Université de Montréal

Méta-modélisation de l'adaptation dynamique du contrôle des systèmes multi-agents

par
Viet Thang Pham

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en Informatique

Août, 2007

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée:

Méta-modélisation de l'adaptation dynamique du contrôle des systèmes multi-agents

présentée par:

Viet Thang Pham

a été évaluée par un jury composé des personnes suivantes:

El Mostapha Aboulhamid,	président-rapporteur
Houari Sahraoui,	directeur de recherche
Laurent Magnin,	codirecteur
Sylvie Hamel,	membre du jury
Sylvain Giroux,	examinateur externe

Thèse acceptée le: 7 septembre 2007

À mes parents

REMERCIEMENTS

Je voudrais avant tout manifester ma profonde gratitude à mon co-directeur de recherche, le Docteur Laurent Magnin. Son attention permanente à mon travail, ses critiques et ses suggestions persuasives, sa patience et sa compréhension quotidienne à mon égard m'ont permis de mener cette thèse jusqu'à la fin. Son enthousiasme pour les agents et les systèmes multi-agents m'a servi de fil conducteur dans ma recherche.

Je tiens énormément à remercier le Professeur Houari Sahraoui, pour son rôle de directeur de thèse, pour ses conseils précieux, pour sa disponibilité et son encouragement permanent. Je lui suis très reconnaissant du point de vue professionnel.

Je remercie vivement les membres du jury qui me font l'honneur de participer à ma soutenance de thèse. Je souhaite mentionner le Professeur El Mostapha Aboulhamid pour accepter de présider mon jury, ainsi que la Professeure Sylvie Hamel et le Professeur Sylvain Giroux de l'Université de Sherbrooke pour accepter de lire et d'évaluer mon travail de thèse.

Je remercie beaucoup le Centre de recherche d'informatique de Montréal (CRIM) qui m'a offert un support financier et un environnement de travail professionnel durant la réalisation de ma thèse.

Je suis très reconnaissant aux membres de l'équipe Génie logiciel et ingénierie de la connaissance (GLIC) du CRIM pour leur sympathie et leur aide précieuse pendant les années de travail commun. Je remercie tout particulièrement le Docteur Arnaud Dury et Nicolas Besson pour les discussions fréquentes qui m'ont été très fructueuses.

Mes sincères remerciements s'adressent à ma famille pour leur amour et leur encouragement touchant, malgré la distance qui nous sépare. Je remercie, enfin, tous mes amis et amies vietnamiens à Montréal pour leur soutien moral tout au long de mes années ici. Leur amitié m'a offert l'atmosphère familiale et aidé à mieux supporter la nostalgie de mon pays.

RÉSUMÉ

Du fait de leurs caractéristiques uniques telles que l'autonomie et la forte décentralisation, les systèmes multi-agents se présentent aujourd'hui comme une solution naturelle pour concevoir des applications distribuées dans des environnements hétérogènes, ouverts et dynamiques. Pourtant, de tels systèmes doivent faire face aux changements rapides et perpétuels de leur environnement et se mettre à jour pour suivre le progrès continu des technologies, tout en assurant la continuité de leur exécution. Dès lors, les systèmes multi-agents doivent être capables de s'adapter dynamiquement afin de réaliser les objectifs pour lesquels ils ont été conçus.

Se concentrant sur l'adaptation dynamique des systèmes multi-agents, notre travail commence par distinguer au sein de l'adaptation la partie fonctionnelle de celle de contrôle. Tandis que la première partie est spécifique à chaque système, la deuxième est commune à plusieurs systèmes. Ceci permet de produire un modèle d'adaptation du contrôle qui peut être utilisé dans la construction de différents systèmes multi-agents adaptatifs. Suivant cette direction, nous proposons un cadre de travail modélisant l'adaptation dynamique de la partie de contrôle dans les systèmes multi-agents en utilisant l'approche de méta-modélisation.

Les contrôles des systèmes multi-agents peuvent être divisés en trois niveaux : niveau plate-forme, niveau agent et niveau multi-agents. Au niveau plate-forme, le problème principal est la compatibilité d'exécution entre les plates-formes multi-agents. Notre solution est alors un modèle d'agents universels qui peut s'adapter dynamiquement à différentes plates-formes, lesquelles sont *a priori* incompatibles. Au niveau agent, nous proposons une architecture d'agents flexible et extensible en se basant sur notre modèle de *plugins* et le formalisme CATN (*Coupled Augmented Transition Network*). Un tel agent, composé du code compilé sous forme de *plugins* et du code interprété, lequel est spécifié par le formalisme CATN, est capable de se modifier et d'évoluer tout au long de sa vie et rend l'adaptation dynamique réalisable. Au niveau multi-agents, deux thèmes principaux de l'adaptation dynamique du contrôle est l'organisation des systèmes multi-agents et l'interaction entre les agents, notre recherche consiste donc en deux volets. Dans le premier

volet, nous introduisons des agents qui peuvent être organisés récursivement suivant un mode centralisé ou distribué, en fonction de leur position géographique. Dans le deuxième volet, notre méta-modèle d'interaction des agents, appelé MetaCATN et basé également sur le formalisme CATN, permet de spécifier les interactions entre les agents et de synchroniser la modification dynamique des comportements de ceux-ci, tout en assurant la continuité de l'exécution du système entier.

La mise en œuvre de ces modèles théoriques a été réalisée au travers de notre plate-forme *Guest*, implémentée en Java. *Guest* peut fonctionner comme une plate-forme multi-agents indépendante ou comme une couche supplémentaire au dessus de plusieurs plates-formes existantes, telles que Jade, Voyager, etc.

Mots clés : agents adaptatifs, méta-modélisation, adaptation dynamique, agents récursifs, formalismes d'agents, adaptation de contrôle.

ABSTRACT

Thanks to their unique properties such as autonomy and strong decentralization, multiagent systems have become one natural solution for building distributed applications in open, heterogeneous and dynamic environments. However, these systems have to face the rapid and perpetual changing of their environment and support frequent updating due to the continuous progress of technologies while ensuring the continuity of their execution. Therefore, multiagent systems must be able to adapt themself dynamically in order to achieve their goals.

Focusing on the dynamic adaptation of multiagent systems, our work begins with the separation between the functional part and the control part of these systems. While the first part is different from one system to another, the second part may be common to multiple systems. Therefore, it's possible to produce a model of the adaptation for the control which can be used for the development of numerous adaptive multiagent systems. With this respect, we propose a framework which models the dynamic adaptation of control in multiagent systems using a metamodelling approach.

The control part of a multiagent system can be divided into three levels : platform level, agent level and multiagent level. At the platform level, the main problem for adaptation is the execution compatibility between multiagent platforms. Our solution is a model of universal agents which can dynamically adapt themself to different platforms that can be incompatible. At the agent level, we propose a flexible and extensible agent architecture, which is based on our plugin model and the CATN formalism (Coupled Augmented Transition Network). These agents, which are composed of compiled code, in the form of plugins and interpreted code, specified by the CATN formalism, are able to modify themself and to evolve during their life. The dynamic adaptation of these agents is therefore realisable. At the multiagent level, two main themes of the dynamic control adaptation are organization of multiagent systems and interaction between agents, follow which are our two research directions. From the one hand, we define agent systems that can be recursively organized following centralized or distributed modes, depending on their geographical position. In the other hand, our metamodel for agent interactions,

which is called MetaCATN and based on the CATN formalism, allows to specify interactions between agents and to synchronize the dynamic modification of the behaviours of these agents, while guarantying the continuity of the execution of the whole system.

All our theoretical models are integrated into our platform *Guest*, which is implemented in Java. *Guest* can function as an independent multiagent platform or as an additional layer on top of numerous existing agent platforms, such as Jade, Voyager, etc.

Keywords : adaptive agents, metamodelling, dynamic adaptation, recursive agents, agent formalisms, adaptation controls.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ANNEXES

CHAPITRE 1

INTRODUCTION

1.1 Contexte et problématique

L'évolution actuelle de l'informatique, notamment à travers Internet, voit l'apparition d'applications informatiques de plus en plus complexes et dynamiques : les logiciels doivent fonctionner sur plusieurs types de plates-formes, se connecter à partir de différentes localisations et s'attendre à des changements continuels. En d'autres termes, l'environnement d'exécution de ce type d'applications est caractérisé par l'hétérogénéité, la distribution spatiale ainsi qu'une évolution perpétuelle [Jennings and Wooldridge, 1998] [Magnin, 1999]. Dès lors, des mécanismes d'adaptation sont nécessaires, voire indispensables, pour ce genre d'applications.

Plusieurs paradigmes de programmation ont été introduits pour assister à la réalisation de ce type d'applications, que ce soit en terme de méthodologie de développement (*Extreme Programming* [Beck, 1999]) ou en terme d'artefact de programmation (Programmation Par Aspects [Kiczales *et al.*, 2001]). Parmi eux, le paradigme d'agents se montre particulièrement prometteur [Shoham, 1993]. En effet, les caractéristiques uniques des agents et des systèmes multi-agents, telles que l'autonomie de l'exécution, la répartition de contrôle et des connaissances, ainsi que l'adaptation [Jennings *et al.*, 1998], semblent idéalement s'adresser aux problèmes posés.

Tout d'abord, les agents sont *autonomes* : ils peuvent exécuter des tâches sans l'intervention humaine, étant guidés seulement par leurs objectifs.

Dans les systèmes multi-agents, le contrôle et la connaissance sont *répartis* : chaque agent n'a qu'une vue partielle du monde, dispose de connaissances partielles du système entier et réalise des sous-tâches du système global. De plus, les agents peuvent être répartis géographiquement, se connectant par des communications asynchrones.

Les agents, suivant leur définition théorique, sont *adaptatifs* : s'il y a des changements dans l'environnement, ils peuvent changer automatiquement leur exécution afin d'achever

leur but ultime, ceci en se basant sur leurs connaissances.

En réalité, les caractéristiques décrites ci-dessus ne sont pas facilement obtenues, en particulier l'adaptation des agents. Une preuve de ce défi est que la plate-forme d'agents *de facto* JADE [Jade, 2003], ne fournit quasiment rien en ce qui concerne l'adaptation, bien que milliers de chercheurs et de développeurs contribuent quotidiennement à cette plate-forme. Ce fait montre que l'adaptation est un vaste problème, qui ne pourra être résolu que partiellement et progressivement.

Toutes les adaptations concernent une forme quelconque de changement des agents. Une question essentielle est alors de déterminer quand ces changements, prévus ou non au moment de lancement initial de l'application, peuvent être programmés et déployés. Suivant ce critère, on obtient deux types d'adaptation, que nous appelons *statique* et *dynamique*. Dans la première forme d'adaptation, qualifiée de *statique*, un changement auprès des agents concernés oblige l'arrêt de leur exécution. Au contraire, l'adaptation *dynamique* consiste en la capacité des agents à se transformer tout au long de leur vie, *sans interruption de leur exécution.*

Probablement influencées par l'aspect de maintenance dans le domaine du génie logiciel, la plupart des recherches sur l'adaptation des systèmes multi-agents sont de type statique, en se limitant à l'étape de la conception (voir par exemple [Bernon *et al.*, 2001] et [Tambe *et al.*, 2000]). Dans ces approches, on essaie de doter les agents d'algorithmes d'adaptation figés, dont la fonction est de choisir les meilleures solutions à partir des conditions présentes dans l'environnement, ou des résultats précédents [Marrow *et al.*, 2002]. Bien qu'elles permettent aux agents de se modifier afin de s'adapter, ces changements doivent être prévus au moment de la conception des agents. Une fois les algorithmes d'adaptation implémentés et déployés, il est alors nécessaire *d'interrompre l'exécution des agents* afin de les mettre à jour[1].

Cependant, les applications d'agents doivent faire face aux changements incessants et imprévus de l'environnement [Wooldridge, 2001], qui est ouvert et hétérogène. De plus, il est nécessaire qu'elles puissent fonctionner le plus longtemps possible. L'arrêt de l'exé-

[1] Il faut toutefois noter que dans certains cas particuliers [Briot *et al.*, 2002], le changement dynamique est possible mais de façon *ad hoc*, applicable pour une application unique

cution pour la mise à jour, même temporaire, est inacceptable dans certaines situations. Dès lors, l'adaptation dynamique des agents est un point essentiel pour maîtriser de tels environnements. Or, cette forme d'adaptation n'a été étudiée que de façon marginale et le support pour l'adaptation dynamique des agents reste un sujet encore trop peu exploré. Notre recherche dans le cadre de cette thèse vise à apporter de l'avancement dans ce domaine.

1.2 Objectifs de la thèse

Les situations d'adaptation sont très variables et largement imprévisibles. En conséquence, l'adaptation de la logique d'affaire (qui est implémentée par *le code fonctionnel*) est différente d'une application à une autre et est dès lors difficilement réutilisable. Cependant, nous constatons qu'une application ne contient pas que du code fonctionnel : il y a une autre grande portion du code non fonctionnel, concernant les aspects d'organisation, d'interaction, etc., que nous appelons *le code de contrôle*. Ce type de code est relativement indépendant des applications, ce qui le rend beaucoup plus générique et réutilisable dans de nombreux cas. Partant de ce constat, nos travaux de recherche se concentrent sur l'adaptation dynamique du contrôle des systèmes multi-agents.

Notre approche pour résoudre ce problème est empirique et progressive. Dès lors, cette thèse a les objectifs concrets suivants :

- Étudier des travaux de recherche existants dans le domaine d'adaptation afin d'identifier les éléments nécessaires de l'adaptation dynamique du contrôle pour les systèmes multi-agents ;
- Développer un cadre théorique pour l'adaptation dynamique du contrôle des systèmes d'agents à partir de cette étude. Ce cadre doit être ouvert en permettant d'intégrer facilement et systématiquement différents mécanismes de l'adaptation ;
- Mettre en œuvre ce cadre au sein d'une plate-forme complète, laquelle supporte la construction des applications multi-agents réelles dotant la capacité d'adaptation dynamique.

1.3 Contributions de la thèse

Notre premier apport dans la recherche sur l'adaptation des agents provient de la séparation entre l'adaptation du contrôle et celle du code fonctionnel des systèmes. En soulignant l'adaptation du contrôle des systèmes multi-agents, la dynamicité de leur adaptation peut être étudiée de manière systématique aux trois niveaux des systèmes multi-agents : plate-forme, agent et système multi-agents.

Au niveau plate-forme, notre contribution est de proposer le modèle d'agents universels *Guest*, lesquels sont adaptables aux différentes plates-formes d'agents existantes et *a priori* incompatibles.

Au niveau agent, nous avons développé une architecture d'agents flexible et extensible en nous basant sur notre modèle de *plugins* et le formalisme CATN (*Coupled Augmented Transition Network*). Un tel agent est capable de se modifier et d'évoluer tout au long de sa vie et rend l'adaptation dynamique réalisable.

Au niveau le plus élevé, le niveau système multi-agents, notre contribution sur l'adaptation dynamique des systèmes d'agents consiste en deux dimensions : l'organisation et l'interaction. D'une part, nous avons introduit deux modèles d'agents récursifs complémentaires, qui sont centralisés et distribués et permettent à un système d'agents hiérarchiques d'adapter son organisation à la distribution spatiale des agents membres. D'autre part, nous avons proposé MetaCATN, un modèle de deux couches qui permet de spécifier les interactions entre les agents et de synchroniser la modification dynamique de ces interactions, tout en maintenant l'exécution continue du système global.

Tous les apports théoriques ci-dessus ont été mis en œuvre au sein de la plate-forme *Guest*, laquelle est implémentée entièrement en Java et peut fonctionner sur plusieurs plates-formes d'agents existantes, telles que Jade, Grasshopper, etc.

Enfin, l'originalité et la contribution des travaux de recherche présentés dans cette thèse ont été reconnues par la communauté d'agents à travers les publications suivantes : [Pham *et al.*, 2001], [Magnin *et al.*, 2002c], [Magnin *et al.*, 2002a], [Magnin *et al.*, 2002b], [Pham *et al.*, 2004].

1.4 Contenu de la thèse

Le présent document est organisé comme suit :

Dans le chapitre 2, nous présentons les concepts de base dans le domaine des agents, puis analysons les différentes dimensions de l'adaptation, tout en introduisant les définitions indispensables pour la compréhension de la suite de cet ouvrage.

Le chapitre 3 introduit notre approche de recherche, celle de diviser l'adaptation du contrôle des systèmes multi-agents en trois niveaux d'architecture : plate-forme, agent et système multi-agent. Nous étudions ensuite des travaux connexes afin d'identifier les éléments importants de contrôle pour l'adaptation à chaque niveau.

Le chapitre 4 présente notre modèle d'agents universels, qui uniformise l'exécution et les services de base des plates-formes multi-agents en suivant l'approche interface. Notre agent est alors entièrement compatible avec plusieurs plates-formes multi-agents existantes qui sont au préalable incompatibles et il est ouvert pour l'intégration dynamique de nouvelles plates-formes.

Le chapitre 5 se concentre sur l'adaptation dynamique au niveau agent, qui peut être achevée par la modification *en-ligne* des comportements et des fonctionnalités des agents. Un modèle de composants dynamiques et un modèle de *plugins* sont proposés afin de rendre extensible des fonctionnalités de l'agent sous forme de code compilé, alors que le formalisme CATN permet de spécifier les comportements de l'agent sous forme de code interprété et de les rendre flexibles.

Le chapitre 6 présente nos deux volets de recherche au niveau système multi-agents : l'organisation et l'interaction. Le premier volet propose une organisation récursive des systèmes d'agents sous deux formes, centralisée ou distribuée, en fonction de la distribution spatiale des agents membres. Le deuxième volet présente le méta-modèle MetaCATN, permettant aux systèmes de modifier globalement et dynamiquement leurs comportements.

Le chapitre 7 détaille la plate-forme *Guest* que nous avons développée afin de mettre en œuvre tous nos modèles théoriques.

Enfin, le chapitre 8 propose une synthèse des travaux effectués tout en proposant des perspectives de travaux futurs.

CHAPITRE 2

ÉTAT DE L'ART SUR L'ADAPTATION DES SYSTÈMES MULTI-AGENTS

Ce chapitre présente une revue de la littérature concernant le concept d'agent informatique et la notion d'adaptation dans ce domaine.

Tout d'abord, les concepts de base dans le domaine des agents, ceux qui seront utiles pour la compréhension de la suite de cette thèse, sont introduits : *agent*, *système multi-agent*[1], *environnement*, *modèle d'agents* et *plate-forme d'agents*. Ces concepts sont étudiés selon trois points de vue différents : celui de *l'utilisateur*, celui du *chercheur / concepteur* et celui du *développeur / administrateur*, lesquels correspondent aux différentes étapes du cycle de vie des logiciels en général (spécification, conception et implémentation / déploiement). En catégorisant les concepts de cette manière, cette étude permet de distinguer les concepts semblant intuitivement similaires mais qui en réalité sont utilisés dans des contextes différents (par exemple *plate-forme d'agents* et *serveur d'agents*), en plus de souligner la dimension temporelle dans la vie des applications d'agents.

L'adaptation est depuis longtemps une des notions les plus importantes dans le domaine des agents. Dans la section 2.2, cette notion est clarifiée en allant du concept général à celui dans les systèmes d'agents ; ses facteurs les plus importants sont identifiés et analysés, en répondant aux questions *pourquoi* (pourquoi une adaptation est-elle nécessaire ?), *qui* (qui initialise et manipule l'adaptation ?), *quoi* (quels sont les sujets d'adaptation ?) et *quand* (quand l'adaptation peut-elle être préparée ?). Ensuite, deux distinctions fondamentales, par lesquelles le sujet principal de cette thèse est positionné, sont proposées : l'adaptation *statique/dynamique* et l'adaptation *fonctionnelle/du contrôle*.

La section 2.3, est dédiée à l'étude de multiples techniques dans l'informatique en général, offrant un support de l'adaptation à différents niveaux (question *comment*). Ces technologies sont étudiées en suivant des dimensions de l'adaptation mentionnées ci-dessus

[1]Dans le présent document, nous utilisons alternativement *application d'agents* et *système d'agents* dans le même sens que le terme *système multi-agent*.

(*qui*, *quoi* et *quand*). Une synthèse sur ces divers mécanismes d'adaptation sera produite, tout en notant les points positifs et les limites de chacun.

2.1 Agent et système multi-agent

Le concept d'agent a été utilisé depuis plusieurs décennies dans différentes disciplines : chimie, militaire, économie, etc. Dans le cadre de cette thèse, nous abordons uniquement les *agents informatiques*, c'est-à-dire les agents sous forme d'entité de logiciel.

Afin de définir systématiquement les concepts fondamentaux des agents et d'éviter la confusion, nous les présentons ci-dessous selon trois points de vue, lesquels représentent différents niveaux d'abstraction des systèmes d'agents. Du plus abstrait au plus concret, il y a les points de vue :

1. **de l'utilisateur** : par lequel, un agent / système multi-agents est considéré comme une boîte noire sur laquelle on ne connaît que son utilisation, ses fonctions et ses caractéristiques externes ;
2. **du concepteur / chercheur** : qui s'intéresse à l'architecture interne des agents / systèmes multi-agents, à leurs modèles d'interaction, de représentation de connaissances, etc. De cette façon, un agent / système d'agent est analysé comme une boîte blanche ;
3. **du développeur / administrateur** : qui ne concerne que le modèle de déploiement et d'exécution des agents / systèmes d'agents, c'est-à-dire la configuration matérielle et l'environnement physique pour l'exécution réelle de ceux-ci.

2.1.1 Concepts d'agent du point de vue de l'utilisateur

Ceci est la vision la plus abstraite tout en étant la plus simple des systèmes d'agents. Par cette vision, le monde d'agents se construit à partir de trois éléments : *les agents*, *les système multi-agents* et *l'environnement* (figure 2.1).

Parmi les nombreuses définitions d'agent dans la littérature [Franklin and Graser, 1996] [Ferber, 1995], nous avons convenu d'adopter la définition suivante, proposée par Jennings, Sykara et Wooldridge [Jennings *et al.*, 1998] : « Un agent est une entité informatique, *si-*

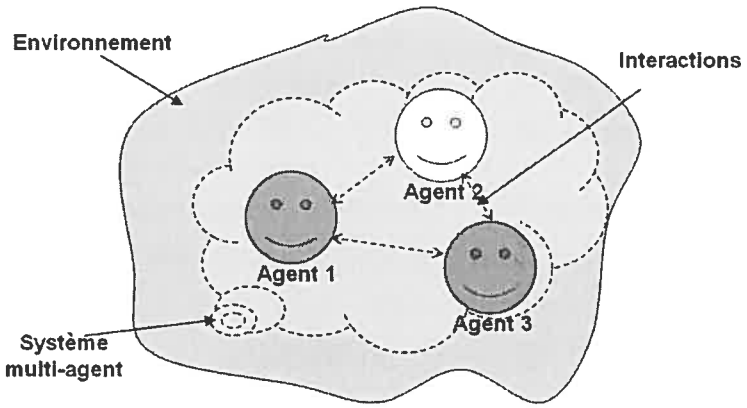

FIG. 2.1 – Éléments fondamentaux d'un système d'agents

tué dans un environnement, et qui agit de façon *autonome* et *flexible* afin d'atteindre les objectifs pour lesquels il a été conçu »[2].

Cette définition souligne les deux caractéristiques les plus importantes qu'un agent doit posséder :

- **autonomie** : un agent n'est pas un objet ou un composant exécutant des commandes externes (par exemple, de l'utilisateur du système informatique). À l'inverse, il s'exécute en se basant sur ses objectifs, ses connaissances et sa propre perception de son environnement. Une fois lancé, un agent autonome possède le contrôle total sur ses comportements et ses états internes [Guessoum and Briot, 1999] [Jennings *et al.*, 1998]. Par exemple, l'agent RemoteAgent a pris le contrôle de la navette NASA Deep Space One pour deux jours à environ 100 millions de kilomètres de la Terre, sans intervention humaine [Muscettola *et al.*, 1998] ;
- **flexibilité** : comme un agent mène sa propre vie dans un environnement en changement continuel, il lui est nécessaire de se modifier au cours de son existence afin d'accomplir ses tâches. Autrement dit, un agent doit être capable de s'adapter à la dynamicité et l'hétérogénéité de son environnement [Guessoum, 2000].

[2] « *An agent is a computer system, situated in some environment, that is capable of flexible autonomous action in order to meet its design objectives* »

Historiquement, le concept d'agent autonome provient du modèle d'acteur, introduit par Hewitt [Hewitt, 1977]. Basé sur un modèle de calcul défini [Agha, 1986], le modèle d'acteur a été concrétisé par Actalk [Briot, 1989a] [Briot, 1989b] [Briot, 2000]. Les acteurs de ce modèle peuvent être vus comme des processus indépendants possédant leur propre zone de code et leur propre zone de données. Chaque acteur a un comportement minimal, lequel se limite à traiter les messages en attente dans leur boîte aux lettres. En principe, il est possible de regrouper les fonctionnalités des acteurs autour de cinq thèmes :

- Le parallélisme : L'ensemble des calculs sont effectués a priori de façon parallèle.
- La communication : Les acteurs peuvent communiquer par envoi de messages asynchrones bufferisés. Le principe même de ce type de communication suppose un envoi de message sans retour avec continuation en argument. Ce type de communication permet d'envoyer un message à un autre acteur sans attendre que ce dernier soit disponible (c'est à dire qu'il soit dans l'état de recevoir le message) et surtout sans avoir à attendre le résultat du message.
- L'autonomie : Il n'y a pas de partage de données entre les différents acteurs si ce n'est bien sûr les données partagées. L'autonomie provient du fait qu'un acteur peut dès lors traiter et manipuler ses données sans avoir à se soucier des autres acteurs du système.
- La dynamicité : Les accointances entre les acteurs sont dynamiques et par là même permettent de modifier le réseau des communications.
- La localité : Un acteur n'a qu'une vue partielle de son environnement et ne peut donc pas communiquer avec un autre acteur s'il ne le connaît pas directement ou indirectement (indirectement dans le cas où, par exemple, un acteur est transmis comme continuation d'un calcul).

Ces aspects du modèle d'acteur a une très importante influence sur le développement des modèles d'agents que nous allons présenter dans les prochaines sections.

Les agents possèdent un lien étroit avec leur environnement, ou même y sont immergés (situés). Cet environnement est défini par Russel [Russel and Norvig, 1995] comme « *le médium commun aux agents du système* ». Selon la classification de Russel, l'environnement dispose des caractéristiques suivantes :

- **accessible versus inaccessible** : un environnement est accessible si un agent peut obtenir l'information complète, mise à jour et précise de son état. La plupart des environnements réels sont inaccessibles dans ce sens ;
- **déterministe versus non-déterministe** : dans un environnement déterministe, chaque action produit un effet précis et prévisible ;
- **discret versus continu** : un environnement est discret si chaque état possible peut être associé à une valeur entière ou un symbole. Au contraire, il est impossible d'associer une valeur entière ou symbolique dans un environnement continu, il faut donc utiliser des valeurs flottantes ;
- **statique versus dynamique** : un environnement est dit statique s'il est possible d'assumer qu'il reste inchangé, sauf par les actions des agents. Cette hypothèse n'est plus correcte dans un environnement dynamique, où les changements sont hors du contrôle des agents.

Russel a également indiqué que la plupart des environnements complexes sont pratiquement inaccessibles, non-déterministes, dynamiques et continus. Ce type d'environnement est souvent dit *ouvert* [Hewitt, 1986].

Internet est un exemple typique et à grande échelle de ce genre d'environnements. De plus, il est vastement distribué et hétérogène : ce réseau est composé de millions de machines se trouvant sur tous les continents, les machines sont largement différentes à tous les niveaux (matériels, systèmes d'exploitation, protocoles de communication, applications) et des millions de services y sont fournis (web, courriel, téléphone, etc.). Dans ce type d'environnements, le paradigme d'agents est un candidat prometteur pour le développement des applications [Durfee and Rosenschein, 1994] [Etzioni and Weld, 1995] [Jennings and Wooldridge, 1998] [Magnin, 1999].

En réalité, il est très courant que l'on rencontre des applications d'agents ayant plus d'un agent : on parle ici de systèmes multi-agents. Selon Briot [Briot and Demazeau, 2001] et Wooldridge [Wooldridge, 2001], un système multi-agent (SMA[3]) est « *un système distribué composé d'un ensemble d'agents interagissant* », dans lequel :

[3]Pour éviter de répéter le terme *système multi-agent*, nous utilisons désormais son abbréviation ***SMA*** tout au long de cette thèse.

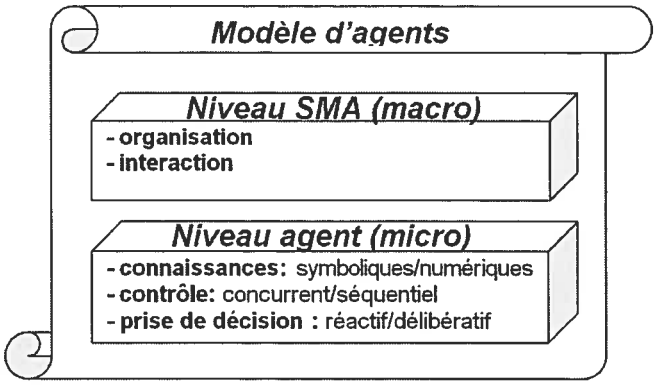

FIG. 2.2 – Deux niveaux de spécification d'un modèle d'agents

- chaque agent possède des informations ou des capacités de résolution de problèmes limitées (ainsi, chaque agent dispose d'un point de vue partiel) ;
- il n'y a aucun contrôle global ;
- les données sont décentralisées ;
- le calcul est asynchrone.

2.1.2 Concepts d'agent du point de vue du concepteur / chercheur

De ce point de vue, on s'intéresse principalement aux modèles d'agents, lesquels sont « *une description interne de l'agent* ». Les modèles d'agents sont nombreux dans la littérature, par exemple : agent « *Subsumption* » de Brooks [Brooks, 1991], agent BDI de Rao [Rao and Georgeff, 1995], agent INTERRAP de Fischer [Fischer *et al.*, 1995], KAOS de Bradshaw [Bradshaw, 1995] et MOISE de Boissier [Hannoin *et al.*, 1999].

Comme la plupart des systèmes d'agents sont en effet des SMAs, un modèle d'agents doit alors décrire un SMA au lieu d'un agent tout seul[4]. Un modèle d'agents est donc spécifié à deux niveaux [Demazeau, 1995] : niveau *micro* (agent) et niveau *macro* (SMA) (figure 2.2).

Au niveau agent, ce modèle identifie les éléments composant l'agent, le type de repré-

[4]D'ici la fin de cette thèse, *modèle d'agents* et *modèle de SMA* seront les même concepts, sauf avis contraire.

sentation des connaissances (uniforme/hétérogène, implicite/explicite, symbolique/numérique), le type de contrôle (centralisé/hiérarchique, concurrent/séquentiel) mis en œuvre ainsi que les interactions avec l'environnement.

Selon le couplage de l'agent à l'environnement et la manière dont la fonction de prise de décision est réalisée, nous avons les *agents réactifs*, *agents délibératifs* et *agents hybrides.*

Au niveau SMA, d'après Jung [Jung and Fischer, 1998], les modèles d'agents définissent notamment l'interaction entre les agents dans un système et l'organisation de ceux-ci.

En se basant sur la vision de l'interaction entre les agents, nous pouvons obtenir ces trois types d'agents : *agent autonome*, *agent interagissant* et *agent social.*

Au-delà des caractéristiques ci-dessus, les modèles sont différents également par les terminologies utilisées, les domaines d'application, le niveau d'abstraction, les aspects mis en avant, etc.

2.1.3 Concepts d'agent du point de vue du développeur / administrateur

Sur le plan pragmatique, un modèle d'agents se concrétise grâce à un code exprimé dans un langage de développement (C++, Java, Tcl, langage propriétaire) et à certains outils de déploiement et d'exécution (les environnements de conception, de déploiement, d'exécution, etc.). Cet ensemble de langages et d'outils est appelé *plate-forme d'agents* ou *plate-forme multi-agents.*

Une plate-forme d'agents fournit une infrastructure pour la construction (le développement, le déploiement et l'exécution) des SMAs, *en offrant aux agents des services fondamentaux* pour trouver et interagir avec d'autres agents ainsi que pour accéder à l'environnement [Ricordel and Demazeau, 2000] [Perdikeas *et al.*, 1999], ou même en supportant la migration des agents [Aridor and Oshima, 1998]. Autrement dit, une plate-forme d'agents joue le rôle de médium entre les agents ainsi qu'avec leur environnement, tout comme un système d'exploitation connecte l'utilisateur / l'application à l'ordinateur. Quelques exemples de plates-formes d'agents sont MADKIT [Madkit, 1999], Zeus [Nwana *et al.*, 1999], FIPA-OS [Fipa-os, 2000]. Un SMA est alors défini comme « *une instance d'un modèle d'agents donné sur une plate-forme concrète* ».

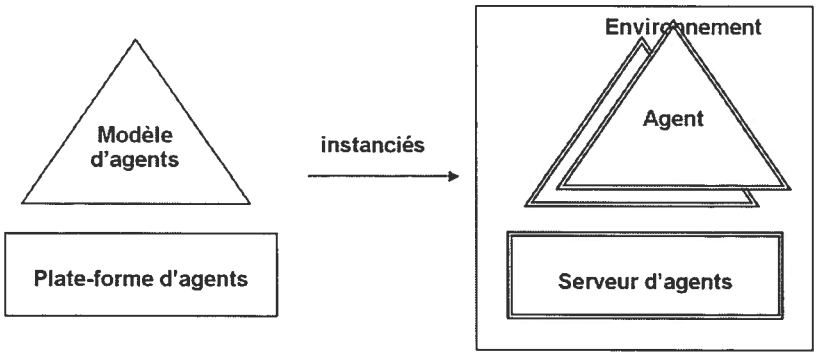

FIG. 2.3 – Relation d'instanciation des modèles/plates-formes vers agents/serveurs

Un serveur d'agents est un terme qui désigne *une instance d'une plate-forme multi-agent en fonctionnement sur un ordinateur spécifique*, et de ce fait capable d'héberger des agents et de leur offrir un environnement d'exécution (créer, interpréter / exécuter, accueillir et mettre un terme au cycle de vie des agents) ainsi qu'un ensemble de services de base : communication, localisation, etc.

Un agent, lorsqu'il s'exécute sur ces serveurs, est composé de trois éléments :

- **le code** : l'implémentation de l'agent dans un langage compris par sa plate-forme ;
- **les données** : les objectifs et des connaissances de l'agent (de son environnement, de ses stratégies et tactiques, ainsi que des autres agents et de lui-même) ;
- **le contrôle** : chaque agent est doté d'un ou plus de « *threads* » d'exécution pour son code. C'est à ce niveau que l'agent est entièrement différent de l'objet : un agent est autonome et capable d'agir indépendamment, en se basant sur ses connaissances.

Comme illustré dans la figure 2.3, le fonctionnement d'un SMA dépend étroitement de son environnement, ou plus précisément, de son contexte d'exécution. Le contexte d'exécution d'un SMA est donc défini comme « *tous les facteurs qui peuvent affecter son exécution* ». Ce contexte concerne l'hétérogénéité, la distribution et l'évolution permanente [Rey and Coutaz, 2004]. Plus précisément, le contexte d'exécution inclut tous les attributs détectables et mesurables des machines hôtes d'agents, de l'environnement autour de celles-ci et des utilisateurs du système [Ayed *et al.*, 2004].

Le contexte d'exécution peut être classifié en deux catégories diverses : (1) le (les) serveur(s) sur lequel (lesquels) l'agent/ le SMA s'exécute et (2) l'environnement concret dans lequel l'agent / le SMA fonctionne. Quand on parle de (1), on se réfère au support d'exécution, au support de communication, etc. Quant à (2), on mentionne les facteurs particuliers dépendant de la nature de l'application, par exemple la température externe dans une application de gestion de chauffage des bâtiments, le nombre d'agents fournissant le service de calcul dans une application de simulation, etc. Lorsque (1) peut être commun pour plusieurs applications d'agents, (2) est spécifique à chacune de celles-ci. Dans le cadre de cette thèse, nous nous intéressons particulièrement à la première catégorie.

2.1.4 Applications multi-agents

Le paradigme d'agents a trouvé sa place dans les systèmes de télécommunication, les robots, le multimédia, les systèmes de simulation, les systèmes embarqués, les systèmes d'apprentissage distribué, le commerce électronique et beaucoup d'autres applications [Magnin, 1996] [Oliveira, 1997] [Giroux, 2001] [Guilfoyle, 1995] [Guttman *et al.*, 1998]. Du fait des caractéristiques uniques des SMAs, cette approche est particulièrement appropriée pour développer des systèmes distribués fonctionnant dans des environnements dynamiques et complexes, tels qu'Internet [Zambonelli *et al.*, 2001] [Joshi and Singh, 1999]. Ce type d'environnements est caractérisé par les attributs suivants :

- **répartition** : les différents composants d'application s'exécutent sur différentes machines ;
- **hétérogénéité** : cette hétérogénéité se trouve à tous les niveaux : différents équipements vendus par des centaines fournisseurs (IBM, Apple, NEC, etc.), des dizaines systèmes d'exploitation (Windows, MacOS, Unix, etc.), de nombreux protocoles (Ethernet, Token Ring, etc.) et des millions de services et d'applications (web, email, clavardage, etc.) ;
- **dynamicité** : de nouvelles machines, des protocoles et services sont ajoutés, supprimés ou modifiés en tout temps, le « *bandwidth* » change de temps en temps.

Il est évident que dans ces environnements, tout peut éventuellement changer : la disponibilité des ressources de calcul, la fiabilité de la ligne de communication, la mise

à jour des services utilisés, etc. Étant donné la forte dépendance des SMAs à leur environnement, l'adaptation devient nécessaire, voire indispensable pour que les applications SMAs puissent survivre dans ce type d'environnements. La section qui suit précisera ces notions d'adaptation.

2.2 Adaptation des systèmes multi-agents

Cette section propose une analyse de synthèse sur la notion d'adaptation dans le domaine des SMAs. Nous commençons par la définition de la notion « adaptation », allant du concept général à celui dans les SMAs. Ensuite, nous identifions et analysons les facteurs les plus importants de l'adaptation, tels que les acteurs de l'adaptation, le moment de l'adaptation, le sujet de l'adaptation, etc. C'est aussi dans cette section que nous proposons nos propres définitions et classifications de l'adaptation des SMAs, celles qui nous permettent de bien positionner le sujet de la thèse.

2.2.1 Dimensions de l'adaptation

Selon le dictionnaire Hachette [Hachette, 2002], *adapter* est un verbe indiquant « *les actions qui rendent (un dispositif, des mesures, etc.) apte à assurer ses fonctions dans des conditions particulières ou nouvelles* ». De plus, *adaptabilité* est « *la qualité de ce qui peut être adapté, de ce qui peut s'adapter* ».

Ces définitions s'adaptent parfaitement aux systèmes informatiques en général et aux SMAs en particulier. En reprenant la définition ci-dessus, une adaptation correspond au processus de modification du système, nécessaire pour que le système réponde à ce que l'on attend de lui dans un contexte précis. Ce processus de modification peut s'opérer de différentes manières. Quant à l'adaptabilité, cette notion fait référence à la potentialité, aux qualités inhérentes d'un système en ce qui concerne l'adaptation.

Dans le cas des SMAs, l'adaptation permet d'assurer l'exécution normale des SMAs[5], notamment pour des systèmes complexes fonctionnant dans des environnements hétéro-

[5]Évidemment, l'adaptation tout seule ne peut pas assurer la réussite des SMAs, mais elle devrait éliminer les échecs du fait des modifications de l'environnement.

gènes, répartis et dynamiques tels qu'Internet. En répondant de manière adéquate aux changements de l'environnement, les SMAs peuvent éviter l'arrêt inattendu causé par des erreurs fatales et poursuivre leur exécution selon leurs objectifs. Il en émerge dès lors un certain nombre de questions récurrentes (*qui*, *quoi* et *quand*) auxquelles chaque solution doit répondre. Les sections qui suivent permettent d'apporter pour chacune d'elles une réponse.

Dans la section 2.2.2, nous expliquons le besoin d'adaptation dynamique dans le cas des applications d'agents. La section 2.2.3 décrit trois étapes successives d'un processus d'adaptation. Nous indiquons dans la section 2.2.4 qu'avec ce qui initialise et pilote le processus d'adaptation, on a deux niveaux d'adaptation : adaptable et adaptatif (question *qui*). La section qui suit indique trois moments différents pour l'intégration du code de l'adaptation au système : avant, à son lancement et durant l'exécution (question *quand*). Dépendamment de ces moments de l'adaptation, on peut parler d'adaptation statique ou dynamique. Finalement, la section 2.2.6 distingue les sujets d'adaptation, lesquels peuvent être des codes fonctionnels ou non-fonctionnels de l'agent (question *quoi*). Ces différentes distinctions nous permettront de bien positionner le sujet de cette thèse.

2.2.2 Nécessité de l'adaptation dynamique

De nombreux progrès, réalisés dans les domaines des systèmes d'informations à grande échelle et de l'informatique mobile, ont introduit de nouvelles problématiques et créé de nouveaux besoins en terme d'adaptabilité pour la construction des applications SMAs. Pour eux, l'environnement présente une hétérogénéité importante, une grande variabilité et de nombreuses possibilités d'évolution, aussi bien au niveau des moyens d'exécution que des moyens de communication. Ainsi, les ressources offertes peuvent être extrêmement disparates selon que l'on utilise un assistant personnel, un ordinateur portable, une station de travail ou un serveur d'exécution dédié. De plus, ces éléments sont soumis à d'importantes variations au cours du temps. La disponibilité des ressources n'est pas figée et peut varier en fonction d'ajout ou de suppression à la volée de périphériques, ou de leur atteignabilité dans le cas de l'utilisation d'une ressource distante.

Les caractéristiques de ces environnements nous obligent à reconsidérer le développe-

ment et l'exécution des SMAs dans un contexte d'un genre nouveau. En effet, dans les environnements traditionnels, le développement et l'exécution des logiciels s'effectuent en supposant que le support d'exécution est stable et connu à l'avance. Lorsqu'une application alloue une ressource, une non-disponibilité de celle-ci en cours d'exécution conduit à une terminaison non prévue de l'application. Dans les environnements que nous considérons, du fait de l'hétérogénéité, de la répartition et de la dynamicité, de nombreux changements peuvent intervenir et ceux-ci ne doivent pas être considérés comme des sources d'erreurs fatales, mais doivent au contraire être pris en compte à travers un processus d'adaptation de l'application [Guessoum, 2004].

Si la situation d'adaptation est prévisible, ce code peut être directement intégré dans les agents dès la phase de développement des SMAs. Pourtant, dans les environnements complexes, où la plupart des situations sont imprévisibles, une telle rigidité n'est plus appropriée. Il devient nécessaire que le code d'adaptation puisse être ajouté aux SMAs après que ceux-ci soient déjà en fonction [Maes, 1994]. De plus, ces SMAs doivent fonctionner sur des périodes de plus en plus longues, l'arrêt d'un SMA pour la mise à jour doit alors être très limité, même inacceptable dans plusieurs cas. Dès lors, l'adaptation dynamique des SMAs est indispensable, mais non optionnelle, dans les environnements complexes.

2.2.3 Processus d'adaptation

Selon David [David and Ledoux, 2002], l'adaptation se fait en cours d'exécution de l'application et son déroulement peut être décomposé en trois étapes successives[6], tel que montré dans la figure 2.4 :

- **Détection** : cette étape consiste à détecter et à notifier un changement du contexte d'exécution. Ces changements peuvent être l'apparition ou la disparition de certains services, l'information de la disponibilité des ressources de calcul, une mise à jour d'un composant, etc. Généralement, ces changements sont détectés par des senseurs, informés par la plate-forme d'exécution ou par les autres intervenants du SMA.
- **Analyse** : l'étape d'analyse consiste à déterminer les modifications qui doivent être

[6] Exceptionnellement, dans les approches bio-inspirées telle que l'adaptation par rétro-action, les étapes d'adaptation ne sont pas nécessairement les mêmes [Ando, 2006].

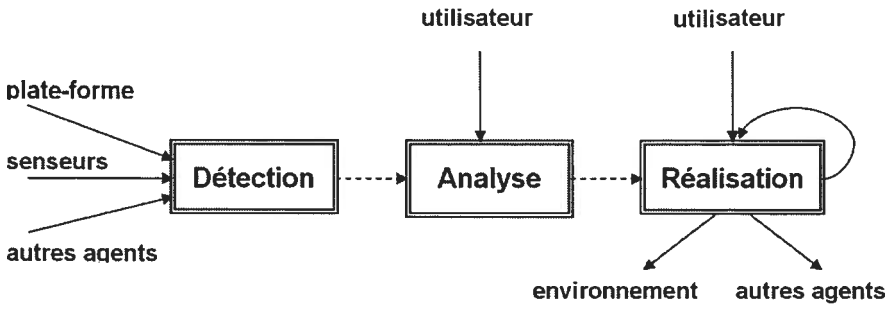

FIG. 2.4 – Processus d'adaptation

effectuées pour réagir aux changements détectés dans la première étape. Ces choix peuvent être effectués par l'humain (programmeur, administrateur, utilisateur, etc.) ou par les agents eux-mêmes en suivant différents algorithmes ou stratégies d'adaptation.

- **Réalisation** : l'étape de réalisation concerne tous les moyens qui vont être mis en œuvre pour appliquer la décision prise à l'étape précédente. La réalisation est souvent effectuée par les agents, ce qui est différent des systèmes traditionnels, dans lesquels la réalisation se fait par l'utilisateur. Pourtant, il est parfois nécessaire d'avoir l'aide de l'utilisateur. Les modifications à effectuer peuvent se restreindre à l'intérieur d'un seul agent (modification locale) ou impliquer de multiples agents (modification globale).

Dans le cas où l'intervention humaine est entièrement absente du processus d'adaptation d'un SMA, celui-ci est appelé *système adaptatif*. Cependant, la plupart des SMAs sont *semi-adaptatifs* : ils demandent une intervention externe dans la deuxième ou même la dernière phase.

2.2.4 Acteur de l'adaptation

Concernant les trois étapes du processus de l'adaptation, nous pouvons distinguer trois types d'acteurs qui y participent :

- **Développeur** : lorsque l'adaptation est prévue à l'avance, le développeur peut dé-

cider de définir une ou plusieurs des étapes d'adaptation. Les choix sur les sujets à adapter, les règles d'adaptation et les mécanismes peuvent être fixés à l'implémentation. L'adaptation est automatique (dans le sens de l'absence des interventions explicites de l'humain), mais ses étapes sont prédéterminées et donc non modifiables.

- **Utilisateur** : si l'on désire ne pas figer une des étapes de l'adaptation à l'implémentation, il est possible d'effectuer les choix au lancement du système ou pendant son exécution. Ces choix peuvent être faits par un utilisateur qui peut alors décider le déclenchement de l'adaptation, des sujets à adapter, de la stratégie ou des mécanismes. L'adaptation est alors non automatique, les étapes sont modifiables.
- **SMA** : si l'on se place dans la même hypothèse que ci-dessus (c'est à dire étapes non figées à l'implémentation et choix retardés), les choix d'adaptation peuvent être pris par une entité logicielle. Par exemple, le déclenchement de l'adaptation peut s'opérer grâce à des sondes d'observation des ressources et des notifications de changement. La décision d'adaptation peut, par exemple, être assurée par un système expert. La réalisation de l'adaptation peut être laissée à un agent spécifique. L'adaptation est alors automatique.

2.2.5 Préparation pour l'adaptation

La préparation pour l'adaptation des SMAs consiste en deux tâches :

- prévoir les possibilités d'adaptation et faire des choix en conséquence, suivant un algorithme particulier ;
- intégrer l'implémentation de cet algorithme aux SMAs.

Bien que l'adaptation des SMAs doit être prise en compte dès la phase de conception [Bernon *et al.*, 2001] et que tous les processus d'adaptation doivent se réaliser à la phase de déploiement ou d'exécution des SMAs, l'implémentation et l'intégration du code d'adaptation peuvent être effectuées à différents moments de leur existence : avant l'exécution, au lancement ou au cours de l'exécution du système.

- **Avant l'exécution du système** (« *compile-time* ») : dans ce cas, le développement du code d'adaptation et son intégration aux SMAs doivent être réalisés avant le démarrage du système. De plus, si l'on prévoit une adaptabilité lors de l'exécu-

tion, il faut choisir une approche permettant aux composants et/ou à leurs liaisons d'exister explicitement à l'exécution, et pas seulement au moment du codage. Par exemple, les « *middlewares* » actuels proposent de séparer les services métiers des services d'infrastructure sans toutefois réifier ces derniers à l'exécution. Ensuite, une intégration peut être réalisée à la compilation, à l'édition des liens en fonction de la plate-forme cible.

- **Au lancement du système** (« *load-time* ») : au moment du déploiement, l'administrateur connaît l'état de l'environnement d'exécution et sa topologie. À cet instant, il peut faire des choix pertinents d'adaptation, en fonction du contexte de déploiement [Ayed *et al.*, 2003] [Campadello, 2001]. Il s'agit d'une intégration au début de l'exécution du programme.
- **Au cours de l'exécution du système** (« *run-time* ») : dans ce cas, il est possible d'intégrer le code d'adaptation aux SMAs lors de son exécution et de rendre ces derniers adaptables aux situations imprévues au moment de leur lancement. Cette modification peut être effectuée à l'aide d'un tiers ou automatiquement.

Dans les deux premiers cas, l'adaptation est *statique*, dans le sens que cette adaptation doit être préparée à l'avance et alors fixée une fois que le système démarre. Dans le dernier cas, l'adaptation est *dynamique*, parce qu'elle n'est pas obligée d'être disponible lors de la mise en place des SMAs et peut être ajoutée après le démarrage de ces derniers. Cette remarque conduit à deux définitions importantes :

Définition 1 (Adaptation statique) *L'adaptation statique est l'adaptation dont l'implémentation et le déploiement doivent être réalisées avant que le système démarre.*

Définition 2 (Adaptation dynamique) *L'adaptation dynamique est l'adaptation dont l'implémentation et/ou le déploiement se réalisent au cours de l'exécution du système.*

À partir de la première définition, nous voyons que les adaptations statiques sont pertinentes seulement pour des situations d'adaptation bien prévues lors de la phase de développement de l'application. Toutefois, n'importe quelle modification sur celles-ci demande l'arrêt de l'application. Par cette contrainte, ce type d'adaptation devient très limité pour des SMAs fonctionnant dans des environnements imprévisibles. Pour ce genre

d'environnements, seul le type d'adaptation dynamique est applicable : il permet aux SMAs de s'adapter aux situations survenant au cours de leur vie, tout en continuant leur exécution. De plus, l'adaptation dynamique peut être construite progressivement et continuellement aux SMAs et ces derniers peuvent évoluer au cours de leur vie afin de s'adapter aux changements imprévus de l'environnement.

Le tableau suivant montre l'implication des acteurs de l'adaptation dans les trois étapes de la vie des SMAs :

Critère/Moment	**Avant l'exécution**	**Au déploiement**	**À l'exécution**
Adaptation	statique	statique	dynamique
Acteurs impliquants	programmeur	programmeur administrateur	programmeur administrateur SMA lui-même

TAB. 2.1 – Implication des acteurs de l'adaptation dans la vie des SMAs

2.2.6 Sujets de l'adaptation

L'adaptation caractérise les systèmes dans lesquels les agents cherchent à améliorer leur fonctionnement individuel ou collectif, afin d'atteindre leurs buts. Cette adaptation peut prendre deux directions : l'optimisation de l'exécution des fonctionnalités courantes d'agents en utilisant des algorithmes adaptatifs pour régler des paramètres de ces fonctionnalités, ou la modification ponctuelle et optimale de leurs fonctionnalités en fonction des modifications de l'environnement. Tandis que la première direction se concentre sur les aspects fonctionnels de l'agent, tels que l'apprentissage ou l'optimisation, la deuxième met l'accent sur les aspects non-fonctionnels de celui-ci.

Généralement, une application est composée de deux types de code :

- **code fonctionnel** : le code permet de résoudre le problème principal pour lequel l'application a été développée. Par exemple, pour un système de réservation de billets d'avion, le code qui cherche le prix le plus bas est du code fonctionnel ;
- **code de contrôle ou code de support** : le code spécifique dont le but est d'assurer le fonctionnement correct (et efficace) du code fonctionnel. Dans les SMAs, les services de communication et ceux de migration sont des exemples du code de contrôle.

Cependant, la distinction entre ces deux types de code n'est pas toujours claire, elle dépend des applications, ou du moins des types d'applications. Par exemple, la communication sécuritaire fait partie du code fonctionnel des applications militaires mais elle n'est que le code du support pour un SMA d'observation de l'état de réseau interne.

Correspondant à ces deux types de code, nous pouvons distinguer deux types d'adaptation : adaptation fonctionnelle et adaptation du contrôle.

Définition 3 (Adaptation fonctionnelle) *L'adaptation fonctionnelle est l'adaptation faite sur le code fonctionnel de l'agent afin de mieux réaliser ses objectifs en fonction des ressources et informations disponibles.*

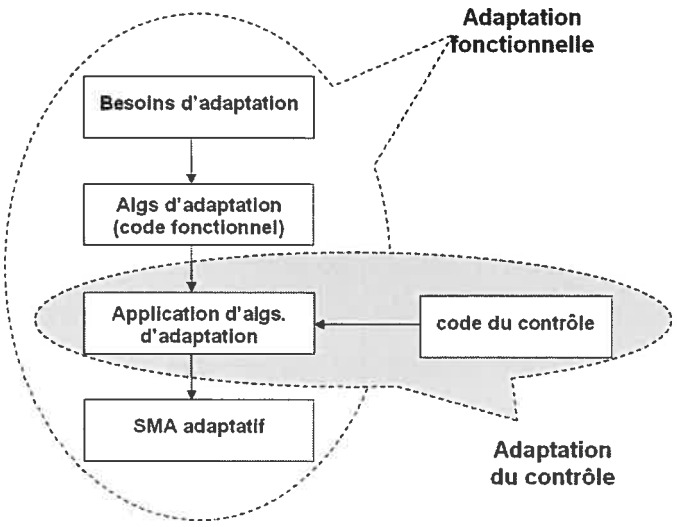

FIG. 2.5 – Adaptation fonctionnelle et adaptation du contrôle

Cette forme d'adaptation est largement dépendante des objectifs des SMAs et du contexte d'exécution. Les travaux concernant cette forme d'adaptation, par exemple [Stone *et al.*, 2001] [Barber and Martin, 2001] [Briot *et al.*, 2002], se concentrent sur les algorithmes d'apprentissage et les stratégies d'optimisation, qui sont spécifiques aux applications. En conséquence, cette forme d'adaptation ne peut pas être généralisable.

Définition 4 (Adaptation du contrôle) *L'adaptation du contrôle est l'adaptation faite sur le code du contrôle de l'agent afin de mieux réaliser les adaptations fonctionnelles.*

L'adaptation du contrôle peut être effectuée au niveau architecture des SMAs, parce qu'elle est indépendante d'un contexte de travail particulier ou même d'un ensemble de contextes similaires. De plus, elle peut être fournie sous forme de services de support aux adaptations fonctionnelles. Par conséquent, ce type d'adaptation est généralisable et réutilisable.

La figure 2.5 montre la relation entre l'adaptation fonctionnelle et celle du contrôle : l'adaptation fonctionnelle résulte des « briques » adaptatives, tandis que l'adaptation du contrôle concerne la façon, les mécanismes, l'infrastructure à utiliser pour rendre un SMA adaptatif avec ces « briques ».

La relation entre ces deux types d'adaptation est similaire à celle entre le système d'exploitation et les applications fonctionnant au-dessus. Comme la distinction entre le code fonctionnel et le code de contrôle n'est pas toujours claire, il n'existe pas non plus de distinction claire entre ces deux types d'adaptation. Au cours de l'évolution de l'environnement et du modèle d'agents, il est possible que certains codes fonctionnels puissent devenir code de contrôle. Par exemple, la gestion d'un protocole de communication par le langage ACL [FIPA, 1998] peut être considérée comme fonctionnelle ou de contrôle, ce qui est dépendant de la plate-forme d'agents utilisée.

Dans le cadre de cette thèse, notre recherche est dédiée à *l'adaptation dynamique du contrôle*. La prochaine section fournit alors un survol sur les techniques qui pourront être utiles dans ce type d'adaptation.

2.3 Techniques générales pour l'adaptation

Dans cette section, nous étudions six approches différentes en informatique offrant un support à l'adaptation : Aspects, *Model-Driven Architecture*, Réflexion, Interprétation, Composants et Services asynchrones.

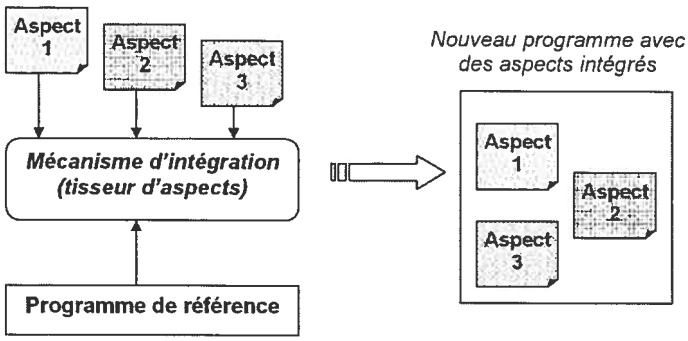

FIG. 2.6 – Modèle général de l'AOP

2.3.1 Aspects

Le paradigme de programmation par aspects (AOP - *Aspect Oriented Programming*) [Kiczales *et al.*, 1997] vise à une séparation des différentes préoccupations des programmeurs dans la réalisation d'un programme. Chaque préoccupation est conçue comme étant un aspect. L'idée de l'AOP est de pouvoir identifier, encapsuler et développer des aspects séparément et indépendamment. Ensuite, on les intègre aux « bons endroits » dans l'application pour qu'elle prenne en compte un nouveau but, un nouveau besoin, etc.

La figure 2.6 montre le principe de base de l'AOP. Le modèle de l'AOP fonde la définition d'une application sur un programme principal, appelé le programme de référence, et un ensemble d'aspects indépendants. Le programme de référence détermine la sémantique métier de l'application. Le programme de référence et les aspects sont, en théorie, développés séparément. Les aspects sont ensuite tissés à ce programme afin de l'étendre ou de l'adapter à un usage particulier. Ce tissage est réalisé à l'aide d'un outil spécifique - le tisseur d'aspects (*aspect weaver*). Ce dernier produit un nouveau programme qui contient à la fois les fonctionnalités de références et celles fournies par les aspects.

Le programme de référence est défini dans un langage de programmation traditionnel, par exemple Java ou C. Il sert de référentiel pour introduire les points servant à ancrer les aspects. Il peut être considéré comme l'environnement ou bien le contexte d'exécution

des aspects attachés.

AspectJ [Kiczales *et al.*, 2001] est une implémentation représentative de l'AOP. AspectJ propose un langage à base de Java pour définir les aspects. Le programme de référence est également développé en langage Java. Dans AspectJ, l'endroit où l'on attache un aspect est appelé « *joint point* » et la manière d'ajouter le code d'un aspect au programme de référence est appelé « *advice* ».

Contrairement à AspectJ, JAC (Java Aspect Component) n'est pas un langage pour l'AOP mais un cadre de travail proposé par [Pawlak *et al.*, 2001]. Ce cadre de travail offre une gestion dynamique et flexible des aspects pour une application. Avec JAC, la localisation et le code ajouté d'un aspect sont définis séparément et peuvent ne pas avoir de référence au programme principal comme dans AspectJ.

AspectJ et JAC prennent les méthodes comme la granularité des points de jonction. Trois possibilités d'exécution d'une méthode d'aspect à un point de jonction (*joint point*) sont proposées : *before*, *after*, *around* pour désigner le fait que cette méthode d'aspect est exécutée avant, après ou à la place de la méthode associée au point de jonction.

Comme les aspects représentent la sémantique additionnelle au programme de référence, ils ne peuvent pas s'exécuter indépendamment du programme de référence. De plus, il est nécessaire d'avoir un mécanisme permettant de passer des paramètres entre le programme de référence et les aspects. AspectJ propose de s'appuyer sur les paramètres définis dans les points de jonction, ce qui n'est pas très intuitif mais relativement performant.

Bien que ses motivations principales soient la clarification du programme et la réutilisation, l'AOP est certainement une voie prometteuse pour l'adaptabilité des composants. Les aspects peuvent être conçus afin de faire évoluer et adapter le programme de référence pour un usage spécifique. Cependant, une fois que les aspects sont attachés, il est difficile de faire évoluer un programme. Même un changement mineur, tel que le renommage d'une méthode ou la modification d'une classe sur laquelle sont attachés les aspects, peut demander de localiser des aspects et éventuellement de changer la façon d'utiliser et d'implémenter ces aspects. De plus, comme la plupart des travaux sur les aspects limitent le tissage à la phase de compilation du programme (parce que ce processus demande l'accès

au code source du programme), l'AOP n'est pas appropriée pour l'adaptation dynamique.

2.3.2 *Model-Driven Architecture* MDA

L'approche MDA [Bezivin and Blanc, 2002] est une approche relativement récente, proposée par l'OMG [OMG, 1999a], qui souligne la réécriture de modèles. Cette approche se base sur le constat que les modèles existent partout dans le processus de construction des applications (de l'analyse jusqu'à l'exécution de celles-ci). Ces modèles peuvent être classifiés en deux groupes :

- **Le groupe 1** contient des modèles qui représentent l'application à différentes étapes de construction, tels que le modèle d'analyse, le modèle de conception, etc. ;
- **Le groupe 2** contient des modèles servant à vérifier ou contrôler l'application dans les différentes étapes de son cycle de vie. Le modèle de test, par exemple, propose des cas de tests qui sont destinés à vérifier la correction de l'application. Le modèle de déploiement, à son tour, spécifie comment déployer l'application. Chacun de ces modèles se concentre sur une préoccupation spécifique liée au processus de construction et d'exécution des applications.

Concernant les modèles du premier groupe, le MDA se base sur l'observation que ces modèles représentent l'application à différents étapes à divers niveaux d'abstraction et/ou à différents points de vue. Parmi ces modèles, il y a des modèles de type **PIM** (*Platform Independant Model*) et d'autres qui sont de type **PSM** (*Platform-Specific Model*). Ce qui est intéressant, c'est le fait qu'un modèle peut être construit à partir d'un autre modèle, grâce à la transformation de modèles. En conséquence, le MDA propose de considérer le processus de construction d'une application comme une chaîne de transformation de modèles à partir d'un modèle de base, lequel représente le plus haut niveau d'abstraction.

En se basant sur ces observations, le MDA propose de considérer le processus de construction d'une application comme une chaîne de transformation de modèles à partir du modèle métier qui est le modèle de plus haut niveau d'abstraction. Cette chaîne, illustrée dans la figure 2.7, contient des transformations directes qui s'appliquent sur un modèle PIM ou PSM, tels que :

- L'optimisation qui sert à améliorer un modèle selon des règles de conception dans

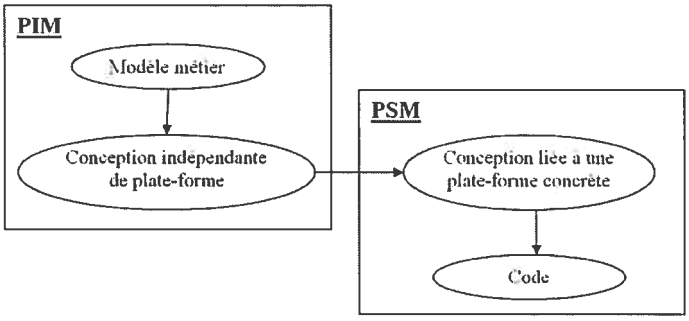

FIG. 2.7 – Transformation de modèles dans l'approche MDA

l'objectif d'en augmenter la lisibilité, l'efficacité ou bien la réutilisabilité ;

- Le raffinement qui consiste à ajouter d'autres informations dans un modèle pour qu'il devienne plus complet et plus détaillé. Par exemple, un objet abstrait peut être raffiné en un ensemble d'autres objets de granularités plus fines.

Des transformations indirectes peuvent également être trouvées dans cette chaîne de transformation. Il s'agit de la transformation de PIM vers PSM ou de PSM vers le code. Ces deux transformations indirectes ont pour but d'être fortement automatisées.

Les modèles du deuxième groupe, liés au contrôle et à la cohérence de l'application, sont jusqu'à présent développés, documentés et maintenus en dehors du code, ce qui implique un risque d'incohérence entre ces modèles et l'application à construire. Le MDA souhaite que ces modèles soient générés à partir de modèles du premier groupe, afin d'assurer la cohérence entre les modèles.

Par l'approche MDA, un changement réalisé sur un modèle de haut niveau pourrait être automatiquement répercuté dans des modèles de niveaux plus bas. Le changement de l'application est alors facilité par cette approche et le temps nécessaire à implémenter un processus d'adaptation est réduit. Pourtant, cette adaptation reste toujours statique. Chaque fois que l'on veut modifier une application, il faut d'abord modifier son modèle puis régénérer le code, ce qui rend inévitable le redémarrage des applications.

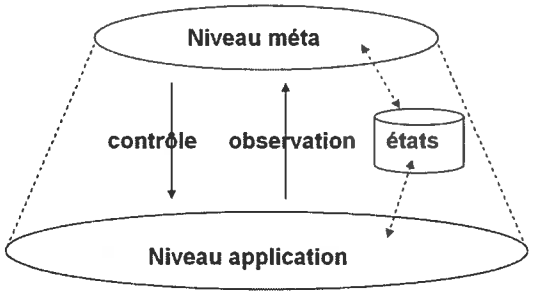

FIG. 2.8 – Deux niveaux de l'application réflexive

2.3.3 Réflexion et l'introduction du niveau méta

Transposée du domaine philosophique au domaine informatique dans le début des années 80s par Smith [Smith, 1982], la réflexion se présente de plus en plus dans des langages récents, tels que Smaltalk ou Java. Elle propose trois mécanismes différents : *la réification*, processus permettant d'encoder, à l'exécution, les données propres au système ; *l'introspection*, mécanisme permettant au système de s'observer et donc de répondre à des questions sur son état ; *l'intercession*, mécanisme permettant au système d'agir sur lui-même, et donc d'altérer sa propre interprétation.

Nous pouvons, par exemple, faire la distinction entre réflexion structurelle et comportementale, comme le fait Ferber [Ferber, 1989], ou alors même entre réflexion implicite ou explicite, suivant que le niveau de base a connaissance ou non de l'existence du niveau méta. Pour tous les cas, la réflexion introduit dans le système un nouveau niveau, fonctionnant *au-dessus* de l'exécution normale de l'application (figure 2.8). Ce niveau est appelé *niveau méta* [Kiczales *et al.*, 1991] [Zimmerman, 1996].

La fonctionnalité principale du niveau méta est d'observer l'état du système et de contrôler l'exécution du niveau de base. Son exécution peut être transparente pour le niveau de base, c'est-à-dire que son existence n'est pas connue par ce dernier, ou en interaction avec celle du niveau de base (les deux niveaux se connaissent et s'exécutent en échangeant les informations). Si le niveau méta ne fait que l'observation de l'état du système, l'exécution du niveau de base est relativement indépendante du niveau méta.

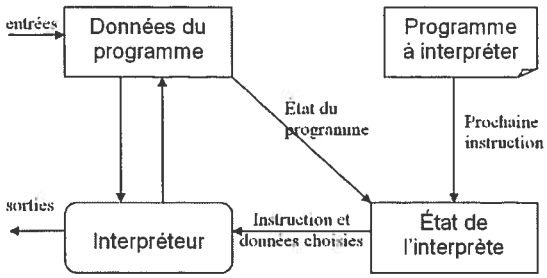

FIG. 2.9 – Principe d'exécution d'un interpréteur

Au cas où le niveau méta peut contrôler l'exécution du niveau de base, le système est capable d'agir sur lui-même. Cette observation et ce contrôle sont très importants pour des systèmes intelligents comme SMAs [Pitrat, 1990].

2.3.4 Interprétation et machine virtuelle

Il existe deux manières principales de modifier une application, la première consiste simplement à changer son code, la deuxième à changer son interprétation.

Dans le cas des langages compilés, pour changer le code, il est obligatoire de le recompiler. Cette recompilation pourrait éventuellement conduire à l'arrêt total de l'application, puis au redémarrage de celle-ci. Il est alors difficile de faire un changement dynamique du code compilé. Il existe pourtant des tentatives pour surmonter ce problème, telles que dans [Malabarba *et al.*, 2000]. Cette opération est simple s'il est possible de déterminer que le code modifié n'est pas en cours d'utilisation. Les choses se compliquent si ce n'est pas le cas. Ainsi, lorsqu'un langage à objets est utilisé, un scénario typique est la modification d'une classe dont il existe déjà au moins une instance. Dans un cadre qui garantit statiquement certaines propriétés (typage, sécurité), il s'agit aussi de garantir que la nouvelle version est compatible avec la version précédente. Traiter ces questions requiert un support d'exécution adapté, par exemple un certain nombre de modifications de la machine virtuelle Java dans le cas du chargement dynamique de classes proposé par [Malabarba *et al.*, 2000].

Un point intéressant est que, plutôt que de charger un code précompilé, il est possible d'effectuer une recompilation prenant en compte l'état de l'exécution ayant déclenché l'adaptation, et donc de faire de la génération de code dynamique [Keppel *et al.*, 1991]. Le problème de la génération de code dynamique est son coût potentiel dans le cas où il y a des contraintes de réaction à la demande d'adaptation. Une possibilité est de compiler les « briques » nécessaires à l'adaptation à l'avance pour juste les assembler de manière adéquate à l'exécution. La génération de code dynamique peut alors s'accompagner d'un processus de spécialisation, par exemple, en utilisant des techniques d'évaluation partielle, comme proposé dans [Consel and Noël, 1996]. En effet, la demande d'adaptation correspond habituellement à de nouvelles valeurs de paramètres significatifs du contexte d'exécution. Il est alors fondamental d'utiliser au mieux ces données.

Ces idées de génération dynamique de code et de spécialisation sont utilisées de manière très ciblée dans des supports d'exécution comme HotSpot [Paleczny *et al.*, 2001] qui adaptent en continu la qualité du code en générant dynamiquement un code plus efficace pour les points chauds repérés par profilage.

Dans le cas des langages interprétés, l'interpréteur charge et exécute les instructions de manière séquentielle (figure 2.9). Dès lors, la modification du code est particulièrement simple : il suffit que l'interpréteur recharge les morceaux du code à modifier (par exemple, par l'intermédiaire des méta-prédicats `assert/retract` en Prolog).

D'un certain point de vue, la combinaison d'un compilateur et d'une machine virtuelle n'est rien d'autre qu'un interprète. On voit donc que recompiler un programme (en supposant que la source ne change pas) peut être considéré, en référence au programme source, comme un premier type de modification du support d'exécution.

2.3.5 Composant

Actuellement, il n'y a pas de consensus sur la définition d'un composant. Toutefois, nous citons ici une définition représentative fournie par Heineman [Heineman, 2001] :

> *A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*

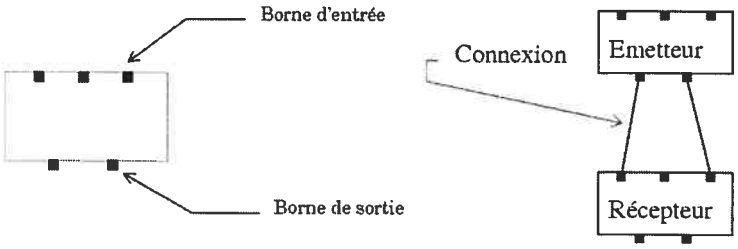

FIG. 2.10 – Modèle composant

Pour supporter les composants et les applications à base de composants, il est nécessaire de fournir un modèle de composant et un ensemble d'outils (cadre de travail) permettant de supporter le modèle de composant, depuis la construction, l'assemblage jusqu'à l'exécution de telle application [Lhuillier, 1998].

Dans la figure 2.10 [Lhuillier, 1998], un composant se présente comme une boîte noire, il cache sa structure interne. Tous ce qu'on peut voir de l'extérieur sont ses ports d'entrées et de sorties, dont la définition sont des interfaces.

Dans ce modèle, il est facile de remplacer un composant par un autre composant, à condition que ce dernier respecte les interfaces du premier. La dynamicité de composition est alors atteignable, pourtant la contrainte est assez stricte : les interfaces, qui sont non-modifiables, doivent toujours être respectées.

2.3.6 Services asynchrones

Depuis ces dernières années, le Web devient la plate-forme par laquelle beaucoup d'entreprises communiquent avec leurs partenaires et proposent leurs services aux clients. Par conséquent, les services Web, qui sont des applications indépendantes pouvant être publiées, localisées et accessibles par le Web [Yang and Papazoglou, 2002] [WSDL, 2002], ont commencé à apparaître. Les exemples de services Web sont des applications pour réserver un hôtel, acheter un billet d'avion, etc.

Individuellement, chaque service fournit une fonctionnalité limitée. Il est nécessaire de composer ces services afin d'obtenir un service composite proposant une fonctionnalité de

plus haut niveau d'abstraction. Il y a deux types principaux de composition de services Web :

- **Composition statique** : il s'agit d'un type de composition dans lequel les services Web à composer sont pré-calculés avant qu'un client fasse une requête du service composite. Ce type de composition peut être appliqué dans des environnements stables où les services Web participants sont toujours disponibles et où le comportement du service composite est le même pour tous les clients ;
- **Composition dynamique** : les services Web à composer sont déterminés lors de l'exécution de la requête du client. Ils peuvent être déterminés selon les besoins, les contraintes de chaque client, la disponibilité des services Web, etc.

Évidemment, la composition dynamique apparaît intéressante et très utile pour une éventuelle adaptation dynamique. D'une part, elle promet d'être capable de faire face à un environnement très dynamique dans lequel des services apparaissent et disparaissent rapidement. Ces services peuvent également être des services similaires. D'autre part, elle permet de mieux satisfaire les besoins de chaque client.

Cependant, cette technologie n'est applicable que pour un nombre limité d'applications, spécifiquement pour fournir des services simples avec peu d'interactions. De plus, elle demande une infrastructure Web qui n'est pas toujours disponible.

Un travail similaire se trouve dans l'approche Asynchronous Service [Oriol, 2002]. M.Oriol considère que les connexions entre différents composants représentent l'obstacle principal limitant l'évolution des logiciels au cours de leur exécution. Il propose une architecture dans laquelle tous les composants sont des services et ceux-ci interagissent à travers des communications asynchrones.

Ce modèle se base sur un gestionnaire de services qui maintient une base de données de description des services et réalise un *« matching »* entre une requête de service et les services disponibles. Le scénario d'invocation d'un service comporte les étapes suivantes (figure 2.11 [Oriol, 2002]) :

1. Une entité demande un service auprès du `Service Manager` ;
2. Le `Service Manager` fait un « *matching* » et génère un tag unique à cette requête ;

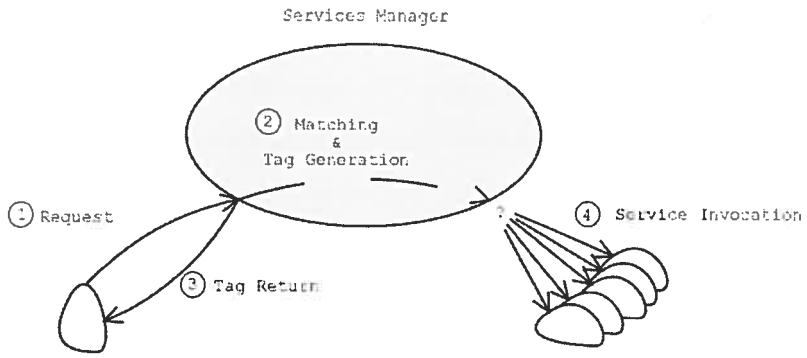

FIG. 2.11 – Modèle de services asynchrones

3. Ce tag est envoyé immédiatement à l'entité demandeuse. Ce tag pourrait être utilisé ultérieurement pour la réception de la réponse ;
4. Le `Service Manager` invoque le service choisi avec les arguments et le tag.

L'évolution dynamique peut alors se faire par la substitution dynamique de ces composants. Cependant, cette approche ne permet pas de spécifier explicitement les interactions entre les composants, il est dès lors impossible de les faire changer à la chaîne, dans le cas où le changement d'un composant doit conduire aux changements des autres.

2.4 Conclusion

Le tableau 2.2 fournit une synthèse sur les six techniques étudiées, selon les dimensions de l'adaptation présentées dans la section 2.2.1 : dynamicité (question *quand*), granularité (question *quoi*) et acteurs (question *qui*).

Techniques / Critères	Aspect	MDA	Réflexion & méta-modélisation	Interprétation	Composant	Service Web
Granularité	méthode	modèle	méthode	primitif	composant	service
Dynamicité	limitée	non	oui	oui	avec interfaces fixées	limitée
Acteurs	programmeur	programmeur	programmeur logiciel lui-même	programmeur logiciel lui-même	programmeur logiciel lui-même	programmeur logiciel lui-même

TAB. 2.2 – Tableau récapitulatif des technologies supportant l'adaptation

En ne tenant compte que de la dynamicité et de la granularité, les six approches

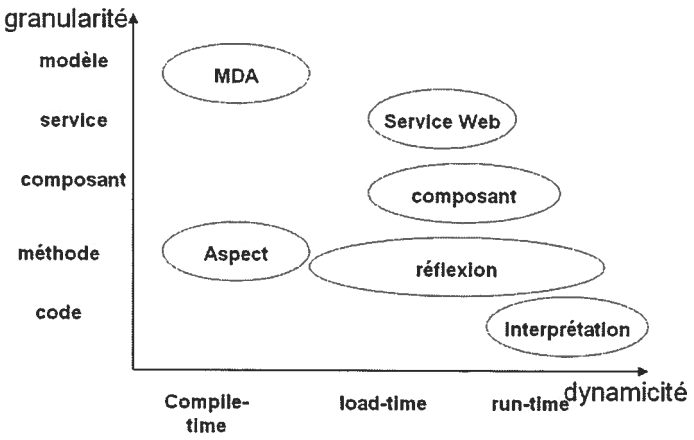

FIG. 2.12 – Espace des technologies d'adaptation

étudiées sont positionnées telles que dans la figure 2.12.

Nous voyons que les techniques les plus prometteuses pour l'adaptation dynamique sont la méta-modélisation, l'interprétation, les composants et les services Web.

Nous avons présenté jusqu'ici les notions fondamentales de l'adaptation des systèmes multi-agents et étudié différentes techniques candidates à l'adaptation dynamique. Avec les définitions de l'adaptation statique/dynamique et de l'adaptation fontionnelle/du contrôle, la recherche dans le cadre de cette thèse a été bien positionnée : c'est *l'adaptation dynamique du contrôle*. Le chapitre suivant vise à identifier et à classifier les éléments nécessaires du contrôle des SMAs, avant de proposer des solutions à l'adaptation dynamique de ceux-ci dans les chapitres 4, 5 et 6.

CHAPITRE 3

ANALYSE DE L'ADAPTATION DYNAMIQUE DU CONTRÔLE DES SYSTÈMES MULTI-AGENTS

La revue de la littérature dans le chapitre précédent a montré que l'adaptation est une caractéristique distincte et mise en avant des systèmes multi-agents. En soulignant l'aspect temporel de sa préparation, l'adaptation peut être catégorisée en deux types principaux : l'adaptation statique et l'adaptation dynamique. Alors que le premier type est assez limité et convient rarement à une situation réelle, le deuxième est le choix unique de l'adaptation dans un grand nombre de situations et attire de plus en plus l'intérêt des recherches dans le domaine des agents. Dans le cadre de cette thèse, nous sommes particulièrement intéressés à l'application de ce deuxième type sur le code de contrôle dans les systèmes multi-agents. Plus précisément, la modélisation de l'adaptation dynamique du contrôle des SMAs est le thème principal de la présente thèse.

De la même façon que « *One size does not fit all* », on a besoin de différentes adaptations pour les cibles ou les différents objectifs : par exemple, l'adaptation à la communication ne peut pas être la même que l'adaptation à la migration des agents. Afin de résoudre ce problème, notre stratégie est de classifier les adaptations dynamiques du contrôle en certains groupes, de modéliser séparément ceux-ci et de fournir un cadre de travail supportant ensemble ces modèles.

Comme nous l'avons indiqué dans le chapitre 2, en dehors de l'environnement, le monde d'agents est constitué de deux éléments : plate-forme et modèle d'agents (cf. la figure 2.3). Un modèle d'agents, quant à lui, est spécifié en deux niveaux : agent - niveau *micro* et SMA - niveau *macro* (figure 2.2). En conséquence, il y a trois niveaux dans l'architecture : plate-forme, agent et système multi-agents ; chacun d'entre eux s'occupe de différentes tâches. Dès lors, une approche classifiant les formes d'adaptation du contrôle à ces trois niveaux convient parfaitement à notre stratégie et devient notre choix naturel pour le commencement du travail. Cette classification nous permet de modéliser l'adaptation dynamique du contrôle de manière non seulement modulaire mais aussi progressive :

de niveau plate-forme monté jusqu'au niveau SMA, l'adaptation à un niveau plus élevé peut se baser sur l'adaptation au niveau inférieur.

Ce chapitre est organisé comme suit : la première section propose une classification de l'adaptation dans les SMAs en trois niveaux : plate-forme, agent et multi-agents. Les besoins d'adaptation du contrôle à chaque niveau sont ensuite identifiés. Les trois prochaines sections étudient des travaux connexes, ce qui permet de justifier et de détailler notre classification et identification. La dernière section fait une synthèse des tâches d'adaptation du contrôle à chacun des trois niveaux proposés.

3.1 Trois niveaux pour l'adaptation du contrôle

3.1.1 Adaptation du contrôle au niveau plate-forme

Une plate-forme multi-agents offre aux agents une infrastructure d'exécution et leur fournit des services de base pour trouver et interagir avec les autres agents ainsi que pour accéder à l'environnement. À côté de ces services de base, qui sont similaires entre les plates-formes (chapitre 4), une plate-forme spécifique peut offrir d'autres services à valeur ajoutée, tels qu'interface graphique, vérification et validation [Yoo, 2000], etc. Toutefois, l'environnement d'exécution et la façon d'accéder et d'utiliser ces services sont forts différents d'une plate-forme à l'autre. Par exemple, Telescript [Tardo and Valente, 1996] utilise un script propriétaire, tandis que Grasshopper [Grasshopper, 2000] utilise le langage de programmation orientée-objet Java, leurs environnements d'exécution sont évidemment incompatibles. Même si deux plates-formes utilisent un langage commun, l'accès aux services de base est également différent : un agent Voyager [Voyager, 1999] et un agent Jade [Jade, 2003] ne peuvent pas communiquer entre eux en utilisant le service de communication fourni par leur plate-forme correspondante.

Ces différences invoquent le problème de compatibilité : si un agent d'une plate-forme ne peut pas fonctionner sur une autre plate-forme, nous disons que ces deux plates-formes sont incompatibles.

Définition 5 (Compatibilité des plates-formes) *Les deux plates-formes multi-agents sont compatibles si elles offrent aux agents l'accès uniforme à un ensemble de services*

de base et leur fournissent des environnements d'exécution identiques, permettant à ces agents de fonctionner de façon identique sur les serveurs de ces deux plates-formes.

La compatibilité entre les plates-formes permet alors de rendre transparente la localisation des agents vis-à-vis de différents types de plates-formes, que ce soit en terme de programmation ou de déploiement.

FIPA [FIPA, 1998] a proposé un ensemble de normes visant à maximiser l'interopérabilité entre des plates-formes multi-agents. Cependant, cette interopérabilité ne souligne que l'aspect de communication entre différentes plates-formes et ne garantit pas l'exécution compatible des agents sur celles-ci. Alors, l'interopérabilité est une contrainte moins forte que la compatibilité : deux plates-formes peuvent être interopérables mais ne sont pas nécessairement compatibles. Nous voyons une similarité dans le cas des systèmes d'exploitation : une plate-forme Windows et une plate-forme Unix ne sont pas compatibles, mais elles sont interopérables et peuvent fonctionner ensemble dans un réseau.

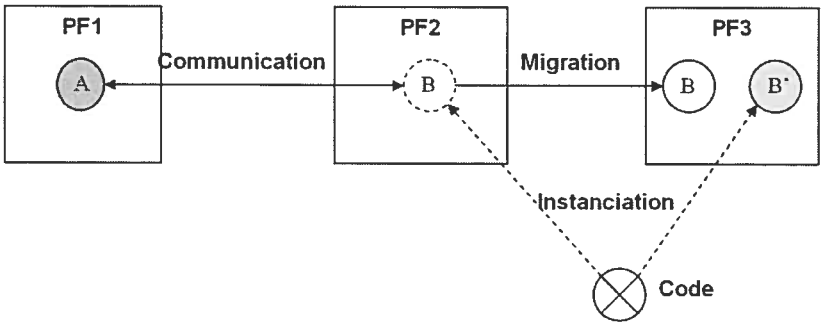

FIG. 3.1 – Compatibilité et interopérabilité

Dans la figure 3.1, l'agent `A` et l'agent `B` se trouvent sur deux différentes plates-formes `PF1` et `PF2`. Comme `A` et `B` peuvent communiquer, on dit que `PF1` et `PF2` sont interopérables. Les deux agents `B` et `B'` sont instanciés du même code mais sur deux plates-formes `PF2` et `PF3`, ces deux plates-formes sont dites compatibles et également interopérables. De plus, l'agent `B` peut migrer de la plate-forme `PF2` vers la plate-forme `PF3`, parce que celles-ci sont compatibles.

La compatibilité entre les plates-formes est un facteur indispensable pour le succès et la popularité du paradigme d'agents dans des environnements hétérogènes tels que Internet [Dillenseger, 1999]. Si les plates-formes multi-agents sont compatibles, les développeurs de SMAs ne devront plus se soucier des différences inter-plates-formes, et pourront se concentrer sur la logique de leurs applications. Or, les plates-formes d'agents existantes sont nombreuses, elles sont déjà déployées sur de multiples sites et pratiquement incompatibles. Dès lors, l'adaptation du contrôle à ce niveau est l'adaptation des agents aux plates-formes, mais non à l'inverse.

3.1.2 Adaptation du contrôle au niveau agent

À ce niveau se trouvent les modèles d'agents (autrement dit, la structure interne d'agents). Comme il existe de nombreux modèles intrinsèquement différents et le choix d'un modèle pertinent dépend largement des besoins spécifiques de chaque application, l'adaptation du contrôle à ce niveau se limite à fournir la flexibilité de l'agent afin de rendre celui-ci modifiable et extensible. Une fois que l'agent est modifiable et extensible, il devient donc adaptable.

- modifiabilité : un agent peut changer ses comportements et mettre à jour ses connaissances ;
- extensibilité : un agent peut voir ajouter de nouvelles fonctionnalités, retirer les fonctionnalités obsolètes ou mettre à jour celles-ci afin de mieux faire ses travaux.

Dès lors, l'adaptation du contrôle à ce niveau concerne la gestion non seulement des éléments internes de l'agent (sa structure, la représentation de ses comportements et de ses connaissances) mais aussi de ses fonctionnalités publiques. D'autres tâches d'adaptation à ce niveau sont optionnelles (dépendantes aux applications concrètes) et largement variables, elles ne peuvent pas être considérées comme l'adaptation du contrôle et ne seront pas évaluées.

3.1.3 Adaptation du contrôle au niveau multi-agent

Comme nous l'avons présenté dans la section 2.1.2, les deux dimensions principales au niveau multi-agents sont l'organisation des SMAs et l'interaction entre les agents membres

(figure 2.2). Évidemment, ce sont également les deux dimensions de toutes formes d'adaptation à ce niveau [Mamei and Zambonelli, 2003] [Capera *et al.*, 2003]. Par conséquent, une organisation flexible et une représentation explicite des interactions d'un SMA sont les deux thèmes de l'adaptation du contrôle à ce niveau.

3.2 Approches d'adaptation au niveau plate-forme

Le problème principal de l'adaptation au niveau plate-forme est la compatibilité entre les plates-formes multi-agents. Inspirées par la similarité de ce problème avec celui existant dans le domaine des systèmes d'exploitation, trois directions de recherche sont possibles [Magnin *et al.*, 2002b] :

1. Proposer un standard pour toutes les plates-formes multi-agents ;
2. Convertir un agent d'une plate-forme en un code exécutable sur d'autres plates-formes ;
3. Offrir un *middleware* permettant aux agents de fonctionner sur toutes les plates-formes multi-agents.

En dehors de ces trois approches, une quatrième serait de fournir une plate-forme universelle qui supporte tous les types d'agents. Cependant, elle ne semble pas réalisable : il faudrait installer cette plate-forme universelle sur tous les serveurs du réseau, et chaque fois qu'un nouveau type d'agent apparaît, la réinstallation de cette plate-forme serait obligatoire.

Les sections suivantes visent à analyser des travaux représentatifs de chacune de ces approches, afin de bien identifier leurs avantages, leurs inconvénients et les problèmes restant à résoudre.

3.2.1 Approches de standardisation

Jusqu'à maintenant, il existe deux familles de standard pour les agents : les spécifications de FIPA et les normes MAF de l'OMG. Tandis que les premières cherchent à rendre interopérables les plates-formes d'agents à travers la standardisation des services de base

d'une plate-forme ainsi que leurs interfaces d'accès, les dernières visent à atteindre un certain niveau de compatibilité des agents, notamment pour les agents mobiles.

3.2.1.1 Spécifications FIPA

FIPA (*The Foundation for Intelligent Physical Agents*) [FIPA, 1998] est une organisation internationale à but non lucratif destinée à promouvoir les technologies agents à travers le développement de spécifications qui maximisent l'interopérabilité entre les agents sur différentes plates-formes. Actuellement, ce sont les normes d'interopérabilité les plus abouties, notamment les spécifications portant sur la gestion des agents (*Agent Management*) et la communication des agents (*Message Transport* et *Agent Communication Language*) [Poslad, 2001].

Les spécifications de FIPA fournissent un cadre normatif dans lequel les agents FIPA existent et opèrent. Ces normes établissent un modèle de référence logique pour la création, l'enregistrement, la localisation, la communication, la migration et la destruction des agents, ce qui ressemble aux capacités d'une plate-forme multi-agent (figure 3.2 [FIPA Agent Management, 2001]). Dans ce modèle, les éléments sont des ensembles de services et aucune configuration physique n'est imposée. Les détails d'implémentation sont donc laissés au choix des développeurs.

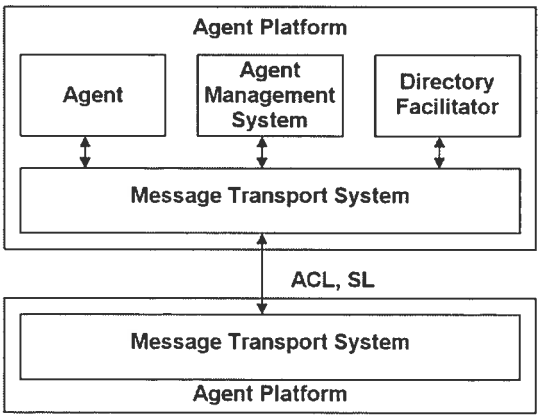

FIG. 3.2 – Modèle de référence de FIPA

Les éléments dans ce modèle sont les suivants :

Agent Platform (AP) fournit l'infrastructure physique dans laquelle des SMAs peuvent être déployés. Une AP se compose de machine(s), de systèmes d'exploitation, de trois services de base pour la gestion des agents (DF, AMS et MTS) et des agents.

Agent est l'élément fondamental dans une AP, lequel élément combine une ou plusieurs capacités dans un modèle d'exécution unifié et intégré (cela peut inclure l'accès à des logiciels externes, aux utilisateurs humains ainsi qu'aux moyens de communications entre agents).

Agent Management System (AMS) est un composant obligatoire de toutes les APs. L'AMS contrôle et surveille l'accès et l'utilisation des APs. Chaque AP doit et ne peut disposer au maximum que d'un AMS. L'AMS maintient un annuaire des identificateurs des agents inscrits à une AP. L'AMS offre des services des Pages Blanches à d'autres agents. Chaque agent doit s'inscrire à un AMS avant son exécution.

Directory Falicitator (DF) est un composant obligatoire de toutes les APs. Le DF fournit des services des Pages Jaunes à d'autres agents. Les agents peuvent enregistrer leurs services par le DF ou questionner le DF pour découvrir quels services sont offerts par d'autres agents. De multiples DFs peuvent coexister dans une AP et peuvent être fédérés.

Message Transport Service (MTS) est un composant qui fournit le service de communication entre les agents dans la même plate-forme et entre les différentes plates-formes, qui peuvent éventuellement être hétérogènes.

Une caractéristique unique des spécifications FIPA tient à la standardisation de la communication entre agents à travers un ACL (*Agent Communication Language*) commun. FIPA classifie trois niveaux d'hétérogénéité pour la communication intentionnelle entre agents :

1. Le transport des messages de communication : la couche TCP/IP ou socket, l'encodage des messages, etc. ;
2. Le langage : la syntaxe et la grammaire des messages (l'identificateur du message, l'adresse de l'agent destinataire, etc.), les protocoles de communication (actes de

langage) ;

3. Les sémantiques nécessaires pour interpréter le contenu des messages.

Au premier niveau, FIPA spécifie les mécanismes de transport de messages par différents protocoles de transport sur différents réseaux, y compris les réseaux sans fil.

Au deuxième niveau, FIPA résout le problème de l'hétérogénéité en proposant un langage de communication d'agents. Il y a deux approches pour la conception d'un langage de communication d'agent [Genesereth, 1997]. La première approche, que l'on retrouve dans les langages de programmation comme Java ou Tcl, est *procédurale*, *i.e.* base la communication sur un contenu exécutable. La deuxième approche, telle qu'adoptée dans KQML (*Knowledge Query and Manipulation Language*) [1], est *déclarative* : la communication est basée sur la théorie des actes de langage, tels la déclaration, l'assertion, etc. [Searle, 1969] [Vanderveken, 1990] [KSE, 1994]. Vu le succès de KQML, la deuxième approche a été choisie par FIPA pour concevoir un langage de communication d'agents, appelé FIPA ACL, lequel définit non seulement la syntaxe et la grammaire des messages échangés, mais également les sémantiques des actes de communication. Certains protocoles d'interaction sont aussi spécifiés à l'aide de ces normes, ce qui leur permet de supporter pleinement l'interaction entre agents.

Au niveau sémantique, FIPA définit le *FIPA Semantic Language* (SL) pour représenter le contenu d'un message. Sa syntaxe et ses sémantiques associées sont suggérées comme un candidat de la langue de contenu (*content language*) à utiliser conjointement avec FIPA ACL. Cependant, FIPA SL n'est pas un élément obligatoire pour toutes les implémentations. Il faut noter que ce langage est un formalisme de représentation à usage universel, applicable dans de nombreux domaines d'agents.

3.2.1.2 OMG Mobile Agent Facility (MAF)

OMG (*Object Management Group*) [OMG, 1999a] propose le standard MAF visant à la standardisation de l'interopérabilité entre les plates-formes d'agents mobiles hétérogènes

[1] KQML [Labrou and Finin, 1994] est le premier langage de communication basé sur la théorie des actes de langage et un standard *de facto* pour l'échange, entre logiciels (agents), d'informations, de connaissances ou de services.

et à la réutilisabilité des services CORBA dans le monde des agents. Tandis que FIPA s'intéresse aux vues logiques et aux services offerts par les agents, MAF est un standard destiné à traiter de la migration des agents et de leur vue physique, car il était prévu au départ pour des agents mobiles [Bellavista and Magedanz, 2001]. MAF ne propose pas une nouvelle plate-forme : l'idée derrière cette norme est d'obtenir un certain degré d'interopérabilité entre les plates-formes d'agents mobiles de différentes compagnies avec le moins de modifications possibles. MAF est prévu pour être utilisé comme un module s'ajoutant aux systèmes existants. Le standard inclut les spécifications IDL CORBA pour le transport et la gestion des agents.

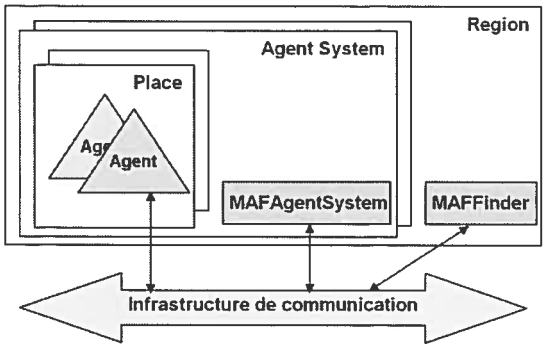

FIG. 3.3 – Interfaces dans MAF

Comme indiqué dans la figure 3.3, MAF adopte les concepts de *places* et *régions* qui ont été utilisés dans de nombreuses plates-formes d'agents existantes (Grasshopper, SOMA). Une *place* groupe les fonctionnalités dans un système d'agents et encapsule certaines capacités et restrictions des agents. Une *région* facilite la gestion d'une plate-forme en spécifiant un ensemble de serveurs d'agents sous le contrôle d'une seule autorité.

MAF ne mentionne pas la standardisation des opérations locales comme l'interprétation, la sérialisation et l'exécution des agents. Au lieu de cela, MAF propose seulement la standardisation de l'identification des agents et des serveurs d'agents, les types des systèmes et la syntaxe de localisation. MAF spécifie deux interfaces :

MAFAgentSystem contrôle certaines opérations de gestion des agents, notamment

celles de mobilité. Un objet `MAFAgentSystem` doit collaborer avec les services des systèmes d'agents concrets et offrir son interface aux utilisateurs externes. N'importe quel système externe peut contrôler des agents d'un système par le biais de cette interface. MAF définit des méthodes pour la suspension, la reprise et la terminaison des agents et pour la migration des agents d'une plate-forme à une autre. Pourtant, le transport des agents entre différents systèmes n'est pas encore complètement spécifié dans le standard, ce qui limite la possibilité d'échange d'agents entre systèmes provenant de différents vendeurs. L'interopérabilité complète n'est obtenue que lorsque les deux systèmes présentent une base de compatibilité (la même langue d'implémentation ou un mécanisme d'externalisation compatible). `MAFAgentSystem` est similaire à AMS dans les spécifications FIPA ;

MAFFinder supporte la localisation d'agents et de serveurs d'agents dans une région. `MAFFinder` fournit le service de nommage pour les agents, car le *Naming Service* existant dans CORBA n'est pas approprié pour les agents qui sont intrinsèquement et fréquemment mobiles. Le rôle de `MAFFinder` ressemble à celui de DF dans les spécifications FIPA.

La communication entre agents est hors de portée de cette spécification. Il faut noter que la spécification courante ne représente que les premiers efforts pour la standardisation des plates-formes multi-agents, notamment les agents mobiles.

Dans l'approche de standardisation, les plates-formes doivent se conformer à des normes afin d'être capables d'interopérer entre elles. Cependant, ces normes sont définies pour de nouvelles plates-formes à venir, et cela peut prendre beaucoup de temps avant que ces plates-formes ne convergent vers un ensemble cohérent de normes. En effet, si une communication standardisée entre les plates-formes semble assurée par de telles normes, le développement et le déploiement des agents doivent être réalisés au cas par cas, en fonction des types de plate-forme présents. Bien que la norme MAF essaie de standardiser la migration entre plates-formes, elle ne garantit pas pour autant la compatibilité d'exécution des agents entre celles-ci. Par conséquent, deux plates-formes supportant cette même norme risquent d'être toujours incompatibles. Pour faire un parallèle avec le monde

de l'informatique classique, des ordinateurs reliés entre eux par un protocole commun se sont mis en place, sans toutefois utiliser un système d'exploitation uniforme : sur chacun d'entre eux, la programmation (des applications / agents) doit être réalisée de façon différente.

3.2.2 Approches de conversion

Par cette direction, un agent essayera de convertir soit sa communication, soit lui-même, en fonction de la plate-forme cible, afin de survivre dans un environnement multi-plates-formes. Lorsque la première approche souligne l'interopérabilité entre les agents fonctionnant sur différentes plates-formes, la deuxième se focalise sur la migration d'un agent entre plusieurs plates-formes, c'est-à-dire la compatibilité inter-plateforme. La différence entre ces deux approches est similaire à celle entre les deux approches de standardisation présentées dans la section précédente.

3.2.2.1 Passerelle de traduction entre plates-formes multi-agents

Cette approche est utilisée pour rendre interopérables la plate-forme OAA (*Open Agent Architecture*) [Martin *et al.*, 1999] et la plate-forme RETSINA [Sycara *et al.*, 2001]. Dans ces deux plates-formes, un système multi-agents est une fédération d'agents. OAA est organisée autour d'un agent spécial, appelé Facilitator, qui gère toutes les communications entre les agents dans la fédération (il n'y a jamais de communication directe entre deux agents OAAs). Dans le cas de RETSINA, un agent spécial Matchmaker permet aux agents de la fédération de trouver les autres et de communiquer directement entre eux. Le RETSINA-OAA InterOperator [Sycara *et al.*, 1998] joue le rôle d'une passerelle entre un système RETSINA et un système OAA, et it permet à n'importe quel agent dans le premier système d'accéder à n'importe quel service ou information fourni par des agents dans le deuxième système, et réciproquement. Pour réaliser cette tâche, InterOperator déclare les agents RETSINA avec OAAFacilitator et les agents OAA avec RETSINA Matchmaker (figure 3.4 [Giampapa *et al.*, 2000]). De plus, il traduit les messages de communications entre les deux systèmes.

Cette approche semble avoir beaucoup de limitations. Elle n'est applicable que dans

les cas où les systèmes d'agents utilisent le modèle de fédération et n'est pas appropriée dans le cas des systèmes d'agents entièrement distribués. De plus, cette approche n'est pas scalable : il faut implémenter $n(n-1)$ mécanismes de transformation afin d'obtenir la compatibilité totale entre n plates-formes. De plus, toutes les traductions des communications doivent être réalisées à un seul endroit (l'agent InterOperator), ce qui limite la possibilité d'utiliser cette solution à grande échelle.

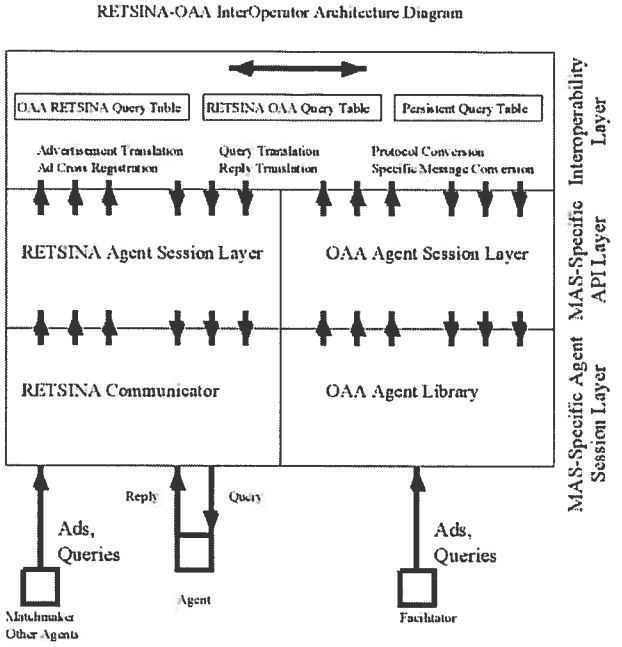

FIG. 3.4 – Diagramme d'architecture de l'agent InterOperator

3.2.2.2 Conversion entre plates-formes multi-agents

Cette approche a été proposée dans [Tjung *et al.*, 1999]. Elle consiste à effectuer un travail de rétro-ingénierie du code de l'agent se trouvant sur la plate-forme source afin de pouvoir le modéliser en vue de le transformer en un code compilable par la plate-forme de destination. Dans leur étude de cas, Tjung et al. ont expérimenté la transformation des agents Aglets en agents Voyager. En réalité, ils ont essayé de faire un « *mapping* »

des appels des fonctions de la plate-forme Aglets vers ceux de la plate-forme Voyager.

Cette solution pose toutefois un grand nombre de difficultés. D'une part, elle est trop exigeante et insécuritaire : elle demande que le code source des agents soit ouvert aux serveurs et que le *mapping* de fonctions soit possible. D'autre part, elle n'est pas du tout scalable : si une nouvelle plate-forme apparaît, il faut implémenter $2n$ nouvelles conversions de et vers les n plates-formes existantes. Enfin l'opération de migration demande un pré-traitement et prend beaucoup de temps.

3.2.3 Approche *middleware*

À notre connaissance, le seul travail utilisant cette approche, en dehors de nos travaux, est le projet CoABS (Control of Agent-Based System) [CoABS, 2000]. Cette approche propose le concept *grid* [Kahn and Cicalese, 2002][2]. *Grid* est un *middleware* qui intègre les systèmes d'agents hétérogènes, les applications à base d'objets et les systèmes préexistants (*legacy systems*). Il inclut une interface des méthodes de programmation d'application pour enregistrer des agents, pour annoncer leurs capacités, pour découvrir des agents avec des capacités spécifiques et pour envoyer des messages entre agents. De plus, il offre un service de sécurité permettant d'authentifier des agents et d'encrypter les messages échangés.

Ce travail rend compatible les plates-formes en installant des *grids* sur les serveurs d'agents afin de fournir des services de base et des *Grid Service Helpers* sur les sites clients pour accéder à ces services (figure 3.5 [CoABS, 2000]). Cependant, il ne travaille qu'avec les plates-formes des communautés de recherche qui sont membres de ce projet. De plus, il n'y a pas beaucoup de documents techniques sur ce projet, celui-ci étant un projet militaire.

[2]Il est à noter que le concept *grid* ici n'est pas le même que celui dans « *grid computing* », bien qu'il y ait de similitudes entre eux.

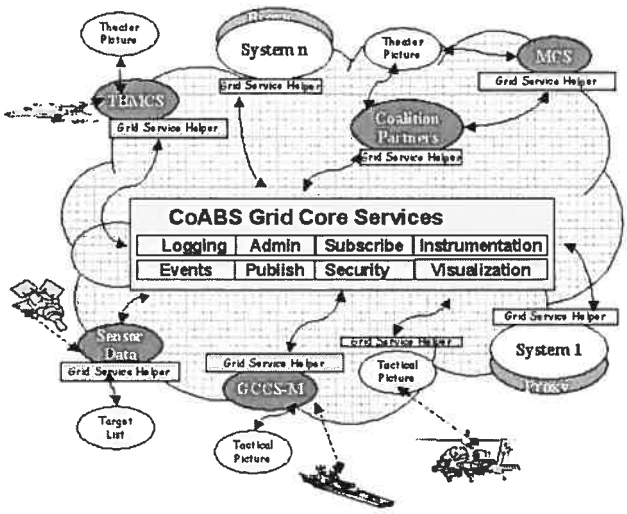

FIG. 3.5 – CoABS Grid

3.3 Approches d'adaptation au niveau agent

Cette section présente trois travaux représentatifs de l'extensibilité des agents en se basant sur les composants et un modèle d'agent générique avec une structure interne explicite et modifiable.

3.3.1 AgentComponent

Krutisch [Krutisch *et al.*, 2003] introduit une architecture d'agent extensible, nommé AgentComponent. Un AgentComponent (AC) se compose d'une structure de base en plus de « *slot* » d'extension (figure 3.6). La structure de base se compose d'une base de connaissance pour contenir les croyances de l'agent, de *slots* de communication, de *slots* d'ontologies et de *slots* de processus (Process Component - PC). Chaque *slot* de communication gère la communication avec un autre agent et un *slot* d'ontologies représente une ontologie pour un domaine spécifique. Tous les processus seront exécutés par des PC. Par conséquent, un AC est un composant générique, qui est réutilisable et configurable par changements des *slots* de communication, d'ontologies ou de processus. Ces slots peuvent être ajoutés, retirés ou remplacés à n'importe quel moment dans la vie de l'agent, alors

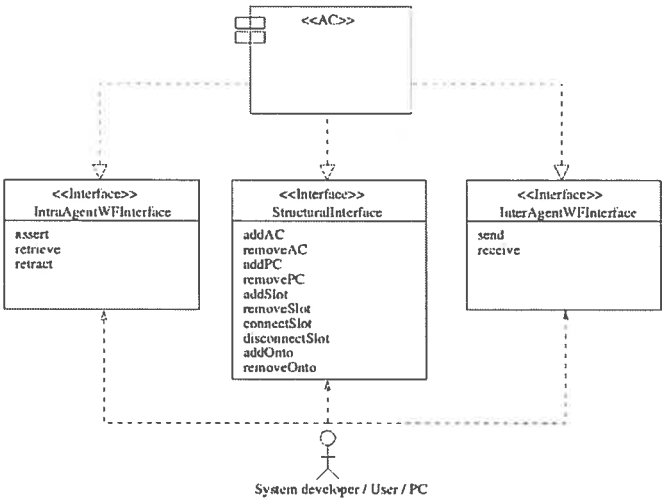

FIG. 3.6 – Architecture d'un AgentComponent

les ACs deviennent également extensibles.

Cependant, l'extensibilité d'un AC est limitée aux types de *slots* connus ; il est impossible d'introduire un nouveau type de *slots* après l'entrée en fonction de l'agent.

3.3.2 AgentFactory

Brazier présente une approche originale, appelée AgentFactory, permettant de générer des agents différents à partir des mêmes éléments [Brazier *et al.*, 2002]. Le principe de cette approche est que le développeur peut écrire plusieurs composants de base d'un agent et ceux-ci seront assemblés différemment en fonction du contexte concret au moment de sa création (figure 3.7).

Un agent peut être généré par AgentFactory s'il peut être décrit à deux niveaux distincts : le niveau conceptuel (la liste des composants, leurs interfaces et les interactions entre eux) et le niveau détaillé (le code des composants, la définition des interfaces). Il n'y a aucune contrainte sur le langage de développement utilisé pour le développement des éléments, lequel peut être Java, C++, etc. Pourtant, la flexibilité de cette approche se trouve au niveau de conception et de déploiement et est alors très limitée.

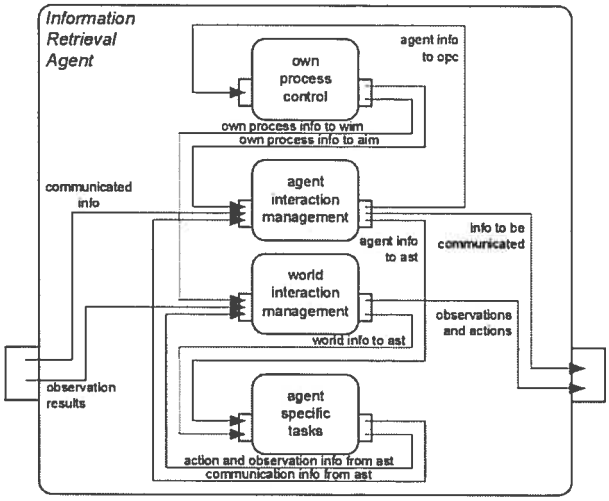

FIG. 3.7 – AgentFactory

3.3.3 Architecture à base de composants

L'architecture à base de composants est proposée par Lhuillier dans [Lhuillier, 1998] et illustrée dans sa plate-forme Maleva (*Modular Architecture for Living and Evoluting Agents*)[3]. Son approche vise à lier réutilisabilité et flexibilité tout en conservant les qualités respectives des architectures plus classiques telles que les architectures à base d'objets, d'acteurs ou les architectures modulaires. De ce fait, il est possible de concevoir des agents, suivant la catégorie d'applications à laquelle ils seront destinés, à partir des composants de base, de manière flexible.

Dans cette architecture, un agent est un regroupement de petites structures, appelées *composants*, lesquelles possèdent à la fois leur propre comportement, leur propre interface et leur propre contrôle. Plus précisément, un composant Maleva est une entité définie par un comportement interne, par un ensemble de bornes d'entrées, par un ensemble de bornes de sorties, et par un gestionnaire de message (figure 3.8 [Lhuillier, 1998]). Le comportement interne définit des méthodes particulières représentant ses bornes d'entrées/sorties.

[3]Une autre architecture componentielle qui est similaire mais plus simple se trouve dans [Horling and Lesser, 1998].

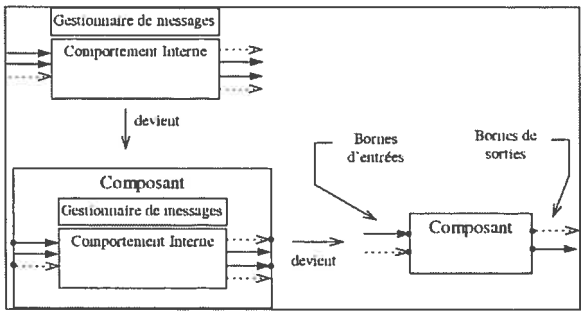

FIG. 3.8 – Un composant Maleva

Ces méthodes sont indiquées au gestionnaire de message, lequel les rend accessibles aux autres composants. Les bornes d'entrées/sorties sont de deux natures différentes, en trait continu sont représentés les flots de données et en pointillé sont représentés les flots de contrôle. Un composant n'est pas un processus autonome, il n'est activé que s'il reçoit un événement de contrôle. Celui-ci peut-être un contrôle externe (les flots de contrôle) ou un contrôle interne (transmis périodiquement par un *Scheduler* qui gère l'ensemble des composants du système).

Les interactions entre composants se font par l'envoi de messages. Chaque message est transféré d'un composant à un autre par un canal spécifique nommé Connexion. Ces connexions doivent être spécifiées lors de la conception des agents.

Les composants peuvent être combinés soit par la composition, où le composant résultat ne définit pas ses propre bornes mais permet l'accès à la totalité de celles de ses sous composants, soit par l'agrégation, où le composant résultat possède ses propres bornes et les sous-composants sont invisibles.

3.3.4 Architecture d'agents génériques

Vidal et Buhler dans [Vidal, 2001] proposent une architecture d'agent générique pour des agents développés dans des langages orienté-objet, ce qui fournit le support pour les conversations, les comportements réactifs et à long terme des agents. Ils supposent que l'environnement est non-déterministe, les agents perçoivent leur environnement à des

intervalles discrets et prennent des actions discrètes. Les éléments de ce modèle sont les suivants :

Input sont des informations que l'agent reçoit de l'extérieur. Il y a trois types d'*Input* : *SensorInput* est l'ensemble des données d'entrées que l'agent reçoit directement de ses senseurs, *Message* encapsule l'information échangée entre les agents, *Event* est une forme spéciale d'*Input*, qui représente un événement interne de l'agent ;

Activity représente des briques de base pour construire un agent. Il existe deux types d'activité : *Behaviour* et *Conversation*. Les *behaviours* sont des *activities* à long terme des agents, ils sont composés d'une série d'activités atomiques. Les *conversations* modélisent les protocoles de communication entre deux agents. Chaque *activity* possède une méthode *inhibit()*, qui représente une forme simple de validation de l'activation d'une *activity* ;

ActivityManager gère toutes les *activities*, il reçoit tous les *inputs* et les distribue aux *activities*. C'est lui qui joue le rôle d'un *Scheduler* des *activities* leur permettant d'être activées. C'est aussi lui qui gère tous les liens possibles entre les *activities* et les *inputs*.

À partir de ce modèle, il est possible d'obtenir le modèle Subsumption [Brooks, 1991]. Le modèle BDI [Rao and Georgeff, 1995] peut être également conçu à partir de ce modèle, toutefois avec certaines petites extensions. Cependant, il y a de nombreuses limitations dans ce modèle. Tout d'abord, il est conçu pour la phase de conception, dès lors il lui manque toute la dynamicité lors de la phase d'exécution des agents. De plus, il laisse au développeur la charge de résoudre tous les problèmes concernant la distribution des *inputs* aux *activities*, sans spécifier la méthode ou l'outil de validation des relations de dépendances entre celles-ci.

3.4 Approches d'adaptation au niveau système multi-agents

Dans cette section, nous étudions une architecture hiérarchique des agents, un méta-modèle d'organisation des agents et un modèle d'interaction dans les SMAs.

3.4.1 Architecture hiérarchique

Une architecture hiérarchique de systèmes multi-agents est proposée dans MAGIQUE[4] [Routier and Mathieu, 2001] . Cette structure rend possible le mécanisme de *délégation de compétences* entre agents, facilitant ainsi le développement. 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 échanges 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. L'ajout de nouvelles compétences (et fonctionnalités) dans une application SMA est donc facilité.

L'ensemble des agents sont regroupés au sein d'un SMA organisé selon une structure hiérarchique. Dans une hiérarchie, les agents feuilles sont appelés spécialistes et les autres superviseurs. Ces derniers doivent être capables (i.e. avoir la compétence pour) de gérer leur équipe d'agents (la sous-hiérarchie) dont ils sont la racine (figure 3.9).

Une hiérarchie représente en fait le support par défaut du réseau des communications entre les agents. Un lien hiérarchique représente donc l'existence d'un canal de communication bidirectionnel entre ces agents, et lorsque deux agents d'une même structure hiérarchique communiquent, le chemin emprunté par les messages qu'ils s'échangent colle, par défaut, à la hiérarchie.

L'organisation des agents peut être amenée à évoluer dynamiquement si une réorganisation de la structure du SMA se justifie en cours d'utilisation.

Cette organisation hiérarchique offre un mécanisme simple de délégation entre les

[4]MAGIQUE signifie pour Multi-AGent hiérarchIQUE.

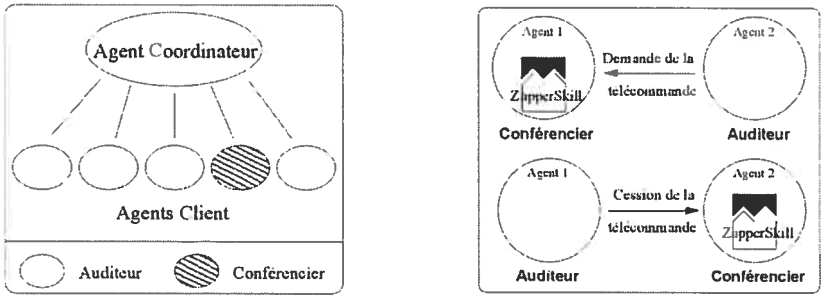

FIG. 3.9 – Une hiérarchie d'agents MAGIQUE et le transfert des compétences

agents présents dans le SMA. Un agent a une tâche à accomplir qui requiert l'exploitation d'un certain nombre de compétences qu'il peut ou non posséder en propre. Dans ce dernier cas, la délégation éventuelle de la réalisation à un autre agent est transparente et prise en compte par l'organisation hiérarchique. Cela a pour conséquence qu'un agent cherchant à déléguer la réalisation d'une tâche n'a pas nécessairement à connaître explicitement l'agent qui réalisera effectivement cette tâche : c'est l'organisation qui se charge de le trouver pour lui. Les invocations ne sont pas nommées et on n'a pas nécessairement à savoir qui fait quoi, c'est l'organisation qui s'en charge. Le principe en est le suivant :

- Si l'agent possède la compétence, il l'invoque directement ;
- Si l'agent ne possède pas la compétence, plusieurs cas de figures sont possibles :
  - Il a une accointance particulière pour cette compétence, il lui demande alors de la réaliser pour lui,
  - Sinon, s'il est superviseur, et qu'un membre de sa hiérarchie possède cette compétence, il transmet (de manière récursive 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 pour lui et celui-ci réapplique ce même mécanisme de manière récursive.

3.4.2 Méta-modélisation d'organisations de systèmes multi-agents

Ferber et Gutknecht proposent un méta-modèle d'organisations et un ensemble des outils pour l'analyse, la conception et l'exécution des SMAs [Ferber and Gutknecht, 1998] [Gutknecht *et al.*, 2001]. Ce méta-modèle est basé sur trois concepts : l'agent, le groupe et le rôle (figure 3.10).

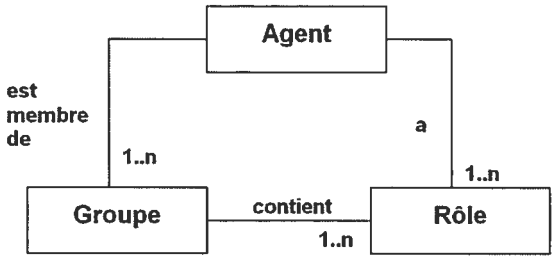

FIG. 3.10 – Modèle Aalaadin ou AGR (Agent - Groupe - Role)

Agent Quasiment aucune contrainte n'est posée sur l'architecture interne ou sur le modèle de comportement de l'agent. L'agent est simplement décrit comme une entité autonome communicante qui joue des rôles au sein de différents groupes. La très faible sémantique associée à l'agent dans ce modèle est volontaire. L'approche consiste à laisser la définition d'agent en retrait, non seulement pour ne pas prendre part au débat de qu'est-ce qu'un agent, mais surtout pour laisser au concepteur la liberté de choisir l'architecture interne appropriée à son domaine d'application.

Groupe Le concept de groupe est souvent rencontré lors de la conception et l'implémentation des SMAs [Baumann and Radouniklis, 1997] [Hannoin *et al.*, 1999], parce qu'un groupe facilite la communication entre agents et l'attribution des tâches parmi ceux-ci [DeLoach, 1999] [Kendall, 2000]. Le groupe est la notion primitive de regroupement d'agents. Chaque agent peut être membre d'un ou plusieurs groupes. D'une façon un peu simpliste, un groupe peut être vu comme un moyen d'identifier par regroupement un ensemble d'agents. Plus classiquement, associé au rôle, il définira la structuration organisationnelle d'un SMA usuel. Les différents groupes peuvent

se regrouper librement.

Rôle Le rôle est une représentation abstraite d'une fonction, d'un service ou d'une identification d'un agent au sein d'un groupe particulier. Chaque agent peut avoir plusieurs rôles, un même rôle peut être tenu par plusieurs agents, et les rôles sont locaux aux groupes. L'hétérogénéité des situations d'interaction est rendue possible par le fait qu'un agent peut avoir plusieurs rôles distincts au sein de plusieurs groupes, les communications étant clôturées par les groupes.

Ce modèle introduit une forme de réflexivité organisationnelle, représentée par trois principes complémentaires :

Appartenance multiple Un agent peut appartenir à la fois à un groupe orienté domaine et à un groupe de méta-niveau qui va gérer les ressources et les activités relatives à la gestion des agents et groupes. Les opérations de niveau méta sont gérées via l'appartenance à un groupe de niveau méta.

Agentification des services Les services de niveau système sont représentés dans ce modèle par des agents qui jouent des rôles spécifiques dans des groupes de niveau méta. Par exemple, la mobilité est fournie par un agent spécifique ayant un rôle de *migrateur* dans un groupe *mobilité*.

Notion de représentant Un agent peut être un représentant d'un groupe A dans un groupe B. Il est donc possible de transformer un agent en groupe par le biais d'un lien le représentant, et une délégation de tâche par l'agent vers les agents du groupe représenté. Ce groupe reste complètement opaque au sein du groupe initial, car le seul lien est fait par le modèle de comportement de l'agent le représentant.

Ce méta-modèle a plusieurs avantages. Premièrement, il permet de décrire n'importe quel type d'organisations des SMAs en se basant seulement sur les concepts agent, groupe et rôle. Deuxièmement, il n'impose aucune contrainte sur le modèle d'agents et de ce fait supporte l'hétérogénéité la dynamicité dans l'architecture des agents. Troisièmement, il est basé sur des sémantiques formelles et opérationnelles [Ferber and Gutknecht, 1999].

Cependant, ce modèle n'aborde que l'aspect organisationnel des SMAs. Les autres aspects, notamment le contrôle dans le modèle, sont quasiment absents.

3.4.3 Modélisation des interactions dans les systèmes multi-agents

Dans sa thèse de doctorat, Dury présente le modèle d'interaction Opéra, qui est constitué en réalité de deux différents modèles : un modèle de description des interactions et un modèle d'exécution de celles-ci [Dury, 2000]. Il définit une interaction entre plusieurs agents comme l'endossement, par chacun de ces agents, d'un rôle donné pendant un certain temps. L'endossement d'un rôle par un agent définit pour l'agent le modèle de comportement que celui-ci suivra durant le déroulement de l'interaction. Un agent endosse un rôle donné en fonction de la présence, dans l'environnement, d'une certaine situation déclenchante pour ce rôle. Un rôle est donc une paire « situation déclenchante - modèle de comportement associé à cette situation », et une interaction est une collection de rôles conçus pour être endossés simultanément, par un ou plusieurs agents. Ce modèle fait l'hypothèse que l'environnement dans lequel les interactions se déroulent existe au préalable, ainsi que les agents eux-mêmes qui sont définis, au minimum, comme des entités capables de perceptions et d'actions dans leur environnement.

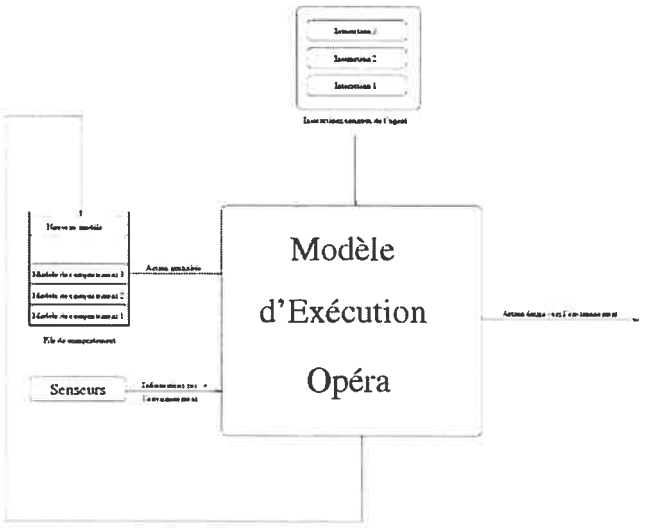

FIG. 3.11 – Modèle d'exécution Opéra

Opéra ne suppose pas que ces agents soient dotés, au préalable, d'un quelconque mo-

dèle de comportement, mais le cas échéant il offre un moyen de combiner ce modèle de comportement existant avec le modèle d'exécution des interactions qu'il fournit. Ce modèle considère un ensemble de modèles de comportement, devant être gérés simultanément, et les ordonne selon leur ordre de déclenchement. Lorsqu'un modèle de comportement devient actif, il est donc empilé au-dessus des modèles en cours à ce moment, et il prend le contrôle total de l'agent. Lorsqu'il se termine, il rend la main au modèle activé précédemment. On définit de cette façon la notion de pile de comportements (figure 3.11). En cours de son exécution, on donne à ce modèle le contrôle total de l'agent afin de garantir la cohérence du comportement de l'agent, et régler de ce fait le problème de l'accès aux effecteurs : un seul modèle de comportement accède, à chaque instant, aux effecteurs de l'agent. Son accès peut néanmoins être suspendu à tout moment par le déclenchement provisoire d'un autre modèle de comportement, jugé sur l'instant plus prioritaire.

3.5 Conclusion

Dans les sections précédentes, nous avons classifié l'adaptation du contrôle en trois niveaux distincts : plate-forme, agent et système multi-agents. Nous avons étudié différents travaux liant à l'adaptation du contrôle à ces trois niveaux afin d'en tirer les tâches d'adaptation du contrôle à chacun d'eux, lesquelles sont résumées dans le tableau 3.5.

Niveaux d'adaptation	**Tâches à réaliser**
Plate-forme	Compatibilité entre plates-formes - environnements d'exécution compatibles - accès uniforme aux services de base des plates-formes
Agent	- modifiabilité des éléments internes de l'agent - extensibilité de fonctionnalités publiques de l'agent
SMA	- organisation - interaction

TAB. 3.1 – Tâches d'adaptation du contrôle aux trois niveaux

Nous allons détailler dans les prochains chapitres nos propres solutions pour les tâches identifiées dans ce tableau : le chapitre 4 sera consacré au niveau plate-forme, alors que le chapitre 5 et 6 seront consacrés, respectivement, au niveau agent et au niveau SMA.

CHAPITRE 4

ADAPTATION PAR L'UNIFORMISATION DES PLATES-FORMES

Bien que les agents soient autonomes, ils ne peuvent pas fonctionner sans le support de plates-formes d'agents (section 2.1). Dans des applications SMAs distribuées et complexes, l'hétérogénéité de ces plates-formes est une réalité inévitable, à cause de la répartition de la gestion, de l'indépendance des participants, etc. Or, ces plates-formes ne sont pas *a priori* compatibles. Dès lors, l'adaptation aux plates-formes incompatibles devient un grand défi pour le développement des applications multi-agents.

Comme nous l'avons justifié dans le chapitre précédent, l'adaptation des agents aux plates-formes incompatibles est le problème principal de l'adaptation du contrôle au niveau plate-forme. Cette adaptation consiste en deux tâches : fournir un environnement d'exécution compatible sur différentes plates-formes et uniformiser l'accès aux services de base de celles-ci.

Si les plates-formes d'agents sont développées en différents langages, les environnements d'exécution fournis par ces plates-formes ne sont pas homogènes. Par conséquent, la première tâche est extrêmement difficile, voire impossible, à résoudre de manière systématique. Alors, l'hypothèse de notre recherche est que *les plates-formes d'agents soient implémentées dans un même langage de programmation*, ce qui garantit que *les environnements d'exécution des agents sont homogènes sur toutes les plates-formes.*

Jusqu'au début des années 1990, les plates-formes d'agents réelles ont été relativement limitées, à cause des difficultés dans l'implémentation des modèles théoriques. Avec la naissance du langage de programmation Java en 1995, de nombreuses plates-formes ont été introduites [Aglets, 1998] [Frost, 1997] [Fipa-os, 2000] [Grasshopper, 2000], [Jade, 2003] etc. Java est alors devenu le langage *de facto* pour l'implémentation des plates-formes multi-agents, grâce à sa puissance, son support pour la programmation distribuée (indépendance des systèmes d'exploitation, sécuritaire, programmation multi-thread, chargement dynamique de code, etc.) et sa popularité [Lange, 1998]. Tenant compte de cette réalité, notre hypothèse n'est plus exigeante et la tâche de fournir un environnement

d'exécution compatible consiste désormais à fournir aux agents un cycle de vie uniforme sur toutes les plates-formes. Comme la gestion du cycle de vie des agents fait partie des services fournis par une plate-forme d'agents, la première tâche devient également une partie de la deuxième tâche, laquelle est d'uniformiser les services de base des plates-formes multi-agents.

Notre approche pour résoudre la deuxième tâche est « *bottom up* » : nous étudions d'abord des plates-formes multi-agents existantes les plus hétérogènes possible (universitaire et industrielle, propriétaire et open-source, support de normes et non, etc.) afin de synthétiser un ensemble de services de base qu'une plate-forme d'agents doit fournir pour la construction des SMAs réels[1]. Cette étude comparative est présentée dans la section 4.1. En se basant sur les résultats de cette étude, nous développons une interface qui uniformise l'accès à l'ensemble des services de base de différentes plates-formes et permet à l'agent de s'adapter à celles-ci. Ce travail est expliqué dans la section 4.2.

4.1 Étude comparative des plates-formes multi-agents

4.1.1 Sélection des plates-formes d'évaluation

Afin de maximiser la précision du résultat, il nous faut sélectionner attentivement et soigneusement un ensemble des plates-formes multi-agents pouvant représenter suffisamment la plupart des plates-formes d'agents existantes. Dès lors, les critères de sélection suivants sont considérés :

- **Langage de l'implémentation** : En tenant compte de l'hypothèse de notre recherche, laquelle requiert que les plates-formes d'agents soient implémentées dans un même langage de programmation, et du fait que Java est le langage *de facto* dans le domaine, nous ne considérons que les plates-formes construites en Java ;
- **License commerciale** : il est nécessaire d'évaluer des plates-formes en provenance de l'industrie (produit commercialisé) et du publique (recherche universitaire et/ou

[1] Bien qu'il existe certains travaux de synthèse des plates-formes d'agents [Aridor and Oshima, 1998] [Gschwind, 2000] [Flores, 1999] [Ricordel and Demazeau, 2000] [Gasser, 2001] mais leurs objectifs sont différents des nôtres et leur résultats ne sont pas pertinents pour l'uniformisation des services fondamentaux des plates-formes.

open-source), parce qu'elles sont largement différentes au niveau de l'accès au code, de la rigueur du processus de développement, etc. ;

- **Popularité** : la popularité est un indice précis de la maturité d'une plate-forme. En plus, une plate-forme originale a beaucoup d'influences sur d'autres et est généralement la plus référenciée ;
- **Support des normes** : les normes sont introduites pour la standardisation et la stabilisation des caractéristiques, alors les plates-formes qui soutiennent des normes sont également plus éprouvées et stables.
- **Documentation** : les documents sont importants pour la compréhension d'une plate-forme.

En considérant ces critères, nous avons sélectionné les quatre plates-formes suivantes :

- **Aglets** [Lange and Oshima, 1998] : Aglets est probablement la première plate-forme d'agents génériques et mobiles écrite en Java. Elle a été créée à l'initiative du laboratoire IBM Tokyo Research en tant que produit commercial, puis transférée à la communauté en devenant *open source* en 2000 ;
- **Grasshopper** [Grasshopper, 2000] : cette plate-forme est un produit commercial de la compagnie allemande IKV++. C'est la première plate-forme d'agents qui supporte le standard MAF [OMG, 1999b] de l'OMG ;
- **Voyager** [Voyager, 1999] : développée par la compagnie ObjectSpace, cette plate-forme offre un ORB complet de l'architecture distribuée CORBA [Corba,] en tant qu'infrastructure pour les agents ;
- **Jade** [Jade, 2003] : une plate-forme *open source*, conçue par les auteurs des normes FIPA et entièrement conforme à cet ensemble de standards. Grâce à sa stabilité, à sa conformité aux normes et à la contribution d'une grande communauté de développeurs, cette plate-forme est maintenant la plate-forme d'agents la plus populaire au monde.

Le tableau 4.1 résume la conformité de ces quatre plates-formes aux critères de sélection et indique, en plus, la version utilisée pour l'évaluation. Le choix de la plate-forme Voyager n'a aucune liaison par sa popularité, mais par son niveau de robustesse et notamment par sa caractéristique spéciale : les agents Voyager sont également des objets

CORBA, un standard industriel *de facto* pour les applications distribuées.

Plate-formes / Critères	**Aglets**	**Grasshopper**	**Jade**	**Voyager**
Langage	Java	Java	Java	Java et IDL
License commerciale	produit commercial puis *open-source*	produit commercial	*open-source*	produit commercial
Popularité	pionnière	utilisée dans plusieurs projets	la plus populaire	-
Norme supportée	non	MAF et FIPA	FIPA	CORBA
Documentation	bonne	très bonne	excellente	limitée
Version évaluée	1.1	2.3	3.x	3.2

TAB. 4.1 – Quatre plates-formes multi-agents d'évaluation

Dans les sections qui suivent, nous étudions les services fournis par ces quatre plates-formes, l'une après l'autre, avant de déterminer quels sont les services fondamentaux (obligatoires) d'une plate-forme d'agents, ainsi que la liste des services optionnels. Notre évaluation est du point de vue de développeur : les aspects implémentatoires sont mis en premier, nous ne considérons pas les aspects de niveaux plus élevés, tels que la structure interne de l'agent, les modèles d'interaction et d'organisation, mais les laissons aux chapitres 5 et 6.

4.1.2 Typage de l'agent

Un des aspects les plus importants pour le développement d'un SMA est le type utilisé pour caractériser les agents[2]. Ce typage est la vue de l'agent par sa plate-forme. En général, un agent est implémenté sous forme d'objet ou ensemble d'objets, parmi lesquels se trouve un objet principal qui contrôle le comportement de l'agent.

Aglets, Grasshopper et Jade utilisent un type spécial pour définir un objet comme un agent. Ces trois plates-formes ont choisi l'héritage de ce type comme le moyen de définir les agents du système. Tandis que Aglets et Jade fournissent une seule classe dont tous

[2]Il est à noter que la sémantique complète d'un agent se définit au niveau du modèle d'agent.

les agents héritent (`Aglet`[3] et `Agent`, respectivement), Grasshopper fournit différentes classes, en fonction de la mobilité et / ou de la persistance de l'agent (`StationaryAgent`, `MobileAgent`). Puisque Java ne supporte pas l'héritage multiple, cette approche limite le développeur de SMAs. Heureusement, certains patrons de conception, tels que l'agrégation, permettent de surmonter ce problème.

Une autre possibilité est de permettre aux développeurs de transformer n'importe quel objet sérialisable en un agent, ce qui favorise la mobilité de l'agent. Cette approche est utilisée par Voyager, ce qui la rend particulièrement flexible car elle permet de convertir un objet existant en agent avec peu d'efforts. Cependant, elle est également dangereuse parce que la conversion pourrait entraîner des erreurs imprévues et empêcher la détection *à priori* de celles-ci. De plus, l'utilisation lourde du package `java.lang.reflect` tend à rendre le code moins lisible.

4.1.3 Cycle de vie

Grâce à la maîtrise de son propre temps d'utilisation de CPU, l'agent est capable d'agir de manière autonome afin d'achever ses objectifs[4]. Ceci constitue la différence principale entre agent et objet/composant [Guessoum and Briot, 1999]. Pour cela, tandis que Aglets, Grasshopper et Jade fournissent à chaque agent un *thread* séparé, un serveur Voyager appelle une méthode *ad hoc* de l'agent afin de lui fournir son temps de calcul.

Chaque agent doit avoir son propre cycle de vie, dont toutes les transitions entre états sont initialisées en principe par lui-même (notion d'autonomie) bien que dans certains cas particuliers, ces transitions soient invoquées par une entité extérieure. Les quatre plates-formes choisies suivent plus ou moins le même modèle de cycle de vie des agents (en utilisant toutefois des termes différents) : **Création - Initialisation - Exécution (Migration) - Terminaison**. La figure 4.1 présente le cycle de vie spécifié par la norme FIPA00002 (*Agent Management Specification*) [FIPA Agent Management, 2001].

Un agent peut être créé par différents moyens. Dans le cas le plus simple, un agent est

[3] Le nom Aglet vient de la combinaison des deux mots : *Agent* et *Applet*.

[4] Évidemment, le fait que l'agent dispose de contrôle sur sa propre exécution n'est que la condition initiale pour être autonome, mais ne la garantit pas.

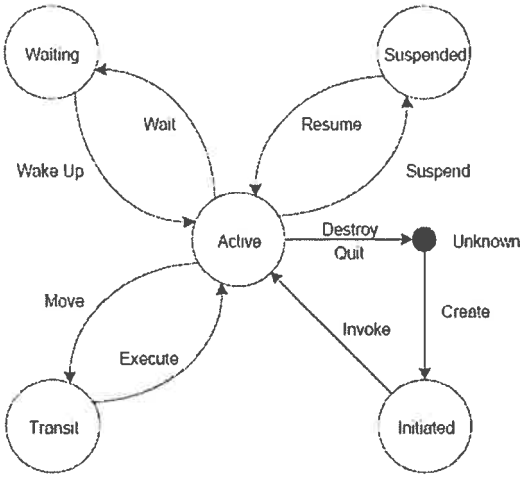

FIG. 4.1 – Cycle de vie de l'agent FIPA

créé en instanciant un objet par l'opérateur `new` de Java. Un autre moyen de créer un agent est l'utilisation du *« factory pattern »*. Alors que Voyager supporte les deux approches, les autres plates-formes n'autorisent que la dernière. L'avantage de la deuxième approche est que l'agent ne doit pas s'inscrire explicitement auprès du serveur et ce dernier peut faire certaines initialisations avant d'instancier l'agent et mettre à jour ses structures de données internes. Cette approche permet aux serveurs de charger dynamiquement le code des agents, notamment dans le cas des agents mobiles, dont le code n'est pas toujours sur le site.

Mis à part le cas de Voyager, pour lequel l'initialisation des agents se fait dans leur méthode constructeur, les trois autres plates-formes permettent aux agents de se configurer par une méthode dédiée à l'initialisation (`onCreation()` pour Aglets, `init()` pour Grasshopper et `setup()` dans le cas de Jade). Cette méthode n'est exécutée qu'une seule fois, immédiatement après la création de l'agent.

Dans le modèle d'exécution des agents des plates-formes Aglets et Grasshopper, la vie d'un agent est implémentée dans une méthode spécifique, appelée `run()` ou `live()`, respectivement. Voyager laisse le soin au développeur d'indiquer cette méthode dans l'in-

terface de définition de l'agent, tandis que Jade supporte un modèle d'exécution plus complexe, permettant de configurer dynamiquement une topologie des comportements (*behaviors*).

Un Aglet termine sa vie en utilisant la méthode `dispose()`. Dans le cas de Grasshopper et Jade, la méthode similaire est `remove()` et `doDelete()`, respectivement. Seule la plate-forme Voyager, qui gère les agents comme les objets, n'offre pas une telle méthode. Un agent Voyager se termine quand il se détache de l'ORB de la plate-forme en déclarant `unbind()`.

4.1.4 Migration

Parmi les attributs des agents, la mobilité est l'une des plus intéressants tout en étant l'une des plus compliqués à réaliser [Kotz and Gray, 1999] [Lange and Oshima, 1999]. Le concepteur d'un SMA peut faire un choix pour la mobilité d'un agent membre : cet agent peut être stationnaire ou mobile [Chia and Kannapan, 1997]. Toutefois, la plate-forme sur laquelle ce système fonctionne doit être capable d'accueillir des agents mobiles pour rendre ce choix possible.

Par définition, un agent mobile est capable d'arrêter son exécution, de migrer vers un autre serveur et enfin de reprendre son exécution (avec ou sans sauvegarde de l'état de son exécution, tel que présenté ci-dessous). Ce service demande la coopération entre l'agent, le serveur de départ et le serveur destinataire de la migration. Il y a donc deux types de migration, *la migration forte* et *la migration faible*[5], suivant le modèle de gestion de la pile d'exécution de l'agent :

- **migration forte** : le modèle dans lequel la pile d'exécution de l'agent est transférée vers la destination de la migration. Plus précisément, une fois arrivé sur son nouveau serveur, l'agent peut poursuivre son exécution exactement au point où elle avait été suspendue avant la migration ; ceci sans qu'il soit nécessaire de programmer spécifiquement l'agent. Il est à noter que ce mode n'est proposé dans aucune plate-forme basée sur le langage Java, du fait des limitations même du langage uti-

[5]Ils sont parfois appelé respectivement *la migration transparente* et *la migration non-transparente* [Funfrocken, 1998].

lisé. Cet obstacle n'existe toutefois pas pour les plates-formes utilisant des scripts d'interprétation comme le langage de programmation des agents.

- **migration faible** : comme la plupart des langages de programmation - Java y compris - ne permettent pas de récupérer la pile d'exécution d'un programme, la migration faible est quasiment la norme dans des applications d'agents. Dès lors, l'agent ne peut que migrer à certains points de son exécution, lesquels doivent être indiqués explicitement par le développeur. De même, arrivé à destination, la restauration du contexte d'exécution de l'agent doit également être gérée par le développeur (à partir de l'appel d'une méthode dédiée à chaque redémarrage d'un agent). Les états de l'agent sont pourtant sérialisés et transmis à la destination puis l'agent y est reconstruit à partir de ces états.

Dans les quatre plates-formes d'agents étudiées, le processus de migration est similaire dans tous les cas, initialisé par l'appel d'une méthode spécifique : `dispatch()` dans le cas des Aglets, `move()` dans le cas de Grasshopper, `doMove()` dans le cas de Jade et `MoveTo()` dans le cas de Voyager.

4.1.5 Adressage d'un agent

À la différence des objets, à chaque agent est assigné un identificateur explicite dès sa création. Cet identificateur est conservé durant toute la vie de l'agent et se doit d'être unique sur le réseau [Agha *et al.*, 2001]. Cette unicité permet aux agents de se déplacer sur les différents serveurs du réseau, sans courir le risque d'une collision d'identificateurs. La plate-forme Aglets offre au développeur d'agents un moyen de générer ces identificateurs uniques, alors que les autres plates-formes laissent au développeur le travail de définition de la méthode de nommage. Puisqu'un agent peut fonctionner sur différents serveurs, seul son identificateur n'est pas suffisant pour le localiser (sans l'aide d'outils *ad hoc*). Pour cela il est également nécessaire de connaître l'adresse du serveur sur lequel l'agent s'exécute, c'est-à-dire l'endroit actuel de l'agent. Par conséquent, l'adresse complète d'un agent peut être obtenue par la formule suivante :

$$adresse_{agent} = adresse_{serveur} + identificateur_{agent}$$

Chaque plate-forme utilise sa propre présentation de l'adresse du serveur, mais souvent sous forme d'une chaîne de caractères :

- Aglets représente l'adresse du serveur sous forme `atp ://hôte :port` ;
- Voyager utilise le service de nom de l'ORB pour trouver un agent, l'adresse du serveur hôte est cachée ;
- Grasshopper utilise la forme `socket ://hôte :port/agence/place` pour représenter l'adresse d'un agent ;
- Jade utilise un IOR (*Interoperable Object Reference*), un string de 32 octets. Elle peut utiliser un chaine de caractères plus lisible comme `corbaloc :URL`, mais cela demande l'utilisation d'un logiciel *ad hoc*.

4.1.6 Communication entre agents

Généralement, il y a deux modes de communication :

- *Communication synchrone* : dans ce mode, un agent envoie son message et attend que l'agent récepteur ait fini de le traiter et lui fasse parvenir sa réponse. L'agent expéditeur est alors bloqué dans son exécution jusqu'à ce que la réponse soit disponible.
- *Communication asynchrone* : dans ce mode, un agent envoie son message et continue son exécution sans devoir attendre que l'agent receveur ait fini de traiter le message.

Afin d'interagir avec d'autres agents, un agent doit être capable de communiquer de manière *asynchrone* et en *point à point* avec ceux-ci. En général, la communication entre agents suppose, de façon générale, plusieurs étapes :

1. *La constitution du message*, ce qui nécessite le formatage des données à transmettre ainsi que l'indication de l'adresse de l'agent destinataire. Cette étape concerne principalement la syntaxe du message ;
2. *La transmission du message entre les deux agents* : la plate-forme multi-agent s'occupe de cette transmission en utilisant des techniques telles que les Sockets et les RPC (*Remote Procedure Call*) ;
3. *La réception et le décodage du message*, qui demande la sémantique de la représen-

tation du message.

Pour la première étape, Aglets, Jade et Grasshopper offrent toutes les trois un objet de contenant (`Message`) pour encapsuler toutes les données à transmettre et l'information de contrôle (expéditeur, destinataire, etc.). Pour la deuxième étape, toutes les plates-formes évaluées fournissent un canal de communication entre les agents, en utilisant soit RMI (Aglets, Grasshopper, Jade), soit ORB (Jade, Voyager). Seule Jade aborde la troisième étape, en supportant la norme de FIPA sur ACL (*Agent Communication Language*).

En plus de la communication asynchrone, Aglets, Grasshopper et Jade fournissent également la communication synchrone. Néanmoins, l'utilisation de ce type de communication est à éviter, parce qu'il ne respecte pas l'autonomie temporelle de l'agent émetteur (lequel est bloqué jusqu'à la réception de la réponse) et de l'agent récepteur (lequel doit répondre immédiatement au message).

4.1.7 Proxy d'agent

Bien qu'il soit toujours possible de rejoindre un agent à une adresse par la voie de communication, trois des quatre plates-formes étudiées, à l'exception de Voyager, offrent un moyen plus facile pour accéder à l'agent, ceci par l'intermédiaire de *proxy*. Un *proxy*, généralement implémenté sous forme d'objet, est un représentant local d'un agent et permet d'accéder à celui-ci, peu importe le lieu où il se trouve. Une demande faite sur un *proxy* sera transférée de manière transparente à son agent propriétaire et c'est à ce dernier de déterminer les actions pertinentes (figure 4.2). L'utilisation de *proxy* vise à empêcher la manipulation directe de l'agent comme dans le cas des objets, ce qui conduirait à la violation de l'autonomie de cet agent. Un agent peut avoir plusieurs *proxies* sur différents serveurs[6].

Nous avons donc la définition suivante :

Définition 6 (Proxy d'agent) *Un proxy est un objet local qui représente l'agent à distance : les actions faites sur un proxy se répercutent sur l'agent, qu'il soit en local ou sur un hôte distant, de manière transparente à l'acteur.*

[6]Sur ce point, le *proxy* ressemble partiellement au pointeur dans la programmation classique.

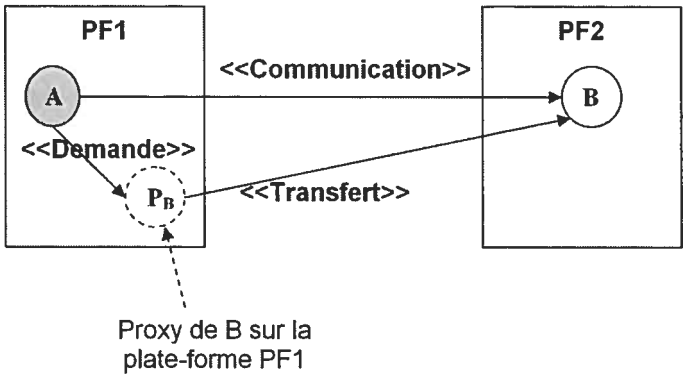

FIG. 4.2 – Proxy - un représentant local de l'agent

En fait, certaines plates-formes (Aglets, Grasshopper) permettent une manipulation directe d'un agent à partir de son *proxy*, mettant à mal cette notion d'autonomie. Néanmoins, ceci illustre bien la différence entre le concept d'agent, et la réalisation qui peut en être faite.

4.1.8 Service des pages blanches

Le service des pages blanches permet à un agent d'obtenir l'adresse d'autres agents à partir de son seul identificateur. L'implémentation de ce service est basée sur le principe suivant : les agents déclarent au service les informations nécessaires quand ils entrent en fonction sans oublier de se désinscrire avant leur terminaison ; le service stocke ces informations dans un dictionnaire pour qu'elles puissent être interrogées ultérieurement par les autres agents.

Alors que Aglets n'offre ce service que de manière indirecte et marginale (avec l'aide de la méthode `getAgletProxies`), Voyager utilise son `Naming Service` pour offrir la capacité de chercher un agent sur le réseau seulement avec son nom. Grasshopper utilise un type de serveur spécial, nommé `Region`, pour gérer tous les agents de tous les serveurs qui s'y sont inscrit au travers de l'agent `MAFFinder`, ceci fait partie du standard MAF. Jade, en respectant la norme FIPA, fournit cet outil sous forme d'un type d'agent spécial, appelé `Directory Facilitator`. Comme dans le cas de Grasshopper, ce service peut observer

des agents sur plusieurs serveurs, à condition de s'y être abonné au préalable.

4.1.9 Service des pages jaunes

Le service des pages jaunes fournit la liste des agents offrant certains services particuliers. Bien que ce service et celui des pages blanches soient utilisés pour des buts différents, leurs principes d'implémentation sont les mêmes, en se basant sur l'inscription et la désinscription des fournisseurs de service auprès d'un annuaire. Néanmoins, seul Jade fournit ce service, en se servant de `Directory Facilitator`.

4.1.10 Service de *broadcast/multicast*

Ce service permet à un agent d'envoyer un même message à plusieurs destinataires. Au point de vue de l'implémentation, pour envoyer un message en *broadcast/multicast*, il est nécessaire de connaître l'adresse de tous les destinataires. Ceci est souvent réalisé en se basant sur la base d'adresses du service des pages blanches. Voyager et Jade supportent cette méthode de communication avec l'aide de leur service des pages blanches.

4.1.11 Outils de gestion

Pour conclure, la disponibilité d'outils facilitant le développement, le déploiement et l'observation des SMAs est une caractéristique important des plates-formes d'agents.

Voyager mis à part, les trois autres plates-formes fournissent un outil graphique permettant d'observer et de contrôler des agents sur le réseau, à partir d'un point central. Ces outils facilitent également le déploiement des agents, en permettant à l'administrateur du système de lancer ou de « tuer » des agents sur différents serveurs, d'obtenir la liste des agents sur le réseau ainsi que leurs états, et enfin de leur envoyer des messages. Néanmoins, chaque plate-forme implémente ses outils selon sa propre approche. Dans le cas d'Aglets, chaque serveur vient avec une interface graphique, appelée `Tahiti`, qui ne permet de manipuler que des agents sur ce serveur. Grasshopper fournit `RegionExplorer`, une interface graphique pour visualiser la base de données d'une `Region`. Jade fournit un ensemble des outils graphiques, permettant de visualiser les agents sur une plate-forme et de lancer certains agents spéciaux, tel que `Remote Agent Management GUI`.

4.1.12 Synthèse et conclusion

Le tableau 4.2 fournit une grille de la disponibilité des services sur les quatre plates-formes d'évaluation.

Services	Aglets	Grasshopper	Jade	Voyager
Typage	v	v	v	-
Cycle de vie	v	v	v	v
Migration	v	v	v	v
Adressage	v	v	v	-
Communication	v	v	v	v
Proxy	v	v	v	-
Pages blanches	v	v	v	v
Pages jaunes	-	-	v	
Broadcast/multicast	-		v	v
Outils de gestion	v	v	v	-

TAB. 4.2 – Grille des services fournis par quatre plates-formes d'évaluation

Nous remarquons que les ensembles de services fournis par les plates-formes d'évaluation sont relativement semblables, à l'exception de la plate-forme Voyager, qui se base majoritairement sur les services de CORBA. Nous pouvons alors regrouper les services en deux types : les services obligatoires et optionnels. Le premier groupe est constitué des services suivants :

- Typage
- Gestion du cycle de vie
- Migration
- Communication
- Proxy
- Adressage et localisation (Pages blanches)
- Outils de gestion

Cet ensemble de services est considéré obligatoire pour une plate-forme d'agents. Les services restants, c'est-à-dire Pages jaunes, *Broadcast/multicast* font partie du deuxième groupe, ils sont optionnels et peuvent être ajoutés ultérieurement.

Après avoir identifié l'ensemble des services qu'une plate-forme d'agent doit fournir,

nous continuons sur la tâche de les uniformiser dans un cadre de travail concret. La section 4.2 présentera ce travail en détails.

4.2 Uniformisation des services de base des plates-formes multi-agents

En notant que l'adaptation dynamique des agents aux plates-formes est notre objectif au niveau plate-forme, notre proposition doit satisfaire aux éxigences suivantes :

1. compatible en arrière aux plates-formes existantes (« *backward compatible* ») ;
2. ouvert pour accepter les nouvelles plates-formes ;
3. *scalable*.

La solution qui semble la plus simple pour l'adaptation aux plates-formes consisterait à définir un standard et à demander à toutes les plates-formes de le suivre [Poslad, 2001]. C'est une solution qui semble logique, tout du moins fort utilisée dans d'autres résolutions d'interopérabilité entre systèmes. Cependant, cette solution ne nous semble pas pertinente, tout du moins dans un avenir proche. Ne comptant pas les difficultés techniques et sociales, cette solution ne satisfait pas à la première contrainte : elle n'a pas de place pour les plates-formes existantes.

La deuxième direction que nous considérons est la conversion des agents. À notre connaissance, le travail de Tjung [Tjung *et al.*, 1999] est l'une des rares solutions opérationnelles suivant cette direction. Ce travail consiste à effectuer un processus de rétro-ingénierie du code de l'agent se trouvant sur la plate-forme source afin de pouvoir le transformer en un code exécutable par la plate-forme de destination. Cette approche n'est pas non plus utilisable pour nous, parce qu'elle n'est pas *scalable* : si l'on considère n plates-formes, il faut être capable d'implémenter $n(n-1)$ mécanismes de transformation. Cela demande un énorme travail, notamment quand les modèles d'exécution peuvent être fort différents et la transformation n'est pas une tâche évidente.

Une autre solution est l'utilisation d'une interface. Le principe de cette approche est de fournir une interface permettant d'uniformiser l'accès aux services fournis par les plates-formes. Cette solution semble satisfaire à nos contraintes, mais elle demande un effort de déploiement : cette interface doit être installée sur tous les serveurs.

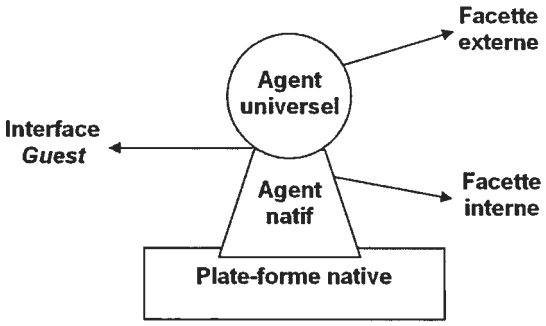

FIG. 4.3 – Interface *Guest*

Pinsdorf [Pinsdorf and Roth, 2002] a suggéré de fournir un serveur qui peut accueillir tout type d'agents, mais cette approche n'est pas vraiment pratique : elle pose trop de contraintes et est très difficile pour l'implémentation. De plus, elle ignore les plates-formes existantes.

Les sections suivantes présentent notre proposition relative au problème d'adaptation aux plates-formes.

4.2.1 Agent universel et interface pour l'adaptation aux plates-formes

Le principe de notre approche est de *définir une interface commune des services fondamentaux des plates-formes multi-agents* et *fournir pour chaque plate-forme une implémentation de cette interface sous forme d'un agent de la plate-forme elle-même.*

La plate-forme hôte est appelée *plate-forme native*, les agents de cette plate-forme sont appelés *agents natifs* et l'interface commune est nommée interface *Guest* (figure 4.3). La partie se trouvant au dessus de l'interface *Guest* est appelée *agent universel.* L'agent natif implémentant l'interface *Guest* est nommé *agent interface.*

Étant composé d'un agent interface et d'un agent universel, chaque agent *Guest*[7] est, en effet, un agent à deux facettes : *interne* et *externe.* La facette interne est associée à

[7]Le nom *Guest* vient du fait que cet agent ne fait partie d'aucune plate-forme native mais peut aller et venir comme un invité sur ces plates-formes. Pour simplifier, nous utilisons également le nom *Guest* pour indiquer les agents universels.

l'agent interface et grâce à cela, l'agent *Guest* est accepté par la plate-forme native comme un agent natif et peut profiter des services de la plate-forme native. La facette externe offre aux développeurs un ensemble uniformisé des services nécessaires à la réalisation des SMAs. Il est évident que chaque agent Guest n'a qu'une facette externe unique mais peut avoir différentes facettes internes sur différentes plates-formes.

Une question est alors posée : quelle est la nature des agents universels ? En effet, celle-ci peut prendre deux formes distinctes : soit liée d'une façon ou d'une autre au fonctionnement des plates-formes sous-jacentes, soit indépendante de celles-ci. Si cette dernière forme peut sembler la plus intéressante, ce n'est pas celle que nous avons choisi. En effet, elle doit se traduire soit sous la forme d'une interprétation du code de l'agent universel en fonction du serveur utilisé, ce qui peut se concevoir sous la forme d'agents décrits par des scripts, soit par l'intermédiaire d'une représentation de haut niveau des agents. Dès lors, cette solution présente tous les inconvénients que l'on retrouve avec les langages interprétés : nécessité de définir un langage qui ne soit pas trop limitatif, besoin d'implémenter un interpréteur ou un compilateur de ce langage, faible efficacité de ce même interprète ou du code généré. Notre choix s'est donc porté sur des agents universels utilisant le même type de code que celui employé par les agents natifs des plates-formes ciblées. Ceci, bien que rendant restrictif le choix des plates-formes utilisables, n'est pas une grande contrainte, en raison de la dominance absolue de Java dans l'implémentation des plates-formes multi-agents.

Le seul accès possible aux plates-formes multi-agents sous-jacentes passe par les agents interfaces, dont la classe hérite de la classe de base fournie par ces plates-formes. C'est pourquoi l'interface *Guest*[8] est implémentée sous la forme d'un agent interface, lequel est lié à un objet Java représentant notre agent universel, qui sera identique sur toutes les plates-formes. Dès lors, pour chaque type de plate-forme cible ont été implémentés des agents interfaces *ad hoc*, lesquels observent le même *modus operandi* avec leurs agents universels quelle que soit leur plate-forme. Par exemple, tout agent universel se voit allouer du temps de calcul par l'appel de sa méthode `live()` tout en étant capable d'envoyer des

[8]Bien que l'approche proposée ne soit pas limitée à la conception des seuls agents *Guest*, nous nous appuyons ici sur cet exemple particulier d'implémentation pour illustrer plus facilement notre propos.

messages de façon uniforme. Ainsi, du point de vue du serveur multi-agent, le couple objet interface / objet universel est vu comme étant un agent purement natif, ce qui lui permet d'être accepté par le serveur pour pouvoir y être exécuté ; alors qu'à l'inverse le concepteur d'un agent *Guest* n'aura à tenir compte que des fonctionnalités proposées par la classe de base fournie par l'agent universel, sans se préoccuper des agents interfaces qui le soutiennent. Autrement dit, l'agent universel ne contient que le code fonctionnel de l'application, toute la tâche d'adaptation aux plates-formes est alors occupée par notre interface *Guest*.

Cette approche permettant de masquer les différences entre plates-formes hétérogènes, cela suppose-t-il que l'on doive se contenter en terme de fonctionnalités du plus petit dénominateur commun entre ces plates-formes cibles ? Il n'en est rien. En effet, trois stratégies pour palier aux différences entre ces plates-formes sont possibles :

1. si les fonctionnalités fournies par les plates-formes sont semblables, alors l'interface *Guest* se contente de faire appel à celles-ci, moyennant l'adaptation du nom des méthodes et des paramètres nécessaires. Dans la figure 4.4, nous montrons que le code de *Guest* (le carré noir de la première colonne) est un simple appel de fonction.

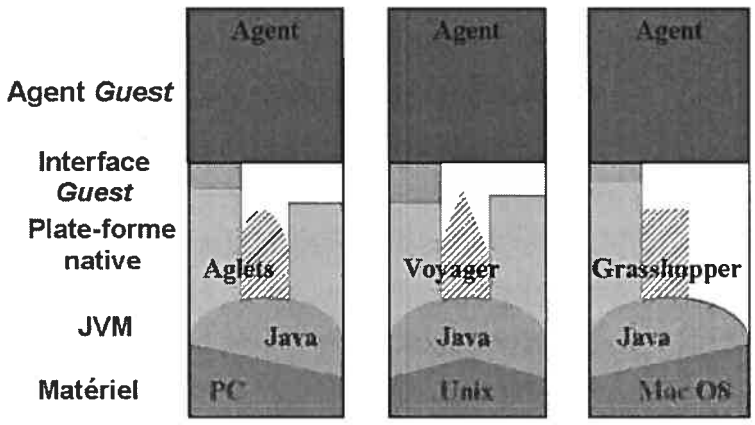

FIG. 4.4 – Fonctionnalités équivalentes fournies par les plates-formes natives

2. si le service recherché n'existe pas ou est présent sous des formes bien différentes au

sein des plates-formes cibles, alors l'interface est chargée d'implémenter *ex nihilo* ce service (figure 4.5).

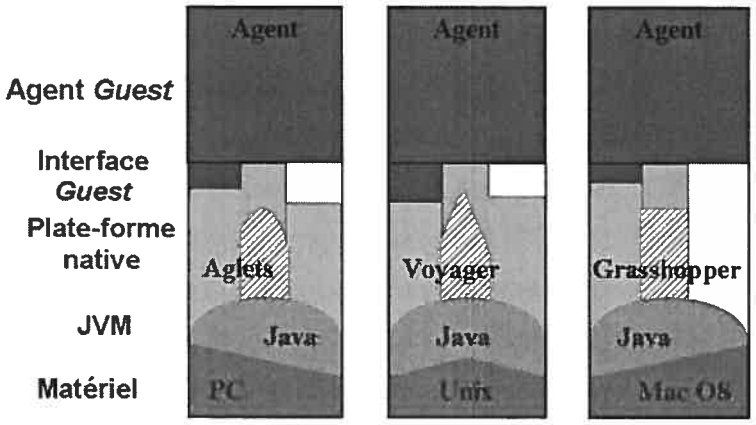

FIG. 4.5 – Fonctionnalités différentes fournies par les plates-formes natives

3. enfin, une approche hybride réunissant les deux précédentes stratégies est possible. Dans ce cas, une fonctionnalité absente ou trop exotique d'une plate-forme sera totalement implémentée par l'agent interface de cette plate-forme (figure 4.6) ; alors que cette même fonctionnalité pourra être déléguée aux mécanismes de base des autres plates-formes plus « standard ».

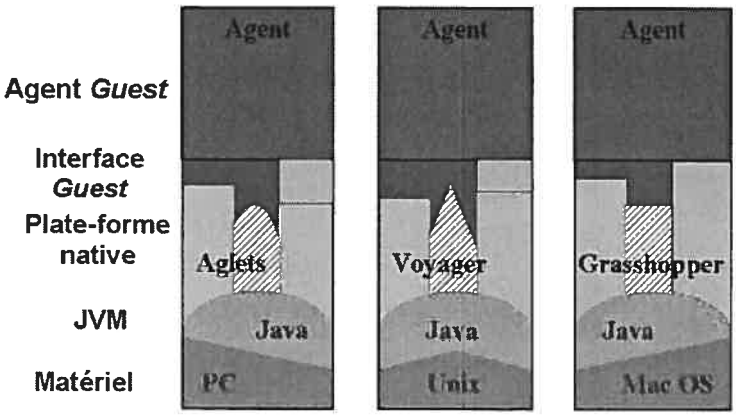

FIG. 4.6 – Fonctionnalités absentes d'une plate-forme native

Afin que notre démarche soit vraiment opérationnelle, l'interface *Guest* doit fournir un ensemble complet des services obligatoires d'une plate-forme (section 4.1.12). Nous détaillons ces services dans les prochaines sections.

4.2.2 Typage de l'agent *Guest*

Dans notre approche, tous les agents de l'utilisateur devront être sous forme d'objets dont la classe doit hériter de la classe de base (que nous appelons `GuestAgent`) afin qu'ils puissent être créés et manipulés de manière indépendante des plates-formes natives. Un tel agent possède un identificateur, un point de démarrage (de leur vie) et offre des fonctionnalités nécessaires au développement des applications multi-agents ; par exemple la communication, la migration.

4.2.3 Cycle de vie de l'agent *Guest*

Dans la section précédente, l'exécution de l'agent est déterminée par la gestion qui en est faite par un serveur d'agents. Dès lors, une question cruciale se pose : est-ce que la migration des agents doit être considérée comme une partie intégrale dans le cycle de vie de l'agent ?

Au niveau de la conception des agents, leur capacité de migration est évidemment optionnelle, parce que dans des systèmes distribués tels que les SMAs, l'endroit de l'exécution des agents est transparent de la logique du programme. Cependant, quand on parle de l'implémentation des agents, la situation change : leur migration doit être gérée par le serveur sur lequel ils s'exécutent. Notre réponse à cette question est donc que la migration doit être prise en compte dans le cycle de vie de l'agent. Dès lors, le cycle de vie de l'agent FIPA présenté dans la figure 4.1 se modifie et devient tel que présenté dans la figure 4.7.

Dans la figure 4.7, un agent *Guest*, au cours de sa vie, peut se trouver dans un des états suivants :

- `STATUS_CREATION`
- `STATUS_EXECUTION`
- `STATUS_SAVE`
- `STATUS_MIGRATION`

– STATUS_REMOVAL

Les deux états STATUS_SAVE_ERROR et STATUS_MIGRATION_ERROR sont des états transitoires et temporaires.

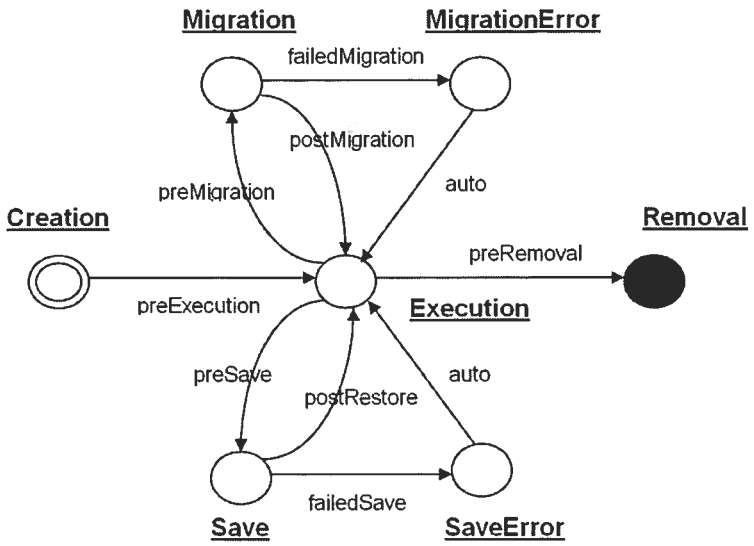

FIG. 4.7 – Cycle de vie complet de l'agent au niveau d'implémentation

Dès lors, la mise en place d'un agent de type *Guest* sur un serveur donné passe par la création sur celui-ci d'un agent interface natif .

La création d'un agent se déroule comme suit : tout d'abord un agent interface est créé sur le serveur indiqué, avec un paramètre d'initialisation étant un objet correspondant à l'agent *Guest*, dont l'état courant est STATUS_CREATION. Ce dernier sera ensuite associé à l'agent interface et son état passe à STATUS_EXECUTION.

Au début de sa phase d'exécution, une méthode spécifique (méthode `live()`) de l'agent sera appelée. Le comportement durant la vie de l'agent doit être décrit dans cette méthode. Ce comportement peut être un calcul, une interaction avec l'être humain ou avec un autre agent, etc.

Au cours de l'exécution de l'agent, il y a probablement des périodes durant lesquelles il ne fait rien. Si ces périodes sont longues, il est préférable d'avoir un mécanisme pour

suspendre (désactiver) cet agent pendant ces intervalles et de le réveiller (réactiver) lorsque nécessaire. *Guest* fournit une telle capacité : un agent *Guest* peut « s'endormir » quelque temps et « se réveiller » ultérieurement. Au cours de son sommeil, l'état de l'agent est `STATUS_SAVE`.

Le principe conceptuel de la désactivation est qu'un agent en pleine exécution interrompe son exécution, se stocke lui-même sur la mémoire externe et ultérieurement reprenne son exécution au point où il l'avait arrêtée. Étant donné que Java ne permet pas d'avoir accès à l'état d'exécution d'un programme (données dans la pile, le compteur ordinal, etc.), l'utilisateur devra donc gérer manuellement la reprise de l'exécution au bon endroit.

La migration dans *Guest* peut être homogène ou hétérogène, et ce, de façon complètement transparente pour l'utilisateur. Le principe conceptuel de la mobilité est qu'un agent en cours d'exécution s'interrompe, ensuite migre vers sa destination et reprenne son exécution au point où il l'avait arrêtée. Pourtant, il est nécessaire de connaître l'emplacement et le type de la plate-forme de destination avant que la migration se déroule. Il faut rappeler que la migration dans *Guest* est de type migration faible, l'utilisateur devra par conséquent gérer la reprise de l'exécution au bon endroit.

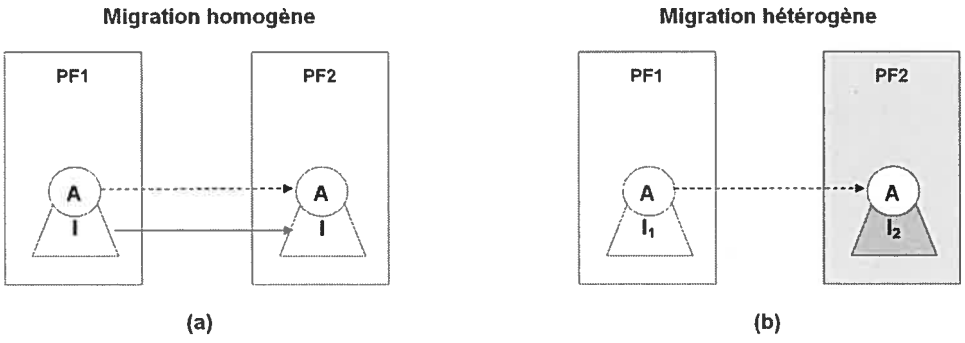

FIG. 4.8 – Migration homogène et hétérogène dans *Guest*

Au cas où la migration est homogène, l'agent *Guest* se déplace ensemble avec l'agent interface (figure 4.8a). Dans l'autre cas, le processus de migration est plus compliqué :

l'agent *Guest* passe à l'état `STATUS_MIGRATION`. Il se sérialise pour être transféré au serveur destinataire. Sur ce serveur, un agent interface est créé et associé à l'agent *Guest*, qui vient d'être envoyé. L'agent *Guest* est ensuite réactivé sur le serveur destinataire : la migration est terminée, l'agent reprend l'état `STATUS_EXECUTION` et continue à fonctionner sur son nouvel hôte (figure 4.8b).

La destruction d'un agent se consiste de mettre fin à l'agent *Guest* et ensuite de détruire l'agent interface qui lui est associé.

4.2.4 Adressage de l'agent *Guest*

Chaque agent *Guest* possède un identificateur, qui ne change jamais tout au long de sa vie. Comme nous l'avons montré dans la section 4.1.5, on a besoin non seulement de son identificateur mais aussi de l'adresse du serveur sur lequel cet agent s'exécute pour localiser un agent. En raison de la mobilité des agents, il est obligatoire que l'identificateur d'un agent soit globalement unique[9]. De plus, comme un agent *Guest* peut se trouver sur les serveurs de différentes plates-formes, l'adresse d'un serveur doit alors inclure le type de sa plate-forme et être convertible en son format réel dans sa plate-forme native.

Dès lors, nous avons choisi le format URI (*Uniform Resource Identifier*) pour représenter l'adresse complète d'un agent comme le suivant :

`PF ://adresse_serveur/identificateur`

dans lequel :

- **PF** : le code de la plate-forme native, qui est généralement attribué par l'auteur de l'agent interface sur cette plate-forme. Il est obligatoire que ce code soit globalement unique. Par exemple, nous avons utilisé `AG` pour Aglets, `GH` pour Grasshopper, etc.[10] ;
- **adresse_serveur** : le format de l'adresse d'un serveur dépend de sa plate-forme, mais il contient souvent le nom de la machine, le numéro de la porte sur laquelle le serveur écoute ;

[9] Si l'identificateur d'agent n'est pas unique, il serait possible que deux agents de même identificateur se retrouvent sur un même serveur, ce qui aurait pour conséquence de ne pas pouvoir les distinguer par le couple (*adresse du serveur, identificateur commun*).

[10] La liste complète des codes utilisés est fournie dans l'annexe II.

- **identificateur** : l'identificateur de l'agent, qui doit être unique sur l'ensemble du réseau.

4.2.5 Communication entre agents

Comme nous avons mentionné dans la section 4.1.12, la communication entre agents se compose de trois étapes : la constitution du message, la transmission du message entre les deux agents et la réception puis le décodage du message. Parmi ces étapes, le point central d'intérêt au niveau plate-forme est la transmission du message. Celle-ci peut prendre deux formes : soit directe, d'agent à agent (approche « *peer to peer* »), soit par l'intermédiaire d'un tiers (approche « *clients / serveur* »). Si ce tiers peut être facilement conçu dans le cas d'une plate-forme propriétaire, cela est bien plus difficile lorsque l'on a affaire à diverses plates-formes dont on n'a accès qu'à l'API. Même si on crée un agent *ad hoc* chargé de jouer ce rôle central, le problème posé reste entier du fait de la nécessité de communication entre les différents agents et cet agent particulier. Notre choix est alors la communication directe (« *peer to peer* »).

Dans la perspective d'agents s'exécutant sur des plates-formes hétérogènes, trois cas peuvent se présenter :

- communication entre agents se trouvant sur le même serveur ;
- communication entre agents se trouvant sur différents serveurs basés sur le même type de plate-forme ;
- communication entre agents se trouvant sur deux serveurs différents de deux plates-formes différentes, lesquelles *a priori* non interopérables ;

Dans les deux premières configurations, la solution repose sur le service de communication fourni par la plate-forme native. Pour la troisième configuration, profitant du fait que l'on est maître de la programmation des agents interfaces et universels (ce qui n'est pas le cas avec les plates-formes), la solution choisie consiste à se baser sur un des mécanismes de communication direct indépendant de ceux fournis par les plates-formes : Sockets, CORBA ou RMI.

Ce choix d'une communication directe, bien que rendant plus facile sa conception, suppose toutefois que l'agent envoyant le message possède en interne une référence de

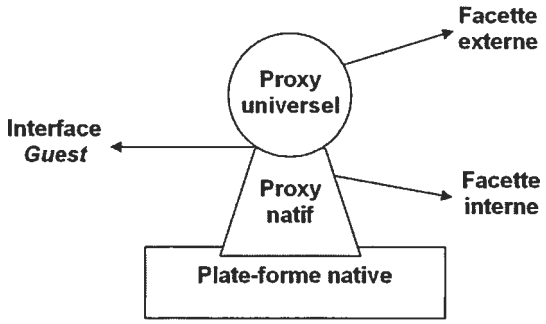

FIG. 4.9 – *Proxy Guest*

l'agent destinataire du message. Celle-ci peut être obtenue soit une fois pour toutes (et mémorisée), soit régénérée à chaque appel. Sans oublier une voie intermédiaire (hybride) également possible : l'agent, à sa création, obtient la référence d'un agent (ou service) spécifique, auprès duquel il peut s'enregistrer avec un nom unique. Par la suite, l'agent peut à tout moment récupérer auprès du service des pages blanches une référence sur un autre agent à partir de son nom ; référence qu'il peut soit conserver dans une mémoire cache, soit régénérer à chaque appel (utile notamment dans le cas où les agents sont susceptibles de migrer). De fait, cette approche ressemble fort aux services de noms offerts par CORBAR ou RMI. Comme nous n'avons de contrôle que sur les *proxies* des agents *Guest*, ce sont les *proxies* que l'on utilise comme références de la communication hétérogène.

4.2.6 Proxy *Guest*

Il est possible d'accéder à un agent à distance par le biais de son *proxy*, qui est un objet local représentant l'agent à distance. Les actions faites sur un *proxy* se répercutent sur son agent de manière transparente à l'acteur. Le rôle du *proxy* pour un agent est similaire à celui du *stub* dans CORBA. Il faut également noter qu'un agent peut très bien être totalement autonome et n'avoir besoin d'aucun intervenant externe pour effectuer sa tâche (laquelle est entièrement décrite dans son *comportement*). Dans ce cas, aucun *proxy*

explicite n'est nécessaire.

Le principe conceptuel d'un *proxy Guest* est similaire à celui d'un agent *Guest*, tel qu'illustré dans la figure 4.9. Un *proxy Guest* dispose donc de deux facettes : la facette externe offre aux utilisateurs des fonctionnalités devant être fournies par le *proxy*, la facette interne est quant à elle associée à un *proxy* spécifique de la plate-forme native.

Nous remarquons que l'interface *Guest* constitue en effet deux parties, une pour les agents *Guest* et l'autre pour les *proxies Guest* (voir la figure 7.2).

4.3 Exemple d'utilisation : le projet IBAUTS

La plate-forme *Guest* a été utilisée dans le projet IBAUTS (*Intelligent Building AUTomation System*). En effet, « Les édifices actuels ne sont pas exploités de façon optimale principalement parce que les systèmes d'automatisation actuels ne peuvent gérer et utiliser intelligemment les nombreuses données qu'ils recueillent. De nombreux édifices disposent de nombreux capteurs et de vastes capacités informatiques réparties, mais ils ne disposent que de capacités rudimentaires pour utiliser ces infrastructures en vue d'exploiter pleinement l'information disponible. Par ailleurs, l'exploitation de nombreux bâtiments souffre de lacunes sur le plan des compétences techniques requises pour que les générations actuelles de systèmes de contrôle automatique de bâtiments soient installés, mis en service, programmés et entretenus correctement. Ce projet a pour objectif le développement d'un système intelligent qui permettra d'obtenir et de maintenir le rendement optimal des immeubles. Ce système sera conçu de manière que les connaissances techniques requises pour l'installation et l'entretien du système soient minimales. » [IBAUTS, 2002].

Dès lors, l'utilisation d'un système multi-agents distribué et adaptatif semble tout indiqué. Dans ce système, la co-existence de différents matériels entraîne l'hétérogénéité des plates-formes multi-agents installées. C'est à ce point que *Guest* devient un choix pertinent, grâce à la capacité de celle-ci de fonctionner sur différentes plates-formes incompatibles[11]. C'est ainsi que l'équipe GLIC du CRIM[12] a développé à partir d'agents *Guest* un modèle de senseurs et d'effecteurs de chauffage et de ventilation étant capable de

[11] Ce choix a été également validé par le modèle d'analyse de traces d'exécution (voir la section 5.2.2.6).
[12] http ://www.crim.ca

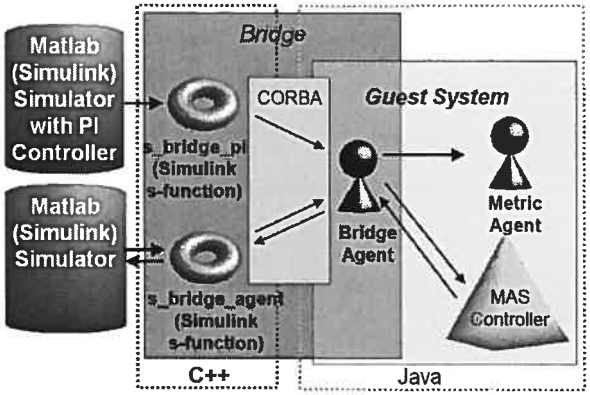

FIG. 4.10 – Passerelle entre l'agent *Guest* et le simulateur

se configurer afin de maintenir une température adaptée. Ce modèle a été testé à travers un simulateur, couplé à *Guest* par une passerelle CORBA (figure 4.10 [IBAUTS, 2002]).

4.4 Conclusion

Dans ce chapitre, nous avons sélectionné quatre plates-formes multi-agents représentatives pour réaliser une étude comparative et complète des services fournis par les plates-formes choisies. Nous avons identifié un ensemble de services qu'une plate-forme doit fournir pour le développement des SMAs réels. L'approche *Guest* a été proposée pour résoudre l'adaptation aux plates-formes, le problème principal de l'adaptation dynamique du contrôle au niveau plate-forme. Cette approche satisfait très bien à toutes les éxigences imposées :

1. compatible en arrière : pour chaque plate-forme existante, il ne faut développer qu'un agent interface ;
2. ouvert : chaque fois une nouvelle plate-forme est introduite, une implémentation de l'interface *Guest* sous forme d'un agent natif de cette plate-forme est suffisante. Par exemple, quand la micro-plateforme CORBAHost (section 7.2.4) est développée, l'intégration de *Guest* à celle-ci est relativement simple : seul un agent natif et un *proxy* sont à écrire afin que l'agent *Guest* fonctionne normalement sur cette nouvelle

plate-forme ;

3. *scalable* : pour rendre compatibles n plates-formes, n agents interfaces sont nécessaires. De plus, l'agent *Guest* se compose toujours d'un agent universel et d'un agent interface, peu importe combien de plates-formes existent.

En plus, le mécanisme d'adaptation aux plates-formes dans *Guest* est dynamique : un agent *Guest* peut visiter une nouvelle plate-forme qui n'est pas encore connue au moment de son démarrage.

Il est à noter que le service des pages blanches et l'interface graphique pour la gestion des agents *Guest* sont indépendants des services présentés ci-dessus. De plus, ils peuvent être implémentés en utilisant ces services. Dès lors, nous ne les présentons pas dans ce chapitre mais nous y reviendrons dans le chapitre 7.

En résumé, le modèle d'agent avec l'interface *Guest* a bien résolu le problème de l'adaptation dynamique du contrôle au niveau plate-forme en rendant compatibles différentes plates-formes multi-agents. L'adaptation du contrôle au niveau agent sera présentée dans le chapitre 5.

CHAPITRE 5

ADAPTATION PAR LA MODIFICATION DYNAMIQUE DU CODE D'AGENTS

Suivant notre analyse dans le chapitre 3, l'adaptation dynamique du contrôle au niveau agent concerne principalement la modifiabilité de sa présentation interne au cours de l'exécution, ainsi que l'extensibilité de ses fonctionnalités. Ces deux caractéristiques conduisent à la capacité de modification dynamique du code de l'agent. Or, un agent peut être décrit soit par le code compilé, sous formes de méthodes ou de composants [Guessoum, 1996] [Horling and Lesser, 1998], soit par le code interprété, sous formes de scripts [Rao and Georgeff, 1995]. Dans le monde des agents, on a des agents interprétés **ou** agents compilés, l'utilisation de ces deux types de code est quasiment exclusive. La raison de cette situation est probablement l'approche pro-conception dans la recherche des agents : au niveau de l'abstraction de la conception, on ne tient pas compte des détails de l'implémentation et il y a toujours une tendance vers l'utilisation soit d'un langage interprété, soit d'un langage compilé. Par conséquent, quand le programmeur développe des SMAs, il tente d'être d'accord avec le concepteur de l'utilisation d'un langage choisi. Dans des SMAs complexes et hétérogènes, il y a toutefois des agents compilés ou interprétés qui fonctionnent et collaborent ensemble. Notre approche vise à fournir un modèle d'agent combinant des codes compilés et interprétés afin de maximiser les avantages des deux dans l'adaptation dynamique de l'agent.

Nous commençons par une comparaison entre le code interprété et le code compilé. Ensuite, la section 5.2 présente deux modèles pour l'adaptation dynamique du code compilé : l'un est le modèle de composants dynamiques et l'autre est le modèle des *plugins*. La section 5.3 explique l'application du formalisme CATN pour spécifier les comportements des agents sous forme du code interprété tout en permettant l'adaptation dynamique de celui-ci. Finalement, la section 5.4 présente la combinaison de ces modèles pour obtenir une architecture interne de l'agent qui est dynamiquement modifiable et extensible.

5.1 Code compilé et code interprété

Selon Schneider [Schneider *et al.*, 2001], un langage interprété dispose des caractéristiques suivantes :

- le but d'un langage interprété est de développer des applications en mettant en œuvre un ensemble de composants existants ;
- un langage interprété favorise la programmation à un niveau plus élevé que la performance : l'introspection de l'exécution est plus importante que la vitesse d'exécution ;
- un langage interprété est extensible et « *scalable* » : il est conçu pour étendre le modèle de langage avec les nouvelles abstractions et pour interopérer avec les composants écrits dans d'autres langages ;
- un langage interprété est incorporable (« *embeddable* ») : il est possible de l'incorporer dans une application existante. De cette façon, il favorise l'adaptation, la configuration et l'extension d'une application ;
- un langage interprété peut offrir un support pour l'introspection.

Grâce à ces caractéristiques, un langage interprété est considéré comme un outil prometteur et potentiel pour la construction des systèmes ouverts et distribués, notamment les systèmes d'agents. De plus, il semble être idéal pour l'adaptation dynamique : il fournit la flexibilité de modification dynamique du code (grâce à son support pour l'introspection), l'indépendance relative aux plates-formes d'exécution (il est incorporable), la modularisation et l'extensibilité (être capable d'interagir avec les composants écrits dans d'autres langages).

Pourtant, un langage interprété ne dispose pas que d'avantages, notamment dans le domaine de la performance : un tel modèle se traduit par l'exécution relativement lente du programme.

Par opposition, le code compilé est largement supérieur quant à la vitesse d'exécution. De plus, la granularité du code est plus fine : lorsque le code interprété doit être réceptif aux appels des méthodes dans son exécution, le code compilé permet de manipuler jusqu'aux instructions de son langage. Cependant, la modifiabilité du code compilé est très modeste : une fois que le code est déjà compilé, il ne peut être modifié que de manière *ad*

hoc et cela, de manière très limitée. Toutefois, des langages de programmation modernes fournissent la capacité de l'introspection et de réflexion, comme dans le cas de Java.

Naturellement, le code compilé et le code interprété apparaissent comme des choix opposés. À notre connaissance, il y a rarement une approche qui essaie d'unifier ces deux directions, afin que leurs avantages soient complémentaires. On peut par exemple citer les travaux de [Valetto and Kaiser, 2002], de [Ashri and Luck, 2001], cependant ces travaux s'arrêtent au niveau de la conception, étant très mal spécifiés au niveau de l'implémentation. C'est sur ce point que nous essayons d'aller plus loin : nous cherchons à unifier ces deux types de code dans un modèle opérationnel afin de profiter des avantages des deux.

5.2 Code compilé modifiable et composable

Nous présentons dans cette section le modèle de composantes dynamiques et le modèle de *plugins* que nous avons conçus. Le premier modèle introduit des composants qui peuvent être substitués au milieu de leur exécution de manière transparente à leurs partenaires. Le deuxième modèle décrit le cadre de travail *plugins* permettant d'étendre dynamiquement les fonctionnalités d'un agent.

5.2.1 Modèle de composants dynamiques

Notre idée d'un modèle de composants dynamiques commence par une remarque : il existe des composants (logiciels) dont les tâches sont quasiment fixées, mais la façon de les réaliser peut évoluer rapidement au fil de temps. Vue de l'extérieur, ces composants ressemblent à des boîtes noires dont les interfaces externes sont bien définies. Par exemple, la technologie a évolué de CRT à plasma ou à LCD, mais seule la qualité d'affichage est améliorée, alors que la tâche d'un écran ne change pas : il affiche ce qu'on lui demande.

Une question est alors posée : est-il possible de changer un tel composant par un autre qui peut mieux exécuter les mêmes tâches, mais ce changement ne conduit-il pas, ou n'affecte-t-il que de façon minimale, son entourage ? Les travaux de Pal [Pal, 1998], Kniesel [Kniesel, 1998], Fung [Fung and Low, 2003] nous ont permis de répondre « oui ».

Néanmoins, les travaux mentionnés proposent des solutions complexes pour résoudre

plusieurs problèmes simultanément, ce qui demande une infrastructure de support relativement lourde. Alors que dans notre cas, nous recherchons une solution légère pour les agents, qui puissent être mobiles avec tout leur code. Dès lors, nous avons conçu un modèle de composants dynamiques dédié aux agents, mettant l'accent sur la simplicité et l'efficacité dans un but unique : remplacement dynamique d'un composant ayant des interfaces externes biens définies.

5.2.1.1 Principe du modèle

Le principe de notre modèle est comme suit :

- Chaque composant est constitué, en effet, d'un composant interne et d'un *proxy* (voir la figure 5.1) ;
- Le composant externe fournit au monde extérieur un ensemble de services, mais c'est le composant interne qui est le réalisateur réel ;
- Chaque invocation d'un service faite sur le composant externe est répercutée sur le composant interne à travers le *proxy*. En fait, toutes les interactions avec les autres composants doivent être passées par le *proxy* ;
- Le remplacement d'un composant interne par un autre n'est connu que par le *proxy* et par le composant externe, mais entièrement transparent de ses partenaires.

Ce *proxy* est différent de celui de l'agent, bien qu'ils jouent des rôles similaires. La définition suivante permet d'éviter toutes les confusions :

Définition 7 (Proxy d'un composant) *Le proxy d'un composant est son représentant pour toutes les interactions (invocations, communications) avec les composants partenaires.*

Seul le proxy peut avoir le contact direct avec le composant dont il est le représentant. Grâce à cette caractéristique, le remplacement d'un composant par un autre est possible, à condition que les deux puissent exécuter les mêmes tâches. Toutefois, notre intention étant d'appliquer aux composants dont les tâches sont fixées et bien spécifiées, cela n'est plus une contrainte.

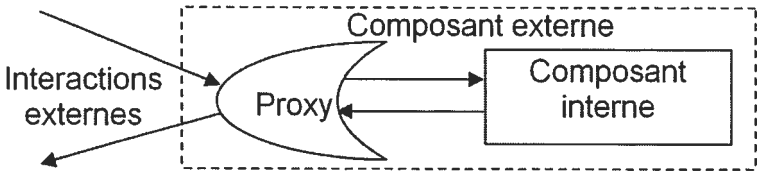

FIG. 5.1 – Composant avec proxy

Définition 8 (Composant équivalent) *Deux composants sont équivalents si et seulement si ils implémentent les mêmes interfaces publiques.*

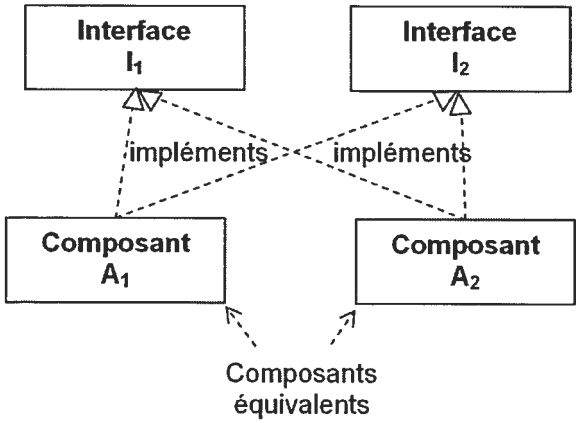

FIG. 5.2 – Composants équivalents

La figure 5.2 montre deux composants équivalents `A1` et `A2`, qui implémentent deux interfaces publiques `I1` et `I2`. Par conséquent, un composant `A1` peut être remplacé par un composant `A2` et inversement. Toutefois, ce remplacement soulève deux problèmes :

1. transfert de données internes du composant courant vers le nouveau ;
2. gestion des interactions durant la période de substitution.

Les deux sections ci-dessous détaillent notre solution pour ces deux problèmes.

5.2.1.2 Conservation des données internes des composants

Gamma dans le célèbre livre « *Design Pattern* » [Gamma *et al.*, 1995] a introduit le patron (*« pattern »*) *Memento*, dont le but est de transférer les données d'un objet à un autre. En appliquant ce patron, nous définissons l'interface suivante pour le transfert de données entre les composants équivalents[1] :

```
public interface IDataTransfer {
    public Memento exportData{};
    public boolean importData(Memento memento);
}
```

5.2.1.3 Conservation des interactions externes avec d'autres composants

Pourtant, le patron *Memento* ne garantit pas la continuation de l'exécution de l'ensemble du système : il demande une période spéciale, durant laquelle un composant transfert ses données vers sa substitution et est dans l'impossibilité d'interagir avec ses partenaires. Or, comme le processus est entièrement caché par les composants restants du système, ces derniers continuent d'essayer de le contacter. Nous fournissons donc un tampon (*« buffer »*) spécial pour enregistrer temporairement ces demandes d'interaction externes, afin qu'elles puissent être prises en compte par le nouveau composant dès qu'il vient en fonction. Certes, cela implique un délai du temps de réponse, cependant ce délai est très mineur, par rapport au temps que prendraient l'arrêt, le remplacement et le redémarrage du système entier.

5.2.1.4 Processus de substitution

Le protocole de substituer dynamiquement un composant par un autre équivalent suit les étapes suivantes (figure 5.3) :

1. le *proxy* coupe la liaison avec le composant courant ;
2. À partir de ce moment, toutes les interactions venant de l'extérieur sont stockées dans un *« buffer »* temporaire ;

[1] Il faut noter que cette interface n'est pas considérée comme une interface publique.

3. le composant en retrait exporte ses données en utilisant l'interface `IDataTransfer` ;
4. le nouveau composant importe ces données ;
5. le nouveau composant récupère les interactions dans l'attente du « *buffer* » ;
6. le *proxy* se connecte désormais avec le nouveau composant. Le processus est maintenant accompli.

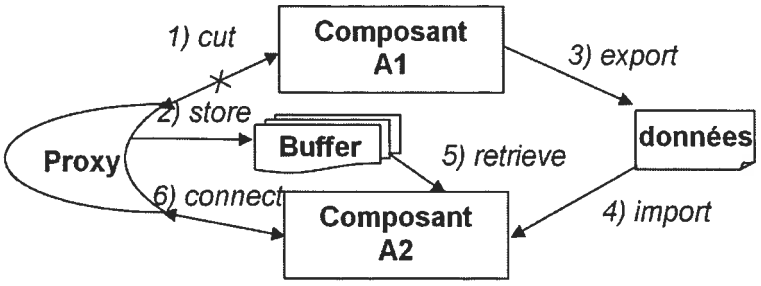

FIG. 5.3 – Processus de substitution des composants équivalents

5.2.1.5 Utilisation

Bien que le modèle de composantes dynamiques ait encore des limites, il est simple, performant et léger. Dès lors, ce modèle pourra être utilisé entre agents (pour en faciliter la migration, par exemple) aussi qu'au sein de ceux-ci (pour la substitution des *plugins*, des interpréteurs).

5.2.2 Modèle de *plugins*

Pour des agents qui fonctionnent longtemps, il est nécessaire d'acquérir de nouvelles compétences et de collectionner de nouvelles données et connaissances afin de conserver leur utilité et d'achever leurs objectifs. En bref, les agents doivent évoluer. Nous avons conçu le modèle *plugin* dans ce but, afin d'apporter l'extensibilité aux agents et de rendre flexible le développement et le déploiement de ceux-ci. Nous présentons dans les sections qui suivent le principe du modèle, les opérations possibles des *plugins*, le déploiement et le retrait des *plugins*, ainsi qu'une évaluation de ses avantages et limites.

5.2.2.1 Principe du modèle

Définition 9 (Plugin d'agent) *Un plugin est un composant logiciel offrant un ensemble de fonctionnalités qui peut être dynamiquement ajouté ou retiré d'un agent, selon les besoins de ce dernier.*

Le cryptage et le décryptage des messages de communication, ou un nouvel algorithme d'apprentissage plus performant sont des exemples de services qu'un *plugin* peut offrir aux agents. Étant capable d'ajouter des *plugins* au cours de son exécution, l'agent devient extensible. Un tel agent se compose donc de deux parties (figure 5.4) :

- un noyau avec la description de son comportement, ses propres données, connaissances et logiques d'affaire (code fonctionnel), ainsi qu'un ensemble minimal de code du contrôle, y compris le code pour la gestion des *plugins* (chargement, déchargement, etc.) ;
- des *plugins*, qui peuvent être chargés dynamiquement et séparément à l'exécution de l'agent. Leurs services deviennent par la suite des fonctionnalités additionnelles de l'agent. Tandis que la première partie est fixée dès le démarrage de l'agent, cette partie est entièrement dynamique et peut évoluer tout au long de la vie de l'agent.

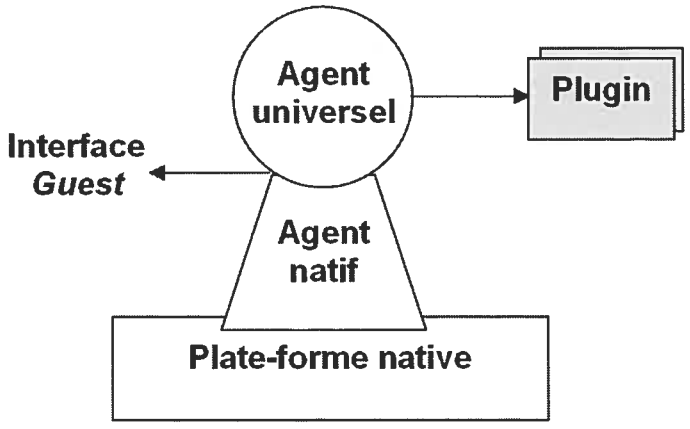

FIG. 5.4 – *Plugins* par rapport à la couche interface *Guest*

5.2.2.2 Opérations des *plugins*

Un *plugin* peut s'intégrer étroitement à des agents, ce qui lui permet non seulement d'*offrir de nouvelles fonctionnalités* à ceux-ci mais aussi de les *observer* et d' *intervenir* dessus. En réalité, un *plugin* peut effectuer les trois types d'opérations suivants :

- être informé avant le changement d'état de l'agent auquel il est attaché, voire empêcher ce changement dans certains cas (par exemple la migration de l'agent) ;
- observer ou même intercepter des messages de communication de cet agent, que ce soit des messages envoyés ou reçus ;
- fournir de nouvelles fonctionnalités.

Les *plugins* peuvent être développés indépendamment des agents et intégrés ultérieurement. Une fois instancié et attaché à un agent, un *plugin* peut *intercepter le flux des messages* de l'agent, ce qui permet d'*effectuer des transformations sur ce flux* avant que l'agent ne le perçoive. Les *plugins* qui veulent manipuler les messages et les états des agents fournissent des intercepteurs et les inscrivent auprès du service de gestion des *plugins* de l'agent. S'il y a plusieurs *plugins*, ils sont enchainés selon l'ordre de l'inscription et on obtient un fil d'intercepteurs. L'agent peut donc changer ces *plugins* afin de changer son comportement par rapport à l'environnement, tout en continuant son exécution.

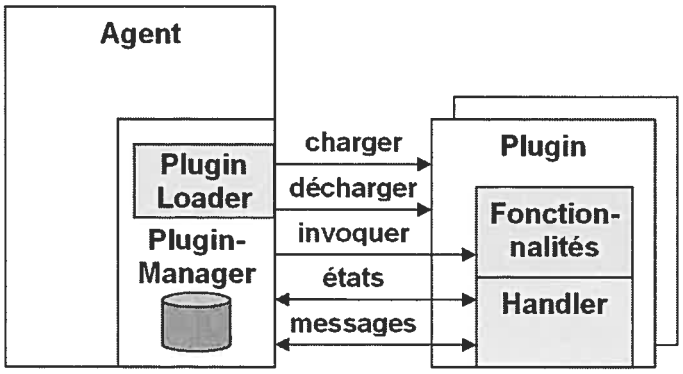

FIG. 5.5 – Architecture des agents avec *plugins*

Dans cette architecture, rien n'empêche un *plugin* d'utiliser les services fournis par les autres. En effet, il est possible qu'un *plugin* se base sur d'autres dans la réalisation de ses propres services. Cependant, les *plugins* n'ont pas la capacité de communication. Ils peuvent toutefois communiquer en demandant à l'agent auquel ils s'attachent de le faire pour eux.

Un *plugin* est, en effet, un couple de deux composants (figure 5.5) :

- un *handler* qui s'occupe des deux premiers types d'opérations : intercepter les messages et / ou observer le changement d'état d'un agent ;
- une librairie offrant de nouveaux services.

Cette séparation renforce la sécurité du modèle et simplifie le développement et l'utilisation des *plugins* :

- Au point de vue du développeur, il peut développer selon ses besoins. Par exemple, s'il ne veut qu'intercepter des messages, il n'aura à s'intéresser qu'à la partie *handler*. Il ne doit pas se soucier de cacher les méthodes publiques de « *callback* » des *handlers* ; il a le choix de publier seulement les méthodes de fonctionnalités qu'il veut offrir. Le chargement de ces deux composants est fait par le service de gestion des *plugins* ;
- Au point de vue de l'utilisateur (agent, les autres plugins ou l'être humain), il ne voit que la partie publique d'un *plugin*.

5.2.2.3 Configuration et déploiement des *plugins*

Un *plugin* peut être développé séparément avec le développement des agents pour les intégrer ultérieurement. Dans certains *« framework »* tels que Eclipse [Eclipse, 2003], les *plugins* doivent être configurés et déployés au moment du lancement de la plate-forme, le changement d'un *plugin* au cours de l'exécution est alors impossible. En revanche, notre modèle *plugin* vise à permettre aux *plugins* d'être ajoutés ou retirés à n'importe quel moment dans la vie de l'agent : au lancement, au cours de l'exécution ou même à la destruction de l'agent. Dès lors, certains problèmes concernant le déploiement d'un *plugin* se posent : la priorité entre *plugins*, comment résoudre le conflit potentiel entre différents *plugins*, etc. Notre approche est de fournir une configuration pour chaque *plugin* et de laisser

la tâche de vérification de celle-ci à un composant spécifique, appelé `PluginLoader`[2].

En fait, la configuration d'un *plugin* consiste en deux parties :

- configuration de description : cette partie est fournie par le développeur du *plugin* et contient la description de celui-ci : la version, les classes implémentant le *plugin*, les *plugins* pré-requis, les *handlers*, la liste des services (méthodes) fournis, les paramètres nécessaires de chaque méthode ainsi que leur type, la possibilité de multiples instances, etc. Tous les *plugins* d'un même code partagent cette configuration.
- configuration de déploiement : cette partie contient toutes les informations contextuelles : le nom donné au *plugin* à déployer, l'endroit où se trouve le code de ce *plugin*, la priorité de chaque *handler*, les paramètres d'initialisation, etc. Cette partie est spécifique à chaque *plugin* au moment de son chargement.

L'invocation des services d'un *plugin* peut être réalisée de manière systèmatique, en se basant sur les descriptions se trouvant dans la configuration de ce *plugin* : le nom du service, les paramètres et leur types, le type du résultat obtenu. Ce principe ressemble à celui du WSDL [WSDL, 2002].

5.2.2.4 Retrait d'un *plugin*

Le retrait dynamique d'un *plugin* doit être traité attentivement, parce qu'il est probablement en utilisation par l'agent ou d'autres *plugins*. Le processus pour retirer un *plugin* est le suivant :

- signaler au service de gestion des *plugins* (`PluginManager`) l'arrêt de ce *plugin* ;
- le service de gestion des *plugins* interdit désormais tous les accès à ce *plugin* et détache les *handlers* de celui de l'agent ;
- comme le *plugin* n'est qu'un objet ordinaire et qu'aucun autre objet n'a une référence vers celui-ci, il peut faire ses propres ménages et se terminer tranquillement.

[2]Cette vérification statique peut être résolue avec l'aide d'une base de connaissances et des règles pertinentes.

5.2.2.5 Substitution d'un *plugin*

L'opération de substitution d'un *plugin* par un autre peut être considérée comme la combinaison de deux opérations successives : retirer le *plugin* actuel puis ajouter le nouveau[3].

5.2.2.6 Exemples de l'utilisation du modèle de *plugins*

Le modèle de *plugins* ont été utilisé intensivement à enrichir les fonctionnalités de la plate-forme *Guest* (chapitre 7) :

- RegionPlugin (section 7.3.1.5) : ce *plugin* permet à un agent d'utiliser un service des Pages blanches ;
- InteractionPlugin (section 7.4.1.2) : ce *plugin* fournit un service de communication de broadcast/multicast à l'agent auquel il est attaché ;
- RecursionPlugin (section 7.4.1.2) : ce *plugin* ajoute aux agents la capacité d'organisation hiérarchique distribuée.

Le projet IBAULTS (section 4.3) s'est servi des *plugins* afin d'implémenter un système de collection des traces d'exécution des agents. L'architecture de ce système est comme suit :

- un agent Observateur, capable de traiter les traces reçues ;
- un *plugin* d'envoi et un autre de réception, s'occupe de prendre une copie du message envoyé et reçu, respectivement, par l'agent et de l'acheminer à l'agent Observateur. Après la notification de l'agent observateur de lŠoccurrence de ces événements, ce dernier se charge de faire la mise en correspondance entre les événements d'envoi et de réception et de les enregistrer dans un fichier XML.

5.2.2.7 Conclusion

Le modèle de *plugins* permet aux agents de charger plusieurs *plugins* et plusieurs instances d'un même *plugin*. Par exemple, un agent peut avoir en même temps deux

[3]Néanmoins, notre modèle de composants dynamiques fournit un autre choix dans l'implémentation de cette opération.

instances d'un *plugin* fournissant l'interface ODBC (*Open DataBase Connector*), une pour une base de données Oracle et l'autre pour une base de données MySQL. Un *plugin* peut être attaché ou retiré d'un agent selon le besoin de ce dernier, lors de l'exécution de celui-ci. Il est donc possible d'ajouter de nouveaux services ou de retirer des services obsolètes des agents de manière dynamique, sans recompiler ou redémarrer ces agents. De plus, les *plugins* rendent la mise à jour des agents bien plus facile, en permettant par exemple de remplacer un *plugin* chargé par une version plus courante.

Étant capable d'intercepter les messages de communication et d'observer le changement d'état des agents, les *plugins* peuvent non seulement enrichir la capacité des agents, mais aussi changer leur comportement. Par exemple, dans l'architecture hiérarchique présentée dans la section 6.1, un *plugin* peut intercepter la migration des agents fils dans certains cas. Le comportement de ces derniers lorsqu'ils ont intégré la hiérarchie peut être alors différent de celui qu'ils avaient au préalable.

Un autre avantage des *plugins* est qu'ils permettent de personnaliser dynamiquement la configuration des agents selon leur besoin et d'optimiser l'utilisation de la mémoire. Par exemple, on peut supposer un SMA tournant sur une seule machine pour un certain temps. Ce même SMA peut être réparti sur plusieurs ordinateurs, en changeant simplement sa configuration des *plugins*. En effet, le même agent est configuré différemment. Dans le cas d'une machine unique, on configure le déploiement par l'ajout du protocole de transport à mémoire partagée, alors que c'est le protocole de transport RMI qui est utilisé dans le système multi-hôtes.

Un autre avantage du modèle *plugin* est qu'il encourage la réutilisation des efforts de développement, car un *plugin* peut être utilisé pour différents agents.

Même un service important des plates-formes multi-agents, le service des Pages blanches, peut être construit et rendu disponible à l'agent par un *plugin* particulier (section 7.3.1.5). C'est la dynamicité et la puissance de notre modèle *plugin* qui sont les différences avantageuses avec plusieurs autres modèles du même genre.

5.3 Spécification des agents en formalisme CATN

Nous présentons dans cette section le formalisme CATN, lequel permet de spécifier le comportement d'un agent sous forme d'un langage interprété. Nous montrons ensuite qu'avec cette représentation, il est possible de modifier le comportement de l'agent au cours de l'exécution de celui-ci.

5.3.1 Choix de langage d'interprétation

Les modèles d'agents utilisant un langage d'interprétation sont nombreux, parmi eux nous pouvons citer comme étant représentatif [Tardo and Valente, 1996] [Odell *et al.*, 2000] [Stolzenburg and Arai, 2003]. Cette popularité a montré qu'un langage interprété est un outil approprié pour la spécification et le développement des agents. Cependant, ces travaux, soit soulignent la spécification des processus internes des agents (MONAD dans [Vu *et al.*, 2003]), soit se focalisent sur les processus de migration des agents (Telescript), soit soulignent les interactions entre eux, comme AUML dans [Odell *et al.*, 2000]. Il y a également certains langages interprétés qui sont introduits spécifiquement pour un modèle d'agents concret ([Fischer *et al.*, 1995] pour le modèle BDI).

Pour notre part, au lieu d'inventer un nouveau langage, nous avons cherché un langage existant permettant de spécifier l'agent, que ce soit pour les processus internes de l'agent ainsi que ses interactions avec le monde extérieur. Nous nous sommes spécialement intéressés à des machines à états finis (*« Finite State Machine (FSM) »*) [Wood, 1987], lesquelles sont largement utilisées pour la spécification des aspects dynamiques et procéduraux d'un logiciel [Stolzenburg and Arai, 2003]. Selon Marini [Marini *et al.*, 2000], un formalisme de type machine à états finis apporte les avantages suivants :

- spécification facile ;
- interprétable (prototypage aisé) ;
- vérification facile (nécessaire pour les applications critiques) ;

En tenant compte de tous ces facteurs, nous avons choisi le formalisme CATN (*Coupled Augmented Transition Network*) [Zavala *et al.*, 2001] pour spécifier les processus internes et les interactions externes de nos agents et pour ensuite modéliser l'adaptation dynamique

du contrôle des agents interprétés. Né d'un formalisme très connu des années 1970 (ATN - *Augmented Transition Network*)[Woods, 1970], le formalisme CATN hérite des avantages d'ATN tout en ajoutant l'aspect des interactions. Dès lors, ce formalisme permet de spécifier non seulement les processus internes des agents mais aussi les interactions entre eux, ce qui est cruciale dans les systèmes multi-agents. De plus, CATN peut être intégré aux plates-formes existantes, tel que dans le cas de la plate-forme FIPA-OS.

5.3.2 Description du formalisme CATN

Coupled[4] ATN (ou CATN) [Zavala *et al.*, 2001] [Lemaître *et al.*, 2003] est un formalisme conçu spécifiquement pour décrire des comportements d'agents ainsi que des interactions entre eux à travers l'échange d'actes de langage.

Un SMA réalise plusieurs tâches, dont chacune demande la collaboration de certains rôles, qui sont réalisés par différents agents [DeLoach, 1999] [Wooldridge *et al.*, 2000]. En utilisant le formalisme CATN, il est possible de spécifier non seulement les rôles que doit exécuter un agent, mais aussi la coopération entre les agents d'un SMA pour réaliser une de ses tâches : le comportement d'un rôle est décrit par un CATN et les interactions avec d'autres rôles sont spécifiées par les transitions d'interaction

Un CATN représente une tâche ou un rôle que doit exécuter un agent. Un agent peut simultanément exécuter plusieurs CATNs. Les éléments qui composent un CATN sont : un nom, des états, des registres, un ensemble d'opérations sur les registres, des transitions (qui à leur tour sont composées d'un ou plusieurs arcs, en fonction du type de transition) et surtout des interactions.

- *un nom* : le nom d'un CATN doit être un nom significatif en accord avec la tâche ou le rôle qu'il représente ;
- *un ou plusieurs registres* : les registres nous permettent de contrôler le fil d'exécution d'un CATN. Chaque registre associe une variable et une valeur, laquelle dépendra du contexte de l'exécution dans une situation concrète. Ils peuvent avoir une portée locale ou globale.

[4] Cet aspect de couplage entre ATN sera développé au chapitre 6.

- *un ou plusieurs états* : chaque état représente l'exécution partielle d'un plan ou d'une tâche. Chaque arc est représenté graphiquement par un cercle étiqueté. Quand le CATN est exécuté, un seul de ses états (celui qui est en cours d'exécution) correspond à l'état courant ;
- *un ou plusieurs arcs* : un arc est représenté graphiquement par un lien au trait continu connectant deux états. Un arc peut être lié à une contrainte de franchissement et est étiqueté par un des types suivants :
  - *CATN* : arc non-terminal, représente l'invocation d'un autre CATN, y compris à lui-même ;
  - *PA (Private Action)* : arc terminal, représente des actions internes qu'un agent peut exécuter ;
  - *CATNMS (CATN Message Sender)* : arc terminal, représente l'envoi d'un message de la part de l'agent propriétaire du CATN à destination d'un autre CATN détenu par un autre agent ;
  - *CATMR (CATN Message Receiver)* : arc terminal, représente la réception par l'agent propriétaire du CATN d'un message en provenance d'un autre CATN détenu par un autre agent ;
  - *UserMR (User Message Receiver)* : arc terminal, représente la réception par l'agent exécutant le CATN d'un message en provenance de l'utilisateur. Ensemble avec les types CATNMS et CATNMR, UserMR est un de trois types d'arcs qui permettent de spécifier les interactions.
- *une ou plus transitions* : les transitions permettent de passer d'un état à un autre et peuvent être composées d'un ou de plusieurs arcs. Le formalisme CATN inclut quatre types de transition (figure 5.6) :
  - transition simple : composée d'un seul arc ;
  - transition XOR : composée de deux ou plusieurs arcs, dont un exactement doit être traversé. Les différents arcs d'une transition XOR viennent du même état origine mais peuvent arriver, chacun, à des états de destination différents ;
  - transition OR : composée de deux ou plusieurs arcs, dont, un au moins doit être traversé. Les différents arcs qui composent une transition OR viennent du même

état d'origine et arrivent au même état de destination ;

- transition AND : composée de deux ou plusieurs arcs, dont tous doivent être traversés. Les différents arcs qui composent une transition AND, viennent du même état et doivent arriver au même état de destination.

Dans toute interaction, le CATN distingue un initiateur et un ou plusieurs interlocuteurs. Il y a deux types d'interactions : terminal et non-terminal. Une interaction non-terminale est celle qui est donnée entre des arcs dont un au moins est non terminal et est représentée par une flèche solide. Une interaction terminale est donnée entre des arcs avec des actions de communication réelles et représentée par une flèche pointillée. Il faut noter qu'à une interaction non-terminale peuvent correspondre plusieurs interactions terminales.

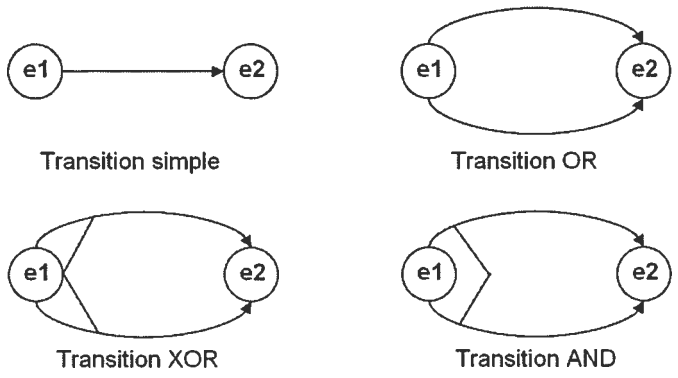

FIG. 5.6 – Représentation graphique de quatre types de transition

Le CATN du niveau le plus élevé dans un ensemble représente tout un processus, qui, suivant le contexte, pourra dériver dans l'exécution de différentes séries d'actions des agents concernés ; il représente en réalité un réseau des CATNs de différents agents concernés.

Exemple de CATN : « Offre de service »

Les deux CATNs dans la figure 5.7 décrivent le comportement d'un agent *Consum-*

mer et de plusieurs agents *Providers* de service suivant le protocole d'offre de service. Tout d'abord, l'agent *Consumer* qui exécute le CATN `applicant.catn` envoie par *broadcast* un message `request`. Une fois ce message reçu, les agents *Providers* exécutant le CATN `serviceProviderListener.catn` calculent alors le coût du service, puis envoient un message `inform` contenant ce coût au demandeur. De sa part, l'agent *Consumer* attend 1 seconde pour recevoir un message, s'il reçoit un message dans cet intervalle, il attend une seconde supplémentaire. Une fois qu'il a reçu les messages il envoie un message `LaunchService` à l'agent *Provider* proposant le coût le plus faible. Cependant, s'il n'a reçu aucun message au bout d'une seconde il affichera un message d'erreur. Du côté des agents *Providers*, l'agent recevant le message `LaunchService` va exécuter son service, tandis que les autres agents attendront 10 secondes puis afficheront un message indiquant qu'ils n'ont pas été sélectionnés.

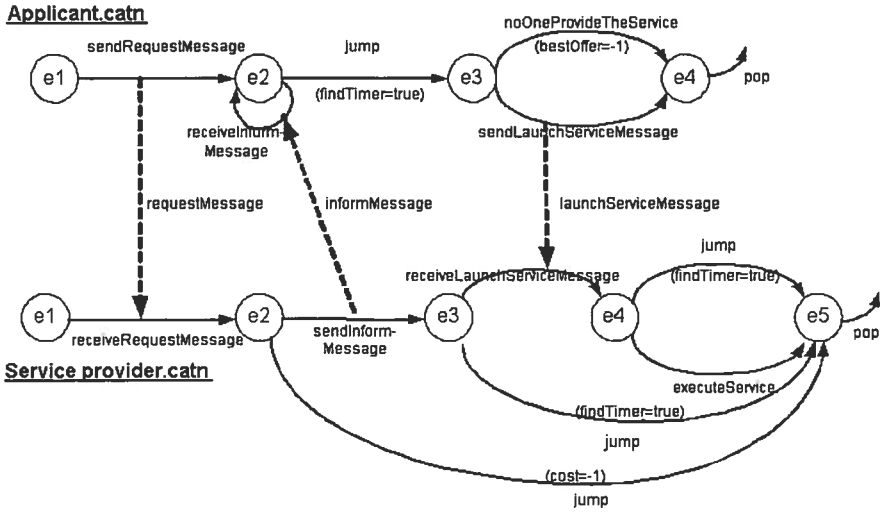

FIG. 5.7 – Schéma représentant le CATN d'offre de service

Un langage appelé CATNML est défini sous forme XML et permet de représenter textuellement un CATN. La syntaxe complète du CATNML se trouve dans l'annexe III.

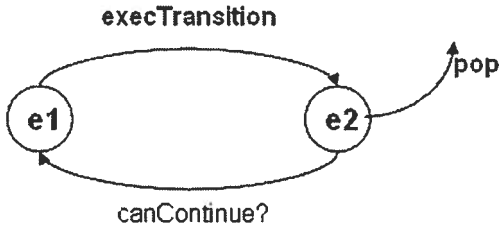

FIG. 5.8 – Interpréteur CATN

5.3.3 Interpréteur CATN

Un interpréteur CATN prend un CATN en entrée et l'interprète comme une machine à transition d'états. À chaque étape, il instancie et exécute une transition dans un contexte spécifique, lequel représente la sémantique opérationnelle de l'exécution. Dans ce chapitre, comme nous ne considérons pas encore les interactions inter-agents (lesquelles seront traitées dans le chapitre 6), un interpréteur CATN est relativement simple. Toutefois, il est intéressant que nous pouvons représenter cet interpréteur lui-même sous forme d'un CATN, tel que montré dans la figure 5.8.

Au point de vue du développeur d'agents, chaque arc d'un CATN, à l'exception des arcs de type CATN, peut être réalisé par une méthode ayant le même nom que celui de l'arc, prenant certains registres comme les paramètres d'entrée et certains autres registres comme les paramètres de sortie. Une fois exécutée, les résultats de cette méthode sont stockés dans des registres. De telles méthodes sont appelées des actions atomiques, celles-ci doivent être exécutées de manière ininterruptible par l'interpréteur. Une action atomique pour un interpréteur CATN est alors définie comme « *un morceau de code sous forme de méthode et est publié de manière à ce que l'interpréteur puisse invoquer* ».

Il est important de vérifier la faisabilité d'un CATN avant son exécution. En plus de vérifier la correction syntaxique et sémantique d'un CATN, il faut vérifier également la disponibilité de ses actions atomiques. Le premier type de vérification pourrait être

résolu complètement, en notant que CATN est un formalisme, alors que le deuxième type de vérification demande que la liste complète des actions atomiques d'un CATN soit disponible lors du chargement de celui-ci.

5.4 Combinaison du formalisme CATN et des *plugins*

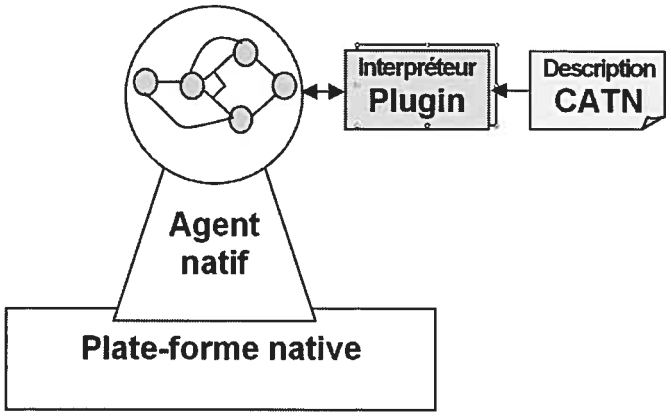

FIG. 5.9 – Agent CATN avec plugins

Les *plugins* fournissent à l'agent les fonctionnalités dont il a besoin au cours de son exécution, alors que le formalisme CATN permet de spécifier les comportements de l'agent. Nous avons combiné ces deux mécanismes afin de construire un modèle d'agents flexibles et extensibles.

Le modèle proposé est illustré dans la figure 5.9. Dans ce modèle, l'interpréteur CATN est fourni sous forme d'un *plugin* particulier. Nous allons plus loin, en fournissant les actions atomiques dont l'interpréteur CATN a besoin dans le processus d'interprétation sous forme de fonctionnalités offertes par les *plugins*.

Comme les *plugins* peuvent être mis à jour ou ajoutés à l'agent de manière dynamique, le code réel des actions atomiques d'un CATN peut être mis à jour au cours de l'exécution de ce dernier, à condition que l'appel de cette action ne soit pas encore fait. Cette « combinaison retardée » est, par certains aspects, similaire au concept « *late binding* » dans la programmation orientée objet. En conséquence, il est possible de changer

dynamiquement la réalisation d'une action atomique d'un CATN.

L'utilisation conjointe du formalisme CATN et le modèle *plugin* est alors une bonne combinaison : on peut décrire l'agent par ce formalisme et se servir des *plugins* pour fournir les actions atomiques nécessaires. Au début, l'implémentation d'un agent peut commencer par une spécification CATN détaillée, avec les actions atomiques qui réalisent de très petites tâches. Au cours de son exécution, il est possible de grouper les actions atomiques afin d'obtenir de nouvelles actions plus compliquées et de les appliquer à l'agent en changeant sa spécification CATN. On obtiendra alors une spécification dont la granularité est moins fine, mais la performance est bien améliorée. Cette évolution limite donc l'inconvénient principal de l'approche d'interprétation (qui est la performance), tout en profitant des avantages du code interprété.

Actuellement, la vérification de ce changement s'arrête au niveau syntaxique : nous ne vérifions que l'existence des actions atomiques dont la nouvelle transition a besoin ainsi que le type des paramètres nécessaires. Toutefois, une vérification complète pour assurer la terminaison normale (sans boucle indéfinie, sans état isolé, etc.) serait possible, pour autant que l'on dispose d'un CATN complet.

CHAPITRE 6

ADAPTATION PAR L'ORGANISATION ET PAR LES INTERACTIONS DES SYSTÈMES-MULTI-AGENTS

Dans les deux chapitres précédents, nous avons étudié l'adaptation dynamique au niveau des plates-formes d'agents ainsi qu'au niveau interne de l'agent. Ce chapitre présente l'adaptation dynamique du contrôle des SMAs au niveau le plus élevé : niveau systèmes d'agents. Comme le montre notre analyse du chapitre 3, à ce niveau, l'organisation du SMA et l'interaction entre agents sont les deux thèmes d'adaptation cruciaux, lesquels sont également nos deux axes de recherche. D'une part, nous présentons *deux modèles d'organisation des SMAs sous forme d'agents récursifs* : agent récursif centralisé et agent récursif distribué. Ces deux modèles d'organisation récursive permettent aux SMAs de changer dynamiquement leurs organisations selon la distribution de l'exécution. D'autre part, nous proposons un *méta-modèle d'interaction se basant sur le formalisme CATN, que nous appelons Méta-CATN.* Ce méta-modèle permet de contrôler à un niveau plus élevé le modèle d'agent CATN introduit dans le chapitre précédent et rend le SMA capable de changer dynamiquement non seulement l'interaction entre ses agents mais aussi l'interprétation de celle-ci.

6.1 Deux modèles d'agent récursif

Nous étudions dans cette section des modèles d'agents récursifs, permettant d'organiser dynamiquement des agents complexes sous forme d'organisations d'agents plus simples. Cette étape est essentielle dans la perspective de méthodologies de conception d'agent basées sur des patrons utilisant des composants eux-mêmes sous forme d'agents, ce qui offre tous les avantages de ces patrons (flexibles, extensibles, réutilisables, etc.).

Les modèles d'agents récursifs ont été présentés dans plusieurs travaux [Tambe, 1995] [Guillemet *et al.*, 1999] [Routier and Mathieu, 2001], [Satoh, 2001]. Ce terme de « modèle d'agent récursif » possède deux acceptions différentes. Dans sa première acception, il

concerne la prise en compte par un agent, lors de sa prise de décision, des connaissances qu'ont les autres agents sur lui, et ce, de façon récursive. Un agent réfléchira donc en fonction de ce qu'il sait, de ce que les autres savent et de ce qu'il sait d'eux, par exemple. Cette approche est particulièrement illustrée dans [Tambe, 1995]. Dans la seconde acception de ce terme, un agent récursif est un agent qui est défini par la composition d'éléments de base. Ces éléments peuvent être eux-mêmes décrits comme des éléments composés, et ce, jusqu'à un niveau où tous les composants sont jugés atomiques. Ce niveau atomique varie selon le domaine d'application visé. Cette approche est par exemple celle suivie dans [Guillemet *et al.*, 1999] et [Aknine *et al.*, 1998].

Dans cette thèse, laquelle se concentre sur l'architecture des agents, nous retenons donc exclusivement ce dernier sens d'agent récursif. Un agent récursif est pour nous un système perçu de l'extérieur comme un agent unique. Il est dit récursif parce qu'il est lui même composé en interne de plusieurs agents, *de facto* plus simples. Cette récursion peut se faire sur un nombre *a priori* illimité de niveaux.

Selon [Mezura *et al.*, 1999], l'intérêt d'une modélisation récursive des agents est de réduire la complexité de la modélisation, de décrire les composants selon différents niveaux d'abstraction, de réutiliser un même modèle d'agent, d'interaction et d'organisation à chacun de ces niveaux, de définir plusieurs niveaux de décomposition, selon la nature du problème et enfin de résoudre des problèmes nécessitant une adaptation dynamique du niveau d'abstraction.

Notre proposition met l'accent sur le premier point, en permettant la réutilisation d'agents ou de systèmes multi-agents existants en les encapsulant dans un nouvel agent, et sur le dernier point, en permettant à un ensemble d'agents de se combiner en un agent récursif de façon dynamique, en fonction des contraintes du problème à résoudre. Pour offrir une réutilisabilité et une encapsulation transparente d'un système multi-agents par un agent, un modèle d'agent récursif doit offrir une interface semblable à celle d'un agent pour ce système. Cela permet l'utilisation simultanée d'agents récursifs et d'agents atomiques [1].

[1] Le niveau défini comme « atomique » dépend du problème à résoudre.

Afin d'offrir une interface unifiée d'agent à un système composé d'agents récursifs, un modèle d'agents récursifs doit répondre aux questions suivantes concernant le cycle de vie d'un agent :

- création : comment plusieurs agents peuvent-ils fusionner pour former une nouvelle entité plus complexe ?
- exécution : que signifie la gestion des interactions de ce nouvel agent avec le reste du système ?
- terminaison : comment un agent peut-il sortir d'un agent récursif ?

La plupart des recherches sur des agents récursifs sont inspirées par le modèle à base de composants ([Yoo, 2000]). Dans cette approche, chaque agent membre d'un SMA est un composant, ces agents sont groupés de manière récursive à une hiérarchie d'agents. Cette approche demande pourtant que des agents se situent dans un même lieu, ce qui n'est pas approprié pour des SMAs répartis sur plusieurs sites. Une autre approche, illustrée dans [Routier and Mathieu, 2001], crée des liens de communication entre les agents d'une hiérarchie d'agents. Cette approche permet la distribution des agents récursifs, cependant il est difficile pour un agent père de contrôler strictement l'exécution de ses agents fils.

Comme chaque approche possède ses points forts et faibles et permet aux agents de s'adapter à des besoins différents, nous avons décidé de prendre les deux approches [Pham *et al.*, 2001] et de développer deux modèles d'agent récursif, dont un a été réalisé par Arnaud Dury, alors en postdoctorat au CRIM dans le cadre du projet *Guest*. L'autre modèle, ainsi que l'intégration de ces deux modèles ont été mis en œuvre par nous-mêmes et seront expliqués en détail dans les sections qui suivent.

6.1.1 Modèle d'agent récursif centralisé par la structuration des flux de contrôle

Ce modèle consiste à offrir *une plate-forme virtuelle* pour les agents internes d'une hiérarchie d'agents. Un agent *Guest* atomique est l'agrégation d'un agent natif et d'un agent universel, qui s'exécute dans un thread séparé de celui de l'agent natif. Un agent récursif est composé d'une hiérarchie d'agents se situant *sur un même serveur*. Cette hiérarchie est associée *à un seul agent natif*, qui représente la plate-forme sous-jacente

(figure 6.1). Dans cette hiérarchie :

- l'agent racine de la hiérarchie est associé au seul agent natif de cette hiérarchie. Seul cet agent peut accéder aux services offerts par la plate-forme native, à travers l'interface *Guest* de l'agent natif ;
- les agents pères sont les agents possédant au moins un agent fils. Un agent père offre à ses fils les services de base d'une plate-forme, à travers l'interface Père/Fils. Les appels des agents fils à ces services peuvent être effectués soit par l'agent père, soit répercutés sur l'interface Père/Fils de ce dernier, processus qui peut monter récursivement jusqu'à l'agent racine. Dans tous les cas, le père contrôle l'exécution de ses appels ;
- les agents fils utilisent les services offerts par leur père, ce qui leur laisse croire qu'ils s'exécutent directement sur une plate-forme.

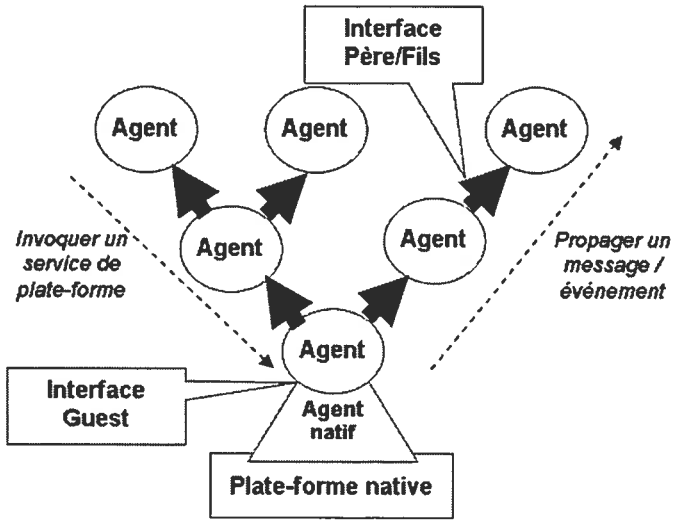

FIG. 6.1 – Modèle d'agent récursif centralisé par structuration du flux de contrôle

On a donc *deux interfaces distinctes* : l'interface entre l'agent racine et l'agent natif (couche interface *Guest*), et l'interface entre l'agent père et l'agent fils. Pour fournir les primitives de récursion les plus transparentes et assurer une vue unifiée d'agent atomique

pour le reste du système, il faut minimiser la différence entre ces deux interfaces.

L'interface Père/Fils doit soutenir la relation *récursive* et *bidirectionnelle* entre l'agent père et l'agent fils. Cette relation est récursive car une action faite sur un agent sera répercutée sur ses fils, puis sur ses petits fils, etc. Elle est aussi bidirectionnelle, car la propagation des actions peut se faire des feuilles à la racine de la hiérarchie, ou de la racine aux feuilles, selon les cas :

- des feuilles à la racine : un service demandé par un agent fils est considéré par l'agent père puis reformaté/encapsulé et transféré à l'agent de niveau plus élevé tant que l'agent père n'est pas l'agent racine. Finalement, le service correspondant de la plate-forme native sera exécuté. La transmission d'un message fonctionne selon ce principe ;
- de la racine aux feuilles : une action sur un agent père invoque d'abord des actions similaires sur ses fils avant de les appliquer sur lui-même ; ce processus s'exécute de façon récursive jusqu'au niveau le plus bas de l'arbre. Par exemple, la destruction d'une hiérarchie fonctionne selon ce principe.

Dans ce modèle, une hiérarchie d'agents est perçue comme un agent unique, soit par les autres agents, soit par les plates-formes natives. Les actions reliées au cycle de vie d'un agent comme la migration, la destruction se font récursivement pour toute hiérarchie. Cependant, l'ordre de réalisation de ces actions peut changer : la destruction est effectuée des feuilles vers la racine, tandis que la migration est effectuée d'abord sur la racine, puis sur les feuilles de la hiérarchie. Les actions spéciales pour la structuration d'une hiérarchie, comme joindre et quitter, peuvent être modélisées comme des actions de migration à l'intérieur de la hiérarchie. Selon la position des agents, il y aura l'interaction, soit entre l'agent père et l'agent fils, soit entre un agent externe et un agent interne d'une hiérarchie. Le deuxième type se déroule en deux phases : interaction entre l'agent externe et l'agent racine de la hiérarchie de l'agent interne, suivie de l'interaction entre l'agent racine et l'agent interne.

6.1.2 Modèle d'agent récursif distribué par la structuration des flux de messages

Ce modèle consiste à *rerouter dynamiquement les messages* émis et reçus par chacun des agents. Le reroutage permet de contrôler les messages émis par les agents composant l'agent récursif de façon à assurer une adresse unique, et un comportement cohérent. Ce comportement cohérent est assuré de façon récursive : chaque père contrôle les messages émis et reçus par ses fils de façon à assurer la cohérence au sein de ceux-ci (figure 6.2).

Un agent peut à tout moment demander à rejoindre un agent (simple ou récursif) existant. Cette demande est effectuée par l'envoi d'un message au père désiré. L'agent recevant cette demande est libre de l'accepter ou de la décliner. À la réception de la réponse, si celle-ci est positive, l'agent demandeur devient le fils de l'agent ayant répondu.

La sortie d'un agent récursif procède du même mécanisme, l'agent souhaitant quitter l'agent récursif émet un message à son père, qui décide du retrait ou non de cet agent de l'agent récursif. [2]

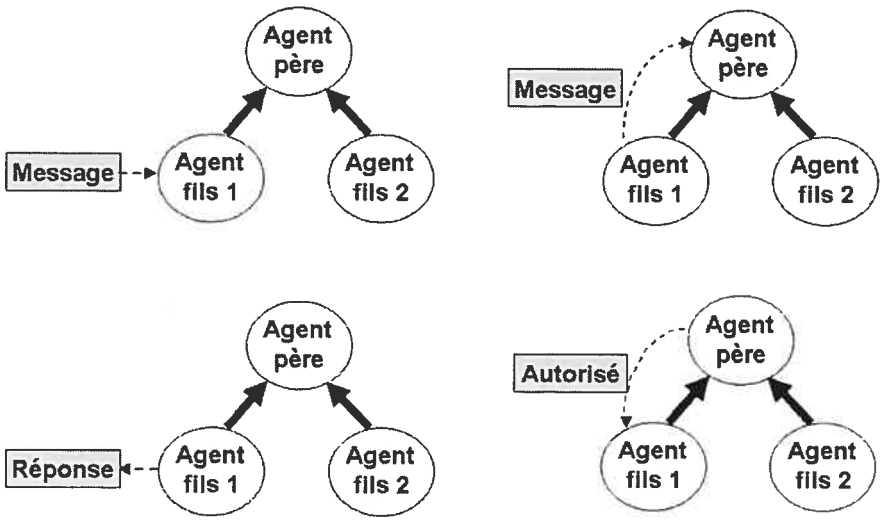

FIG. 6.2 – Reroutage dynamique des messages dans le modèle d'agent récursif distribué

[2]De façon récursive, la décision du père direct de l'agent peut elle-même remonter la hiérarchie si celui-ci possède également un père

Lorsque un agent accepte de devenir le père d'un autre agent, *il prend la responsabilité* d'autoriser les messages émis et reçus de son fils, de façon à fournir aux autres agents du système une *apparence cohérente d'agent atomique*. Les messages émis et reçus d'un agent fils sont reroutés de façon automatique vers son père. Dans le cas d'une hiérarchie multi-niveaux, le message émis par un agent pourra remonter plusieurs niveaux, et subir plusieurs transformations, avant d'être réellement émis.

Ce modèle d'agent récursif a été conçu et implémenté par un autre membre de l'équipe, Arnaud Dury, en appliquant le modèle *plugin*. Pour cela, nous avons intégré les primitives requises à la manipulation des agents récursifs dans un *plugin Guest*, qui peut être chargé et déchargé d'un agent au cours de la vie de ce dernier. De ce fait, un agent, initialement incapable de participer à un agent récursif, peut être capable de le faire par le chargement de ce plugin, à n'importe quel moment de sa vie. Au point de vue des agents extérieurs, un agent récursif est perçu comme un agent atomique. Au point de vue de la plate-forme d'exécution, un agent récursif est perçu comme un ensemble organisé d'agents distincts.

6.1.3 Intégration de deux modèles

Le tableau 6.1.3 fournit une comparaison des deux modèles d'agent récursif ci-dessus. Du point de vue organisationnelle, il n'y a pas de grandes différences entre ces deux modèles. Cependant, la véritable diférence des deux modèles se trouve au niveau de mise en œuvre et d'exécution des agents.

Nous avons intégré ces deux modèles, en permettant la combinaison de deux types d'agents récursifs dans une même hiérarchie selon les règles suivantes :

- si après une opération de migration, l'agent père et fils dans une hiérarchie distribuée se trouvent sur le même serveur, cette branche devient automatiquement une hiérarchie centralisée, afin d'optimiser la performance ;
- si après une opération de migration, l'agent fils ne se trouve plus sur le même serveur que son agent père, la hiérarchie centralisée change automatiquement en mode distribué.

Cette combinaison permet à ces deux types d'agents récursifs de fonctionner non seulement en parallèle mais aussi en collaboration. De cette manière, il est possible d'obtenir

Critères d'évaluation	Modèle d'agent récursif centralisé	Modèle d'agent récursif distribué
Temps de réponse	Plus rapide et sécuritaire parce que les actions sont invoquées directement	Délais introduits par l'envoi des messages de commande
Niveau de la répartition	Un agent récursif est forcément situé sur un seul site	Un agent récursif peut être réparti sur plusieurs sites
Relations entre les agents membres d'une hiérarchie	Moins de liberté	Plus de liberté
Liaison entre agents	Par la structuration du contrôle des agents	Par la structuration des flux de messages
Finesse du contrôle	Un agent père contrôle directement et finement le flux d'exécution de ses fils	Un agent père ne contrôle qu'indirectement le flux d'exécution de ses fils
Vue par la plateforme	un agent récursif est perçu comme un agent unique	un agent récursif est perçu comme un ensemble organisé d'agents distincts
Vue par un autre agent	un agent récursif est perçu comme un agent atomique	un agent récursif est perçu comme un agent atomique
Consommation de ressources	moins coûteux parce que les agents membres d'une hiérarchie partagent le même agent natif	plus coûteux parce qu'il demande un agent natif pour chaque agent membre d'une hiérarchie

TAB. 6.1 – Comparaison de deux modèles d'agent récursif

une organisation récursive et adaptive tout en profitant des avantages des deux modèles.

Une autre architecture, plus horizontale, des agents est disponible dans le modèle Reactalk [Giroux *et al.*, 1994] [Giroux, 1995]. Ce modèle, développé à partir du modèle Actalk [Briot, 1989b], introduit le concept méta-acteur qui permettre de réifier le comportement de l'agent et de assembler leur modèle de calcul. L'aspect méta de ce modèle rend les agents réflexifs, ceux-ci peuvent acquérir/retirer des compétences somatiques et sont donc capables de s'adapter dans leur environnement, alors que nos deux modèles d'agents récursifs laissent le soin du modèle de calcul des agents à un niveau plus élevé, que nous présenterons dans la section 6.2. De plus, nos deux modèles soulignent l'aspect de mise en œuvre des systèmes d'agents, notamment la distribution spatiale des agents membres.

6.1.4 Exemple de prédateurs et proie

Cet exemple consiste en un jeu dans lequel plusieurs prédateurs cherchent à encercler une proie. La proie est capturée lorsque au moins n prédateurs se retrouvent sur un même endroit avec elle.

Cet exemple est modélisé comme suit :

- Un proie est un agent mobile ;
- Il y a plusieurs prédateurs affamés, chacun est également un agent mobile ;
- Chaque endroit est un serveur d'agents ;
- Les prédateurs et la proie se déplacent de manière aléatoire d'un serveur à un autre ;
- Si, à un moment donné, la proie et n prédateurs se trouvent sur le même serveur, la proie est considérée capturée.

Les prédateurs, dont l'intérêt commun est de chasser des proies, se groupent en un groupe de discussion sous forme d'un agent récursif distribué. De cette manière, non seulement les prédateurs peuvent communiquer les uns avec les autres, mais ils peuvent également migrer de façon transparente, tout en continuant à recevoir leurs messages sans que leurs partenaires ne soient explicitement prévenus de leurs déplacements. La chasse des proies est alors consituée de deux étapes :

1. Patrouiller : Les prédateurs se déplacent séparément, en supposant que cette stratégie permet d'augmenter la chance de trouver une proie. Chaque fois qu'un prédateur détecte une proie (le prédateur et la proie se trouvent en même temps sur le même serveur), il notifie les autres prédateurs dans le groupe.
2. Chasser : À partir de ce moment, chaque fois que deux prédateurs se croisent (sur un même serveur), ils se forment en un agent récursif centralisé et se déplacent désormais ensemble. En restant groupés, les prédateurs espèrent être en nombre suffisant afin de capturer éventuellement la proie.

L'exemple ci-dessus souligne la nécessité des organisations pertinentes d'un SMA afin d'achever son objectif et illustre l'utilité de nos deux modèles d'agents récursifs. De plus, les serveurs d'agents peuvent être de différentes plates-formes : ce fait démontre bien la capacité de migration hétérogène des agents *Guest*.

6.2 Modèle MetaCATN

Dans cette section, nous présentons le modèle d'agent MetaCATN, en suivant l'approche méta-modélisation et en nous basant sur le formalisme CATN. Ce modèle peut ensuite être utilisé au-dessus de différentes plates-formes multi-agents, tells que *Guest* [Magnin *et al.*, 2002c] ou Jade [Jade, 2003]. En appliquant ce modèle, il est possible de développer des SMAs capables d'adaptabilité dynamique [Pham *et al.*, 2004].

En suivant l'approche méta-modélisation, qui met en avant la représentation explicite et séparée du contrôle [Sahraoui Houari, 1995], il est possible d'achever une dynamicité dans l'exécution des systèmes. Nous modélisons alors les SMAs suivant une architecture à deux niveaux : au niveau de base, le comportement des agents est décrit par un formalisme interprétable. Au niveau méta, ces descriptions peuvent être interprétées tout en se prêtant aux modifications dynamiques. Ceci permet dès lors de modifier le comportement des agents à tout moment, y compris lors de la phase d'exécution.

6.2.1 Description générale

MetaCATN est un modèle de comportement d'agents, à base de CATN réflexif. Un agent MetaCATN a une architecture à deux niveaux (figure 6.3) :

- Le niveau de base consiste en la spécification du comportement de l'agent, y compris ses interactions avec d'autres agents, sous la forme de CATN. C'est aussi à ce niveau que se trouve l'implémentation des actions atomiques dans un langage réflexif, tels que Java, Smalltalk, etc. Ces spécification et implémentation sont disponibles au niveau méta ;
- Au niveau méta, il y a un interpréteur CATN dont le rôle est d'interpréter le CATN spécifiant l'agent. Le point particulier ici est que cet interpréteur est lui-même modélisé par le formalisme CATN. Il se trouve également à ce niveau deux autres composants principaux : le CATNCoordinator et le CommunicationManager. Le CATNCoordinator est responsable de contrôler l'exécution de l'interpréteur CATN (c'est-à-dire interpréter celui-ci) et de synchroniser les modifications sur le CATN au niveau de base avec tous les CATNs concernés des autres agents, en coopérant

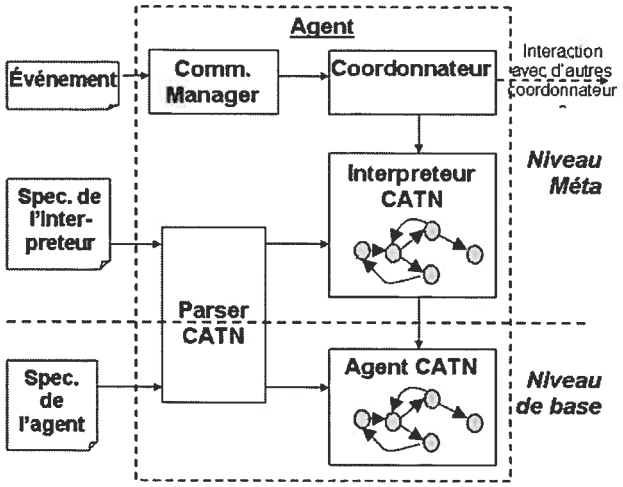

FIG. 6.3 – Modèle MetaCATN

avec ses homologues de ces derniers, à travers les messages de communication. Le CommunicationManager travaille avec la plate-forme d'agents pour l'envoi et la réception de ces messages.

Ces deux niveaux partagent un parseur CATN, dont l'utilité est de transformer une spécification en langage CATNML en une représentation interne, celle-ci sera ultérieurement utilisée par un interpréteur CATN.

Dans ce modèle, les agents sont spécifiés par le formalisme ATN et les interactions entre eux sont représentées par les arcs d'interaction. Deux types de changement sont alors possibles :

- changement du modèle spécifié par CATN : c'est le changement horizontal. Ce type de changement peut être fait continuellement avec l'aide de l'interpréteur CATN ;
- changement de l'interprétation : cela demande le changement de l'interpréteur ; nous avons ainsi un changement vertical. Notre approche dans ce type de changement est atteinte en rendant l'interpréteur CATN dynamique par un niveau méta : l'interpréteur CATN est, quant à lui, représenté par le formalisme CATN.

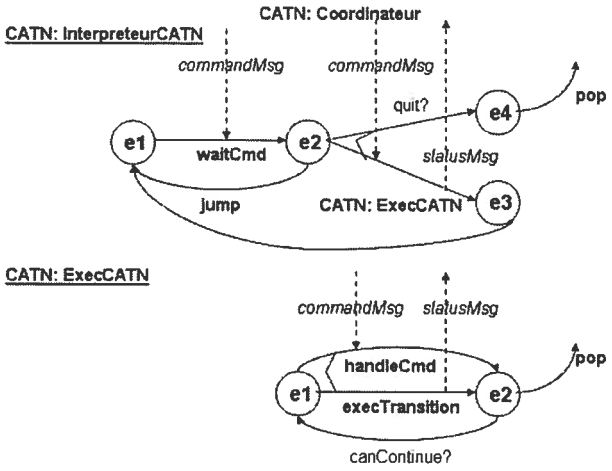

FIG. 6.4 – Interpréteur CATN modifié

6.2.2 Interpréteur CATN modifié

Cet interpréteur CATN est une extension de l'interpréteur CATN de base présenté dans le chapitre 5, avec l'introduction du composant CATNCoordinator (figure 6.4). Dans la transition de l'état *e1* à *e2*, il attend la commande du CATNCoordinator. Si la commande est d'exécuter un CATN, il exécute celui-ci en prenant l'arc ExecCATN, qui est en effet une boucle d'exécution des transitions. Cependant, cette interprétation peut être modifiée, en fonction du message de commande envoyé par le CATNCoordinator lors de l'exécution d'une transition. Cette commande peut être d'arrêter l'interprétation du CATN courant, d'informer qu'un CATN partenaire a été changé ou de demander que l'interprétation de la transition prochaine soit changée, etc. À la fin de l'interprétation d'une transition, un message informant l'état d'exécution sera envoyé au CATNCoordinator.

6.2.3 CATNCoordinator

Ce composant est responsable de deux tâches principales : la manipulation (c'est-à-dire l'interprétation, la modification ou le remplacement) de l'interpréteur CATN et la collaboration de l'évolution coordonnée des CATNs reliés dans une même tâche du SMA,

au cas où un changement sur l'un d'eux pourrait demander la prise en compte des autres ou même conduire à des changements de ceux-ci.

Les interactions entre l'interpréteur CATN et le Coordinator sont réalisées sous la forme d'envoi de messages, comme pour tous les autres CATNs. En réalité, le CATN-Coordinator est vu par l'interpréteur comme un CATN particulier, bien que celui-ci soit virtuel. Dès lors, le comportement d'un interpréteur CATN peut lui-même être décrit par le formalisme CATN.

L'évolution enchaînée des CATNs reliés est fondée sur une communication d'événement par publication/inscription. Chaque fois qu'il y a une modification de l'un d'eux, son interpréteur publie un événement auprès du CATNCoordinator. Cet événement sera propagé par ce dernier à tous les CATNCoordinators des agents reliés pour que ceux-ci puissent prendre en compte des modifications nécessaires. Ce processus permet qu'un changement à l'intérieur d'un agent puisse être conscient par les agents concernés et conduire à une co-évolution de ceux-ci.

6.2.4 CommunicationManager

Ce composant gère tous les messages d'interaction du CATN au niveau de base et de l'interpréteur CATN au niveau méta. Sa première tâche est de distribuer les messages d'interaction à son CATN destinataire ; l'autre tâche est d'envoyer des messages d'interaction aux autres CATNs, soit en direct, soit par le mode « *broadcast* ». Généralement, ce service est déjà fourni par la plate-forme d'agents et ce composant se contente de « *wrapper* »ce service.

6.2.5 Exemple de l'exécution du modèle

Nous revenons à l'exemple des agents implémentant le protocole d'offre de service introduit dans la section 5.3. Supposons que l'agent `Applicant` veuille changer le mode de communication normale par celui encrypté, c'est-à-dire encrypter les messages avant de les envoyer et décrypter les messages après réception. Afin de réaliser cette modification, il est nécessaire de changer non seulement l'agent `Applicant` mais aussi les agents `ServiceProvider`. Trois voies de modification sont alors possibles :

- Modification des actions atomiques : dans ce cas, les actions atomiques réalisant les opérations « *send* » et « *receive* » seront substituées par une nouvelle version supportant encryptage et décryptage. Le CATNCoordinator de l'agent `Applicant` analyse sa spécification et détecte que ces modifications impliquent les `ServiceProvider`, il les notifie alors par l'envoi d'un message décrivant ces modifications à ses homologues de ces agents. Les CATNCoordinators des agents `ServiceProvider` reçoivent cette notification et entraînent leurs interpréteurs CATN de substituer les actions atomiques correspondantes.
- Modification des CATNs : par cette voie, le CATN est la cible de modification. L'agent `Applicant` demande à l'interpréteur CATN de substituer tous les arcs *« send »* par un sous-CATN se composant de deux arcs : un arc d'encryptage et un arc *« send »* normal. Tous les arcs *« receive »* seront substitués par un autre sous-CATN de deux arcs : un arc *« receive »* et un arc de décryptage. Deux nouvelles actions atomiques réalisant l'encryptage et le décryptage devront également être ajoutées. Le CATNCoordinator de l'agent `Applicant` analyse ensuite sa spécification et détecte que ces modifications impliquent les agents `ServiceProvider`, il les notifie alors en envoyant un message décrivant ces modifications à ses homologues de ces agents. Les CATNCoordinators des agents `ServiceProvider` reçoivent cette notification et entraînent leurs interpréteurs CATN de substituer les arcs correspondants.
- Modification des interpréteurs : en suivant cette direction, la modification se fait au niveau méta, à l'interpréteur CATN. L'agent `Applicant` modifie son interpréteur CATN de manière que l'interprète d'un arc *« send »* constitue désormais de l'encryptage et de l'envoi de message et l'interprétation d'un arc *« receive »* constitue maintenant de la réception et du décryptage de message. Le CATNCoordinator de l'agent `Applicant` analyse ensuite sa spécification et détecte que ces modifications impliquent les agents `ServiceProvider`, il les notifie alors en envoyant un message décrivant ces modifications à ses homologues de ces agents. Les CATNCoordinators des agents `ServiceProvider` reçoivent cette notification et réalisent les modifications correspondantes à leurs interpréteurs CATN.

Finalement, tous les agents partagéant ce CATN (l'agent `Applicant` et les agents `ServiceProviders`) passent du mode de communication normale au mode de communication encryptée. Il est à noter que, dans les trois cas, la description de toutes les modifications nécessaires peuvent éventuellement être groupée dans un « *script* » de modification et inclue dans le message de notification.

6.2.6 Évaluation du modèle

Dans le modèle MetaCATN, le comportement d'agents est modélisé par des CATNs et peut être ensuite interprété au niveau méta. Par conséquent, il existe trois formes de modification de ce comportement :

- modifier le CATN spécifiant ce comportement ;
- modifier l'interprétation du modèle par la modification de son interpréteur ;
- modifier la cible de cette interprétation, c'est-à-dire l'implémentation des actions atomiques de l'agent, ce qui demande de réécrire ou d'ajouter le code dans le langage choisi pour développer l'agent.

Comme toutes ces formes de modification peuvent être réalisées lors de l'exécution des CATNs et conduisent immédiatement à des changements dans le comportement d'agents, l'adaptation dynamique des agents peut alors être achevée en différentes directions au moyen d'une de ces formes ou par la combinaison de certaines parmi elles :

- de bas en haut : le comportement d'agents peut être changé tout d'abord au niveau CATN puis au niveau de son interpréteur ;
- de haut en bas : l'interpréteur CATN au niveau méta se modifie et ce dernier effectue des changements au niveau de CATN ;
- du local vers le global : une modification dans un CATN entraîne de manière enchaînée des changements dans les autres CATNs concernés ;
- du global vers le local : on peut changer simultanément plusieurs CATNs reliés ;
- du mode débranché vers le mode en-ligne : on peut développer et tester des CATNs ou des interpréteurs CATN séparément, tout en laissant les agents exécuter leur version courante. Par la suite, la nouvelle version peut être chargée dynamiquement par les agents, ce qui provoque immédiatement le changement de leur comportement,

sans nécessiter leur arrêt.

Bien que le niveau méta dégrade la performance de nos agents (à cause de l'interprétation de l'interpréteur CATN lui-même), ces derniers sont désormais entièrement réfléxifs et modifiables. Plus important, le niveau méta s'occupe les interactions au niveau multi-agents, l'adaptation ne se limite plus à l'intérieur d'un agent mais être disponible dans le SMA entier. Cette adaptabilité nous semblent indispensables à de futures applications ouvertes. Il nous reste cependant à travailler sur l'exploitation de tels mécanismes ; ce qui passe notamment par la détection des situations nécessitant une adaptation des agents, ainsi que la détermination des modifications à effectuer par la suite.

6.3 Conclusion

Dans la première partie de ce chapitre, nous avons présenté deux modèles d'organisation récursive des agents, l'un est centralisé et l'autre est distribué. Ces deux modèles peuvent être utilisés en combinaison et la transformation de l'un à l'autre peut être réalisé de manière automatique, en fonction de la distribution spatiale des agents membres. De cette façon, il est possible d'obtenir une organisation récursive, distribuée et adaptive de systèmes multi-agents.

Dans la deuxième partie du chapitre, nous avons introduit le modèle MetaCATN, lequel rend possible la dynamicité des agents à deux niveaux : d'une part, il est possible de modifier en cours d'exécution le CATN décrivant le comportement d'un agent, tout en permettant de modifier l'interprète de ce même CATN (lequel est également décrit sous forme de CATN). Cette dynamicité est renforcée par l'aspect distribué des CATNs. Il est en effet possible de décrire et d'exécuter la modification d'un protocole de communication entre agents de manière unitaire, et non agent par agent. De plus, cette approche pourrait être appliquée à d'autres formalismes d'automates, pour autant que le couplage entre agent soit pris en compte.

Le modèle MetaCATN peut s'exécuter au-dessus de plusieurs plates-formes d'agents en utilisant leurs services à travers des connecteurs. Une version de ce modèle a été implémentée en Java avec un connecteur vers la plate-forme *Guest* (section 7.4.2).

CHAPITRE 7

MISE EN ŒUVRE AU TRAVERS DE LA PLATE-FORME *GUEST*

Dans ce chapitre, nous présentons la plate-forme multi-agent *Guest*, développée par l'équipe GLIC (Génie logiciel et ingénierie de la connaissance) du CRIM (Centre de recherche informatique de Montréal)[1], auquel nous appartenions. Cette plate-forme concrétise nos modèles théoriques proposés dans les chapitres précédents et fournit un cadre de travail pour le développement des systèmes multi-agents adaptatifs.

Après une courte introduction des caractéristiques importantes et distinctes de la plate-forme *Guest* dans la section 7.1, les prochaines sections détaillent la mise en œuvre de nos modèles d'adaptation dynamique du contrôle aux trois niveaux d'adaptation : niveau plate-forme, niveau agent et niveau SMA. Au premier niveau, nous avons conçu les agents génériques *Guest* en nous basant sur le modèle d'agent universel introduit dans le chapitre 4. Cette réalisation est décrite au travers des classes et des interfaces implémentant le modèle d'agents *Guest* (section 7.2.1), le modèle d'événements (section 7.2.2) et le modèle de communication (section 7.2.3). Nous décrivons dans la section 7.2.4 la micro-plateforme CORBAHost permettant à nos agents *Guest* de fonctionner de manière complètement indépendante des plates-formes multi-agents existantes. Au niveau agent, nous présentons l'implémentation du modèle d'extension par *plugins* (section 7.3.1) et de l'interpréteur CATN (section 7.3.2). Au niveau le plus élevé, niveau SMA, la mise en œuvre des deux modèles d'agents récursifs et des composants opérationnels du modèle MetaCATN est détaillée (section 7.4.1 et section 7.4.2). En mettant tout cela ensemble, le cadre de travail *Guest* permet de réaliser des applications multi-agents adaptatives réelles.

[1] http ://www.crim.ca

7.1 Introduction

La plate-forme *Guest* est une plate-forme multi-agents générique et extensible, permettant de développer et de déployer des SMAs répartis, mobiles et multi-plateformes. Développée à partir de zéro (*« built from scratch »*) au sein de l'équipe GLIC du CRIM, cette plate-forme est devenue *open-source* en août 2003. Les caractéristiques principales de la plate-forme *Guest* sont les suivantes :

- ***pur Java*** : Le langage de développement utilisé dans *Guest* est Java, un langage orienté objet très populaire dans le domaine des applications multi-agents. Ce choix limite pourtant *Guest* à des environnements basés sur Java (version 1.2 ou plus) ;
- ***Indépendant des plates-formes*** : *Guest* rend interopérables entre elles plusieurs plates-formes multi-agents *a priori* incompatibles et offre un modèle d'agent universel qui est indépendant des types de plates-formes, tout en leur offrant de nouvelles fonctionnalités. Actuellement, *Guest* supporte cinq plates-formes natives[2] :
  1. Aglets [Aglets, 1998], la plate-forme d'agents générique et mobile pionnière en Java ;
  2. Grasshopper [Grasshopper, 2000], une plate-forme commerciale qui est compatible au standard MAF ;
  3. Jade [Jade, 2003], la plate-forme multi-agents la plus populaire et la plus active dans le domaine ;
  4. Voyager [Voyager, 1999], une première tentative d'intégration des agents mobiles et CORBA ;
  5. CORBAHost, notre propre micro-plate-forme d'agents fonctionnant au-dessus de CORBA (plus spécifiquement, la version CORBA fournie en standard à partir de la version Java 1.4).
- ***Extensible*** : *Guest* offre un modèle d'agents flexible, permettant de développer et de déployer facilement de nouvelles fonctionnalités par le mécanisme de *plugins* ;

[2] L'intégration de la plate-forme Concordia [Concordia, 1999] a été interrompue faute d'une version publique fonctionnant sous Java 1.2.

- ***Dynamique*** : *Guest* offre un modèle d'agents dynamique facilitant le développement de systèmes multi-agents adaptatifs ;
- ***GUI*** : *Guest* fournit des outils graphiques, tels que `RegionViewer` (figure 7.1) et `GuestLauncher`, lesquels permettent de visualiser et de gérer de façon centralisée des applications multi-agents distribuées tout en automatisant le lancement des différents serveurs et agents.

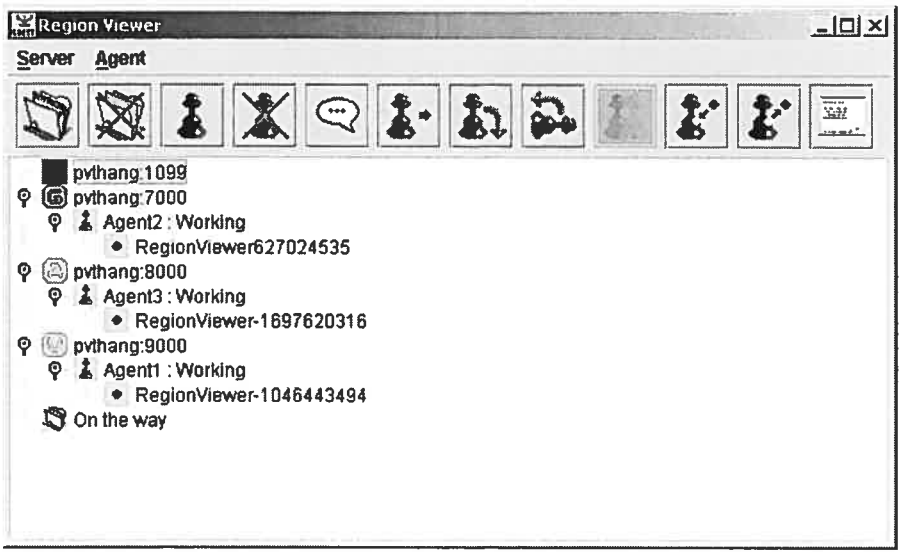

FIG. 7.1 – Gestion centrale à travers l'interface graphique RegionViewer

La plate-forme *Guest* est conçue pour trois types de développeurs, selon leur besoin et leurs connaissances de la plate-forme :

- **les développeurs** de la plate-forme : ils connaissent très bien la plate-forme et peuvent la modifier et ajouter une nouvelle interface pour une nouvelle plate-forme native ;
- **les contributeurs** qui contribuent à l'extension de la plate-forme en offrant des *plugins* et des interpréteurs : ils travaillent principalement avec l'interface des *plugins* et l'interpréteur CATN ;
- **les utilisateurs** qui développent des applications multi-agents en utilisant la plate-

forme : ils ne s'intéressent qu'aux interfaces publiques de *Guest*.

La plate-forme *Guest* est composée de plusieurs classes, telles que montrées dans la figure 7.2. Sauf les agents natifs, les autres classes sont indépendantes des plates-formes natives.

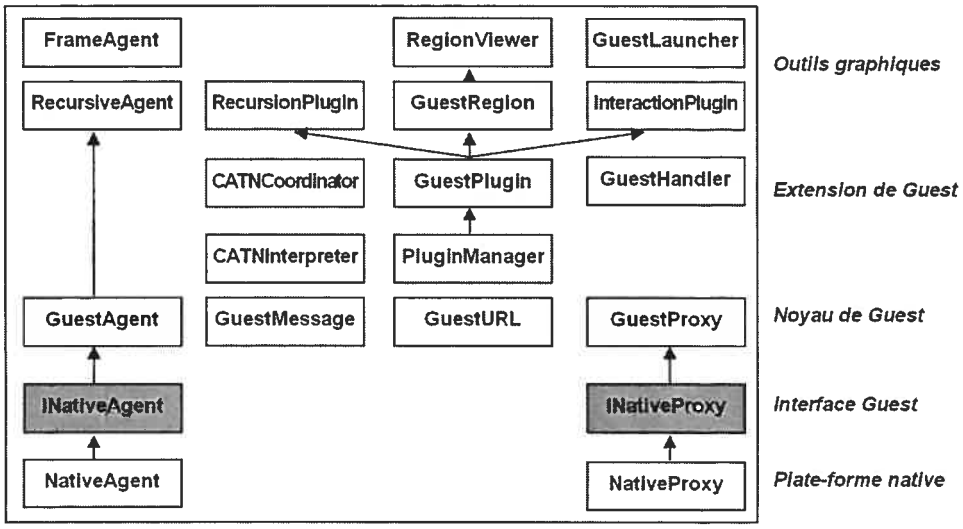

FIG. 7.2 – Classes principales de la plate-forme *Guest*

7.2 Mise en œuvre au niveau plate-forme

7.2.1 Modèle d'agent *Guest*

L'agent *Guest* est au cœur de la plate-forme ; grâce à lui l'utilisateur peut créer et manipuler des agents indépendamment des plates-formes natives. Cet agent, dont le modèle d'objet est illustré dans la figure 7.3, dispose d'un identificateur unique et offre les fonctionnalités nécessaires au développement des applications multi-agents ; par exemple la migration, la communication, etc.

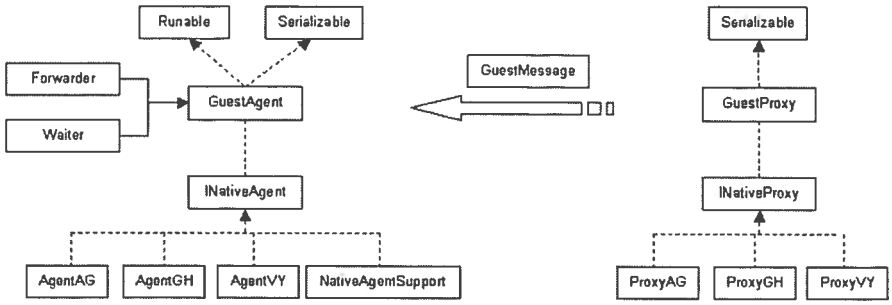

FIG. 7.3 – Modèle d'objet de l'agent *Guest*

7.2.1.1 Interface *Guest*

Cette interface constitue, en effet, deux interfaces : une entre l'agent natif et l'agent *Guest* et l'autre entre le *proxy* natif et le *proxy Guest.*

L'interface entre un agent natif et un agent *Guest* contient les méthodes concernant le cycle de vie de l'agent, les méthodes établissant le lien entre l'agent natif et l'agent *Guest*, ainsi que les méthodes de communication. Cette interface est alors définie telle que la suivante :

```
public interface INativeAgent {
    public void moveGuestAgent(GuestURL guestURL, long fwdLiveTime);
    public void removeGuestAgent();
    public void saveGuestAgent(long sleepTime);
    public void cloneGuestAgent(GuestId id, GuestURL address);
    public void restoreGuestAgent(GuestMessage guestMsg);
    public Object handleSyncGuestMessage(GuestMessage guestMsg);
    public void handleAsyncGuestMessage(GuestMessage guestMsg);
    public void setGuestAgent(GuestAgent agent);
    public Object getNativeServerInfo();
    public Object getNativeAgent();
    public Object invokeOnNativeAgent();
}
```

Nous utilisons Java Reflection afin de charger correctement les classes d'un agent natif

correspondant à une plate-forme native. Dès lors, la classe représentant un agent natif sur une plate-forme doit implémenter l'interface *INativeAgent* et prendre le nom « Agent + code de la plate-forme native »Par exemple, l'agent natif de la plate-forme Voyager doit avoir le nom `AgentVY`[3].

D'une façon similaire, la classe représentant un *proxy* natif pour une plate-forme doit implémenter l'interface `INativeProxy` et prendre le nom « Proxy + code de la plate-forme native ». Par exemple, le *proxy* natif de la plate-forme Voyager doit prendre le nom `platform.ProxyVY`. L'interface INativeProxy est donc définie comme suit :

```
public interface INativeProxy {
    public void createGuestAgent(GuestURL guestURL, GuestAgent guestAgent);
    public Object sendSyncGuestMessage(GuestMessage guestMsg);
    public void sendAsyncGuestMessage(GuestMessage guestMsg);
    public INativeProxy getGuestProxy(GuestId identifier, GuestURL guestURL);
}
```

7.2.1.2 Classe GuestAgent

La classe GuestAgent est la classe principale de la plate-forme *Guest*. Cette classe offre plusieurs méthodes pour des buts différents :

- les méthodes de comportement : ces méthodes sont vides et devront être implémentées par le développeur dans les classes héritées selon les besoins. Par exemple, la méthode `init()` est une méthode qui est appelée juste avant l'exécution de l'agent *Guest*. Dans cette méthode, l'utilisateur peut mettre du code pour initialiser l'agent, charger des *plugins*, etc. La méthode `live()`, quant à elle, décrit le comportement de toute la vie de l'agent. Le traitement des messages de la communication asynchrone et synchrone est spécifié dans les méthodes `handleSyncGuestMessage()` et `handleSyncGuestMessage()`, respectivement ;
- les méthodes finalisées fournissant l'accès aux services de base offerts dans *Guest* : l'appel à la méthode `move()` cause la migration de l'agent sur un autre serveur, alors que l'invocation de la méthode `remove()` signifie la fin de l'agent. Les méthodes

[3]Pour faciliter l'intégration d'une nouvelle plate-forme et assurer les droits d'accès aux méthodes de l'agent *Guest*, une classe spéciale a été développée : `NativeAgentSupport`

`sendSyncGuestMessage`, `sendAsyncGuestMessage` et `sendFutureGuestMessage()` sont dédiées à l'envoi des messages de différents modes de communication. La méthode `invokeOnNativeAgent()` permet à l'agent d'exploiter les méthodes propriétaires de la plate-forme native ;

- les méthodes privées de l'agent pour la gestion interne de la communication ;
- les méthodes d'accès au `PluginManager` et au MetaCATN.

7.2.1.3 Classe GuestURL

Chaque agent *Guest* possède un identificateur. Un agent peut être localisé par l'adresse du serveur sur lequel l'agent s'exécute suivit de son propre identificateur. Pour identifier un agent, il faut donc s'assurer que le couple (adresse, identificateur) est unique. C'est la responsabilité du développeur d'assurer l'unicité universelle du nom d'un agent[4].

L'adresse du serveur sur lequel l'agent *Guest* s'exécute est encapsulée dans la classe `GuestURL`. Les composants suivants sont obligatoires :

- le code de la plate-forme native : jusqu'à date, *Guest* accepte les cinq plates-formes VY (Voyager), GH (Grasshopper), AG (Aglets), JD (Jade) et CH (CorbaHost) ;
- le nom de la machine : soit une adresse IP, soit un nom de domaine[5] ;
- le port sur lequel le serveur écoute : un entier $<$ 65585 ;

Les composants suivants sont valides seulement pour la plate-forme Grasshopper :

- l'agence : telle que définie dans la plate-forme Grasshopper ;
- la place : la valeur par défaut est `InformationDesk`.

Le composant suivant est valide seulement pour la plate-forme Jade :

- container : la valeur par défaut est `Main-Container`.

La représentation textuelle pour une adresse `GuestURL` est de la forme suivante :

```
plate-forme://host:port[/AgencyName[/PlaceName]|ContainerName]
```

[4] *Guest* offre néanmoins la méthode `createUUID()` pour la génération d'un nom globalement unique.
[5] Grasshopper ne reconnaît pas le nom `localhost`.

7.2.2 Cycle de vie d'un agent *Guest*

Au cours de sa vie, un agent *Guest* peut se trouver dans un des états suivants : `CREATION`, `EXECUTION`, `SAVE`, `MIGRATION`, `REMOVAL`. La validité du passage d'un état à un autre est montrée dans le tableau 7.2.2 (le passage d'un état sur la première colonne à un autre sur la première ligne) :

	CREATION	EXECUTION	SAVE	MIGRATION	REMOVAL
CREATION	non	oui	non	non	non
EXECUTION	non	non	oui	oui	oui
SAVE	non	oui	non	non	non
MIGRATION	non	oui	non	non	non
REMOVAL	non	non	non	non	non

TAB. 7.1 – Validité du passage d'un état à un autre dans la vie de l'agent *Guest*

Guest permet d'observer la plupart des changements d'état d'un agent : migration, exécution, désactivation et réactivation suivant le *pattern* « Événement - Observateur ». Il est possible d'avoir plusieurs observateurs d'un événement, lesquels peuvent être ajoutés ou retirés à n'importe quel moment de la vie de l'agent. Tous les observateurs seront signalés à chaque événement, cependant dans l'ordre séquentiel de leur ajout.

7.2.2.1 Création d'un agent

Cette action se compose de trois étapes : créer un agent natif sur le serveur indiqué, ensuite lui envoyer l'objet agent *Guest* et associer cet objet à l'agent natif, enfin initialiser et activer l'agent *Guest*.

7.2.2.2 Migration

Guest ne fournit que la migration faible. Le principe d'implémentation consiste à sérialiser les différents objets constituant l'agent, puis à les envoyer au serveur de destination qui va recréer les différents objets. Comme Java ne permet pas d'avoir accès à l'état d'exécution d'un programme (données dans la pile, le compteur ordinal, etc.), l'utilisateur devra alors gérer la reprise de l'exécution au bon endroit.

Pour faire migrer un agent *Guest* entre deux serveurs, un nouvel agent natif est crée sur le serveur de destination tandis que le premier agent natif devient un agent spécial de

type `Forwarder` qui joue le rôle d'un représentant de l'agent à son ancienne position. Sa responsabilité est de recevoir les messages envoyés à l'agent lorsque l'agent est en route et de les transmettre à l'agent quand celui-ci arrive à destination. Ce `Forwarder` est « invisible » et n'est pas accessible, sauf par l'agent propriétaire. De plus, un `Forwarder` termine son existence après une période prédéterminée.

Pour observer la migration d'un agent, l'observateur doit implanter l'interface suivante :

```
public interface IMigrationListener {
    public void migrationAccepted(GuestURL newDest);
    public void migrationAborted(GuestURL newDest);
    public void migrationFailed(MigrationException e);
    public void postMigration(GuestURL newDest);
}
```

7.2.2.3 Activation / Désactivation

Le principe d'implémentation consiste à sérialiser les différents objets constituant l'agent, puis à les écrire sur le disque dur. On constate que seules les données dans le tas sont accessibles, mais non l'état d'exécution (cela est la limite de Java). L'utilisateur devra donc gérer la reprise de l'exécution au bon endroit, tout comme lors d'une migration.

Durant la désactivation d'un agent *Guest*, l'agent natif associé vit toujours mais l'agent *Guest* est remplacé par un objet spécial `Waiter`. La responsabilité de `Waiter` est d'attendre un signal pour reprendre l'exécution de l'agent `Guest`. Personne ne peut accéder directement au `Waiter`.

Pour observer la désactivation / activation d'un agent, l'observateur doit implanter l'interface suivante :

```
public interface ISaveListener {
    public void saveAccepted();
    public void saveAborted();
    public void saveFailed(SaveException e);
    public void postRestore ();
}
```

7.2.2.4 Destruction d'un agent

Cette action se compose de deux étapes : mettre fin à l'agent *Guest* et ensuite détruire l'agent natif associé à cet agent. Avant la destruction, un agent informe tous ses `Forwarders` pour que ceux-ci lui envoient tous les messages restant dans leurs boîtes aux lettres.

Pour être au courant de la fin d'un agent, l'observateur doit implanter l'interface suivante :

```
public interface IRemovalListener {
    public void removalAccepted();
    public void removalAborted();
    public void removalFailed(GuestException e);
}
```

7.2.3 Communication dans *Guest*

L'agent *Guest* est en principe autonome [6] : la façon correcte d'interagir avec un agent est de lui envoyer un message, lequel sera interprété dans le sens voulu par l'agent.

Les agents *Guest* communiquent entre eux par l'envoi de messages de façon *point-à-point*. Pour envoyer un message à un agent, il faut obligatoirement passer par son *proxy*, préalablement récupéré. Si deux agents sont sur le même serveur, la communication est faite directement (appel de méthode), sinon *Guest* utilise le service de communication offert par la plate-forme native, lequel est RMI, dans le cas de Grasshopper et Aglets, ou CORBA dans le cas de Voyager. Enfin, lorsque la communication a lieu entre deux agents se trouvant sur des plates-formes différentes, elle est réalisée par une connexion RMI mise en place par *Guest*. Que la communication soit directe ou indirecte, elle est réalisée de manière transparente à l'utilisateur.

Guest offre trois types de communication de base : *synchrone, asynchrone et future réponse*. Le processus détaillé de traitement des messages de ces trois types est décrit dans les deux schémas qui suivent (figures 7.4 et 7.5).

[6] Toutefois, afin de faciliter la conception et la mise en place de tels agents, il est possible de les « piloter » (migration, etc.) directement à partir de l'outil graphique RegionViewer.

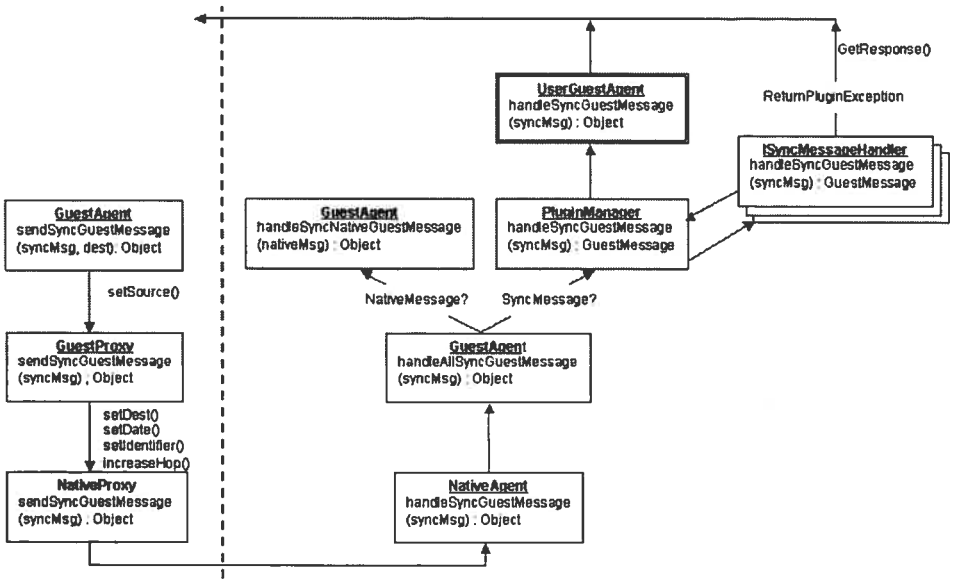

FIG. 7.4 – Schéma de communication synchrone dans *Guest*

- Communication synchrone : lorsque l'agent établit une communication synchrone, il envoie son message et attend que l'agent receveur ait fini de le traiter et lui fasse parvenir sa réponse[7] ;
- Communication asynchrone : lorsqu'un agent (ou un processus) établit une communication asynchrone, il envoie son message et continue son exécution sans attendre que l'agent receveur ait fini de traiter le message. Au cas où le message ne peut pas être envoyé, une exception est générée ;
- Communication « future réponse » : seuls deux agents *Guest* peuvent utiliser cette procédure entre eux. Avant d'envoyer un message, il faut préparer un gestionnaire de futures réponses, qui traitera éventuellement la réponse de l'agent receveur.

[7] L'utilisation de ce type de message est fortement déconseillée, d'autant plus que le mode « réponse future » permet d'obtenir la même fonctionnalité.

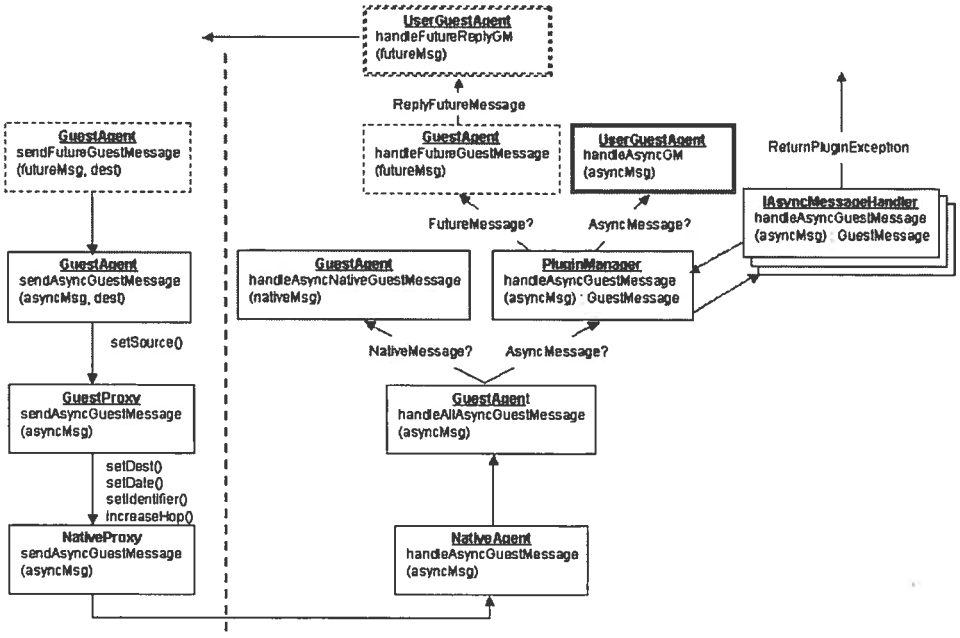

FIG. 7.5 – Schéma de communication asynchrone dans *Guest*

7.2.3.1 Classe GuestMessage

Les informations transmises entre les agents *Guest* sont encapsulées dans la classe `GuestMessage`. Cette classe contient :

- l'étiquette d'un message permettant de connaître le but du message ;
- l'identificateur et l'url de l'agent expéditeur permettant de savoir qui envoie le message. Ils sont initialisés quand un agent *Guest* envoie le message ;
- l'identificateur et l'url de l'agent destinataire permettant de connaître la destination du message. Ils sont initialisés quand un *proxy Guest* d'un agent envoie le message ;
- les arguments d'un message, qui sont stockés ensemble et distingués par leurs noms respectifs ;
- l'identificateur d'un message, qui est unique et le distingue d'un autre, même si les deux messages ont la même étiquette. Il est généré automatiquement lorsqu'un *proxy Guest* envoie le message ;

- le *hop* d'un message, qui est utilisé dans un but interne : empêcher un message de rester trop longtemps sur le réseau. Il est initialisé à 0 et est incrémenté de 1 chaque fois qu'un *proxy Guest* tente d'envoyer le message. Si la valeur de hop dépasse une valeur prédéfinie (`HOPMAX`), le message est détruit au lieu d'être envoyé ;
- la date permettant de savoir quand le message a été créé et envoyé. Ce champ est initialisé automatiquement lorsque le *proxy Guest* envoie le message ;

Pour personnaliser selon les besoins spécifiques, il est possible de créer des classes héritant de cette classe. Ci-dessous est la description de trois types de messages spécifiques dans *Guest* :

- `ForwardGuestMessage` : Par exemple, quand un agent reçoit un message qui n'est pas pour lui, il essaie de le transférer à sa destination en encapsulant ce message dans un `ForwardGuestMessage` puis en envoyant ce nouveau message.
- `NativeGuestMessage` : Les messages internes de la plate-forme *Guest* sont des instances de la classe `NativeGuestMessage` et ne peuvent pas être observés ou interceptés comme les messages normaux. Les messages natifs sont pris en compte par les méthodes `handleSyncNativeGuestMessage` et `handleAsyncNativeGuestMessage` qui sont déclarées finales dans la classe `GuestAgent`.
- `FutureGuestMessage` : Le principe d'implémentation du type de communication « réponse future » est comme suit : un agent envoie un message de ce type en créant une instance de cette classe puis en envoyant ce message par la méthode `sendAsyncGuestMessage()` comme un message asynchrone. Une fois ce message arrive à la destination, il est traité par la méthode `handleFutureGuestMessage()`. Déclarée finale, cette méthode donne le message à l'agent (en appelant la méthode `handleSyncGuestMessage()`), puis encapsule la réponse dans un autre FutureGuestMessage et le renvoie au premier agent. Ce message est de type `ReplyMessage` et sera de ce fait traité dans la méthode `handleFutureReplyGuestMessage()` du premier agent.

7.2.3.2 Gestion des messages

La gestion des messages permet à l'agent de traiter les messages (synchrones et asynchrones) qu'il reçoit. C'est notamment grâce à la gestion de ces messages que l'agent peut interagir avec son environnement et d'autres agents *Guest* ou modifier son comportement en conséquence.

Guest sépare les messages en deux types de base : message asynchrone et synchrone. Le traitement de ces deux types de messages est différent. Pour la communication asynchrone, il n'est pas demandé de retourner une réponse. Les messages asynchrones reçus sont traités par la méthode `handleAllAsyncGuestMessage()`. De façon similaire, les messages synchrones sont traités par la méthode `handleAllSyncGuestMessage()`. Cependant, la communication synchrone demande une réponse à retourner à l'agent émetteur.

Les deux méthodes ci-dessus sont les deux points centraux pour accueillir des messages arrivés. Elles classifient les messages par leur type et les distribuent aux gestionnaires correspondants :

- les messages natifs (messages internes de *Guest*) sont traités par les méthodes `handleSyncNativeGuestMessage()` et `handleAsyncNativeGuestMessage()` ;
- les messages à transférer sont traités par la méthode `handleForwardGuestMessage()` de manière asynchrone ;
- les messages de type « future réponse » sont traités de façon asynchrone par la méthode `handleFutureGuestMessage()` ;
- les réponse sont traités par la méthode `handleFutureReplyGuestMessage()` ;
- les messages utilisateurs sont traités par les méthodes `handleSyncGuestMessage()` et `handleAsyncGuestMessage()`.

7.2.4 Micro-plateforme CORBAHost

Afin de permettre aux agents *Guest* de fonctionner indépendamment d'autres plates-formes existantes et de justifier notre approche interface, nous avons construit la micro-plateforme CORBAHost. En fait, cette plate-forme n'est composée que de deux interfaces : CHServer et CHAgent. La définition de ces deux interfaces en IDL (*Interface Definition Language*) de CORBA est la suivante :

```
interface CHServer {
  string createCHAgent(in string className, in string agentName,
   in ValueBase obj) raises (CHException);
  string getCHAgentIOR(in string agentName);
  oneway void removeAgent(in string agentName);
};

interface CHAgent  {
  oneway void live();
  oneway void handleAsyncCHMessage(in ValueBase message);
  ValueBase handleSyncCHMessage(in ValueBase message) raises (CHException);
  oneway void remove();
};
```

La première interface spécifie un serveur d'agents qui fournit trois opérations : la création et la destruction d'un agent, ainsi que la récupération de 'adresse d'un agent . La deuxième interface définit un agent avec un thread d'exécution (`méthode live()`) et la capacité de traiter les messages synchrones et asynchrones. En plus de ces deux interfaces, une classe spécifique, `CHProxy` ajoute le concept de *proxy* d'agent à cette plate-forme.

Cette plate-forme est basée sur l'implémentation de CORBA fournie dans JDK et fournit une plate-forme multi-agents simple, légère (la taille de cette plate-forme n'est que 25kb) mais robuste et de haute performance.

7.3 Mise en œuvre au niveau du code d'agent

7.3.1 Conception et implémentation du modèle de *plugins*

Un *plugin* est un composant pouvant être ajouté ou retiré de l'agent *Guest* selon le besoin de ce dernier. Un *plugin* peut :

- être informé avant le changement d'état d'un agent *Guest*, voire empêcher ce changement dans certains cas (migration ou désactivation) ;
- intercepter un message de communication, quoi que ce soit synchrone ou asynchrone ;

- fournir de nouvelles fonctionnalités sous forme de méthodes.

Dans sa conception, un *plugin* est la combinaison de deux composants :

- un *handler* qui s'occupe des deux premiers rôles : intercepter les messages et / ou observer le changement d'état d'un agent ;
- une librairie offrant de nouvelles fonctionnalités.

Un *plugin* doit hériter de la classe `GuestPlugin` ou d'une de ses sous-classes. Un *handler* d'un *plugin* doit hériter de la classe `GuestHandler` ou d'une de ses sous-classes et contient certaines méthodes « *call-back* ».

7.3.1.1 Observation et interception de changements de l'état d'un agent

Pour intercepter les changements d'état d'un agent, le *handler* du *plugin* doit implanter l'interface suivante :

```
public interface IStatusHandler {
    public boolean vetoMigration(GuestURL newDest);
    public boolean vetoSave ();
    public boolean vetoRemove();
}
```

Pour observer la migration d'un agent, le *handler* du *plugin* doit implanter l'interface `IMigrationListener`.

Pour observer la désactivation / activation d'un agent, le *handler* du *plugin* doit implanter l'interface `ISaveListener`.

Pour être informé qu'un agent va se détruire, le *handler* du *plugin* doit implanter l'interface `IRemovalListener`.

Pour observer le clonage d'un agent, le *handler* du *plugin* doit implanter l'interface `ICloneListener`.

Pour observer les chargements et les déchargements des autres *plugins*, le *handler* du *plugin* doit implanter l'interface suivante :

```
public interface IPluginListener {
    public void postLoadPlugin(GuestPluginInfo gpi);
    public void postUnloadPlugin(GuestPluginInfo gpi);
}
```

7.3.1.2 Observation et interception de la communication par *plugin*

Pour observer / intercepter la réception des messages synchrones, le *handler* du *plugin* doit implanter l'interface suivante :

```
public interface ISyncGuestMessageHandler {
    GuestMessage handleSyncGuestMessage(GuestMessage guestMsg);
}
```

Cette méthode `handleSyncGuestMessage()` est appelée quand un message synchrone arrive à l'agent auquel ce *plugin* est attaché. Cette méthode peut :

- retourner immédiatement la réponse à ce message : dans ce cas, elle doit créer une exception de type `ReturnPluginException` contenant la réponse, puis lancer cette exception. Dès lors, les *plugins* suivants dans la chaîne des *plugins* attachés à l'agent et l'agent lui-même ne recevront plus le message ;
- retourner un objet de type `GuestMessage` ; cet objet pouvant être le message originel ou un message modifié. Le message de retour sera utilisé par le *plugin* suivant dans la chaîne des *plugins* attachés à l'agent ou par la méthode `handleSyncGuestMessage()` de l'agent lorsque tous les *plugins* auront été consultés.

Pour intercepter des messages asynchrones, le *handler* du *plugin* doit implanter l'interface suivante :

```
public interface IAsyncGuestMessageHandler {
    GuestMessage handleAsyncGuestMessage(GuestMessage guestMsg);
}
```

La méthode `handleAsyncGuestMessage()` est appelée quand un message asynchrone arrive à l'agent auquel ce *plugin* est attaché. Cette méthode peut :

- retourner immédiatement une réponse : dans ce cas, elle doit lancer une exception de type ReturnPluginException. Dès lors, les *plugins* suivants dans la chaîne des *plugins* attachés à l'agent et l'agent lui-même ne recevront plus le message ;
- retourner un objet de type GuestMessage, cet objet peut être le message originel ou un message modifié. Le message de retour sera utilisé par le *plugin* suivant dans la chaîne des *plugins* attachés à l'agent ou par la méthode handleAsyncGuestMessage de l'agent lorsque tous les *plugins* auront été consultés.

Pour observer / intercepter l'envoi de messages synchrones, le *handler* du *plugin* doit implanter l'interface suivante :

```
public interface ISyncGuestMessageSender {
    public GuestMessage sendSyncGuestMessage(GuestMessage guestMsg,
           GuestId destId, GuestURL destURL);
}
```

La méthode `sendSyncGuestMessage()` est appelée avant l'envoie d'un message synchrone de l'agent auquel ce *plugin* est attaché. Cette méthode peut :

- annuler immédiatement l'envoi de ce message : dans ce cas, elle doit lancer une exception de type `ReturnPluginException`. Dès lors, les *plugins* suivants dans la chaîne des *plugins* attachés à l'agent ne traiteront plus le message ;
- retourner un objet de type `GuestMessage` ; cet objet pouvant être le message original ou un message modifié. Le message de retour sera utilisé par le *plugin* suivant dans la chaîne des *plugins* attachés à l'agent avant d'être envoyé à la destination lorsque tous les *plugins* auront été consultés.

Pour intercepter l'envoi de messages asynchrones, le processus est similaire, sauf que le *handler* du *plugin* doit implanter l'interface suivante :

```
public interface IAsyncGuestMessageSender {
    public GuestMessage sendAsyncGuestMessage(GuestMessage guestMsg,
           GuestId destId, GuestURL destURL) ;
}
```

7.3.1.3 Enrichir les fonctionnalités de l'agent par *plugins*

Pour ajouter une nouvelle fonctionnalité à un agent par l'approche *plugin*, le *plugin* correspondant doit implémenter cette fonctionnalité dans une méthode publique. De plus, une description complète de toutes ces fonctionnalités doit être fournie à l'agent lors du chargement de ce *plugin*.

7.3.1.4 Gestion des *plugins*

Toutes les tâches de gestion des *plugins* sont gérées par la classe `PluginManager`.

```
public final class PluginManager implements Serializable {
    public boolean loadPluginConfigFile() ;
    public boolean loadPlugin() ;
    public boolean unloadPlugin() ;
    public Object invokeOnPlugin();
    public Object invokeOnPlugin();
    public Vector listPluginGuestInfos() ;
    public boolean isLoadedPlugin() ;
    public GuestPlugin getGuestPlugin() ;
}
```

La méthode `loadPlugin()` permet de charger un *plugin* et d'obtenir une référence de celui-ci pour y accéder ultérieurement :

Il faut noter qu'un agent peut charger un même type de *plugin* plusieurs fois mais avec différents paramètres d'initialisation. Autrement dit, un agent peut avoir différentes instances d'un même type de *plugin*. Par exemple, un agent peut s'inscrire à deux différentes `Régions` en utilisant deux instances du `RegionPlugin`.

Si ce *plugin* est chargé, la méthode `loadPlugin()` retourne une référence de ce *plugin*, sinon une exception de type `PluginException` sera lancée.

Il est courant qu'un *plugin* soit chargé durant la phase d'initialisation d'un agent, c'est-à-dire dans la méthode init() de l'agent. Toutefois, vous pouvez charger un *plugin* à n'importe quel moment de la vie de l'agent. Le morceau de code suivant est un exemple du chargement d'un *plugin* :

```
try{
  regionViewerPlugin = (RegionViewerPlugin) loadPlugin(
            "ca.crim.guest.ext.plugin.RegionViewerPlugin",
            "Region;VY://rhea.crim.ca:9000");
}
catch (PluginException e) {
  GuestSystem.displayException(e);
}
```

Pour déployer des *plugins*, il est possible d'utiliser un fichier de configuration. Le codage de ce fichier de configuration peut être sous forme XML, *.properties* de Java ou

n'importe quel format texte. C'est le `PluginLoader` qui est responsable de lire ces fichiers et de les interpréter. Au cas où l'on veut ajouter de nouveaux types d'information à ces fichiers, une nouvelle version de `PluginLoader` sera nécessaire. Grâce au modèle des composants dynamiques, la substitution de ce `PluginLoader` est rendue possible. Par conséquent, le contenu des fichiers de configuration d'un *plugin* est ouvert et extensible.

Pour décharger un *plugin* ou utiliser ses services, il faut tout d'abord obtenir une référence de celui-ci, ce qui est possible à partir de son nom de classe en utilisant la méthode `getGuestPlugin()`. Au cas où il y a plusieurs instances de cette classe, on obtient le premier *plugin* chargé. Le cas échéant, la réponse est « null ».

Pour décharger un *plugin*, il est demandé de disposer d'abord d'une référence de ce *plugin* (laquelle peut être obtenue en utilisant la méthode `getGuestPlugin` présentée ci-dessus). Ensuite, l'appel de la méthode `unloadPlugin()` permet de décharger ce *plugin*.

Après avoir été chargé, toutes les méthodes publiques d'un *plugin* seront exposées sous forme de services offerts et peuvent être invoquées par les appels sur une de ses références.

7.3.1.5 RegionPlugin et RegionViewerPlugin

Le `RegionPlugin` est un *plugin* permettant à l'agent de communiquer de façon automatique avec une `GuestRegion`.

Une `GuestRegion` est un agent *Guest* non mobile. Les agents *Guest* mobiles peuvent s'y inscrire. À chaque fois qu'ils migrent, ils indiquent leur nouvelle adresse à la GuestRegion. Les autres agents du système peuvent alors demander à la `GuestRegion` l'adresse courante des agents inscrits.

Ce processus peut se faire de façon automatique en ajoutant à un agent le `RegionPlugin`. Celui-ci va, à son chargement, inscrire l'agent à une `GuestRegion`, prévenir la `GuestRegion` des changements d'adresse et des états de l'agent, puis se désinscrire de la `GuestRegion` à son déchargement.

L'agent `RegionViewer` est une extension de l'agent `GuestRegion` avec une visualisation graphique. De plus, elle permet de manipuler graphiquement les agents (par exemple, lors de tests).

7.3.2 Parseur et interpréteur CATN

Le rôle du parseur CATN est d'obtenir une spécification en formalisme CATN en entrée et de fournir en sortie une représentation interne de celle-ci pour l'interprétation ultérieure. Afin de supporter plusieurs parseur, nous fournissons ici une interface appelée `ICATNParser`, prenant le CATN source d'un fichier ou d'une adresse Web :

```
public interface ICATNParser {
    public CATN parseCATN(String urlStr);
    public CATN parseCATN(URL url);
    public CATN parseCATN(InputStream is);
    public ICATNWriter getCATNWriter();
}
```

Un parseur par défaut a été implémenté dans la classe `DefaultCATNParser`. De plus, nous fournissons la classe `ParserFactory` permettant d'initialiser un parseur à partir du nom de la classe d'implémentation. Ce parseur utilise différents objets pour représenter les éléments correspondants d'un CATN :

- **CATN** : les objets de ce type encapsule un CATN entier ;
- **Constant** : ce type est pour encapsuler une constance dans le formalisme CATN ;
- **Data** : ce type est pour encapsuler un type de data dans le formalisme CATN ;
- **Register** : ce type est pour encapsuler un registre dans le formalisme CATN ;
- **State** : ce type est pour encapsuler un état dans le formalisme CATN ;
- **Arc** : il y a de différents types d'arcs : ArcTerminal, ArcPA, ArcSystem, ArcCATN, ArcInteraction, ArcMRListener, ArcCATNMS, ArcCATNMR ;
- **Transition** : il y a TransitionSingle, TransitionOR, TransitionAND, Transition-XOR, qui correspondent aux différents types de transition dans le formalisme CATN ;
- **Constraint** : représente une contrainte pour un arc.

Nous fournissons également une implémentation de référence de l'interpréteur CATN dans la classe `CATNManager`. Chaque action sera interprétée dans un contexte séparé, similaire au concept contexte d'exécution des programmes :

- **CATNContext** : avant d'exécuter un CATN ou un arc de type ArcCATN, l'interpréteur initialise les registres et les rend disponibles pour l'exécution de ce CATN ;

- **ArcContext** : encapsule le contexte d'exécution d'un arc (de n'importe quel type), tel que la contrainte de début de cet arc, l'objet cible fournissant des actions atomiques ;
- **TransitionContext** : encapsule le contexte d'exécution d'une transition, tels que l'état de départ et l'état destinataire.

7.4 Mise en œuvre au niveau du système

7.4.1 Implémentation des modèles d'agent récursif

7.4.1.1 Agent récursif centralisé

Le modèle d'agent centralisé est implémenté dans le noyau de la plate-forme *Guest* en introduisant l'adresse récursive et en distinguant le traitement de demande de communication et de migration en fonction de cette adresse.

Une adresse récursive se compose de deux parties : l'URL de standard et l'extension. Tandis que l'URL de standard indique le serveur réel sur lequel l'agent se trouve, l'extension est le chemin complet pour accéder à l'agent à partir de l'agent racine, ce qui est similaire au concept de chemin dans le système de gestion des fichiers. La combinaison de ces deux parties est sous la forme « *URL normal$extension* ». Par exemple, à partir de l'adresse d'un agent « *AG ://localhost :4000$grandpère/père* », on peut dire que cet agent est le fils de l'agent « *père* », qui est lui-même le fils de l'agent « *grand-père* » et que l'ensemble de la hiérarchie se trouve sur le serveur « *localhost* ».

Les opérations concernant les agents fils sont gérées par la classe `HierarchyManager`. Ces opérations sont la migration interne de la hiérarchie, l'ajout, l'enlèvement d'un agent de la hiérarchie et la communication entre eux. Grâce à cette gestion, un agent fils qui se connecte à son agent père croit qu'il est hébergé sur un serveur réel. La communication entre l'agent père et ses fils se faisant par les appels directs des méthodes, la performance de la communication est largement optimisée.

7.4.1.2 Agent récursif distribué

La mise en œuvre du modèle d'agent récursif distribué se base entièrement sur le modèle de *plugins* présenté dans le chapitre 5. Plus concrètement, ce modèle est réalisé par l'intermédiaire d'un handler particulier de message (`IRecursionHandler()`) qui est chargé en même temps que le *plugin* de hiérarchie distribué (RecursionPlugin). Le « *broadcast* » et « *multicast* » des messages sont assurés par `InteractionPlugin`.

En principe, `IRecursionHandler` intercepte tous les messages destinés à l'agent l'ayant chargé et le redirige sur son père direct dans la hiérarchie, en demandant une autorisation de traitement. Si le père direct accepte, il le transmet de-même à son père, et ce récursivement jusqu'à l'agent prenant la décision finale : soit le sommet de la hiérarchie, soit le premier agent qui refuse d'autoriser le traitement. En cas contraire, c'est-à-dire la décision est un refus, le message est détruit.

7.4.1.2.1 InteractionPlugin Le *plugin* d'interaction permet de gérer des « communautés d'agents en interaction » de façon simple. Nous définissons une interaction entre agents, dans *Guest*, comme un ensemble d'échanges de messages entre agents, se rapportant à un objectif donné. Le rôle de ce *plugin* dans ce cadre est de permettre à un agent d'initier simplement une interaction, de lui donner un identifiant partagé par tous les partenaires, de permettre à d'autres agents de participer à l'interaction en cours, et enfin d'offrir un moyen à chacun des participants de parler aux autres participants, et uniquement à eux. Le *plugin* se charge entièrement du mécanisme de référenciation (nommage) de l'interaction, du suivi des participants et de l'émission en diffusion de messages à tous les participants.

L'API est la suivante :

- `broadcastMessageToRegion()` : cette méthode crée une interaction en envoyant un message donné, sur tous les agents connus de la région (le RegionPlugin est indispensable pour que le *plugin* `Interaction` fonctionne), avec un identifiant d'interaction donné. Ce faisant, l'agent émetteur est enregistré comme participant à l'interaction par les *plugins* d'interactions de tous les agents recevant ce message ;
- `broadcastMessageToCommunity()` : cette méthode envoie un message donné à tous

les agents connus, par l'agent émetteur, pour une interaction donnée. Ce faisant, l'agent émettant ce message est enregistré comme participant à l'interaction par les *plugins* d'interactions de tous les agents recevant ce message (il n'est pas enregistré une deuxième fois s'il l'était déjà) ;

- `sendInteractionMessage()` : cette méthode envoie un message donné à un agent donné, sous un nom d'interaction donnée ;
- `boolean contains()` : renvoie un booléen exprimant si l'interaction d'identifiant donné est connue de l'agent ou non ;
- `registeredAgents()` : renvoie un vecteur contenant les identificateurs des agents connus de cet agent pour l'interaction « interaction ». Ce vecteur est une copie des données du *plugin* ; une modification de ce vecteur n'aurait de ce fait aucune répercussion sur le fonctionnement du *plugin* ;
- `forget()` : purge le *plugin* de l'interaction dont le nom est passé en paramètre. Renvoie *true* si l'interaction existait, *false* sinon.

7.4.1.2.2 RecursionPlugin Ce *plugin* est responsable de la gestion des hiérarchies distribuées sur plusieurs serveurs. Il ne peut être chargé que par un agent implantant l'interface `IRecursionHandler`. Il permet de représenter des hiérarchies d'agents, de leur création à leur destruction. Une hiérarchie d'agents est un arbre, dont la racine est appelée « père de la hiérarchie » et dont les liens sont de type « père -> fils ». Lorsqu'un agent devient le fils d'un autre agent, il donne à ce dernier le contrôle sur les messages qu'il est autorisé à recevoir. En d'autres termes, tout message reçu par un fils devra être approuvé par son père avant que le fils ne puisse y avoir accès. Lorsque le message est approuvé, le fils est libre de répondre comme il le souhaite. Dans le cas d'une hiérarchie à plusieurs niveaux (un fils, un père et un grand père par exemple), il suffit que l'un des ancêtres de l'agent recevant le message s'oppose à son traitement pour que l'agent ne le reçoive pas. Inversement, il faut que tous les ancêtres autorisent le traitement du message pour que celui-ci puisse avoir lieu.

La création d'une hiérarchie se fait dynamiquement, à l'exécution du système, en utilisant l'API offerte par le RecursionPlugin :

- `registerTo()` : demande au *plugin* d'inscrire l'agent au père d'identifiant « père ». L'inscription étant asynchrone, cette fonction ne renvoie rien. La réponse sera reçue via l'interface `IRecursionHandler` qu'un agent est obligé d'implanter pour pouvoir charger le `RecursionPlugin` ;
- `unregisterFromParent()` : demande au *plugin* de quitter le père actuel. Un agent ne pouvant avoir qu'un seul père, il est inutile de passer celui-ci en paramètre. À nouveau, la demande est asynchrone ;
- `childEnumeration()` : renvoie la liste des identificateurs des fils de l'agent ;
- `getFather()` : renvoie un `GuestAgentInfo` décrivant le père de l'agent ;
- `isDirectChild()` : renvoie un « *boolean* » indiquant si l'agent fils est ou non le fils *direct* de l'agent ;
- `isChild()` : renvoie un « *boolean* » indiquant si l'agent fils est ou non le fils *direct ou indirect (petit-fils, arrière-petit-fils, etc)* de l'agent ;
- `sendAsyncToAllDirectChild()` : transmet un message en mode asynchrone à tous les fils directs de l'agent.

En résumé, `RecursionPlugin` offre les fonctions suivantes :

- s'enregistrer auprès d'un père ;
- quitter son père ;
- obtenir la liste des fils inscrits à un agent donné ;
- déterminer si un agent donné est le fils direct ou indirect d'un autre ;
- propager un message par diffusion à tous ses fils directs.

L'interface `IRecursionHandler` offre à l'agent l'API suivante :

- `Register()` : si un agent d'identifiant « identifier » veut s'enregistrer, l'agent doit implanter la politique d'acceptation ou de refus ;
- `Unregister()` : si un agent d'identifiant « identifier » veut se désenregistrer, l'agent doit implanter la politique d'acceptation ou de refus ;
- `RequestAuthorization()` : un fils a reçu un message et demande à l'agent père si celui-ci autorise ou non la lecture de ce message ;
- `RegisterReplyReceived()` : une réponse à une demande d'inscription a été reçue, l'agent ayant demandé à s'inscrire en est prévenu ;

- `UnregisterReplyReceived()` : une réponse à une demande de désinscription a été reçue, l'agent ayant demandé à s'inscrire en est prévenu.

7.4.1.3 Intégration de deux types

Suivant le modèle théorique, dans notre implémentation, les deux types d'agent récursif peuvent fonctionner non seulement en parallèle mais aussi en combinaison. Après une opération de migration, si l'agent père et fils se trouvent sur le même serveur, ils se combinent automatiquement en une hiérarchie centralisée. En revanche, si après une opération de migration, l'agent fils n'est plus sur le même serveur que l'agent père, les deux agents forment alors une hiérarchie distribuée.

L'implémentation de ces transformations est rendue possible en utilisant un plugin observant l'état des agents fils. Après qu'un événement de migration d'un agent membre d'une hiérarchie s'est retiré, une vérification des positions des agents pères-fils sera faite et le regroupement nécessaire se produira.

7.4.2 Réalisation du modèle MetaCATN

7.4.2.1 CATNCoordinator

Le `CATNCoordinator` est chargé de deux tâches très importantes : contrôler l'interpréteur CATN et propager le changement dans le SMA. Pour la première tâche, le CATNCoordinator envoie des messages de contrôle à l'interpréteur CATN. Rappelant que l'interpréteur CATN est lui-même un CATN, ces messages d'échange peuvent être vus comme des arcs d'interaction entre les CATNs et le CATNCoordinator est, aux yeux de l'interpréteur CATN, un autre CATN. Les messages envoyés à l'interpréteur CATN sont les suivants :

- demande de modifier le CATN courant. Dans ce cas, l'interpréteur fait le changement demandé seulement si cette modification n'affecte que des arcs qui ne sont pas encore interprétés ;
- demande de remplacer l'interpréteur CATN courant. Quand un interpréteur CATN reçoit une telle demande, il continue à interpréter l'arc courant. Dès la terminaison

de l'exécution de cet arc, l'interpréteur sauvegarde tous les états d'exécution afin de les transférer au nouvel interpréteur.

Pour la deuxième tâche, le `CATNCoordinator` doit analyser la description du CATN qui est en train d'être interprété. Si l'arc à modifier concerne des interactions avec les autres CATNs, le CATNCoordinator enverra un message indiquant le changement à tous ces CATNs. Ces messages arrivent tout d'abord au `CATNCoordinator` des agents impliqués et ceux-ci réalisent les changements locaux nécessaires.

Un interpréteur CATN est lui-même spécifié par le formalisme CATN, dès lors on peut utiliser le même interpréteur CATN pour l'exécuter (voir la section 7.3.2).

7.4.2.2 Intégration à la plate-forme *Guest*

Il faut noter que jusqu'à maintenant, la réalisation du modèle MetaCATN est entièrement indépendante de la plate-forme *Guest*, ce qui permet l'utilisation de ce modèle sur d'autres plates-formes que *Guest*. Toutefois, l'intégration de MetaCATN avec la plate-forme *Guest* reste relativement directe :

- Un nouveau type de message est fourni : `GuestCATNMessage`. Les messages de ce type sont réservés uniquement aux CATNCoordinators, afin de les distinguer des autres types de messages. Chaque message de ce type contient le nom du CATN source et celui du CATN destinataire.
- Un interpréteur peut être implémenté sous forme d'un *plugin*. Nous avons fourni un tel *plugin*, qui est appelé `DefaultCATNManagerPlugin` et offre les actions atomiques suivantes : « executeCATN », « waitingMessage » et « doNothing ».
- Un type particulier d'agent *Guest* est introduit : `CATNAgent`. Ce type d'agent peut traiter les messages `GuestCATNMessage` et est chargé par défaut avec un interpréteur CATN.

CHAPITRE 8

CONCLUSION

8.1 Synthèse des travaux effectués

Cette thèse a pour objectif d'étudier l'adaptation dynamique du contrôle des systèmes multi-agents et de la modéliser afin de faciliter la construction de systèmes d'agents adaptatifs.

Les résultats obtenus n'ont été rendus possibles qu'en suivant une approche adaptée au problème. Notre approche est de distinguer entre l'adaptation dynamique du code fonctionnel et celle du contrôle dans les systèmes multi-agents pour ensuite nous concentrer sur la dernière. Cette distinction ouvre la voie à la généralisation de l'adaptation dynamique des systèmes d'agents et à une méthodologie de développement de ceux-ci.

Notre approche est pratique, se basant notamment sur l'étude comparative des plates-formes d'agents existantes. Cette approche nous a conduit à une solution ouverte permettant aux agents de s'adapter dynamiquement à diverses plates-formes d'exécution, qu'elles soient existantes ou à venir.

Nous avons également suivi une voie théorique : nous avons identifié trois niveaux d'adaptation dynamique pour un système multi-agents : niveau plate-forme, niveau agent et niveau inter-agent, ceux-xi correspondent aux trois niveaux de l'architecture des systèmes multi-agents. Cette classification couvre toutes les formes d'adaptation possibles et permet de les modéliser de manière progressive : l'adaptation d'un niveau plus élevé peut être réalisée en se basant sur l'adaptation aux niveaux plus bas.

8.1.1 Apports théoriques

Notre contribution théorique est tout d'abord *l'uniformisation des plates-formes multi-agents* en fournissant une interface commune aux services de base de celles-ci et un contexte d'exécution uniforme pour les agents. Les résultats obtenus apportent un avancement signficatif dans le domaine des systèmes multi-agents en *affranchissant des limites*

concernant l'incompatibilité des principales plates-formes d'agents actuelles. Le développeur des applications multi-agents est désormais libéré des soucis de l'hétérogénéité des plates-formes et il peut se concentrer sur sa propre application.

Nous avons également proposé *une combinaison adaptative de deux modèles d'agents récursifs* : centralisé et distribué. Ces deux modèles peuvent être utilisés en parallèle : la transformation d'un modèle en l'autre peut être automatique, en fonction des positions des agents membres, et elle ne demande aucun effort supplémentaire de codage. Dès lors, *l'organisation des systèmes multi-agents devient adaptative à sa distribution géographique*, ce qui est essentiel pour obtenir des organisations multi-agents de plus en plus complexes et massivement distribuées.

Finalement, nous avons introduit *MetaCATN, un méta-modèle d'interaction des agents* dans un système multi-agents. Ce méta-modèle, basé sur le formalisme CATN et composé de deux niveaux, permet de spécifier les interactions entre agents et de synchroniser la modification dynamique de ces interactions, tout en assurant l'exécution continuelle du système global. *Seule une telle forme d'évolution véritablement multi-agent permettra à aboutir à une autonomie réelle des sociétés d'agents* survivant dans des environnements dynamiques tel qu'Internet.

8.1.2 Apports pratiques

Au-delà des aspects purement théoriques, nous avons également développé *un paradigme de programmation des agents en combinant le code compilé sous forme de plugins et le code interprété spécifiant par le formalisme CATN.* Ces agents peuvent alors évoluer au cours de leur vie, tout en étant capables d'étendre leurs fonctionnalités et de modifier leurs comportements de manière dynamique, sans demander l'arrêt de l'exécution des agents. De cette façon, *le développement des agents ne doit pas s'arrêter au moment qu'ils commencent leur vie, mais peut continuer jusqu'à la fin de leur existence.* Cette évolution continuelle est une condition indispensable de toutes formes d'adaptation.

Tous les modèles théoriques et le paradigme proposés ont été complètement concrétisés dans notre plate-forme multi-agents *Guest.* Dès lors, cette plate-forme peut fonctionner au-dessus des plates-formes d'agents existantes et *fournir un cadre de travail opérationnel*

pour le développement de nouvelles applications multi-agents ou l'extension d'applications existantes.

8.2 Problèmes ouverts et perspectives futures

Au vu de nos travaux de recherche, il est apparu encore plus manifeste que l'adaptation dynamique reste un vaste et difficile problème. En nous concentrant sur l'adaptation dynamique du contrôle, nous avons pu apporter des solutions à l'adaptation de la partie du contrôle dans les systèmes multi-agents.

Cependant, il demeure encore des problèmes à résoudre, tant théoriques que pratiques. Au niveau théorique, la validation et la vérification de changements dans le système d'agents, ainsi que la correction de la combinaison des *plugins* sont nécessaires mais elles ne sont résolues que partiellement, considérant que celles-ci sont encore le sujet de recherche de nombreux travaux en cours. De notre point de vue, l'approche « *model checking* » [Hallal *et al.*, 2001] semble prometteuse [Nourchene, 2003]. Une autre perspective de recherche dans l'adaptation dynamique des agents est également très intéressante à modéliser : l'adaptation par émergence [Deguer, 2005].

Au niveau pratique, la plate-forme *Guest* n'est pour l'instant qu'une implémentation de référence de nos modèles. Cependant, elle a été déjà utilisée dans un prototype d'application industrielle (projet IBAUTS) et a ainsi démontré son utilité. En notant que son code a été libéré de tout *copyright*, cette plate-forme *open source* ayant une architecture ouverte peut espérer obtenir la contribution de la communauté d'agents pour devenir plus robuste, plus performante et riche de nouvelles fonctionnalités.

BIBLIOGRAPHIE

[Agha *et al.*, 2001] Gul Agha, Nadeem Jamali, and Carlos A. Varela. Agent naming and coordination : Actor based models and infrastructures. In *in A. Omicini et al. (editors), Coordination of Internet Agents : Models, Technologies and Applications, Chap. 9, Springer Verlag*, 2001.

[Agha, 1986] Gul Agha. Actors : A model of concurrent computation in distributed systems. Technical report, Massachusetts Institute of Technology, 1986.

[Aglets, 1998] Aglets workbench by ibm tokyo rl. http ://www.trl.ibm.co.jp/aglets/, 1998.

[Aknine *et al.*, 1998] Samir Aknine, Suzanne Pinson, and Manuel Zacklad. Un système d'agents récursifs pour l'aide au travail coopératif. In Hermes, editor, *Journées Francophones de l'Intelligence Artificielle Distribuée et des Systèmes Multi-Agents*, pages 67–80, Pont-à-Mousson, France, 1998.

[Ando, 2006] Yasushi Ando. Performance of pheromone model for predicting traffic congestion. In *Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, Hakodate, Japan, 2006.

[Aridor and Oshima, 1998] Y. Aridor and M. Oshima. Infrastructure for mobile agents : Requirements and design. In *Mobile Agent 98*, 1998.

[Ashri and Luck, 2001] Ronald Ashri and Michael Luck. Towards a layered approach for agent infrastructure : the right tools for the right job. In *Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001.

[Ayed *et al.*, 2003] Dhouha Ayed, Chantal Taconet, and Guy Bernard. Context-aware deployment of multi-component application. In *Proceedings of the 5th Generative Programming and Component Engineering (GPCE03)*, 2003.

[Ayed *et al.*, 2004] Dhouha Ayed, Chantal Taconet, and Guy Bernard. Architecture à base de composants pour le déploiement adaptatif des applications multi-composants. In *Journées Francophones de Composants*, Lille, France, Mars 2004.

[Barber and Martin, 2001] K.S Barber and C.E. Martin. Dynamic adaptive autonomy in multiagent systems : representation and justification. *International Journal of Pattern Recognition and Artificial Intelligence*, 15(3) :405–433, 2001.

[Baumann and Radouniklis, 1997] Joachim Baumann and Nikolaos Radouniklis. Agent Groups in Mobile Agent Systems. In *Distributed Applications and Interoperable Systems (DAIS'97)*, 1997.

[Beck, 1999] Kent Beck. Embracing change with extreme programming. *Computer*, 32 :70–77, 1999.

[Bellavista and Magedanz, 2001] Paolo Bellavista and Thomas Magedanz. *Coordination of Internet Agents : Models, Technologies and Applications*, chapter Middleware technology : CORBA and Mobile Agents, pages 110–152. Springer Verlag, 2001.

[Bernon *et al.*, 2001] Carole Bernon, Valérie Camps, Marie Pierre Gleizes, and Pierre Glize. La conception de système multi-agents adaptatifs : contraintes et spécifités. In *Atelier de Méthodologie et Environnements pour les Systèmes multi-agents*, Grenoble, France, 2001.

[Bezivin and Blanc, 2002] J. Bezivin and X. Blanc. Mds : vers un important changement de paradigme en génie logiciel. In *Développeur Référence*, Juillet 2002.

[Bradshaw, 1995] Jeffrey M. Bradshaw. KAoS : A Generic Agent Architecture for Aerospace Applications. In Tim Finin and James Mayfield, editors, *Proceedings of the CIKM '95 Workshop on Intelligent Information Agents*, Baltimore, Maryland, 1995.

[Brazier *et al.*, 2002] F.M.T Brazier, B.J. Overeinder, M. vanSteen, and N.J.E. Wijngaards. Agent factory : Generative migration of mobile agents in heterogeneous environments. In *ACM Symposium on Applied Computing*, Madrid, Spain, 2002.

[Briot and Demazeau, 2001] J. P. Briot and Y. Demazeau. *Principes et architecture des systèmes multi-agents.* Hermes, 2001.

[Briot *et al.*, 2002] J.P. Briot, Z. Guessoum, S. Charpentier, S. Aknine, O. Marin, and P. Sens. Dynamique adaptation of replication strategies for reliable agents. In *Symposium on Adaptive Agents and Multi-Agent Systems (AAMAS-2), AISB'02 Convention*, pages 20–28, London, England, 2002.

[Briot, 1989a] Jean-Pierre Briot. Actalk : a testbed for classifying and designing actor languages in the Smalltalk-80 environment. In Steve Cook, editor, *European Conference on Object-Oriented Programming (ECOOP'89)*, British Computer Society Workshop Series, pages 109–129, Nottingham, U.K., July 1989. Cambridge University Press, United-Kingdom.

[Briot, 1989b] Jean-Pierre Briot. Actalk : une plateforme de modélisation de langages d'acteurs en Smalltalk-80. In *7ème Congrès AFCET Reconnaissance des Formes et Intelligence Artificielle (RFIA'89)*, volume I, pages 147–161, Paris, France, November-December 1989. AFCET.

[Briot, 2000] Jean-Pierre Briot. Actalk : A framework for object-oriented concurrent programming - design and experience. In Jean-Paul Bahsoun, Takanobu Baba, Jean-Pierre Briot, and Akinori Yonezawa, editors, *Object-Oriented Parallel and Distributed Programming*, pages 209–231. Hermès Science Publications, Paris, France, 2000.

[Brooks, 1991] Rodney A. Brooks. Intelligence without representation. In *Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, Australia, 1991.

[Campadello, 2001] Stefano Campadello. Dynamic composition of execution environment for adaptive nomadic applications. *Lecture Notes in Computer Science*, 2164 :73–80, 2001.

[Capera *et al.*, 2003] Davy Capera, Marie Pierre Gleizes, and Pierre Glize. Self-organizing agents for mechanical design. In Serugendo et al. [2004], pages 169–185.

[Chia and Kannapan, 1997] T. Chia and S. Kannapan. Strategically Mobile Agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Proceedings of the First International Conference on Mobile Agents (MA'97)*, volume 1219, pages 149–161, Berlin, Germany, 1997. Springer-Verlag : Heidelberg, Germany.

[CoABS, 2000] Coabs of darpa. http ://coabs.globalinfotek.com/, 2000.

[Concordia, 1999] http ://www.meitca.com/HSL/Projects/Concordia, 1999.

[Consel and Noël, 1996] C. Consel and F. Noël. A general approach for run-time specialization and its application to c. In *POPL'96 : The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg, FL, USA, 1996.

[Corba,] Common object request broker architecture. http ://www.omg.org.

[David and Ledoux, 2002] P.C. David and T. Ledoux. An infrastructure for adaptable middleware. In *LNCS 2519*, Irvine, California, USA, 2002.

[Deguer, 2005] Joris Deguer. A complexity based feature to support emergence in mas. *CEEMAS 04*, 3690(9) :616–619, 2005.

[DeLoach, 1999] S. DeLoach. Multiagent systems engineering a methodology and language for designing agent systems, 1999.

[Demazeau, 1995] Y. Demazeau. From interactions to collective behaviour in agent-based systems. In *The First European Conference on Cognitive Sciences*, pages 117–132, 1995.

[Dillenseger, 1999] Bruno Dillenseger. Towards full agent interoperability. In *Advanced Communications Technologies and Services Workshop*, 1999.

[Durfee and Rosenschein, 1994] E. H. Durfee and J. Rosenschein. Distributed problem solving and multiagent systems : Comparisons and examples. In M. Klein, editor, *Proceedings of the 13th International Workshop on DAI*, pages 94–104, Lake Quinalt, WA, USA, 1994.

[Dury, 2000] Arnaud Dury. *Modélisation des interactions dans les systèmes multi-agents.* PhD thesis, Université Henri Poincaré - Nancy 1, Grenoble, France, 2000.

[Eclipse, 2003] Ide eclipse. http ://www.eclipse.org, 2003.

[Etzioni and Weld, 1995] Oren Etzioni and Daniel S. Weld. Intelligent agents on the internet : Fact, fiction, and forecast. *IEEE Expert*, 10(3) :44–49, 1995.

[Ferber and Gutknecht, 1998] J. Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *The third International Conference on Multi-agent Systems*, pages 128–135, 1998.

[Ferber and Gutknecht, 1999] Jacques Ferber and Olivier Gutknecht. Operational semantics of a role-based agent architecture. In *Proceedings of the 6th International Workshop on Agent Theories, Architectures and Languages.* Springer-Verlag, 1999.

[Ferber, 1989] J. Ferber. Computational reflection in class based object-oriented languages. *ACM SIGPLAN Notices*, 24(10) :317–326, 10 1989.

[Ferber, 1995] Jacques Ferber. *Les Systèmes Multi-Agents, Vers une Intelligence Collective.* InterEditions, 1995.

[FIPA Agent Management, 2001] Fipa - agent management specification. http ://www.fipa.org/, 2001.

[Fipa-os, 2000] Fipa-os agent toolkit. http ://sourceforge.net/projects/fipa-os/, 2000.

[FIPA, 1998] Foundation for intelligent physical agents. http ://www.fipa.org/, 1998.

[Fischer *et al.*, 1995] K. Fischer, J.P. Muller, M. Pischel, and D. Schier. A pragmatic bdi architecture. In Springer-verlag, editor, *Intelligent Agents : Theories, Architectures, and Languages*, pages 203–218, 1995.

[Flores, 1999] R.A. Flores. Towards the standardization of multi agent systems architectures : An overview. In *ACM Crossroads - Special Issue on Intelligence Agents*, volume 5, 1999.

[Franklin and Graser, 1996] S. Franklin and A. Graser. Is it an Agent, or just a Program ? : A Taxonomy for Autonomous Agents. In *Intelligent Agents III. Agent Theories, Architectures and Languages (ATAL'96)*, volume 1193, Berlin, Germany, 1996. Springer-Verlag.

[Frost, 1997] H. R. Frost. Java agent template. http ://cdr.stanford.edu/ABE, 1997.

[Funfrocken, 1998] S. Funfrocken. Transparent migration of java-based mobile agents : Capturing and reestablishing the state of java programs. In *Mobile Agent 98*, 1998.

[Fung and Low, 2003] Kam Hay Fung and Graham Cedric Low. Design notation for dynamic evolution in component based distributed systems. In *EDOC*, pages 302–307. IEEE Computer Society, 2003.

[Gamma *et al.*, 1995] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gasser, 2001] Les Gasser. Mas infrastructure definitions, needs, prospects. In Springer-Verlag, editor, *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, 2001.

[Genesereth, 1997] Michael R. Genesereth. *Software Agent*, chapter An agent-based framework for interoperability. AAAI Press, 1997.

[Giampapa *et al.*, 2000] Joseph Andrew Giampapa, Massimo Paolucci, and Katia Sycara. Agent interoperation across multagent system boundaries. In *Proceedings of the Fourth International Conference on Autonomous Agents (Agents 2000)*. Association for Computing Machinery, 2000.

[Giroux *et al.*, 1994] Sylvain Giroux, Alain Senteni, and Guy Lapalme. Reactalk, du monde réel à des systèmes informatiques ouverts et adaptatifs. In *Reconnaissance des formes et intelligenge artificielle*, volume 2, pages 269–280, Paris, France, 1994.

[Giroux, 1995] Sylvain Giroux. Open reflective agents. In Michael Wooldridge, Jörg P. Müller, and Milind Tambe, editors, *Intelligent Agents II, Agent Theories, Architectures, and Languages, IJCAI '95, Workshop (ATAL)*, volume 1037 of *Lecture Notes in Computer Science*, pages 315–330, Montreal, Canada, 1995. Springer.

[Giroux, 2001] Sylvain Giroux. Personnalisation, mobilité et information géo-referencée. In *Journées francophones pour l'intelligence artificielle distribuée et les systèmes multi-agents*, pages 17–30, Montreal, Canada, 2001. Hermès.

[Grasshopper, 2000] Grasshopper by ikv++. http ://www.grasshopper.de, 2000.

[Gschwind, 2000] T. Gschwind. Comparing object oriented mobile agent systems, 2000.

[Guessoum and Briot, 1999] Zahia Guessoum and Jean-Pierre Briot. From active objects to autonomous agents. *IEEE Concurrency*, 7(3) :68–76, July-September 1999.

[Guessoum, 1996] Z. Guessoum. *Un environnement opérationnel de conception et de réalisation de systèmes multi-agents.* PhD thesis, Université Pièrre et Marie Curie, Paris, 1996.

[Guessoum, 2000] Z. Guessoum. Systèmes multiagents adaptatifs. Technical report, Laboratoire d'informatique de Paris 6, 2000.

[Guessoum, 2004] Jahia Guessoum. Adaptive agents and multiagent systems. *IEEE distributed systems online*, 5(7), 2004.

[Guilfoyle, 1995] C. Guilfoyle. Vendors of agent technology. In *UNICOM Seminar on Intelligent Agents and their Business Applications*, pages 135–142, 1995.

[Guillemet *et al.*, 1999] Alexandre Guillemet, Grégory Haïk, Thomas Meurisse, Jean-Pierre Briot, and Marc Lhuillier. Mise en oeuvre d'une approche componentielle pour la conception d'agents. In Hermes, editor, *Journées Francophones de l'Intelligence Artificielle Distribuée et des Systèmes Multi-Agents*, pages 53–65, 1999.

[Gutknecht *et al.*, 2001] Olivier Gutknecht, Jacques Ferber, and Fabien Michel. Integrating tools and infrastructures for generic multi-agent systems. In Jörg P. Müller, Elisabeth Andre, Sandip Sen, and Claude Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 441–448, Montreal, Canada, 2001. ACM Press.

[Guttman *et al.*, 1998] R. Guttman, A. Moukas, and P. Maes. Agent-mediated electronic commerce : A survey. *Data and Knowledge Engineering*, 13, 1998.

[Hachette, 2002] Le dictionnaire universel francophone en ligne. http ://www.francophonie.hachette-livre.fr, 2002.

[Hallal *et al.*, 2001] H. Hallal, A. Petrenko, A. Ulrich, and S. Boroday. Using sdl tools to test properties of distributed systems. In *Proceeddings of the Formal Approches to Testing of Software (FATES'01), Workshop of the International Conference on Concurrency Theory (CONCUR'01)*, pages 125–140, Aalborg, Denmark, 2001.

[Hannoin *et al.*, 1999] Mahdi Hannoin, Olivier Boissier, Jaime Simao Sichman, and Claudette Sayettat. Moise : un modèle organisationel pour la conception de systèmes multi-agents. In *VIIe Journées Francophones de l'Intelligence Artificielle Distribuée et des Systèmes Multi-Agents*, pages 105–118. Hermes, 1999.

[Heineman, 2001] G. Heineman. *Component-based software engineering.* Addison Wesley, 2001.

[Hewitt, 1977] Carl Hewitt. Viewing control structures as pattern of passing messages. *Artificial Intelligence*, 8(3) :323–364, 1977.

[Hewitt, 1986] C. Hewitt. Offices are open systems. *ACM Transaction on Office Information Systems*, 4(3) :271–287, 1986.

[Horling and Lesser, 1998] Bryan Horling and Victor Lesser. A reusable component architecture for agent construction. Technical report, University of Massachusetts/Amherst, 1998.

[IBAUTS, 2002] Intelligent building automation system. http ://cetc-varennes.nrcan.gc.ca/fichier.php/codectec/Fr/2001-65/2001-65f.pdf, 2002.

[Jade, 2003] Java agent development framework. http ://jade.cselt.it, 2003.

[Jennings and Wooldridge, 1998] N. R. Jennings and M. Wooldridge. Applications of intelligent agents. In *Agent technology : foundations, applications, and markets*, pages 3–28. Springer-Verlag New York, Inc., 1998.

[Jennings *et al.*, 1998] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1) :7–38, 1998.

[Joshi and Singh, 1999] A. Joshi and M. P. Singh. Multiagent systems on the net. *Communications of the ACM*, 42(1) :39–49, 1999.

[Jung and Fischer, 1998] Christoph Jung and Klaus Fischer. Methodological comparison of agent models. Technical Report RR-98-01, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, 1998.

[Kahn and Cicalese, 2002] M. L. Kahn and Cynthia Della Torre Cicalese. The coabs grid. In *Goddard/JPL Workshop on Radical Agent Concepts, Tysons Corner*, January 2002.

[Kendall, 2000] E.A. Kendall. Role modelling for agent systems analysis, design and implementation. *IEEE Concurrency*, 8(2) :34–41, 2000.

[Keppel *et al.*, 1991] D. Keppel, S.J. Eggers, and R.R. Henry. A case for runtime code generation. Technical Report Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington, 1991.

[Kiczales *et al.*, 1991] Gregor Kiczales, Jim des Rivieres, and Daniel Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Massachusetts, 1991.

[Kiczales *et al.*, 1997] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Kiczales *et al.*, 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072 :327–355, 2001.

[Kniesel, 1998] Günter Kniesel. Type-safe delegation for dynamic component adaptation. In Serge Demeyer and Jan Bosch, editors, *ECOOP Workshops*, volume 1543 of *Lecture Notes in Computer Science*, pages 136–137. Springer, 1998.

[Kotz and Gray, 1999] David Kotz and Robert S. Gray. Mobile agents and the future of the internet. *ACM Operating Systems Review*, 33(3) :7–13, 1999.

[Krutisch *et al.*, 2003] Richard Krutisch, Philipp Meier, and Martin Wirsing. The agent component approach, combining agents and components. In *Proceding 1st German Conf. Multiagent System Technologies (MATES'03), Lect. Notes Artif. Intell.*, pages 1–12, Berlin, Germany, 2003. Springer.

[KSE, 1994] Knowledge sharing effort. http ://www.cs.umbc.edu/kse, 1994.

[Labrou and Finin, 1994] 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.

[Lange and Oshima, 1998] Danny Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agent with Aglets.* Computer and Engineering Publishing Group, 1998.

[Lange and Oshima, 1999] D. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 43(3), March 1999.

[Lange, 1998] Danny B. Lange. Present and future trends of mobile agent technology. In *Mobile Agent 98*, 1998. Invited talk.

[Lemaître *et al.*, 2003] C. Lemaître, X. Prat, L. Magnin, and A. Dury. Description, programmation et validation d'interactions par catn. In *Secondes Journées Francophones sur les Modèles Formels d'Interactions*, Lille, France, 2003.

[Lhuillier, 1998] Marc Lhuillier. *Une approche à base de composants logiciels pour la conception d'agents. Principes et mise en oeuvre à travers la plate-forme Maleva.* PhD thesis, Université Paris VI, 1998.

[Madkit, 1999] Madkit. http ://www.madkit.org, 1999.

[Maes, 1994] P. Maes. Modeling adaptive autonomous agents. *Artificial Life, I*, (1&2)(9), 1994.

[Magnin *et al.*, 2002a] L. Magnin, H. Snoussi, V. T. Pham, A. Dury, and J. Y. Nie. Agents need to become welcome. In *The 3rd International Symposium on Multi-Agent Systems, Large Complex Systems, and E-Businesses (MALCEB'02)*, Erfurt/Thuringia, Germany, 10 2002.

[Magnin *et al.*, 2002b] Laurent Magnin, Viet Thang Pham, Arnaud Dury, Nicolas Besson, and Houari Sahraoui. Adaptive agents to heterogeneous platforms, new protocols and evolving organizations. In *Symposium on Adaptive Agents and Multi-Agent Systems (AAMAS-2), AISB'02 Convention*, pages 9–19, London, England, 2002.

[Magnin *et al.*, 2002c] Laurent Magnin, Viet Thang Pham, Arnaud Dury, Nicolas Besson, and Arnaud Thiefaine. Our guest agents are welcome to your agent platforms. In *ACM Symposium on Applied Computing*, Madrid, Spain, 2002.

[Magnin, 1996] Laurent Magnin. *Modélisation et simulation de l'environnement dans les systèmes multi-agents. Application aux robots footballeurs.* PhD thesis, Université Pièrre et Marie Curie, Paris, 1996.

[Magnin, 1999] Laurent Magnin. Internet, environnement complexe pour agents situés. In *Conférence sur l'Intelligence Artificielle Située (IAS)*, pages 213–221, Paris, France, 1999.

[Malabarba *et al.*, 2000] S Malabarba, R. Pandey, J. Gragg, E. Barr, and J.F. Barnes. Runtime support for type-safe dynamic java classes. In E. Bertino, editor, *European*

Conference on Object-oriented Programming (ECOOP 2000), pages 337–361, Cannes, France, 2000.

[Mamei and Zambonelli, 2003] Marco Mamei and Franco Zambonelli. Self-organization in multi agent systems : A middleware approach. In Serugendo et al. [2004], pages 233–248.

[Marini *et al.*, 2000] Simone Marini, Maurizio Martelli, Viviana Mascardi, and Floriano Zini. Specification of heterogeneous agent architectures. In *Agent Theories, Architectures, and Languages*, pages 275–289, 2000.

[Marrow *et al.*, 2002] Paul Marrow, Cefn Hoile, Fang Wang, and Erwin Bonsma. Evolving preference among emergent groups of agents. In *AISB Convention, Symposium on Adaptive agents and multi-agent systems*, pages 67–74, London, England, 2002.

[Martin *et al.*, 1999] D. Martin, A. Cheyer, and D. Moran. The Open Agent Architecture : a framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1/2) :91–128, 1999.

[Mezura *et al.*, 1999] Carmen Mezura, Michel Occello, Yves Demazeau, and Christof Baeijs. Récursivité dans les systèmes multi-agents : Vers un modèle opérationnel. In Hermes, editor, *Journées Francophones de l'Intelligence Artificielle Distribuée et des Systèmes Multi-Agents*, pages 41–52, 1999.

[Muscettola *et al.*, 1998] Muscettola, Pandurang Nayak, Barney Pell, and Brian C Williams. Remote agent : To boldly go where no ai system has gone before. *Artificial Intelligence*, 1998.

[Nourchene, 2003] Ben Ayed Nourchene. *Analyse des traces d'exécution pour la vérification des protocoles d'interaction dans les systèmes multi-agents.* PhD thesis, Université de Montréal, 2003.

[Nwana *et al.*, 1999] Hyacinth Nwana, Divine Ndumu, Lyndon Lee, and Jaron Collis. Zeus : A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence Journal*, 13(1), 1999.

[Odell *et al.*, 2000] J. Odell, H. Parunak, and B. Bauer. Extending uml for agents, 2000.

[Oliveira, 1997] Luiz A. G. Oliveira. An agent-based approach for quality of service negociation and management in distributed multimedia systems. In *Mobile Agent MA97*, 1997.

[OMG, 1999a] OMG, Meta Object Facility, http ://www.omg.org, 1999.

[OMG, 1999b] OMG. Maf : Mobile agent facility. http ://www.fokus.gmd.de/research/cc/ima/masif/index.html, 1999.

[Oriol, 2002] Manuel Oriol. Evolution of Code through Asynchronous Services. In *Workshop on Unanticipated Software Evolution.* European Conference on Object-Oriented Programming (ECOOP'02), 2002.

[Pal, 1998] Chris Pal. A technique for illustrating dynamic component level interactions within a software architecture. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON*, page 18. IBM, 1998.

[Paleczny *et al.*, 2001] M. Paleczny, C. Vick, and C. Click. The java hotspot server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM 2001)*, 2001.

[Pawlak *et al.*, 2001] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. Jac : A flexible solution for aspect-oriented programming in java. In *REFLECTION '01 : Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 1–24. Springer-Verlag, 2001.

[Perdikeas *et al.*, 1999] Menelaos K. Perdikeas, Fotis G. Chatzipapadopoulos, Iakovos S. Venieris, and Gennaro Marino. Mobile agent standards and available platforms. *Computer Networks Journal*, 31(10), 1999.

[Pham *et al.*, 2001] Viet Thang Pham, Laurent Magnin, and Arnaud Dury. Guest, système d'agents récursifs, mobiles et multi-plates-formes. In Hermes, editor, *Journées Francophones de l'Intelligence Artificielle Distribuée et des Systèmes Multi-Agents*, pages 303–306, 2001.

[Pham *et al.*, 2004] Viet Thang Pham, Laurent Magnin, and Houari Sahraoui. Adaptation dynamique des systèmes multiagents basée sur le concept de méta-catn. In *Recherche Informatique Vietnam- Francophone*, 2004.

[Pinsdorf and Roth, 2002] Ulrich Pinsdorf and Volker Roth. Mobile agent interoperability patterns and practice. In *Proceedings of Ninth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2002)*, Computer Graphics Edition, pages 238–244, University of Lund, Lund, Sweden, April 2002. Institute of Electrical and Electronics Engineers, IEEE Computer Society Press. ISBN 0-7695-1549-5.

[Pitrat, 1990] J. Pitrat. An intelligent system must and can observe its own behavior. In *Proceedings of Third Cognitiva Symp.*, pages 119–128, Elsevier, 1990.

[Poslad, 2001] Stefan Poslad. Standardizing agent interoperability : the fipa approach. In *Advanced Course on Artificial Intelligence ACAI-01*, 2001.

[Rao and Georgeff, 1995] A.R. Rao and M. Georgeff. Bdi agents : From theory to practice. In *First International Conference on Multiagent Systems (ICMAS-95)*, pages 312–319, Sanfrancisco, USA, 1995.

[Rey and Coutaz, 2004] Gaëtan Rey and Joëlle Coutaz. The contextor infrastructure for context-aware computing. In *Component-oriented approaches to context-aware computing 2004*, Oslo, Norway, 2004.

[Ricordel and Demazeau, 2000] Pierre-Michel Ricordel and Yves Demazeau. From analysis to deployment : A multi-agent platform survey. In *Engineering Societies in the Agents World*, pages 93–105. Springer-Verlag, 2000.

[Routier and Mathieu, 2001] Jean-Christophe Routier and Philippe Mathieu. Une contribution du multi-agent aux applications de travail coopératif. *Technique et Science Informatiques*, 20(6), 2001.

[Russel and Norvig, 1995] S.J. Russel and P. Norvig. *Artificial Intelligence : A modern Approach.* Prentice Hall, 1995.

[Sahraoui Houari, 1995] A. Sahraoui Houari. *Application de la méta-modélisation à la génération des outils de conception et de mise en oeuvre des bases de données.* PhD thesis, Université Paris 6, 1995.

[Satoh, 2001] Ichiro Satoh. Network processing of mobile agents, by mobile agents, for mobile agents. In *Proceedings of 3rd International Workshop on Mobile Agents for Tele-*

communication Applications (MATA'2001), volume 2146 of *Lecture Notes in Computer Science*, pages 81–92, 2001.

[Schneider *et al.*, 2001] Jean-Guy Schneider, Markus Lumpe, and Oscar Nierstrasz. Agent coordination via scripting languages. In *Coordination of Internet agents : models, technologies, and applications*, pages 153–175. Springer-Verlag, London, UK, 2001.

[Searle, 1969] J.R. Searle. *Speech Acts.* Cambridge University Press, 1969.

[Serugendo *et al.*, 2004] Giovanna Di Marzo Serugendo, Anthony Karageorgos, Omer F. Rana, and Franco Zambonelli, editors. *Engineering Self-Organising Systems, Nature-Inspired Approaches to Software Engineering [revised and extended papers presented at the Engineering Self-Organising Applications Workshop, ESOA 2003, held at AAMAS 2003 in Melbourne, Australia, in July 2003 and selected invited papers from leading researchers in self-organisation]*, volume 2977 of *Lecture Notes in Computer Science.* Springer, 2004.

[Shoham, 1993] Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1) :51–92, 1993.

[Smith, 1982] Brian Smith. *Procedural reflection in programming languages.* PhD thesis, Massachusetts Institute of Technology, 1982.

[Stolzenburg and Arai, 2003] Frieder Stolzenburg and Toshiaki Arai. From the specification of multiagent systems by statecharts to their formal analysis by model checking : Towards safety-critical applications. In *Proc. 1st German Conf. Multiagent System Technologies (MATES'03), Lect. Notes Artif. Intell.*, Berlin, Germany, 2003.

[Stone *et al.*, 2001] P. Stone, M. L. Littman, S. Singh, and M. Kearns. Attac-2000 : An adaptive autonomous bidding agent. In *Autonomous Agents*, pages 238–245, 2001.

[Sycara *et al.*, 1998] Katia Sycara, Jianguo Lu, and Matthias Klusch. Interoperability among heterogeneous software agents on the internet. Technical report, Robotics Institute, Carnegie Mellon University, 1998.

[Sycara *et al.*, 2001] K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa. The RETSINA MAS infrastructure. Technical Report CMU-RI-TR-01-05, Robotics Institute Technical Report, Carnegie Mellon, 2001.

[Tambe *et al.*, 2000] M. Tambe, D. Pynadath, N. Chauvat, A. Das, and G. Kaminka. Adaptive agent integration architectures for heterogeneous team members. In *Proceedings of the International Conference on Multiagent Systems*, pages 301–308, Boston, USA, 2000.

[Tambe, 1995] M. Tambe. Recursive agent and agent-group tracking in a real-time dynamic environment. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems (ICMAS'95)*, San Francisco, CA, USA, 1995. AAAI Press.

[Tardo and Valente, 1996] Joseph Tardo and Luis Valente. Mobile agent security and telescript. In *COMPCON '96 : Proceedings of the 41st IEEE International Computer Conference*, page 58. IEEE Computer Society, 1996.

[Tjung *et al.*, 1999] D. Tjung, M. Tsukamoto, and S. Nishio. A converter approach for mobile agent system integration : A case of aglet to voyager. In *First International Workshop on Mobile Agents for Telecommunication Applications*, pages 179–195, Ottawa, Canada, 1999.

[Valetto and Kaiser, 2002] Giuseppe Valetto and Gail Kaiser. A case study in software adaptation. In *Proceedings of the first workshop on Self-healing systems 2002*, pages 73 – 78, Charleston, South Carolina, USA, 2002.

[Vanderveken, 1990] D. Vanderveken. *Meaning and speech acts.* Cambridge University Press, 1990.

[Vidal, 2001] José M. Vidal. A generic agent architecture for multiagent systems. Technical report, University of South Carolina, 2001.

[Voyager, 1999] Voyager by objectspace. http ://www.objectspace.com/voyager/, 1999.

[Vu *et al.*, 2003] Thuc Vu, Jared Go, Gal Kaminka, Manuela Veloso, and Brett Browning. Monad : a flexible architecture for multi-agent control. In *Proceedings of the second international joint conference on Autonomous agents and multiagent*, pages 449 – 456, Melbourne, Australia, 2003.

[Wood, 1987] Derick Wood. *Theory of Computation.* Harper and Row, Publishers, Inc., New York, 1987.

[Woods, 1970] W.A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10), 1970.

[Wooldridge *et al.*, 2000] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3) :285–312, 2000.

[Wooldridge, 2001] Michael Wooldridge. *An introduction to Multi Agent Systems.* John Wiley and Sons, 2001.

[WSDL, 2002] Web service description language. http ://www.w3.org/TR/wsdl, 2002.

[Yang and Papazoglou, 2002] J. Yang and M.P. Papazoglou. Web component : A substrate for web service reuse and composition. In *LNCS 2348, CaiSE2002*, pages 21–36, 2002.

[Yoo, 2000] Min-Young Yoo. *Une approche componentielle pour la modélisation d'agent coopératifs et leur validation.* PhD thesis, Université Paris VI, 2000.

[Zambonelli *et al.*, 2001] F. Zambonelli, N. Jennings, A. Omicini, and M. Wooldridge. Agent-oriented software engineering for internet applications. In *Coordination of Internet Agents : Models, Technologies and Applications.* Springer-Verlag, London, UK, 2001.

[Zavala *et al.*, 2001] L. Zavala, A. Careño, and C. Lemaître. Catnagent toolkit : una plataforma para el diseño y ejecución de sistemas multiagentes. In *Mexican International Conference on Computer Science*, Mexico, 2001.

[Zimmerman, 1996] Chris Zimmerman. *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press, Inc., 1996.

Annexe I

API de la plate-forme *Guest*

I.1 Organisation du code de *Guest*

Pour assurer l'uniformité des agents universels, le code de *Guest* doit être séparé entre une moitié dépendant des plates-formes natives et l'autre entièrement indépendant d'elles. En général, les utilisateurs ne travaillent qu'avec la partie indépendante, sauf dans le cas où ils veulent accéder explicitement aux fonctionnalités des plates-formes natives.

Guest est prévue pour trois types de développement, selon le besoin et les connaissances de la plate-forme :

- **les développeurs** de la plate-forme : ils connaissent très bien la plate-forme et peuvent la modifier et ajouter une nouvelle interface pour une nouvelle plate-forme native ;
- **les contributeurs** qui contribue à l'extension de la plate-forme en offrant des plugins : ils travaille principalement avec l'interface des plugins ;
- **les utilisateurs** qui développent des applications multi-agents en utilisant la plate-forme : ils ne s'intéressent qu'à l'interface publique de *Guest*.

Le code de la plate-forme *Guest* est divisé en plusieurs packages, dont les noms doivent toujours commencer par **ca.crim-udem.guest**[1] (figure I.1) :

- le package **main** contient l'interface publique de *Guest*, qui est indépendant des plates-formes natives. Les utilisateurs travaillent principalement avec ce package ;
- le package **platform** n'est utilisé qu'en interne à *Guest*. Les classes de ce package jouent le rôle d'interface avec les plates-formes natives. En conséquence, ce package est la seule partie dépendant des plates-formes natives. Seuls les développeurs peuvent toucher à ce package ;
- le package **ext** et ses sous-packages contiennent les extensions de *Guest*. Tous les

[1] Les utilisateurs de *Guest* peuvent utiliser leurs propres noms de packages pour développer leurs applications.

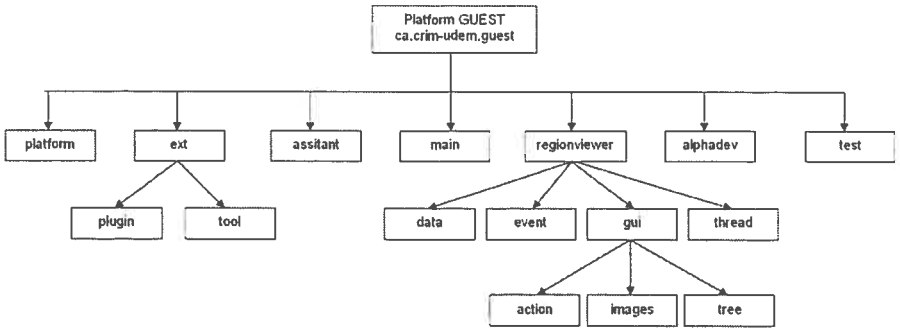

FIG. I.1 – Structure des packages de *Guest*

plugins sont mis dans le sous-package **ext.plugin**, y compris certains plugins fournis avec la distribution de *Guest*. Tout le monde peut contribuer à ce package ;

- Le package **regionviewer** et ses sous-packages offre un outil sous forme d'interface graphique pour la manipulation visuelle des agents *Guest* sur le réseau ;
- Le package **alphadev** contient des classes expérimentales de *Guest*, écrites par les développeurs ;
- Le package **test** contient les tests de fonctionnalités de *Guest* ;
- Le package **example** contient des examples d'utilisation de *Guest*.

I.2 Installation

I.2.1 Pré-requis

Il faut tout d'abord télécharger la ou les plate(s)-forme(s) de travail. Voici les adresses des sites où vous pouvez télécharger les plates-formes :

- Java à http ://java.sun.com/ (english)
- Aglets à http ://aglets.sourceforge.net/ (english)
- Voyager à http ://www.recursionsw.com/products/voyager/ (english)
- Grasshopper à http ://www.grasshopper.de/ (english)
- Jade à http ://jade.cselt.it/ (english)
- *Guest* à http ://www.iro.umontreal.ca/ phamviet/guest

I.2.2 Configuration des paramètres systèmes

La configuration des variables d'environnement permet au système de trouver les programmes et librairies essentielles à leur bon fonctionnement. Pour ce faire, nous indiquons, dans les paragraphes qui suivent, les informations nécessaires.

I.2.2.1 Les éléments principaux du paramétrage

Au moins deux variables (indispensables) doivent être renseignés :

- PATH : cette variable inclut les répertoires à visiter pour trouver les programmes à exécuter. En général, les applications sont dans le répertoire bin.
- CLASSPATH : celle-ci informe le compilateur java où trouver les fichiers class dont il a besoin. En général, ils sont regroupés dans la librairie, répertoire lib.

Il faut indiquer les chemins des programmes suivants :

- Chemin du répertoire du compilateur Java
- Chemin de la (ou des) plate(s)-forme(s) choisie(s)
- Chemin du répertoire contenant la plate-forme *Guest*

Vous aurez à créer des scripts sous Unix ou des fichiers Batch sous Windows. Cela facilite grandement les manipulations : vous n'êtes pas obligés de retaper les lignes de commandes fastidieuses en indiquant le chemin complet des répertoires des fichiers concernés.

I.2.2.2 Le paramétrage sous Unix

Il est conseillé de passer sous l'environnement **tcsh** (celui que nous utilisons). Dans un terminal, tapez `tcsh` pour basculer dans l'environnement **tcsh** et placez-vous dans le répertoire guest/bin. Les sous sections suivantes montrent comment écrire un script et comment positionner les variables d'environnements.

Écriture d'un script de configuration

Pour écrire le script de configuration, vous pouvez utiliser n'importe quel éditeur de texte. Par exemple, celui avec lequel vous êtes le plus familiarisé. vi est probablement le plus courant. Pensez à enregistrer votre script sous un nom explicite (par exemple setup).

L'instruction pour modifier les valeurs des variables est SETENV dont les arguments

sont :

- $1 : le nom de la variable. Par la suite, si vous voulez utiliser la variable (par substitution et obtenir sa valeur), faites-la précéder par le symbole $.
- $2 : le chemin à ajouter.

Dans le cas de PATH et CLASSPATH, il est préférable de joindre respectivement ces variables pour conserver les chemins déjà déclarés. Pour séparer les chemins de deux répertoires, vous utilisez le séparateur deux point « : ». Par exemple : `setenv CLASSPATH guest/lib/guest.jar :$CLASSPATH`

Remarques :

- entre le mot clé de la variable et le chemin, il y a un espace. Et surtout pas de symbole égal « = ».
- les fichiers .jar doivent être explicitement déclarés dans la variable CLASSPATH.

Voici un exemple de script de configuration, lequel peut être trouvé dans le répertoire `guest/bin/setenv.csh`. Prenez garde de bien modifier les chemins en fonction de vos paramètres :

```
setenv JAVA_HOME /usr/local/java/jdk1.5.0.1

setenv GUEST_HOME $HOME/guest
setenv GUEST_LIB $Guest_HOME/bin/guest.jar:
$Guest_HOME/bin/GuestLauncher.jar

setenv VOYAGER_HOME $HOME/voyager
setenv VOYAGER_LIB "$VOYAGER_HOME/lib/voyager.jar:
$VOYAGER_HOME/lib/jgl3.1.0.jar"

setenv GH_HOME $HOME/grasshopper
setenv GH_LIB "$GH_HOME/lib/gh.jar:$GH_HOME/lib/ldap.jar:
$GH_HOME/lib/jdni.jar:
$GH_HOME/lib/providerutil.jar"
```

```
setenv AGLET_HOME $HOME/aglet
setenv AGLET_LIB $AGLET_HOME/lib/aglets.jar

setenv CLASSPATH "$Guest_LIB":"$VOYAGER_LIB":"$GH_LIB":
"$AGLET_LIB": vos propres répertoires

setenv PATH $JAVA_HOME/bin:$GH_HOME/bin:$VOYAGER_HOME/bin:
$GH_HOME/bin:$AGLET_HOME/bin:$GUEST_HOME/bin:$PATH
```

Mise à jour des variables d'environnement

Pour positionner vos variables `$PATH` et `$CLASSPATH`, vous utilisez la commande `SOURCE` suivi du nom du script qui contient vos instructions.

- Ouvrez un terminal.
- tapez : `tcsh` //basculer sous TCSH.
- tapez : `source guest/bin/setup` //mise à jour des variables d'environement (ici, setup est le nom du script).

Remarque : Vous devez ouvrir autant de terminaux que de serveurs à exécuter sur votre machine. Dans chacun d'eux, repositionnez les variables (exécution du setup).

I.2.2.3 Le paramétrage sous Windows

Pour lancer un serveur sous Windows, il suffit d'ouvrir une fenêtre ms-dos et d'y exécuter le fichier Batch de configuration que vous aurez, au préalable, crée. Les sous-sections suivantes montrent comment écrire un fichier batch qui regroupe les lignes de commandes nécessaires au paramétrage de votre environnement.

Écriture d'un fichier Batch de configuration

Pour écrire le fichier Batch de configuration, vous pouvez utiliser n'importe quel éditeur de texte. Par exemple, celui avec lequel vous êtes le plus familiarisé. Notepad est le plus couramment utilisé. Tout d'abord, placez-vous dans le répertoire `guest/bin`. Ouvrez un nouveau fichier dans Notepad que vous enregistrez sous un nom explicite (par exemple

winsetup). L'extension d'un fichier Batch est `.bat` (par exemple `winsetup.bat`).

L'instruction pour modifier les valeurs des variables est SET dont les arguments sont :

- %1 : le nom de la variable. Par la suite, si vous voulez utiliser la variable (par substitution et utiliser sa valeur), faites là encapsuler par le symbole % (pourcentage).
- %2 : le chemin à ajouter.

Dans le cas de PATH et CLASSPATH, il est préférable de joindre respectivement ces variables pour conserver les chemins déjà déclarés. Pour séparer les chemins de deux répertoires, vous utilisez le séparateur point virgule « ; ». Par exemple :

```
set CLASSPATH=guest/lib/guest.jar ;%CLASSPATH%
```

Remarques :

- entre le mot clé de la variable et le chemin, il y a le symbole égal « = ».
- les fichiers jar doivent être explicitement déclarés dans la variable `CLASSPATH`.

Voici un exemple de fichier Batch de configuration, lequel peut être trouvé dans `guest/bin/set_env.bat`. Prenez garde de bien modifier les chemins en fonction des vos paramètres.

```
set HOME=u:
set Guest_HOME=%HOME%\guest
set Guest_LIB=%Guest_HOME%\bin\guest.jar;
%Guest_HOME%\bin\GuestLauncher.jar

set VOYAGER_HOME=%HOME%\voyager
set VOYAGER_LIB=%VOYAGER_HOME%\lib\voyager.jar;
%VOYAGER_HOME%\lib\jgl3.1.0.jar

set GH_HOME=%HOME%\grasshopper
set GH_LIB=%GH_HOME%\lib\gh.jar;%GH_HOME%\lib\ldap.jar;
%GH_HOME%\lib\jndi.jar;%GH_HOME%\lib\providerutil.jar

set AGLET_HOME=%HOME%\aglet
```

```
set AGLET_LIB= %AGLET_HOME%\lib\aglets.jar

set CLASSPATH=%Guest_LIB%;%VOYAGER_LIB%;%GH_LIB%;
%AGLET_LIB%; vos propres répertoires

set PATH=%PATH%;%Guest_HOME%\bin;%VOYAGER_HOME%\bin;
%GH_HOME%;%AGLET_HOME%
```

Mise à jour des variables d'environnement

Pour positionner les variables %PATH% et %CLASSPATH%, ouvrez une fenêtre MS-DOS, tapez le nom du fichier Batch. S'il ne se trouve pas dans votre répertoire de travail, vous devez préciser le chemin d'accès.

- Ouvrez une fenêtre ms-dos (menu Démarré/programmes/Lignes de Commandes)
- Tapez `cd guest/bin` //changer de répertoire
- Taper `winsetup.bat` //exécution du batch dont le nom du batch est winsetup.bat)

Remarque : Vous devez ouvrir autant de fenêtres que de serveurs à exécuter sur votre machine. Dans chacune d'elles, repositionner les variables (en exécutant le batch).

I.2.3 Procédure de lancement des plates-formes

I.2.3.1 Par la ligne de commande

Voici les différentes étapes à suivre :

- Ouverture d'un terminal sous Unix ou fenêtre ms-dos sous Windows
- Exécution des scripts dans un terminal ou Batch de configuration dans une fenêtre (Cf. configuration des paramètres systèmes ci-dessus).
- Exécution du serveur de la plate-forme choisie

Remarque : vous pouvez avoir plusieurs serveurs sur la même machine. Pour cela, il vous suffit d'ouvrir autant de terminaux ou de fenêtres configurés que de serveurs à exécuter. Les serveurs doivent avoir des numéros de port différents. Recommencez les étapes au-dessus autant de fois que cela est nécessaire.

Pour chaque plate-forme, nous avons détaillé les lignes de commande pour lancer un serveur sur votre machine :

Ligne de commande pour Aglets : N/A

Ligne de commande pour Voyager `voyager numéro de port -r`

Ligne de commande pour Grasshopper `java de.ikv.grasshopper.Grasshopper a -tu -n nom d'agence`
`-ser socket numéro de port`

Ligne de commande pour Jade `java jade.Boot -container nom de container -gui`

Remarques :

- Dans votre répertoire home, Grasshopper crée un répertoire .grasshopper dont l'attribut *caché* est positionné. Ce dossier contient le statut courant de votre plate-forme.
- Il contient, entre autre, le sous-répertoire locks où il place des fichiers interdisant la redondance d'un serveur du même nom d'agence. A chaque exécution de la plate-forme, vous devez supprimer ce fichier .lock pour le bon fonctionnement de Grasshopper.
- Le mieux est d'écrire un script ou batch avec les 2 lignes correspondant aux opérations de suppression du fichier .lock et de lancement du serveur Grasshopper (commande ci-dessus).

I.2.3.2 Par l'outil graphique GuestLauncher

GuestLauncher est une interface graphique qui aide à configurer et à lancer les différents serveurs multi-agents de façon transparente avec ces plates-formes. Les étapes de lancement est comme suit :

- Ouvrir d'un terminal ou fenêtre MS-DOS ;
- Entrer dans le répertoire où se trouve le fichier launcher.jar ;
- Éxecuter la commande : `java -cp ./ -jar launcher.jar`

I.2.4 Procédure de lancement l'outil graphique RegionViewer

RegionViewer est une interface graphique qui aide à visualiser les agents *Guest* et les serveurs sur lesquels évoluent les agents créés. Les utilisateurs peuvent observer le déroulement de ces derniers. Les administrateurs peuvent en plus interagir avec les agents, soit par l'intermédiaire des serveurs, soit directement en leur donnant des nouvelles instructions par le biais d'envois de message. Son utilisation est détaillée dans la partie suivant.

Voici les différentes étapes à suivre :

- Ouverture d'un terminal ou fenêtre ms-dos
- Exécution des scripts dans un terminal ou Batch de configuration dans une fenêtre (Cf. configuration des paramètres systèmes ci-dessus).
- Exécution de la ligne de commande. Après l'exécution de cette ligne de commande, l'interface graphique RegionViewer est affichée.

Ligne de commande sous Unix :

```
java -cp $CLASSPATH ca.crim-udem.guest.gui.RegionViewer
mode adresse du serveur
```

Ligne de commande sous Windows :

```
java -cp %CLASSPATH% ca.crim-udem.guest.gui.RegionViewer
mode adresse du serveur
```

Vous remarquez que la syntaxe est presque identique à l'exception de l'écriture de la variable.

Remarques :

- mode : *view* pour pouvoir simplement visualiser les agents *Guest* ou *admin* pour être en mode administrateur (i.e., pour pouvoir créer, détruire, déplacer, désactiver et réactiver un agent *Guest* ou envoyer un message à un agent *Guest*) ;
- un string représentant le `GuestURL` du serveur sur lequel l'agent `RegionViewer` sera créé : `Plate-forme ://nom_machine :port [/extension]`

I.3 Outil de visualisation : RegionViewer

RegionViewer est un agent GuestRegion disposant d'interface graphique et ajoutant les fonctionnalités de manipulation des agents *Guest*. Cet agent est une contribution de Nicolas Besson, l'agent de recherche de l'équipe GLIC au CRIM.

Pour lancer un agent RegionViewer, il faut s'assurer que le serveur sur lequel vous voulez créer cet agent soit en fonction.

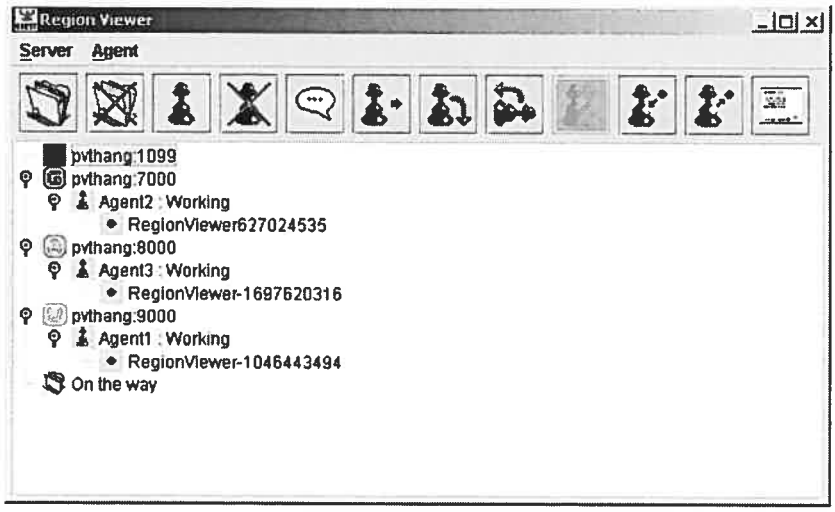

FIG. I.2 – Agent RegionViewer

L'interface graphique se compose de trois zones (figure I.2) :

- une barre de menu ;
- une barre des outils ;
- une liste des serveurs et leurs agents *Guest* ;

Dans la liste des serveurs, on indiquera pour chaque serveur la liste des agents qui s'y trouvent. Il y a un nud spécial : il contient des agents en train de migrer. Dans une fenêtre serveur on visualise la liste de tous les agents se trouvant sur ce serveur.

Pour sélectionner un serveur, choisissez-le dans la liste des serveurs et des agents. Les commandes dédiées aux serveurs s'effectueront sur le serveur sélectionné.

Pour sélectionner un agent, choisissez-le dans la liste des serveurs et des agents. Les

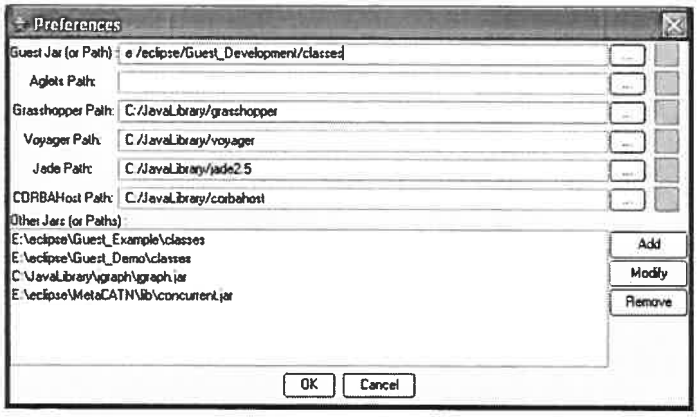

FIG. I.3 – Boite de dialogue de paramètrage des plates-formes

commandes dédiées aux agents s'effectueront sur l'agent sélectionné.

I.4 GuestLauncher

`GuestLauncher` est une application avec l'interface graphique qui aide à configurer et à lancer de différents serveurs d'agents de façon transparente avec leur plates-formes. *GuestLauncher* est une contribution de Arnaud Thiefene, un stagiaire au CRIM.

I.4.1 Première utilisation de l'application

Le launcher, pour fonctionner convenablement, a besoin de connaître l'emplacement de *Guest* et des différentes plates-formes. C'est pour cette raison qu'à la première exécution du launcher, une boite de dialogue (figure I.3) apparaît pour permettre à l'utilisateur d'indiquer ces emplacements.

Il n'est pas nécessaire de remplir tous les champs (mais pour celui de *Guest*, c'est préférable). Seulement, par la suite, il ne sera pas possible de choisir une plate-forme dont le chemin n'a pas été indiqué. L'ensemble des chemins est stocké dans un fichier `path.properties`, situé à la racine du répertoire de `GuestLauncher`.

Si l'on souhaite modifier les chemins, cette boite de dialogue est accessible par le menu `File/Edit Preferences` du launcher.

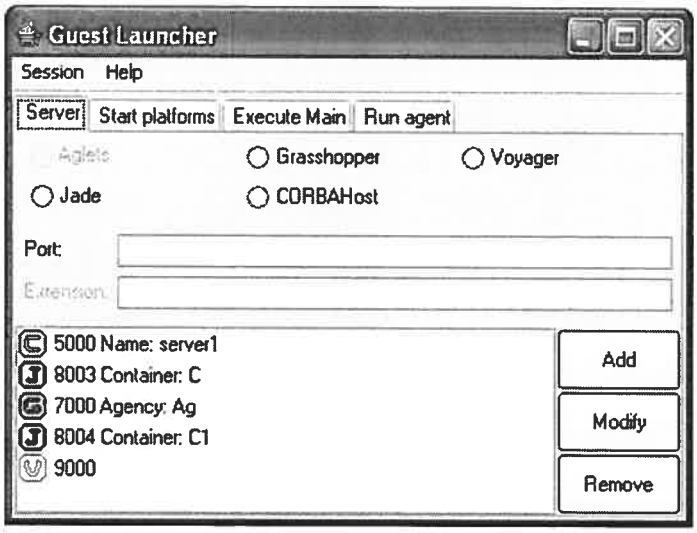

FIG. I.4 – Launcher - onglet des configurations de plates-formes

I.4.2 Utilisation du launcher

Le launcher se présente sous la forme d'une petite fenêtre, comportant plusieurs onglets (figure I.4). On trouve également un menu déroulant.

Le premier onglet permet de définir un ensemble de configurations de plates-formes, cet ensemble est géré dans une liste affichée au bas de l'onglet. On peut ajouter, modifier ou retirer des éléments dans la liste. Pour chaque élément, on choisit une des plates-formes disponibles, on indique son port d'exécution et une extension si nécessaire. On dispose ensuite d'un « *pool* » de configurations de lancements de plates-formes.

L'onglet suivant, `Start Platforms` (figure I.5), affiche la liste de l'ensemble des lancements configurés dans le premier onglet. Le fait d'appuyer sur le bouton Start déclenche l'exécution des plates-formes que l'on a cochées.

Pour chaque plate-forme lancée, une fenêtre permet de suivre l'évolution de la plate-forme (figure I.6). Chaque plate-forme tourne dans une instance de machine virtuelle Java qui lui est propre, dans un processus distinct. Néanmoins, le fait de quitter le launcher entraîne l'arrêt de toutes les plates-formes lancées (on évite ainsi d'avoir à la longue des

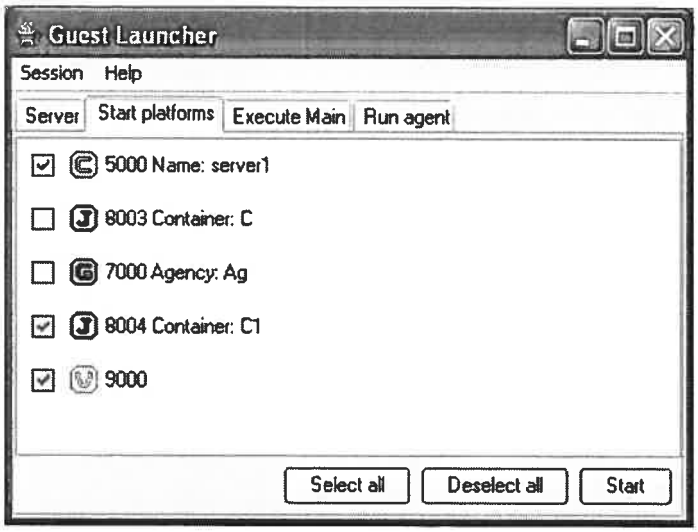

FIG. I.5 – Launcher - onglet de lancement des plates-formes

dizaines de machines Java qui tournent en même temps).

Dans l'exemple de la figure I.5, on lancerait les trois instances de plates-formes (un `Voyager`, un `CorbaHost` et un `Jade`) qui sont cochées.

Le troisième onglet, quant à lui, permet de spécifier un nom de classe Java qui contient une fonction `main()`, dont on peut indiquer les arguments. Le bouton permet d'exécuter le programme Java spécifié. Traditionnellement, on utilise cet onglet pour lancer la `RegionViewer`.

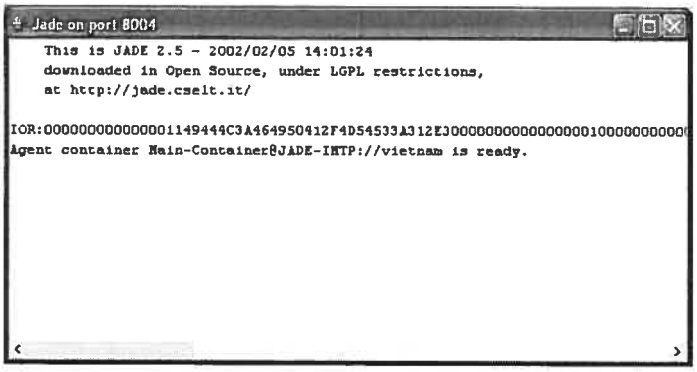

FIG. I.6 – Fenêtre affichant la sortie d'une plate-forme, ici Jade

Annexe II

Liste des plates-formes supportées par *Guest*

Le tableau suivant fournit la liste complète des plates-formes multi-agents sur lesquelles l'agent *Guest* peut fonctionner, et leur code dans la représentation de l'adresse universelle *Guest*.

Plate-formes	**Codes**
Aglets	AG
CORBAHost	CH
Grasshopper	GH
Jade	JD
Voyager	VY

Annexe III

Syntaxe complète de CATNML

Ci-dessous est la syntaxe complète du langage CATNML.

```
<!ELEMENT CATN (Register* , State+)>
<!ATTLIST CATN name NMTOKEN  \#REQUIRED >

<!ELEMENT Register EMPTY>
<!ATTLIST Register name         NMTOKEN  #REQUIRED
                   access       (private | public | inherited) #REQUIRED
                   dataType     CDATA    #REQUIRED
                   initialValue CDATA    #IMPLIED >

<!ELEMENT State (Transition?)>
<!ATTLIST State  name NMTOKEN  #REQUIRED >

<!ELEMENT Transition (Arc)+>
<!ATTLIST Transition type  (single | and | or | xor) #REQUIRED
                     parallel (true | false) #IMPLIED
                     arcsMustBeFired CDATA #IMPLIED
                     arcsMustBePassed CDATA #IMPLIED >

<!ELEMENT Arc (Constraint)?>
<!ATTLIST Arc  actionType          (CATN | PA | System |
CATNMR | CATNMS |
userMR | MRListener) #REQUIRED
               label               CDATA    #REQUIRED
               nextState           NMTOKEN  #REQUIRED
               param               CDATA    #IMPLIED
```

```
                result                CDATA    #IMPLIED
                targetObject          CDATA    #IMPLIED
                interactionWithCATN   CDATA    #IMPLIED
                agent                 CDATA    #IMPLIED
                channelName           CDATA    #IMPLIED
                isBroadcast           CDATA    #IMPLIED >

<!ELEMENT Constraint EMPTY>
<!ATTLIST Constraint type             (comparison | methodCall) #REQUIRED
                     leftType         CDATA  #IMPLIED
                     leftValue        CDATA  #IMPLIED
                     relationalOp     (equals |
                                       unequals |
                                       lessThan |
                                       lessThanEqual |
                                       greaterThan |
                                       greaterThanEqual )  #IMPLIED
                     rightType        CDATA  #IMPLIED
                     rightValue       CDATA  #IMPLIED
                     param            CDATA  #IMPLIED
                     label            CDATA  #IMPLIED>
```