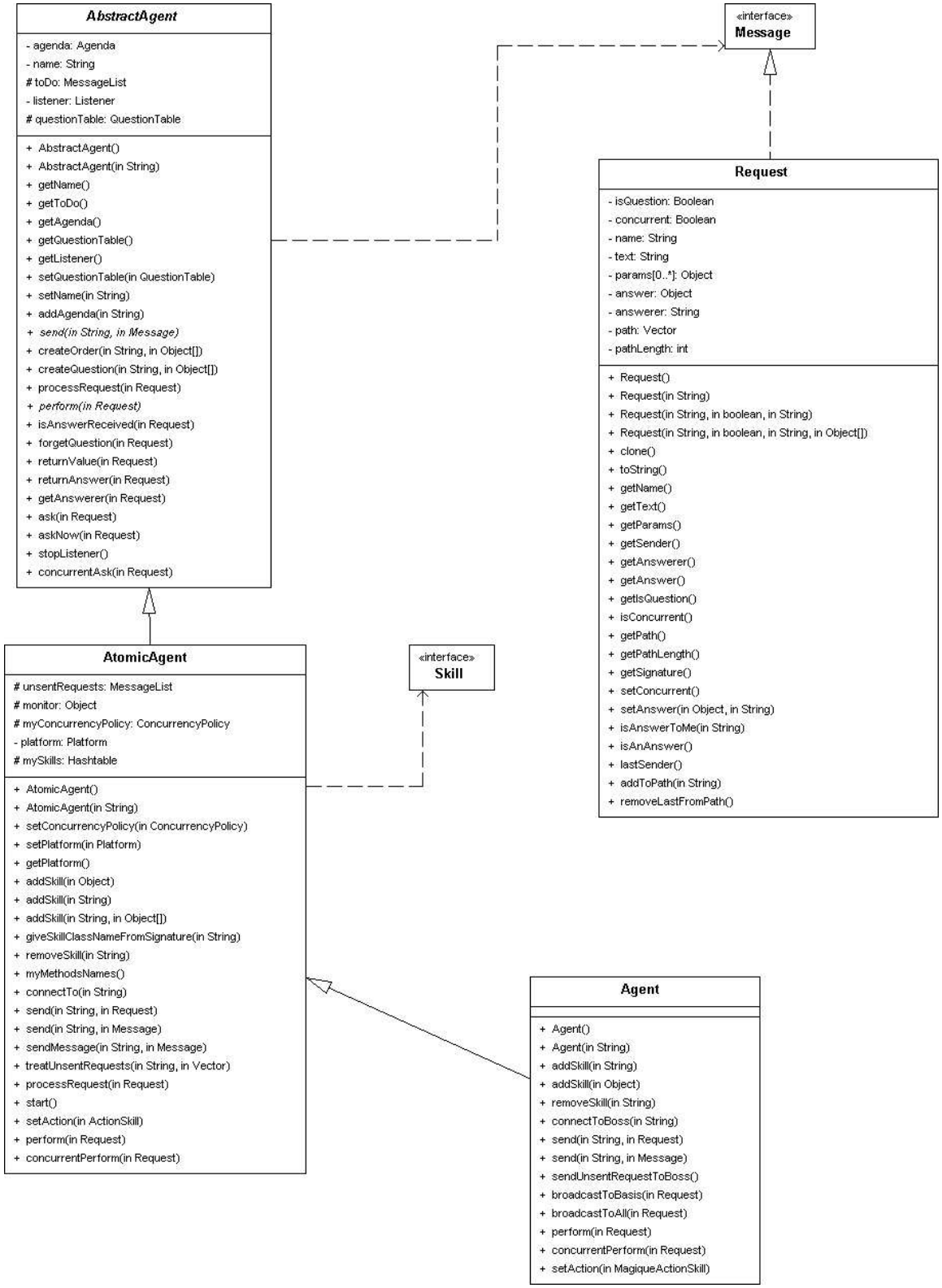

FIG. 4.1 – *Les classes et interfaces principales de* MAGIQUE

la classe `Agent` qui aurait pu être appelée `MagiqueAgent`, puisque c'est elle qui définit les compétences initiales de l'agent au travers de la méthode `initBasicSkills()` :

```
protected void initBasicSkills() throws SkillAlreadyAcquiredException
{
    // System skills (from fr.lifl.magique.skill.system)
    this.addSkill(new ConnectionSkill(this));
    this.addSkill(new AddSkillSkill(this));
    this.addSkill(new LearnSkill(this));
    this.addSkill(new DisplaySkill());
    this.addSkill(new KillSkill(this));

    // Organizational skills (from fr.lifl.magique.skill.magique)
    this.addSkill(new BossTeamSkill(this));
    this.addSkill(new ConnectionToBossSkill(this));
}
```

Le premier groupe de compétence se rapporte aux fonctions *système* : on retrouve les deux compétences du modèle d'agent minimal, la compétence de communication (`ConnectionSkill`) et d'acquisition de compétence (`AddSkillSkill`, la compétence `LearnSkill` n'étant qu'une version *applicative* de la compétence *système* `AddSkillSkill`), ainsi que la compétence d'affichage (`DisplaySkill`) et la compétence de destruction d'un agent (`KillSkill`).

Le deuxième groupe de compétence se rapporte à la gestion de l'organisation et plus spécifiquement dans le cas de MAGIQUE à la hiérarchie (se reporter à la section suivante pour une description du modèle organisationnel).

La dernière classe présentée dans le diagramme de classe de la figure 4.1 est la classe `Request` qui implémente l'interface `Message`. Cette interface n'est qu'un marqueur, et sert a obliger les messages du système à être sérialisables (c'est-à-dire pouvant être transformés en une représentation persistante). La classe `Request` défini la structure des messages échangés dans MAGIQUE, qui peuvent être soit des ordres (`perform`), soit des requêtes (`ask/askNow`). Plus concrétement, l'information contenue dans un message est l'agent à qui il est destiné (dans le cas d'un appel nommé), le nom de la méthode a invoquer, et la liste de ses arguments (transmise sous la forme d'un tableau d'`Object`).

4.1.2 Le modèle organisationnel : Magique

L'organisation des agents peut être amenée à évoluer dynamiquement si une réorganisation de la structure du SMA se justifie en cours d'utilisation. Dans MAGIQUE, cette dynamique opère à plusieurs niveaux :

- *individuel* : un agent peut acquérir ou oublier des compétences. Dans MAGIQUE, cela se traduit par un échange effectif de compétences entre agents, éventuellement distants, à l'initiative des agents eux-mêmes ;
- *relationnel* : Des liens d'accointance peuvent être créés lorsque des relations favorisées apparaissent entre deux agents de la hiérarchie. Cela a pour effet de supprimer

les communications récurrentes le long de l'arborescence, puisque dès lors, la communication entre les agents est directe. La décision de la création de tels liens est une prérogative des agents eux-mêmes.

Couplés avec le mécanisme de délégation, cette création dynamique de relations de communication contribue à rendre *non déterministe* le fonctionnement d'une application SMA : deux exécutions successives ne produiront pas nécessairement la même structure de communication et les compétences ne seront pas nécessairement réalisées par les mêmes agents.

- *organisationnel/architectural* : des agents peuvent être créés ou détruits afin d'adapter le système à certaines contraintes. Deux exemples parmi d'autres :
  - un agent peut se voir submerger par un afflux de requêtes pour l'une de ses compétences ; il peut alors décider de créer une équipe d'agents qui auront cette compétence vers laquelle il pourra rediriger tout ou partie des requêtes qui lui parviennent,
  - un agent doit pouvoir quitter temporairement le système et retrouver sa place ultérieurement, les messages qui lui sont destinés doivent alors avoir été mis en attente. Cela peut être essentiel pour des agents situés sur des ordinateurs portables (ou des téléphones ou tout appareil ne fournissant pas un accès continu au réseau).

Dans MAGIQUE, la structure organisationnelle de base est une structure hiérarchique. L'organisation hiérarchique des agents permet un routage par défaut des messages qui facilite le développement des agents. Ce type de structure est fréquent dans les organisations humaines, mais aussi dans de nombreux logiciels informatiques tels que le mécanisme de DNS (*Domain Name Server*), les langages orientés objets avec la notion d'arbre d'héritage ou encore les systèmes de fichiers arborescents.

Une hiérarchie est un arbre dont la racine est un agent et dont les fils sont, quand il y en a, également des hiérarchies. Les agents feuille sont appelés "*spécialistes*" et les autres "*superviseurs*", ceux-ci doivent être capables (i.e. avoir la compétence leur permettant) de gérer leur *équipe* d'agents (la sous-hiérarchie) dont ils sont la racine (figure 4.2).

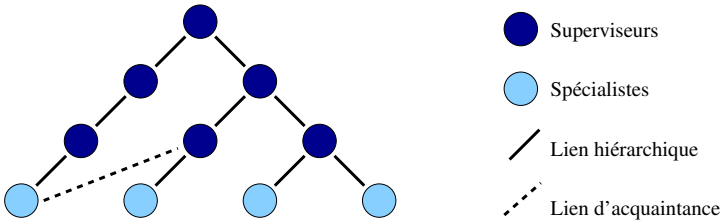

FIG. 4.2 – *La structure hiérarchique choisie*

Une hiérarchie représente la structure initiale des accointances du SMA et induit le support par défaut des communications entre les agents. Un lien hiérarchique représente donc l'existence d'un lien de communication entre ces agents, et lorsque deux agents d'une même structure hiérarchique communiquent, le chemin emprunté par les messages qu'ils

s'échangent suit les relations d'accointances et colle, par défaut, à la hiérarchie. Cette communication est alors essentiellement verticale.

Avec uniquement ces communications, le modèle serait probablement trop rigide. C'est pourquoi le modèle MAGIQUE prévoit la possibilité de créer des liens directs (i.e. en dehors de la hiérarchie) entre deux agents, appelés liens d'accointance. L'idée visée est cependant la suivante : après quelques temps d'évolution du système, si il apparaît des requêtes privilégiées (fréquentes) entre deux agents pour une compétence, la décision peut être prise de créer dynamiquement un lien d'accointance entre ces deux agents pour la compétence concernée. L'intérêt est évidemment de diminuer les communications et de mettre en évidence des liens "naturels" d'interaction entre les agents.

Ainsi, dans le modèle MAGIQUE, il existe une organisation de communication par défaut : la hiérarchie. Mais cette structure est destinée à évoluer en accord avec le comportement dynamique du système multi-agents en favorisant les relations les plus fréquentes. Il en résulte, qu'après un certain temps, la topologie du réseau d'accointances sera celle d'un graphe et non plus un arbre. L'intérêt est que, la structuration étant dynamique et adaptative, le concepteur d'un SMA peut se reposer en partie sur ce mécanisme lors de la conception de l'organisation de son SMA. De plus, parce qu'elle offre un réseau d'accointances par défaut, la structure hiérarchique permet de disposer d'un mécanisme automatique de délégation de requêtes.

4.1.3 Un mécanisme de gestion et de délégation de requêtes

La réalisation d'une tâche par un agent requiert l'exploitation d'un certain nombre de compétences auxquelles il devra faire appel, qu'il les possède ou pas. Dans les deux cas, le mode d'invocation de ces compétences se réalise de manière identique. La délégation éventuelle de la réalisation à un autre agent est transparente pour lui grâce à la structure hiérarchique. Le principe en est le suivant :

- l'agent "possède" la compétence, il "l'invoque" directement,
- l'agent ne "possède" pas la compétence, plusieurs cas de figure possibles :
  - il a une accointance particulière pour cette compétence, il lui demande alors de la réaliser pour lui,
  - sinon, il est superviseur et un membre de sa hiérarchie possède cette compétence, il transmet (récursivement *via* la hiérarchie) la délégation de réalisation de cette compétence à qui de droit,
  - sinon, il demande à son superviseur de trouver quelqu'un de compétent et celui-ci ré-applique ce même mécanisme récursivement.

Un des avantages de cette délégation transparente est d'accroître la fiabilité du système : l'agent qui exécutera la compétence étant indifférent pour l'invocateur, il peut changer entre deux "appels" à la même compétence (qu'il ait disparu du système suite à une panne ou qu'il soit surchargé ou...) (cf. Figure 4.3). Un autre avantage apparaît lors de la phase de conception des agents et des compétences. Dans la mesure où la recherche d'un agent compétent est automatiquement réalisée par la hiérarchie, quand une requête pour une compétence est écrite, il n'est pas nécessaire de désigner à la conception un agent particulier pour la réalisation d'une compétence donnée. Les destinataires des requêtes

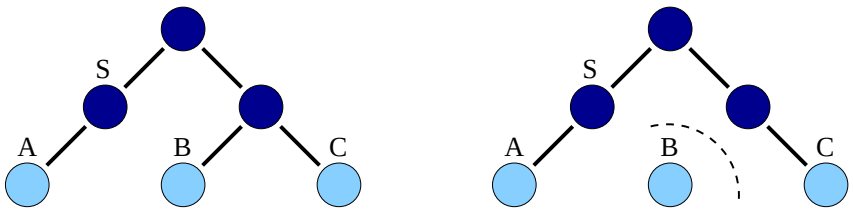

FIG. 4.3 – *L'agent A a besoin d'une compétence, il l'invoque et sa requête est transmise via le superviseur S , qui la transmet à B via la hiérarchie (schéma de gauche) – Le lien avec B a été supprimé, la délégation est automatiquement transmise à C sans que A en ait nécessairement conscience (schéma de droite).*

ne seront donc pas nommés. Il en résulte qu'un même agent pourra être plus facilement réutilisé dans différents contextes (i.e. différentes applications multi-agent) pour autant qu'un agent compétent (peu importe lequel) soit présent. Il en résulte que lorsque vous concevez un SMA avec MAGIQUE, le point important n'est pas tant les agents présents que les compétences présentes.

4.1.4 L'acquisition dynamique de compétence

Nous l'avons dit, la construction et l'évolution d'agents dans MAGIQUE se base sur l'acquisition (et l'oubli) dynamique de compétences. Avec cette possibilité d'évolution dynamique d'un agent, il n'est plus possible d'utiliser le terme de "classe" d'agents. Même si pour différents agents vous partez d'une base commune, dans la mesure où ils peuvent (et vont probablement) recevoir une éducation différente due à leurs "expériences" individuelles, ils vont bientôt diverger et il sera de ce fait impossible de les considérer comme appartenant à une même "classe". Cette notion n'a définitivement plus de sens dans ce contexte, ce qui constitue une différence fondamentale entre la programmation orientée agents et la programmation orientée objets.

4.1.5 Environnement de développement

Un environnement graphique permettant la construction de la structure hiérarchique, la création des agents et l'enseignement de leurs compétences existe (cf. figure 4.4). Vous pourrez le récupérer pour le tester sur le site de MAGIQUE. Cet environnement permet également la distribution automatique des agents sur le réseau et fournit une console d'administration distante de ces agents.

4.1.6 Avantages et limitations de Magique

Comme nous l'avons vu, MAGIQUE est un intéressant *framework* pour le développement de systèmes multi-agents. La simplicité du modèle et de l'API favorise une transition aisée pour le développeur habitué à la conception orientée objets. En outre, les mécanismes d'appel à la cantonnade et de délégation de service, ainsi que la gestion transparente de l'aspect distribué, sont autant de caractéristiques intéressantes offertes par le système.

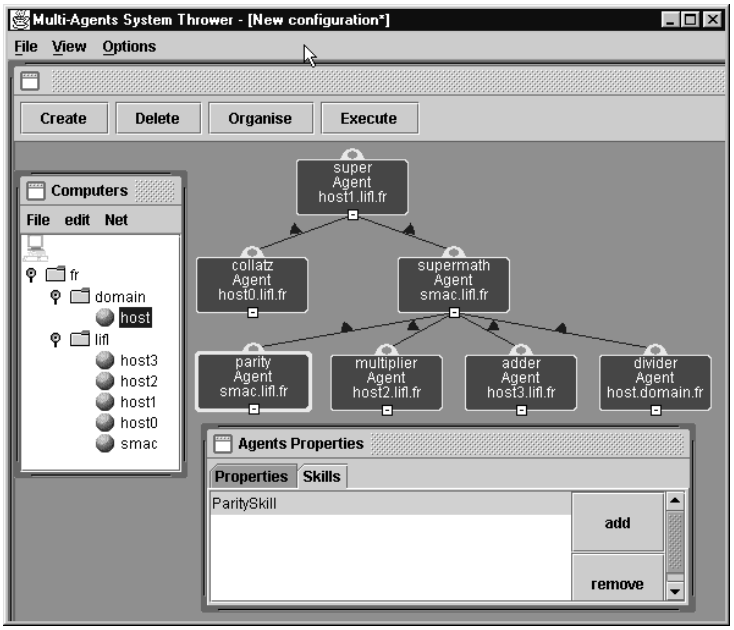

FIG. 4.4 – *Le SMA de collatz créé avec l'environnement de conception graphique*

Cependant, nous avons vu que l'échange de compétences pose certains problèmes au niveau de l'utilisation de librairies tierces. Le passage à un modèle de composant pour l'implémentation des compétences est inévitable pour résoudre ces limitations lors du déploiement. De plus, l'absence de représentation des dépendances entre les compétences est une faiblesse importante. Ces interactions se trouvent enfouies dans le code des différentes compétences, et un agent ne peut savoir lorsqu'il en acquiert une nouvelle, quelles sont les compétences nécessaires à son exécution.

D'autre part, l'organisation dans MAGIQUE est hiérarchique et n'est pas réifiée. Nous avions pendant une période, imaginé la possibilité d'introduire des hiérarchies multiples. Le principe était de proposer au concepteur différentes plans de communications en fonction des besoins applicatifs. Ainsi, tous les agents du système dépendraient de l'agent plateforme sur le plan de la communication *système*. Le concepteur pourrait ensuite définir ses propres plans de communication et ainsi distinguer les différents réseaux de dépendances.

Ce sont l'ensemble de ces raisons qui nous ont mené à développer notre modèle de spécifications exécutables de protocole d'interaction et notre nouvelle plateforme.

4.2 G : une plateforme multi-agents pour Rio

Comme nous l'avons vu dans le premier chapitre, la notion de plateforme bien qu'inhérente aux systèmes réels n'est que bien peu souvent étudiée. Les aspects orthogonaux tels que la fiabilité, la montée en charge ou une conception logicielle facilitant l'évolution et la maintenance de tels systèmes, sont souvent sous estimés, voir ignorés. Tout au long du développement de notre plateforme, nous nous sommes attachés à trouver un

compromis entre les aspects fonctionnels du système et les principes mis en avant par notre méthodologie. Pour cela, nous avons étudié différentes technologies, et nous nous sommes basés sur celles qui nous semble répondre aux mieux à ces contraintes. Dans cette section, nous allons présenter l'architecture et l'implémentation de notre plateforme, ainsi que les choix qui l'ont guidée. Puis, nous étudierons les perspectives de développement en terme d'outils tels qu'un environnement de développement, de déploiement et de supervision.

4.2.1 Conception et implémentation de notre plateforme

La plateforme G, est un environnement minimum permettant de créer, de faire évoluer et de détruire des agents. Ces agents sont une implémentation de notre modèle d'agent présenté dans le chapitre 2. Voulant à terme participer à des réseaux de plateformes hétérogènes telles qu'AgentCities[53], notre plateforme repose sur certaines spécifications de laFipa[54]. Enfin, tout au long des développements nous avons veillé à utiliser le plus de librairies ou d'outils librement distribuables pour pouvoir facilement diffuser notre plateforme.

Le premier choix que nous avons dû effectuer fut celui du langage d'implémentation. Nous avons choisi la plateforme Java, comme pour Magique, pour l'ensemble des raisons suivantes :

- langage orienté objets,
- la richesse des Api de base,
- la richesse et la disponibilité de projets tiers,
- l'indépendance vis-à-vis de l'architecture d'exécution,
- la disponibilité de la plateforme du téléphone portable (J2me) aux serveurs (J2ee),
- le support du monde industriel (particulièrement Ibm avec Jikes),
- le support de la communauté Open Source (tout du moins en terme de projets développés en Java).

Les services de la plateforme

G repose sur la spécification d'architecture abstraite de la Fipa[55]. Cette spécification définie les éléments architecturaux et leurs relations, ainsi que des règles de conformité pour les agents et les plateformes.

Plus précisément, ne désirant pas ré-implémenter l'ensemble de la couche de communication (transport et encodage des messages), et des services de base (nommage, pages jaunes/blanches), nous avons utilisé l'Api Jas, *Java Agent Services*[56]. Cette Api est développée au sein du Jcp, *Java Community Process*[57], et propose une Api Java

53. AgentCities est un projet européen visant à mettre en place un réseau de plateformes conformes aux spécifications de la Fipa proposant des services en ligne : `http://www.agentcities.org`

54. Plus particulièrement les spécifications en relation avec l'architecture abstraite (`SC00001`) proposée par la Fipa dans `http://www.fipa.org/specs/fipa00001/`

55. Spécification SC00001L, disponible à l'adresse : `http://www.fipa.org/specs/fipa00001/SC00001L.pdf`

56. Plus d'informations sur Jas : `http://www.java-agent.org`

57. Plus d'informations sur le Jcp : `http://www.jcp.org`

implémentant l'architecture abstraite de la FIPA. Bien que cette API ne soit pas encore une extension standard de la plateforme JAVA, elle est au dernier stade (celui de l'évaluation publique) et est librement accessible et utilisable (pour plus d'informations se reporter à la sous-section 1.3.3 page 49).

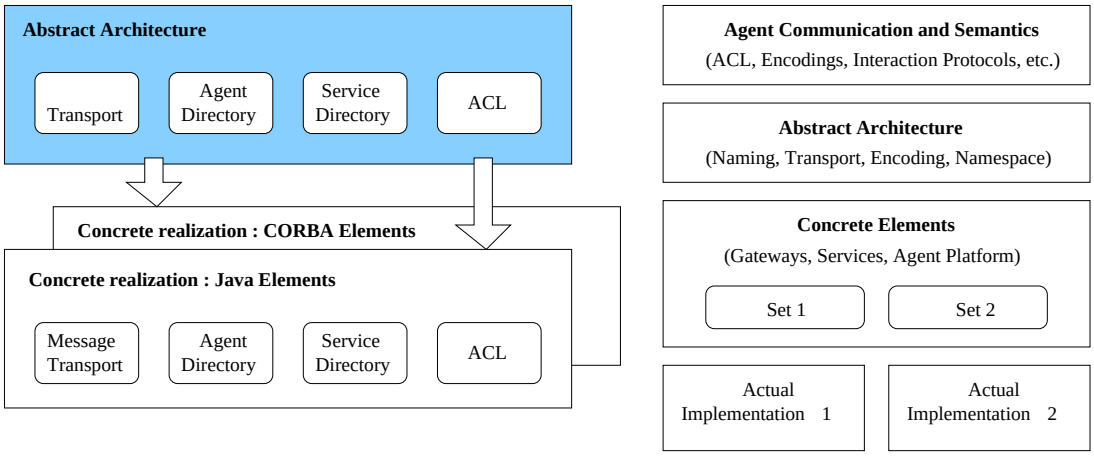

FIG. 4.5 – *Les différents éléments de l'architecture abstraite spécifiée par la* FIPA

L'architecture abstraite (figure 4.5) repose sur un ensemble de services comme le service de nommage (création d'identifiants uniques), de transport (routage des messages), et de pages blanches/jaunes (le *directory facilitator*). D'autre part, un service de gestion de services permet d'ajouter des services supplémentaires à la plateforme. Nous avons donc utilisé ce service pour ajouter un service de base de données (figure 4.6). Ce service est utilisé pour gérer la persistance des messages reçus ou émis par la plateforme, mais aussi pour stocker et manipuler les informations relatives à la gestion de la plateforme.

Ces services sont uniquement accessibles et utilisables par la plateforme. Les agents, qu'ils appartiennent à la plateforme ou non, n'y ont pas accès directement, et doivent effectuer leurs requêtes auprès de l'agent plateforme (PLATFORM AGENT de la figure 4.6). L'implémentation actuelle ne contient pas encore les conteneurs pour les composants J2EE et CCM. Nous avons préféré nous concentrer sur l'utilisation des composants OSGI, car leur modèle de gestion des dépendances est beaucoup plus avancé. Pour nous abstraire du modèle de composant utilisé, nous avons utilisé WSIF, *Web Service Invocation Framework* qui permet d'invoquer un service via une description WSDL, quelque soit l'implémentation de ce service.

Les agents systèmes

Les agents systèmes sont ceux qui sont démarrés en même temps que la plateforme et qui encapsulent des fonctionnalités de base permettant de gérer une plateforme. Pour l'instant, nous n'utilisons qu'un seul agent système : l'agent plateforme ou *PlatformAgent*. Il est envisageable d'ajouter d'autres agents système tel qu'un agent de gestion de la sécurité (authentification et autorisation), ou un agent d'administration et de déverminage (fournissant un point d'entrée dans la plateforme).

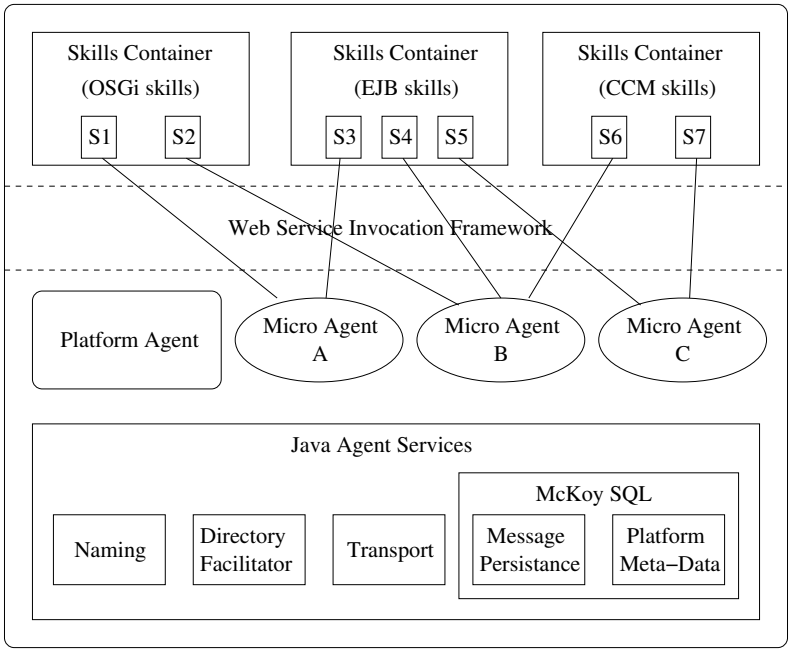

FIG. 4.6 – *L'architecture de notre plateforme*

L'agent plateforme gère les fonctionnalités essentielles de la plateforme : son arrêt, et la gestion des agents (création, enregistrement dans le *directory facilitator*). Du point de vue de l'implémentation, deux choix se présentent : soit cet agent est *dédié* à la plateforme, soit c'est un agent utilisant le même modèle que les agents applicatifs. Dans l'implémentation actuelle, c'est un agent dédié, c'est-à-dire qu'il ne peut pas modifier son comportement dynamiquement. Cette solution limite l'évolutivité dynamique de la plateforme, mais garantit en contrepartie sa stabilité et son incorruptibilité.

Les agents applicatifs

Les agents applicatifs suivent notre modèle générique d'agent. Ils sont donc constitués d'un ensemble de compétences réalisant leur modèle d'agent spécifique et leur comportement. La possibilité qu'ont les agents d'acquérir de nouvelles compétences, pose des problèmes de sécurité et de fiabilité similaires aux problématiques qui ont été identifiés dans le domaine des agents mobiles. Il est en effet difficile de garantir une isolation totale entre le code de la plateforme et le code mobile. Le langage JAVA fournit un premier niveau de sécurité grâce au vérificateur de *bytecode*, qui garantit la conformité structurelle du *bytecode* avant de le charger. Cependant, des attaques de type "harassement de ressources", tels que la saturation de la mémoire ou du processeur, sont toujours possibles.

D'intéressants travaux ont proposé des mécanismes de filtrage[36], qui autorisent une plateforme à modifier dynamiquement le code mobile pour éviter ce genre de situation. Ces mécanismes reposent sur une liste d'opérations surveillées, auxquelles sont associées des règles de transformation. Ainsi, lorsqu'une création de JAVA.LANG.THREAD, ou d'une sous-classe est repérée dans le code, une règle indique qu'il faut remplacer par la classe PACKAGE.MONTHREADSÉCURISÉ qui borne le nombre de créations possible. L'inconvénient

de ce genre d'approche est qu'il est difficile de déterminer a priori les ressources qui vont être attaquées, et qu'il est difficile d'avoir des règles générales : certaines compétences pouvant nécessiter la création d'un nombre important de processus (un gestionnaire de base de données créera typiquement un *pool de threads* ...).

Une approche différente consiste à modifier l'architecture des machines virtuelles JAVA, pour qu'elles puissent garantir l'exécution concurrente d'applications, tout en garantissant leur étanchéité. C'est le but poursuivi par les auteurs de la *Java Specification Request 121* intitulée *Application Isolation API Specification*. Cette API permettra le partage de ressources systèmes au sein d'une même JVM, tout en garantissant différents niveaux d'intéropérabilité : partage de *bytecode*, de code natif, d'instances ... Malheureusement, cette spécification n'est devenue publique que depuis le 18 mars 2003, et ne sera disponible que dans une prochaine version de la JVM de Sun (cette fonctionnalité devait être intégrée à la prochaine version 1.5, mais elle a finalement été rejetée).

Ces problèmes nous ont poussés à séparer les environnements d'exécution des fonctionnalités fondamentales de la plateforme, des fonctionnalités applicatives fournies par les compétences des agents. Cela signifie qu'en attendant la disponibilité du *JSR 121*, il est nécessaire d'utiliser une JVM pour la plateforme, et une ou plusieurs autre JVM pour les conteneurs de compétences.

4.2.2 Implémentation des compétences avec OSGi

L'implémentation d'une compétence peut s'effectuer avec différentes technologies. Pour des problèmes de réutilisabilité et d'*acceptation industrielle*, nous pensons qu'il est nécessaire d'associer l'implémentation des compétences à un modèle de composants logiciels. Cependant, comme il existe un grand nombre d'initiatives dans ce domaine, et que ces propositions s'adressent à différents domaines d'applications, notre plateforme n'impose pas l'utilisation d'un type particulier de composants. Pour cela, nous avons défini les fonctionnalités nécessaires que doit supporter un conteneur de composants pour pouvoir interagir avec notre plateforme.

Parmi les différents modèles de composants disponibles sur la plateforme JAVA, nous nous sommes plus particulièrement intéressés à trois d'entre eux : CCM, EJB et OSGI. Les deux premiers sont des modèles de composants pour les applications réparties côté serveur. Comme nous l'avons vu, CCM est la proposition de l'OMG comme modèle de composant pour les bus à objets répartis, tandis que les EJB ont été définis par SUN et d'autres vendeurs pour la plateforme J2EE qui est principalement utilisée dans les modèles trois-tiers. Nous avons retenu le modèle OSGI comme pour l'implémentation de référence, car la gestion des dépendances structurelles et fonctionnelles est particulièrement fine.

Les conteneurs de compétences

La plateforme doit pouvoir instancier et détruire des composants dans les différents conteneurs. La difficulté est d'abstraire ces opérations pour qu'elles puissent être effectuées par les conteneur, sans que la plateforme ne manipule d'informations spécifiques liées au modèle de composant. Pour cela, nous avons défini les messages du protocole entre la plateforme et les conteneurs, qui s'appuient sur le référencement des ressources via des

Uri.

```
<add>
  <skill instanceName="monBrowser"
           uri="http://skills.lifl.fr/applicative/browser/v1.0.1.osgi"/>
</add>

<instance name="monBrowser" action="start" />
<instance name="monBrowser" action="suspend" />
<instance name="monBrowser" action="stop"/>

<remove>
  <skill instanceName="monBrowser"/>
</remove>

<list>
  <instances/>
  <services/>
</list>
```

Ainsi, dans l'exemple ci-dessus, la plateforme demande à un conteneur de composant OSGi d'ajouter une nouvelle instance d'un composant fournissant un navigateur. Le conteneur charge le fichier "v1.0.1.osgi" qui contient toutes les informations et les données permettant de charger et d'instancier un composant OSGi. De même, pour d'autres types de composants, tels que *Corba Component Model* (Ccm) dans le monde Corba, ou les *Enterprise Java Beans* (Ejb) de la plateforme J2ee, l'ajout de compétence serait similaire, seule l'Uri différerait, ainsi que la structure et les informations contenues dans l'archive de compétence.

Le protocole prévoit aussi une gestion du cycle de vie des composants, au travers des actions *start*, *suspend* et *stop*. Il est aussi possible de demander la liste des instances, ainsi que la liste des services disponibles. Nous allons voir dans le paragraphe suivant que nous nous sommes inspirés du modèle OSGi en ce qui concerne la gestion des composants.

Le modèle de composants OSGi

Les spécifications OSGi, *Open Service Gateway*, visent à définir une infrastructure de gestion de services embarqués pouvant être administrés à distance. La figure 4.7 précise le cadre dans lequel se place le *framework* OSGi, ainsi que les technologies qui peuvent lui être associées. Les motivations derrière cette spécification sont le déploiement dynamique d'applications sans interruption de la passerelle, la programmation orientée services et le déploiement dans des environnements à mémoire restreinte. Pour cela, trois notions sont mises en avant : la notion de conteneur, la notion de *bundle* et la notion de service. Le conteneur est l'environnement dans lequel il est possible d'installer des *bundles*, qui peuvent ensuite rechercher et enregistrer des services. Le *bundle* correspond à une archive (plus pragmatiquement une archive Java, un Jar), qui contient l'ensemble des ressources nécessaires à l'exécution du ou des services.

Le *bundle* (figure 4.8) est l'unité de base de l'architecture OSGi. Il est composé d'un fichier `Manifest` contenant un ensemble de méta-données, de ressources (fichiers binaires), de classes Java implémentant les services. Le fichier `Manifest` contient des méta-données concernant le *bundle*, plus particulièrement les services et les paquetages nécessaires à l'instanciation de ce *bundle*, ainsi que les services et les paquetages que ce *bundle* exporte.

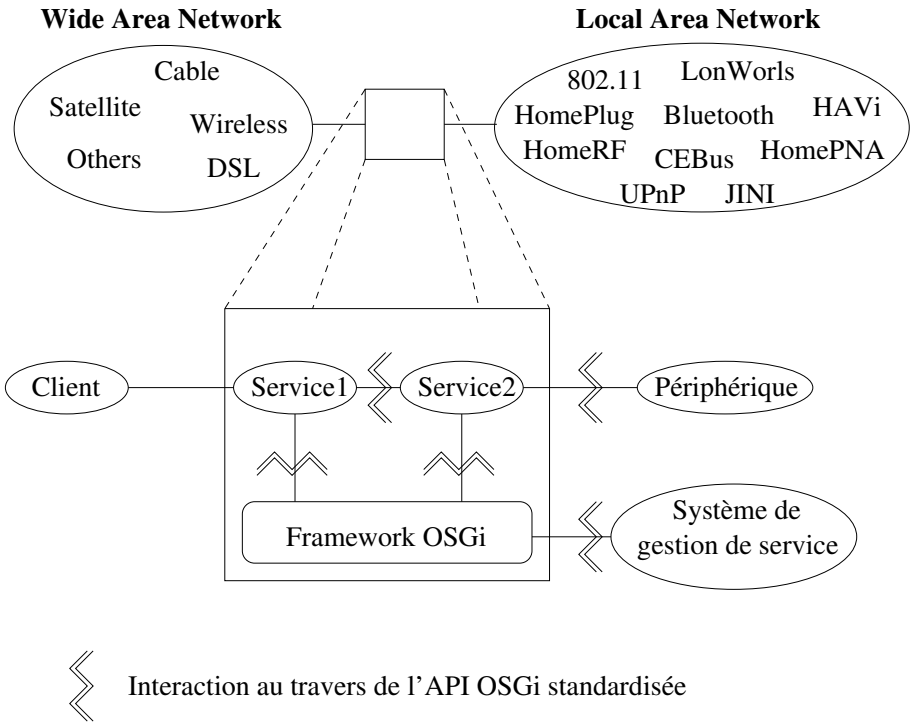

FIG. 4.7 – *Le positionnement du modèle* OSGI

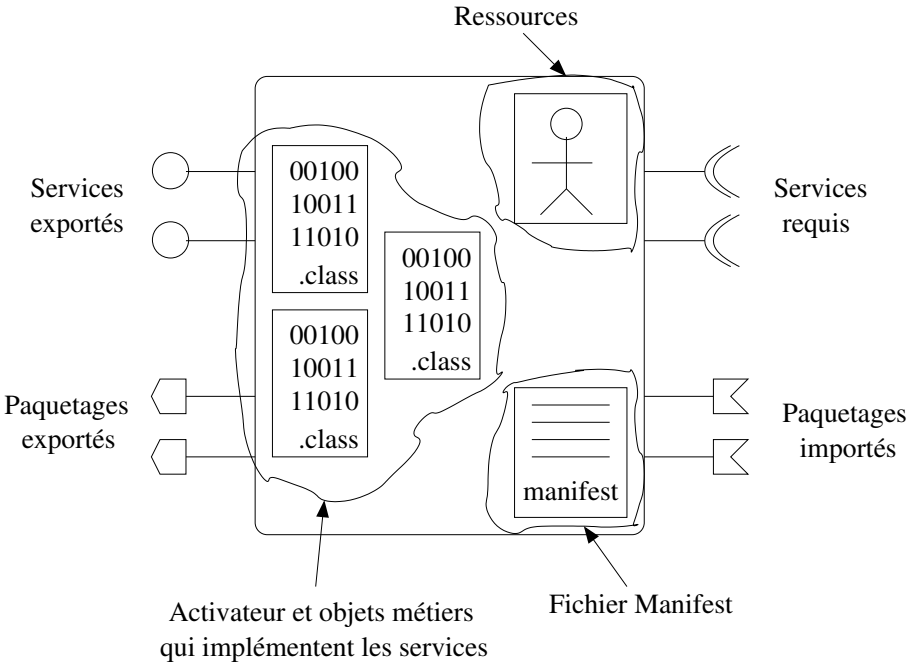

FIG. 4.8 – *Eléments constituants un* bundle OSGI

Le cycle de vie d'un *bundle* est représenté dans la figure 4.9. Lorsqu'un *bundle* est ajouté au conteneur, ce dernier charge le fichier `Manifest` et vérifie les dépendances, à la fois du point de vue des paquetages nécessaires (dépendance structurelle) et des services (dépendance fonctionnelle). Si les dépendances sont satisfaites, le *bundle* passe alors dans l'état `Resolved`, il peut donc être démarré. Une fois démarré, et si la classe invoquée lors de l'instanciation du *bundle*, qui doit nécessairement implémenter l'interface `Activator`, ne retourne pas d'erreur, le *bundle* passe dans l'état `Active`. Les services sont alors disponibles dans le *framework*.

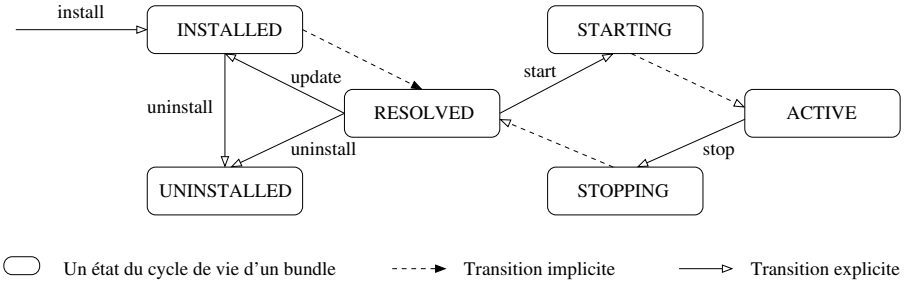

Fig. 4.9 – *Le cycle de vie d'un* bundle OSGi

La gestion des dépendances avec OSGi

Les problèmes liés à la gestion des dépendances se retrouvent à plusieurs niveaux : entre les compétences d'un agent, entre les composants d'une compétence. Les dépendances entre compétences sont fonctionnelles, c'est-à-dire qu'une compétence ne pourra fonctionner que si la compétence dont elle dépend est déjà présente. Par exemple, une compétence de supervisation d'un processus quelconque pourrait être dépendante d'une compétence de visualisation de pages Html. Ces dépendances peuvent se décomposer en deux classes : celles qui sont nécessaires, c'est-à-dire que la compétence ne peut être instanciée tant que la compétence dont elle dépend n'est pas présente, ou optionnelles, lorsque la compétence ne fournit pas une fonctionnalité indispensable. Ce deuxième cas peut être vu comme un mécanisme de gestion de *plugin*. Une autre illustration d'une compétence optionnelle pouvant être utilisée par tous les composants du conteneur : la compétence de gestion de *log*.

L'exemple ci-dessous illustre les deux étapes fondamentales de définition d'un *bundle* OSGi. Dans un premier temps, il faut écrire la classe qui implémentera l'interface `BundleActivator`, et qui enregistrera les services proposés par le *bundle*.

```
package foobar.example1;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class MyActivator implements BundleActivator
{
    public void start(BundleContext context)
    {
        System.out.println("Registering MyService into the framework.");
Properties props = new Properties();
        props.put("Language", "English");
```

```
        context.registerService(MyService.class.getName(),
    new MyServiceImpl(), props);
    }

    public void stop(BundleContext context)
    {
        System.out.println("Stopping MyService ...");
// Unregistration automatically done by the framework !
    }
}
```

Puis, dans un second temps, il faut écrire le fichier `manifest` qui ajoute les méta-données décrivant le *bundle* et plus particluièrement les dépendances statiques (importation et exportation de paquetage) ainsi que les dépendances dynamiques (publication et utilation de services). Ainsi, le `manifest` ci-dessous décrit un *bundle* dépendant des paquetage `org.appache.jakarta.commons` et `foobar.example2` et exportant le paquetage `foobar.example1`. Ainsi, le framework ne pourra installer ce *bundle* qu'à la condition que les deux paquetages soient présents. Pour les déclarations de dépendances dynamiques, avec le service `OtherService`, l'information est purement informative dans la norme OSGi. Ainsi, si le service `OtherService` n'est pas présent lors de l'appel à la méthode `start` de `MyActivator`, c'est au développeur de prévoir un mécanisme d'abonnement auprès du *container* pour être notifié lors de l'enregistrement du service nécessaire.

```
Bundle-Activator: foobar.example.MyActivator
Import-Package: org.appache.jakarta.commons, foobar.example2
Import-Service: foobar.example2.OtherService
Export-Package: foobar.example1
Export-Service: foobar.example1.MyService
Bundle-Name: Useless bundle
Bundle-Description: A bundle that displays messages at startup and shutdown
Bundle-Vendor: SMAC Team
Bundle-Version: 1.0.0
```

Bien que la gestion des dépendances définie par la spécification OSGi soit très intéressante, car elle permet de gérer les dépendances à une granularité très fine, l'écriture de la classe implémentant la classe `Activator` n'est pas très aisée dans le cadre d'environnements dynamiques, c'est-à-dire lorsque des services peuvent joindre ou quitter le système dynamiquement. En effet, pour gérer cette dynamicité, il est nécessaire de s'abonner aux événements de types `ServiceEvent` et de s'abonner/désabonner lorsque le service dont on dépend apparait/disparait du *framework*.

C'est pourquoi, nous nous sommes basés sur le *bundle* `ServiceBinder` de Humberto Cervantes et Richard S. Hall[58]. Ce service facilite la gestion des dépendances en externalisant leur définition dans un fichier de description XML (figure 4.10). Ce fichier permet de déclarer les interfaces de service nécessaires à l'instanciation du *bundle*, ainsi que les services qu'il exporte. Dans cet exemple[59], le service `SpellChekService` est déclaré, et il nécessite la présence d'au moins une instance du service `DictionnaryService`. De plus, il est précisé que cette dépendance est dynamique, et qu'il est donc possible que des services `DictionnaryService` joignent ou quittent dynamiquement le système. Si cela se produit, les méthodes `addDictionnary` et `removeDictionnary`

58. Plus d'informations sur le `ServiceBinder` : `http://gravity.sourceforge.net/servicebinder/`
59. Cet exemple est extrait du tutoriel du `ServiceBinder`.

```
<?xml version="1.0" encoding="UTF-8"?>
<bundle>
  <instance class="tutorial.example7.SpellCheckServiceImpl">
    <service interface="tutorial.example5.service.SpellCheckService"/>
    <property name="version" value="1.0" type="string"/>
    <requires
      service="tutorial.example2.service.DictionaryService"
      filter="(Language=*)"
      cardinality="1..n"
      policy="dynamic"
      bind-method="addDictionary"
      unbind-method="removeDictionary"
    />
  </instance>
</bundle>
```

Fig. 4.10 – *Le fichier de description des dépendances de* ***ServiceBinder***

seront automatiquement invoquées. Ce mécanisme de gestion des dépendances par le service `ServiceBinder` facilite l'écriture des *bundles* et leur modularisation.

En ce qui concerne l'implémentation, nous utilisons le conteneur Oscar qui est (presque) conforme aux spécifications OSGi 3.0 et qui a l'avantage d'être *open source*. Théoriquement, le conteneur doit pouvoir être changé sans entraîner de grosses modifications dans le système. Nous prévoyons de vérifier cela en proposant la possibilité d'utiliser d'autres conteneurs OSGi tels que Java Embedded Server de Sun ou encore Jeffree du groupe ObjectWeb[60].

Actuellement, l'intégration d'Oscar n'est pas finalisée. En effet, avec Wsif nous pouvons facilement invoquer des classes locales ou des Ejb, mais il est nécessaire de développer un connecteur pour pouvoir utiliser les services OSGi. D'autre part, la compétence de gestion des protocoles d'interaction n'a pas encore été testée dans le cas de conversations simultanées.

60. Plus d'informations sur Jeffree : `http://jeffree.objectweb.org/`

Chapitre 5

Applications

> "*Faced with the admitted difficulty of managing the creative process, we are doubling our efforts to do so. Is this because science has failed to deliver, having given us nothing more than nuclear power, penicillin, space travel, genetic engineering, transistors, and superconductors? Or is it because governments everywhere regard as a reproach activities they cannot advantageously control? They felt that way about the marketplace for goods, but trillions of wasted dollars later, they have come to recognize the efficiency of this self-regulating system. Not so, however, with the marketplace for ideas.*"
>
> John C. Polanyi

Ce chapitre présente deux exemples d'applications développées selon la méthodologie RIO. Ces applications ont été développées avec MAGIQUE et nous ont permis de répérer les faiblesses de ce premier *framework* (cf. section 4.1.6). La première application permet d'organiser des conférences ou des cours sur Internet, elle illustre l'intérêt de l'échange de compétences comme média de transfert de rôle. La deuxième application est un framework de calcul distribué destiné aux scientifiques non-informaticiens.

Ce sont ces applications qui nous ont amené à travailler sur l'explicitation des interactions entre rôles, et à développer notre modèle d'interaction. Nous n'avons pas à l'heure actuelle d'applications fonctionnant sur la plateforme G car elle est en cours de développement.

5.1 Une application de salle de conférences virtuelles

Depuis quelques années, de nombreux chercheurs et industriels se sont penchés sur les applications de travail coopératif ([5, 7, 26, 28]). Il est en effet de plus en plus nécessaire d'offrir des outils permettant à différentes personnes éloignées physiquement et qui doivent travailler collectivement, de communiquer, de s'échanger des documents et de présenter leurs informations à l'ensemble d'une équipe. Actuellement, force est de constater que les quelques outils présents sur le marché n'offrent en général qu'une partie de ces possibilités. Ces outils sont difficiles à

appréhender et surtout difficiles à étendre par les développeurs, tant les nouveaux concepts à ajouter sont parfois divers. Chaque application a en effet ses propres spécificités et, au delà de l'aspect traditionnel propre à chaque collecticiel, il est souvent nécessaire d'ajouter de nouvelles fonctionnalités.

L'objectif de cette application est d'offrir à l'utilisateur la possibilité d'échanger des informations, de type diapositive au format HTML (et ce qui va avec : applets, sons, images, hyper-liens, etc.), à l'ensemble des collaborateurs. L'analyse fait apparaître deux types de rôles : le professeur ou *conférencier* et les élèves ou *auditeurs*. La différence sur ces rôles se détermine par la détention par le *conférencier* d'une unique télécommande. Elle permet de passer des diapositives aux auditeurs et elle peut éventuellement être demandée par les auditeurs. Il n'y a donc qu'un seul conférencier à un instant donné, mais le détenteur de ce rôle peut être amené à changer au cours de la conférence pour qu'un auditeur puisse lui aussi passer ses diapositives. A côté de cette fonctionnalité *sine qua non*, un ensemble d'outils de manipulation est fourni (voir [82] pour un autre exemple d'approche du collecticiel avec un système multi-agents).

Le développement de cette application se base sur le système multi-agents MAGIQUE, et illustre la notion d'échange dynamique de compétences[55]. MAGIQUE offre des fonctionnalités importantes de gestion et d'échange de ces compétences, ce qui rend très aisé le passage de la télécommande d'un utilisateur à un autre grâce à cet échange de compétences. Dès lors, les changements de rôles ne sont pas simplement des drapeaux disposés dans le code source, mais bien des changements dynamiques de rôles. Un agent pouvant avoir autant de compétences qu'il le souhaite, il est aisé de rajouter de nouvelles fonctionnalités au système pour qu'il couvre le champ souhaité par l'entreprise. Nous avons par exemple, après avoir réalisé une première version du prototype, ajouté une compétence "*assistant*" à nos *auditeurs* sans que cela n'ait aucun impact sur l'existant. L'assistant est une entité "intelligente", proactive et réactive, qui aide l'utilisateur dans sa gestion du travail coopératif en lui permettant de mieux contribuer à la conférence. L'assistant permet également une modélisation des autres intervenants aux conférences en déterminant leurs champs de compétences.

Le but essentiel n'est donc pas tant de montrer un nouveau collecticiel que de démontrer comment celui-ci, grâce à l'approche multi-agents et à la plate-forme MAGIQUE et sa notion de compétences, est facilement développable, maintenu et extensible. Dans une première partie, nous présenterons l'application et ses différentes fonctionnalités, et notamment l'assistant. Puis, nous décrirons le principe de réalisation d'une telle application à l'aide d'un système multi-agents.

5.1.1 L'application diapo-conférence

L'objectif de cette application de travail coopératif est de fournir un support permettant la tenue d'une conférence entre des intervenants physiquement distribués.

Chacun des intervenants dispose d'un ensemble de ressources documentaires ("*diapositives*"). Dans une première version de cette application, ces ressources sont décrites par des documents HTML. Le choix de ce format de document se justifie facilement, dans la mesure où il offre un support unique pour des documents de formes très différentes : texte, images, animations, dynamicité *via* les applets, etc. Un second argument en sa faveur est la facilité à se procurer des interprètes et visualisateurs pour de tels documents. Cependant, le choix de ce format n'est pas une contrainte forte de notre application. Comme nous le montrerons par la suite, le développement basé sur le framework MAGIQUE et son principe de compétence, permet une évolution facile del'application. Des documents au format XML pourraient ainsi être pris en compte par l'application sans qu'il y ait beaucoup à changer.

L'application doit lui permettre de diffuser ces documents auprès des autres. Il ne peut cependant le faire qu'à la seule condition qu'il soit détenteur de la *télécommande* (unique) associée à l'application. Cette télécommande lui permet de parcourir et de diffuser ses ressources. Mais elle tient également lieu de "pointeur" que le conférencier peut utiliser pour attirer l'attention des autres utilisateurs sur un point précis du document diffusé, ce pointeur étant visible par tous les utilisateurs en temps réel.

On distingue donc deux *rôles* dans cette application : celui de *conférencier* pour celui qui détient la télécommande, et celui d'*auditeur* pour les autres participants. Comme dans une conférence réelle, le conférencier peut à tout moment céder la télécommande à un auditeur et les rôles respectifs tenus par les intervenants évoluent alors dynamiquement. Une attention particulière a été portée au respect de la conformité avec la tenue des conférences réelles.

Les figures 5.1 et 5.2 donnent un aperçu de l'application du point de vue du conférencier et du point de vue d'un auditeur quelconque. On peut distinguer différents outils sur les deux vues : la visionneuse sur la gauche des écrans (on peut remarquer que celle de l'auditeur a bien la même taille que celle du conférencier), le pointeur de souris que l'on peut apercevoir dans la visionneuse du conférencier est visualisée par un point dans la fenêtre de l'auditeur. D'autres outils comme la télécommande (active chez le conférencier et passive chez l'auditeur), les fenêtres d'équipe et de messagerie, ainsi que l'assistant en haut de chaque écran (il fait une annonce dans le fenêtre du conférencier) sont également visibles.

Les fonctionnalités de base

Dans l'application réalisée, les ressources sont désignées par des URL. L'avantage de ce format est évident. Il permet de désigner des documents correspondant à différents supports (texte, image, sons, vidéo, etc.). De plus, il permet de profiter des capacités de navigation liées aux hyper-liens et permet aussi d'obtenir des documents dynamiques grâce aux applets par exemple. Chaque participant possède ses propres ressources et il peut les regrouper et les organiser afin de constituer un exposé structuré.

Afin de permettre la tenue de la conférence, les deux outils indispensables sont la *visionneuse* et la *télécommande*.

La visionneuse Chaque participant dispose d'un support permettant la visualisation des ressources (il s'agit dans notre cas d'un navigateur d'URL).

Le conférencier a le contrôle des interactions de cet outil chez l'ensemble des participants. Les vues dans les différentes visionneuses sont commandées par le chargement d'un document dans sa visionneuse par le conférencier, selon le principe du WYSIWIS[80]. En fait, les différentes visionneuses des intervenants sont totalement synchronisées sur celle du conférencier. Ainsi, le pointeur de sa souris dans cette visionneuse est visualisé chez tous les intervenants, ce qui lui permet de mettre en évidence tel ou tel point de la diapositive (cf. figures 5.1 et 5.2). Un autre exemple : si le conférencier modifie la taille de la visionneuse sur son poste, la taille des visionneuses de chaque auditeur est également modifiée ; il en est de même lors du défilement du document à l'aide des ascenseurs.

La télécommande La seconde fonctionnalité indispensable est bien sûr la télécommande. Elle est active chez le conférencier et inactive chez les auditeurs. A tout moment, un auditeur peut la demander et c'est au conférencier que revient la décision de céder ou non la télécommande. Son rôle premier est, lorsqu'elle est active, de permettre la diffusion à toute l'assemblée d'une diapositive ou d'enchaîner une séquence de diapositives automatiquement. En fait, le conférencier

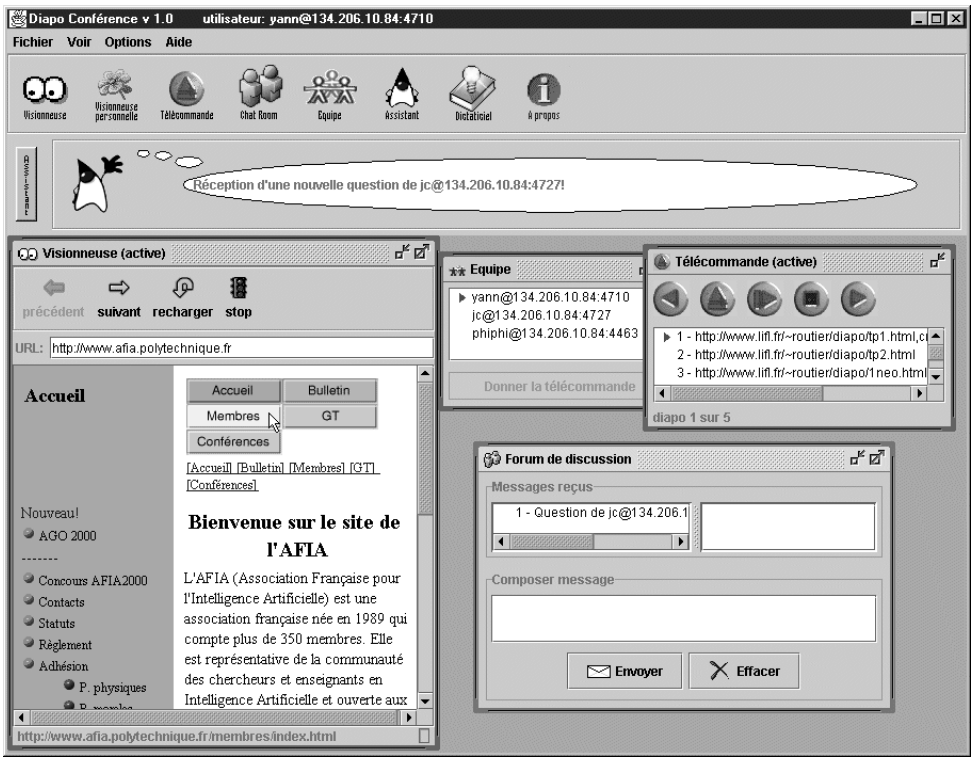

FIG. 5.1 – *L'application du côté* conférencier

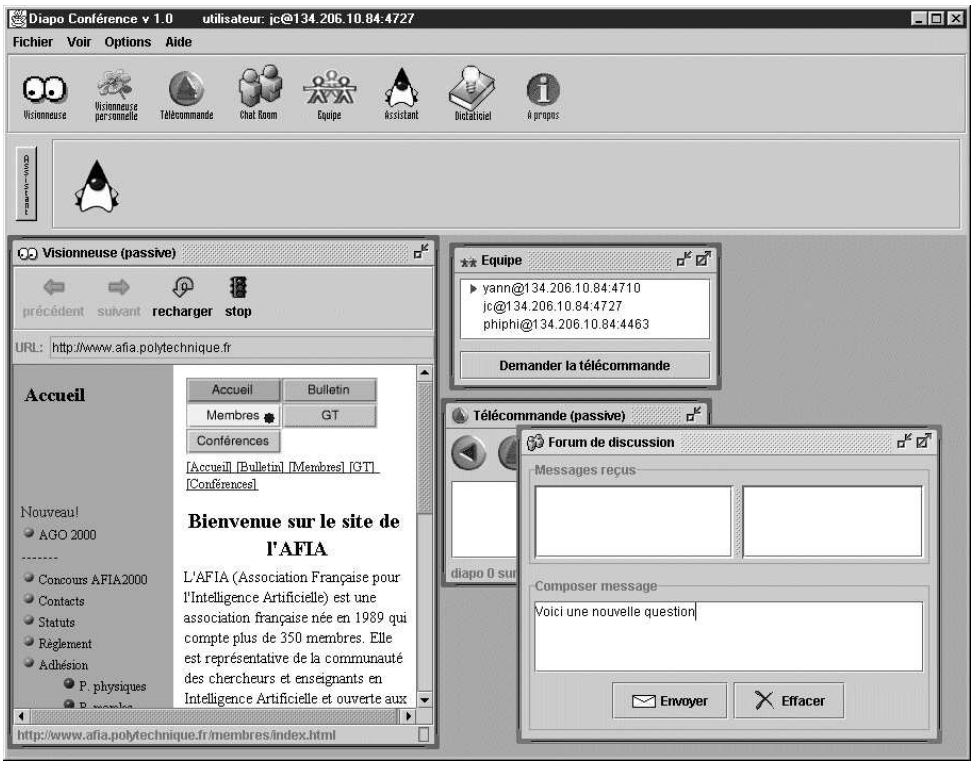

FIG. 5.2 – *L'application du côté* auditeur

transmet aux différents intervenants l'adresse de l'URL du document qu'il souhaite diffuser, les visionneuses de chaque client doivent ensuite télécharger la ressource et l'interpréter.

La télécommande fournit également les outils permettant de rassembler et d'organiser des diapositives en un exposé (cf. figure 5.3). C'est pourquoi la télécommande n'est pas simplement masquée pour le conférencier lorsqu'elle est inactive. Tout participant peut en effet construire, comme lors de la préparation d'un cours, un scénario pour l'enchaînement de ses diapositives. Il peut alors réarranger à tout moment l'ordonnancement des diapositives et réorganiser l'exposé. La possibilité de prévisualiser localement les diapositives facilite cette tâche. Les différents scénario, construits peuvent évidemment être sauvegardés pour une réutilisation ultérieures.

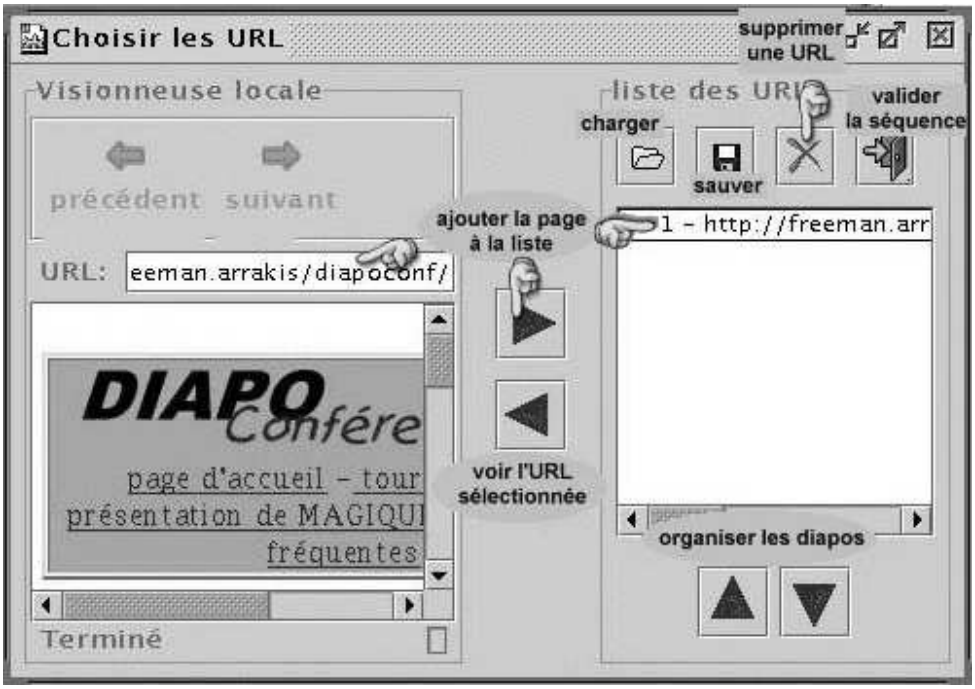

FIG. 5.3 – *Création de séquences de diapositives*

Les outils d'aide à la coopération

En plus des fonctionnalités minimales présentées précédemment, l'application offre plusieurs outils qui favorisent la coopération et la prise de conscience par chacun des autres intervenants. Citons en trois principalement :

- la «visualisation » de l'assistance,
- un assistant,
- un système de messagerie.

Nous allons les présenter brièvement.

La fenêtre des intervenants Le premier outil a pour objectif de souligner auprès de l'utilisateur qu'il "n'est pas seul au monde". La liste des participants est ainsi visualisable dans une fenêtre dans laquelle le conférencier courant est clairement identifié. Cette nécessité de matérialiser une "conscience collective" est un problème bien connu des collecticiels[13].

La demande de télécommande peut se faire depuis cet outil. Les différentes requêtes effectuées par les différents auditeurs sont visualisables, ainsi que leur “ancienneté”, par tous. Les noms des demandeurs sont en effet surlignés, et cet effet disparaît progressivement avec le temps.

L'intérêt est double : d'une part le conférencier peut directement visualiser depuis combien de temps une demande a été faite et peut par exemple considérer qu'une demande est "ancienne“ et n'est probablement plus justifiée ; d'autre part, du point de vue de l'auditeur, celui-ci peut quant à lui hésiter à émettre une demande lorsqu'il s'aperçoit qu'il y en a déjà de nombreuses autres avant la sienne (cela correspond au nombre de doigts levés dans une assistance réelle).

L'assistant Un second outil est présent pour permettre à chacun de mieux apporter sa contribution à la coopération. Il s'agit d'un *assistant* qui, comme son nom l'indique, doit faciliter l'exploitation de l'outil pour l'utilisateur.

On peut citer trois rôles principaux pour cet assistant :

- mise en évidence d'événements,
- aide à la participation à la coopération,
- modélisation des autres intervenants en vue de futures collaborations.

Mise en évidence d'événements La nature distribuée de l'application implique la nécessité de valoriser un certain nombre d'événements qui seraient plus immédiatement perceptibles dans une conférence réelle. On peut citer par exemple les arrivées/départs des intervenants, le changement de conférencier, les messages envoyés par les autres (cf. figure 5.4), etc. La contextualisation des réactions de l'interface graphique de l'assistant en fonction des événements qu'il signale aide l'utilisateur en l'orientant vers les outils concernés. Cette fonction contribue donc à faciliter l'utilisation de l'application.

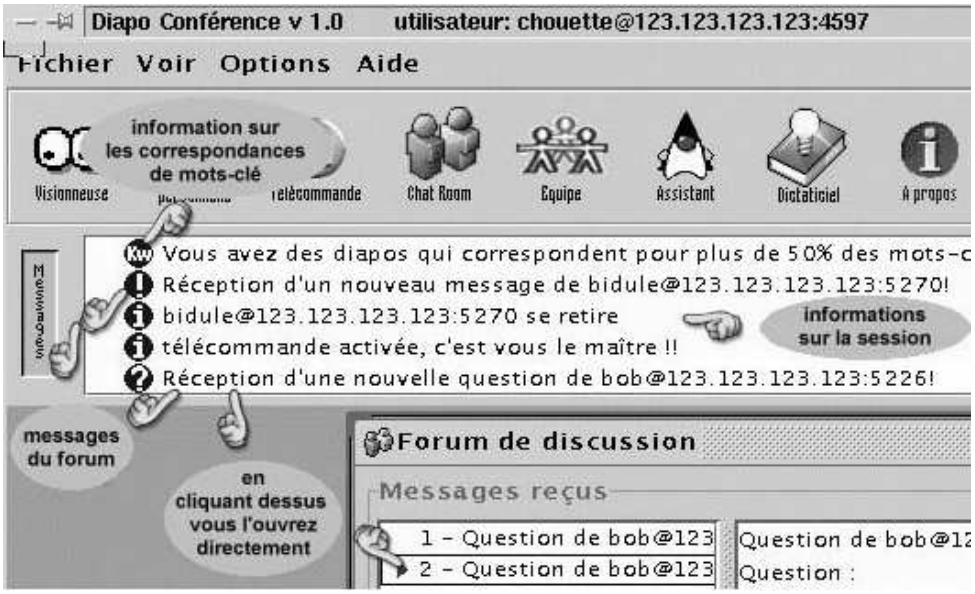

FIG. 5.4 – *Evénements dans l'assistant*

Aide à la coopération Parmi les "réactions" mentionnées, l'une d'entre elles fournit une aide à la construction de la coopération. En effet, lors de la tenue d'une diapo-conférence, l'assistant va aider l'utilisateur auditeur à apporter sa contribution à l'exposé. Pour chaque nouvelle diapositive transmise par le conférencier, l'assistant prévient l'auditeur quand il existe dans sa base de ressources une diapositive dont la thématique est corrélée à la diapositive courante. L'auditeur peut alors consulter le ou les documents sélectionnés par l'assistant dans une visionneuse locale, ce qui lui permet de juger par lui-même de la pertinence ou de la plus-value dans le discours courant de celui-ci. Il peut ensuite demander la télécommande pour devenir conférencier et ainsi "prendre la main". Mais il peut également choisir de demander à son assistant de transmettre la diapositive à l'assistant du conférencier. Ce dernier prévient alors son utilisateur qu'une diapositive lui est proposée par un auditeur et celui-ci peut la diffuser s'il la juge pertinente.

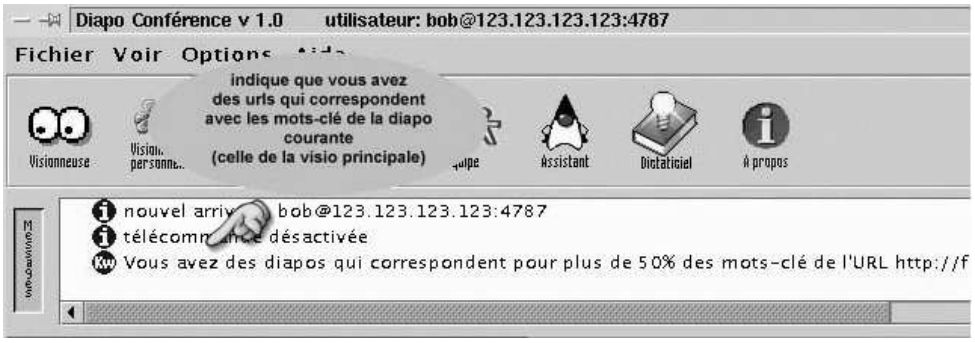

FIG. 5.5 – *L'assistant m'avertit que j'ai des ressources corrélées*

Dans la version actuelle, l'aide à la décision fournie par l'assistant se fonde sur un système de mots-clés qui doivent avoir été précisés au niveau de chaque ressource (cf. figure 5.5). L'assistant consulte la base de ressources de son utilisateur et il se manifeste dès qu'un nombre suffisant de mots-clés correspond. Le critère de corrélation retenu est paramétrable par l'utilisateur.

Modélisation des intervenants Mais les mots-clés peuvent avoir une autre utilité et être exploités différemment par l'assistant. Il est en effet possible de relever pour chaque diapositive diffusée, les mots-clés concernés et construire ainsi progressivement une image des domaines de compétences du conférencier, mais aussi une image des sujets d'intérêt des auditeurs. Ainsi, au fil des différentes sessions de conférence, une modélisation des différents intervenants et de leurs champs d'expertise peut être réalisée. Cette connaissance est ensuite utilisée pour faciliter la construction de conférences en invitant les experts sur les thèmes visés ou en prévenant les auditeurs qu'un sujet les intéressant sera abordé. Cette information pourra également être exploitée en dehors du cadre des conférences lorsque l'utilisateur cherche une information sur un domaine précis, il peut demander à son assistant s'il a connaissance d'une personne compétente sur le sujet.

Le système de messagerie Enfin, un système de messagerie permet à chaque auditeur de poser une question au conférencier (et rien qu'à lui), alors que ce dernier peut diffuser un message (annonce ou réponse à une question) vers tous les auditeurs. Il est donc possible au conférencier de s'adresser à tous les auditeurs et de leur transférer les questions qui lui sont posées s'il trouve cela pertinent.

5.1.2 Diapo-conférence : la réalisation avec Magique

Nous allons maintenant étudier comment l'application de télé-conférence présentée précédemment peut être développée facilement avec MAGIQUE. En fait, comme nous l'avons vu précédemment, le surcoût de développement lié à MAGIQUE et aux aspects multi-agents est presque inexistant et n'est quasiment dû qu'à l'agent coordinateur que nous présenterons par la suite. Les fonctionnalités applicatives comme la visionneuse, la gestion des équipes ou autres doivent de toute façon être mises en œuvre.

Examinons plus en détail cette réalisation. Il convient de déterminer les agents intervenants et les compétences de chacun, puis de définir l'organisation du système et les interactions possibles.

Analyse

Il s'agit dans un premier temps d'identifier clairement les rôles nécessaires à l'application et les compétences qui y sont attachées.

Les rôles Nous avons déjà clairement identifié deux de ces rôles : ceux de *conférencier* et d'*auditeur*. Il existe cependant un troisième rôle qui n'a pas encore été explicitement nommé. Il est en effet nécessaire, pour que la conférence puisse avoir lieu, que les intervenants puissent la quitter ou la rejoindre et qu'une entité fasse le lien entre les différents participants. Nous appellerons *coordinateur* ce rôle mais il peut être vu comme l'*environnement*, notion habituelle des systèmes multi-agents, puisqu'il constitue le support de l'interaction et concrétise par son existence la cohabitation de l'ensemble des participants à la conférence.

Les compétences Pour le rôle *coordinateur*, les compétences attachées sont assez limitées, il suffit de fournir aux agents un point d'entrée à la conférence et de maintenir la cohérence en cas de déconnexion (notamment si c'est le conférencier qui quitte la session, il faut passer le relais – la télécommande – à un des auditeurs).

Pour les rôles de *conférencier* et d'*auditeur*, les compétences sont assez similaires et correspondent aux fonctionnalités décrites dans la première section. Les voici de manière informelle (la figure 5.6 présente le diagramme de classe de ces compétences) :

- *visionneuse* : pouvoir récupérer, interpréter et afficher les ressources, gérer le pointeur et les manipulations (`HTMLSkillInterface`),
- *la fenêtre des intervenants* : gérer les demandes de télécommande ainsi que ses transferts (`RemoteControlSkillInterface`),
- *la messagerie* : permettre les échanges entre les participants en respectant le statut *conférencier* et *auditeur*, cette compétence doit donc être contextualisée au rôle (classe `MessageSkill`).
- *assistant* : fournir les fonctionnalités d'aide à la coopération et à la modélisation des intervenants mentionnées précédemment (`AssistantSkillInterface`).

Et il reste pour ces deux rôles à traiter la compétence de gestion de la *télécommande*. Cette compétence doit permettre :

- au rôle *conférencier* de diriger la conférence (passage de diapositives, gestion du pointeur et de la zone de visualisation),
- aux deux rôles de gérer leurs ressources et d'organiser des séquences de diapositives (cette fonctionnalité étant un outil fournit en plus.

Le cas de cette compétence est particulièrement intéressant, car c'est justement elle qui crée la différence entre les deux rôles de *conférencier* et d'*auditeur*. L'évolution dynamique des rôles en

cours de session (de *conférencier* à *auditeur* et réciproquement) implique une évolution dynamique de la compétence connue par les agents. Il est en effet nécessaire d'accorder au nouveau conférencier les fonctions pour mener la conférence et de les retirer à l'ancien conférencier.

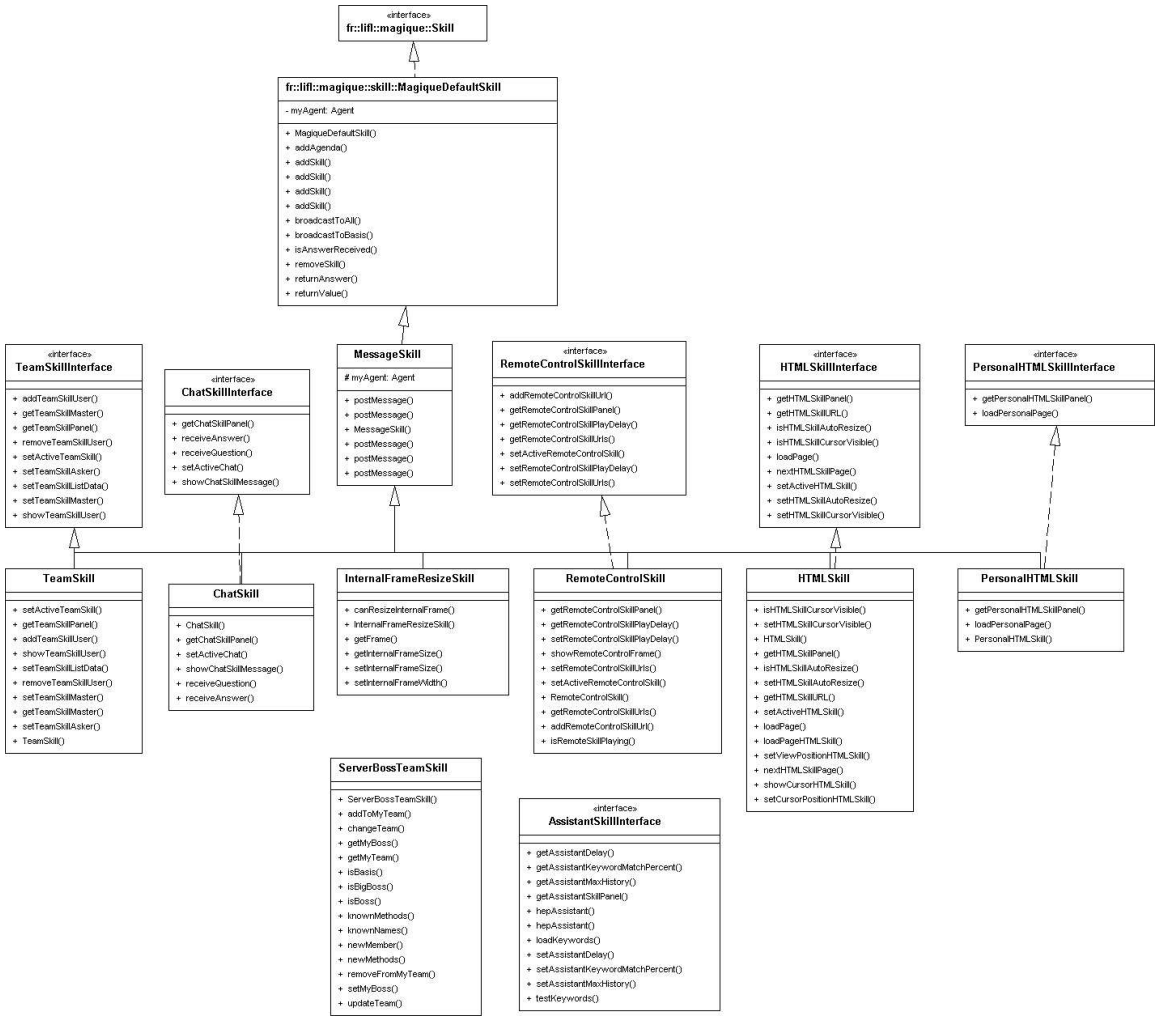

FIG. 5.6 – *Diagramme de classes des compétences définies pour diapo-conférence*

Les choix d'organisation du SMA

Comme cela a déjà été mentionné, nous avons utilisé la plateforme multi-agents MAGIQUE pour développer un prototype pour cette application. Chaque intervenant est défini par un agent qui le représente, un de ces agents aura le rôle de *conférencier* et nles autres seront donc *auditeurs* (cf. figure 5.7). Nous avons décidé de créer en plus un agent particulier, pour qu'il joue uniquement le rôle de *coordinateur* (même si ce rôle aurait pu être joué par un des autres agents, l'initiateur de la conférence par exemple, ou pourrait même évoluer dynamiquement).

Nous avons expliqué que les systèmes multi-agents dans MAGIQUE étaient organisés hiérarchiquement. Dans ce cas, nous avons choisi de placer l'agent *coordinateur* à la racine du système et de placer tous les autres agents au niveau inférieur. Remarquons qu'une autre possibilité était

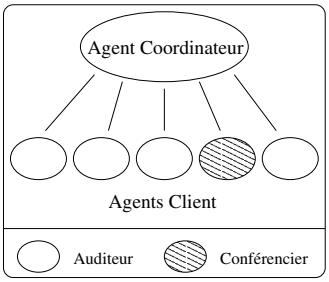

FIG. 5.7 – *L'organisation hiérarchique du SMA*

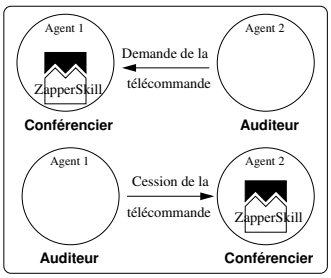

FIG. 5.8 – *Evolution dynamique de rôles par échange de compétences*

de placer l'agent jouant le rôle de *conférencier* à la racine, ce qui impliquait une réorganisation de la hiérarchie lors du changement des rôles. Ce qui est réalisable avec MAGIQUE.

L'évolution dynamique des rôles *conférencier*$\longleftrightarrow$*auditeur* est facilement prise en compte par MAGIQUE. Cette plateforme a en effet l'avantage de permettre l'évolution dynamique des compétences des agents par échanges entre eux. Dans MAGIQUE, les agents apprennent et oublient *effectivement* des compétences. Les échanges de compétences sont réels, indépendamment de toute hypothèse sur le code de ces compétences et de la distribution sur le réseau des agents. L'agent *conférencier* pourra donc, comme cela se passe lors du passage de micro dans une conférence réelle, donner la compétence de gestion de la télécommande au nouveau *conférencier* et de ce fait la *perdre* (cf. figure 5.8). Et il ne s'agit pas ici de la simple modification de la valeur d'un attribut quelconque, mais d'une mutation concrète de l'agent et du rôle qu'il tient et donc en quelque sorte de ses droits dans l'application.

La programmation

Pour réaliser cette application avec MAGIQUE, il "suffit" de développer les différentes compétences mentionnées précédemment.

Chacune représente un composant. Une fois l'interface de la compétence définie (cf. figure 5.9), il convient de l'implémenter sous la forme d'objets JAVA, la seule différence à prendre en compte représente la nécessité d'effectuer des interactions entre les agents et les différentes autres compétences et nous avons déjà vu précédemment que cela se faisait facilement grâce au mécanisme de délégation à l'aide des primitives `perform`, `ask`, etc.

Ensuite, la construction des agents se fait très simplement, il suffit d"'enseigner" à l'agent de base les compétences requises une par une. Ce qui donne quelque chose comme ce qui est présenté à la figure 5.10.

La couche MAGIQUE gère ensuite les connexions des agents et le transport des messages dus aux interactions entre agents. Ces communications n'apparaissent pas explicitement au niveau du code, puisqu'elles sont induites par les invocations de compétences et donc prises en charge par MAGIQUE.

Evolutivité et adaptativité

Le concept de compétence facilite l'évolution de l'application et son adaptation à différents contextes.

Il est ainsi possible d'étendre facilement les fonctionnalités offertes à travers la création de

```
public interface HTMLSkillInterface
{
  public JPanel getHTMLSkillPanel();
  public URL getHTMLSkillURL();
  public void setActiveHTMLSkill(Boolean b);

  /** Charger une diapo. */
  public void loadPage(URL url);

  public Boolean nextHTMLSkillPage();
  /** Options */
  public Boolean isHTMLSkillCursorVisible();
  public void setHTMLSkillCursorVisible(Boolean b);
  public Boolean isHTMLSkillAutoResize();
  public void setHTMLSkillAutoResize(Boolean b);
}
```

FIG. 5.9 – *L'interface de la compétence* `HTMLSkill` *(sans les commentaires). Il suffit donc de l'implémenter pour l'adapter à l'agent indépendamment des autres compétences. Il est notamment possible d'adapter la compétence de chargement de page en fonction du support et ce pour chaque agent indépendamment des autres*

nouvelles compétences qu'il suffit ensuite d'"enseigner" aux agents. On peut ainsi sans aucun problème envisager une construction incrémentale de l'application.

Par exemple, une compétence permettant le support de la voix pourrait ainsi être ajoutée pour faciliter l'échange d'informations, sans que cela ait d'impact sur le reste de l'application. Il suffirait en effet de prévoir les interactions entre les agents représentant l'application chez les différents utilisateurs et de leur "enseigner" la compétence créée pour qu'il soit possible de l'utiliser.

L'adaptation à différents supports ou différents types de ressources (XML par exemple) peut également être facilement obtenue. Il suffit de changer l'interprète de document pour le visualisateur. De même, si l'un des auditeurs souhaite utiliser l'application sur un client léger (type *Palm Pilot* ou *Psion*), il peut adapter la compétence de visualisation en "enseignant" à son

```
import fr.lifl.magique.*;
    ...
    // création d'un agent "vide"
    Agent myAgent = new Agent("myName");
    // l'agent apprend ses compétences
    myAgent.addSkill("MaZapperSkill");
    myAgent.addSkill("MaHTMLSkill");
    myAgent.addSkill("MonAssistantSkill");
    ... autres enrichissement de skill ...
    ...
```

FIG. 5.10 – *Création d'un agent*

agent la compétence visionneuse adaptée à ce client à la place de celle prévue par défaut.

Enfin, pour donner un dernier exemple illustrant l'ouverture d'une telle application développée avec MAGIQUE, il est facile d'envisager d'adapter l'assistant pour chacun des utilisateurs. Il est par exemple évidemment envisageable de définir différentes stratégies pour déterminer la corrélation des ressources du propriétaire de l'agent avec les documents diffusés ; ou encore différents critères pour réaliser la modélisation des autres intervenants.

5.2 Un framework de calculs distribués : Rage

Le second exemple d'application que nous allons présenter est un framework de calcul distribué, RAGE, qui a été implémenté avec la plateforme MAGIQUE. Cet exemple illustre la facilité de conception et d'implémentation de telles classes d'applications, lorsque l'on analyse ces systèmes en terme de rôles et de compétences.

Nous présenterons dans un premier temps le framework RAGE du point de vue de l'utilisateur, c'est-à-dire de la personne désirant développer un calcul distribué. Puis, nous verrons dans un second temps la démarche que nous avons suivi lors de la conception de RAGE. Comme avec l'application précédente, l'utilisateur final n'a pas à avoir de connaissances préalables sur les systèmes multi-agents, mais il doit cependant connaître la programmation orientée objets (et plus particulièrement le langage JAVA) pour pouvoir utiliser RAGE.

5.2.1 Le framework *Rage* : la boîte à outils de calculs distribués pour le non-informaticien

Le framework RAGE, acronyme de Reckoner AGEnt, est un cadre de développement destiné aux scientifiques non-informaticiens, dédié à la réalisation de calculs distribués. Ce système supporte l'exécution de plusieurs calculs simultanément et peut voir sa puissance de calcul augmenter ou diminuer dynamiquement en fonction de la disponibilité des plateformes hébergeant les calculs. Dans RAGE, il y a deux types de clients : ceux qui fournissent des calculs à effectuer, et ceux qui proposent leur puissance de calcul (figure 5.11). Le premier type de client correspond au scientifique désirant effectuer des calculs, et nécessite donc l'écriture d'un programme. Tandis que le second client est fourni par le framework, et correspond à un programme que les utilisateurs n'ont qu'à exécuter pour augmenter la puissance de calcul de RAGE.

La *simplicité* d'utilisation est le critère principal de ce framework. Cet environnement étant destiné à des non-informaticiens, nous avons défini un faible nombre de concepts que l'utilisateur devra s'approprier pour pouvoir alimenter RAGE en calculs. Ces concepts sont principalement celui de *tâche* et de *résultat*. Une *tâche* est un algorithme qui pourra être distribuée dans le système, tandis qu'un *résultat* représente les données produites par la réalisation d'une *tâche* et devant être sauvegardées. Le modèle de *tâche* est simple et suppose qu'il n'y a pas de communications nécessaires à la réalisation de la *tâche* : on est proche sur ce point de systèmes tels que SETI[61] ou le DECRYPTHON.

L'utilisateur du framework doit définir ce qui est distribué, et l'ordonnancement de son algorithme principal. Prenons l'exemple du calcul d'une matrice qui est ensuite utilisée pour résoudre un système d'équations linéaires. L'algorithme principal, c'est-à-dire celui s'exécutant chez le client, se décompose en deux phases distinctes : d'abord le calcul de la matrice (distribué),

61. Plus d'informations sur SETI : `http://www.seti.org`.

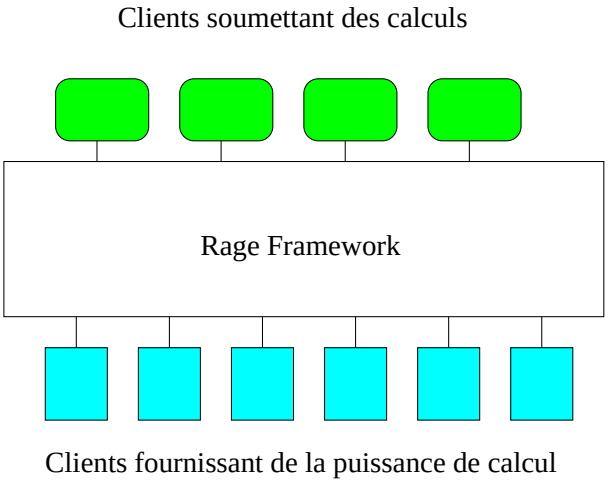

FIG. 5.11 – *Une vue globale de* RAGE *avec ces deux types de clients*

puis la résolution du système d'équation (non distribué). Lors de la première étape, l'utilisateur distribue les *tâches* qui vont produire la matrice, tandis que la seconde étape consiste à récupérer la matrice pour résoudre le système.

L'utilisateur doit donc définir ce qu'est une *tâche*, ie. quel algorithme sera attribué aux agents calculateurs. Pour cela l'utilisateur doit implémenter l'interface `Task`, ou plus simplement hériter de la classe `AbstracTask` et définir les deux méthodes suivantes :

```
abstract public void compute();
abstract public boolean finished();
```

En effet, le modèle de *tâche* dans RAGE est celui d'une boucle de calcul revenant à invoquer la méthode `compute()` jusqu'à ce que la méthode `finished()` renvoi `true`. Ce choix a été dicté par des impératifs technologiques : principalement la difficulté de suspendre proprement un processus, ou thread, dans le langage Java et de conserver son état. Ainsi, l'utilisation de ce cycle de vie (figure 5.12) des *tâches* rend possible la terminaison ou la migration de *tâches* en cours d'exécution.

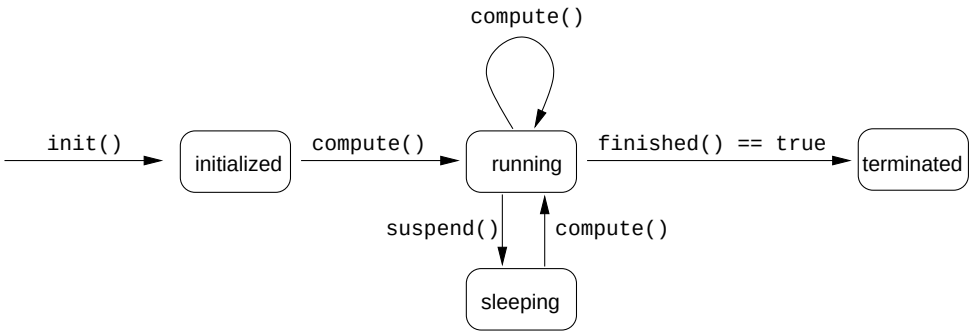

FIG. 5.12 – *Le cycle de vie d'une* tâche *dans* RAGE.

Ceci implique que l'utilisateur doit penser son calcul, ou plus précisément l'écriture de la

méthode `compute()`, comme intrinsèquement coopératif vis-à-vis du modèle de *threading* [62]. Plus concrètement, l'utilisateur doit éviter d'utiliser des boucles demandant un temps d'exécution trop important dans la méthode `compute()`.

Ainsi, pour l'utilisateur, la définition d'une *tâche* n'est pas beaucoup plus compliquée que celle d'une `Applet`. L'utilisateur a ainsi une vision purement orientée objets du framework : il étend une classe pour la spécialiser à un type de calcul et utilise le framework sous-jacent pour exécuter ses calculs sans nécessairement savoir que c'est un système multi-agents qui effectue le travail.

Pour illustrer ce que l'utilisateur doit écrire, nous allons détailler un exemple simple : le calcul d'une approximation du nombre π. Le principe de l'algorithme est de choisir aléatoirement des points dans un carré unitaire, et de vérifier si ces points appartiennent au quart de cercle (figure 5.13). Ensuite, il suffit d'appliquer la formule suivante : $\pi = 4 * I/N$, où N est le nombre total de points et I est le nombre de points appartenant au quart de cercle.

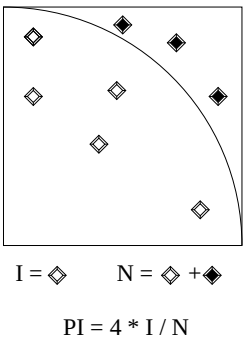

FIG. 5.13 – *Un algorithme de Monte-Carlo pour le calcul d'une valeur approchée de* π

La précision de l'approximation s'accroît avec N, le nombre de points tirés. Ainsi, l'utilisateur doit définir une `PiTask` qui implémentera l'algorithme précédemment décrit, et devra créer un ensemble de ces tâches avant de les distribuer dans RAGE. Le code source complet de la classe `PiTask` est le suivant :

62. Cette approche est inspiré des travaux menés sur le projet FAIRTHREADS de l'INRIA : `http://www-sop.inria.fr/mimosa/rp/FairThreads/`.

```
public class PiTask extends AbstractTask {
  public Double pi;
  private int crtIter = 0, inner = 0, niter, chunk;
  public PiTask(String factId, String taskId, int niter){
    super(factId, taskId);
    this.niter = niter; chunk = niter/10;
  }
  public void compute(){
    double x, y;
    for(; (crtIter % chunk) < chunk; crtIter++){
      x = Math.random(); y = Math.random();
      if (Math.sqrt(x*x + y*y) <= 1.0)
        inner ++;
    }
    pi = new Double(4*((double)inner/(double)niter));
  }
  public boolean finished(){
    return crtIter == niter;
  }
  public double percent(){ // Percent of progress
    return (double) (crtIter * 100)/(double)niter;
  }
}
```

En dépit de sa simplicité, cet exemple illustre bien le modèle de programmation d'une *tâche* dans RAGE :

- étendre la classe `AbstractTask` (ou implémenter l'interface `Task`),
- implémenter les méthodes `compute()` et `finished()`,
- marquer les attributs devant être sauvegardés avec le modificateur de visibilité `public`.

Ensuite, l'utilisateur n'a plus qu'à créer l'ensemble des tâches (i.e. des instances de ses classes de tâche), et à déléguer leur traitement au framework. Il pourra ensuite récupérer les résultats de ses calculs et effectuer la moyenne pour obtenir une valeur approchée de π.

Un point particulièrement intéressant de la méthode `compute()` est qu'elle illustre la délégation de contrôle de la tâche à l'agent qui la gère. Si l'agent reçoit une requête lui intimant de terminer ou de migrer une tâche, il ne pourra la satisfaire que si l'utilisateur "rend la main" à l'agent. Ainsi, dans l'exemple précédant, une `PiTask` nécessitera 10 invocations de la méthode `compute()` avant d'être achevée. La granularité de cette méthode doit être précautionneusement étudiée par l'utilisateur si il veut bénéficier des facilités de terminaison ou de migration des tâches de RAGE.

Maintenant que nous avons vu ce que l'utilisateur doit effectuer pour "nourrir" le système, nous allons expliquer le cycle que suivent les tâches dans le système. Lorsque les utilisateurs soumettent des tâches, elles sont stockées par un agent qui les distribuent ensuite aux agents calculateurs. Les agents calculateurs, lorsqu'ils ont une tâche à traiter, n'ont qu'à invoquer la méthode `compute()` jusqu'à ce que le calcul de la tâche soit terminée (ie. lorsque la méthode `finished()` retourne `true`). Ensuite, une instance de résultat est générée par introspection des attributs publics de la tâche, et cet objet est envoyé à un agent gérant la base de données des résultats. L'utilisateur peut ainsi régulièrement interroger cet agent pour récupérer les nouveaux résultats.

5.2.2 Rage : conception et implémentation avec Magique

Comme l'application précédente, Rage a été développée avec Magique. Nous nous sommes inspiré dans un premier temps la méthodologie Gaia[88] pour réaliser l'analyse, puis nous avons été amené à développer les concepts de Rio dans un second temps. En effet, la méthodologie Gaia fournit un cadre intéressant d'analyse, mais ne fournit pas de support facilitant la transition de la phase d'analyse à la phase de réalisation.

Les principales étapes que nous avons suivies lors de l'analyse et la conception de Rage sont les suivantes :

- définition des rôles impliqués dans le système, de leurs interactions et de leurs compétences,
- définition de l'organisation des rôles au sein du système,
- l'association des rôles aux agents concrets.

La première étape identifie les principales entités de l'application, ainsi que leurs compétences. C'est aussi lors de cette étape que les interactions entre les différentes entités sont explicitées. La seconde étape définit l'occurrence des rôles au sein du système multi-agents, c'est-à-dire le nombre d'agents qu'il sera possible de créer pour un rôle donné, ainsi que leur position au sein de l'organisation. Finalement, la dernière étape consiste à effectuer la projection des rôles sur les agents Magique, cela correspond au déploiement du système multi-agents.

L'identification des rôles

La première phase consiste à identifier les différentes entités qui interviennent dans le système. Cela revient à déterminer les fonctions que le système doit fournir, il faut ensuite regrouper ces fonctions de manière homogène. Dans le cas de notre framework de calcul distribué, nous avons choisi les rôles suivants :

- Le rôle *Boss* qui est responsable des interactions avec les utilisateurs,
- Le rôle *Task Dispatcher* se charge de la répartition des *tâches* et gère aussi la tolérance aux pannes.
- Le rôle *Platform Manager* gère les agents qui ont à effectuer les calculs des *tâches*, et doit s'assurer que l'approvisionnement en *tâche* est toujours suffisant.
- Le rôle *Reckoner Agent* calcule les *tâches* et retourne les *résultats*.
- Le rôle *Repositories Manager* gère le stockage et l'accès aux *résultats*.
- Le rôle *Result Repository* est un miroir de la base de données des *résultats*.

Dans la mesure où nous avons implémenté ce système avec Magique, nous avons concrétisé ces rôles en développant les compétences nécessaires. Par exemple, le rôle *PlatformManager* est défini par une compétence Magique qui définie son comportement : maintenir un *pool* de tâches et gérer leur distribution aux *Reckoner Agent*s. Un rôle est ainsi défini dans Magique par un ensemble de compétences et les interactions ne sont pas réifiées et se trouvent enfouies dans le code des compétences.

Maintenant que nous avons défini les entités constituant le système, nous allons décrire le cycle de vie d'une *tâche* dans Rage (figure 5.14). Ce schéma représente les interactions se produisant entre les compétences lors de l'ajout d'une tâche dans le système.

Lorsque l'utilisateur dépose des *tâches* au *Boss*, ce dernier les envoie au `TaskDispatcher` qui les stocke jusqu'à ce que les `Platform Manager` les demandent. Lors de cette requête, le `TaskDispatcher` conserve la tâche, et lui associe un *timestamp* pour pouvoir la relancer après un temps donné (cette information est délivrée par la tâche elle même). Les `Platform Manager` maintiennent un *buffer* de `Task`s et les allouent ensuite aux différents `ReckonerAgent`s qu'ils

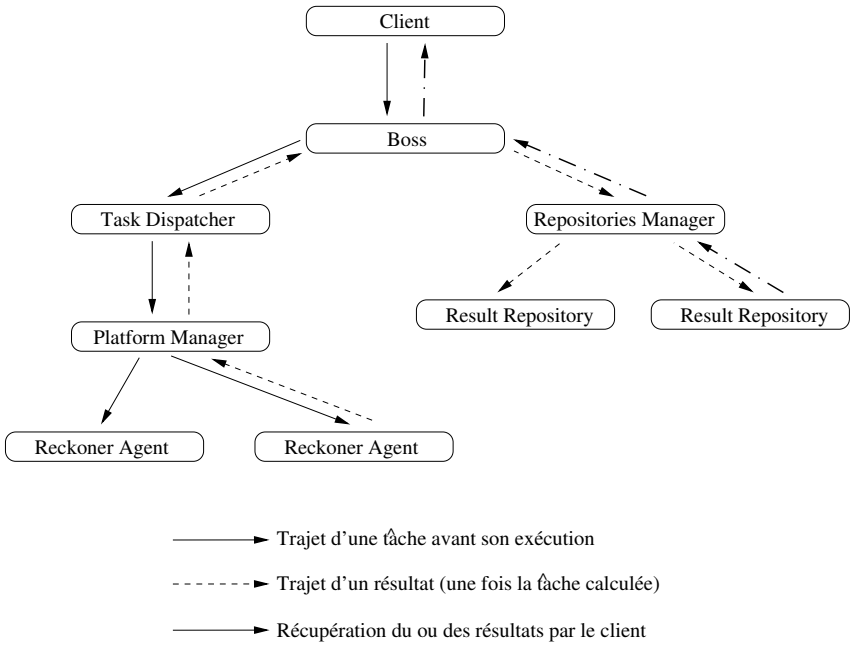

FIG. 5.14 – *Cycle de vie d'une* tâche *et de son* résultat.

gérent. Une fois la tâche calculée, le résultat est généré et envoyé au `Repositories Manager` qui se charge de le distribuer aux différents `RepositoryManager`. Ensuite, lorsque l'utilisateur en fait la requête auprès du *Boss*, il peut interroger la base de `Results` et les récupérer et poursuivre le déroulement de son algorithme principal.

Suite à cette phase d'analyse, qui a identifié les principaux acteurs intervenant dans le système, leur responsabilité et leurs interactions, il est possible de définir les interfaces de compétence. Dans MAGIQUE, cela consiste à définir des interfaces JAVA, c'est-à-dire uniquement des signatures de méthodes.

La figure 5.15 détaille les interfaces des compétences intervenant lors de l'exécution d'une tâche. Cependant, cette figure est un diagramme de classe et ne fait pas apparaître les dépendances *enfouies* dans le code des classes implémentant ces interfaces. En effet, les communications s'effectuant au travers des primitives `perform`, `ask` ou `askNow`, il n'y a même pas d'association au sens d'UML.

L'implémentation de ces interfaces pourra être multiple, notamment si l'on désire pouvoir utiliser cette compétence dans différents environnements (J2ME, PERSONALJAVA, J2SE ou J2EE). Lors du déploiement, il sera alors nécessaire de lier les interfaces avec leur implémentation avant d'affecter les rôles aux agents concrets. Cette étape reste entièrement à la charge du développeur, et il n'existe pas d'outil gérant l'hétérogénéité des environnements.

La définition de l'organisation

Après avoir décrit les rôles, il est nécessaire de définir l'architecture du système à savoir l'organisation des agents dans le système multi-agents. Lorsque l'on travaille avec MAGIQUE, cela signifie que l'organisation doit satisfaire deux contraintes : d'abord un groupement logique des rôles, mais aussi un schéma de communication par défaut facilitant la délégation de requête. Si l'on observe les flux d'informations dans RAGE(figure 5.14), on en déduit la hiérarchie suivant :

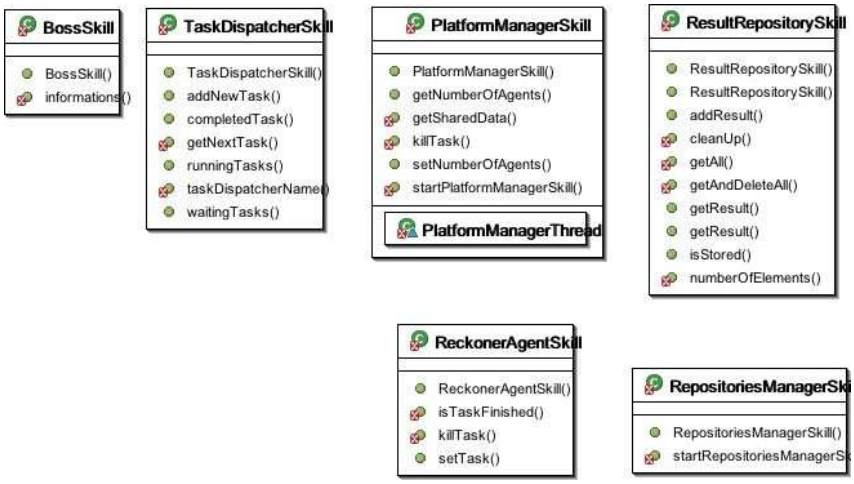

FIG. 5.15 – *Les compétences définissant* RAGE.

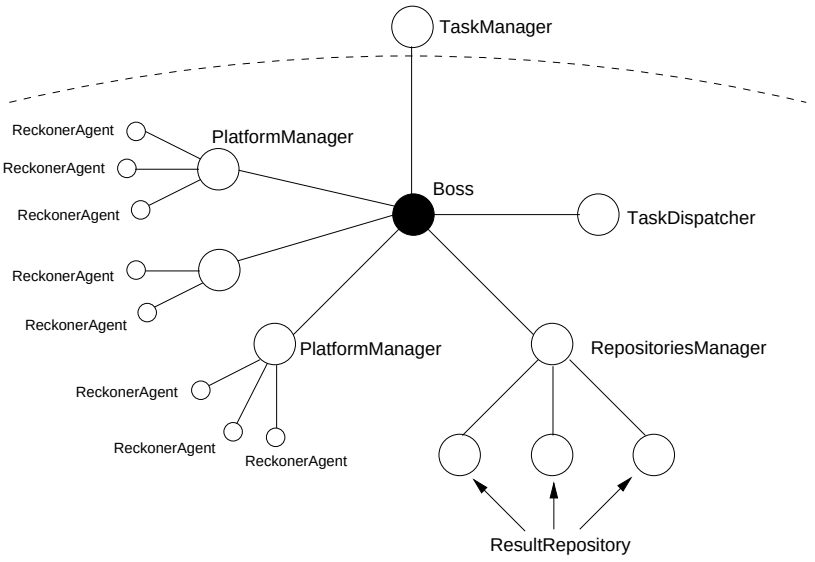

FIG. 5.16 – *La hiérarchie des rôles dans* RAGE.

La figure 5.16 représente la hiérarchie des rôles dans RAGE : elle ne définit pas comment les rôles sont associés aux agents concrets, ni comment ces agents sont répartis sur le réseau, mais décrit la topologie des liens hiérarchiques. Cette topologie est l'organisation initiale, qui pourra évoluer au gré des interactions entre les agents.

Lors de l'affectation des rôles aux agents concrets, il n'est pas nécessaire de respecter la règle "un rôle = un agent", l'ensemble de la hiérarchie pourrait être projeté sur un seul agent concret (bien que cela ne serait pas très utile). Il est important de signaler que nous aurions appliqué la même décomposition de rôle avec un autre système multi-agents, par contre nous aurions probablement à utiliser une autre topologie d'organisation. Si nous avions développé RAGE avec MADKIT[41], l'organisation aurait été obtenue en utilisant le modèle AALAADIN[31], et ainsi nous aurions organisé les agents en groupes plutôt qu'en une hiérarchie.

Le code ci-dessous illustre la création de l'ensemble des agents et de la hiérarchie sur une machine. On retrouve la phase de création d'un agent MAGIQUE, l'ajout de la compétence qui caractérisera le rôle de l'agent et la création de la hiérarchie. Les deux dernières méthodes `perform` déclenchent la création des sous-hiérarchies propres au `PlatformManager` et au `RepositoryManager`.

```
// Platform
Platform platform = new Platform(port);
System.out.println("Platform " + platform.getName() + " launched");

// Boss agent
Agent boss = new Agent(bossName);
boss.addSkill(new BossSkill(boss));
platform.addAgent(boss);

// Task Dispatcher
Agent taskDispatcher = new Agent("TaskDispatcher");
taskDispatcher.addSkill(new TaskDispatcherSkill(taskDispatcher));
platform.addAgent(taskDispatcher);
taskDispatcher.connectToBoss(bossName);

// Repositories Manager
Agent repositoriesManager = new Agent("RepositoriesManager");
repositoriesManager.addSkill(new RepositoriesManagerSkill(repositoriesManager));
platform.addAgent(repositoriesManager);
repositoriesManager.connectToBoss(bossName);

// Platform Manager
Agent platformManager = new Agent("PlatformManager");
platformManager.addSkill(new PlatformManagerSkill(platformManager));
platform.addAgent(platformManager);
platformManager.connectToBoss(bossName);

System.out.println("Hierarchy built !! Boss name is " + boss.getName());
boss.perform("startPlatformManagerSkill");

System.out.println("Start Repositories Manager ...");
boss.perform("startRepositoriesManagerSkill");
```

5.2.3 Quelques expérimentations réalisées avec Rage

Pour évaluer notre framework, nous avons réalisé plusieurs expériences dont la complexité varie du cas d'école à des applications complètes. La première expérience est le tutoriel que nous avons vu dans la sous-section 5.2.1, le calcul d'une valeur approché de *pi*. Cet exemple est simpliste mais permet de présenter les concepts fondamentaux de Rage.

La deuxième expérience est aussi un tutoriel, qui est une implémentation naïve de la décomposition en facteur premier de grand nombre. L'algorithme utilisé est une recherche exhaustive de facteur premier en distribuant des segments de l'espace de recherche (qui s'étend entre 2 et $sqr(N)$). Le but de cette expérience était de tester la montée en charge du framework en terme de nombre de machines de calcul (les *Platform Manager*s).

Le troisième exemple est plus conséquent : c'est une étude de la structure sous-jacente des jeux de taquins multiformes, et plus particulièrement du jeu de l'âne rouge (*Donkey sliding block game*[29]). L'algorithme principal est intéressant car il nécessite trois phases distinctes :

- le calcul de l'ensemble des états valides d'un jeu donné,
- le calcul des graphes d'état des jeux,
- la caractérisation de l'intérêt des graphes.

Ainsi, la première étape consiste à générer l'ensemble des états valides d'un jeu donné. Un jeu est défini par un plateau et un ensemble de pièces. Un état du jeu est représenté par un mot (obtenu en lisant le plateau de gauche à droite et de haut en bas). La première phase consiste donc à générer l'ensemble des mots possibles, en calculant les permutations de la position initiale. Cependant, il est possible que les mots obtenus ne représentent pas un état valide du jeu (ie. une pièce "sort" du plateau de jeu), il faut donc au fur et à mesure de la génération des permutations, vérifier qu'elles correspondent à des états valides.

Une fois cette première étape effectuée, la base de résultat contient l'ensemble des états valides, qui vont constituer autant de graines à partir desquelles il sera possible de créer les graphes d'états du jeu. La seconde étape consiste donc à prendre des états au hasard, et de s'en servir comme position initiale pour pouvoir créer le graphe complet de ce jeu. Cette deuxième phase est intéressante car elle illustre la terminaison de tâches dans Rage. En effet, il est possible que deux tâches calculent le même graphe, il faut donc mettre en place un mécanisme permettant de repérer ces redondances et de terminer la ou les tâches les moins avancées (la fusion de ces graphes n'étant pas aisée). L'algorithme utilisé actuellement n'est pas optimal mais a l'avantage d'être facile à mettre en place : lorsqu'une tâche se termine, l'ensemble des nœuds du graphe qu'elle vient de créer est communiqué aux autres tâches qui peuvent se terminer si elles calculent le même graphe. A la fin de cette étape, la base de résultats contient l'ensemble des graphes du jeu. Il se trouve que la plupart de ces graphes ne sont pas intéressants : leur taille est trop réduite pour pouvoir contenir des jeux intéressants pour un humain. Dans le cas de l'Ane Rouge, il n'y a qu'un graphe intéressant, et il est constitué d'environ 26000 nœuds.

La dernière étape n'a pas été implémentée pour l'instant, mais l'objectif serait de trouver des heuristiques permettant de déterminer "automatiquement" les jeux intéressants à partir des graphes obtenus à la fin de la deuxième étape. Ainsi, cette dernière étape consisterait à tester les différentes heuristiques sur les plus gros graphes, afin d'obtenir une métrique permettant de caractériser des jeux intéressants.

Le dernier exemple qui a été réalisé est une application en mécanique des solides : c'est une implémentation d'un calcul de déplacement d'une discontinuité dans un environnement en deux dimensions (*two-dimensional displacement discontinuity method*[22]). Ce dernier exemple introduit une nouvelle possibilité du framework Rage : l'accès à une mémoire partagée par l'ensemble des tâches. Ce mécanisme est réalisé par les *Platform Manager*s : ils récupèrent les informations

auprès du client et les rendent accessibles à leur *Reckoner Agent*s. Ce n'est cependant qu'une implémentation partielle car cette mémoire n'est accessible qu'en lecture (nous n'avions pas de besoin en écriture, et les problèmes de cohérence qui se seraient alors posés auraient nécessité un travail trop conséquent par rapport à l'intérêt de cet exemple).

Ces expérimentations, ainsi que le framework RAGE, peuvent être chargées à l'adresse suivante : `http://www.lifl.fr/SMAC/projects/magique/examples`.

5.2.4 Les extensions possibles de Rage

Le framework RAGE ne fournit pas toutes les fonctionnalités qui pourraient faciliter son utilisation ou élargir son domaine d'application. Dans cette section, nous allons présenter les principales améliorations qu'il faudrait apporter au framework.

La principale fonctionnalité manquante dans RAGE est la gestion des communications entre tâches. Cela n'a pas été implémenté, cependant grâce au système multi-agents sous-jacent, cette fonctionnalité peut facilement être rajoutée. Chaque tâche est identifiée de manière unique et appartient à un *Reckoner Agent*, il est donc possible de tirer partie de la hiérarchie pour rechercher une tâche dans le système. Le principal problème pour que cette communication inter-tâche soit vraiment utile à l'utilisateur est de pouvoir exprimer des critères de recherche plus complexes que l'identifiant de tâche. Cela revient à attacher des méta-données décrivant la tâche ou encore à utiliser le *design pattern* `Visitor` sur les tâches permettant ainsi à la requête de déterminer si la tâche correspond à ses critères.

Une autre amélioration qui pourrait être ajoutée concerne l'ordonnancement multi-applications. Actuellement, plusieurs clients peuvent effectuer simultanément leurs calculs dans RAGE. Cependant, le *Task Dispatcher* ne gère pas de priorité et les distribuent dans l'ordre de leur arrivée (FIFO). Il serait possible d'introduire des politiques d'ordonnancement qui autoriseraient un client à exécuter deux fois plus de tâches qu'un autre. Ou bien, nous pourrions introduire une forme de politique économique[87] qui favoriserait les utilisateurs fournissant des ressources de calcul (en faisant tourner des *Platform Manager*) en leur attribuant de plus fortes priorités.

Une dernière importante amélioration concerne la gestion de la base de résultats. Elle est pour l'instant implémentée sous la forme d'une base de données orientée objets pour faciliter le déploiement du framework. En effet, pour éviter à l'utilisateur d'avoir à installer sa propre base et à la configurer, RAGE contient sa propre base de donnée *embarquée*(nous utilisons DB4O[63]). L'implémentation actuelle ne repose que sur une seule base détenu par un agent *Result Repository*. Or, la base de résultats est utilisée à la fois par les clients de RAGE, mais aussi à des tâches en cours d'exécution. De plus, l'objectif de RAGE étant de fonctionner sur des réseaux n'appartenant pas aux mêmes domaines, nous avions introduit l'agent *Repositories Manager* pour gérer le *mirroring* entre les différents *Result Repository* répartis sur le réseau.

Deux points restent délicats avant de passer à l'implémentation : le choix de la base et la définition de métriques sur l'usage des *Result Repository*. Le premier point pose le problème du choix entre les base de données relationnelles, dont la fiabilité n'est plus à démontrer, et les bases de données objets, qui s'intègrent aisément dans un framework déjà orienté objets. Le second point concerne la définition de critères caractérisant l'utilisation de la base de résultats par les tâches en cours d'exécution. Il serait en effet très intéressant de définir un mécanisme de *mirroring* adaptatif, qui ajusterait le nombre d'agents *Result Repository* dans le système en fonction de la charge. L'ajout de cette fonctionnalité ne poserait d'ailleurs pas trop de problèmes techniques : la création dynamique d'agent, ainsi que l'échange de compétences, permettent d'instancier

63. Plus d'informations sur DB4O : `http://www.db4o.com/`

de nouveaux agents*Result Repository* qui n'auraient qu'à se synchroniser avec la base la plus proche (trouvée naturellement via la hiérarchie qui conserve la topologie physique du réseau) avant d'être disponibles. Une autre approche d'implémentation de la réplication de la base de résultats pourrait être l'utilisation d'un mécanisme *peer-to-peer* de diffusion d'information comme le propose le système Freenet[18]. Le principal avantage de cette approche étant que la réplication se produit en fonction de l'utilisation du système et surtout qu'il n'y a pas de métrique à définir.

Conclusion

"*Connaître, ce n'est point démontrer, ni expliquer. C'est accéder à la vision. Mais pour voir, il convient d'abord de participer : cela est un dur apprentissage...*"

Antoine de Saint-Exupéry

Contexte et problématiques

Le développement de l'informatique personnelle, conjointement à celui des réseaux, a profondément modifié l'environnement physique et conceptuel d'analyse et de réalisation de logiciels. Plus récemment, la croissance rapide du marché des téléphones portables et des assistants a renforcé cette tendance vers une infrastructure de plus en plus distribuée et inter-connectée. Nous assistons ainsi à l'émergence d'un nouvel écosystème informatique, dans lequel les resources disponibles, à la fois en termes de performance d'exécution, mais aussi en termes de capacités de communication, sont omniprésentes mais très hétérogènes.

Face à l'évolution de la conjoncture relative à l'infrastructure matérielle, les évolutions au niveau des infrastructures logicielles, telles que les systèmes d'exploitation ou les langages, ne sont pas aussi rapides. Les langages les plus utilisés actuellement ont été créés pour la majorité d'entre eux avant que ces bouleversements technologiques ne soient complètement affirmé, et ne correspondent donc pas de par leur conception et leurs principes aux problématiques actuelles[64]. Particulièrement, la gestion de l'aspect distribué et la gestion des dépendances entre les différentes entités constituent des problèmes majeurs lors de la conception de systèmes répartis. C'est pourquoi il est nécessaire de développer et proposer de nouveaux paradigmes et outils pour pouvoir mieux tirer partie des avancées technologiques et mieux répondre aux besoins applicatifs émergeants.

L'ingénierie des systèmes distribués Notre travail consiste à proposer de nouveaux concepts et outils pour faciliter la conception de systèmes répartis. Nous nous sommes donc principalement concentré sur l'ingénierie des systèmes distribués. La conception de tels systèmes a été l'objet de nombreuses études, qui ont abouti sur de multiples paradigmes tels que la

64. Nous mettons à part deux technologies : le langage Java, qui peut être considéré comme l'un des premiers langages à prendre en compte l'aspect distribué, avec les possibilités fournies par Rmi d'une part, mais aussi et surtout par le modèle de sécurité du code d'autre part, et Corba qui est un premier pas vers la conception d'applications réparties. Cependant, ces technologies restent dans le cadre de la programmation orientée objets, et ne définissent pas d'approche de haut niveau pour la conception de systèmes distribués.

Programmation Parallèle, le Méta-Computing, les bus à objets répartis ou encore les espaces de Linda. Une dernière approche, particulièrement récente, est la conception orientée agents. Cette approche se repose sur les concepts des systèmes multi-agents et apporte d'intéressantes réponses pour la conception de systèmes constitués d'entités en interaction.

Les systèmes multi-agents Le domaine des systèmes multi-agents se trouve à la croisée de nombreux autres domaines tels que l'Intelligence Artificielle Distribuée, la Théorie des Actes de Langages, mais aussi la Théorie des Organisations et bien évidemment le Génie Logiciel. Le concept de base de ce domaine est la notion d'agent, qui représente une entitée autonome capable de percevoir, de se représenter et d'agir sur son environnement. L'utilisation d'un seul agent revient à travailler sur le modèle de raisonnement de ce dernier. C'est un des objets de recherche en Intelligence Artificielle. Par contre, lorsque l'on dispose d'un ensemble d'agents, de nouvelles problématiques émergent : l'échange ou le partage d'informations et de connaissances, la coopération ou la collaboration entre plusieurs agents, l'organisation des agents, la délégation de tâches ou de responsabilités ... Ces problématiques sont traitées de manières très différentes en fonction de la nature des sociétés d'agents étudiées. Ainsi, en vie artificielle, les agents ont un comportement simple, par contre l'environnement est complexe et la taille de la population d'agents est importante. A l'inverse, dans le domaine des systèmes multi-agents, le comportement de chaque agent est complexe, et la population est peu nombreuse, rarement plus d'une centaine d'agents. C'est ce type de systèmes que nous avons utilisé pour améliorer l'ingénierie des systèmes distribués.

Les principaux défis Tout au long de nos travaux, nous avons peu à peu été amenés à identifier quelques défis actuels au sein de la communauté des systèmes multi-agents. Le premier, et le plus crucial, est l'absence de définition formelle de "ce qu'est un agent". Nous ne parlons pas ici d'une définition générale, qui pourrait couvrir l'ensemble des domaines utilisant la notion d'agent, mais juste d'une définition spécifique, qui puisse être globalement reconnue par l'ensemble de la communauté, pour les sociétés d'agents à gros grain. En effet, la programmation et la conception orientée multi-agents ne pourra, nous semble-t-il, se développer au niveau industriel tant que ce concept de base ne sera pas clairement défini. Si l'on établit un parallèle avec les technologies orientées objets, même si au début il y avait de fortes dissensions sur "ce qu'est un objet", des langages tels que SmallTalk, puis C++ et Java ont établi cette définition en marginalisant les langages à prototypes et en imposant les langages de classes. On constate aujourd'hui l'importance de cette entente sur les concepts de classe et d'objet avec le développement de la programmation orientée modèles, qui s'appuyant sur le langage de description neutre Uml, permet d'aller plus loin au niveau des concepts en faisant abstraction des *détails* d'implémentation.

Le deuxième défi, qui d'une certaine manière est induit par le premier, est la diversité des propositions à la fois sur les modèles cognitifs d'agents, mais aussi sur les modèles organisationnels. Cette diversité se retrouve ensuite au niveau des outils proposés pour la création d'application multi-agents. Il y a donc nécessité d'une convergence à la fois sur les concepts mais aussi sur les technologies pour que le domaine puisse passer de la phase académique à une phase plus industrielle. Malheureusement, ces transitions du monde universitaire au monde de l'entreprise, ne restent bien souvent que ponctuelles et n'aboutissent pas pour l'instant à une pénétration significative des technologies orientées agents dans l'entreprise. C'est dans le but de répondre, ne serait-ce que partiellement à ces défis, que nous avons concentré nos travaux à la fois sur les aspects fondamentaux, mais aussi sur des aspects plus pragmatiques (liés à l'ingénierie de tels

systèmes) trop rarement étudiés dans le domaine.

Contributions de nos travaux

Mes travaux ont débuté lors du stage de DEA sur "la notion de service dans les systèmes multi-agents". Ce travail m'a permis d'identifier et de définir notre modèle minimal générique d'agent : une coquille vide, dotée d'une compétence de communication et d'une compétence d'acquisition de compétences. Ce noyau permet de faire évoluer l'agent vers différents modèles d'agents et offre aussi d'intéressantes possibilités au niveau applicatif, particulièrement grâce au principe d'échange dynamique de compétence. L'introduction de la notion de compétence et des fonctionnalités de construction incrémentale des agents a entraîné d'importantes modifications à la fois sur la manière de concevoir des systèmes distribués, mais aussi sur l'implémentation de la plateforme MAGIQUE. Suite à ces améliorations, nous avons développé plusieurs applications avec cette nouvelle version de MAGIQUE, ce qui nous a amené à débuter une démarche méthodologique singulière, et nous a aidé à identifier certaines faiblesses et limitations de notre *framework*. Nous étions particulièrement gênés par l'absence de représentation des dépendances entre les compétences et par la rigidité de gestion des aspects organisationnels. En effet, les dépendances fonctionnelles se trouvaient enfouies dans le code des compétences, et les interactions se produisant entre les agents n'étaient ainsi pas explicitées dans le système.

C'est la raison pour laquelle nous avons commencé notre travail sur la notion d'interaction, ce qui nous a conduit à développer notre modèle de spécification exécutable de protocole d'interaction. L'objectif de ce modèle est de donner au concepteur une vue globale d'un protocole d'interaction, et de laisser le système interpréter cette description globale pour chacun des rôles impliqués dans l'interaction. Ce modèle explicite les différents *micro-rôles* participants à une interaction et définit l'ensemble des informations nécessaires à la description du protocole d'interaction : le flux des messages, les services nécessaires à leur traitement, les informations consommées et produites, ainsi que la nature des messages échangés. Un algorithme de transformation permet ensuite de scinder cette vue globale en un ensemble d'automates représentant les vues partielles qu'ont chacun des *micro-rôles* de l'interaction. Ces automates peuvent être interprétés par une compétence de gestion des protocoles d'interaction, dont doivent disposer les agents désirant participer au systèmes.

Au niveau organisationnel, MAGIQUE repose sur un modèle hiérarchique. De plus il ne peut y avoir qu'une organisation au sein d'une application. Cependant, le réseau de communication par défaut et son mécanisme de routage et de délégation de tâche en font un élément fondamental de MAGIQUE. Nous avons donc étudié cette notion d'organisation et identifié un certain nombre de fonctionnalités qui devraient être supportées par l'organisation. L'organisation peut constituer un réseau de communication par défaut, pour peu qu'on lui associe un algorithme de routage. L'organisation permet aussi de structurer la société d'agents, en groupant logiquement les agents selon leurs fonctionnalités. Il est ensuite possible d'ajouter les fonctionnalités d'annuaires, telles que les pages blanches pour la localisation d'agents ou les pages jaunes pour la recherche de services. Par ailleurs, dans le cadre d'applications où la prestation d'un service donné n'est pas dépendante de l'agent le possédant, l'organisation peut améliorer dynamiquement le routage des messages en créant des accointances entre les agents possédant de fortes dépendances fonctionnelles. En un sens, l'organisation peut être vue comme un gestionnaire d'accointances évolué dans ce cas. Pour l'ensemble de ces raisons, nous pensons que l'organisation est un élément de premier ordre au sein des systèmes multi-agents, et nous pensons que les différents modèles d'agents devraient réifier la notion d'organisation, plutôt que de gérer eux-mêmes les informa-

tions organisationnelles au sein de leur propre base de connaissances (ou autre représentation interne).

Parallèlement à ces travaux, les problématiques liées à l'implémentation, m'ont conduit à m'intéresser aux dernières évolutions des technologies orientées objets et plus particulièrement à l'approche orientée composants pour réaliser les compétences dans notre système. J'ai ainsi, dans le même esprit que l'approche MDA, séparé la description des services de leur implémentation. Pour cela, l'initiative DAML-S, m'a fourni un formalisme intéressant de description des interfaces de services, et nous avons pu grâce aux récents travaux sur les Services Webs, faire le lien vers les conteneurs de composants. Cependant, la *jeunesse* de DAML-S, mais aussi d'OWL, pénalise leur intégration à cause du manque d'outils, que cela soit au niveau de l'édition (création d'ontologies), de la validation ou de l'interprétation (moteurs d'inférence). La nouvelle version du standard WSDL apportera sûrement quelques facilités au niveau de la jonction entre DAML-S et WSDL. En outre, l'utilisation de langages XML pour les différentes descriptions du système facilite l'ouverture vers d'autres plateformes existantes. Particulièrement, la description des réseaux de Pétri générés pour chacun des micro-rôles pourrait être interprétée par des agents s'exécutant sur d'autres plateformes (même si dans ce cas, il n'y aurait probablement pas les mêmes possibilités au niveau de l'évolution dynamique de l'agent).

Prenant conscience de l'importance fondamentale de la notion d'interaction, j'ai ensuite amélioré ma démarche méthodologique, en plaçant cette analyse et réification des interactions au cœur de notre approche. Nous avons tenté de formaliser la démarche méthodologique que nous adoptons lors de la réalisation de systèmes multi-agents, en nous focalisant sur la phase de modélisation des interactions. L'objectif que nous poursuivons avec RIO est de fournir une couche facilitant et automatisant la gestion des protocoles d'interaction, ainsi que leur déploiement dynamique au sein de systèmes multi-agents s'exécutant. Ainsi, RIO propose quatre étapes pour la conception de systèmes multi-agents :

- l'analyse et la description des différents protocoles d'interaction, qui impliquent l'identification des micro-rôles, compétences et messages échangés,
- l'agrégation de ces protocoles d'interaction au sein de rôles composites, qui correspond à la création d'un rôle prenant part à différents protocoles,
- la définition des agents abstraits et de leur occurrence. Un agent abstrait peut regrouper un ensemble de rôles composites et constitue un *patron* d'agent. La définition des organisations et leur liaison avec les rôles composites ou micro-rôles s'effectue aussi lors de cette étape.
- le déploiement effectif dans le système, en associant les agents abstraits à des agents concrets du système s'exécutant, et en précisant les liaisons entre les interfaces de compétences et leur implémentation.

La méthodologie RIO, pour *Rôles Interactions et Organisations*, combinée à la plateforme G que nous développons, définit une nouvelle approche de la conception de systèmes distribués. Le cycle habituel de création d'un logiciel, qui débute par une phase d'analyse, suivie d'une phase de conception, puis d'implémentation et enfin de maintenance, est une approche dont le but est de fournir un logiciel monolithique dont l'évolutivité est assurée par la production de versions successives (parfois non totalement compatibles). A cette approche *itérative*, nous substituons une approche *incrémentale* ou *continue*, consistant non plus à considérer une création de logiciel, mais à fournir un système flexible pouvant être adapté et enrichi au fur et à mesure de l'évolution des besoins. Ceci est rendu possible grâce à la dynamicité de notre modèle d'agent, et à une gestion fine des dépendances fonctionnelles (via la réification des interactions) et structurelles (via le modèle de composant choisi).

Perspectives

La première des tâches qu'il nous reste à effectuer est l'achèvement de la première version stable de notre plateforme G. L'objectif est ici de pouvoir déployer des spécifications de protocoles d'interaction et de vérifier la fiabilité de la plateforme, notamment au niveau des différents sous-systèmes la constituant (montée en charge de JAS, de la base de donnée, de l'agent `AgentPlatform`, et d'OSCAR le conteneur OSGI).

La première validation consistera à redévelopper le framework RAGE avec G. Cette application est intéressante à la fois de par sa complexité, qui illustre les principaux concepts de nos propositions, mais aussi de par son utilité pratique. Nous avons travaillé sur un autre *framework* dont le domaine d'application est la simulation, et la phase de visualisation demande de lourds calculs[65]. Effectuer l'ensemble de ces calculs avec RAGE, au dessus de G, illustrerait la stabilité de notre plateforme.

La deuxième validation consistera à joindre le réseau OPENNET de plateformes respectant les normes de la FIPA. Enfin, la *vraie* validation aura lieu lorsque nous diffuserons la première version publique de notre plateforme, et que nous essayerons d'établir une communauté d'utilisateurs. Mais pour cela, en plus des problèmes liés à l'implémentation, il nous faudra au préalable avoir rédigé divers documents tels que des guides pour les utilisateurs et les développeurs.

Après avoir présenté une vue de nos objectifs des mois à venir, nous allons détailler plus spécifiquement, les perspectives que nous envisageons pour les différentes propositions effectuées dans le cadre de cette thèse.

Au niveau du modèle générique minimal d'agent : Une première piste de travail consistera à améliorer et préciser notre modèle d'agent, particulièrement au niveau de la gestion des organisations et la gestion de la base de connaissances. En effet, nous n'avons pas encore clairement identifié les différents niveaux permettant d'articuler la représentation logique avec l'infrastructure de communication physique. Nous aimerions à terme pouvoir rajouter ou enlever dynamiquement de nouveaux protocoles au niveau des plateformes sans que le niveau applicatif en soit conscient. Par ailleurs, au niveau de la base de connaissances, nous aimerions nous en servir comme un espace de Linda, et ainsi préciser la nature des dépendances *implicites* qui se produisent entre les différentes compétences d'un agent. Pour cela, nous pensons utiliser OWL, maintenant que quelques moteurs d'inférences commencent à être disponibles (JTP ou PELLET).

Une deuxième piste consisterait à implémenter différents modèles d'agents pour fournir aux concepteurs plus de souplesse et de possibilités. Notre travail s'est en effet principalement concentré sur la partie "système" de l'agent, avec les compétences de gestion des protocoles d'interaction et des organisations. Il faudrait maintenant préciser les fonctions déléguées à la base de connaissances, et réimplémenter un modèle de type BDI. Il serait intéressant aussi de porter le modèle d'agent de JADE car il est maintenant bien diffusé.

Au niveau du modèle d'interaction : Plusieurs améliorations doivent être apportées au niveau du modèle d'interaction. Le plus impératif est de figer la structure du fichier XML décrivant les protocoles. Pour cela, il faudra ajouter la gestion des pré et post-conditions sur les

65. Brièvement, la phase de visualisation consiste à générer des animation en *ray-tracing* à partir des données de la simulation. Ce type de calcul s'adapte tout à fait au modèle de tâche de RAGE puisque les images peuvent être calculées indépendamment. D'autre part, cela illustrerait aussi les possibilités d'utilisation de code natif fournies par OSGI pour le calcul du rendu avec POVRAY.

nœuds du graphe d'interaction, et aussi améliorer le mécanisme de gestion des *timeout*. Nous pensons ensuite étudier la possibilité de projeter les protocoles vers AgentUml.

Nous nous interrogeons aussi sur l'opportunité d'un outil de simulation des protocoles d'interaction. Le principe de cet outil serait d'instancier un protocole donné, et de le faire tourner *à vide* pour vérifier le bon déroulement du protocole. D'un point de vue plus formel, nous pourrions étudier les travaux effectués sur les problématiques d'accessibilité et de recherche de *deadlock*s dans les Réseau de Pétri Colorés.

Il serait aussi très important de proposer une librairie de protocoles d'interaction génériques. Le principe serait de constituer des *template* de protocoles d'interaction que le concepteur pourrait ensuite spécialiser en affinant le typage des motifs de message, et en adaptant les interfaces de service utilisées.

Conjointement au développement de cette librairie, et pour rendre son utilisation plus pratique, il faudra développer un éditeur graphique de protocoles d'interaction. Nous avons de ce côté là effectué une première étude sur l'Api JGraph et sur le *framework* JGraphPad, qui facilitent le développement d'outil de conception graphique de graphes.

D'autre part, depuis mai 2003, un groupe de travail a été créé au sein de la Fipa pour étudier et produire une spécification sur les protocoles d'interaction. Cette spécification est fondamentale, car elle serait la première à dépasser les aspects bas niveau d'intéropérabilité. Nous suivons donc les échanges actuels, mais faute de temps, nous n'avons pas encore pu nous impliquer dans le *Interaction Protocol Technical Comitee*.

Au niveau de la plateforme : L'implémentation actuelle utilise toujours des classes Java *classiques*. Nous devons valider notre approche en ajoutant les possibilités d'utilisation de composants de type Ejb, mais surtout OSGi. Pour les Ejb, l'intégration devrait être simple car c'est l'une des possibilités offertes par Wsif. Par contre, pour les composants OSGi, il faudra développer un connecteur pour établir la liaison entre Wsif et Oscar.

Nous pensons aussi développer des outils pour faciliter l'administration de la plateforme. Les premières expérimentations effectuées consistent à utiliser une approche de type client/serveur, et de fournir une interface web d'administration. Cependant, la première approche qui consistait à utiliser des *servlets* s'est avérée peu pratique et difficile à maintenir lorsque le nombre de fonctions augmentent. Nous nous sommes donc orienté vers une *applet* dont l'interface est décrite avec un langage Xul[66].

Une extension de cette plateforme d'administration serait l'intégration d'un environnement de développement des protocoles d'interactions et des interfaces de services. L'aspect le plus intéressant dans ce projet serait l'utilisation de la plateforme elle-même (et donc d'un système multi-agents) pour le développement, le déploiement et la supervision du système.

Enfin, une dernière validation pourra avoir lieu lorsque nous joindrons le réseau openNet. Cependant, mis-à-part la vérification des protocoles de communication de bas niveau (qui dépendent de Jas dans notre implémentation), il faudrait trouver des partenaires pour faire fonctionner des applications sur différentes plateformes. Si le projet openNet réussit à devenir un réseau d'excellence européen (la demande est en cours), nous pourrons apporter une intéressante contribution avec notre plateforme qui est à notre connaissance, l'une des seules à se reposer intrinséquement sur les technologies du Web Sémantique (l'un des axes de recherche mis en avant dans le document de référence d'openNet).

66. Nous utilisons ce langage : `http://www.thinlet.com`

Au niveau de la méthodologie : Au niveau de la méthodologie, nous pensons que la disponibilité des outils de conception des protocoles d'interaction et déploiement permettra de préciser les différentes étapes de notre démarche. Comme nous l'avons vu précédemment, l'analyse et la conception de Rage avec Rio permettront de disposer d'un premier exemple complet d'utilisation de Rio.

De plus, comme avec le modèle d'interaction, un nouveau groupe a été créé à la Fipa pour étudier les méthodologies existantes et tenter d'unifier les propositions au sein d'une spécification. Nous avons suivi partiellement les discussions en cours, et l'un des apports directs, qui nous paraît le plus important, est la production *indirecte* d'une ontologie sur les systèmes multi-agents et leur développement. Nous aimerions donc aussi nous impliquer au sein du *Methodology Technical Comitee* de la Fipa, et proposer d'aller jusqu'au bout de la démarche initiée en formalisant cette ontologie en Owl. Il deviendrait ainsi possible de l'utiliser au sein de la plateforme, en interne, pour interagir avec le système. Cette ontologie pourrait aussi fournir un outil de description pour les modèles d'interaction et d'organisation.

Epilogue

Je voudrai terminer ce document avec un extrait de la conclusion du livre *Les langages à objets*[53], qui caractérise assez justement (et inopinément) l'exercice académique que représente le document de thèse :

> "*Il faudrait encore de nombreuses pages pour aborder tous les problèmes encore en suspens et examiner les solutions proposées. Toutefois,* une thèse[67] *reste un outil de diffusion de la pensée et ses moindres qualités sont la maniabilité et un poids raisonnable. C'est pourquoi nous nous arrêterons là, pour ne pas fatiguer le lecteur, non par des problèmes théoriques ardus, mais par le poids trop conséquent de l'outil qu'il a dans les mains ...*"

67. *un livre* dans le texte original.

Annexe A

Publications effectuées dans le cadre de la thèse

Journaux

Using agents to build a distributed calculus framework
Philippe Mathieu, Jean-Christophe Routier, Yann Secq
The Interdisciplinary Journal of Artificial Intelligence and the Simulation of Behaviour

Conférences Internationales

Dynamic Skill Learning: A Support to Agent Evolution
Philippe Mathieu, Jean-Christophe Routier, Yann Secq
Proceedings of the AISB'01 Symposium on Adaptive Agents and Multi-Agent Systems, page 25-32

RAGE : An agent framework for easy distributed computing
Philippe Mathieu, Jean-Christophe Routier, Yann Secq
Proceedings of the AISB'02 Symposium on Artificial Intelligence and Grid Computing, selected as "Best Paper"

Dynamic organization of multi-agent systems (Poster)
Philippe Mathieu, Jean-Christophe Routier, Yann Secq
Proceedings of the first international joint conference on Autonomous Agents and Multi-Agent Systems (AAMAS'02)

Ubiquitous Computing : vanishing the notion of application (Poster)
Philippe Mathieu, Jean-Christophe Routier, Yann Secq

UbiAgents'02 : AAMAS Ubiquitous Agents Workshop

Principles for Dynamic Multi-Agent Organizations
Philippe Mathieu, Jean-Christophe Routier, Yann Secq
5th Pacific Rim International Workshop on Multi Agents (PRIMA 2002)
Intelligent Agents and Multi-Agent Systems, Lecture Notes in Computer Science 2413

RIO : Roles, Interactions and Organizations
Philippe Mathieu, Jean-Christophe Routier, Yann Secq
The 3rd International/Central and Eastern European Conference on Multi-Agent Systems (CEEMAS 2003)
Multi-Agent Systems and Applications III, Lecture Notes in Artificial Intelligence 2691

Bridging the Gap Between Semantic and Pragmatic
Philippe Mathieu, Jean-Christophe Routier, Yann Secq
Proceedings of the 2003 International Conference on Information and Knowledge Engineering (IKE'03)

Runnable Specifications of Interactions for Open Multi-Agent Systems
Philippe Mathieu, Jean-Christophe Routier, Yann Secq
Proceedings of the 2003 International Conference on Information and Knowledge Engineering (IKE'03)

Towards a pragmatic use of ontologies in multi-agent platforms
Philippe Mathieu, Jean-Christophe Routier, Yann Secq
The 7th International Conference on Knowledge-Based Intelligent Information and Engineering Systems, Invited Session Ontology and Multi-Agent Systems Design (OMASD'03)

Towards a pragmatic methodology for open multi-agent systems
Philippe Mathieu, Jean-Christophe Routier, Yann Secq
14th International Symposium on Methodologies for Intelligent Systems

Conférences Francophones

RIO : Rôles, Interactions et Organisations
Philippe Mathieu, Jean-Christophe Routier, Yann Secq
Secondes Journées Francophones sur les Modèles Formels de l'Interaction (MFI'03)

Annexe B

Le “nouveau chapitre” de la thèse

Après leur thèse, les jeunes docteurs sont confrontés au marché du travail - qu'il s'agisse du secteur public ou privé - où leur aptitude à la recherche est évaluée en parallèle à de multiples autres critères propres à chaque établissement ou entreprise. Une réponse : la “Valorisation des compétences - un nouveau chapitre de la thèse”.

Pour faciliter l'insertion professionnelle de leurs doctorants, trois écoles doctorales en Sciences de l'Univers d'Ile de France et de Toulouse, en partenariat avec l'Institut National des Sciences de l'Univers du Cnrs (Insu) et l'Association Bernard Gregory (Abg), ont imaginé en 2000 le concept du “nouveau chapitre de la thèse”.

L'idée force de ce ”nouveau chapitre” est d'encourager les doctorants à préparer leur ”après-thèse” en les aidant à faire le point sur les compétences et savoir-faire professionnels développés au cours de leurs trois années de recherche. Une chose est en effet de savoir que dans l'absolu, les jeunes docteurs ont acquis des compétences très intéressantes pour leur insertion dans la vie active, que ce soit dans l'univers de l'entreprise ou dans celui de l'enseignement supérieur et de la recherche académique. Mais passer de cette observation générale à son appropriation personnelle constitue encore souvent, pour le doctorant, un véritable “challenge” ; au-delà des compétences évidentes qu'il sait directement héritées de son expérience de thèse (être un bon chercheur, pour faire simple), il peut éprouver des difficultés à discerner et, par conséquent, à mettre en avant d'autres acquis tout aussi fondamentaux, tels que la conduite de projet, la gestion du temps, le montage de partenariats, etc...

Avec le nouveau chapitre, les doctorants en fin de thèse font une analyse critique de la manière dont ils ont conduit et géré leur projet de recherche, et en tirent des conclusions quant aux qualités personnelles et aux savoir faire qu'ils ont développés durant leur thèse, sachant que la valorisation de ces aptitudes leur servira pour leur insertion professionnelle.

L'une des originalités du nouveau chapitre réside dans le fait que les doctorants qui se prêtent à l'exercice ne sont pas livrés à eux-mêmes face à la page blanche. Des “mentors”, qui sont des consultants spécialistes du recrutement formés par l'Abg à l'encadrement de ce travail, les accompagnent et les guident dans cette démarche, et leur regard extérieur aide considérablement les doctorants à identifier et à valoriser les acquis de leur thèse.

Cette analyse doit déboucher sur un document aisément compréhensible par des non-spécialistes du domaine de recherche, d'une longueur de 5 à 6 pages environ, dont le canevas est présenté dans l'appel à propositions qui a été adressé à toutes les écoles doctorales.

Présentation de l'expérience de doctorant recherche, enseignement et administration

Doctorant : Yann Secq, Accompagnatrice : Florence Breuzard
Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille

Présentation du sujet de thèse

L'évolution des réseaux informatiques, suite au développement du marché des ordinateurs personnels et d'Internet, a amené de profondes modifications quant à la manière de concevoir et de développer des applications. La notion d'application est passée de celle d'un programme s'exécutant sur un ordinateur, à un ensemble de programmes répartis sur un réseau de machines. Ainsi, l'environnement logiciel qui s'apparentait à un ensemble de *programmes autistes* ne pouvant communiquer entre eux, tend à devenir un ensemble de ***systèmes ouverts* en *interaction*** les uns avec les autres. Concevoir de tels systèmes distribués pose alors le problème de la gestion des interactions. Cela revient à définir comment les différentes composantes du système peuvent s'identifier, à décrire leurs fonctionnalités, à spécifier les interactions possibles entre ces entités, et à préciser les modalités leur permettant d'entrer en contact les unes avec les autres.

Cette description pourrait correspondre au monde de l'entreprise actuel : chaque jour des entreprises se créent et proposent des services ou des produits à leurs clients. Pour cela, elles doivent s'identifier et communiquer sur leur offre, elles doivent aussi trouver les entreprises qui interviendront comme partenaires (fournisseurs ou clients), et s'accorder avec elles sur leurs échanges d'informations. De plus, ces échanges d'informations, bien que structurés, reposent aussi souvent sur un ensemble de non-dits, de sous-entendus, ou plus précisément de conventions tacites, qui ne peuvent s'exprimer aisément dans la rigueur des langages informatiques.

Par conséquent, les langages permettant de créer des logiciels ont dû prendre en compte cette nouvelle dimension. Les premiers langages étaient appelés *langage machine*, et demandaient au développeur une connaissance de la structure interne de l'ordinateur. Ensuite, chaque nouvelle famille de langage a apporté un niveau d'abstraction supplémentaire, éloignant le développeur des considérations matérielles.

Cependant, bien que les langages aient évolué, ils n'ont malheureusement pas proposé de modèle complet d'analyse, de conception, de réalisation et de maintenance pour les systèmes ouverts distribués. La faiblesse des propositions actuelles provient du faible niveau d'abstraction de leurs concepts, et du manque de méthodologie prenant en compte l'ensemble du cycle de vie des logiciels distribués. **Construire un système ouvert distribué de nos jours**, en utilisant les technologies actuelles, s'apparenterait à la conception d'un gratte-ciel à l'aide d'un papier et d'un crayon !

Le but de mes recherches est de **contribuer au développement de nouvelles technologies "orientées agents"**, qui remporteraient le même succès qu'ont connu les technologies "orientées objets" avec le langage *SmallTalk*, première rupture dans l'histoire des langages informatiques.

Le principe de la thèse

Les systèmes multi-agents sont des systèmes constitués d'un ensemble d'entités autonomes appelées agents. Ils peuvent apporter des réponses qui aident à maîtriser la complexité des systèmes distribués. Ce domaine est né à l'intersection de domaines tels que l'intelligence artificielle distribuée, la représentation des connaissances, la théorie des actes de langages ou encore la vie artificielle. Et bien que la notion de système multi-agents soit assez ancienne, elle n'a connu un fort développement scientifique que depuis la fin des années 1980. L'avénement d'Internet, et plus encore l'apparition du commerce électronique, a amené les industriels à s'intéresser plus particulièrement à cette approche. Ainsi, au milieu des années 1990, de nombreuses entreprises privées ont proposé leur propre système : IBM, British Telecom, Fujitsu, Toshiba ou même Boeing. D'un autre côté, la recherche publique a aussi produit de nombreux systèmes, sans qu'aucune définition formelle de "*ce qu'est un agent*" n'apparaisse.

Par rapport à l'ensemble de ces influences, mon travail se place principalement dans le cadre du génie logiciel et consiste à **proposer une méthode, des concepts et des outils facilitant la conception, la réalisation et la maintenance de systèmes distribués**.

Ainsi, une des premières tâches fut l'identification des fonctionnalités communes aux différents modèles d'agents existants. Cela m'a amené à définir la notion de *compétence* comme brique de base de la construction d'agent (une compétence est un composant logiciel regroupant un ensemble de fonctionnalités). Ensuite, le développement de plusieurs applications avec ce principe m'a amené à placer la notion d'*interaction* au centre de l'analyse de tels systèmes complexes. En effet, l'aspect le plus difficile à gérer dans les systèmes distribués est la gestion de la synchronisation. J'ai donc proposé un modèle qui explicite les *interactions* entre différents *rôles*, et automatise le déploiement de ces *interactions* dans un système multi-agent. Enfin, la notion d'*organisation* est nécessaire pour lier les agents en fonction de leur *rôles* et de leurs *interactions*.

Le contexte dans lequel s'est déroulé mon travail de thèse

Le **Laboratoire d'Informatique Fondamentale de Lille** (LIFL) est structuré autour de trois axes : le premier s'attache aux recherches fondamentales (vérification, calcul formel), tandis que le deuxième et le troisième se concentrent sur des recherches plus appliquées. Les principaux thèmes de ces deux axes sont la bio-informatique, l'optimisation, le parallélisme et les systèmes distribués, les technologies objets, les cartes à puces, et le graphisme.

L'équipe Systèmes Multi-Agents et Coopération (SMAC), dans laquelle j'effectue ma thèse, est une jeune équipe. Elle a été créée par le Professeur Philippe Mathieu, rapidement rejoint par Jean-Christophe Routier et Bruno Beaufils. Les principales thématiques de recherche sont la conception de systèmes multi-agents, la modélisation de comportements et la théorie des jeux. L'objectif à long terme est de proposer des plateformes de développement de systèmes multi-agents, ainsi que des simulateurs comportementaux. Malgré sa jeunesse, l'équipe est fortement impliquée au niveau national, et à une moindre échelle au niveau européen.

Au niveau national, notre participation à un groupe de travail consacré aux systèmes multi-agents au sein de l'*Association Française d'Intelligence Artificielle*, facilite l'identification de notre équipe. Cette reconnaissance est d'autant plus renforcée cette année avec **l'organisation des *Journées Francophones d'Intelligence Artificielle Distribuée et des Systèmes Multi-Agents*, principale conférence de notre domaine**. De plus, **l'organisation de**

la conférence *Modèles Formels de l'Interaction* en mai 2003, renforcera la visibilité de l'équipe.
Au niveau européen, **l'équipe SMAC appartient au réseau d'excellence AgentLink** depuis 3 ans. C'est un réseau européen bénéficiant d'un financement de l'IST (*Information Society Technology*), visant à coordonner et regrouper les universitaires et les industriels intéressés par les technologies agents, et favoriser ainsi le développement industriel des technologies orientées agents. Par ailleurs, en ce qui concerne les collaborations avec des entreprises privées, l'équipe a travaillé sur un important projet financé par le *Centre National de la Cinématographie* et l'entreprise *Cryo Interactive*, visant à établir un prototype de plateforme de simulation de comportements pour les jeux vidéos. Cette collaboration a été fructueuse, l'un des étudiants ayant participé au projet a intégré le département R&D de *Cryo Interactive*.

Suite à un projet de Maîtrise, je pensais m'orienter vers l'équipe bio-informatique. L'aspect qui me passionnait le plus dans cette thématique était la pluridisciplinarité. Malheureusement, suite à une réunion regroupant des chercheurs de l'Institut Pasteur et du LIFL, je me suis rendu compte de l'état plus qu'embryonnaire de cette collaboration. De plus, lors de la première partie du DEA, le cours de Mr Mathieu sur l'intelligence artificielle et les systèmes multi-agents m'a passionné; ses qualités humaines et sa disponibilité m'ont convaincu d'effectuer mon stage au sein de son équipe.
L'intitulé du stage était : "*la notion de service dans les systèmes multi-agents*". Depuis, l'évolution des travaux a découlé naturellement de ce sujet au travers de nombreuses discussions avec Mr Mathieu, mon directeur de thèse, et Mr Jean-Christophe Routier, mon co-directeur. En effet, **j'ai profité d'une totale autonomie dans la définition de mes objectifs et de leur réalisation**. Mon poste de travail se trouvant dans le bureau de Mr Mathieu et Mr Routier, j'ai pu informer quasi continuellement mes encadrants de l'état de mes recherches et bénéficier ainsi de leurs remarques, critiques et conseils, qui ont toujours été constructifs.
De plus, mes encadrants sont fortement impliqués dans l'enseignement et en gestion de formation. Ils m'ont ainsi accompagné dans ma fonction d'enseignant. Ils ont été aussi très compréhensifs vis-à-vis de l'investissement en temps que cela m'a demandé. En outre, la gestion d'équipe de Mr Mathieu est très transparente et implique les personnes. J'ai donc pu appréhender la gestion d'une équipe de recherche. J'ai particulièrement constaté les difficultés qu'il pouvait y avoir à gérer l'aspect financier et comptable d'une équipe de recherche, au sein d'un laboratoire qui est finalement jeune, et où les infrastructures administratives sont très faibles. Ceci implique qu'en plus du travail de recherche, d'enseignement et de gestion de formation, un chef d'équipe doit avoir maîtriser les relations avec les différents organes comptables ou juridiques de l'administration centrale de l'Université.

Les compétences acquises ou développées

Explorer et exploiter

Mon sujet de thèse étant ouvert, j'ai été amené à effectuer **une veille technologique** sur différents domaines. Je me suis ainsi intéressé aux technologies orientées objets (principalement le langage Java et ses extensions, les modèles de composants), aux systèmes distribués (RMI, CORBA, le Peer-To-Peer), ainsi qu'aux technologies proposées par le W3C, tels qu'XML, XSLT, et aussi les nouveaux langages pour le Web Sémantique, RDF et DAML. Ceci m'a amené à

développer un regard critique sur ces technologies, ainsi que sur les critères de leur utilisation (maturité du produit, processus de standardisation, importance de la communauté d'utilisateurs, réactivité des développeurs ...). En effet, si l'on désire qu'une technologie soit acceptée dans le monde industriel, il est important de connaître celles qui existent actuellement, et la façon dont elles sont utilisées en entreprise.

Du point de vue scientifique, j'ai étudié diverses thématiques, comme celle du *génie logiciel* (cycle de développement, outils de gestion de qualité, modèles de composants et architecture de logiciels objets), du *calcul distribué*, de la linguistique avec *la théorie des actes de langages*, de la sociologie avec les *théories organisationnelles*, et un peu de *théorie des jeux*. Ce travail m'a permis d'évaluer leur intérêt pour mon sujet de recherche, bien que je ne sois pas devenu spécialiste de chacun de ces domaines. Je crois que ce travail d'exploration à la fois technologique et scientifique provient à la fois de ma curiosité naturelle, mais aussi de la nature de mon sujet de thèse et du domaine étudié.

D'autre part, j'ai appris à rédiger des articles scientifiques en anglais, et à les présenter lors de conférences. Plus encore que les compétences linguistiques, la prise de conscience de **l'importance du discours** m'est apparu : certes le travail en lui même est important, mais sa présentation et sa justification le sont encore plus.

Enseigner, encadrer et représenter

D'autre aspects intéressants de la thèse sont l'enseignement et l'encadrement de projets et de stages. Malgré une transition plutôt rapide entre le statut d'étudiant et celui d'enseignant lors du DEA, l'exercice qu'est l'enseignement est vraiment passionnant. Heureusement, j'ai bénéficié de stages de pédagogie en tant que moniteur dans le cadre du Centre d'Initiation à l'Enseignement Supérieur. Ces stages m'ont permis de prendre conscience des enjeux, des techniques et des problématiques liés à la **transmission du savoir**. J'ai effectué des enseignements en DEUG, en première et deuxième année d'IUT, ainsi qu'en formation continue. Cette dernière expérience fut aussi très intéressante : l'approche étant vraiment différente lorsque vous devez enseigner à des personnes qui pourraient être vos parents !

Parallèlement aux enseignements (en Travaux Dirigés et en Travaux Pratiques), j'ai pu aussi encadrer un certain nombre de projets que les étudiants avaient à réaliser dans le cadre de leur formation. Ces projets m'ont donné la possibilité, d'une part de satisfaire ma curiosité en explorant des problématiques informatiques qui ne sont pas centrales dans mes recherches, d'autre part de partager une relation pédagogique différente avec les étudiants, beaucoup **plus interactive et enrichissante qu'en situation d'enseignement classique**. J'ai ainsi réalisé l'intérêt d'une approche basée sur une discussion ouverte lors des choix d'organisation du projet ou des choix de conception. L'ensemble des personnes étant impliqué dans les différents aspects de la gestion du projet, elles s'investissent plus facilement et remettent moins aisément en cause les choix effectués, puisqu'elles y ont pris part.

De plus, j'ai dû faire preuve de patience et de persévérance pour mener chacun de ces projets à terme. Le premier projet que j'ai encadré m'a appris l'importance d'un suivi constant, ainsi que la nécessité d'un cadrage explicite des objectifs et des méthodes. J'ai aussi découvert l'importance de l'attention particulière qu'il faut porter à chacun, afin qu'il soit impliqué dans le projet.

D'un point de vue plus technique, ces projets m'ont aussi sensibilisé à l'importance des outils de gestion de projets, qui peuvent être utilisés comme support à la gestion de la communication, ou encore à la gestion de la qualité. Ces projets interviennent dans différentes formations (DUT, école d'ingénieurs EUDIL, DESS ou Maîtrise) et dans différents cycles (premier, second et

troisième cycle, de bac+1 à bac+5). Ils m'ont permis de me rendre compte **du potentiel dont disposait l'Université en terme de compétences et de force de travail disponibles**.

Enfin, j'ai encadré des stages de fin d'étude en DUT d'Informatique. Cette activité est importante à différents titre : pour les étudiants d'abord, car cela leur donne leur première expérience professionnelle; pour les enseignants ensuite, car cela leur permet de mieux connaître les besoins des entreprises et donc d'avoir quelques éléments permettant de juger de l'adéquation des enseignements avec les besoins des entreprises; pour la formation enfin, car c'est **un outil important de communication à destination des entreprises**. L'encadrement de stage est totalement différent de l'encadrement de projet : le rôle de l'enseignant est plus proche d'une fonction de soutien et de médiation, le tuteur en entreprise ayant à sa charge les aspects techniques ou propres à l'entreprise.

L'Université vue de l'intérieur

Outre mon travail de recherche, j'ai appris à connaître mon environnement et à mieux cerner ses avantages et ses inconvénients. Depuis l'entrée à l'Université, et mon implication dans le milieu associatif (théâtre, puis association de formation avec l'Association des Etudiants en Informatique), j'ai pu découvrir l'Université sous un autre aspect que celui d'un lieu d'étude. C'est un espace ouvert, où il y a un potentiel important, bien qu'il ne soit pas toujours judicieusement exploité. De plus, en participant à la vie de l'établissement à différents niveaux, *Conseil de Laboratoire*, *Conseil d'Unité de Formation et de Recherche* et *Conseil Scientifique* de l'Université, j'ai découvert le fonctionnement interne de l'Université. Ceci m'a éclairé sur l'importance de la RELATION DIRECTE avec les différents interlocuteurs, qui facilite toujours l'avancement du dossier ou du projet, ainsi que **la nécessité d'être force de proposition**.

Les enseignements principaux de cette expérience

Pour présenter ce que je retire de cette expérience qu'est la thèse, il me paraît judicieux d'expliquer pourquoi faire une thèse est formateur pour la vie professionnelle, ce que cela implique comme apprentissage, et finalement ce qui pourrait être amélioré au niveau de l'environnement de la thèse, pour que cette expérience puisse être encore plus enrichissante. Chaque thèse étant unique, je ne peux présenter ici que mon expérience propre de doctorant.

En effet, dans mon cas, la thèse a été un moment singulier de ma vie d'étudiant : celui de **la transition entre le statut d'élève à celui d'enseignant**. Ceci implique que le doctorant est à la fois un apprenti, vis-à-vis de ses encadrants lors des travaux de recherche, et un maître, vis-à-vis des étudiants qu'il doit former. Cela entraîne une prise de conscience à la fois sur le passé, mais aussi sur l'avenir : sur les études qu'il a fallu mener pour arriver en thèse, et sur le chemin qu'il reste à parcourir pour acquérir un métier.

Le principal enseignement est la découverte des dimensions multiples du monde du travail.

L'Université, comme tout autre organisation humaine, est un ensemble complexe de réseaux de relations. Les plus évidentes de ces relations sont explicites et constituent l'infrastructure de communication. L'Université est un lieu de formation et de recherche, la formation implique la présence d'enseignants et d'étudiants, interagissant selon certains protocoles établis; il en est de même avec la recherche.

Cependant, en s'impliquant dans les différents espaces de l'Université (recherche, pédagogie, gestion de formation, gestion de l'Université, milieu associatif ...), **on découvre rapidement**

d'autres réseaux, implicites (en ce sens qu'ils ne sont pas rattachés à une fonction de l'Université), et dont l'importance ne peut être négligée. En effet, l'Université reposant sur la liberté et l'auto-gérance, ces réseaux sont les principaux lieux qui font progresser le système. Ainsi, il n'est pas rare d'être en relation avec la même personne dans différentes instances et selon différents rapports. Ceci illustre une particularité de l'Université : il faut faire preuve de beaucoup de diplomatie, et prendre en compte les objectifs et les contraintes de chacun, **pour réussir non pas à imposer des décisions, mais à les faire accepter par la majorité des participants**.

En ce qui concerne le travail de thèse en lui même, on peut regretter un certain nombre de choses, et principalement **le manque d'outils**. Que cela soit pour la gestion bibliographique, ou pour la méthodologie de rédaction de documents scientifiques, il y a de fortes lacunes. Les écoles doctorales, si elles poursuivent leurs efforts pourraient être le principal lieu d'amélioration des conditions de travail des doctorants. D'autre part, la généralisation du monitorat (mesure qui devrait être effective en 2004), permettrait à chaque doctorant d'appréhender l'enseignement et la pédagogie, et améliorerait aussi considérablement sa situation matérielle. N'est-il pas aberrant que **la majorité des enseignants-chercheurs de demain n'ait eu aucun cours de pédagogie ?** Sur ce dernier point, il me semble qu'un rapprochement entre les écoles doctorales des différentes universités et le CIES (organisme gérant le monitorat) permettrait de donner cohérence à la formation du doctorant, avec des stages orientés pédagogie grâce au CIES et des formations plus méthodologiques pour les écoles doctorales.

Un autre regret concerne **le manque de valorisation du document de thèse par les universités**. Il est surprenant de constater que la quasi-totalité des thèses est rédigée sous forme numérique, mais que leur archivage se fait encore sous forme de micro-fiches. Bien que l'archivage sous forme électronique se généralise, les facilités de diffusion que propose ce support ne sont pas encore développées. Ce problème de la valorisation des thèses soulève, il me semble, une autre lacune importante de l'Université : **la gestion de l'innovation**. C'est d'autant plus regrettable, car **le potentiel en terme d'encadrants ou de force de travail (via les projets ou stages) est présent**, il manque juste une volonté politique et une équipe qui soutiendrait et pérenniserait ces actions.

Cependant, malgré les améliorations qu'il est toujours possible d'apporter, **l'Université reste le seul lieu permettant d'étudier et de s'investir pendant plusieurs années sur un sujet de recherche**. Ce système arrive à maintenir un équilibre entre les besoins de formation, qui sont fondamentaux pour le monde de l'entreprise, et les besoins de recherche, qui participent au **rayonnement scientifique** mais aussi aux **innovations technologiques**.

Annexe : Evaluation partielle du coût de la thèse

En informatique, pour pouvoir effectuer une thèse, il faut obtenir un bon classement en DEA, ce qui permet d'obtenir un financement. Ce financement peut prendre différentes formes, mais dans mon cas, ce fut une bourse MESR (attribuée par le Ministère de la Recherche), à laquelle s'est ajoutée une allocation d'enseignement, le monitorat. Le monitorat est un contrat de formation et d'enseignement : le doctorant qui l'accepte s'engage à suivre des formations portant principalement sur la pédagogie (20 jours à répartir sur deux années), et à enseigner en premier

cycle un volume de 64h de travaux dirigés par année. Ces financements prennent la forme d'un CDD d'un an, renouvelable deux fois au maximum.

Salaire : 14400 euros/an (hors charges patronales)	43200 euros
Matériel : stations de travail pour 3750 euros/an	11250 euros
Déplacements : Université d'été et conférences	5000 euros
Soit un total sur 3 ans de	59450 euros

A cette somme s'ajoute un certain nombre de dépenses dont le montant ne précis peut être facilement déterminé (car un laboratoire, ou même une Université, n'est pas géré comme une entreprise) :

- les stages du monitorat : 20 jours de formation à la pédagogie, et un déplacement d'une semaine à Grenoble pour les journées nationales du monitorat.
- l'encadrement *au fil de l'eau* de Messieurs Mathieu et Routier pour la recherche et l'enseignement.
- les services communs du laboratoire : le bureau, l'électricité, l'accès au réseau, au mail, au serveur web, aux ressources bibliographiques en ligne, aux services de reprographie, au téléphone et au fax, ainsi qu'aux services du secrétariat.

Ainsi, ma thèse a coûté au moins 60000 euros sur trois ans. Il est probable, que cette somme soit doublée si l'on ajoute l'ensemble des dépenses difficilement évaluables citées précédemment. On peut toutefois souligner le fait qu'une thèse en Informatique ne coûte pas très cher, en comparaison de thèses nécessitant des équipements d'expérimentation ou encore de projets industriels.

Index des auteurs

Cet index référence les auteurs cités dans cette thèse, il correspondre à la liste des auteurs se trouvant dans la bibliographie. Cependant, il facilite la localisation d'une référence dans le document, en fonction de son contexte d'utilisation.

Bibliographie

Cette bibliographie regroupe l'ensemble des références faite dans ce document. De plus, chaque entrée est accompagnée d'un résumé succinct des principales idées développées dans le document, ainsi que d'un pointeur vers le document sous forme électronique lorsqu'il est disponible sur Internet.

[1] Department of Computer Science and Engineering, University of California, San Diego.

[2] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Syste*. The MIT Press Series in Artificial Intelligence, 198.

[3] Gul Agha, Akinori Yonezawa, Peter Wegner, and Samson Abramsky. Foundations of object-based concurrent programming (panel session). In *Proceedings of the European conference on Object-oriented programming addendum : systems, languages, and applications*, pages 9–14. ACM Press, 1991.

[4] DAML Services Coalition (alphabetically A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. Daml-s: Semantic markup for web services. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, 2001.

[5] R. Baecker, editor. *Readings in groupware and computer supported cooperative work*. Morgan Kaufman, 1993.

[6] M. Barbuceanu and M. S. Fox. Cool: A language for describing coordination in multiagent systems. In *Proceedings of the First International Conference oil Multi-Agent Systems (ICMAS-95)*, pages 17–24, San Francisco, CA, 1995.

[7] M. Baudouin-Lafon, editor. *Computer-Supported Cooperative Work*. John Wiley Sons Ltd, 1998.

[8] J. K. Bennett. The design and implementation of distributed smalltalk. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 318–330, New York, NY, 1987. ACM Press.

[9] N.E. Bensaid and P. Mathieu. A hybrid and hierarchical multi-agent architecture model. In *Proceedings of PAAM'97*, pages 145–155, London-United-Kingdom, 1997. The Practical Application Company Ltd.

[10] Tim Berners-Lee. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. third edition edition, 2000.

[11] Grady Booch. *Object-Oriented Analysis And Design With Applications*. Benjamin Cummings, ISBN 0-8053-5340-2.

[12] F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. *Int Journal of Cooperative Information Systems*, 6(1):67–94, 1997.

[13] T. Brinck and S.E. McDaniel. Chi 97 workshop on awareness in collaborative systems. In *Proc. of CHI'97: Human Factors in Computing Systems*, (position papers available via `http://www.usabilityfirst.com/groupware/awareness/`), 1997.

[14] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2:14–23, 1986.

[15] Norman Carver and Victor R. Lesser. A new framework for sensor interpretation: Planning to resolve sources of uncertainty. In *National Conference on Artificial Intelligence*, pages 724–731, 1991.

[16] Jaelson Castro, Manuel Kolp, and John Mylopoulos. A requirements-driven development methodology. *Lecture Notes in Computer Science*, 2068:108–??, 2001.

[17] Gero Vierke Christian Gerber, Jorg Siekmann. Holonic multi-agent systems. Technical report, DFKI GmbH, 1999.

[18] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[19] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *The second ACM SIGPLAN conference on History of programming languages*, pages 37–52. ACM Press, 1993.

[20] Rosaria Conte, Cristiano Castelfranchi, and Frank Dignum. Autonomous norm acceptance. In Jörg Müller, Munindar P. Singh, and Anand S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 99–112. Springer-Verlag: Heidelberg, Germany, 1999.

[21] R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, and Yun Peng. Modeling agent conversations with colored petri nets. In *Third Conference on Autonomous Agents (Agents-99), Workshop on Agent Conversation Policies*, Seattle, May 1999. ACM Press.

[22] S.L. Crouch and Starfield. *Boundary Element Methods in solid mechanics*. Georges Allen & Unwin, 1983.

[23] W. S. Mc Culloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. In *Bulletin of Mathematical Biophysics*, volume 5:-133. 1943.

[24] Randall Davis, Howard E. Shrobe, and Peter Szolovits. What is a knowledge representation? *AI Magazine*, 14(1):17–33, 1993.

[25] Yves Demazeau. From interactions to collective behaviour in agent-based systems. In *Proceedings of the First European conference on cognitive science, Saint Malo, France*, pages 117–132, 1995.

[26] A. Derycke and F. Hoogstoel. Le travail coopératif assisté par ordinateur : quels enjeux pour les concepteurs. In *Actes du XIIIe Congrès INFORSID (INFormatique des ORganisations et Systèmes d'Information et de Décision)*, Grenoble, 1995.

[27] Virginia Dignum, John-Jules Meyer, and Hans Weigand. Towards an organizational model for agent societies using contracts. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 694–695. ACM Press, 2002.

[28] CA. Ellis, SJ. Gibbs, and GL. Rein. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):38–58, 1991.

[29] Richard K. Guy Elwyn R. Berlekamp, John H. Conway. *Winning Ways for your mathematical plays*. Academic Press, 1982.

[30] Marc Esteva, Julian A. Padget, and Carles Sierra. Formalizing a language for institutions and norms. In *Revised Papers from the 8th International Workshop on Intelligent Agents VIII*, pages 348–366. Springer-Verlag, 2002.

[31] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings of ICMAS'98*, 1998.

[32] J. Ferber and O. Gutknecht. Operational semantics of a role-based agent architecture. In *Proceedings of ATAL'99*, jan 1999.

[33] T. Finin, R. Fritzson, and D. McKay. A Language and Protocol to Support Intelligent Agent Interoperability. In *Proceedings of the CE & CALS Washington'92 Conference*, 1992.

[34] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, MD, USA, 1994. ACM Press.

[35] Tim Finin, Jay Weber, Gio Wiederhold, Mike Genesereth, Don McKay, Rich Fritzson, Stu Shapiro, Richard Pelavin, and Jim McGuire. Specification of the KQML Agent-Communication Language – plus example agent policies and architectures, 1993.
Ce document spécifie le langage KQML.
Source: `http://citeseer.nj.nec.com/finin94specification.html`.

[36] M. Florio, R. Gorrieri, and G. Marchetti. Coping with denial of service due to malicious java applets, 2000.

[37] S. Franklin and A. Grasser. Is it an agent, or just a program?: A taxonomy for autonomous agent. In *Proceedings of the 3rd International Workshop on Agent Theories, Architectures, and Languages*. Springer-Verlag, 1996.

[38] A. Fuxman, P. Giorgini, M. Kolp, and J. Mylopoulos. Information systems as social structures. In *Proceedings of the Second International Conference on Formal Ontologies for Information Systems (FOIS'01), Ogunquit, USA, October 2001.*, 2001.

[39] L. Gasser, C. Braganza, and N. Herman. *MACE: A Flexible Testbed fo Distributed AI Research*, volume Chapter 5, 119-152 of *Distributed Artificial Intelligence, Research Notes in Artificial Intelligence*. Pitman, third edition edition, 1987.

[40] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In M. Diekhans and M. Roseman, editors, *Fourth Annual Tcl/Tk Workshop (TCL 96)*, pages 9–23, Monterey, CA, 1996.

[41] Michel F. Gutknecht O., Ferber J. The madkit agent platform architecture. Technical report, Laboratoire d'Informatique, de Robotique, et de Microélectronique de Montpellier, Université de Montpellier II, 2000. `http://www.madkit.org/papers/rr000xx.pdf`.

[42] Kurt Gödel. Über die vollständigkeit des logikkalküls. doctoral dissertation, University Of Vienna, 1929.

[43] F. Hayes-Roth, D. J. Mostow, and M. S. Fox. Understanding speech in the hearsay-ii system. In L. Bolc, editor, *Speech Communication with Computers*, pages 9–42. Macmillan, München: Carl Hanser Verlag and London., 1978.

[44] C. Hewitt. Viewing control structures as patterns of passing messages. In *Artificial Intelligence: An MIT Perspective*. MIT Press, Cambridge, Massachusetts, 1979.

[45] C.E. Hewitt and H. Lieberman. Design issues in parallel architectures for artificial intelligence. Technical report, MIT, Cambridge, Massachusetts, 1983.

[46] Marc-Philippe Huget. *Une ingénierie des protocoles d'interaction pour les SMA*. PhD thesis, Université de Paris 9, Juin 2001.

[47] Kurt Jensen. Coloured petri nets - basic concepts, analysis methods and practical use, vol. 1: Basic concepts. In *EATCS Monographs on Theoretical Computer Science*, pages 1–234. Springer-Verlag: Berlin, Germany, 1992.

[48] Nobuyashu Osato Kazuhiro Kuwabara, Toru Ishida. Agentalk : Describing multiagent coordination protocols with inheritance. In *Proc. 7th International Conference on Tools with Artificial Intelligence (ICTAI'95)*, 1995.

[49] Yannis Labrou and Timothy W. Finin. A semantics approach for KQML - a general purpose communication language for software agents. In *CIKM*, pages 447–455, 1994.

[50] Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, 1988.

[51] Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys (CSUR)*, 26(1):87–119, 1994.

[52] Gauthier Picard Marie-Pierre Gleizes, Thierry Millan. Adelfe: Using spem notation to unify agent engineering processes and methodology. Technical Report IRIT/2003-10-R, IRIT, 2003.

[53] Masini, Napoli, Colni, Léonard, and Tombre. *Les langages à objets*, chapter Représentation des connaissances, pages 237–320. InterEditions, 1989.

[54] P. Mathieu and J.C. Routier. Une contribution du multi-agent aux applications de travail coopératif. *TSI Hermès Science Publication*, Numéro Spécial : Télé-applications, To appear: 2001.

[55] P. Mathieu, J.C. Routier, and Y. Secq. Dynamic skills learning. Technical Report 2000-06, LIFL, 2000.

[56] P. Mathieu, J.C. Routier, and Y. Secq. Dynamic skill learning: A support to agent evolution. In *Proceedings of the AISB'01 Symposium on Adaptive Agents and Multi-Agent Systems*, pages 25–32, 2001.

[57] P. Mathieu, J.C. Routier, and Y. Secq. Principles for dynamic multi-agent organisations. In *Proceedings of Fifth Pacific Rim International Workshop on Multi-Agents (PRIMA2002)*, August 2002.

[58] Hamza Mazouzi, Amal El Fallah Seghrouchni, and Serge Haddad. Open protocol design for complex interactions in multi-agent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 517–526. ACM Press, 2002.

[59] Frank G. McCabe and Keith L. Clark. April – agent process interaction language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI volume 890)*, pages 324–340. Springer-Verlag: Heidelberg, Germany, 1995.

[60] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[61] Marvin Minsky. *Society of Mind*, chapter A theory of memory, pages 81–92. Touchstone books, 1998.

[62] John Von Neumann. Zur theorie der gesellschaftsspiele. In *Matematische Annalen*, pages 295–320, 1928.

[63] John Von Neumann. Theory of self reproducing automata. University of Illinois Press, 1966.

[64] John Von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, third edition edition, 1953.

[65] Michel Ocello. Méthodologie et architectures pour la conception de systèmes multi-agents, avril 2003.

[66] J. Odell, H. Parunak, and B. Bauer. Extending uml for agents, 2000.

[67] Charles J. Petrie. Agent-Based Software Engineering. In Jeffrey Bradshaw and Geoff Arnold, editors, *Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 2000)*, Manchester, UK, 2000. The Practical Application Company Ltd.

[68] Massimo Cossentino Piermarco. Introducing pattern reuse in the design of multi-agent systems.

[69] John Poole. Model-Driven Architecture: Vision, Standards and Emerging Technologies. In *Workshop on Metamodeling and Adaptative Objects Models, ECOOP*, 2001.

[70] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[71] Pierre Michel Ricordel. *Programmation Orientée Multi-Agents :Développement et Déploiement de Systèmes Multi-Agents Voyelles*. PhD thesis, Institut National Polytechnique de Grenoble, octobre 2001.

[72] W. Shen and D.H Norrie. Agent-based systems for intelligent manufacturing: A state-of-the-art survey. In *International Journal of Knowledge and Information Systems*, pages 129–156, 1999.

[73] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[74] Yoav Shoham and Moshe Tennenholtz. On social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, 73(1-2):231–252, 1995.

[75] Jaime Simão Sichman, Rosaria Conte, Cristiano Castelfranchi, and Yves Demazeau. A social reasoning mechanism based on dependence networks. In A. G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 188–192, Chichester, 8–12 1994. John Wiley & Sons.

[76] Munindar P. Singh. Toward interaction-oriented programming. Technical Report TR-96-15, Department of Computer Science, North Carolina State University, 16, 1996.

[77] Barry Smith. *Speech Acts, Meanings and Intentions. Critical Approaches to the Philosophy of John R. Searle*, chapter Towards a History of Speech Act Theory, pages 29–61. De Gruyter, 1990.

[78] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In *Proceedings of the 1st ICDCS*, pages 186–192. IEEE Computer Society, 1979.

[79] Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In *IEEE Transactions on Computers*, pages 1104–1113, 1980.

[80] M. Stefik, DG. Bobrow, G. Foster, S. Lanning, and D. Tatar. Wysiwis revised: Early experiences with multi-user interfaces. *ACM Transactions on Office Information Systems*, 5(2):147–167, 1987.

[81] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[82] F. Tarpin-Bernard and B. David. Amf: un modèle d'architecture multi-agents multi-facettes. *Techniques et Sciences Informatiques, Hermès*, 18(5):555–586, Mai 1999.

[83] April Technical. Commitment and effectiveness of situated agents.

[84] Alan Turing. Can a machine think. In James R. Newman, editor, *The World of Mathematics*, volume 4, pages 2099–2123. Simon and Schuster, 1956.

[85] S. Vestal. A cursory overview and comparison of four architecture description languages. Technical report, Honeywell Technology Center, February 1993.

[86] Douglas H. Norrie Weiming Shen and Jean-Paul Barthès. *Multi-Agent Systems For Concurrent Intelligent Design and Manufacturing*, chapter Communication, Coordination and Cooperation, pages 161–162. Taylor & Francis, 2001.

[87] R. Wolski, J. Plank, J. Brevik, and T. Bryan. Analyzing market-based resource allocation strategies for the computational grid, 2000.

[88] M. Wooldridge, NR. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 2000.

[89] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. HTTP://www.doc.mmu.ac.uk/STAFF/mike/ker95/ker95-html.h (Hypertext version of Knowledge Engineering Review paper), 1994.

[90] Michael Wooldridge and Nicholas R. Jennings, editors. *Modelling Reactive Behaviour in Vertically Layered Agent Architectures*, volume 890 of *Lecture Notes in Computer Science*. Springer, 1995.

[91] Erick S. K. Yu and John Mylopoulos. Enterprise modelling for business redesign: the i* framework. *ACM SIGGROUP Bulletin*, 18(1):59–63, 1997.