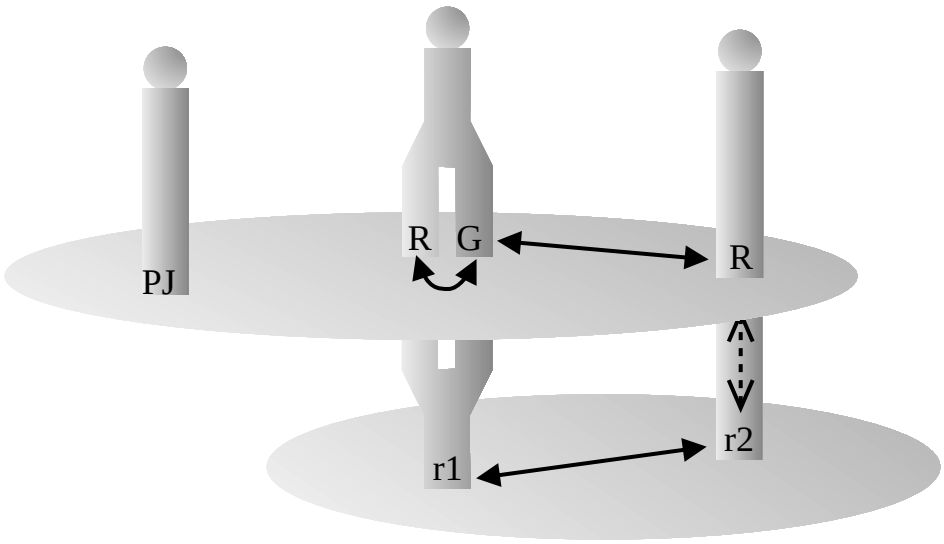
MOCA :
Un modèle componentiel dynamique pour les systèmes multi-agents organisationnels

THÈSE
soumise à la Faculté des sciences, pour obtenir
le grade de Docteur ès sciences, par

Matthieu Amiguet



Présentée le 24 janvier 2003 devant un jury composé de

- Jean-Pierre Müller, Université de Neuchâtel et Cirad de Montpellier, directeur.
- Olivier Besson, Université de Neuchâtel, rapporteur.
- Jean-Pierre Briot, LIP6, Paris, rapporteur.
- Jacques Ferber, LIRMM, Montpellier, rapporteur.
- Kilian Stoffel, Université de Neuchâtel, rapporteur.

Université de Neuchâtel
Institut d'Informatique et d'Intelligence Artificielle
rue Emile-Argand 11
2000 Neuchâtel
Suisse

We used to think that if
we knew one, we knew
two, because one and one
are two. We are finding
that we must learn a great
deal more about “and”.

Sir Arthur Eddington

(1882-1944)

IMPRIMATUR POUR LA THESE

MOCA : un modèle componentiel dynamique pour les systèmes multi-agents organisationnels

de M. Matthieu AMIGUET

UNIVERSITE DE NEUCHATEL

FACULTE DES SCIENCES

La Faculté des sciences de l'Université de Neuchâtel, sur le rapport des membres du jury

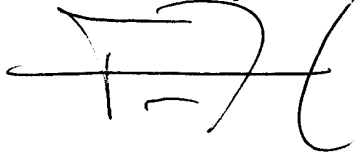
MM. J.-P. Müller (directeur de thèse), O. Besson, K. Stoffel, J. Ferber (Montpellier F) et J.-P. Briot (Paris)

autorise l'impression de la présente thèse.

Neuchâtel, le 5 mai 2003

Le doyen:

François Zwahlen



Remerciements

En premier lieu, je tiens à remercier tout particulièrement mon directeur de thèse Jean-Pierre Müller, qui a su me laisser très libre dans ma recherche tout en restant disponible chaque fois que j'avais besoin de conseils. Sa large culture et son goût de la partager m'ont fait découvrir de nouveau horizons philosophiques, scientifiques, culinaires et thérapeutiques, entre autres. Merci Jean-Pierre pour ton amitié et ta confiance.

Je remercie chaleureusement Olivier Besson, Jean-Pierre Briot, Jacques Ferber et Kilian Stoffel d'avoir accepté de former mon jury. Leur relecture attentive ainsi que leur regard à la fois critique et bienveillant sur ce travail m'auront permis de prendre du recul et de mieux situer cette contribution dans les différents contextes où elle se place.

L'équipe CASCAD de l'IIUN à l'Université de Neuchâtel aura constitué un cadre particulièrement propice aux discussions scientifiques et métaphysiques de tous ordres. Je remercie donc Éric Batard, Antoine Berner, Fabrice Bourquin, Yassine Faihe, Adina Nagy, Luc-Laurent Salvador, Luba Sidorova et Min-Jung Yoo pour les échanges enrichissants que j'ai pu avoir avec eux au long de ces années de thèse. Un merci tout particulier à José Báez pour la collaboration fructueuse que nous avons eue en développant la plate-forme MOCA.

Même si cela n'apparaît qu'en filigrane dans ce document, ces années ont été pour moi l'occasion d'explorer divers questionnement philosophiques et épistémologiques qui m'ont beaucoup apporté. Je remercie les nombreuses différentes personnes qui ont nourri cette réflexion, dont les membres et organisateurs des Journées Francophones de Rochebrune ainsi que Mioara Mugur-Schächter et les membres du CESEF à Paris.

Le contenu d'une thèse en informatique peut sembler rébarbatif au profane ; c'est pourquoi je remercie ceux de mes amis et de ma famille qui ont eu le courage de me demander ce que je faisais dans ma thèse... et d'écouter la réponse ! Le fait de devoir expliquer, en mots simples, à des non-spécialistes l'essence de ma démarche à grandement contribué à sa maturation. Parmi ces amis courageux, Gaël Liardon est probablement celui qui a montré le plus de ténacité à me comprendre et je l'en remercie.

Durant toute cette période, j'ai puisé dans la musique une grande partie de mon énergie. Je remercie donc les différentes personnes avec qui j'ai partagé cette passion et qui ont ainsi, indirectement mais indéniablement, contribué à ce mémoire. Un merci tout particulier à Isaline Dupraz pour les très beaux moments que nous avons partagés ensemble.

Enfin, pour toute la musique que nous avons faite ensemble ; pour son intérêt jamais démenti pour ma recherche ; pour ses relectures attentives et constructives de ce document ; pour son soutien constant au cours de ces années ; pour des raisons bien trop nombreuses pour être énumérées ici, je remercie infiniment ma compagne Barbara Minder.

L'utilisation de LyX a grandement facilité la rédaction de ce document et je remercie toute l'équipe de développement de son remarquable travail. Mais LyX ne serait rien sans tout un environnement qui assure son bon fonctionnement et mes remerciements s'étendent à la communauté des développeurs Linux dont les compétences et la générosité m'ont toujours impressionné.

Table des matières

Table des figures

Première partie

Contexte

Chapitre 1

Introduction

1.1 De l'intelligence artificielle aux systèmes multi-agents

Un peu d'histoire...

L'idée d'une créature fabriquée par l'homme à son image est probablement presque aussi vieille que l'humanité elle-même, comme en témoignent les nombreux mythes ancestraux tournant autour de ce thème. Cependant, les avancées techniques apparues au milieu du vingtième siècle dans le domaine du traitement automatique de l'information, alliées à une certaine mécanisation de la vision du monde, ont fortement ravivé les espoirs (et les craintes) de voir ce rêve se réaliser.

C'est ainsi que dans les années 50 est apparue l'expression "intelligence artificielle" (IA) pour désigner un champ relativement mouvant de la science, touchant aussi bien l'informatique, les mathématiques ou la logique que la psychologie, la linguistique, la sociologie, etc. Malgré ce nom prometteur on peut dire que, quelques 50 ans après la fondation de la discipline, celle-ci n'a toujours pas apporté de réponse à la question de sa définition même : "Qu'est-ce que l'intelligence ?"[1]. Il en résulte une branche aux contours flous et à large couverture, englobant à la fois les "correcteurs grammaticaux" disponibles dans certains traitements de textes actuels, des "simulations de colonies de fourmis" et des "animaux de compagnie" artificiels.

Pour comprendre comment on en est arrivé à cette diversité surprenante, et pour replacer brièvement cette thèse dans un contexte un peu plus large, nous proposons un survol bref (et partial !) des grands mouvements qui ont marqué la recherche en IA.

La première période de l'IA est caractérisée par un accent presque exclusif sur les capacités intellectuelles. De la preuve automatique de théorèmes aux systèmes experts, les recherches sont centrées sur les capacités de réflexion et de représentation du monde. Cette vision de l'IA culmine en 1997 avec la victoire d'un ordinateur, le RS/6000 d'IBM, dit *Deep Blue,* sur le champion du monde d'échecs de l'époque, Garry Kasparow.

Entre temps néanmoins, les limites d'une telle approche ont commencé à se faire sentir : l'intelligence, quelle que soit sa nature, ne saurait se résumer aux pures capacités intellectuelles. Un nouveau courant se dégage alors, qui désire prendre en compte les capacités sen-

[1] On pourrait même retracer l'histoire des succès de l'IA comme celle des définitions négatives de l'intelligence. A chaque nouveau succès d'une machine (victoire aux échecs, monter des escaliers, ...), l'évidence semble s'imposer que "ce n'est pas ça l'intelligence"...

sorielles et motrices dans l'approche de l'IA [PS99]. Le mot-clé de ce que l'on a parfois appelé la *nouvelle IA* devient alors l'*embodied cognition*[2], relevant l'importance des interactions physiques (réelles ou simulées) entre le monde et un artefact que l'on voudrait intelligent. Les notions de représentation du monde, de calcul ou de raisonnement laissent alors leur place à des notions comme la structure sensori-motrice, le maintien d'un équilibre interne et la réaction aux perturbations.

On a ainsi passé avec la nouvelle IA — du moins dans le discours — d'un artefact pensant et isolé à une entité agissante, ou *agent*, et ouverte sur son entourage. C'est dans ce mouvement vers l'extérieur de l'agent que l'on peut replacer l'apparition d'un intérêt pour les communications entre agents, leurs interactions, et plus généralement le comportement des systèmes composés de plusieurs agents. L'émergence de la dimension sociale dans le champ de l'IA a alors donné naissance au domaine connu actuellement sous le nom de *systèmes multi-agents* (SMA).

Dans ce passage à la socialité, certains chercheurs sont restés dans la lignée des travaux qui les ont précédés et se sont intéressés à des systèmes formés d'un certain nombre (souvent petit) d'agents complexes, appelés généralement *agents cognitifs*. D'autres en ont profité pour drastiquement simplifier la structure des agents, en renonçant explicitement à leur "intelligence". Ces agents suivent souvent des règles de comportement extrêmement simples et réagissent directement aux changements de leur environnement, sans passer par la phase typiquement cognitive de *délibération*. On parle dans ce cas d'*agents réactifs*[3]. Ces deux types d'approches coexistent actuellement dans la recherche.

L'aile cognitive de la recherche en SMA est donc dans la ligne directe des périodes précédentes de l'IA : elle a hérité des techniques "intellectuelles" de la première période ainsi que des concepts d'*embodiment* de la deuxième ; la problématique sociale vient simplement compléter le tableau.

Cette filiation de l'IA classique est par contre beaucoup moins claire pour l'aile réactive des SMA : si le terme "intelligence" est parfois encore présent dans la notion d'*intelligence collective*, la question centrale est devenue celle des rapports entre deux niveaux de description (local et global) d'un phénomène, et les concepts centraux sont plutôt ceux d'émergence, (auto-)organisation, etc.

Pour résumer de manière imagée, on pourrait dire que l'IA a tout d'abord pris pour métaphore centrale le cerveau humain ; la nouvelle IA a tenté d'élargir cette métaphore au corps humain dans son entier ; l'aile cognitive des SMA a encore élargi cette image à celle des sociétés humaines tandis que l'aile réactive des SMA prendrait plutôt comme métaphore de base celle des sociétés d'insectes (fourmis, abeilles, ...).

Le petit historique ci-dessus ne donne bien sûr qu'une vision très partielle de l'origine des SMA ; de nombreux autres facteurs ont joué dans l'apparition de cette problématique. Ainsi la généralisation des réseaux a provoqué une recherche de décentralisation des algorithmes dans tous les domaines de l'informatique. L'IA n'a pas fait exception et les SMA peuvent être vus comme un des produits de l'approche distribuée de l'IA.

Parallèlement, la formidable évolution de l'ergonomie des ordinateurs dans ces dernières décennies a provoqué une attente toujours plus grande de la part de l'utilisateur en matière de

[2]Il est particulièrement délicat de trouver une bonne traduction de cette expression : la traduction littérale *cognition incarnée* étant peu satisfaisante, nous laissons ici l'expression anglaise d'origine.

[3]Il existe bien sûr toute une variété d'approches intermédiaires entre les agents cognitifs complexes et les agents réactifs purs. La présentation ci-dessus résulte d'une grande simplification.

personnalisation des services qu'une machine peut lui rendre. L'idée de représenter cet utilisateur à l'intérieur même de la machine par une entité informatique à laquelle on peut déléguer des tâches influence considérablement tout un pan de la recherche en SMA actuellement[4].

Une vision plus centrée sur le génie logiciel fournit encore une autre filiation des SMA : de l'apparition des langages de haut niveau au paradigme orienté objet, en passant par la programmation structurée, l'histoire des techniques de programmation peut être vue comme une évolution vers toujours plus de structuration et de modularité. On peut alors voir l'apparition des agents comme un renforcement de l'encapsulation des modèles objets.

Enfin, de manière un peu plus large, on pourrait replacer l'émergence de la problématique multi-agent dans le contexte épistémologique de son époque : elle s'inscrit en effet dans un courant général qui, de toutes les branches touchées par le mouvement systémique [Dur83] jusqu'aux mathématiques elles-mêmes [Ami98], tend à s'intéresser moins aux entités d'un système qu'aux relations qui les unissent.

Une étude détaillée du contexte dans lequel sont nés les SMA dépasserait néanmoins largement la prétention de cette introduction et nous laisserons de côté ces réflexions pour nous intéresser de plus près au questions abordées par cette thèse.

Vers une structuration des SMA

Comme nous l'avons vu, le centre d'intérêt dans la recherche multi-agents s'est progressivement déplacé de l'agent au système dans son entier : de questions du type "Que dois-je ajouter à un agent pour qu'il devienne social ?", on a passé à des problèmes du type "Si je veux un SMA possédant tel comportement dans son ensemble, comment dois-je en programmer les agents ?".

Cette question est encore ouverte ; nous ne possédons pas de recette pour passer de la spécification d'un SMA complet à celle des agents le composant[5]. Néanmoins, plusieurs pistes existent dans cette direction. Parmi celles-ci, il en est une qui nous paraît particulièrement prometteuse et qui constitue une des bases de ce document : c'est ce que nous appellerons l'approche organisationnelle[6].

Cette approche s'applique aux systèmes dont les interactions ont tendance à se stabiliser sur des motifs stables ; on peut alors tenter de "factoriser" la spécification du système par ces *patterns* récurrents d'interaction, obtenant ainsi une étape intermédiaire entre le SMA dans son ensemble et les agents.

On voit ainsi apparaître de nouveaux concepts qui constituent à la fois une factorisation du comportement du SMA et une abstraction de celui des agents. La terminologie décrivant l'abstraction de ces patterns est encore peu stable : on trouve les termes de *groupe*, *organisation*, *structure*, voie même *protocole*, etc. On rencontre par contre un consensus un peu plus large pour appeler *rôle* le comportement qu'un agent doit avoir pour produire ces interactions.

Nous pouvons maintenant expliciter la question qui a initié la recherche menant à cette thèse : "Que doit-il se passer lorsqu'un agent endosse plusieurs rôles dans un système ?" ou

[4] C'est même probablement dans ce domaine que les premières grandes applications commerciales des techniques multi-agents feront leur apparition.

[5] On pourrait d'ailleurs penser que le jour où une méthode générale existera pour ce problème, on sera parvenu à la fin de la recherche en SMA !

[6] Parmi les approches alternatives, on pourrait citer l'émergentisme [Jea97] ou les approches orientées agent [WJK00]. Nous n'aborderons ces travaux que dans la mesure où ils possèdent des intersections avec l'approche organisationnelle.

autrement dit " Les rôles constituant une sorte de spécification partielle du comportement de l'agent, comment combiner ces éléments pour éviter les conflits et obtenir un comportement cohérent, aussi bien au niveau de l'agent que du système ?"

Or nous avons constaté qu'il n'existait pas de cadre formel adéquat pour pouvoir poser cette question précisément. Cette thèse va donc présenter le cadre que nous avons développé pour donner une signification précise à la question de la prise de rôles multiples, puis les méthodes que nous proposons pour la résoudre.

1.2 Contributions

La particularité de ce mémoire est de proposer une approche complète des SMA organisationnels, de la spécification à l'implémentation. Il en résulte un modèle nommé MOCA (pour M*odèle* O*rganisationnel et* C*omponentiel pour les systèmes multi-*A*gents*), dont le développement a nécessité la résolution de différents problèmes :

Modèle organisationnel comportementaliste dynamique Comme nous le verrons au chapitre 2, les approches organisationnelles des SMA se répartissent en deux types : d'une part, celles qui étudient la dynamique des structures sociales sans vraiment associer de comportements aux rôles qu'un agent peut jouer et d'autre part celles qui associent un comportement aux rôles mais qui ne permettent pas une dynamique sociale. Notre première contribution, détaillée principalement aux chapitres 5 et 7, est de proposer un modèle opérationnel qui combine ces deux aspects.

Architecture componentielle Pour réaliser la combinaison ci-dessus de manière souple et générale, nous avons choisi un modèle componentiel d'agent. Nous explicitons ainsi les liens présentis dans certains travaux entre les approches componentielles et organisationnelles [Yoo99]. Ceci nous permet également de donner une sémantique précise à la prise et au rejet de rôles sous forme d'ajout ou de retrait d'un composant de l'agent. Cet aspect apparaîtra plus particulièrement aux chapitres 5 et 6.

Prise de rôle multiple Question initiale de la thèse, la combinaison dynamique de plusieurs rôles au sein d'un agent sera traitée au chapitre 6. Nous introduirons le concept d'*ignorance mutuelle*, que nous proposons de réaliser par une exclusion mutuelle sur certaines parties de l'exécution des rôles. Si les rôles sont décrits dans le formalisme que nous proposons au chapitre 5, la détermination de ces *zones critiques* peut se faire automatiquement.

Opérationnalisation Tous les éléments ci-dessus sont opérationnalisés sous la forme d'une plate-forme qui sera présentée au chapitre 9. Des tests réalisés sur cette plate-forme seront exposés au chapitre 10.

Éléments de validation La validation de SMA est un problème particulièrement délicat : si le système est conçu de manière centralisée, dans une approche "top-down", il est possible de récupérer certains éléments des méthodes traditionnelles de validation. Par contre, on perd une bonne partie de la souplesse que peut fournir un SMA. Cette souplesse est beaucoup plus grande dans une approche de type "bottom-up", centrée sur les agents, mais la validation formelle est très difficile à réaliser dans ce cas. Notre approche se situe quelque part entre ces deux pôles et pourrait être décrite comme une approche "*tops-down*", ou *multi-centrée* : les organisations que nous définissons sont autant de vues partielles du système global, autant de "tops" desquels descendre. Nous argumenterons

au chapitre 8 que cette vision des choses fournit des pistes intéressantes pour ce qui est de la validation.

1.3 MOCA, simplement un modèle de plus ?

On trouve déjà dans la littérature de nombreux modèles de SMA, souvent accompagnés de leur plate-forme, présentant chacun ses qualités et ses défauts propres. On peut donc sérieusement se demander si MOCA n'est pas simplement un modèle de plus, implémenté sous la forme d'une plate-forme de plus. La réponse est à la fois oui et non, suivant le point de vue :

- Pour un utilisateur désirant concevoir un système, la réponse est oui : MOCA fournit un nouvel environnement de développement, avec de nouveaux concepts à maîtriser. De ce point de vue, la seule justification de notre approche est que nous pensons apporter avec MOCA des avantages en termes de puissance, de réutilisation ou de facilité de conception.
- À un niveau plus théorique, MOCA est assez différent de la plupart des autres approches sous-tendant une plate-forme : en effet, il ne s'agit pas vraiment d'une plate-forme multi-agent mais plutôt d'une *plate-forme organisationnelle*. La plate-forme MOCA s'appuie sur une plate-forme SMA existante[7] pour fournir un nouvelle couche de services, dits *organisationnels*.

Pour mieux saisir l'apport de notre démarche en matière de conception de SMA, on pourrait établir une comparaison avec la programmation orientée objet. Dans certains cas, le paradigme objet n'est pas approprié et une programmation "traditionnelle" est tout aussi efficace ; cependant, la structuration et la réutilisabilité apportées par la programmation orientée objet l'ont rendue indispensable lors du développement de systèmes de grande taille. Notre approche a également pour but de structurer le développement d'un SMA et d'augmenter la réutilisabilité de ses composants. Les techniques que nous proposons ne sont certainement pas utiles dans toutes les circonstances, mais nous pensons qu'une telle approche pourrait aider à développer des SMA à large échelle beaucoup plus facilement.

1.4 Plan de la thèse

La première partie de cette thèse présente le contexte dans lequel s'inscrivent nos travaux. Après la présente introduction, le chapitre 2 propose une revue de la littérature pour permettre de mieux cerner les spécificités de notre approche. Le chapitre 3 présente ensuite le formalisme de représentation des rôles de Vincent Hilaire [Hil00] sur lequel nous basons la partie formelle de notre approche.

La deuxième partie est consacrée aux aspects théoriques de notre modèle. Le chapitre 5 présente les concepts principaux de MOCA et leur articulation. Le chapitre 6 se concentre ensuite sur les interactions entre les composants constituant un agent et la gestion des conflits qui pourrait en résulter. Le chapitre 7 utilise les concepts développés par les deux chapitres précédents pour proposer une méthode de gestion de la dynamique organisationnelle. Enfin, le chapitre 8 suggère des pistes pour la validation formelle en mettant à profit les caractéristiques de notre approche.

[7] En l'occurrence MadKit, cf. chapitres 2 et 9.

La troisième partie présente les réalisations pratiques basées sur notre modèle. Le chapitre 9 présente la plate-forme que nous avons développée pour opérationnaliser MOCA, alors que le chapitre 10 expose les expérimentations réalisées sur cette plate-forme.

La quatrième et dernière partie est constituée uniquement du chapitre 11 qui tente de faire le point sur les travaux que nous avons effectués et ceux qui, inévitablement, restent à faire.

Chapitre 2

Revue de la littérature

L'approche proposée dans cette thèse se situe à la rencontre de deux courants actuels des systèmes multi-agents : les *modèles organisationnels* et les *modèles componentiels*. Si ces deux domaines semblent assez éloignés l'un de l'autre, on remarquera cependant qu'ils partagent au moins un but central : augmenter la réutilisabilité des éléments d'un système multi-agents.

Dans notre approche cependant, ces deux domaines ont des positions très distinctes : l'approche organisationnelle constitue le point de départ de notre réflexion, le domaine où la question initiale a été posée. Le choix d'une architecture componentielle est venu plus tard, comme un *moyen* de répondre à certains problèmes posés par cette première question.

Cette différence de statut se reflète dans ce survol de la littérature : la présentation des travaux organisationnels du prochain paragraphe se veut critique, relevant les manques des approches actuelles pour bien faire comprendre le pourquoi de notre démarche. Le paragraphe suivant, par contre, présentera les travaux componentiels de manière beaucoup plus factuelle et succincte.

2.1 Approches organisationnelles

Au cours des dernières années, les notions d'*organisation*, de *rôle* et d'autres concepts apparentés ont fait leur apparition dans le domaine des systèmes multi-agents, et ceci pour au moins deux raisons :

1. Ce type d'approche peut résulter d'un resserrement des liens qui unissent SMA et sociologie. Que ce soit pour tester des modèles existants ou pour tenter d'en développer d'autres, la simulation multi-agent semble un outil particulièrement bien adapté à la sociologie [CEMS01] et il n'est pas étonnant que des notions évoquant des collectifs d'agents fassent leur apparition dans ce contexte.
2. D'un point de vue génie logiciel, la conception de systèmes multi-agents à large échelle nécessite des moyens de structuration. Si les agents sont homogènes et que le système est fermé, cette structuration peut se faire de manière implicite dans la conception des agents eux-mêmes ; cependant, dans un système ouvert et hétérogène, la représentation explicite de cette structuration devient nécessaire [Gut01].

Notre démarche se situe clairement dans l'optique *génie logiciel* évoquée au deuxième point. Cependant, les deux types de démarche ne sont pas si éloignés en pratique et des retombées sur l'aspect simulation sociale d'approches telles que la nôtre sont tout à fait possibles [AMBN].

Indépendamment de cette première classification, une deuxième grande séparation dans les approches organisationnelles des SMA peut être dégagée :

1. D'un côté, les approches dites *mentalistes*, qui mettent l'accent sur le fonctionnement interne des agents [PNJ99, Ken99, BGM98]. Les notions organisationnelles sont alors généralement normatives : faire partie d'un groupement social implique essentiellement d'adapter son comportement de manière à suivre les normes y étant rattachées. Les rôles sont vus comme des ensembles de responsabilités, dépendances, etc. D'un point de vue technique, ces approches sont généralement des extensions de l'architecture BDI (Belief-Desire-Intention[1], voir p. ex. [PNJ99])
2. De l'autre côté, les approches *comportementalistes* s'intéressent aux groupements sociaux en tant que *patterns*[2] institutionnalisés d'interactions. Les notions organisationnelles décrivent alors généralement de tels *patterns*, indépendamment de la structure interne de l'agent.

Le présent travail se situe dans une perspective comportementaliste.

Nous allons maintenant proposer une présentation des principales approches organisationnelles comportementalistes des SMA existant dans la littérature. Nous tenterons de relever, pour chacune d'entre elles, ses motivations, ses forces et ses faiblesses et le positionnement relatif de notre approche.

La thèse de Benoît Durand

Un des premiers textes à aborder le problème de l'organisation dans les SMA sous l'angle de patterns collectifs de comportements est la thèse de Benoît Durand [Dur96]. L'idée de cette approche, née d'une problématique de modélisation et simulation des systèmes complexes, est d'associer à chaque *point de vue* sur un système un *schéma d'organisation*, c'est à dire un ensemble de rôles conçus pour s'exécuter de manière coordonnée.

Le modèle de Durand se décline en trois niveaux dont les relations sont représentées à la figure 2.1 :

1. Le *niveau structurel* représente l'ancrage du système dans une approche orientée objet ; il consiste en la description des classes et méthodes qui forment le système. Pour le spécifier, Durand a développé un *framework* exprimé dans le langage *Airelle*. C'est en particulier à ce niveau que sont décrites les différentes *classes d'agents*.
2. Le *niveau des compétences* définit les schémas d'organisation et les rôles correspondants ; il constitue le noyau de l'aspect organisationnel du système.
3. Le *niveau performatif* spécifie les liens qui unissent les deux niveaux précédents ; c'est à ce niveau qu'apparaissent les agents eux-même, ainsi que la structure organisationnelle du système (attribution des rôles aux agents).

Pour Durand, un *rôle* décrit une activité possible pour un agent. À chaque rôle est associé un ensemble d'états, et à chaque état correspondent trois comportements : le *comportement proactif*, représentant l'activité du rôle indépendamment des influences extérieures, le *comportement réactif*, qui spécifie la façon dont le rôle réagit aux perturbations qu'il reçoit et le *comportement fonctionnel*, déclenché sur demande par ses accointances.

[1] Croyance-Désir-Intention.

[2] Nous avons décidé, afin de nous conformer à l'usage, d'utiliser tout au long de cette thèse le mot anglais de *pattern* au lieu de *motif*, qui n'en constitue qu'une traduction approximative.

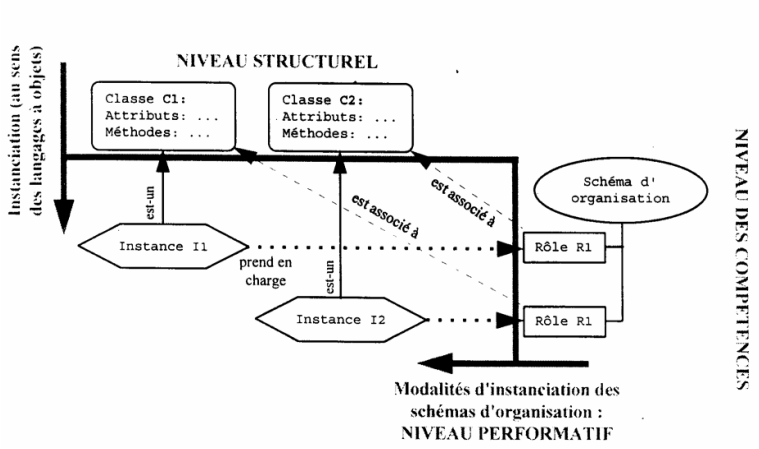


FIG. 2.1 – Les trois niveaux du modèle de Durand

Un *comportement* est lui composé d'un *script comportemental* et d'un ensemble de *liens d'accointances*. L'exécution d'un comportement correspond à l'exécution du script associé, suivie du déclenchement d'actions ou de comportements sur les accointances. On notera que l'exécution des comportements se repose sur le fait que les agents doivent fournir un certain nombre de services standard (par exemple pour la gestion des accointances).

Les *liens d'accointances*, on l'a vu, sont spécifiés au niveau des comportements et servent à désigner les agents avec lesquels un comportement interagira. Ils sont définis par un identifiant, une expression permettant de calculer la liste effective des accointances au moment de l'exécution, la classe des agents aux deux extrémités du lien, l'arité et éventuellement le lien inverse. Lorsque le lien sert au déclenchement du comportement fonctionnel de l'accointance, il sera appelé *lien fonctionnel*, et les rôles ainsi liés seront nommés *rôles associés*.

L'attribution des rôles aux agents est effectuée par le développeur au moment de la conception ; elle doit tenir compte de la classe d'agent (une classe donnée ne peut prendre en charge que certains rôles) et des autres rôles pris par l'agent (en raison des rôles associés). Cette attribution reste fixe pour la durée de vie du système.

Un des grands mérites de Durand est d'avoir mis en évidence le double lien d'instanciation nécessaire à une approche organisationnelle des SMA :

- d'une part, le lien qui unit un agent à sa classe ; il correspond généralement à une instanciation au sens orienté objet du terme.
- d'autre part, le lien qui unit une organisation du SMA à son schéma d'organisation, et dont l'interprétation précise est plus sujette à discussion.

Cet aspect se retrouvera d'ailleurs dans la plupart des travaux que nous citons ici, bien que le premier type d'instanciation soit parfois laissé implicite dans le discours. On obtient alors une structuration en deux niveaux : le niveau qui décrit de manière abstraite les patterns d'interactions et celui où des agents les mettent en oeuvre.

Cependant, le travail de Durand souffre de quelques imperfections :

- Les formalismes utilisés sont très près de l'implémentation, nuisant ainsi à la généricité du modèle.
- Les bénéfices de la structuration en niveaux sont atténués par une mauvaise indépendance de ceux-ci : la conception des classes d'agents nécessite une connaissance des

schémas d'organisation et inversement. Ceci nuit beaucoup à la réutilisation des agents aussi bien que des structures organisationnelles.
- Bien qu'un agent puisse prendre plusieurs rôles, il n'existe aucun mécanisme pour gérer les éventuelles interférences qui pourraient survenir entre ceux-ci.
- Enfin, l'absence de dynamique organisationnelle représente une limitation sérieuse du champ d'application du modèle. En effet, cela interdit l'entrée ou la sortie d'agents dans le système en cours d'exécution, aussi bien que le changement de fonction d'un agent.

Le modèle que nous proposons dans ce travail présente une parenté certaine avec le travail de Durand pour ce qui est de la structuration en niveaux, mais apporte des améliorations sur les points suivants : un formalisme plus général, une bonne indépendance des niveaux conceptuels et la prise en compte des conflits de rôles et de la dynamique organisationnelle. De plus, l'introduction dans notre modèle de la notion de *compétence* nous permet de généraliser les notions de *comportement fonctionnel*, de *rôles associés* et de *services d'agents* ; le regroupement de ces divers aspects en un seul concept en permet une expression à la fois plus claire et plus simple.

Cassiopée

La méthode *Cassiopée* [CPD96] se présente comme un cadre méthodologique pour la conception de SMA permettant la mise en oeuvre de comportements collectifs.

Dans *Cassiopée*, les agents sont dotés de trois niveaux de comportement :

1. Les *comportements élémentaires*, correspondant aux capacités de base de l'agent (les "primitives").
2. Les *comportements relationnels*, permettant la coordination entre agents et la mise en place de comportements collectifs.
3. Les *comportements organisationnels*, constituant une sorte de niveau "méta" par rapport aux précédents, dans le sens où ils gèrent la mise en œuvre ou l'inhibition de ces derniers.

La méthodologie se décompose alors en trois points correspondant au développement de ces trois types de comportements :

1. Expression des comportements élémentaires. Cette étape comporte aussi la définition des *types d'agents* qui comportent chacun un certain nombre de comportements élémentaires.
2. Étude des dépendances comportementales. Pour cela, on construit le *graphe de couplage* qui représente toutes les dépendances possibles entre les différents comportements élémentaires. Ce graphe est ensuite projeté sur chaque type d'agent, puis le résultat est épuré en retirant les dépendances jugées non pertinentes ; on obtient alors un *graphe d'influence* dont les chemins et circuits représentent les différentes possibilités de regroupement des agents.

   Il reste alors à décentraliser cette vue structurelle au niveau des agents : chaque type d'agent *influent* doit pouvoir envoyer un *signe d'influence* aux types d'agents *influencés* correspondants. Les comportements relationnels sont ensuite définis au niveau de l'agent influencé pour décrire leur réaction (en termes de comportements élémentaires) à la réception d'un signe d'influence.
3. Définition de la dynamique de l'organisation. Lorsque des agents collaborent selon les mécanismes mis en place au point 2, on dit qu'ils forment un groupe. Il faut maintenant

définir des *comportements de formation de groupes*, qui permettent par exemple de déterminer quel groupe former lorsque plusieurs possibilités concurrentes se proposent, des *comportements de participation*, qui déterminent le comportement d'un agent suivant s'il appartient ou non à un groupe donné, et enfin des *comportements de dissolution*, qui déterminent la manière de dissoudre un groupe existant.

La distinction entre les différents niveaux de comportement proposée par *Cassiopée* permet une expression beaucoup plus claire de ce qui, chez Durand, est disséminé dans les concepts de service d'agent, de comportement fonctionnel et dans les notions associées.

Par contre, l'absence du double lien d'instanciation que nous avons vu chez Durand condamne *Cassiopée* à un certain flou qui provoque une quasi-superposition des notions de type d'agent, de comportement élémentaire et de rôle.

Nous verrons que notre modèle intègre, avec la notion de *compétence*, un concept proche des comportements élémentaires de *Cassiopée* dans un cadre explicitant également la double instanciation. On notera également que dans *Cassiopée*, les comportements élémentaires de l'agent sont fixés pour sa durée de vie, tandis que dans MOCA, un agent peut acquérir ou perdre des compétences.

Aalaadin

Aalaadin [FG98, FG99, Gut01] est sans doute la plus connue des approches organisationnelles comportementalistes des SMA. Menée par Olivier Gutknecht et Jacques Ferber, cette démarche s'articule selon quatre axes :

1. Un axe conceptuel présentant un modèle s'articulant autour des trois concepts d'*agent*, de *groupe* et de *rôle*.
2. Un axe proposant une sémantique formelle au modèle Agent-Groupe-Rôle.
3. Un axe implémentatoire fournissant la plate-forme *MadKit*.
4. Un axe étudiant des pistes pour une méthodologie de conception des SMA organisationnels.

Cette approche trouve son origine dans une réflexion sur les différents écueils des modèles centrés agents. Le principal d'entre eux est la prise en compte de l'*hétérogénéité* : "Comment concevoir des systèmes ouverts, aux agents très différents les uns des autres, parlant éventuellement plusieurs "langages" différents, etc. ?" Les auteurs relèvent aussi les problèmes de la modularisation du développement des SMA, la sécurité des applications, etc.

La solution proposée pour surmonter ces diverses difficultés est d'introduire une structuration exprimable au niveau du système multi-agent et non de l'agent. Nous allons maintenant présenter le modèle qui en résulte selon les quatre axes évoqués ci-dessus :

Le modèle Agent-Groupe-Rôle Centré autour des trois concepts d'*agent*, de *groupe* et de *rôle*, il est parfois appelé *modèle AGR*. Les rapports entre ces trois notions sont présentés sous forme de diagramme UML à la figure 2.2. [Gut01] en fait le commentaire suivant :

> « Un agent peut intervenir dans plusieurs communautés (que nous appellerons "groupes" dans notre modèle) en parallèle. Il peut jouer dans chacun de ces groupes un ou plusieurs rôles correspondant à ses activités ou interactions. Ces rôles peuvent être portés par un nombre d'agents arbitraire, dépendant de la situation et des normes de l'organisation. »

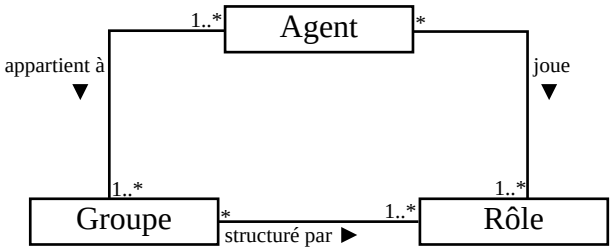


FIG. 2.2 – Les notions centrales d'Aalaadin

La simplicité du modèle AGR résulte de la recherche d'une certaine minimalité : le but est de dégager un ensemble minimal de concepts permettant un discours sur les SMA organisationnels.

La sémantique opérationnelle Le modèle AGR est muni d'une sémantique opérationnelle basée sur la rencontre de deux formalismes :

1. Le π-calcul, qui est un formalisme visant à fournir une fondation formelle à l'expression de processus concurrents. En caricaturant un peu, on peut le décrire comme le λ-calcul du parallélisme.
2. La *Chemical Abstract Machine* (CHAM en abrégé), comme son nom l'indique, est une machine abstraite basée sur une métaphore chimique. Une CHAM est définie par les *molécules* (éléments syntaxiques) qui la composent et par un ensemble de *règles de réaction* qui définissent des transformations d'un ensemble de molécules en un autre.

L'idée est alors de faire correspondre une expression en π-calcul à chacun des comportements d'un agent, et une CHAM à chacun des groupes du SMA[3]. Le résultat, appelé MAAM (pour Multi-Agent Abstract Machine), fournit une machine abstraite décrivant le fonctionnement du SMA dans son ensemble. Le cloisonnement effectué par la répartition dans des CHAM différentes permet de bien séparer le fonctionnement des différents groupes. Des primitives sont en outre définies pour créer des nouvelles solutions et démarrer un processus dans une CHAM donnée, ce qui permet d'exprimer la création de nouveaux groupes et leur gestion.

MadKit Le modèle AGR est opérationnalisé par une plate-forme multi-agent nommée MadKit (pour Multi-Agent Development Kit) [Mad]. Cette plate-forme, réalisée en Java, est organisée autour d'un micro-noyau fournissant les services indispensables (gestion des groupes et rôles locaux, gestion du cycle de vie des agents, passage de message local et observation de l'exécution). Tous les autres services sont *agentifiés*, c'est à dire qu'ils sont fournis par des agents. Le résultat de cette conception est une grande souplesse d'utilisation ainsi que des possibilités considérables d'adaptation à une situation particulière (mémoire réduite, exécution distribuée, etc.).

On remarquera que dans MadKit, les groupes et les rôles ne sont que des étiquettes, qui permettent notamment l'adressage des messages. Au niveau de la plate-forme, les

[3]Cette seconde correspondance n'est pas vraiment stricte : on doit aussi associer une CHAM à chaque agent pour ses communications internes, et ajouter une "solution originelle" pour mettre les comportements non associés à des rôles.

rôles ne sont donc pas associés à des comportements particuliers, même si cela est généralement réalisé tout de même par la conception de l'agent. Par contre, la structure organisationnelle du système est dynamique : l'apparition et la disparition d'agents, la prise, le changement et l'abandon d'un rôle sont possibles en cours d'exécution.

Aspects méthodologiques Pour [Gut01], l'enjeu à long terme d'une méthodologie organisationnelle des SMA est de

> « dégager des *patterns de conceptions* organisationnels, c'est à dire des descriptions *éprouvées* d'organisations classiques (réseaux contractuels, organisations hiérarchiques, structure de représentant, ...) pour en faciliter la réutilisation. »

Pour aller dans cette direction, Gutknecht propose une méthodologie en cinq phases :

1. Identification des groupes. Le seul prérequis strict est l'existence au sein du groupe potentiel d'un mécanisme de communication commun, le reste étant lié à des considérations particulières au contexte.
2. Choix ou conception d'un modèle organisationnel spécifique. Cette étape correspond à l'identification des rôles au sein des groupes dégagés au point 1.
3. Spécification de la structure organisationnelle. [Gut01] propose une structure de document XML permettant de définir les rôles, leur arité, leurs relations, etc. On peut aussi introduire des dépendances entre les rôles : certains rôles ne peuvent être pris que par un agent possédant déjà un autre rôle.
4. Description des schémas d'interaction. Les protocoles d'interaction entre les rôles peuvent être spécifiés par des diagrammes de séquence UML, des réseaux de Petri, des scenarii ACL ou KQML, etc.
5. Définition des architectures individuelles. C'est à cette dernière étape que le lien est effectué avec les approches centrées agent : il s'agit de choisir une architecture interne pour les agents qui soit compatible avec la spécification organisationnelle qui précède.

On notera que cette méthodologie a fait apparaître un certain nombre de concepts extérieurs au modèle AGR de base (modèle et structure organisationnels, schémas d'interaction, etc.). Ces concepts sont qualifiés d'*abstraits*, par opposition aux trois concepts *concrets* d'agent, de groupe et de rôle. On retrouve donc ici, sous forme un peu plus implicite, la structuration entre un niveau descriptif et un niveau exécutif mise en évidence par Durand et qui se trouve au centre de notre approche.

La minimalité du modèle AGR est à la fois sa principale force et sa première faiblesse : une force car le côté peu contraignant de ce modèle en permet l'adoption par de nombreuses personnes qui l'acceptent comme noyau de leur approche, et une faiblesse car l'expressivité du modèle pur est tout de même relativement limitée. Nous verrons donc ci-dessous quelques propositions d'extensions du modèle Aalaadin.

La plate-forme MadKit hérite de ce double aspect : si elle est assez fréquemment utilisée, il semblerait que ce soit autant pour la souplesse qu'elle permet au niveau de l'architecture des agents que pour l'utilisation du modèle AGR lui-même. Ceci est probablement dû en partie au fait que la structuration en groupes sert essentiellement de système d'adressage des agents,

mais que le mécanisme permettant l'attribution d'un comportement à un rôle est entièrement laissé à la charge du développeur.

La sémantique formelle d'Aalaadin est à notre connaissance la seule approche formelle permettant la description complète d'un SMA organisationnel. Cette approche d'une grande rigueur souffre toutefois d'un formalisme relativement lourd à manipuler ; en particulier, on constate que les agents possèdent une représentation complètement éclatée (essentiellement un jeu d'équations partageant un paramètre formel). La preuve reste donc encore à faire que ce formalisme soit réellement utilisable, que ce soit pour la spécification ou la vérification de SMA.

Pour ce qui est de l'aspect méthodologique, on remarquera que Gutknecht et Ferber proposent plus des *lignes de conduite* que de réelles solutions. Une bonne partie de la présente thèse peut être vue comme une tentative d'apporter des solutions concrètes compatibles avec ces lignes méthodologiques. Cependant, nous nous démarquerons de l'approche ci-dessus en un point important : le modèle que nous proposons n'admet pas l'expression directe de dépendances entre rôles. En effet, celles-ci réduisent la réutilisabilité des patterns organisationnels. Nous proposerons par contre avec la notion de *compétence* un mécanisme plus général, permettant l'expression de ce type de dépendance sans briser la modularité de l'approche.

Parunak et Odell

La minimalité caractérisant l'approche Aalaadin a poussé divers auteurs à proposer des compléments ou extensions. En particulier, H. Van Dyke Parunak et J. Odell proposent dans [PO01] trois extensions au modèle AGR :

Contenu des rôles Les rôles ne doivent plus être considérés comme de simples éléments de nommage dans un groupe, mais doivent être liés à un ensemble de dépendances et de comportements récurrents.

Environnement Un groupe doit être défini non seulement par ses rôles, mais également par l'environnement dans lequel ceux-ci interagissent.

Dimension holonique Il doit être possible pour un groupe de prendre un rôle dans un autre groupe, alors qu'Aalaadin ne l'autorise que pour les agents. Ceci permettrait de voir un groupe tantôt comme une entité atomique et tantôt comme une entité composée selon les besoins. C'est ce que [PO01] appelle la *dimension holonique*.

Ces différentes propositions sont soutenues par des représentations sous forme de diagrammes UML. Par exemple, la deuxième proposition ci-dessus (intégration de l'environnement) peut être représentée par le diagramme de classes de la figure 2.3.

En plus de ces propositions de fond, [PO01] suggère l'utilisation de différents diagrammes UML pour la spécification des divers aspects d'un SMA organisationnel : les lignes de nage (*swimlanes*) pour la structure organisationnelle, les diagrammes de séquence ou d'activité pour le contenu des rôles, etc.

Les propositions d'extension d'Aalaadin faites par [PO01] nous semblent tout à fait pertinentes :

- La première, concernant le lien entre un rôle et un *pattern* interactionnel, est un des sujets centraux de cette thèse.
- La deuxième, concernant l'environnement, pose un problème de taille : si la spécification d'un groupe inclut celle de son environnement, le système doit être prêt à gérer plusieurs environnements (pas forcément indépendants) simultanément. Ce problème est

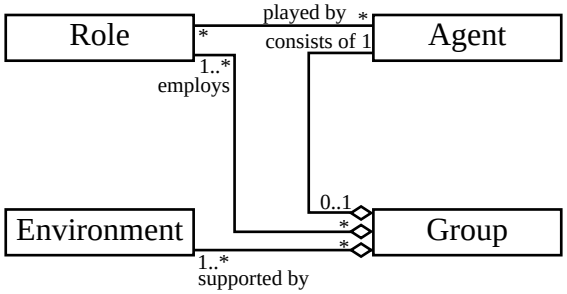


FIG. 2.3 – Intégration de la notion d'environnement dans le modèle AGR

loin d'être résolu en pratique, mais on trouvera des pistes intéressantes à ce sujet dans [Sou01].

- La troisième extension, concernant l'inclusion d'une dimension holonique, semble particulièrement difficile à intégrer au modèle Aalaadin. A notre connaissance, personne n'est actuellement en mesure de proposer une sémantique opérationnelle pour une approche à la fois organisationnelle, comportementaliste et holonique des SMA. Des pistes pourraient toutefois être explorées dans un rapprochement avec des approches récursives des SMA (cf. par exemple [Cor01] ou [Aek99]).

Nous émettons par contre quelques réserves en ce qui concerne les propositions de formalismes :

- Les formalismes proposés pour la représentation des rôles (diagrammes d'activité et de séquence) ne nous semblent pas posséder une expressivité suffisante. Nous proposons avec MOCA l'utilisation d'un autre formalisme UML, les statecharts, qui nous semble plus approprié.
- Une représentation sous forme de diagramme de classe, telle celle de la figure 2.3 permet une description précise des relations entre les différents concepts ; cependant, rien n'est précisé sur la sémantique des relations elles-mêmes. Ainsi par exemple, la relation "played by" qui unit un rôle à un agent n'est pas explicitée. On aura pourtant l'occasion de se rendre compte au cours de cette thèse que la sémantique de cette relation est loin d'être triviale...
- La représentation de la structure organisationnelle sous forme de lignes de nage ne permet que la description d'une structure statique. Ceci représente donc un pas en arrière important par rapport à la souplesse d'Aalaadin.

De ces travaux, nous retiendrons donc plutôt les innovations conceptuelles que les propositions de formalisation, qui ne nous semblent pas encore mûres.

La thèse de Vincent Hilaire

Dans sa thèse [Hil00], Vincent Hilaire se base également sur le modèle AGR ; cependant, il se concentre particulièrement sur une formalisation des concepts qualifiés d'*abstraits* par Gutknecht et Ferber.

Le coeur de la thèse est constitué par le développement d'un *framework* pour les SMA organisationnels. Hilaire utilise un formalisme intégrant les statecharts et Object-Z pour définir des classes génériques d'agent, de rôle, d'organisation et de quelques autres concepts liés.

Il est ensuite possible de définir par héritage des organisations, groupes et agents parti-

culiers formant une spécification exécutable d'un SMA complet. Ce processus, permettant la simulation ou la vérification de la spécification, est envisagé comme faisant partie de la phase de conception ; il doit être suivi d'une phase de réalisation dans laquelle on choisira le langage d'implémentation, l'architecture des agents, le mode de communication, etc. Cette phase n'est cependant pas abordée dans les travaux d'Hilaire.

Le formalisme de représentation des rôles proposé par Hilaire est le plus complet qui existe actuellement ; il permet de spécifier de manière intuitive et confortable aussi bien les aspects réactifs que fonctionnels du comportement d'un rôle. Nous en proposerons une exposition détaillée au chapitre 3 avant de l'utiliser dans notre approche.

De manière plus large, l'approche d'Hilaire présente cependant quelques faiblesses :

- Bien que ceci ne soit pas clairement exprimé, l'approche semble se limiter à des systèmes dont la structure organisationnelle est fixe. En tout cas, l'apparition ou la disparition d'agents ou d'organisations ainsi que les changements de rôles ne sont jamais évoqués.
- La représentation des agents est particulièrement pauvre : un agent se réduit à un ensemble de rôles. Ses capacités et ses attributs se limitent à ceux définis dans ses rôles et son comportement est entièrement défini par ces derniers.
- Bien qu'un agent puisse prendre plusieurs rôles simultanément, aucune réflexion n'est proposée quant à la gestion d'éventuelles interférences entre eux.

Dans cette thèse, nous allons donc reprendre le formalisme de représentation des rôles d'Hilaire, mais en l'intégrant dans un cadre beaucoup plus souple permettant une évolution de la structure organisationnelle, redonnant à l'agent une fonction plus centrale et traitant les possibles conflits de rôles. De plus, l'approche d'Hilaire s'arrête à la spécification alors que notre démarche permet la réalisation d'un SMA complet.

Gaia

L'approche *Gaia* [WJK00] est basée sur la constatation que les techniques classiques de génie logiciel, notamment les approches orientées objet, ne sont pas appropriées à une programmation orientée agent. En particulier, les approches classiques

> « fail to adequately capture an agent's flexible, autonomous problem-solving behaviour, the richness of an agent's interactions, and the complexity of an agent system's organisational structure[4]. »

Les auteurs proposent donc une méthodologie de conception de SMA basée sur une décomposition en deux niveaux (figure 2.4) : le niveau *abstrait* contient notamment les notions de *rôle* et d'*interaction* et correspond à l'étape d'analyse ; le niveau *concret* contient les notions classiques de *type d'agent*, de *service* et d'*accointance* et correspond à la conception.

Le niveau abstrait contient un *modèle de rôles* et un *modèle d'interactions* :

1. Le *modèle de rôles* décrit les différents rôles du système. Un *rôle* est défini par quatre éléments :

   (a) *Responsabilités*. Représentant ce que l'agent doit être capable d'assurer dans le système, elles sont divisées en deux classes, les *propriétés de vivacité* et les *propriétés de sûreté*, avec les significations habituelles de ces concepts. Les premières sont

[4] "ne parviennent pas à représenter de manière adéquate le comportement flexible et autonome d'un agent résolvant un problème, ni la richesse de ses interactions, pas plus que la complexité des structures organisationnelles d'un système d'agents."

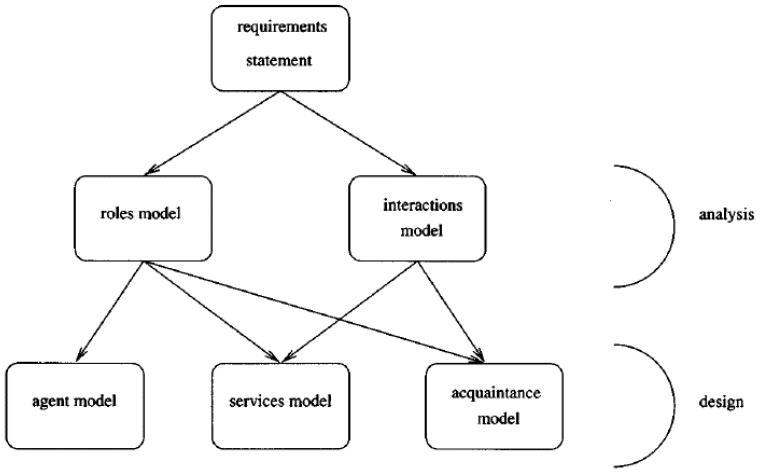


FIG. 2.4 – Les principaux concepts de Gaia

exprimées sous forme d'expressions régulières dont les éléments constitutifs sont des activités ou des protocoles ; les secondes sont quant à elles exprimées par une liste de prédicats.

(b) *Permissions.* Elles représentent les ressources auxquelles le rôle a accès et consistent essentiellement en la liste des valeurs que le rôle a le droit de lire ou de modifier.

(c) *Activités.* Décrivant les calculs pouvant être effectués par l'agent sans interaction avec l'extérieur, elles sont à ce stade considérées comme des éléments atomiques.

(d) *Protocoles.* Il s'agit ici de simples liens vers les protocoles définis dans les modèles d'interaction.

2. Le *modèle d'interaction* définit quant à lui une liste de protocoles ; décrivant les communications possibles entre les rôles, ils sont définis par un initiateur, un interlocuteur, des entrées, des sorties ainsi qu'une description textuelle sur le type d'interaction et son déroulement.

La phase d'analyse consiste en un aller-et-retour entre ces deux modèles pour obtenir un ensemble cohérent. Elle est suivie d'une phase de conception dont le but est de ramener la description du système à un niveau d'abstraction suffisamment bas pour que les techniques traditionnelles de conception puissent être employées. Cette phase consiste en la mise au point :

1. d'un *modèle d'agent* ; celui-ci est constitué de types d'agents, lesquels consistent en un ensemble (généralement petit) de rôles.
2. d'un *modèle de services,* décrivant comme on peut s'y attendre les services fournis par chaque type d'agents. La spécification des services est dérivée des protocoles, activités et responsabilités des rôles correspondants.
3. d'un *modèle d'accointance*, décrivant les liens de communication existant dans le système sous la forme d'un graphe orienté.

Si Gaia est probablement l'approche méthodologique la plus complète que nous présentons ici, c'est aussi la moins profondément organisationnelle : toute trace de concept organisationnel disparaît déjà dans la phase de conception. Ceci est à contraster avec Aalaadin, ou les concepts abstraits "laissent des traces" (sous la forme d'une structuration en groupes) jusqu'à l'implémentation et même jusqu'à l'exécution.

On relèvera d'ailleurs que le concept même d'*organisation* est absent de cette approche : les rôles ne sont reliés que par un réseau non structuré d'interactions. Ceci interdit presque complètement le développement d'entités organisationnelles réutilisables et limite le niveau de complexité atteignable par une telle approche.

Opéra

La thèse d'Arnaud Dury [Dur00] a pour but de développer

> « un modèle permettant de spécifier des interactions dans un SMA, et permettant de rendre cette définition opérationnelle en instanciant effectivement les interactions ainsi définies au sein d'un système multi-agents donné. »

Pour ce faire, il propose le modèle *Opéra*, qui est centré sur la notion de *rôle* comme "participation locale à une interaction".

Ici, un rôle est défini par :

- un *arbre binaire d'actions*, dont chacun des noeuds correspond à une action. Le parcours de l'arbre est déterminé à chaque pas par la réussite ou l'échec de l'action entreprise.
- une *situation déclenchante*, exprimée en termes d'états des capteurs de l'agent.

Un agent prend un rôle donné lorsque sa situation déclenchante est vérifiée. Il va alors exécuter les actions correspondantes. Comme ceci peut amener l'agent à exécuter plusieurs rôles à la fois, Dury propose un *modèle de composition dynamique des comportements* : l'agent dispose d'une pile de comportements, dont seul le sommet est actif. Lorsqu'un nouveau comportement activable est jugé plus prioritaire, il sera rajouté au sommet de la pile, interrompant l'exécution en cours. Lors du dépilement, le comportement suivant peut recommencer.

Ce modèle a été implémenté sous forme d'une API java conçue pour pouvoir être intégrée à un modèle d'agent préexistant.

Le modèle *Opéra* constitue une des seules approches actuellement qui propose une description des rôles en tant que comportements récurrents tout en permettant une dynamique organisationnelle. Malheureusement, il souffre de plusieurs faiblesses par rapport aux modèles évoqués ci-dessus :

- La prise de rôle résulte de l'occurrence d'une situation déclenchante plutôt que d'un choix délibéré de l'agent.
- Le modèle de comportement est particulièrement pauvre : il n'existe pas d'autre possibilité de branchement que l'échec d'une action.
- Certains rôles sont conçus pour fonctionner ensemble, mais cette connaissance reste implicite : aucune notion d'organisation n'est présente pour structurer le système.

Une autre spécificité d'*Opéra* est la prise en compte de la composition des rôles au moment de l'exécution. Si la question a le mérite d'être abordée, la réponse est par contre relativement limitée. En effet le modèle proposé, basé sur une composition par piles, comporte plusieurs désavantages :

- Un seul comportement peut s'exécuter à un moment donné, alors que certains comportements ne seraient pas conflictuels et pourraient être exécutés en parallèle.

- Ce modèle n'est pas approprié pour une bonne gestion des ressources : la gestion par pile se prête bien aux situations où l'environnement peut être ramené exactement à son état d'origine lors du dépilement. Ce n'est évidemment pas le cas lorsque l'agent dispose de ressources consommables, mais cet écueil n'est pas du tout abordé par Dury.

Dans cette thèse, nous proposerons une approche qui reprend les points forts évoqués ci-dessus (associer des comportements aux rôles sans perdre la dynamique, gestion des conflits de rôles) mais qui propose des améliorations sur chacun des défauts que nous avons relevés.

MOISE+

L'approche MOISE+ [HSB02b] a pour but de proposer une synthèse entre deux types d'approches globales des SMA :

1. Les approches basées sur des *plans globaux*, s'intéressant à l'allocation des tâches aux agents, à la coordination pour exécuter un plan, etc.
2. Les approches basées sur la notion de rôle et de structure organisationnelle.

Pour ce faire, les auteurs proposent de décrire une *structure organisationnelle* en trois volets : la *spécification structurelle*, la *spécification fonctionnelle* et la *spécification déontique*.

La spécification structurelle est constituée d'un ensemble de *rôles*, de *liens* entre ces rôles et de *groupes*.

- Les rôles ne sont à ce stade que des étiquettes. Ils peuvent être structurés par une relation d'héritage.
- Les liens peuvent être de trois types : lien d'*accointance* (autorisant la représentation d'un agent par un autre), lien de *communication* (autorisant bien sûr la communication) et lien d'*autorité* (autorisant le contrôle d'un agent par un autre).
- Les groupes sont constitués de rôles et de sous-groupes. À cela s'ajoutent des informations sur la *cardinalité* des rôles et des sous-groupes, une relation de *compatibilité* entre les rôles (indiquant si deux rôles peuvent être joués par le même agent), et une donnée sur la *portée* des liens (inter- ou intra-groupe).

La spécification structurelle de MOISE+ correspond donc au niveau abstrait d'Aalaadin ou aux concepts similaires dans les autres approches. Mais la traditionnelle structuration en niveaux est ici complétée par d'autres types de spécification.

La spécification fonctionnelle est constituée de *Schémas Sociaux*. Un tel schéma est un ensemble de *plan globaux*, organisés en *missions*.

- Un plan global est une arborescence de *buts collectifs*, décomposés en un arbre admettant trois types de décomposition : séquentielle, sélective ou parallèle.
- Une mission est un ensemble de buts contenus dans un même plan global. Chaque mission possède une cardinalité décrivant combien d'agents peuvent l'accomplir simultanément.

La spécification déontique établit le lien entre les deux premiers types de spécification. Elle est formée de permissions et d'obligations qui spécifient les missions sur lesquelles un agent prenant un rôle donné peut ou doit s'engager, respectivement.

Une spécification MOISE+ peut être simulée par la plate-forme du même nom [HSB02a] ; après avoir exprimé les trois types de spécification dans des fichiers XML, on peut créer des groupes et des agents, faire prendre des rôles aux agents, etc. Ceci permet d'étudier l'évolution

des obligations des agents en fonction de la structure du système. On notera que les "agents" manipulés par la plate-forme ne sont que des entités passives enregistrant quels rôles ils jouent, quelle mission ils remplissent, etc.

MOISE+ propose un formalisme qui sous bien des aspects est plus riche que les autres approches présentées ici : l'héritage entre rôles, les différents types de liens, la représentation explicite d'une hiérarchie de buts collectifs sont des aspects peu ou pas abordés dans les autres approches. Ce modèle nous semble néanmoins souffrir de quelques imperfections :

- L'absence de modèles d'interactions représente une lacune importante pour la spécification d'un SMA. Ce point devrait cependant être vite corrigé puisque [HSB02b] signale qu'un tel modèle est en cours de développement (ainsi d'ailleurs qu'un modèle environnemental).
- L'approche n'est pas du tout orientée vers la réutilisabilité ; en particulier, l'existence de liens inter-groupes exclut l'utilisation du groupe comme entité réutilisable dans un autre contexte.
- L'approche se limite à la spécification et n'aborde pas du tout le passage à l'implémentation. De fait, les agents sont les grands absents de MOISE+. Une réflexion semble donc s'imposer pour réduire le fameux "implementation gap".

Le modèle que nous proposons dans cette thèse possède un langage de spécification moins riche, mais il est en revanche plus axé sur la réutilisabilité et le passage à l'implémentation.

Discussion

Le tableau de la figure 2.5 résume pour chacune des approches que nous avons évoquées ci-dessus :

- quelles motivations les auteurs avancent pour justifier leur choix d'une approche organisationnelle
- quels sont les principaux points que nous reprenons dans notre approche
- sur quels points MOCA apporte des améliorations ou des nouveautés.

Cette synthèse nous permet de bien situer notre approche par rapport aux travaux existant dans un contexte organisationnel :

Motivations Le développement de MOCA est centré autour de l'idée de réutilisabilité ; toute l'approche tend à assurer la plus grande indépendance possible entre les entités constituant les différents niveaux d'un système, de manière à pouvoir les réutiliser dans un autre contexte. Ce choix conditionne évidemment la présentation et l'organisation des concepts contenus dans cette thèse ; cependant, on constatera qu'il n'est pas incompatible avec les autres motivations évoquées dans le tableau récapitulatif (à l'exception peut-être de celles de MOISE+). En particulier, la prise en compte de l'hétérogénéité est une conséquence directe de la modularité induite par notre souci de réutilisabilité.

Points centraux La figure 2.5 met en évidence les points qui distinguent MOCA de chacune des approches ci-dessus ; nous résumons maintenant les points qui caractérisent notre travail d'un point de vue organisationnel :

- structuration du système en niveaux et prise en compte du double lien d'instanciation
- attribution de comportements aux rôles
- dynamique organisationnelle
- gestion des conflits de rôles

Approche	Motivations	Éléments retenus	Éléments ajoutés
Durand	– points de vue partiels sur le système	– double instanciation – attribution de comportements aux rôles	– dynamique organisationnelle – indépendance des niveaux conceptuels
Cassiopée	– mise en oeuvre de comportements collectifs	– expression des comportements élémentaires de l'agent	– structuration en niveaux – dynamique des compétences – approche plus systématique des concepts organisationnels
Aalaadin	– structuration exprimable au niveau du SMA – prise en compte de l'hétérogénéité – réutilisation	– modèle AGR – structuration en niveaux – dynamique organisationnelle	– attribution de comportements aux rôles – réification des concepts "abstraits"
Parunak & Odell	– cf. Aalaadin	– attribution de comportements aux rôles	– représentation formelle et exécutable
Hilaire	– abstraction et formalisation des comportements collectifs	– représentation des comportements associés aux rôles sous forme hybride Object-Z/Statecharts	– dynamique organisationnelle – rôle plus central pour l'agent – gestion des conflits de rôles – couverture de la phase de conception
Gaia	– méthodologie de conception réellement orientée agent	– représentation explicite et formelle des rôles et des interactions	– les concepts organisationnels existent à toutes les phases du développement
Opéra	– spécification des interactions	– attribution de comportements aux rôles – dynamique organisationnelle – gestion des conflits de rôles	– la prise de rôles est un choix délibéré de l'agent – gestion des conflits de rôles plus fine – représentation des comportements plus riche – explicitation de la notion d'organisation
MOISE+	– combinaison des approches centrées sur les plans globaux et de celles centrées sur les rôles.	– structuration en niveaux – dynamique organisationnelle	– implémentabilité – réutilisation

FIG. 2.5 – Vue synthétique des approches organisationnelles

Notre approche se caractérise également par l'introduction de la notion de compétence qui, nous l'avons vu, généralise de nombreuses notions apparaissant dans les diverses approches évoquées. Mais pour bien comprendre le statut de cette notion, il est nécessaire d'étudier également l'aspect componentiel de notre système. Ce sera le sujet du prochain paragraphe.

2.2 Architectures componentielles

L'approche componentielle est un paradigme de programmation relativement récent ; [Ric01] la présente comme l'aboutissement (provisoire) d'une démarche qui, de la programmation procédurale en passant par la programmation objet, vise à une toujours plus grande réutilisabilité du code.

Une des sources d'inspiration de cette approche vient de l'électronique, où un *composant* est un élément destiné à être utilisé dans la conception de circuits complexes. Il est doté d'un certain nombre de bornes d'entrée et de sortie ; seules ces bornes doivent être documentées, le reste pouvant être considéré comme une boîte noire[5]. Cette encapsulation des fonctionnalités permet une très bonne réutilisabilité du composant dans les contextes les plus divers.

Si l'on cherche à transposer cette situation au génie logiciel, on constate que l'encapsulation fournie par les objets n'est pas assez forte : si l'explicitation des "bornes d'entrée" (méthodes) d'un objet est bien réalisée, ce n'est pas le cas des "bornes de sortie", puisqu'un objet ne précise pas explicitement quels appels il va faire à son entourage.

Le développement d'un composant passe donc au minimum par la spécification de ses bornes d'entrée et de sortie. Ceci dit, il n'existe pas encore de consensus précis sur le contenu exact de cette notion.

Il existe plusieurs travaux dans le domaine des SMA qui se réclament d'une approche componentielle. Nous allons brièvement les présenter ci-dessous pour mieux cerner la signification de cette notion dans un contexte multi-agent.

DESIRE

L'approche DESIRE [BJT97, BDJT97, DES] se situe parmi les premiers travaux adoptant une approche componentielle dans le domaine des SMA. Il s'agit d'un cadre déclaratif permettant la modélisation du raisonnement à l'intérieur des agents, des interactions entre agents et avec le monde extérieur.

Un composant DESIRE est essentiellement une boîte possédant deux bornes d'entrée et deux de sortie. Les composants peuvent être imbriqués récursivement, permettant une décomposition hiérarchique du problème à résoudre. A un niveau donné, les composants sont connectés par des liens décrivant le flux des données dans l'agent.

DESIRE prend en compte explicitement la distinction entre un certain niveau d'information et son méta-niveau, contenant de l'information sur cette information. Ainsi, on peut par exemple exprimer qu'une valeur est indéterminée. C'est la raison pour laquelle les composants ont deux bornes d'entrée ou de sortie : une pour les informations et une pour les méta-informations.

La figure 2.6 représente un agent DESIRE typique ; il s'agit d'un composant possédant divers sous-composants et des liens entre eux. Chacun de ces sous-composants peut encore

[5] A l'exception de contraintes sur les conditions d'utilisation (température, etc.).

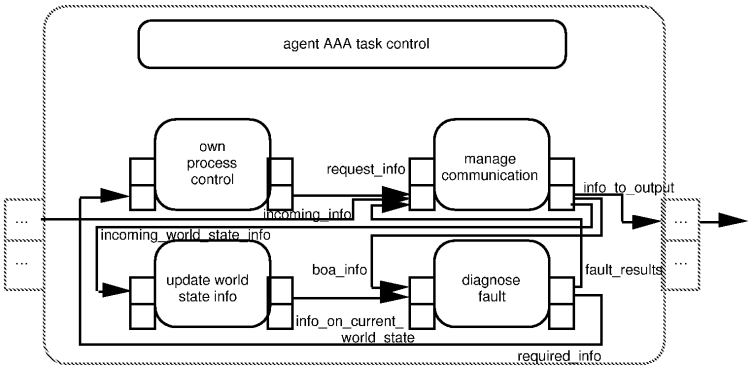


FIG. 2.6 – Un agent DESIRE

être raffiné si nécessaire. L'environnement peut aussi être représenté par un composant, et le SMA dans son ensemble consiste alors en un ensemble de composants reliés entre eux.

L'approche est accompagnée d'outils logiciels de génération de code et de vérification, qui permettent notamment son utilisation en milieu industriel.

DESIRE fournit une méthodologie systématique de conception d'agents basée sur la décomposition des tâches. Il s'agit d'une méthode solide pouvant être utilisée pour le développement d'applications réelles. Par contre, on relèvera que le problème de la réutilisabilité des composants est peu abordé dans cette approche. De plus, l'approche choisie exclut la possibilité d'une évolution de la composition de l'agent en cours d'exécution.

BRIC

BRIC (Block-like Representation of Interactive Components) [Fer99] est un langage visant à permettre la conception et la réalisation de SMA à partir d'une approche modulaire. Un composant BRIC est une structure logicielle caractérisée extérieurement par un certain nombre de bornes d'entrée et de sortie ; de manière interne, un composant peut être défini soit par un ensemble de composants soit par un réseau de Petri[6]. Un composant peut aussi être une boîte noire pour représenter des fonctionnalités externes à l'agent ou au système.

Comme pour DESIRE, agents et environnements sont des composants et un SMA consiste en un ensemble de composants reliés. La figure 2.7 est une représentation (partielle) d'un agent BRIC avec un composant de perception et un de délibération. Notons que dans [Fer99], Ferber donne un exemple d'un SMA comportant, en plus des agents et de l'environnement, un composant chargé de la synchronisation du système.

Les réseaux de Petri possèdent de remarquables propriétés [Bra83a, Bra83b] qui en font un outil de vérification formelle très apprécié. Cependant, les réseaux complexes deviennent vite illisibles. L'idée de les encapsuler dans des composants permet donc à la fois d'améliorer leur lisibilité et leur réutilisabilité. Elle a par conséquent été reprise et développée dans divers travaux orientés vers la vérification formelle, dont ceux de Yoo (cf. ci-dessous) et ceux d'Hameurlain (cf. chapitre 8).

[6]Plus précisément, un réseau de Petri coloré avec arcs inhibiteurs.

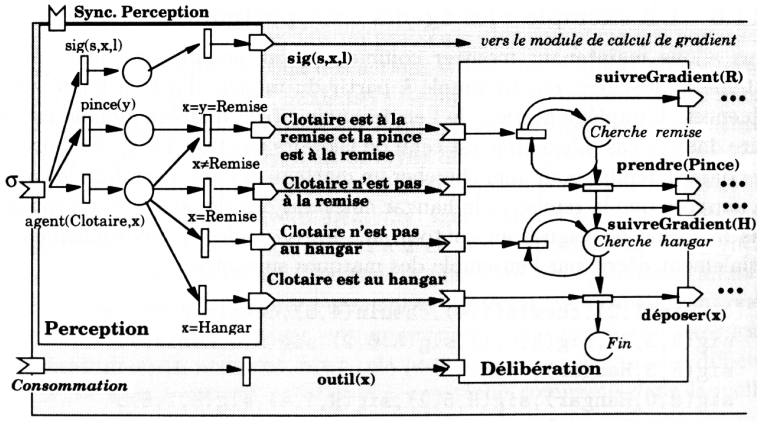


FIG. 2.7 – Un agent BRIC

Maleva

L'approche *Maleva* [Lhu98, GH+99] a pour objectif de fournir un modèle de conception simple et des outils de réalisation pour les agents et les SMA. Le modèle componentiel a été choisi pour la réutilisabilité qu'il offre, permettant ainsi, une fois une bibliothèque constituée, une conception "à la souris".

Un composant Maleva est caractérisé par

- un comportement interne. Celui-ci est spécifié soit par une combinaison de sous-composants, soit directement dans le langage d'implémentation (en l'occurrence Delphi).
- un ensemble de bornes d'entrée et de sortie. Une particularité de Maleva est de distinguer le *flot de données* du *flot de contrôle*[7] ; à ces deux flots correspondront donc deux types de bornes. Un composant n'est activé que s'il reçoit un signal sur une borne de contrôle.
- un gestionnaire de message. Celui-ci est responsable de la gestion des bornes et des communications inter-composants.

La figure 2.8 illustre un agent Maleva typique, constitué d'une hiérarchie de sous-composants. Les traits pleins représentent les liens de données alors que les traits discontinus correspondent aux liens de contrôle.

Ce modèle a été réalisé sous la forme d'un ensemble de classes Delphi desquelles on peut hériter pour créer de nouveaux composants ; une interface graphique (orientée simulation) permet ensuite la création d'agents comme regroupement de composants et leur exécution.

Ce travail fait actuellement l'objet d'une réimplémentation en Java [MB01] basant la notion de composant sur celle de *JavaBean*. Parmi les améliorations proposées, on relèvera la possibilité d'une évolution de l'architecture de l'agent en cours d'exécution.

L'idée de séparer les flots de donnée et de contrôle constitue l'un des points forts de Maleva. Malheureusement, une formalisation très proche de l'implémentation réduit la généricité de cette approche. De plus, on remarquera que dans sa version d'origine, la structure des agents est fixée à la conception, excluant toute dynamique en cours d'exécution. Il se peut néanmoins que ces diverses faiblesses disparaissent dans la réimplémentation en cours.

[7] Ceci est à contraster avec l'approche orientée objet, dans laquelle une transmission de données s'accompagne toujours de la transmission du contrôle.

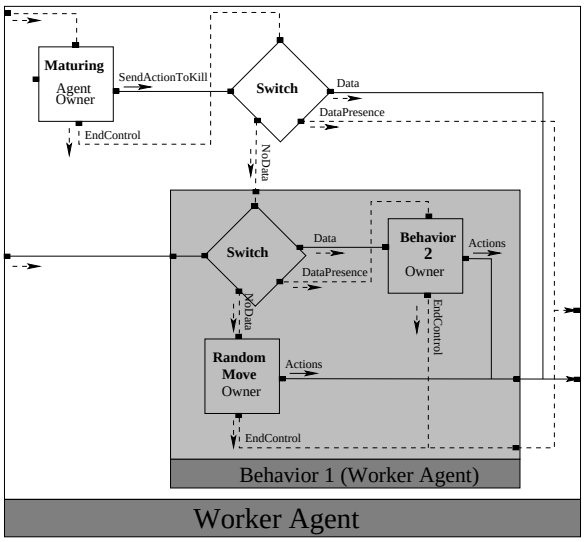


FIG. 2.8 – Un agent Maleva

JAF

JAF (Java Agent Framework) [HL98, VHL01] est une approche orientée vers le développement rapide de divers types d'agents. Le cadre d'origine est l'environnement de simulation MASS [VHL01], mais le modèle d'agent obtenu peut être intégré dans diverses plate-formes.

Les composants de JAF sont essentiellement des *Beans* Java qui peuvent être manipulés dans la Beanbox de Sun, par exemple. Ceci permet de développer une bibliothèque de composants standard pouvant être intégrés à un agent de manière "plug and play", ainsi qu'illustré à la figure 2.9.

Les connexions entre composants se font sur une base de publication/abonnement, ce qui ouvre la porte à une évolution de la composition de l'agent en cours d'exécution.

JAF fournit plus d'une trentaine de composants pour les tâches courantes : communication, agencement d'actions, stockage d'information, diagnostic de fonctionnement, etc.

JAF se distingue de la plupart des autres travaux présentés ici par son mode de communication inter-composants : le choix d'un mode de connexion implicite par évènements ouvre la porte à une dynamique de la composition de l'agent. Nous verrons que MOCA utilise un principe similaire, basé sur la notion de *compétence*, pour permettre cette dynamique.

SCD

Les travaux de M.-J. Yoo et J.-P. Briot [Yoo99, YB01] résultent d'une volonté de prise en compte, dans un cycle de développement d'agents logiciels, des trois critères suivants :

1. La facilité de conception
2. L'efficacité de l'implémentation
3. La validité du modèle

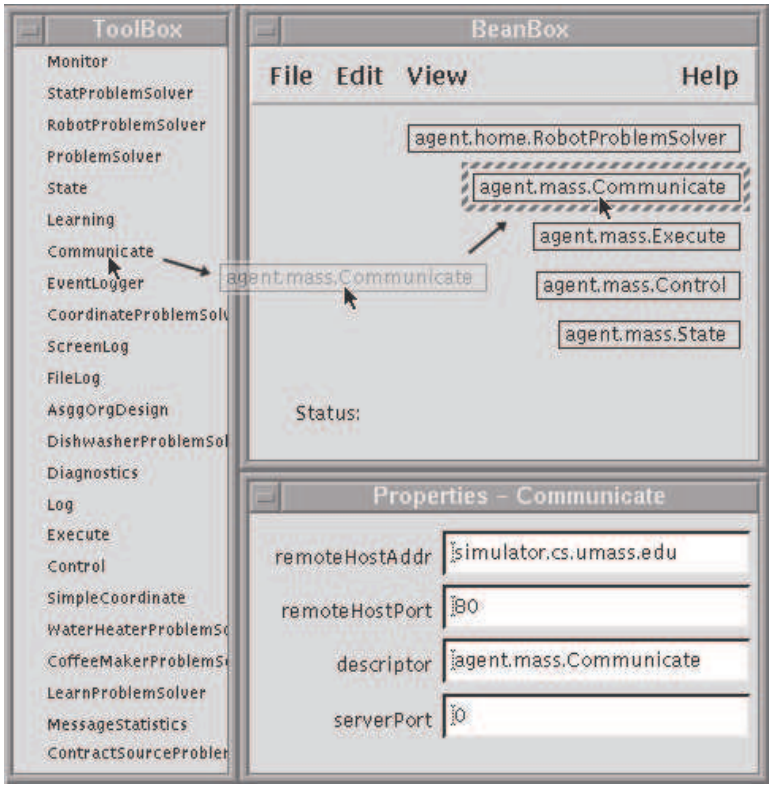


FIG. 2.9 – Un agent JAF en cours de conception dans la Beanbox

Pour ce faire, elle développe le langage SCD (SoftComponent Description), qui est un langage (textuel) de description basé sur les notions d'état et de transition. Une spécification SCD peut ensuite être traduite (automatiquement) en code Java exécutable ou en réseau de Petri pour validation. Il est à noter que la génération de code est indépendante de la plate-forme multi-agent utilisée.

Les composants disposent d'une représentation graphique inspirée de BRIC, mais enrichie d'une distinction entre bornes de communication synchrone ou asynchrone.

La figure 2.10 illustre la structure standard d'un agent dans cette approche : il se présente comme une "coquille vide" possédant des places pour trois types de composants :

1. Les composants de communication, qui sont responsables du traitement des messages entrants et sortants.
2. Les composants de coopération, qui permettent la description de protocoles de coopération.
3. Les composants de tâche, qui décrivent les capacités propres à l'agent.

Les connexions entre les composants de communication et ceux de coopération peuvent s'effectuer automatiquement sur la base du nom des bornes, alors que la connexion entre ces derniers et les composants de tâche doit être effectuée manuellement.

L'approche de Yoo et Briot se caractérise par son orientation vers la validation formelle ; nous reviendrons sur cet aspect au chapitre 8. Yoo relève cependant que "ce qui reste à

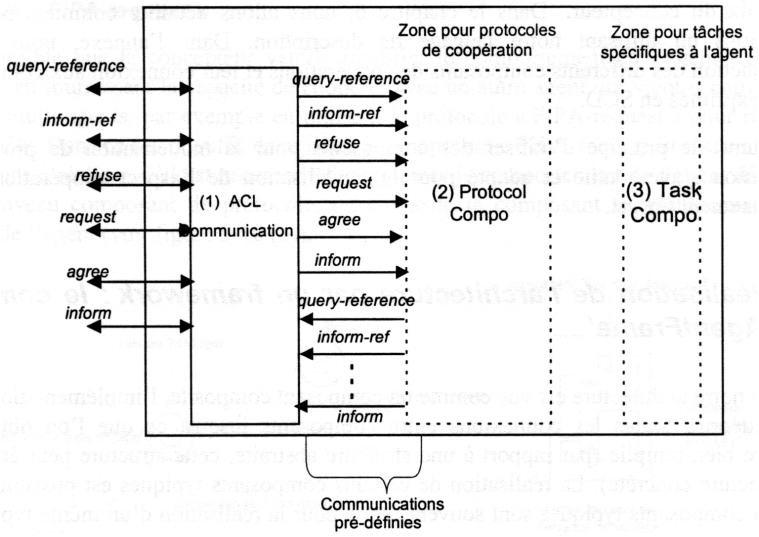


FIG. 2.10 – Un agent pouvant contenir des composants SCD

résoudre, c'est le problème de l'*organisation* des agents qui met en valeur l'aspect coopération des agents" [Yoo99]. De ce point de vue, notre thèse constitue une extension de ce travail ; en effet, l'architecture des agents MOCA possède des similarités avec celle de Yoo, mais nous remplaçons les "composants de protocoles" par des rôles, explicitement intégrés dans une structure organisationnelle.

Voyelles

Voyelles est une approche intégrée des systèmes multi-agents basée sur la décomposition de ceux-ci en quatre briques constituantes :

1. Les agents, qui concernent les modèles (ou les architectures) utilisés pour la partie active de l'agent, d'un simple automate à des cas plus complexes, comme des systèmes à base de connaissances.
2. Les environnements, qui sont les milieux dans lesquels sont plongés les agents. Ils sont spatiaux dans la plupart des applications proposées.
3. Les interactions, qui concernent les infrastructures, les langages et les protocoles d'interaction entre agents, depuis de simples interactions physiques à des interactions langagières par actes de langage.
4. Les organisations, qui structurent les agents en groupes, hiérarchies, relations, etc.

Cette décomposition en briques est orthogonale au découpage du système en agents ; en particulier, il ne faut pas confondre la brique agents, qui définit les modèles d'agent présents dans le système avec les agents en tant qu'entités informatiques.

Dans sa thèse, P.-M. Ricordel [Ric01] s'inspire des approches componentielles pour augmenter la réutilisabilité des briques. Celles-ci sont alors constituées des éléments suivants :

– Des composants distribués, qui sont des entités logicielles pouvant être distribuées sous forme de composants au sens habituel du terme au sein des agents.

– Des services globaux, qui sont des éléments ne pouvant être distribués et qui sont nécessaires au fonctionnement des composants distribués.
– Des éditeurs qui permettent au cours de la phase de développement d'instancier de manière confortable les deux premiers éléments.

La structure des briques peut être spécifiée dans un langage *ad hoc* nommé *Madel*, la partie opérationnelle étant codée en Java.

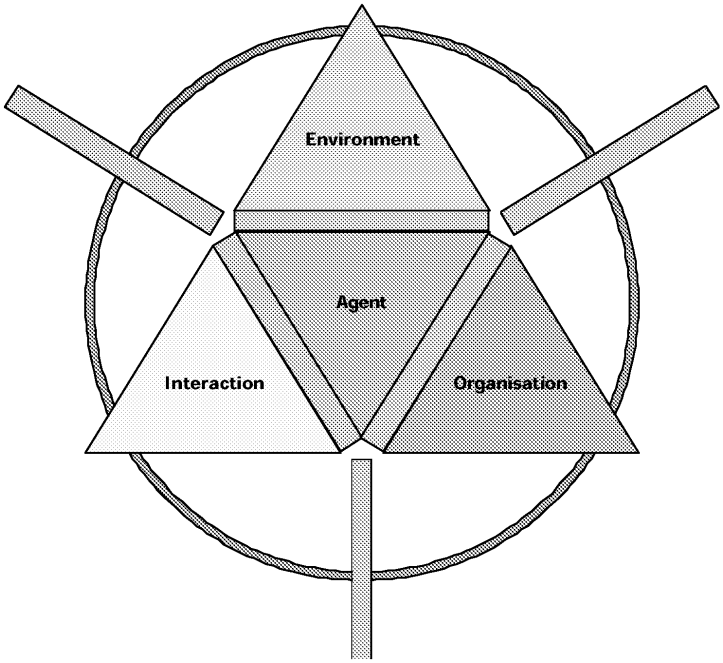


FIG. 2.11 – Un SMA selon l'approche Voyelles

La connexion des composants entre eux est basée sur un modèle par évènements : les composants génèrent des évènements qui peuvent être exploités par d'autres. On a donc une connexion implicite par le type d'évènement, contrairement aux modèles explicites que nous avons vu ci-dessus. On notera cependant que pour éviter des conflits de noms, Ricordel introduit des composants spéciaux, les interbriques, chargés d'assurer la compatibilité entre les briques et de traduire les évènements si nécessaire. On obtient donc la structure de la figure 2.11 : les triangles représentent les briques et les rectangles les interbriques.

L'approche Voyelles se caractérise par une granularité beaucoup plus grosse que les autres travaux que nous avons présentés. Les briques pouvant être des objets assez complexes, on comprend bien le souci de réutilisabilité qui a amené à cette formulation componentielle. On peut cependant se demander si l'architecture choisie est la plus efficace : l'introduction de la notion d'interbriques, conçue pour minimiser les contraintes sur le développement des briques, nécessite pour chaque nouvelle combinaison de briques le développement d'interbriques adaptées. Ces éléments pourraient en principe être réutilisés, mais "généralement des interbriques plus complexes sont nécessaires, mettant en jeu des corrections sémantiques, suivant des protocoles complexes, ou complémentant des lacunes pour que l'une des briques soit exploitable par l'autre" [Ric01]. Il nous semble donc que cette approche est, parmi les travaux que nous

présentons ici, celle qui propose l'encapsulation la moins forte, et par conséquent demande le plus gros effort pour la réutilisation.

2.3 Discussion

La figure 2.12 propose une vision synthétique des différentes approches componentielles que nous avons étudiées, ainsi qu'une comparaison avec notre proposition, selon les critères suivant :

- Récursivité : l'approche admet-elle une composition récursive des composants (c'est-à-dire des composants composés eux-mêmes de composants) ?
- Dynamique : l'approche permet-elle une évolution de la structure de l'agent en cours d'exécution (adoption/rejet de composants).
- Composants : cette colonne décrit à quel niveau du système les composants interviennent et quel est leur formalisme de représentation.
- Bornes : cette colonne relève les particularités de l'approche en ce qui concerne les connexions entre composants.

MOCA, nous l'avons dit, utilise une approche componentielle comme outil pour développer une approche organisationnelle souple. Cela a bien entendu fortement conditionné notre approche. En particulier, il ne nous a pas paru nécessaire d'inclure une dimension récursive. Par contre, l'aspect dynamique était pour nous tout à fait indispensable. Il est évident que cet aspect nécessite une connexion automatisée des composants. Pour résoudre ce problème, JAF, et bientôt Maleva, ont choisi un modèle de communication par évènements, hérité des Java Beans. Nous proposons avec MOCA une solution similaire, basée sur la notion de *compétence*. On peut cependant remarquer que notre implémentation ne s'appuie pas sur les mécanismes des Beans, mais sur la notion d'interface java (cf. chapitre 9).

2.4 Conclusion

Cette présentation de la littérature nous a permis de replacer cette thèse dans son contexte, pour à la fois rendre justice aux travaux qui nous ont inspirés, relever les points qui méritaient d'être étudiés plus en détail et mettre en évidence les spécificités de notre approche.

Avant de passer à la présentation de notre modèle, qui sera l'objet de la deuxième partie de ce document, nous désirons encore exposer plus en détail le formalisme de Vincent Hilaire sur lequel nous basons notre formalisme de représentation des rôles. Ce sera donc l'objet du prochain chapitre.

Approche	Récursive	Dynamique	Composants	Connexions
DESIRE	oui	non	Agents, environnement et leur décomposition. Les composants élémentaires ne sont pas décrits.	Distinction entre information et méta-information.
Bric	oui	non	Agents, environnement et leur décomposition. Composants élémentaires sous forme de réseaux de Petri.	Pas d'autre distinction que bornes d'entrée/-bornes de sortie.
Maleva	oui	prévue	Agents et leur décomposition. Composants élémentaires codés en Delphi.	Distinction entre bornes de données et bornes de contrôle.
JAF	non	oui	Premier niveau de décomposition des agents. Les composants sont des JavaBeans.	Connexion automatique basée sur le principe publication/abonnement.
SCD	non	non	Premier niveau de décomposition des agents.	Distinction entre bornes synchrones et asynchrones. Connexion partiellement automatique sur la base du nom des bornes.
Voyelles	non	non	En nombre fixe à l'intérieur de chaque agent. Description mixte Madel/Java.	Connexions par événements. Notion d'interbriques pour éviter les collisions de noms.
MOCA	non	oui	Premier niveau de décomposition des agents.	Connexion automatique et dynamique basée sur la notion de *compétence*

FIG. 2.12 – Vision synthétique des approches componentielles

Chapitre 3

Le formalisme de Vincent Hilaire

Parmi les travaux que nous avons présenté au chapitre précédent, le plus avancé en ce qui concerne la spécification formelle de comportements pour des rôles dans une organisation est sans conteste celui de Vincent Hilaire [Hil00].

L'approche de Hilaire se base sur la constatation que, parmi les formalismes disponibles pour la spécification de systèmes multi-agents, aucun ne permet de tenir compte à la fois de tous les aspects pertinents. Il propose ainsi une combinaison de deux formalismes : les statecharts pour l'aspect réactif et Object-Z pour l'aspect fonctionnel.

Nous allons donc d'abord présenter brièvement ces deux formalismes, avant d'en exposer la combinaison proposée par Hilaire.

3.1 Statecharts

Les statecharts sont un formalisme proposé en 1987 par David Harel [Har87] pour la spécification de systèmes complexes. Il s'agit d'une extension des automates à états finis incluant les compositions hiérarchique et parallèle d'états.

États et configurations

Les états d'un statechart peuvent donc être décrits comme un ensemble structuré par une décomposition hiérarchique et-ou. La composition "ou" se représente graphiquement par juxtaposition à l'intérieur de l'état parent et la composition "et" par le traçage d'une ligne discontinue au niveau du parent. La figure 3.1 donne un exemple d'arbre et-ou d'états (a) et de sa représentation en statecharts (b).

On remarque que les états-ou directement inclus dans un état-et (comme $A1$ et $A2$ dans l'exemple) n'ont pas de vraie représentation graphique. Dans la pratique, on omet même de représenter leur nom et ils deviennent tout à fait implicites.

Tout état-ou doit posséder dans ses descendants directs un *état par défaut*, qui sera activé en même temps que son ancêtre à défaut d'autre indication. Les états par défaut sont représenté en grisé dans la figure 3.1[1].

[1] Il est à noter que nous employons ici une notation différente de celle d'origine ; dans les travaux de Harel, les états par défaut sont indiqués par une transition particulière. Nous avons opté pour le grisé par mesure de simplification.

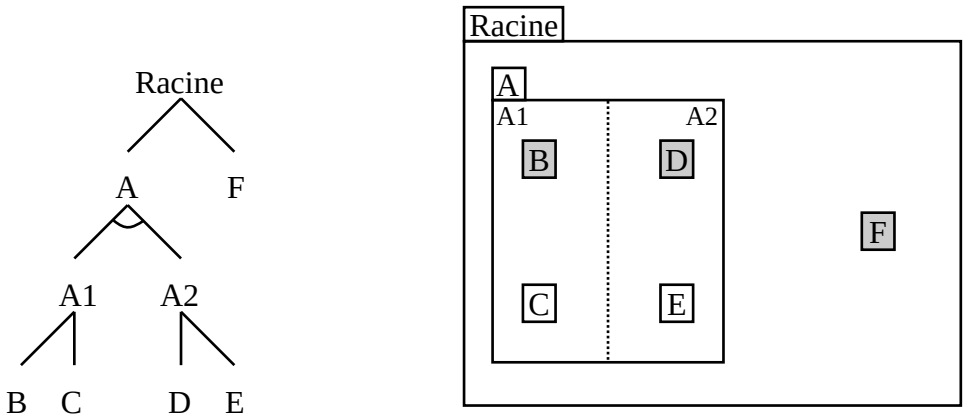


FIG. 3.1 – Une décomposition et-ou et sa représentation en statechart

La composition hiérarchique des états d'un statechart nécessite une extension de la notion classique d'état d'un automate ; en effet, un statechart peut être dans plusieurs états simultanément. C'est pourquoi on introduit la notion de *configuration*, qui correspond à un ensemble maximal d'états dans lesquels le système peut être simultanément. Plus précisément, [HN96] définit une configuration comme un ensemble C d'états vérifiant les axiomes suivants :

1. C contient la racine
2. Si C contient un état-ou, il doit également contenir exactement un de ses sous-états
3. Si C contient un état-et, il doit également contenir tous ses sous-états
4. Les seuls états de C sont ceux nécessités par les règles ci-dessus.

Par exemple, dans la figure 3.1, $\{\text{Racine}, A, A1, A2, B, D\}$ et $\{\text{Racine}, F\}$ sont des configurations.

A cette notion classique, nous ajoutons ici celle de *sous-configuration*, définie comme étant un sous-ensemble d'une configuration. Ainsi, un statechart peut être simultanément dans tous les états d'un ensemble donné E si et seulement si E est une sous-configuration du statechart.

Transitions

On peut maintenant définir les transitions d'un statechart comme étant des couples (source,but) de sous-configurations. Le fait que la source soit une sous-configuration assure que la transition peut devenir déclenchable, et le fait que le but soit une sous-configuration assure que le déclenchement ne conduit pas le statechart dans une situation contradictoire.

La représentation graphique des transitions est très intuitive. La figure 3.2 donne un exemple de transitions rajoutées aux états de la figure 3.1(b). La transition r a B pour source et C pour cible ; la u est de source F et de but A ; la w a $\{C, E\}$ pour source et F pour cible ; etc.

La figure 3.2 ne présente encore pas tout à fait un statechart complet. En effet, le formalisme permet d'associer encore des étiquettes aux transitions ; celles-ci sont de la forme

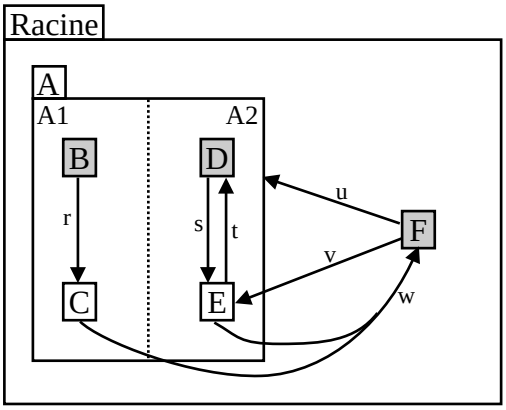


FIG. 3.2 – Un statechart avec transitions

$e[c]/a$, où e est un événement déclencheur, c une condition de déclenchement et a l'action à entreprendre lors du franchissement de la transition. Chacun de ces trois éléments peut être omis ; si l'action consiste en la génération d'un événement, on peut simplement en noter le nom à la place de l'action. La figure 3.3 présente donc un statechart complet tel qu'il pourrait apparaître dans une spécification. La transition de B à C ne peut être franchie qu'en présence de l'événement $e1$ et lorsque la condition $c1$ est vérifiée ; le franchissement de la transition génère l'événement $e2$. Les autres transitions sont étiquetées de manières diverses pour illustrer les différents cas possibles.

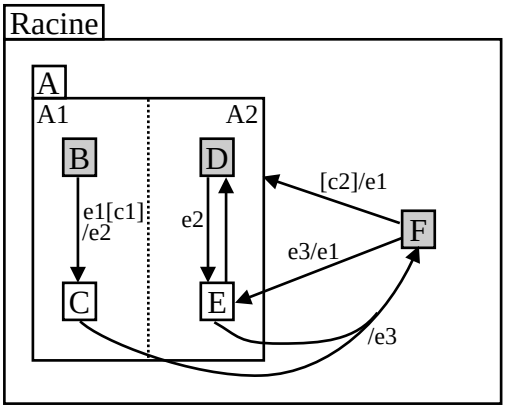


FIG. 3.3 – Un exemple complet de statechart

Nous allons maintenant décrire de manière plus précise la sémantique opérationnelle d'un tel statechart. Mais avant cela, nous introduisons une dernière notion concernant les transitions et qui nous sera utile par la suite : on appelle *portée* d'une transition le plus petit état-ou qui contient strictement sa cible et son but [HN96]. Ainsi, sur la figure 3.3, les trois transitions touchant F ont la *Racine* pour portée, alors que les autres ont respectivement $A1$ et $A2$.

Sémantique

Les statecharts ont donnée lieu à diverses interprétations au niveau de la sémantique opérationnelle. Nous nous baserons sur celle de [HN96], outillée par le logiciel *Statemate* et dont nous avons implémenté une simplification dans MOCA.

Nous ne donnerons ici qu'un bref aperçu de la sémantique opérationnelle des statecharts. Pour une description complète, on pourra se référer au texte précité.

Le modèle d'exécution des statecharts est synchrone. L'état du système avant un pas est défini par la configuration courante C ainsi que par l'ensemble E des événements générés au pas précédent (cet ensemble étant initialement vide). L'exécution d'un pas se déroule de la manière suivante :

1. Ajouter à E d'éventuels événements de provenance externe.
2. Calculer l'ensemble T des toutes les transitions déclenchables.
3. Extraire de T un ensemble maximal non conflictuel T'. Les transitions les plus extérieures ont la priorité ; en cas de conflit à priorité égale, un tirage au sort est effectué.
4. Exécuter les actions de T'. En particulier, remplacer E par l'ensemble des événements générés par les éléments de T'.
5. Adapter la configuration courante C.

Remarques

La présentation ci-dessus ne couvre qu'une partie du formalisme de Harel ; il existe encore d'autre types d'états et de transitions (pour tenir compte de l'histoire, simplifier les branchements conditionnels, etc.) que nous ne prenons pas en compte[2].

D'un certain point de vue, un statechart n'est qu'une abréviation d'un automate à états finis. La composition hiérarchique des états permet de diminuer le nombre de transitions[3] et la composition parallèle permet de diminuer le nombre d'états [PS98]. La seule réelle difficulté pour effectuer une traduction vers un simple automate est la gestion des événements. Il n'est cependant pas clair de savoir si on a à faire à une réelle extension des automates ou à une simple abréviation.

Dans tous les cas, le résultat est un formalisme simple d'emploi, assez lisible et d'une grande expressivité. On peut par contre lui reprocher de se prêter moins bien que certains de ses concurrents à du calcul formel de vérification de propriétés. Pour remédier à ce défaut, deux voies sont possibles : la première est de redéfinir la sémantique opérationnelle de manière à la rendre plus facilement manipulable [Kan96, LMM99, FT00]. La seconde est de définir une traduction des statecharts vers un formalisme possédant de bons outils de vérification formelle, tels les réseaux de Petri [BMM99, HM00, SS00], les systèmes de transitions [US94, EW00, Hil00] ou le μ-calcul [PS98, Lev99].

3.2 Object-Z

Z est un langage de spécification formelle *orienté modèle*, ce qui signifie que sa sémantique est constituée de la définition d'un modèle en termes d'ensembles, de relations et de fonctions.

[2] Il est à noter que ces éléments étaient déjà ignorés dans [Hil00].

[3] La hiérarchie des états définit en même temps une relation de priorité sur les transitions.

On peut alors utiliser une notation logique pour exprimer des contraintes sur ces notions ensemblistes.

Le langage Z a rencontré un certain succès, mais il est apparu que pour la spécification de projets importants, il lui manquait des moyens de structuration. C'est ainsi que le langage Object-Z a fait son apparition. L'idée est d'étendre Z en y ajoutant les notions classiques de classe, instanciation, héritage, etc.

Nous donnons ici une brève présentation, basée sur [Ros92], des concepts et notations d'Object-Z ; notre travail ne faisant usage de ce langage que de manière très superficielle, nous n'entrerons pas plus dans les détails.

La figure 3.4 donne la syntaxe générale d'une classe Object-Z.

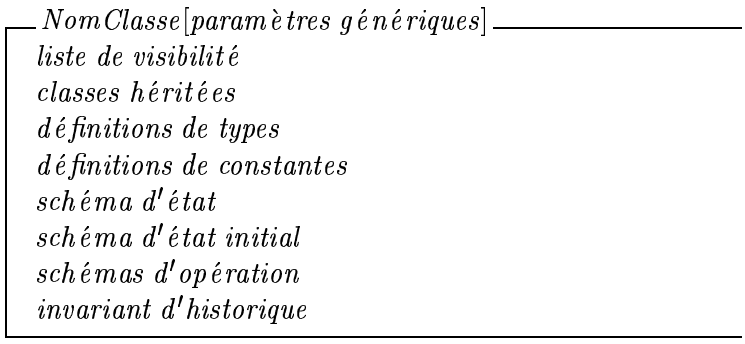


FIG. 3.4 – La syntaxe d'une classe Object-Z

La *liste de visibilité* décrit quels éléments de la classe sont référençables depuis l'extérieur ; une absence de liste sous-entend que tous les éléments de la classe sont visibles. Nous reviendrons ci-après sur l'*héritage* qui présente quelques subtilités par rapport à son acception habituelle dans un contexte orienté objet ; notons au passage qu'Object-Z accepte l'héritage multiple. Les définitions de *types* et de *constantes* ont leur signification usuelle. Le *schéma d'état* définit les variables de la classe ainsi que d'éventuelles contraintes sur leurs valeurs possibles, appelées *invariants de classe*. Le *schéma d'état initial* décrit bien sûr l'état d'une instance de la classe au moment de son initialisation. Les *schémas d'opération* décrivent les opérations de la classe. Enfin, l'*invariant d'historique* est constitué d'un prédicat, généralement en logique temporelle, qui contraint les comportements possibles des instances de la classe.

Pour rentrer un peu plus dans les détails de la notation, nous prenons l'exemple de la spécification d'une pile (figure 3.5).

La figure représente la spécification d'une classe *Stack* dépendant d'un paramètre formel T. Ce paramètre pourra être précisé au moment de l'instanciation ; par exemple, $Stack[\mathbb{N}]$ représentera une pile d'entiers.

Le premier élément à l'intérieur du schéma de classe est la spécification d'une *constante* max, représentant en l'occurrence le nombre maximal d'éléments de la pile. Cette constante est ici limitée à 100 au maximum.

L'élément suivant est le *schéma de classe* : l'état d'une instance de la classe est défini par le contenu de la variable *items*, pouvant contenir une suite d'éléments de type T. C'est à cet endroit que la constante max est utilisée pour borner le nombre d'éléments de la pile.

L'élément suivant est le *schéma d'état initial*, précisant qu'à l'initialisation la pile est vide.

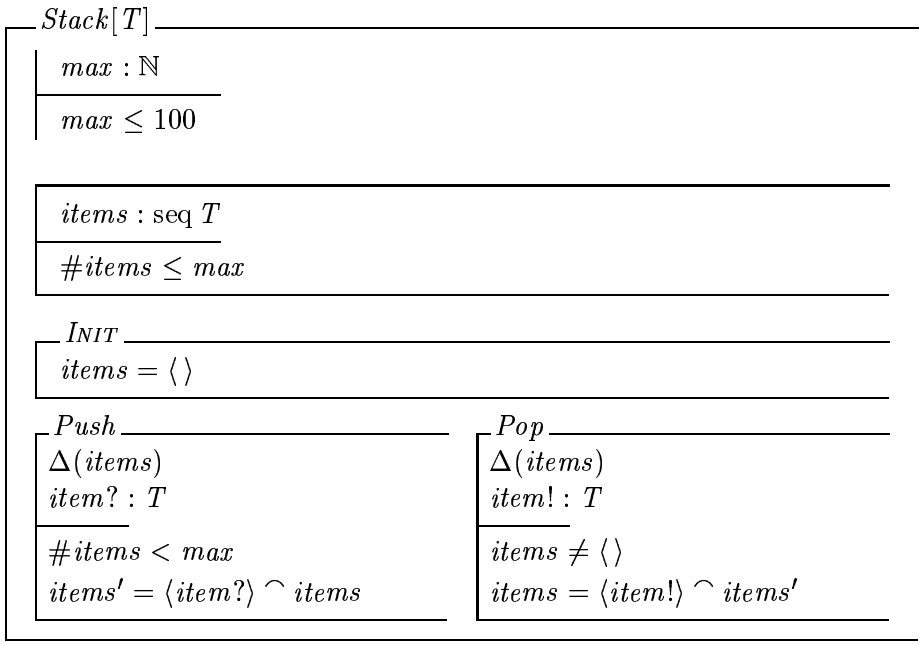


FIG. 3.5 – Une pile en Object-Z

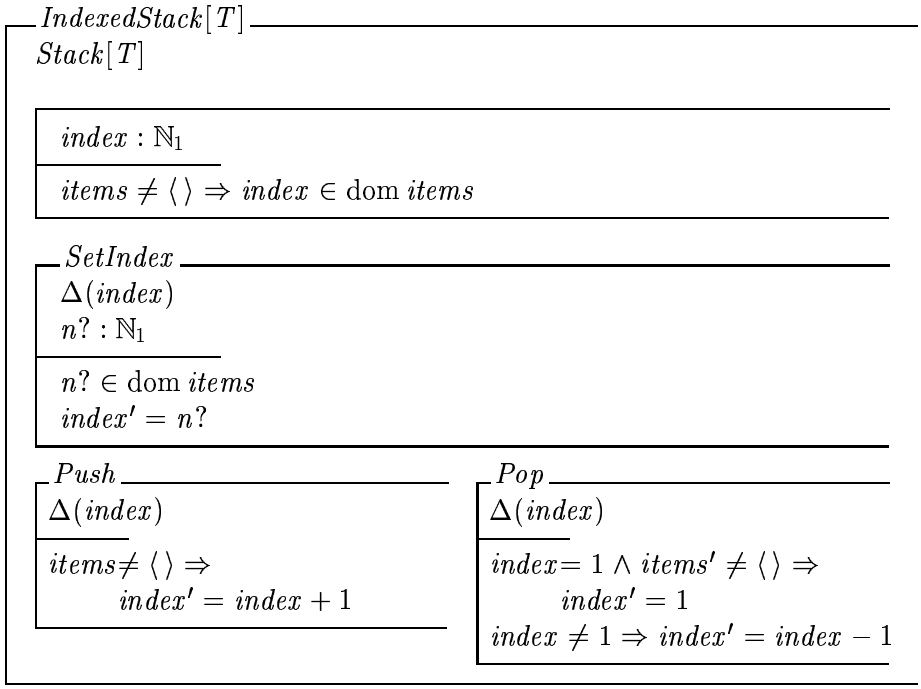


FIG. 3.6 – Une pile indexée héritant de la classe *Stack*.

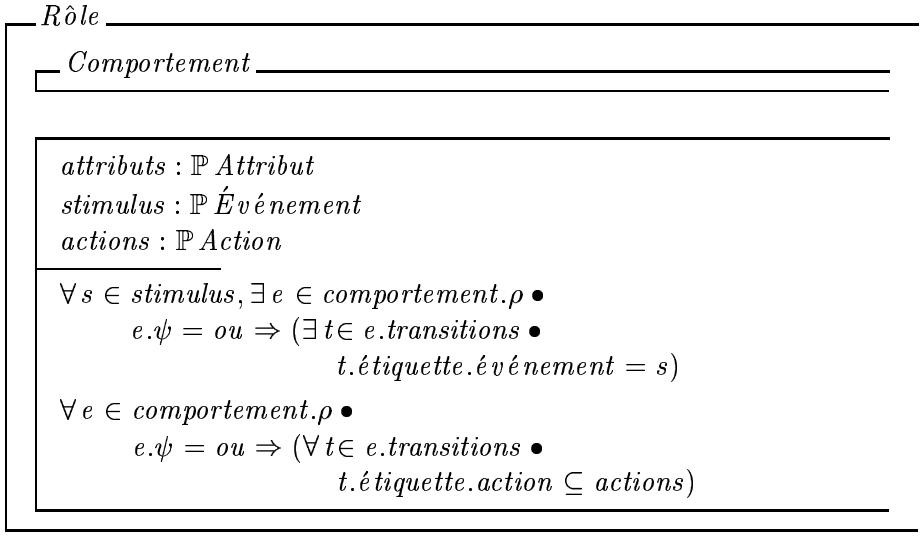


FIG. 3.7 – La classe *Rôle* de Hilaire

Enfin, les deux derniers éléments sont des spécification d'*opérations*. Toute opération doit spécifier, dans une Δ-liste, les variables qu'elle est susceptible de modifier. Les paramètres en entrée sont suivis d'un point d'interrogation ($item?$) et ceux en sortie d'un point d'exclamation ($item!$). Enfin, les valeurs des variables avant l'appel de l'opération sont référencés par le nom de la variable ($items$) et les valeurs après l'opération par ce nom primé ($items'$).

Pour illustrer la manière dont l'héritage est réalisé en Object-Z, la figure 3.6 représente une pile héritant de la classe *Stack* et rajoutant des informations permettant d'avoir une position distinguée dans la pile. Il est à noter que dans cet exemple, l'héritage se fait d'une manière cumulative : les opération *Push* et *Pop* de la classe *IndexedStack* sont définie par la réunion de leur définition dans cette classe et dans la classe mère. Si l'on ne désire pas hériter de la classe mère, on utilisera le mot-clé *redef* après la déclaration d'héritage (par exemple $Stack[T]$ [**redef** $Push$]) ; la nouvelle définition vient alors *remplacer* celle de la classe mère.

Cette présentation ne donne qu'une idée générale du fonctionnement d'Object-Z. En particulier, il existe encore plusieurs opérateurs permettant de définir des nouvelles opérations à partir des opérations existantes ([] qui représente un choix non déterministe entre deux opérations, ‖ qui ressemble à un *pipe* Unix, • qui fusionne les contextes de deux opérations, etc.). Nous renvoyons le lecteur intéressé à [Ros92] ou [DRS94] pour plus de détails.

3.3 Intégration des formalismes

Les statecharts étant bien adaptés à la spécification des aspects réactifs des agents et Object-Z à celle de leur aspect fonctionnel, [Hil00] propose une intégration des deux formalismes pour combiner leurs avantages respectifs. Pour ce faire, il encapsule un statechart dans une classe Object-Z et définit un cadre formel qui permet de référencer les états et transitions du statechart depuis Object-Z, et les opérations, variables et constantes Object-Z depuis le statechart. Nous n'entrerons pas dans les détails techniques de comment cette intégration est faite, mais le résultat est une fusion des formalismes très naturelle d'usage.

En plus de cette intégration syntaxique, Hilaire propose une intégration sémantique réalisée par une traduction des deux formalismes en un système de transition. Il est à noter que cette traduction doit encore se faire à la main (son automatisation fait partie des perspectives de la thèse).

À l'aide de ce formalisme intégré, Hilaire définit les classes *Rôle*, *Interaction* et *Organisation* qui lui permettent ensuite de développer des spécifications par héritage de ces classes. La figure 3.7 montre la classe *Rôle* que nous utiliserons dans notre approche. Le comportement est vide, mais la définition d'un rôle redéfinit ce schéma en y plaçant un statechart. La première formule d'invariant de classe signifie que le rôle doit savoir traiter tous les événements qu'il définit dans ses *stimulus*. La deuxième exige que toute action déclenchée par une transition du statechart doit être définie dans la partie Object-Z. Nous relâcherons cette deuxième exigence dans notre modèle.

Cette présentation du formalisme de Vincent Hilaire clos la première partie de cette thèse, consacrée au contexte de notre contribution. Nous pouvons donc maintenant entrer dans le vif du sujet et présenter les différents aspects de notre modèle MOCA.

Deuxième partie

MOCA

Chapitre 4

Introduction

Dans la deuxième partie de cette thèse, nous présentons notre modèle MOCA, dont le but est de fournir un modèle opérationnel de SMA organisationnel permettant d'associer la prise de rôle à un comportement récurrent tout en permettant une dynamique de la structure organisationnelle.

Pour traiter avec la plus grande souplesse possible l'évolution du comportement de l'agent qui résulte de cette dynamique, nous avons adopté une approche componentielle. Ainsi la prise et le rejet d'un rôle par un agent se traduisent au niveau technique par l'adoption ou le rejet d'un composant.

De plus, pour préserver au maximum l'indépendance entre les entités organisationnelles, nous limitons les interactions entre agents à celles prenant place *à l'intérieur d'un groupe*. Cette caractéristique est un des éléments centraux de notre modèle et nous y reviendrons plusieurs fois au cours des pages qui suivent ; elle permet de concevoir les organisations indépendamment les une des autres et de maximiser ainsi leur réutilisabilité. La seule manière de coordonner des groupes est donc qu'ils partagent des agents ; autrement dit, les agents sont responsables localement de la coordination des groupes. Ceci souligne donc le besoin d'un mécanisme de coordination inter-rôles au sein des agents.

Pour présenter plus en détails ces différents points, nous avons adopté la structure suivante :

- Le chapitre 5 présente les concepts fondamentaux de notre approche ainsi que leur formalisation. Nous y introduisons les deux niveaux d'abstraction qui composent notre modèle, les concepts de chacun de ces niveaux et les mécanismes qui les relient. Ceci nous permet en outre de bien articuler les dimensions componentielle et organisationnelle de notre architecture.
- Le chapitre 6 se concentre sur la dimension componentielle, en particulier sur les interactions entre composants. Ceci permet de préciser comment s'exécute un rôle, comment l'agent gère les échanges entre ses rôles et comment il peut éviter les interférences destructrices entre eux.
- Le chapitre 7 décrit ensuite comment la dynamique organisationnelle est gérée dans MOCA. Nous décrivons pour cela une organisation particulière, l'*Organisation de Gestion*, dont le rôle est de gérer la création de groupes, l'entrée des agents dans un groupe existant et leur sortie.
- En guise de conclusion de cette partie, nous présentons au chapitre 8 quelques éléments de validation. Nous y montrons que la conception très modulaire à tous les niveaux de notre système permet de réduire la validation globale à une série de validations locales.

Toute cette deuxième partie se concentre sur les aspects théoriques de notre approche ; les détails d'implémentation du modèle MOCA dans une plate-forme et les expérimentations réalisées seront quant à eux présentés dans la troisième partie.

Chapitre 5

Les concepts de MOCA et leur formalisation

5.1 Présentation générale

Dans ce chapitre nous présentons les différents concepts de notre modèle. Ceux-ci se répartissent selon deux critères orthogonaux comme illustré à la figure 5.1.

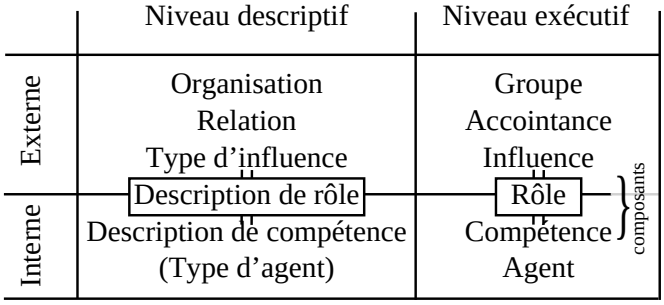


FIG. 5.1 – Les principaux concepts de MOCA

La distinction entre *niveau descriptif* et *niveau exécutif* correspond à la structuration en niveaux que nous avons relevée pour plusieurs des approches citées au chapitre 2, et notamment à la distinction entre concepts abstraits et concrets d'Aalaadin. Le rapport entre ces niveaux est très comparable à celui entre classe et objet dans les langages orientés objet ; c'est pourquoi nous emprunterons parfois la terminologie de ce domaine, notamment en parlant d'*instanciation* des concepts descriptifs[1].

La distinction entre concepts *internes* et *externes* se réfère à leur position par rapport à l'agent : les concepts internes participent à la structuration intra-agent, alors que les concepts externes fournissent la structuration inter-agents. D'un point de vue plus technique, les premiers constituent la partie componentielle de notre approche, et les seconds la partie organisationnelle. La figure 5.1 montre la position critique de la notion de *rôle* dans notre modèle :

[1]Il est à noter cependant que l'analogie conceptuelle ne se prolonge pas forcément au niveau de l'implémentation ; nous verrons en effet au chapitre 9 que dans notre implémentation des concepts de MOCA, le lien entre les niveaux est du type agrégation plutôt qu'instanciation.

à la fois interne et externe, c'est ce concept qui assure le lien entre l'intérieur de l'agent et le reste du système ; c'est également lui qui constitue l'articulation entre les dimensions componentielle et organisationnelle de notre modèle.

Nous allons maintenant présenter plus en détail le fonctionnement de notre système en décrivant les concepts de la figure 5.1. Pour clarifier l'exposé, nous illustrerons ces développements au fur et à mesure en construisant (partiellement) une organisation qui réalise le protocole *Contract Net* de la FIPA [FIP], illustré à la figure 5.2.

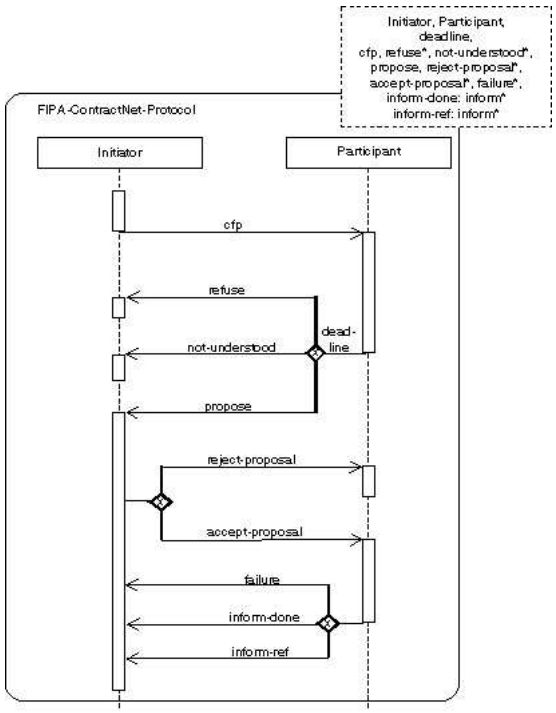


FIG. 5.2 – Le protocole *Contract Net* de la FIPA

5.2 Le niveau descriptif

Le niveau descriptif contient les descriptions statiques des différents comportements et interactions pouvant survenir dans le système ; ces descriptions seront ensuite mises en oeuvre par les éléments du niveau exécutif.

Organisations et relations

Nous définissons une *organisation* comme un *pattern* récurrent d'interactions constituant un point de vue global sur le système. Ainsi une organisation peut se concentrer sur des échanges financiers ou de biens, des comportements collectifs, des échanges langagiers, etc.

Nous formalisons une organisation sous la forme d'un graphe dont les noeuds correspondent à des descriptions de rôles et les arcs à des relations. Chaque noeud possède une cardinalité qui indique combien de fois le rôle correspondant pourra être joué dans un groupe

instanciant cette organisation. De même, les relations possèdent une cardinalité à chaque extrémité indiquant combien de fois chaque rôle pourra instancier cette relation sous forme d'accointance.

Par exemple, le *Contract Net* de la figure 5.2 donne le graphe de la figure 5.3 ; cette organisation contient deux descriptions de rôles : le rôle *initiator* et le rôle *participant* ; dans une instanciation donnée de cette organisation, il ne peut y avoir qu'un seul *initiator*, mais bien sûr plusieurs *participants*, chacun d'entre eux étant possiblement relié à *l'initiator* par une accointance.

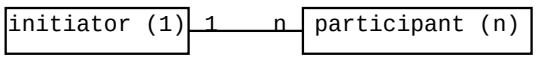


FIG. 5.3 – Une organisation pour le *Contract Net* de la FIPA.

Types d'influence

Un *type d'influence* spécifie le genre de perturbation qu'un rôle peut recevoir ou générer. Nous utilisons le terme influence pour englober dans la même abstraction des concepts tels que les forces pour des agents situés, les évènements et même les actes de langages (ACL, KQML, etc.) pour des agents sociaux. On trouvera dans [FM96] un exposé sur cette notion et son utilité pour décrire les situations d'exécution concurrente.

Du point de vue formel, un type d'influence est défini par son nom et la liste de ses attributs. Pour intégrer cette notion dans le framework de Vincent Hilaire, que nous avons présenté au chapitre 3, nous la formalisons en notation Object-Z :

$$\begin{array}{|l} \textit{InfluenceType} \\ \hline name : \textit{InfluenceName} \\ attributes : \mathbb{P}\, \textit{Attribute} \\ \hline \end{array}$$

où *Attribute* est un type approprié.

Par exemple le premier message de la figure 5.2 aurait *cfp* pour nom et une liste d'attributs décrivant l'offre :

$$\begin{array}{|l} \textit{cfpInfluence} \\ \hline \textit{InfluenceType} \\ \hline name = cfp \\ attributes = \{\ldots\} \\ \hline \end{array}$$

Descriptions de rôles

Une *description de rôle* est définie comme un pattern récurrent de comportement individuel dans une organisation. Comme annoncé, nous basons notre formalisation sur le framework d'Hilaire que nous avons présenté au chapitre 3, avec quelques adaptations.

Le changement principal concerne les transitions des statecharts, dans lesquels nous ajustons à notre contexte la sémantique des étiquettes : si nous gardons bien un triple de la forme *déclencheur[condition]/action*, la nature en est un peu différente. En effet, le *déclencheur* dans MOCA est un type d'influence, alors que la *condition* et l'*action* sont des appels de compétence. Il peut s'agir de compétences internes au rôle, modélisées par des opérations Object-Z, auquel cas on rejoint le cas considéré par Hilaire, mais il peut également s'agir

de compétences externes fournies par l'agent. Ceci nous renvoie alors au fonctionnement de l'architecture componentielle de notre agent que nous discuterons ci-après.

Pour permettre une gestion correcte des compétences, une description de rôle doit indiquer la liste des compétences externes qu'elle utilise. Comme un rôle peut éventuellement aussi — et nous y reviendrons plusieurs fois par la suite — fournir des compétences à l'agent, il doit aussi en indiquer la liste. Ainsi nous spécialisons la classe *Rôle* de Hilaire de la manière suivante[2] :

MOCARoleDescription

Rôle

$req\_competences, prov\_competences : \mathbb{P}\ CompetenceDescription$

A titre d'exemple, nous donnons à la figure 5.4 une réalisation possible du rôle *initiator* de la figure 5.3. Ce rôle nécessite trois compétences extérieures, respectivement pour envoyer des messages (*send*), pour évaluer les offres (*evaluate*) et pour repérer l'arrivée de la date limite (*timeout*). De plus, il fournit une compétence (*delegate_task*) qui permet à un autre composant de l'agent d'utiliser ce rôle pour déléguer une tâche à un autre agent. Les autres compétences sont internes.

À l'appel de la compétence *delegate_task*, le rôle génère un évènement *start* pour son statechart, ce qui provoque l'envoi d'un appel d'offre (*cfp*) ; ensuite il attend et enregistre les offres jusqu'à ce que la compétence *timeout* s'évalue à *vrai*. A partir de ce moment là, il rejette toute nouvelle offre et commence l'évaluation. Après avoir choisi la meilleure offre, il notifie les participants de sa décision puis attend la réponse du participant choisi avant de terminer son exécution[3].

Descriptions de compétences

Les descriptions de compétences sont les spécifications des services nécessités ou fournis par un composant ; des exemples typiques en sont la capacité d'envoyer des messages, de calculer une offre ou d'effectuer un choix entre plusieurs offres dans un réseau contractuel. Cette notion permet de décrire le rôle à un niveau très abstrait, comprenant essentiellement la gestion des interactions.

Nous avons vu ci-dessus des descriptions de compétences dans le rôle *initiator* de la figure 5.4. Il n'y a donc pas grand chose à rajouter sur la formalisation de ce concept. Nous verrons en décrivant le niveau exécutif que les compétences sont toujours fournies par des composants ; c'est donc au chapitre 6 sur les interactions entre composants que nous étudierons comment les appels de compétence sont gérés à l'intérieur de l'agent.

Type d'agent

Dans notre approche, l'autonomie de l'agent réside principalement dans le choix des composants qu'il adopte ou rejette, et en particulier des rôles qu'il prend ou quitte. Toutefois, ceci ne fait pas à strictement parler partie de la couche organisationnelle. MOCA fournit donc des primitives pour la gestion organisationnelle, qui sont mises à la disposition de l'agent pour

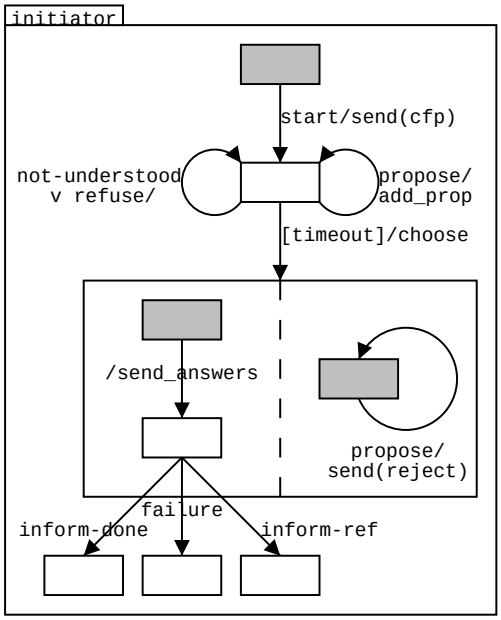
[2]Nous verrons plus tard dans ce chapitre que cette spécialisation correspond au fait à un héritage de la notion de composant.

[3]Il s'agit bien sûr ici d'un exemple de principe. Une vraie réalisation nécessiterait au moins de préciser quelle tâche est déléguée, et probablement de retourner ensuite un résultat au composant appelant.

Initiator

$MOCARoleDescription[redef\ \ behavior]$

behavior



$propositions : \mathbb{P}\, Proposition$
$chosen : Proposition$

$req\_competences = \{send, evaluate, timeout\}$
$prov\_competences = \{delegate\_task\}$

delegate_task

$generate\_event(start)$

add_prop

$\Delta propositions$
$prop?$

$propositions' = propositions \cup prop?$

choose

$\Delta chosen$

$\forall p \in propositions \bullet evaluate(chosen') \geq evaluate(p)$

send_answers

$\forall p \in propositions - \{chosen\} \bullet send(p, reject)$
$send(chosen, accept)$

FIG. 5.4 – Le rôle *initiator*

qu'il puisse gérer ses groupes et ses rôles (cf. figure 5.6 ci-après) ; mais les motivations pour ce faire dépendent du domaine d'application et ne seront donc pas discutées dans cette thèse. C'est pour cela que la notion de type d'agent est entre parenthèses dans la figure 5.1.

Notons cependant que notre architecture laisse place à une grande flexibilité : toutes les possibilités sont ouvertes pour implémenter ces choix, du codage direct dans les agents jusqu'à des processus délibératifs complexes. Dans ce sens-là, MOCA est indépendant du modèle d'agent, comme MadKit ou Gaia.

Une remarque importante au sujet de la gestion des rôles est que, comme les rôles peuvent fournir des compétences, prendre ou quitter un rôle peut avoir une influence sur les compétences disponibles. Ainsi le processus de gestion des rôles doit-il également prendre les compétences en considération.

5.3 Le niveau exécutif

Le niveau exécutif est formé de l'instanciation des concepts du niveau descriptif. Pour bien comprendre comment il fonctionne, il est important de se souvenir que nous n'autorisons les agents à communiquer qu'à l'intérieur des groupes. Cette affirmation reflète un point de vue externe à l'agent qui met en évidence la communication inter-agents et nécessite la clarification des concepts de *groupe*, d'*accointance* et d'*influence*.

Dualement, on peut proposer une expression interne de cette même affirmation, en disant que les rôles appartenant à différents groupes ne sont autorisés à communiquer qu'à l'intérieur d'un agent. Pour gérer cette communication de manière souple, nous avons choisi d'adopter une architecture componentielle dynamique pour nos agents. Les notions résultantes d'*agent*, *composant*, *compétence* et *rôle* vont donc également être décrite dans ce paragraphe.

Le niveau exécutif de MOCA est conceptuellement proche du niveau concret d'Aalaadin. Les différentes notions que nous allons présenter ci-dessous ont donc pour la plupart leur pendant dans MadKit (à l'exception des notions de *composant* et de *compétence*). Pour éviter toute confusion, nous prendrons soin de relever dans ce qui suit les similitudes et les différences entre les concepts apparentés de ces deux approches.

Groupes

Un groupe est une instance d'une organisation ; il est formé d'agents jouant les différents rôles de l'organisation correspondante. Dans MadKit, une groupe est seulement un ensemble d'agents muni d'un nom ; la nouveauté introduite par MOCA est l'explicitation de l'instanciation d'une organisation (i.e. les groupes "savent" quelle organisation ils instancient).

Accointances

La notion d'accointance dans MOCA correspond à son acception classique : les accointances d'un agent sont les agents avec lesquels il peut communiquer.

Le lien entre accointance et relation est le suivant : un agent A peut être en accointance avec un agent A' si et seulement si A et A' ont au moins un groupe G en commun et s'il existe au moins une relation entre un des rôles joués par A et un des rôles joués par A' dans G.

Influences

Les influences sont simplement des instances d'un type d'influence. Comme nous l'avons dit ci-dessus, elles peuvent représenter des forces, des événements, des actes langagiers, etc. qui sont échangés entre les agents le long de leur accointances. Il s'agit d'une généralisation de la notion de message présente dans MadKit.

Rôles

Dans MadKit, un rôle est simplement un nom dans un groupe ; on ne suppose pas que deux rôles de même nom dans différents groupes présentent le même comportement. Dans notre approche, deux rôles portant le même nom dans des groupes instanciant la même organisation auront le même comportement. Par contre, nous incluons une dimension polysémique par le fait que deux rôles de même nom peuvent présenter des comportements différents s'ils appartiennent à des groupes instanciant des organisations différentes (contrairement à l'approche de [PO01]).

Concrètement, les rôles sont des composants dans le sens que nous définirons ci-dessous ; leur comportement est décrit par le statechart de la description de rôle correspondante. Les rôles doivent fournir une méthode *receiveInfluence* que l'agent peut appeler quand il veut leur transmettre des influences venant d'autres rôles. Les influences sont alors transformées en événements de statechart et traitées selon la sémantique habituelle.

Agents

L'aspect interne de notre approche inclut les concepts d'*agent*, de *composant*, de *rôle* et de *compétence*. Les articulations entre ces concepts sont représentés à la figure 5.5.

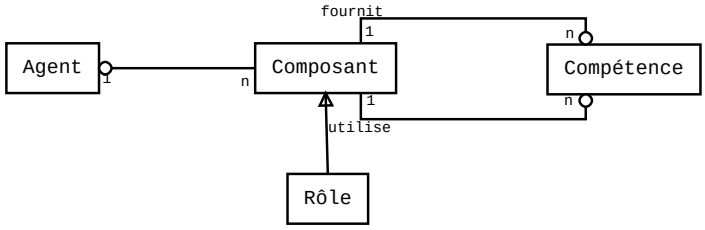


FIG. 5.5 – Les concepts internes

Comme nous l'avons dit ci-dessus, le but de notre modèle est de fournir des primitives à l'agent pour lui permettre de gérer librement les structures organisationnelles auxquelles il prend part. Pour ce faire, nous proposons d'introduire dans les agents un *Module de Gestion des Composants* (MGC) qui prend en charge les tâches suivantes :

1. Recevoir les influences des autres agents et les répartir à ses rôles.
2. Gérer les communications entre ses composants.
3. Fournir à l'agent des primitives lui permettant de gérer ses composants (ajout/retrait).

La première tâche n'est pas problématique : la réception des influences fait partie de la plateforme SMA sous-jacente et les influences sont indexées par leur groupe et leur rôle de destination. Ceci permet un adressage non-ambigu tant qu'un agent ne joue pas simultanément deux fois le même rôle dans le même groupe.

La deuxième tâche est bien plus délicate : le MGC doit lier dynamiquement ses composants, gérer l'exécution de ses rôles et éviter les interférences destructrices entre eux. Ce processus sera discuté séparément au chapitre 6.

La troisième tâche implique aussi quelques subtilités et sera discutée au chapitre 7.

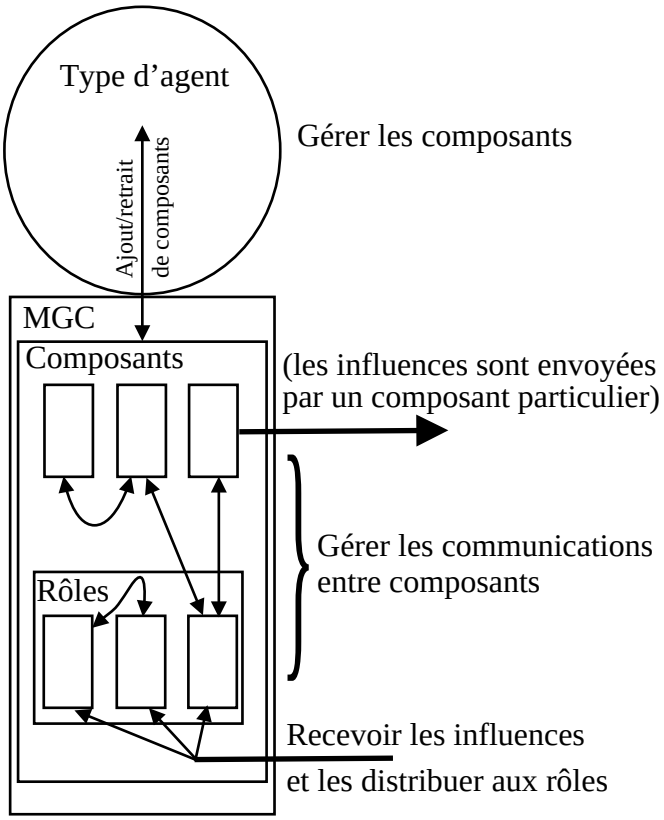


FIG. 5.6 – L'architecture d'un agent MOCA

L'architecture d'un agent MOCA est illustrée à la figure 5.6 : tout le fonctionnement organisationnel et componentiel est encapsulé dans le MGC, qui fournit des primitives permettant à l'agent la gestion de ses composants. L'utilisation de ces primitives dépend par contre du type d'agent et n'est pas couverte par notre approche.

Composants et compétences

[Lhu98] définit la notion de composant comme "une entité définie par un comportement interne et un ensemble de bornes externes (plugs, ports). Le comportement interne n'est pas visible de l'extérieur et n'est accessible qu'au travers des bornes".

Dans MOCA, les bornes d'entrée d'un composant sont les compétences qu'il fournit est les bornes de sortie celles qu'il requiert :

Component

$req\_competences, prov\_competences : \mathbb{P}\ CompetenceDescription$

Le comportement interne des composants dépend évidemment du domaine d'application ; l'idée de l'approche componentielle est justement de s'abstraire de l'implémentation interne des composants pour se reposer sur leur interface. Notons cependant que le comportement de nos composants peut être spécifié par un statechart, de la même manière que nous l'avons fait pour les descriptions de rôles.

Le prochain chapitre sera consacré aux communications entre composants, mais avant de l'aborder, nous désirons encore faire la remarque suivante : pour des raisons de clarté rédactionnelle, nous avons présenté la notion de description de rôle avant celle de composant. Cependant, en revenant sur la définition de description de rôle de la page 60, on se rendra compte que le seul ajout effectué par rapport au formalisme de Vincent Hilaire est l'explicitation des compétences fournies et requises. Object-Z acceptant l'héritage multiple, on pourra donc avantageusement redéfinir la notion de description de rôle de la manière suivante :

MOCARoleDescription

Rôle
Component

Ceci permet de reconnaître au niveau formel le fait qu'un rôle est un cas particulier de composant.

5.4 Quelques considérations sur la réutilisabilité

Après avoir présenté les concepts du modèle MOCA, nous désirons revenir brièvement sur la question de la réutilisabilité.

Il est trop tôt pour savoir si MOCA sera réellement utilisé ; *a fortiori*, il est actuellement impossible de se prononcer sur la réutilisation effective des organisations d'un système MOCA. On pourrait donc penser que la prétention d'une réutilisabilité accrue est usurpée, ou du moins prématurée ; ce serait oublier que MOCA — loin de sortir du néant — propose une systématisation d'un certain nombres de travaux existants.

Dans [Sau], Sylvain Sauvage se base sur une étude approfondie de la littérature pour tenter de dégager des motifs récurrents dans la conception de SMA. Ce travail l'amène à définir deux méta-motifs, qui

> « sont plus que des motifs dans le sens où ils décrivent un principe fondamental du paradigme des systèmes multi-agents. De plus, ces *méta-motifs* engendrent chacun une série de motifs *fils*, véritables instances de leurs géniteurs. »

Les deux méta-motifs de Sauvage sont d'une part les *schémas d'organisation* et d'autre part les *protocoles*. Ces deux méta-motifs sont bien sûr intimement liés, notamment par l'utilisation de la notion de *rôle*.

Le dégagement par Sauvage de ces deux méta-motifs prouve donc une réutilisation effective, avérée, d'un certain nombre d'organisations ou de protocoles. Par contre, on constate que cette réutilisation est assez informelle : elle se limite généralement à une réimplémentation complète ou partielle s'inspirant de travaux déjà réalisés.

Dans ce contexte, MOCA propose un pas en avant significatif : le modèle permet de décrire des organisations et des protocoles sous une forme directement utilisable à l'implémentation. De plus, les contraintes sur la communication entre agents assurent qu'une organisation peut être intégrée à un système sans adaptation préalable.

MOCA permet donc de réutiliser d'un système à l'autre et sans aucune modification des éléments qui étaient jusque là réutilisés régulièrement mais en les adaptant à la main ; il constitue ainsi une avancée sur le chemin de la réutilisabilité dans la conception de SMA.

Chapitre 6

Interactions entre composants et gestion des conflits

6.1 Généralités

Dans ce chapitre, nous présentons les mécanismes d'interaction entre les composants d'un agent dans MOCA. Comme nous l'avons déjà dit, ces interactions sont réalisées à l'aide d'*appels de compétence* ; avant de décrire précisément comment ces appels sont gérés, nous désirons préciser quelles fonctionnalités exactes ce mécanisme devra assurer dans notre système. Celles-ci sont au nombre de trois :

Séparation interactions/actions Les rôles ne décrivent souvent que la structure des interactions et délèguent le traitement effectif à des composants spécialisés. Par exemple, dans la figure 5.4 de la page 61, la description du rôle *initiator* décrit la structure d'interaction nécessaire à un réseau contractuel, mais délègue des fonctionnalités essentielles — l'évaluation des offres, l'envoi des réponses et la gestion du temps — à d'autres composants.

Coordination inter-groupes Dans notre approche, des groupes distincts ne peuvent interagir que par des agents communs. Nous avons donc besoin d'un mécanisme de coordination/collaboration des rôles au sein d'un agent ; ce mécanisme sera localement responsable de la coordination des groupes au niveau du système. Nous verrons un exemple important de ce type de coordination au chapitre 7 en introduisant le *Groupe de Gestion*.

Ignorance mutuelle Les composants peuvent avoir un état interne qui détermine leur comportement externe. Par exemple, de nombreux composants servent à gérer l'usage d'une ressource. La troisième fonction du mécanisme d'interactions entre composants est donc d'éviter les interférences destructrices dans l'utilisation des composants. Pour ce faire, nous introduirons la notion d'*ignorance mutuelle* qui permet à un composant d'utiliser les compétences d'un autre composant comme s'il était tout seul.

Il est à noter que pour assurer l'ignorance mutuelle, ainsi que pour permettre à l'agent de garder le contrôle sur l'exécution de ses composants, tous les appels de compétence transitent par le Module de Gestion des Composants (MGC). Un appel de compétence fait donc intervenir trois entités : le composant appelant, le MGC et le composant appelé.

Remarquons encore que pour simplifier la présentation, nous supposons dans cette thèse qu'il n'y a jamais deux composants distincts fournissant simultanément la même compétence au sein d'un agent. Techniquement, il ne serait pas très difficile de supprimer cette restriction, car ceci ne nécessiterait presque aucun changement dans les algorithmes qui vont suivre. Cependant, il serait nécessaire d'introduire des mécanismes de choix pour déterminer quel composant va être appelé à quel moment. Un tel algorithme dépendant fortement du domaine d'application, nous n'avons pas jugé prioritaire d'inclure cette problématique dans le coeur de MOCA.

Ce chapitre se concentrant sur l'aspect componentiel de notre modèle, les concepts organisationnels vont momentanément passer au second plan ; en particulier, la notion de rôle disparaîtra presque complètement derrière celle de composant. Cependant, on gardera toujours à l'esprit que l'architecture componentielle n'est pour nous qu'un moyen de développer un système organisationnel souple et le chapitre suivant, en traitant de la dynamique des groupes, rendra à la dimension organisationnelle sa place prépondérante dans notre approche.

6.2 L'ignorance mutuelle

Dans notre approche, les composants sont souvent utilisés pour encapsuler des ressources. Par exemple, un agent peut posséder une certaine somme d'argent, ce qui pourrait être représenté par un composant *porte-monnaie*, fournissant des compétences *dépense* et *encaisse* pour modifier cette somme et *argent-disponible* pour la connaître. Si dans cette situation deux composants de l'agent effectuent simultanément une négociation de prix sur la base de cette somme disponible, il y a de fortes chances que l'agent ne possède pas assez d'argent pour honorer les deux paiements.

Bien sûr, une solution "simple" consiste à concevoir les comportements des composants de manière à gérer ce genre de situation. Mais en général, cette approche nécessite de connaître au moment de la conception toutes les interférences qui pourraient survenir avec n'importe quel autre composant dans le système. Cette solution est donc prohibitivement compliquée au niveau de la conception, même si elle est simple du point de vue de la gestion des interférences. Comme un des buts fondamentaux de notre approche est de préserver au maximum l'indépendance entre les organisations — pour en faire des entités réutilisables — nous voulons bien entendu éviter cette pseudo-solution et gérer les éventuels conflits de composants, en particulier les conflits de rôles, au moment de l'exécution.

Pour ce faire, nous introduisons la notion suivante : nous dirons qu'un système assure l'*ignorance mutuelle des composants par rapport aux compétences* si chaque composant peut utiliser les compétences disponibles sans avoir besoin de savoir si et comment d'éventuels autres composants utilisent ces mêmes compétences. Il y a quelques points à relever concernant cette notion :

1. Assurer l'ignorance mutuelle a bien sûr un prix : généralement ce sera le fait que les composants doivent attendre avant de pouvoir accéder à une compétence. Dans un système temps-réel, le simple fait d'attendre peut rompre l'ignorance mutuelle, mais nous ne considérerons pas les systèmes temps-réel ici.
2. L'ignorance mutuelle n'est pas équivalente à l'exclusion mutuelle sur les appels de compétence ; le fait d'autoriser un seul composant simultanément à faire un appel à une compétence donnée serait en même temps trop fort et trop faible : trop fort car dans certaines situations (comme l'accès à une variable en lecture seule) aucune exclusion

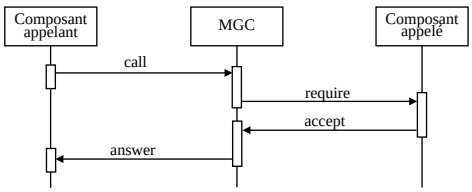
n'est nécessaire ; trop faible car dans d'autres situations (comme demander le montant disponible et payer après) l'exclusion doit s'étendre sur plusieurs appels successifs.

3. L'ignorance mutuelle n'est pas équivalente à l'exclusion mutuelle sur l'exécution des composants : le fait de n'autoriser l'exécution que d'un seul composant à la fois serait clairement trop fort, car seules les portions de l'exécution qui sont en compétition pour une compétence donnée doivent être considérées.

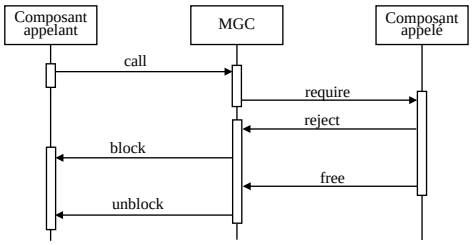
Le cadre étant maintenant bien posé, nous pouvons présenter les algorithmes assurant la communication des composants dans MOCA.

6.3 Interactions entre composants

Le mécanisme d'interactions entre composants comporte trois protocoles qui sont représentés aux figures 6.1 à 6.3. Le *protocole d'appel* en constitue le coeur et peut se dérouler de deux manières différentes suivant si l'appel est accepté ou refusé ; les *protocoles de relâchement* et de *renoncement* sont tous deux déclenchés par le composant appelant lorsqu'il n'a plus besoin d'une compétence. La différence entre les deux est que le *relâchement* est utilisé après le dernier appel (accepté) à une compétence pour traiter des cas comme celui de l'exemple du porte-monnaie ci-dessus, alors que le *renoncement* est utilisé pour des compétences que le composant appelant est en train d'attendre (après un appel refusé).



(a) appel accepté



(b) appel refusé

FIG. 6.1 – Le protocole d'appel

Ces protocoles correspondent à un appel synchrone de compétence : le composant appelant est bloqué jusqu'à réception de la réponse ou du refus. Le cas asynchrone est similaire mais

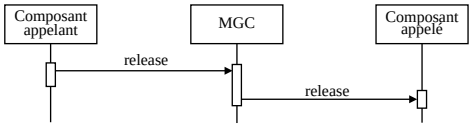


FIG. 6.2 – Le protocole de relâchement

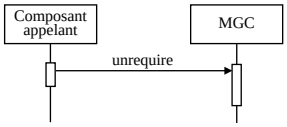


FIG. 6.3 – Le protocole de renoncement

comporte une étape de plus : il commence par un appel synchrone dont le retour est l'acceptation ou le refus de l'appel (et non la réponse) ; en cas d'acceptation de l'appel, le composant appelant peut continuer son exécution en attendant la réponse du composant appelé.

Nous allons maintenant décrire plus en détail comment chacune des entités en présence implémente ces protocoles, en commençant par le composant appelé.

L'algorithme du composant appelé

Quand un composant reçoit un appel de compétence (*require*), il doit déterminer s'il l'accepte (*accept*) ou pas. En cas de refus (*reject*), il a la responsabilité d'informer le MGC quand il est à nouveau prêt à l'accepter (*free*). La politique d'acceptation dépend entièrement du composant appelé et de son utilisation prévue ; nous donnons ci-après quelques exemples possibles de politique d'acceptation :

Envoi La compétence standard *send*, qui sert à envoyer des messages à d'autres agents, peut accepter tous les appels, car elle n'a *a priori* pas de raison d'être source de conflit. Notons cependant que l'implémentation de cette compétence pourrait tout à fait donner des raisons de refuser certains appels : par exemple, si les messages sont stockés dans un tampon en attendant leur envoi, les appels doivent être rejetés quand le tampon est plein. Le composant doit alors envoyer un message *free* au MGC lorsque le tampon est de nouveau utilisable.

Porte-monnaie Dans l'exemple du porte-monnaie du paragraphe 6.2, une politique d'acceptation correcte est d'accepter le premier appel — disons du composant C — et de refuser tout appel ne provenant pas de C jusqu'à ce que C effectue son *dernier* appel à la compétence *dépense*. Dans ce cas, on dit que les compétences sont *réservées* pour C. Ceci permet à C de consulter le montant disponible et de pouvoir compter sur cette information jusqu'à ce qu'il n'en ait plus besoin. Le composant appelé restera dans cet état *réservé* jusqu'à ce que C envoie un *release* concernant la compétence *dépense*, signalant ainsi qu'il n'a plus besoin de la réservation. Le composant appelant doit alors envoyer un *free* au MGC.

Porte-monnaie revisité Dans certains cas, on peut raffiner la politique ci-dessus en acceptant tout appel à la compétence *encaisse* indépendamment de la réservation de la

compétence *paie*. Ceci ne posera pas de problème tant que l'important est d'avoir *assez* d'argent et que le fait d'en avoir *trop* n'est pas gênant.

Les exemples ci-dessus montrent que la localisation de la politique d'acceptation dans chaque composant permet une grande flexibilité. Bien sûr, le but est généralement d'avoir un taux de refus aussi bas que possible pour augmenter l'efficacité du système.

L'algorithme du MGC

A première vue, en observant les figures 6.1 à 6.3, il peut sembler que le MGC fait simplement office de relais dans le mécanisme d'appel de compétence : en recevant un *call*, il transmet un *require*, en recevant un *accept*, il transmet un *answer*, etc.

Il est vrai que dans le cas d'un appel accepté, le rôle du MGC est négligeable. Cependant, il a un rôle tout à fait central, qui n'apparaît pas dans les protocoles eux-mêmes, lorsque des appels sont rejetés : il s'occupe en effet de la gestion des listes d'attente.

En recevant un *reject*, le MGC notifie le composant appelant[1] et il le met dans une liste d'attente (généralement une par composant appelé). Lorsqu'il reçoit le message *free*, il doit choisir un composant dans la liste d'attente et lui envoyer un *unblock*. Or ce choix est un point tout à fait crucial, puisque c'est là que peuvent être gérés les problèmes classiques du calcul parallèle tels qu'interblocage (*deadlock*) et famine (*starvation*). Il n'y a malheureusement pas de solution générale à ces problèmes, mais il existe un certain nombre d'approches standard (cf. par exemple [SP88]) qui couvrent la majorité des cas que l'on peut rencontrer.

On notera qu'au moment de l'exécution, la gestion des interblocages et de la famine est un pur problème d'allocation de ressources : les blocages dus à de mauvais protocoles de communication devraient avoir été éliminés au moment de la conception. Ainsi le MGC doit "simplement" allouer les compétences à ses composants d'une manière qui leur permette de suivre les protocoles pour lesquels ils sont conçus. Dans le cas simple où l'agent n'a qu'un seul rôle dans chaque groupe auquel il participe, le problème peut être traité avec des solutions comme les *graphes d'allocation de ressources* [SP88], puisque les rôles ne peuvent pas échanger de messages quand ils ne sont pas dans un même groupe. Si par contre l'agent a plusieurs rôles dans un même groupe, le MGC doit également prendre en compte les éventuels échanges de messages entre ces différents rôles, ce qui complique d'autant le problème.

Quand le MGC reçoit un *release* du composant appelant, il le transmet simplement au composant appelé, et quand il reçoit un *unrequire*, il retire le composant appelant de la liste d'attente correspondante. Nous allons maintenant voir quand ces messages sont générés.

L'algorithme du composant appelant

Quand un composant a besoin d'une compétence, il envoie un message *call* au MGC. Si l'appel est accepté, il reçoit la réponse (*answer*) et son exécution continue. Dans le cas contraire, il reçoit un message *block*.

On peut penser que le plus simple est alors de suspendre l'exécution du composant jusqu'à ce que la compétence soit à nouveau disponible. Cependant dans de nombreux cas cette solution est beaucoup trop stricte ; si le comportement du composant est décrit par un statechart, il est possible de définir une réaction beaucoup plus fine que la simple suspension de l'exécution. La figure 6.4 en donne un exemple : elle représente la situation où une action doit

[1] Nous verrons ci-dessous pourquoi nous envoyons une notification plutôt que de simplement bloquer l'exécution du composant.

être exécutée dans un certain délai après lequel une autre transition agit préemptivement. Si le composant tente d'exécuter l'*action* et que son appel est refusé, il n'est pas correct de simplement suspendre son exécution : en effet, si le *timeout* arrive avant la reprise de l'exécution, celui-ci sera ignoré et le composant va à la fois manquer la suite de son exécution et effectuer l'*action* lorsqu'elle n'est plus adéquate.

Pour éviter ce problème, le composant appelant marque la transition concernée comme étant *non disponible* et continue son exécution. Il peut éventuellement déclencher une autre transition (disponible) ou attendre si aucune n'est déclenchable. Lors de la réception de *free*, la transition est à nouveau marquée comme disponible et l'exécution continue. Si par contre le composant déclenche une autre transition avant de recevoir le message *free*, il doit marquer la transition bloquée comme disponible et envoyer un *unrequire* au MGC pour lui signaler qu'il n'a plus besoin de la compétence concernée[2].

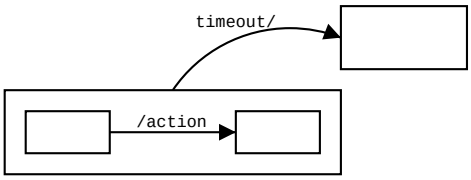


FIG. 6.4 – Un exemple de statechart avec préemption

Quand un appel de compétence a été accepté, le composant appelant a la charge d'envoyer un *release* après son dernier appel à cette compétence. Ceci sert à gérer les situations telles que l'exemple du porte-monnaie du paragraphe 6.2.

Si le comportement du composant est décrit par un statechart, il est possible de déterminer automatiquement quand ce *release* doit être envoyé. L'idée est de garder à jour au cours de l'exécution du statechart une liste des compétences susceptibles d'être encore appelées. Quand une compétence disparaît de cette liste, le *release* correspondant est envoyé.

La fin de ce paragraphe est consacrée à l'exposition détaillée de cet algorithme ; mais avant cela, nous désirons faire quelques remarques sur la manière dont il a été établi :

Comportements cycliques Si le composant a une activité cyclique, il n'y aura jamais de "dernier appel" d'une compétence. Celle-ci va donc rester indisponible pour d'autres composants aussi longtemps que le composant appelant fonctionne, provoquant un risque clair de famine. Pour éviter cette situation, nous décidons de relâcher (*release*) toutes les compétences utilisées par un composant chaque fois qu'il retourne à son état initial. Du point de vue de l'ignorance mutuelle, nous traitons donc les comportements cycliques comme des comportements "one-shot" itérés. Il faut noter cependant que ceci ne concerne pas les cycles "internes", à savoir ceux qui ne reviennent pas à l'état initial.

Relâchement manuel Dans certains cas, la détection syntaxique de zones critiques peut être impossible. Par exemple, si un composant achète plusieurs objets, alternant entre négociation de prix et paiement, il est préférable de relâcher la compétence *porte-monnaie* après chaque paiement, même si l'analyse syntaxique du comportement pousse

[2]Ceci ne doit bien sûr être effectué que si la transition déclenchée provoque un changement d'état. Dans le cas d'une transition dont la source et le but sont le même état, il est préférable de laisser la transition dans son état *non disponible*.

à un unique relâchement final. Pour gérer ce genre de situation, le concepteur d'un statechart peut y insérer manuellement des points de relâchement.

Graphe des transitions possibles Un statechart est essentiellement la représentation implicite d'un automate à états finis (avec priorités sur les transitions et quelques autres extensions). Cependant, comme l'automate "déployé" peut être très grand, nous voulons éviter de le considérer explicitement. Ainsi, pour détecter le dernier appel d'une compétence, nous allons considérer un "graphe des transitions possibles" entre les états du statechart au lieu de la vraie relation de succession sur les configurations. La sémantique de la relation "transition possible" est la suivante : un état A est en relation avec un état B si et seulement si le fait que A est actif peut provoquer le déclenchement d'une transition résultant en l'activation de B. Cependant, pour éviter une explosion combinatoire, les circonstances nécessaires au déclenchement de cette transition (p.ex. le fait qu'un autre état C soit aussi actif) ne sont pas explicitées. Cela signifie qu'en certaines occasions, le relâchement d'une compétence peut se produire un peu plus tard que ce qui est réellement nécessaire, mais l'efficacité de l'ensemble du processus en est clairement améliorée. Nous étudierons au paragraphe 6.4 les conséquences exactes de cette simplification.

Considérons maintenant un statechart SC ; une illustration de l'algorithme sera donnée sur la base du statechart de la figure 6.5.

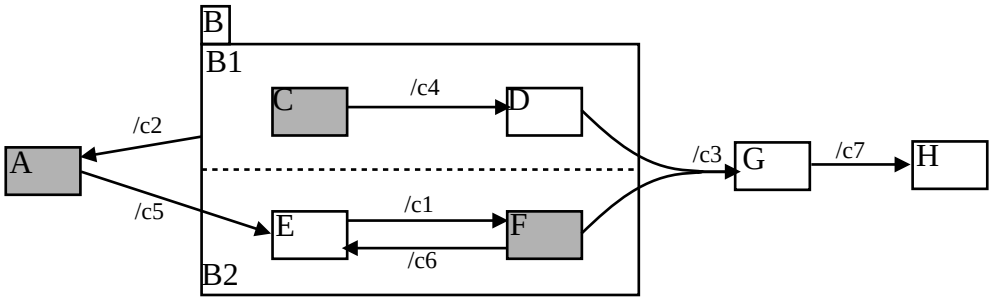


FIG. 6.5 – Le statechart SC de départ

Soient S l'ensemble des états du statechart et Tr l'ensemble de ses transitions. On a alors deux fonctions $s, b : Tr \rightarrow \wp(S)$ qui donnent pour chaque transition $tr \in Tr$ l'ensemble de ses états source $s(tr) \subset S$ et celui de ses états buts $b(tr) \subset S$. Relevons qu'en général $b(tr)$ est inclus dans l'ensemble "réel" des buts de tr. Par exemple, sur la figure 6.5, la transition étiquetée par $/c5$ possède seulement un but explicite $b(tr) = E$ mais son déclenchement active l'ensemble d'états $\{B, B1, B2, C, E\}$. Nous noterons $\overline{b(tr)}$ l'ensemble complet des buts (explicites *et* implicites) de la transition tr. Pour calculer $\overline{b(tr)}$, il suffit de compléter $b(tr)$ jusqu'à obtenir une configuration du sous-statechart dont la racine est la *portée* de tr (cf. chapitre 3).

On peut maintenant construire le graphe $TP(SC)$ des transitions possibles de SC :

1. On commence avec l'ensemble des noeuds N de $TP(SC)$ égal à l'ensemble S des états de SC. Notons que le fait de parler d'*ensembles* implique que l'on "oublie" la structure hiérarchique des états de SC. On peut à ce stade retirer de N les *états-ou* "implicites"

de SC, c'est-à-dire ceux qui sont directement inclus dans des *états-et*, comme $B1$ et $B2$ dans la figure 6.5[3].

2. Dans le graphe PT, on rajoute une arrête entre les noeuds n et m si et seulement s'il existe une transition $tr \in Tr$ dans SC avec $n \in s(tr)$ et $\overline{b(tr)}$. Si tr appelle une compétence E, on associe une étiquette $\text{call}(n, m) = E$ à l'arrête (n, m).
3. Soit i l'état initial de SC. On ajoute un noeud i' à N et on remplace chaque arrête de la forme (n, i) par une arrête de la forme (n, i'). Ceci permet de traiter les comportements cycliques comme des comportements "one-shot".

On a ainsi obtenu un graphe $PT(SC)$ qui décrit les transitions possibles entre les états de notre statechart SC. La figure 6.6 illustre le graphe des transitions possibles du statechart de la figure 6.5.

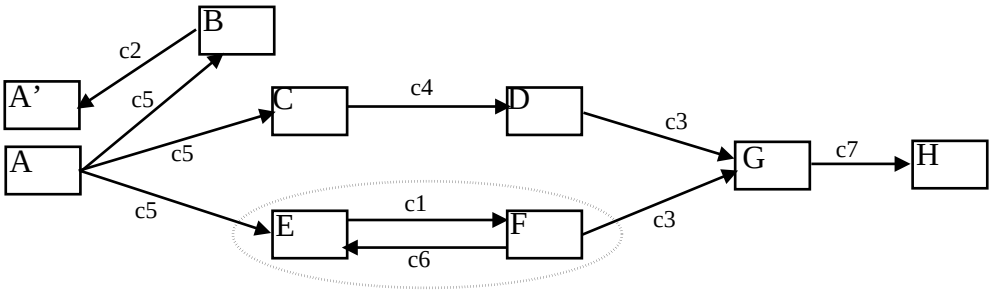


FIG. 6.6 – Le graphe des transitions possibles

L'étape suivante est de construire le graphe réduit $PT'(SC)$ obtenu à partir de $PT(SC)$ en identifiant les noeuds appartenant à la même composante fortement connexe. Il existe différents algorithmes plus ou moins classiques pour ce faire, comme par exemple celui de [AHU87]. Ensuite, on associe à tout noeud N' de $PT'(SC)$ correspondant à un ensemble N_i de noeuds et un ensemble A_i d'arêtes de PT une étiquette $\text{calls}(N') = \cup\,\text{call}(A_i)$. La figure 6.6 contient une composante fortement connexe non triviale qui est entourée d'un pointillé et qui sera réduite en un seul état sur la figure 6.7. A la fin de cet étape, on aura donc $\text{calls}(E + F) = c1, c6$.

Enfin on effectue un parcours récursif en profondeur de $PT'(SC)$, en commençant par l'état initial, au moyen de la fonction *listCalls* ci-dessous. Chaque noeud doit posséder un attribut booléen *visited*, initialisé à *faux*, et l'attribut *calls* que nous venons d'utiliser ci-dessus. À la fin du parcours, l'attribut calls d'un noeud n contiendra l'ensemble des compétences qui peuvent éventuellement être appelées après que le statechart a été dans l'état n.

listCalls(s:state):list

1. si *s.visited* alors retourne *s.calls*
2. pour tous les successeurs *succ* de *s* faire *s.calls* :=*s.calls*∪listCalls(*succ*)
3. pour toutes les arrêtes *a* de la forme (*s,s'*) faire *s.calls* :=*s.calls*∪call(*a*)
4. *s.visited*=*vrai*

[3]Il n'est pas strictement nécessaire de retirer ces états, mais comme ils n'ont jamais de transition entrante ou sortante, ils ne jouent aucun rôle dans l'algorithme qui va suivre.

5. retourne *s.calls*

Le résultat de ce parcours dans le cas de notre exemple est présenté à la figure 6.7.

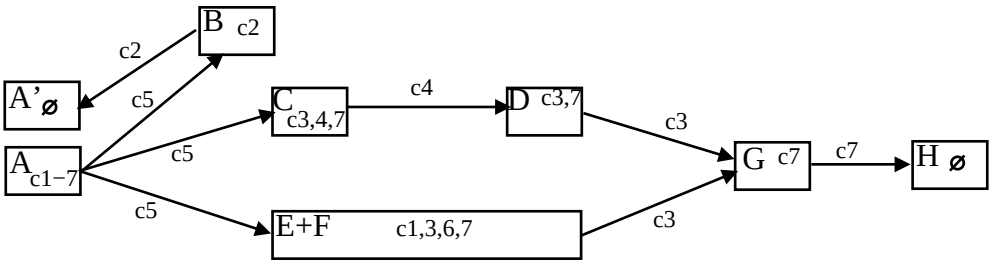


FIG. 6.7 – Le graphe réduit après le parcours récursif

Il ne reste plus qu'à reporter les étiquettes calls des noeuds de $PT'(SC)$ sur les états du statechart d'origine. Une manière élégante et facile de faire cela est de se contenter d'une construction implicite de $PT'(SC)$ par une simple annotation de SC, ce qui permet de sauter cette dernière étape.

Toute cette annotation du statechart peut se faire juste après la conception du composant ; au moment de l'exécution, elle sera utilisée de la manière suivante : après chaque transition, le composant compare la réunion des attributs calls des états appartenant à sa configuration précédente avec la réunion des calls de sa configuration courante. Si le nom d'une compétence a disparu, le composant envoie un *release* au MGC. De plus, lors d'un retour à la configuration initiale, le composant relâchera toutes les compétences qu'il n'a pas encore relâchées.

6.4 Justification et discussion

Ce paragraphe propose quelques réflexions sur les différents algorithmes que nous avons exposés au paragraphe précédent. Nous commencerons par prouver la correction de l'algorithme de relâchement ; nous mettrons ensuite en évidence le fait que l'ensemble des algorithmes ci-dessus permet effectivement d'assurer l'ignorance mutuelle des composants par rapport aux compétences ; enfin, nous terminerons par quelques considérations sur la puissance du mécanisme décrit et les possibilités d'extensions.

Preuve de l'algorithme de relâchement

Nous avons proposé au paragraphe 6.3 un algorithme pour automatiser le relâchement d'une compétence dans le cas d'un rôle décrit par un statechart (SC). Pour ce faire, nous avons introduit un objet intermédiaire, le graphe des transitions possibles (PT), sur lequel nous avons effectué des calculs. Cependant, pour décrire les exécutions possibles du rôle, nous avons besoin d'une troisième structure, que nous nommerons le *graphe des configurations* (GC). Il s'agit du graphe dont les noeuds sont les configurations de SC et dont les arcs correspondent aux ensembles non conflictuels de transitions de SC (cf. paragraphe 3.1). On notera que la construction de GC ne tient pas compte des événements déclencheurs ni des conditions sur les transitions de SC.

Pour clarifier les choses, nous relevons rapidement quelques points importants de chacune des trois structures que nous considérons :

1. SC est formé d'états organisés en une structure hiérarchique et-ou, et de transitions ayant pour source et but des ensembles d'états (cf. paragraphe 3.1). Pour une transition tr, nous noterons $s(tr) = \{E_i\}$ et $b(tr) = \{E_j\}$ l'ensemble de ses états sources et buts respectivement. Alternativement, nous abrégerons ces deux formules sous la forme de l'expression $tr : \{E_i\} \to \{E_j\}$. Nous noterons call(tr) la compétence appelée lors du déclenchement de tr.
2. PT est obtenu par un calcul formel à partir de SC (cf. paragraphe 6.3) ; il possède le même ensemble d'états que SC, mais sans structure ; les transitions de PT ont un seul état source et un seul état but. Il existe dans PT exactement une transition t pour chaque couple $(E, E') \in s(tr) \times b(tr)$ de chaque transition tr de SC. Comme pour SC, nous noterons $t : E \to E'$, $s(t) = E$ et $b(t) = E'$. Étant donné qu'à chaque transition de PT correspond exactement une transition de SC, on peut étendre la fonction call ci-dessus aux transitions de PT.
3. GC est défini par la sémantique opérationnelle des statecharts (cf. paragraphe 3.1) ; il est formé de configurations, qui sont des ensembles d'états de SC compatibles avec la structure et-ou, et de transitions qui correspondent en général à des ensembles de transitions de SC. Nous décrirons comme ci-dessus la source et le but d'une transition T par $s(T) = C$ et $b(T) = C'$ ou $T : C \to C'$. Nous définissons calls(T) comme étant la réunion des call(tr) pour les tr correspondant à T.

Dans cette situation relativement complexe, il peut être difficile de se convaincre que l'algorithme de relâchement exposé ci-dessus donne effectivement le résultat voulu. Nous allons donc démontrer deux propriétés du marquage défini par notre algorithme (théorème 1) desquelles nous déduirons la correction du relâchement.

Avant d'aborder la démonstration proprement dite, nous désirons préciser quelques points :

- Dans ce qui suit, nous supposerons que SC n'est pas *globalement cyclique*, c'est à dire que son état initial n'est but d'aucune transition. Si ce n'est pas le cas, on se ramènera à cette situation en dupliquant l'état initial (selon la même technique que celle que nous avons utilisée pour la figure 6.6).
- Pour simplifier les notations, nous supposerons que les transitions de SC mentionnent explicitement tous leurs buts. Avec les notations du paragraphe précédent, cette hypothèse s'écrit $\forall tr : b(tr) = \overline{b(tr)}$. On notera cependant que cette supposition n'est pas indispensable au raisonnement qui va suivre, et l'on peut s'en passer à condition de remplacer chaque occurrence de $b(tr)$ par $\overline{b(tr)}$.
- L'algorithme de relâchement exposé ci-dessus calcule un étiquetage des états E de PT que nous avons noté calls(E). Cet étiquetage s'étend sans peine aux états de SC (puisque ce sont les mêmes) et aux configurations de GC (par réunion).

Nous commençons par une série de lemmes clarifiant les rapports entre PT et GC :

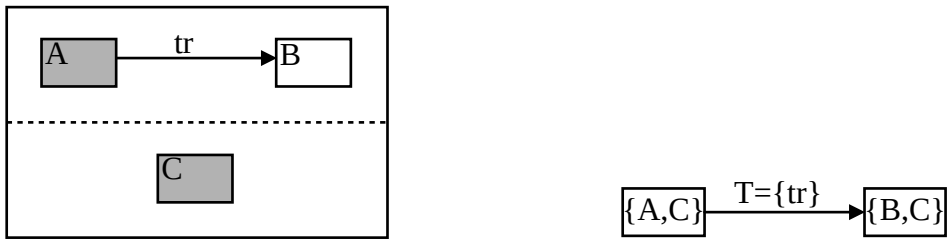
Lemme 1. *Soient $T : C \to C'$ une transition de GC et $a \in$ calls(T). Alors il existe une transition $t : E \to E'$ dans PT avec $E \in C$, $E' \in C'$ et* call(t) $= a$.

Démonstration. Par construction de GC, il existe une transition tr dans SC avec call(tr) $= a$, $s(tr) \subset C$ et $b(tr) \subset C'$. Par construction de PT, il existe pour tout $(E, E') \in s(tr) \times b(tr)$ une transition $t : E \to E'$ dans PT avec call(t) $= a$. □

Nous aurions également besoin d'un résultat semblable :

Lemme 2. *Soient $T : C \to C'$ une transition de GC et $E' \in C'$. Alors il existe une transition t et un état E dans PT avec $t : E \to E'$ et $E \in C$.*

Malheureusement, ce résultat n'est pas vrai en général. La figure 6.8 en donne un contre-exemple minimal : le statechart en (a) correspond au graphe des configurations en (b). On voit facilement que le lemme 2 n'est pas vérifié pour $E' = C : C$ n'est le but d'aucune transition.



(a) *SC* (b) *GC*

FIG. 6.8 – Un contre-exemple au lemme 2

Cependant, ce lemme peut être rendu vrai en ajoutant l'hypothèse que tout état E de SC possède une transition $id_E : E \to E$ d'étiquette vide. Cette supposition ne change rien à l'exécution du statechart : elle ne fait qu'exprimer dans le statechart lui-même qu'il est possible, lors d'un pas d'exécution, de ne rien faire dans un descendant d'un état-et donné. Nous supposerons donc sans perte de généralité que cette hypothèse est vérifiée, ainsi donc que le lemme.

Lemme 3. *Soient C une configuration de GC, $E \in C$ un état de PT et $t : E \to E'$ une transition de PT avec* $\mathrm{call}(t) = a$. *Alors il existe une transition T de GC de source C et vérifiant* $a \in \mathrm{calls}(T)$.

Démonstration. Par construction de PT, il existe une transition tr de SC avec $E \in s(tr)$, $E' \in b(tr)$ et $\mathrm{call}(tr) = a$. Par construction de GC il existe donc une transition T de source C avec $a \in \mathrm{calls}(T)$. □

Lemme 4. *Soient C une configuration de GC, $E \in C$ un état de PT et $t : E \to E'$ une transition de PT. Alors il existe une transition $T : C \to C'$ dans GC avec $E' \in C'$.*

Démonstration. Par construction de PT, il existe une transition tr de SC avec $E \in s(tr)$ et $E' \in b(tr)$. Par construction de GC il existe une transition $T : C \to C'$ avec $s(tr) \subset C$ et $b(tr) \subset C'$, d'où le résultat. □

Nous passons maintenant à un résultat sur l'étiquetage calls des étiquettes de PT qui nous servira d'amorce pour le théorème qui suivra :

Lemme 5. *Pour tout état E de PT, on a* $\mathrm{calls}(E) = \bigcup_{s(t)=E} (\mathrm{call}(t) \cup \mathrm{calls}(b(t)))$.

Démonstration. Cette propriété est évidente sur le marquage du graphe réduit PT'. Pour la vérifier sur PT, il suffira donc de vérifier cette propriété sur les composantes fortement connexes ; ceci découle du fait que le marquage y est constant et contient toutes les actions effectuées à l'intérieur de la composante. □

On peut maintenant montrer le résultat central de la preuve :

Théorème 1. *Pour toute configuration C de GC, on a*

$$\mathrm{calls}(C) = \bigcup_{s(T)=C} (\mathrm{calls}(T) \cup \mathrm{calls}(b(T)))$$

Démonstration. Par construction, on a

$$\begin{array}{rcll} \mathrm{calls}(C) & = & \bigcup_{E\in C} \mathrm{calls}(E) & \text{dans } SC \\ & = & \bigcup_{E\in C} \mathrm{calls}(E) & \text{dans } PT \end{array}$$

Le lemme 5 permet de récrire $\mathrm{calls}(E)$, ce qui nous donne

$$\mathrm{calls}(C) = \bigcup_{E\in C} \left(\bigcup_{s(t)=E} (\mathrm{call}(t) \cup \mathrm{calls}(b(t))) \right).$$

Il faut donc montrer que

$$\bigcup_{s(T)=C} (\mathrm{calls}(T) \cup \mathrm{calls}(b(T))) = \bigcup_{E\in C} \left(\bigcup_{s(t)=E} (\mathrm{call}(t) \cup \mathrm{calls}(b(t))) \right)$$

ce que nous ferons par double inclusion.

$\boxed{\subset}$ Soit $a \in \bigcup_{s(T)=C} (\mathrm{calls}(T) \cup \mathrm{calls}(b(T)))$. Deux cas peuvent se présenter :

1. Il existe une transition T dans GC telle que $s(T) = C$ avec $a \in \mathrm{calls}(T)$. Dans ce cas, il existe $t : E \to E'$ dans PT avec $E \in C$, et $\mathrm{call}(t) = a$ (lemme 1), d'où le résultat.
2. Il existe $T : C \to C'$ dans GC avec $a \in \mathrm{calls}(C')$. Donc il existe $E' \in C'$ dans PT avec $a \in \mathrm{calls}(E')$. Le lemme 2 nous assure l'existence de $t : E \to E'$ dans PT avec $E \in C$, montrant du même coup l'inclusion.

$\boxed{\supset}$ Soit $a \in \bigcup_{E\in C} \left(\bigcup_{s(t)=E} (\mathrm{call}(t) \cup \mathrm{calls}(b(t))) \right)$. Donc il existe un état $E \in C$ de PT avec $a \in \bigcup_{s(t)=E} (\mathrm{call}(t) \cup \mathrm{calls}(b(t)))$. Nous considérons à nouveau deux situations :

1. Il existe une transition t dans PT avec $s(t) = E$ et $\mathrm{call}(t) = a$. Par le lemme 3, il existe $T : C \to C'$ avec $a \in \mathrm{calls}(T)$, ce qui permet de conclure dans ce cas.
2. Il existe un état E' de PT et une transition $t : E \to E'$ avec $a \in \mathrm{calls}(E')$. Il existe donc (lemme 4) $T : C \to C'$ avec $E' \in C'$, ce qui complète la preuve.

□

De ce résultat on déduit immédiatement deux propriétés fondamentales de l'étiquetage fourni par notre algorithme :

Corollaire. *La fonction* calls : $GC \to A$ *vérifie les deux propriétés suivantes :*

1. *Pour toute configuration* C *et toute transition* T *de source* C*, on a* $\text{calls}(T) \subset \text{calls}(C)$.
2. calls *est décroissante, dans le sens suivant : Soient* C, C' *des configurations telles qu'il existe une transition* $T : C \to C'$*. Alors* $\text{calls}(C) \supset \text{calls}(C')$.

La première propriété nous assure qu'il n'y aura pas d'appel de compétence inattendu : si le déclenchement d'une transition provoque un appel de compétence, celui-ci est listé dans l'étiquette calls de la configuration source. La seconde propriété assure que lorsqu'une compétence a disparu de la liste, elle n'y réapparaîtra plus, et donc qu'on peut la libérer.

Ces deux propriétés ensemble prouvent donc que notre algorithme est correct : les compétences ne seront jamais relâchées trop tôt. Cependant, on pourrait craindre que le relâchement ne se fasse trop tard, bloquant ainsi inutilement des compétences. Le corollaire suivant apporte une réponse à cette question :

Corollaire. *La fonction* calls : $GC \to A$ *est la fonction minimale réunissant les deux propriétés ci-dessus, dans le sens suivant : toute fonction* $c : GC \to A$ *possédant simultanément les deux propriétés du corollaire vérifie* $\text{calls}(C) \subset c(C)$ *pour toute configuration* C *de* GC.

Démonstration. Considérons une fonction $c : GC \to A$ telle qu'il existe une configuration C de GC avec $c(C) \subset \text{calls}(C)$ et $c(C) \neq \text{calls}(C)$. Donc il existe $a \in \text{calls}(C) = \bigcup_{s(T)=C} (\text{calls}(T) \cup \text{calls}(b(T)))$ avec $a \notin c(C)$. Si $a \in \text{calls}(T)$, c ne vérifie pas la première propriété ci-dessus, alors que si $a \in \text{calls}(b(T))$, c n'est pas décroissante. □

Notre algorithme, en effectuant les calculs sur PT, permet de ne jamais construire explicitement GC en entier ; si cette solution est économique, on pourrait craindre que la perte d'information entre GC et PT ne provoque des relâchements tardifs. Ce résultat prouve qu'il n'en est rien : le marquage obtenu est aussi bon que n'importe quel autre obtenu directement sur GC.

On relèvera cependant que GC lui même ne contient pas toutes les informations du statechart de départ : lors de sa construction, nous n'avons tenu compte ni des conditions ni des événements déclencheurs des transitions de SC. Cependant, il est évident que la prise en compte des conditions provoque en général une explosion combinatoire, puisqu'elle nécessite de considérer les variables du rôle et éventuellement d'autres éléments de contexte. Quant aux événements déclencheurs, comme ils consistent en des perturbations venant d'autres agents, leur prise en compte nécessite une connaissance du SMA dans son entier.

Ces différentes considérations permettent donc d'affirmer que notre algorithme de relâchement des compétences présente un bon rapport entre la complexité du calcul et la finesse du résultat.

Ignorance mutuelle

Au paragraphe 6.2, nous avons défini l'ignorance mutuelle par le fait qu'un composant peut utiliser les compétences disponibles sans avoir besoin de savoir si et comment d'éventuels autres composants utilisent ces mêmes compétences.

Selon cette définition, le mécanisme décrit ci-dessus permet effectivement d'assurer l'ignorance mutuelle : le composant appelant n'a besoin d'aucune connaissance sur d'éventuels concurrents. Pour en arriver là, nous avons dû introduire la possibilité qu'un appel de compétence soit refusé et la nécessité de signaler le dernier appel à une compétence donnée. En

toute généralité, la gestion de ces deux nouveautés peut se révéler délicate ; cependant, dans le cas où le comportement du composant est décrit par un statechart, nous avons proposé des manières d'automatiser complètement ces tâches. Ainsi, lors du développement d'un composant décrit par un statechart, le concepteur peut ignorer complètement la problématique de la gestion des conflits.

On notera cependant que lors du développement d'un composant *fournissant* des compétences, il est nécessaire de définir une politique d'acceptation des appels ; à ce niveau, on ne peut donc plus ignorer la gestion des conflits. Ceci résulte d'un choix délibéré : la diversité des situations pouvant mener à un conflit dans l'utilisation des compétences étant très grande, nous avons préféré laisser au concepteur le choix de la politique la plus adaptée. De plus, comme nous avons défini l'ignorance mutuelle en termes d'utilisation des compétences et non en termes de mise à disposition, cet état de fait ne brise pas l'ignorance mutuelle de notre système.

Le manque de méthode générique pour gérer les interblocages et la famine au niveau du MGC peut apparaître comme un point faible de notre architecture : même s'il est envisageable de construire un MGC suffisamment évolué pour gérer la majorité des situations susceptibles de se présenter, il semble impossible dans l'état actuel de nos connaissances de proposer un mécanisme parfaitement générique dans ce cas. Il s'agit cependant d'un problème inévitable dès que l'on aborde un problème de gestion de ressources par l'exclusion mutuelle ; une solution évitant cet écueil, si elle est possible, devrait aborder le problème sous un angle radicalement nouveau.

Discussion

Le mécanisme de communication entre composants que nous avons exposé dans ce chapitre permet, grâce à une séparation claire des partie traitées par le composant appelant, le MGC et le composant appelé, d'assurer l'ignorance mutuelle des composants par rapport aux compétences d'une manière à la fois générique et souple.

On pourrait être tenté de relever que cet algorithme, s'il permet d'éviter les conflits lors de l'accès aux compétences, ne permet pas de planifier l'usage des compétences lorsqu'elles représentent des ressources consommables. Par exemple, dans le cas du *porte-monnaie* du paragraphe 6.2, notre algorithme assure que les deux composants appelants ne vont pas simultanément négocier des prix qui, ensemble, représentent plus que l'avoir du composant appelé ; il n'y aura donc pas d'erreur d'exécution. Cependant, comme une des négociation va être retardée, il peut arriver qu'il ne reste plus d'argent au moment où elle pourra s'effectuer.

Il est vrai que cette limitation est une conséquence de notre approche : l'opacité étant un élément définitoire du composant, le mieux que nous puissions assurer est l'ignorance mutuelle. Demander à un composant de prendre en compte dans sa négociation le fait qu'il doit laisser de l'argent pour un autre composant constitue une approche fondamentalement différente, qui rompt l'indépendance conceptuelle entre ces entités, diminuant d'autant leur réutilisabilité.

Il existe toutefois au moins deux manières d'étendre notre modèle, sans renoncer à l'indépendance entre composants, pour inclure une forme de planification de ressources :

- La première consiste simplement à développer un composant *porte-monnaie* muni d'une gestion plus fine de la réservation : s'il est possible, par exemple, de ne réserver qu'une partie du montant disponible, cela permet à deux négociations de cohabiter.
- La deuxième consiste à ce que l'agent "trompe" les composants appelants — par exemple

en indiquant au premier composant appelant une somme plus petite que celle qui a été communiquée par le composant appelé, de manière à ce qu'il reste de l'argent pour le deuxième composant appelant. Comme tous les appels et réponses transitent par le MGC, il "suffirait" de développer un MGC fournissant à l'agent des primitives lui permettant une telle gestion et un type d'agent capable de les utiliser.
Cette solution constitue une extension tout à fait plausible de notre mécanisme ; cependant, comme elle nécessiterait une grande sophistication de l'agent et du MGC, nous ne la développerons pas plus dans cette thèse.

6.5 Conclusion

Les mécanismes que nous avons présentés ci-dessus ne sont pas particulièrement novateurs en ce qui concerne la gestion du parallélisme ; par contre, le fait de les appliquer à la gestion des conflits de rôles dans les SMA organisationnels est à notre connaissance une nouveauté.

Nous pensons avoir montré l'importance de la réflexion sur la gestion des conflits de rôles dans une approche organisationnelle des SMA. En effet, pour bien profiter de la puissance d'une telle approche, il faut parvenir à un niveau d'abstraction suffisant, qui ne peut être atteint qu'avec une gestion automatique des conflits.

Notre approche constitue donc un premier pas dans cette direction. Il est sans doute possible de développer des mécanismes bien plus fins pour traiter ce problème, mais leur étude complète serait pour le moins le sujet d'une nouvelle thèse !

Nous avons maintenant présenté les mécanismes d'interaction entre composants dans un agent MOCA ; le prochain chapitre montrera comment ce fonctionnement est utilisé pour assurer la dynamique organisationnelle.

Chapitre 7

Dynamique organisationnelle

Dans ce chapitre, nous présentons les principes qui permettent aux agents dans MOCA de créer des groupes instanciant des organisations, d'y entrer et d'en sortir. Nous rappelons que MOCA fournit les primitives nécessaires à la dynamique des groupes, mais que les motivations qui poussent un agent à utiliser ces primitives dépendent du *type d'agent* (cf. page 60) et ne sont donc pas abordées dans cette thèse.

7.1 L'Organisation de Gestion

Toute plate-forme doit fournir un certain nombre de services qui gèrent le fonctionnement "méta" du système ; pour une plate-forme multi-agents, il s'agit généralement de la création ou de la destruction d'agents, de la gestion des messages, éventuellement de la gestion de la migration, etc.

Une particularité de MadKit est l'idée d'*agentifier* ces services, c'est à dire de faire qu'ils soient fournis par des agents [FG98]. Cette sorte de repliement du niveau "méta" sur le niveau multi-agent fournit à la fois une grande homogénéité conceptuelle et une souplesse considérable au niveau de la plate-forme. Cela permet en effet de remplacer un agent de service par un autre, proposant une implémentation différente du même service, et ceci éventuellement même en cours d'exécution.

Dans une plate-forme organisationnelle, les services à fournir concernent la gestion des organisations, groupes et rôles : créer un groupe, y prendre un rôle, le quitter, etc. Pour profiter des avantages évoqués ci-dessus, nous avons décidé d'effectuer un repliement similaire à celui de MadKit et de fournir les services organisationnels par une organisation particulière.

Il y aura donc dans notre système une *Organisation de Gestion*, instanciée en un *Groupe de Gestion*, chargé de gérer la dynamique organisationnelle. En y prenant un rôle, les agents acquièrent des compétences qui leur permettent de gérer leurs groupes. Comme dans le cas de MadKit, cette solution offre une grande souplesse : pour changer la gestion de la dynamique des groupes dans MOCA, il suffit d'introduire une nouvelle *Organisation de Gestion*, sans apporter le moindre changement aux agents ou aux autres organisations. Il est concevable d'effectuer ce changement en cours d'exécution et on peut même imaginer avoir plusieurs *Organisations de Gestion* qui proposent différents types de gestion simultanément dans un même système.

Cette manière de faire comporte également un autre avantage : en séparant clairement ce qui concerne la dynamique organisationnelle de ce qui concerne les fonctionnalités propres

d'une organisation particulière, on simplifie grandement la conception de nouvelles organisations. En effet, étant déchargé de leur gestion "méta", le concepteur peut se concentrer sur le contenu réel des organisations qu'il développe.

L'*Organisation de Gestion* que nous proposons est représentée à la figure 7.1 ; elle comporte trois descriptions de rôles que nous décrivons succinctement[1] :

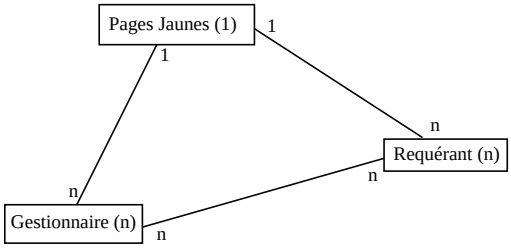


FIG. 7.1 – L'organisation de Gestion

Les Pages Jaunes sont responsables de tenir à jour une liste des organisations et des groupes existant dans le système et de fournir ces informations aux requérants. Toute création de groupe passe également par ce rôle, ce qui lui permet si nécessaire d'exercer un contrôle sur la structure globale du système (par exemple sur le nombre de groupes instanciant une organisation donnée).

Le Gestionnaire est responsable d'un groupe dans le système ; c'est à lui qu'un requérant s'adresse lorsqu'il veut entrer ou sortir de ce groupe. Ceci lui permet d'exercer un contrôle sur la structure du groupe. Dans notre implémentation (cf. chapitre 9), le gestionnaire se contente de vérifier que la cardinalité des rôles n'est pas violée et que l'agent requérant un rôle possède les compétences nécessaires à son exécution, mais de nombreuses autres politiques sont envisageables (cf. [FG98]).

Le Requérant peut interroger les Pages Jaunes sur les groupes et les organisations existants et s'adresser à un Gestionnaire pour entrer ou sortir d'un groupe géré par ce dernier.

Nous allons maintenant décrire plus précisément comment cette organisation, instanciée sous la forme d'un *Groupe de Gestion*, permet de gérer la dynamique organisationnelle d'un système MOCA.

7.2 Dynamique des groupes

La figure 7.2 décrit en quatre étapes le fonctionnement du *Groupe de Gestion*. Il s'agit d'un scénario typique de création et gestion d'un groupe dans MOCA. Nous allons donc suivre ces étapes en détail. Cependant, nous nous limiterons volontairement à une présentation assez informelle des processus impliqués. En effet, le fait de fournir les services de notre plate-forme au moyen d'un groupe fait qu'il est très facile d'adapter la gestion organisationnelle aux

[1] On pourrait se demander pourquoi nous ne présentons pas le comportement de ces descriptions de rôles sous forme de statecharts. La raison en est simple : la nature particulière de ces rôles fait que leur fonction principale est de fournir des compétences plutôt que de gérer un protocole. La représentation sous forme de statechart n'est donc pas particulièrement informative dans ce cas précis.

circonstances particulières d'une application. Nous désirons donc ici présenter les principes généraux de fonctionnement sans aborder les détails techniques. Nous présenterons par contre au chapitre 9 les détails d'une implémentation possible de l'*Organisation de Gestion*.

- La figure 7.2(a) présente l'état du système après son initialisation. Il existe seulement un groupe, le *Groupe de Gestion* ; un agent prend en charge le rôle de *Pages Jaunes* et tous les autres agents sont des *Requérants*. Ceux-ci peuvent alors entamer une conversation avec les *Pages Jaunes* pour savoir quelles organisations sont disponibles. Dans le cas de cette figure, l'*Agent 2* a trouvé une organisation Γ qui l'intéresse et demande sa création au rôle *Pages Jaunes* de l'*Agent 1*.
- Un agent demandant la création d'un groupe prend automatiquement un rôle de *Gestionnaire* dans le *Groupe de Gestion*. En général, il garde également le rôle de *Requérant* qu'il avait déjà. En (b), l'*Agent 2* a donc deux rôles dans le *Groupe de Gestion*. Il existe également un nouveau groupe G, mais celui-ci ne contient pour l'instant aucun agent. Le rôle de *Requérant* de l'*Agent 2* peut maintenant entamer une conversation avec le rôle de *Gestionnaire* du même agent pour négocier la prise du rôle *r1* dans le groupe G.
- En (c), le *Gestionnaire* a accepté l'entrée de l'*Agent 2* dans le groupe G avec le rôle *r1*. C'est au tour de l'*Agent 3* d'entrer en jeu : son rôle de *Requérant* demande à son tour au rôle *Pages Jaunes* de l'*Agent 1* quelles sont les organisations disponibles. Comme l'*Agent 3* s'intéresse également à l'organisation Γ, il demande si un groupe l'instanciant existe déjà. *Pages Jaunes* lui répond que oui et lui donne une accointance avec L'*Agent 2*. Le rôle de *Requérant* de l'*Agent 3* peut donc s'adresser au rôle de *Gestionnaire* de l'*Agent 2* pour négocier son entrée dans le groupe G.
- En (d), l'*Agent 3* a désormais un rôle *r2* dans le groupe G et peut donc converser dans ce groupe avec l'*Agent 2*, via le rôle *r1*. Si dans cette situation le rôle *r2* désire un changement organisationnel pour son Agent (prise d'un nouveau rôle, etc.), il ne peut pas s'adresser directement au rôle de *Gestionnaire* de l'*Agent 2* ou au rôle *Pages Jaunes* de l'*Agent 1*, car ceux-ci ne sont pas dans le même groupe. Il peut par contre utiliser les compétences fournies par le rôle *Requérant* de son Agent pour demander ces changements. Ceci est représenté par la flèche discontinue entre R et *r2*.

Cette dernière figure est très typique d'un système MOCA en cours d'exécution ; nous la reproduisons ci-dessous dans une version légèrement modifiée (figure 7.3) pour bien mettre en évidence ses éléments importants : les agents prennent un ou plusieurs rôles dans un ou plusieurs groupes ; les communications "horizontales" (intra-groupe) se font par échanges d'influences, alors que les communications "verticales" (inter-groupes) se font par appels de compétence. Notons encore que les échanges d'influences entre rôles se font de la même manière entre deux rôles d'un même agent ou de deux agents différent.

7.3 Conclusion

Le fonctionnement de principe présenté ci-dessus permet de réunir tous les services organisationnels dans une seule organisation ; de plus, grâce aux mécanismes présentés dans le chapitre précédent, la gestion organisationnelle peut se faire sans échange d'influences hors des groupes.

Nous sommes maintenant au bout de la présentation des concepts et principes de fonctionnement de MOCA, qui s'étendait du chapitre 4 jusqu'ici. Le prochain chapitre prendra un peu

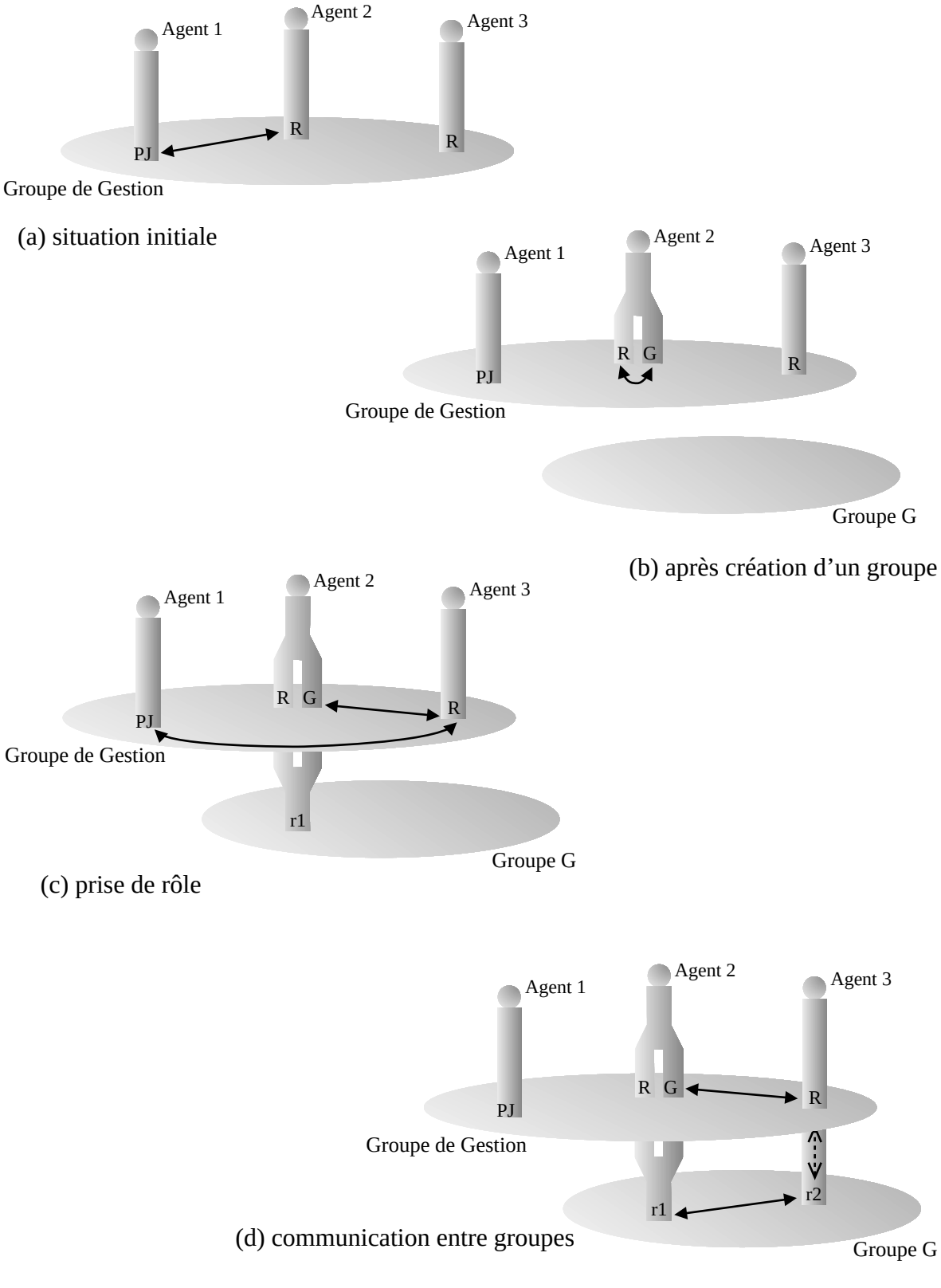


FIG. 7.2 – La dynamique organisationnelle

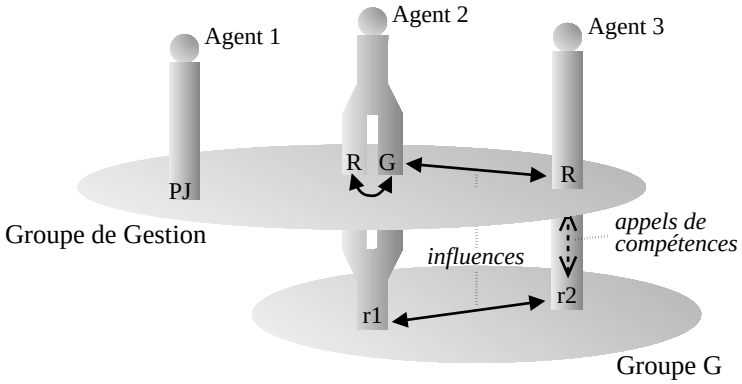


FIG. 7.3 – Un système MOCA typique

de recul en proposant quelques éléments de validation pour les approches organisationnelles telles que la nôtre.

Chapitre 8

Éléments de validation

8.1 Introduction

La *validation* d'une spécification consiste à prouver, par des moyens formels, qu'un système implémentant cette spécification possède certaines propriétés[1]. Celles-ci peuvent être générales, telle l'absence de blocage, ou plus spécifiques au système étudié, comme par exemple le fait que tout agent finira par recevoir un message donné.

La validation est une étape importante dans le cycle de développement d'un logiciel, et plusieurs techniques ont été mises au point pour l'aborder dans les divers contextes méthodologiques et formels existants [FH01]. Cependant, le paradigme multi-agent représente un défi particulier pour la validation, de par l'évolutivité et l'ouverture revendiquées pour les SMA.

Pour mieux situer le problème, nous commençons par un bref survol de la littérature. Le sujet étant vaste et encore largement ouvert, il mériterait bien entendu une présentation plus détaillée que celle que nous proposons ci-dessous. Cependant, la validation constituant plus une ouverture que le coeur de notre travail, nous nous limiterons à une présentation très succincte des différentes approches.

Survol de la littérature

Burkhard s'intéresse dans [Bur93] à des propriétés classiques (absence d'interblocage, vivacité, équité) qu'il exprime aussi bien pour les agents individuellement que pour le SMA dans son entier. Il montre ensuite que les propriétés globales du système ne découlent pas des propriétés locales des agents. Par exemple, un système formé d'agents équitables ne sera pas forcément un système équitable. L'article de Burkhard représente donc essentiellement un résultat négatif ; notons cependant qu'il finit sur une ouverture :

> « [...] there may be special conditions and laws of interaction and cooperation which permit the composition of special agents in order to obtain special system properties or which allow a separate analysis, respectively. »[2]

[1] Il semble y avoir un léger flou quant à l'acception de cette notion : si certains textes [Yoo99, Hug01] appellent *validation*, ou *validation formelle*, la preuve de propriétés, d'autres [Hil00] appellent cela *vérification* et réservent le mot *validation* à l'étude de la conformité entre spécification informelle et formelle.

[2] "Il pourrait exister des conditions particulières et des lois d'interaction ou de coopération permettant soit la composition de certains agents pour obtenir certaines propriétés du système, soit une analyse séparée [des propriétés des agents]."

Ceci pose donc un cadre clair sur les possibilités de validation : soit la démarche doit s'attaquer au SMA dans son entier, soit elle doit explorer les "conditions particulières" permettant une séparation de la validation. La figure 8.1 propose dans cette optique une vision synthétique des travaux que nous allons aborder dans ce paragraphe. Nous allons maintenant décrire rapidement chacun d'eux dans l'ordre des feuilles de l'arbre.

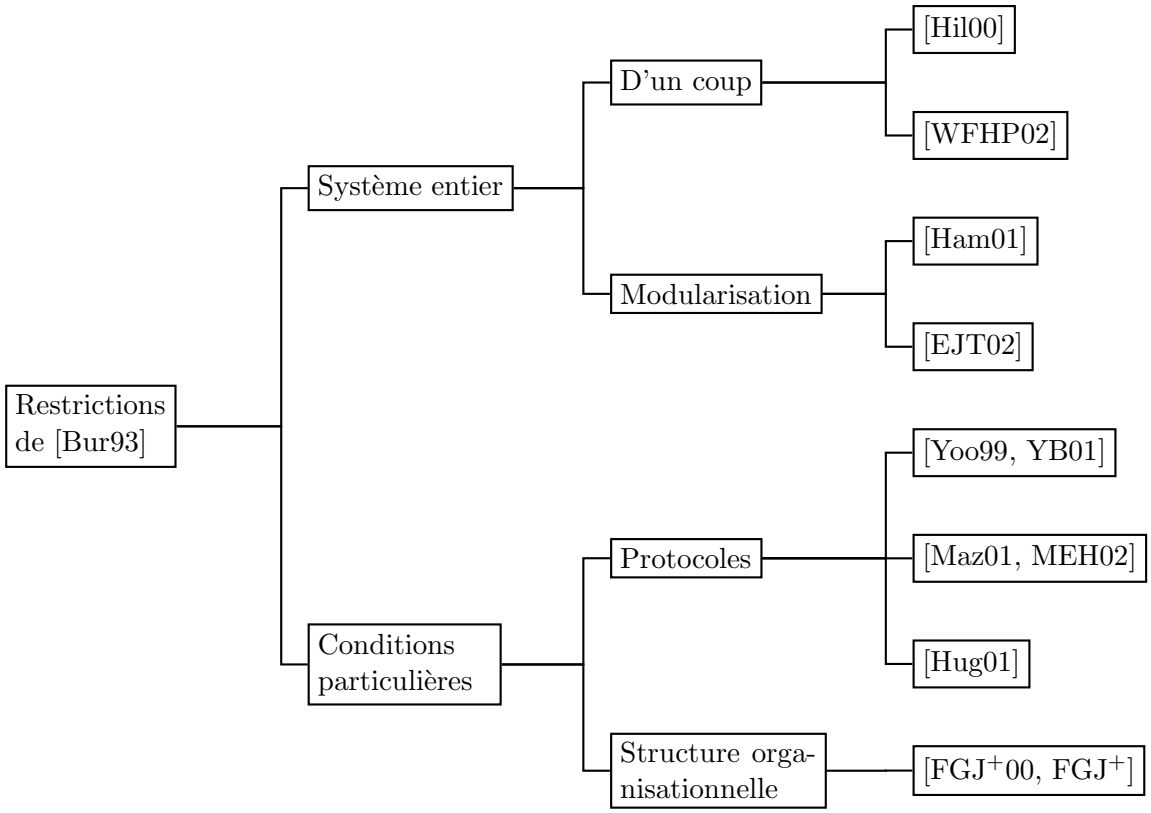


FIG. 8.1 – Vue synthétique des approches de validation des SMA

- [Hil00] propose une spécification du SMA dans le formalisme hybride Object-Z/Statechart que nous avons exposé au chapitre 3. Cette spécification est ensuite transformée (à la main) en un système de transition. On peut alors utiliser un outil de *model checking* (en l'occurrence STeP) pour vérifier des propriétés exprimées en logique temporelle.
- [WFHP02] propose quant à lui le langage impératif MABLE pour la spécification d'un système formé d'agents BDI. Un compilateur permet ensuite de générer du code utilisable par le *model checker* SPIN. Les propriétés sont exprimées en *MORA*, une *logique linéaire temporelle BDI*.
- [Ham01] propose une approche originale basée sur la substituabilité comportementale de composants. Après avoir défini une notion de composant basée sur BRIC (cf. chapitre 2), il définit deux relations, les *substitutions faible* et *forte*, capturant l'idée qu'un composant peut être remplacé par un autre en préservant tout ou partie de son comportement. Il étudie ensuite sous quelles conditions ces relations conservent des propriétés telles que la vivacité, l'absence de blocage, etc. Ces résultats ouvrent des portes intéressantes vers la préservation de propriétés lors du développement incrémental d'un

SMA.

- [EJT02] se base également sur une approche componentielle pour gérer la complexité du processus de vérification d'un SMA. L'idée est d'effectuer une décomposition récursive des propriétés à vérifier : on exprime les propriétés du système dans son entier en termes des propriétés externes des composants du niveau directement inférieur, puis on reprend ces dernières pour les propager au niveau inférieur, etc. Ceci permet de réduire la complexité de la vérification à faire en fin de processus (sur les composants atomiques).
- [Hug01] cherche à développer une approche complète de l'ingénierie des protocoles. Il propose pour cela un langage de type automate nommé CDPL. Il s'intéresse ensuite à la génération du graphe d'accessibilité et à la traduction d'un protocole CDPL en PROMELA et en réseau de Petri, chacune de ces traductions permettant de valider certaines propriétés.
- [Yoo99, YB01] suivent une démarche similaire en proposant le formalisme SCD (cf. chapitre 2), lui aussi traduisible en réseaux de Petri. Ceci permet donc de vérifier par *model-checking* différentes propriétés de sûreté, de vivacité, de terminaison, etc. du protocole. Les auteurs relèvent cependant que la taille de l'espace des états pose de sérieux problèmes pour cette démarche.
- [Maz01, MEH02] partent d'une expression des protocoles en AMUL pour en proposer une traduction en réseaux de Petri colorés récursifs [EH96]. La coloration permet de représenter des protocoles à nombre d'acteurs variable ; quant à la récursivité, elle permet à la fois de réduire la complexité de modélisation et de personnaliser dans chaque agent certaines parties du protocole, d'une manière ressemblant à la notion de compétence de MOCA.
- [FGJ$^+$00, FGJ$^+$] se basent sur le modèle AGR (cf. chapitre 2) pour structurer le processus de validation d'un cahier des charges (*requirements*). L'idée est d'exprimer les propriétés désirées à divers niveaux du système (sur le comportement des agents et des rôles, les interactions intra- et inter-groupes, le SMA dans son ensemble, etc.). Des modèles de preuves (*proof patterns*) sont ensuite proposés pour permettre de déduire les propriétés des niveaux globaux sur la base des niveaux locaux.

Approche

Le rapide survol de la littérature que nous venons d'effectuer nous permet de classer les approches de validation des SMA en deux catégories :

1. D'un côté, les approches visant à terme à vérifier le SMA complet. Cela englobe toutes les branches de l'arbre de la figure 8.1 à l'exception de la branche "Protocoles". Malgré la diversité des approche proposées, toutes nécessitent de fixer la structure du SMA avant de le vérifier. Il est donc impossible de traiter l'apparition ou la disparition d'agents en cours d'exécution, ni d'ailleurs la dynamique organisationnelle pour [FGJ$^+$00, FGJ$^+$]. On notera de plus que les approches visant à une validation globale du SMA par *model checking* se heurtent vite à des problèmes d'explosion combinatoire de l'espace d'états.
2. De l'autre côté, les approches basées sur les protocoles permettent de considérer un nombre de participants variable, et même éventuellement des variantes locales de comportement pour [Maz01, MEH02]. Cependant, il est évident que la validation d'un SMA ne saurait se limiter à celle de ses protocoles de coopération.

Ce que nous nous proposons de faire est donc de compléter le deuxième type d'approche en montrant comment différentes validations locales peuvent être combinées pour fournir une validation globale du système.

Pour être plus précis sur notre but, il faut remarquer que dans la démarche de validation, la représentation d'un protocole est souvent sous la forme d'un objet monolithique (un réseau de Petri, par exemple). Une des difficultés fondamentales est alors de savoir ce qui se passe quand celui-ci va être "éclaté" pour être réparti dans les différents agents. Nous allons donc étudier ci-dessous à quelles conditions cette répartition préserve les propriétés du protocole de départ.

Ceci constitue une nette avancée sur le chemin de la validation d'un SMA : en effet, les conditions de préservation de la validité d'un protocole lors de sa répartition peuvent être vues comme des "conditions au bord" sur le comportement des agents. Si tous les agents ont un comportement localement valide et respectent ces "conditions au bord", on peut alors en déduire des résultats globaux sur le système. Cette démarche permet donc d'aborder une validation modulaire d'un SMA à composition variable, tout en réduisant les problèmes liés à l'explosion de l'espace d'états.

Notre contribution se limitera ici à la première étape, à savoir l'expression des conditions de préservation de la validité.

8.2 Protocoles et organisations

Nous allons commencer par établir un lien entre les protocoles de communication et les notions organisationnelles de rôles et d'organisations. Pour ce faire, nous choisirons une approche résolument externe : seuls nous intéressent les échanges entre entités communicantes, indépendamment des mécanismes qui produisent ces échanges. Nous allons donc considérer essentiellement des séquences de messages[3], ce qui nous amène tout naturellement au choix d'un formalisme en termes de *langage*. En effet, il s'agit du formalisme naturel lorsqu'il s'agit de préciser quelle séquence de messages sera considérée comme valide et quelle autre ne le sera pas[4]. Notons que le langage sera défini sur la composante illocutoire des échanges et non sur le contenu, renforçant ainsi le point de vue externe que nous avons choisi.

Cette position sera également soulignée par une terminologie correspondante : nous définirons ci-dessous les notions de *nom de rôles*, *interface de rôle* et *d'organisation*, etc. qui mettent bien en évidence le fait que nous ne nous intéressons dans ce chapitre qu'aux structures des interactions, sans tenir compte des représentations internes des notions organisationnelles.

Protocoles

Pour cette discussion, nous nous plaçons dans la situation où les messages sont adressés et où cet adressage se déduit sans ambiguïté du message envoyé (par exemple en indexant les messages).

[3]Pour permettre une compréhension plus intuitive des développements de ce chapitre, nous avons opté pour une terminologie en termes de *messages* plutôt que d'*influences*. On pourra cependant généraliser les résultats de ce chapitre en remplaçant *message* par *influence* (et *composante illocutoire* par *type d'influence*) aussi longtemps que les échanges entre agents restent discrets.

[4]C'est même précisément la définition de la notion de *langage*, un langage étant un sous-ensemble de l'ensemble de toutes les séquences possibles d'un vocabulaire donné. L'étude des protocoles de communication en termes de langages, bien qu'inhabituelle, n'est pas une idée tout à fait nouvelle [Haa86, Cha90, And93, Bur93].

Définition 1. Soient NR et M des ensembles, et $em, rec : M \to NR$ deux fonctions. M est appelé *ensemble de messages adressés* par NR. Les éléments de NR sont appelés *noms de rôles*. Les fonctions em et rec sont appelées *émetteur* et *récepteur* respectivement.

On notera M_{NR} pour signifier que l'ensemble M de messages est adressé par NR. Lorsqu'on veut spécifier explicitement l'émetteur et le récepteur d'un message $m \in M_{NR}$, on écrit $message^{em(message)}_{rec(message)}$.

> **Remarque.** On notera que cette définition est équivalente à celle d'un graphe dont les noeuds seraient les noms de rôles, les arcs (orientés) les messages et les deux fonctions indiqueraient la source et le but des messages. Au fait, en rajoutant des cardinalités sur les noeuds et les arrêtes et en identifiant les arrêtes de même source et même but, on obtient une organisation au sens de MOCA.

Sur cette base, la définition la plus naturelle d'un protocole semble être la suivante :

Définition 2. Soit M_{NR} un ensemble de messages adressés. Un *protocole séquentiel* sur M est un langage de vocabulaire M, c'est à dire un sous-ensemble de l'ensemble M^* des concaténations d'éléments de M.

Cependant, cette définition est trop restrictive pour fournir une bonne représentation d'un protocole : il lui manque la possibilité d'exprimer que plusieurs messages peuvent être envoyés simultanément. Ceci nous donne donc la définition suivante :

Définition 3. Soit M_{NR} un ensemble de messages adressés. Un *protocole* sur M est un langage de vocabulaire $\wp(M)$, où $\wp(M)$ est l'ensemble des parties de M.

On remarquera que, via l'inclusion canonique de M dans $\wp(M)$[5], on peut voir un protocole séquentiel comme un protocole.

Exemple Pour illustrer ces définitions, nous reprendrons le protocole FIPA-ContractNet-Protocol [FIP], illustré à la figure 5.2 de la page 58.

Dans ce cas, l'ensemble des noms de rôles comporte deux éléments

$$NR = \{\text{initiator}, \text{participant}\},$$

l'ensemble des messages est celui des noms qui apparaissent sur les flèches horizontales de la figure

$$M_{NR} = \{\text{cfp}^{\text{initiator}}_{\text{participant}}, \text{refuse}^{\text{participant}}_{\text{initiator}}, ...\}$$

et le protocole est le langage engendré par la grammaire régulière suivante :

$$\begin{array}{llll} \text{cfp} & (\text{refuse}| & & \\ & \text{not}-\text{understood}| & & \\ & (\text{propose} & (\text{reject}-\text{proposal}| & \\ & & (\text{accept}-\text{proposal} & (\text{failure}| \\ & & & \text{inform}-\text{done}| \\ & & & \text{inform}-\text{ref}))))) \end{array}$$

[5] A savoir l'application $i : M \to \wp(M)$ qui envoie chaque $m \in M$ sur l'ensemble $i(M) = \{m\} \in \wp(M)$.

Localisation de protocoles

Dans un protocole impliquant plusieurs noms de rôles, il est probable que tous les rôles ne soient pas concernés par tous les messages. On peut donc s'intéresser à la restriction du langage aux seuls messages concernant un nom de rôle donné :

Définition 4. Soit P un protocole sur l'ensemble M_{NR} de messages adressés. Pour un nom de rôle $r \in NR$, on note $P{\downarrow_r}$ le sous-langage de P obtenu en ne retenant que les messages dont l'émetteur ou le récepteur est r. On appelle $P{\downarrow_r}$ la *projection* de P sur r.

Pour simplifier la notation, on écrira la composition de projections $(P{\downarrow_{r_1}}){\downarrow_{r_2}}$ sous la forme $P{\downarrow_{r_1 r_2}}$.

> **Remarque.** Si on veut définir plus formellement cette notion de projection, il faut suivre le chemin suivant : pour $r \in NR$, on définit la fonction $\bullet{\downarrow_r} : \wp(M) \to \wp(M)$ qui envoie $A \subset M$ sur $\{m \in A | em(m) = r \vee rec(m) = r\}$. On définit ensuite $\bullet{\downarrow_r} : \wp(M)^* \to \wp(M)^*$par induction, en appliquant la fonction ci-dessus successivement sur chaque élément d'une concaténation. Enfin, on définit la projection d'un protocole comme la restriction de cette dernière fonction au domaine concerné.

Exemple Prenons un protocole à trois noms de rôles que nous appellerons *client*, *fournisseur* et *intermédiaire* : $NR = \{c, f, i\}$. Le client peut passer une commande à l'intermédiaire et celui-ci la transmettre au fournisseur. Ce dernier peut livrer l'intermédiaire qui peut livrer le client. Enfin, le client peut payer directement le fournisseur : $M_{NR} = \{\mathrm{com}_i^c, \mathrm{com}_f^i, \mathrm{liv}_i^f, \mathrm{liv}_c^i, \mathrm{pay}_f^c\}$ et[6]

$$P = \{\mathrm{com}_i^c \mathrm{com}_f^i \mathrm{liv}_i^f \mathrm{liv}_c^i \mathrm{pay}_f^c\}.$$

Pour clarifier le déroulement du protocole, la figure 8.2 en donne une représentation graphique.

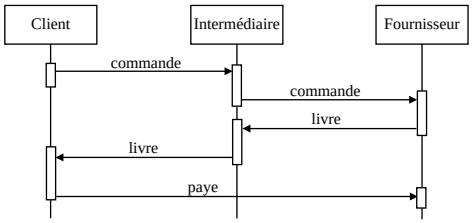


FIG. 8.2 – Un exemple de protocole

On a alors les projections $P{\downarrow_c} = \{\mathrm{com}_i^c \mathrm{liv}_c^i \mathrm{pay}_f^c\}$, $P{\downarrow_i} = \{\mathrm{com}_i^c \mathrm{com}_f^i \mathrm{liv}_i^f \mathrm{liv}_c^i\}$ et $P{\downarrow_f} = \{\mathrm{com}_f^i \mathrm{liv}_i^f \mathrm{pay}_f^c\}$.

[6] On notera la différence de notation entre M_{NR} qui est un ensemble de 5 éléments et P qui est un ensemble formé d'un seul élément, ce dernier étant la concaténation des 5 éléments de M_{NR}. Il s'agit donc d'un protocole séquentiel ne possédant qu'une exécution possible.

Lemme 6. *Soient $r_1, r_2 \in NR$. Alors la projection sur r_1 est idempotente*[7] *et commute avec la projection sur r_2. De plus, la projection sur plus de deux noms de rôles distincts donne toujours le langage vide.*

Démonstration. L'idempotence et la commutation découlent directement des définitions (la fonction $\bullet\downarrow_r : \wp(M) \to \wp(M)$ définie dans la remarque ci-dessus possède ces propriétés qui sont conservées par les constructions subséquentes).

Pour la troisième propriété, il suffit de constater que chaque message concerne au plus deux noms de rôles et que chaque projection ne garde que les messages concernant un agent donné. □

Interfaces de rôles et d'organisation

Nous pouvons maintenant commencer à établir des liens entre les protocoles et les notions organisationnelles.

Définition 5. Un protocole I sur un ensemble M de messages adressés par NR est appelé *interface du rôle de nom r* ($r \in NR$) si $I\downarrow_r = I$. En d'autres termes, l'interface d'un rôle est un protocole ne contenant que des messages concernant ledit rôle (i.e. dont il est l'émetteur ou le récepteur).

L'idempotence de la projection permet alors de poser la définition suivante :

Définition 6. Soit P un protocole sur un ensemble M de messages adressés par NR. Pour chaque $r \in R$, on pose $\mathit{Interface}(r) = P\downarrow_r$. On nomme ces protocoles *Interfaces associées au protocole P*.

On sait donc passer d'un protocole à son ensemble d'interfaces. Le passage inverse est plus délicat et n'est pas toujours possible.

Définition 7. Soit $O = \{R_i\}_{i \in NR}$ un ensemble d'interfaces de rôles sur le même ensemble de messages M adressés par NR. Les interfaces de rôles de O sont dites *interopérables* s'il existe un protocole P tel que les R_i sont les interfaces associées à P.

Il est facile de trouver un critère nécessaire pour assurer cette interopérabilité :

Définition 8. Soit $O = \{R_i\}_{i \in NR}$ un ensemble d'interfaces de rôles sur le même ensemble de messages M adressés par NR. Les interfaces de rôles de O sont dit *interopérables deux à deux* si $R_i\downarrow_j = R_j\downarrow_i i$, $\forall i, j \in NR$.

Intuitivement, cela signifie que les rôles doivent s'accorder deux à deux sur leur manière de communiquer. Cependant, ce critère n'est malheureusement pas suffisant.

Théorème 2. *Soit $O = \{R_i\}_{i \in NR}$ un ensemble d'interfaces de rôles sur le même ensemble de messages M adressés par NR. Alors le fait que les R_i soient interopérables deux à deux est une condition nécessaire et non suffisante pour qu'ils soient interopérables.*

[7] i.e. $P\downarrow_{r_1 r_1} = P\downarrow_{r_1}$. Incidemment, cela justifie le nom de "projection", qui est habituellement réservé aux opérateurs idempotents.

Démonstration. La nécessité découle directement de la commutativité des projections : supposons qu'il existe un protocole P tel que $P{\downarrow_i} = R_i$. Alors $R_i{\downarrow_j} = P{\downarrow_{ij}} = P{\downarrow_{ji}} = R_j{\downarrow_i}$.

Pour montrer que ce n'est pas suffisant, nous construisons un contre-exemple en reprenant les interfaces de rôles de la figure 8.2, mais en inversant les messages *liv* et *pay* dans le rôle du fournisseur : $R_c = \{\mathrm{com}_i^c \mathrm{liv}_c^i \mathrm{pay}_f^c\}$, $R_i = \{\mathrm{com}_i^c \mathrm{com}_f^i \mathrm{liv}_i^f \mathrm{liv}_c^i\}$ et $R_f = \{\mathrm{com}_f^i \mathrm{pay}_f^c \mathrm{liv}_i^f\}$ (En clair, ceci veut dire que le fournisseur attend d'être payé pour livrer et que le client attend d'être livré pour payer). Alors ces interfaces de rôles sont interopérables deux à deux ; mais tout protocole se projetant sur c et i doit avoir pay_f^c après liv_c^i, alors que tout protocole se projetant sur i et f doit avoir pay_f^c avant liv_c^i. Un protocole se projetant à la fois sur les trois interfaces est donc impossible. □

Ce résultat montre que la possibilité de reconstruire un protocole à partir d'interfaces de rôles n'est pas observable de manière locale aux agents, ni en admettant des "collaborations" deux à deux. En règle générale, il faut donc réunir les informations disponibles à plus de deux agents simultanément pour décider si les interfaces de rôles sont interopérables.

Nous pouvons maintenant exprimer formellement le rapport que nous proposons entre organisations et protocoles :

Définition 9. Une *interface d'organisation* est un ensemble O d'interfaces de rôles interopérables.

Il découle directement des définitions ci-dessus que l'ensemble des interfaces de rôles associées à un protocole forme une interface d'organisation et que pour une interface d'organisation donnée O, il existe un protocole P tel que les interfaces de rôles associés à P soient exactement celles de O.

Nous avons donc exposé une manière de voir une (interface d')organisation comme l'expression décentralisée d'un protocole. Comme la vérification de ce point ne peut se faire localement par les agents, c'est généralement le concepteur qui la fera avant l'implantation. Ceci suppose donc un travail d'ingénierie de protocole avant la description des interfaces de rôles. Un tel travail comprend généralement la vérification de certaines propriétés du protocole. Nous allons maintenant voir quel est le type de propriétés auxquelles notre démarche peut s'intéresser.

8.3 Propriétés

Dans [Hug01], l'auteur recense 12 propriétés que l'on peut valider sur un protocole quel que soit son formalisme de représentation. Il s'agit de :

1. La progression : absence de blocage.
2. La solidité : le protocole n'atteint jamais un état inacceptable.
3. La vivacité : tous les états du protocole sont accessibles.
4. Le non dépassement de la capacité des canaux.
5. La terminaison ou réinitialisation : le protocole termine ou retourne dans son état initial dans un temps fini.
6. L'absence de cycle bloquant : un cycle bloquant est un cycle pour lequel il n'y a pas de progression et dans lequel les mêmes messages sont échangés indéfiniment.

7. L'exclusion mutuelle : deux utilisateurs du protocole ne tentent pas d'accéder simultanément à la même ressource critique.
8. La conformité partielle : le protocole est en mesure de réaliser un service donné en atteignant son état final.
9. La conformité totale : le protocole vérifie les propriétés de terminaison (point 5 ci-dessus) et de conformité partielle (point 8).
10. La complétude : le protocole est capable de répondre à tous les événements possibles y compris les réceptions non spécifiées.
11. La tolérance aux fautes : le protocole est capable de retourner dans un état stable au bout d'un temps fini après une erreur.
12. L'équité : un protocole cherche toujours à progresser quelles que soient les opérations réalisées par les autres entités concurrentes utilisant ce protocole.

Cependant, notre approche qui sépare clairement l'interface du protocole de son implémentation sous forme d'entités communiquantes pose des conditions assez fortes sur le type de propriétés pouvant être considérées à ce stade. Nous survolons brièvement les 12 points ci-dessus en les regroupant :

Propriétés internes Les propriétés 2, 3, 5, et 6 font intervenir l'état interne des entités communicantes ; nous ne pouvons donc pas les prendre en compte pour l'instant. Nous y reviendrons par contre au paragraphe 8.5.

Propriétés de ressources Les propriétés 4, 7 et 12, en faisant intervenir la notion de ressource, font indirectement référence à un état interne et rejoignent donc de ce point de vue la catégorie ci-dessus.

Propriétés de sortie du protocole Les propriétés 10 et 11 considèrent le comportement des entités communicantes lorsqu'elle sont confrontées à une situation non prévue dans le protocole. Elle font donc aussi référence implicitement à un état interne et rejoignent les deux premières catégories.

Propriétés de conformité Les propriétés 8 et 9 concernent l'adéquation "en amont" de la spécification du protocole (i.e. par rapport à une spécification précédente), alors que nous nous intéressons à assurer une adéquation "en aval". Nous choisissons donc de les ignorer.

Propriétés externes À première vue, la récolte est maigre en ce qui concerne les propriétés qui s'expriment de manière purement externe et qui sont par conséquent concernées par notre approche : dans la liste ci-dessus, seule la première remplit cette condition. On remarquera cependant qu'en reformulant la troisième sous la forme "tous les déroulements du protocole sont possibles", elle devient une propriété externe. De même, une reformulation externe possible de la cinquième serait "tout déroulement du protocole est fini"[8].

Nous nous intéresserons donc pour l'instant uniquement aux propriétés externes, qui sont les seules accessibles à notre formulation en termes de langages. En plus de celles que nous avons vues ci-dessus, on peut en imaginer des plus spécifiques, telles que "tout déroulement du protocole finit par un message x", "le message y apparaît au plus une fois dans tout déroulement du protocole", etc.

[8] Ces formulations de propriétés sont évidemment assez informelles ; pour des formulations externes beaucoup plus précises, on se référera à [Bur93] ou [Cha90].

8.4 Conditions de préservation

La question que nous nous posons maintenant est la suivante : "En supposant qu'un protocole possède une propriété externe donnée, peut-on assurer que celle-ci reste valable lorsqu'on considère les interfaces de rôles associées au protocole ?" ; ou en termes plus formels : "L'application qui associe à un protocole l'ensemble de ses rôles associés préserve-t-elle les propriétés ci-dessus ?".

Or nous verrons que cette question se ramène en fait à la suivante : "Soient P un protocole, $O = \{N_i\}_{i \in NR}$ l'interface d'organisation associée à P, et soient $\{R_i\}_{i \in NR}$ des automates tels que chaque R_i ait pour interface N_i — c'est-à-dire que la liste des séquences d'entrées-sorties pour toutes les exécutions possibles de R_i soit exactement N_i. Peut-on alors assurer que l'ensemble des R_i est capable de produire toutes les séquences de P et rien que celles-ci ?".

Dans le cas général, la réponse est non. En effet, deux sources de problèmes sont possibles :

Localité du contrôle Dans un protocole à trois protagonistes r, s et t défini par $P = \{a_s^r b_s^t c_t^r\}$, rien n'indique à t qu'il doit attendre que le message a ait été envoyé avant d'envoyer b (son interface est $b_s^t c_t^r$) et de même r ne peut pas savoir quand envoyer c. Donc certaines exécutions pourraient amener un des protagonistes à recevoir un message inattendu. Au fait, il est prouvé dans [CSS01] que le contrôle de ce protocole ne peut être local. On voit donc que pour assurer le bon déroulement d'un protocole réparti, il faut que son contrôle soit local (ce qui n'est pas vraiment une surprise).

Bifurcation cachée Considérons l'automate de la figure 8.3. Son interface est $\{abc, abd\}$. Supposons que a et c sont des messages entrants et que b est sortant. Si cet automate est confronté à un autre avec lequel il est interopérable (au sens ci-dessus), il risque tout de même d'y avoir un problème, car il pourrait recevoir un message auquel il ne s'attend pas.

Ceci vient du fait qu'un choix du déroulement est pris à la première transition sans que cela n'apparaisse d'un point de vue externe. Nous dirons d'un tel automate qu'il est à bifurcation cachée.

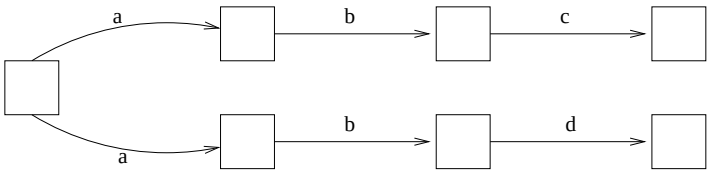


FIG. 8.3 – Un exemple d'automate à bifurcation cachée

Ces deux problèmes sont les seuls qui peuvent survenir :

Théorème 3. *Soient P un protocole, $O = \{N_i\}_{i \in NR}$ l'interface d'organisation associée et $\{R_i\}_{i \in NR}$ des automates tels que l'interface de R_i est N_i pour tout i. Supposons que P est à contrôle local et que les R_i sont sans bifurcation cachée. Alors les exécutions des $\{R_i\}$ sont exactement celles de P.*

Démonstration. Montrons d'abord que tout déroulement p de P est accessible : le fait que P soit à contrôle local nous assure que chaque R_i reçoit les messages de p au moment prévu

et l'absence de bifurcation cachée assure qu'il est en mesure de répondre selon p. Donc le déroulement p est possible.

Montrons maintenant qu'aucun déroulement non contenu dans P ne peut se produire. Supposons par l'absurde qu'un tel déroulement arrive. Donc il existe un moment où un R_i envoie un message non prévu dans P. Mais cela veut dire que son interface N_i est distincte de $P{\downarrow_{r_i}}$, ce qui est contraire aux hypothèses. □

Munis de ce résultat, nous pouvons maintenant revenir à notre question concernant les propriétés de progression et de vivacité :

Corollaire. *Sous les hypothèses du théorème précédent, la localisation d'un protocole préserve les propriétés externes.*

Démonstration. Les propriétés externes étant précisément celles qui s'expriment en fonction des exécutions du protocole, l'affirmation découle directement du théorème. □

Ainsi, l'absence de blocage, la vivacité et les propriétés similaires sont préservées par la localisation d'un protocole.

8.5 Discussion

Il convient maintenant de replacer les résultats ci-dessus dans leur contexte : les conditions de préservation ci-dessus sont exprimées d'un point de vue tout à fait externe. Bien sûr, si un agent participe à un protocole et qu'un blocage se produit à l'intérieur de cet agent, le protocole va aussi se retrouver bloqué. Le corollaire ci-dessus n'assure le non-blocage du protocole que dans le cas où l'agent est capable d'assurer son interface.

Dans ce contexte plus large, le théorème 3 est donc un théorème de *localisation de la validation* : il traduit des conditions de validité inter-agents en conditions de validité intra-agent. C'est à ce stade que l'on peut revenir sur les différentes propriétés que nous avons mises de côté au paragraphe 8.3 ; en réintroduisant l'agent dans la démarche, les notions d'état interne reprennent leur sens et avec elles les propriétés de solidité, d'exclusion mutuelle, etc.

Le problème de validation d'un SMA complet que nous nous étions posé au départ se retrouve donc éclaté en une multitude de problèmes, à savoir la validation de chacun des agents. Il pourrait sembler que cette multiplication constitue un pas en arrière. Cependant, on considérera que :

- la validation se fait généralement par *type d'agent* et non par *agent* ; or les types d'agents sont généralement en nombre considérablement plus réduit que les agents eux-même[9]
- en ramenant la validation au niveau de l'agent, nous la ramenons dans un contexte où l'application de techniques existantes est beaucoup plus facile.

Ainsi, nous pensons que les considérations ci-dessus constituent un pas en avant dans la direction d'une démarche de validation des SMA. Bien sûr, un travail considérable reste à faire : si quelques pistes ou propositions de solutions sont données dans le chapitre 6 quant à des propriétés telles que le non-blocage (interne !) ou l'exclusion mutuelle, tout le travail de vérification formelle reste à faire.

[9] On trouve rarement plus d'une dizaine de types d'agents dans un système donné ; souvent ce nombre est plutôt de l'ordre de quelques unités. Le nombre d'agents, par contre, peut être de l'ordre du millier, voire plus dans certaines applications.

Nous conclurons ce chapitre avec la remarque suivante : on pourrait s'étonner que nous effectuions une réduction, concernant la validation, du niveau du système à celui des agents, alors que l'idée de non-réductibilité du comportement d'un SMA à celui de ses agents est une idée forte du domaine. Nous répondrons en rappelant que cette démarche de validation s'inscrit dans une vision particulière des SMA — que nous avons exposée en détail tout au long de cette partie — et dont une des hypothèses centrales est que les communications n'ont lieu qu'à l'intérieur des groupes. Sans cette hypothèse, la totalité des développements de ce chapitre s'écroule.

La non réductibilité d'un SMA à ses agents est souvent exprimée de la manière suivante : les interactions sont aussi importantes que les agents eux-mêmes et il n'est pas possible de les négliger quand on considère un SMA. Or l'hypothèse que nous évoquons permet de contrôler les interactions aussi bien que les comportements individuels des agents, et donc de *prendre en compte les interactions dans le processus de réduction*. C'est donc la représentation explicite et structurée de toutes les interactions possibles du système sous forme d'un "niveau descriptif" qui nous permet d'esquisser cette démarche de validation.

Nous parvenons maintenant au terme de la deuxième partie de cette thèse, consacrée à la présentation conceptuelle du modèle MOCA. Nous espérons avoir convaincu le lecteur que l'approche organisationnelle que nous proposons apporte un bon compromis entre la souplesse caractéristique des approches multi-agents, l'aisance de conception et le contrôle formel de l'exécution. La prochaine partie décrira plus en détail l'implémentation du modèle que nous avons réalisée, ainsi que les expérimentations subséquentes.

Troisième partie

Implémentation et expérimentation

Chapitre 9

Implémentation de MOCA

9.1 Introduction

La troisième partie de cette thèse est consacrée aux réalisations pratiques basées sur le modèle présenté dans la deuxième partie. Nous décrirons donc dans ce chapitre la plate-forme implémentant le modèle MOCA et dans le suivant les tests que nous avons réalisés.

Le prochain paragraphe présentera l'implémentation de la plate-forme proprement dite alors que le suivant s'attachera à la description de l'*Organisation de Gestion* que nous avons implémentée.

9.2 La plate-forme MOCA

Le modèle MOCA a fait l'objet d'une première implémentation réalisée en 2001 par José Báez dans le cadre de sa quatrième année d'études en informatique à l'Université de Neuchâtel. Cette première version de la plate-forme était largement simplifiée par rapport au modèle actuel ; en particulier, les caractéristiques suivantes n'étaient pas encore présentes :

- compétences dynamiques.
- gestion des conflits de rôle
- gestion générique de la dynamique organisationnelle par le Groupe de Gestion

Malgré ces simplifications, cette première implémentation nous a permis de mettre à l'épreuve les concepts centraux de MOCA, et d'apporter quelques compléments ou modifications au modèle pour assurer son côté opérationnel. Notons que cette première version de la plate-forme a fait l'objet d'une démonstration aux JFIADSMA'01 [MABN01].

Les clarifications amenées au modèle par cette première étape ont donné jour à une deuxième version, implémentée en grande partie par José Báez dans le cadre de son travail de diplôme. Il s'agit d'une réimplémentation presque complète de la plate-forme, incluant les points cités ci-dessus. Nous présentons ici les principales caractéristiques de cette version :

- L'implémentation a été réalisée en Java.
- La plate-forme multi-agents sous-jacente est MadKit ; notons cependant que le code dépendant de cette plate-forme est localisé au sein d'un *package* bien défini, facilitant ainsi un éventuel portage vers une autre plate-forme. La grande majorité du code a ainsi été rendue indépendante de la plate-forme.
- Les composants sont réalisés par des objets et les compétences par des interfaces. Un composant fournit donc une compétence si l'objet correspondant implémente l'interface

correspondante.
- Le lien entre les niveaux de description et d'exécution, bien que correspondant conceptuellement à une instanciation, est réalisé sous forme d'agrégation ; ainsi par exemple, un *rôle* n'est pas, techniquement parlant, une instance de *description de rôle*, mais il possède une variable pointant vers sa description de rôle.
- La spécification d'un système MOCA se fait sous la forme d'un fichier XML. On en trouvera à l'annexe A la grammaire exacte (fichier DTD), ainsi qu'un exemple de spécification. Les parties en Object-Z sont remplacées par des références à des classes Java.
- Les échanges d'influences ne peuvent prendre place qu'au sein d'un groupe. De plus, un agent n'est apte à recevoir une influence d'un autre agent que s'ils sont déjà en accointance. La création d'une nouvelle accointance entre deux agents ne peut donc se faire que par l'intermédiaire d'un troisième, déjà en accointance avec chacun des deux autres (par exemple le gestionnaire du groupe). Ce contrôle strict des échanges entre agents vise à une amélioration de la sécurité et de la stabilité d'un SMA hétérogène et ouvert.

Il ne s'agit là bien évidemment que d'un survol de la plate-forme. On pourra trouver les principaux diagrammes de classes à l'annexe B. Pour toutes les informations sur les choix d'implémentation, on pourra se référer à [Báe02]. Enfin, la plate-forme MOCA elle-même peut être téléchargée sur la page internet de MadKit [Mad].

9.3 L'Organisation de Gestion

Avant de présenter notre implémentation de l'*Organisation de Gestion*, il faut relever que l'implémentation de MOCA a hérité une bonne partie de son vocabulaire de l'approche de Durand (cf. chapitre 2), ce qui provoque quelques différences de terminologie par rapport à ce qui précède. En particulier, ce que nous avons appelé *Organisation* et *Groupe* dans la deuxième partie de cette thèse est appelé respectivement *Schéma d'organisation* et *Organisation* dans la plate-forme MOCA. Nous respectons dans ce chapitre la terminologie de la plate-forme.

La structure de l'*Organisation de Gestion* que nous avons développée est représentée à la figure 9.1. Elle se compose des trois rôles *YellowPages*, *Manager* et *Requester* qui correspondent respectivement aux rôles *Pages Jaunes*, *Gestionnaire* et *Requérant* décrits au chapitre 7.

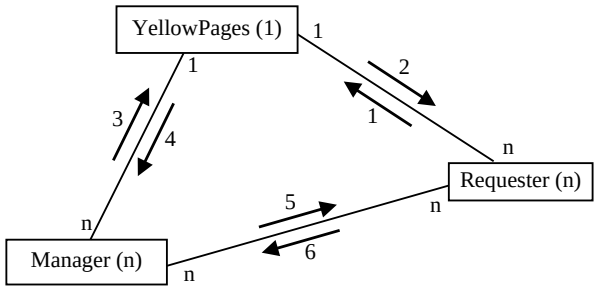


FIG. 9.1 – L'Organisation de Gestion

Les types d'influences pouvant transiter le long des accointances de la figure 9.1 sont les suivants :

1. AskAcquaintance, AskAgents, AskManager, AskRole, GetAllOrganisations, LeaveRole ;
2. AnswerAskAcquaintance, AnswerAskAgents, AnswerAskManager, AnswerAskRole, AnswerGetAllOrganisations, LeaveRole ;
3. AnswerAskAcquaintance, LeaveRole, RegisterOrganisation ;
4. AnswerRegisterOrganisation, AskAcquaintance, LeaveRole ;
5. AnswerAskAcquaintance, AnswerAskAgents, AnswerAskRole, AskAcquaintance, LeaveAcquaintance, LeaveRole ;
6. AnswerAskAcquaintance, AskAcquaintance, AskAgents, AskRole, LeaveAcquaintance, LeaveRole.

Nous donnons ci-dessous une brève description de chacun de ces types d'influences ; pour plus de détails, on pourra se reporter à [Báe02] :

- *AnswerAskAcquaintance* est la réponse à une demande d'accointance ;
- *AnswerAskAgents* est la réponse à une demande d'énumération des agents jouant un certain rôle dans une organisation donnée ;
- *AnswerAskManager* est la réponse à une demande pour obtenir la référence du *Manager* d'une organisation ;
- *AnswerAskRole* est la réponse à une demande pour obtenir un rôle dans une organisation ;
- *AnswerGetAllOrganisations* est la réponse à une demande d'énumération de l'ensemble des organisation du système ;
- *AnswerRegisterOrganisation* est la réponse à une demande d'enregistrement d'une organisation ;
- *AskAcquaintance* est une demande d'accointance directe ou indirecte ;
- *AskAgents* est une demande d'énumération des agents jouant un certain rôle dans une organisation donnée ;
- *AskManager* est une demande pour obtenir la référence du *Manager* d'une organisation ;
- *AskRole* est une demande pour obtenir un rôle dans une organisation ;
- *GetAllOrganisations* est une demande d'énumération de l'ensemble des organisation du système ;
- *LeaveAcquaintance* est une demande pour que le destinataire oublie une accointance ou l'indication que l'émetteur en oublie une ;
- *LeaveRole* est une demande pour que le destinataire libère un certain rôle ou l'indication que l'émetteur en libère un ;
- *RegisterOrganisation* est une demande d'enregistrement d'une organisation.

Le rôle *Manager* fournit une compétence lui permettant d'enregistrer l'organisation qu'il gère auprès des *YellowPages*. Un appel à cette compétence déclenche le protocole décrit à la figure 9.2.

Un agent désirant entrer dans une organisation devra dérouler le scénario de la figure 9.3. Celui-ci peut paraître complexe, mais on remarquera que le développeur ne doit s'occuper que de l'envoi de la première influence et de la réception de la dernière, tout le reste étant fourni en standard.

Avant de finir ce paragraphe, nous tenons encore à apporter une petite précision concernant le rôle de *Manager* : dans MadKit, tout groupe possède un gestionnaire qui fait partie

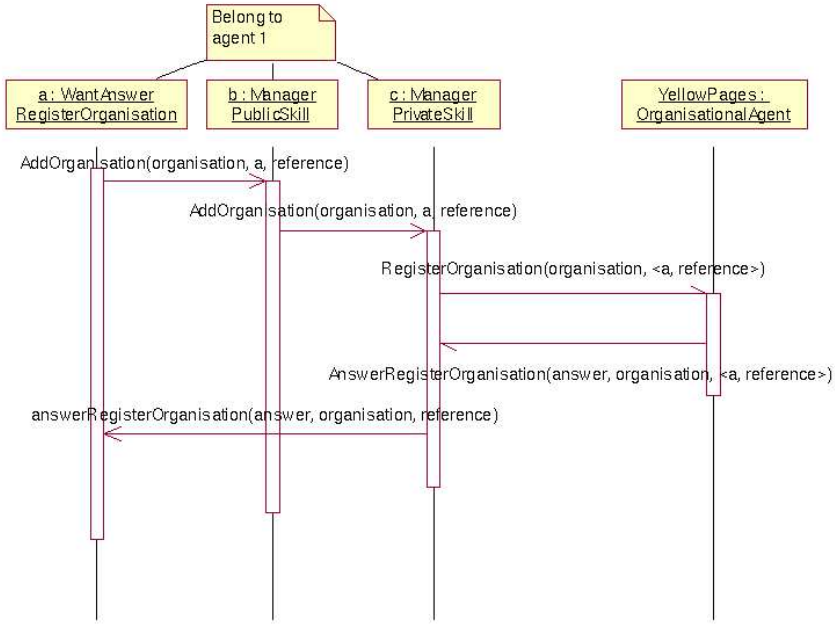


FIG. 9.2 – Protocole d'enregistrement d'une organisation

du groupe. Nous avons préféré dans notre approche déléguer toute la gestion des groupes et des organisations à un organe spécialisé ; ceci permet de grandement simplifier la conception de nouvelles organisations en séparant clairement le fonctionnement spécifique de la gestion générique du méta-niveau. Cependant, comme MOCA est basé sur MadKit, un agent créant un groupe recevra, en plus du rôle de *Manager* dans l'Organisation de Gestion, un rôle de gestionnaire dans le groupe nouvellement créé. Ce rôle ne sera alors qu'un rôle MadKit, c'est-à-dire vide de toute sémantique opérationnelle. Ce fonctionnement, parfaitement transparent pour l'utilisateur, n'a généralement pas à être pris en compte ; il existe toutefois une situation où cet état de fait gagne à être connu, à savoir lorsqu'on désire modifier le mécanisme générique de gestion dans une organisation particulière. Par exemple, on peut imaginer que le fait de quitter un groupe soit soumis à une condition, comme avoir fini son travail. Dans ce cas, il suffira d'inclure explicitement la description du rôle de *Manager* dans l'organisation concernée et c'est ce rôle-là qui sera utilisé pour la gestion de l'organisation plutôt que le rôle standard de l'Organisation de Gestion.

9.4 Conclusion

Dans ce chapitre, nous avons présenté succinctement les caractéristiques de la plate-forme MOCA et de l'*Organisation de Gestion* que nous avons implémentée. Pour plus de détails sur les choix d'implémentation, on pourra se référer à [Báe02].

Le prochain chapitre présentera les expérimentations que nous avons réalisées à l'aide de cette plate-forme.

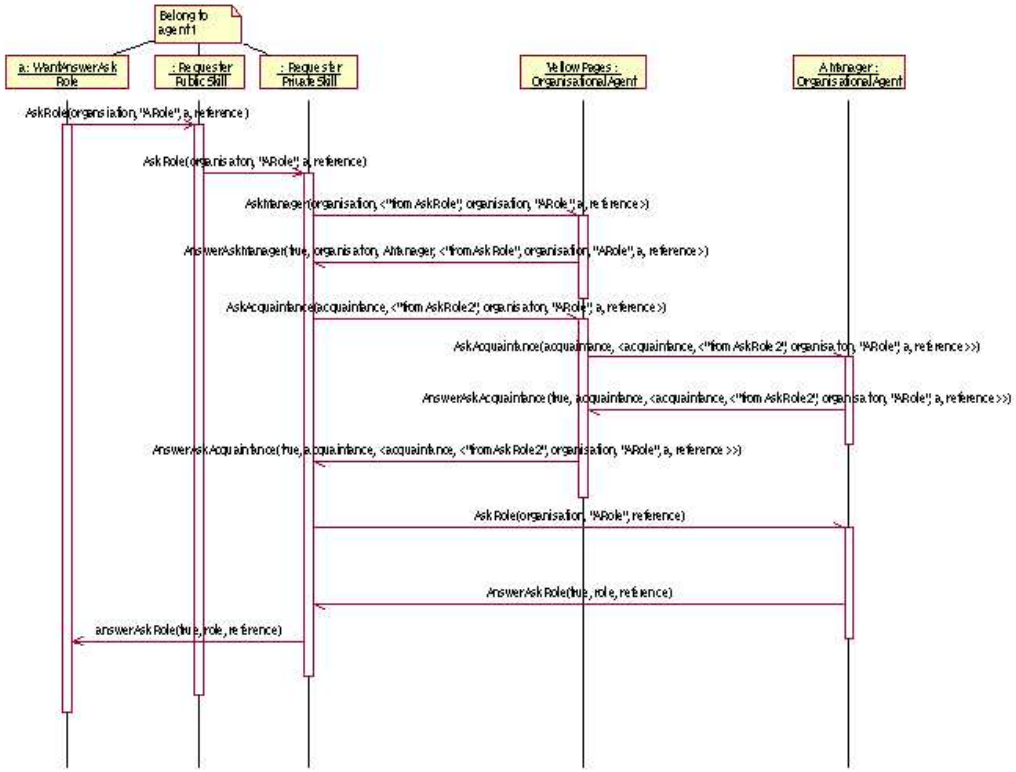


FIG. 9.3 – Protocole d'entrée dans une organisation

Chapitre 10

Expérimentation

Ce chapitre présente les expériences que nous avons réalisées pour tester la plate-forme MOCA. Celles-ci s'articulent selon deux directions : la première effectue des tests ciblés sur le mécanisme de gestion des conflits de rôles et sera présentée au paragraphe 10.1 ; la seconde vise à démontrer l'utilisabilité de la plate-forme et sera présentée au paragraphe 10.2.

10.1 Tests du mécanisme de gestion des conflits de rôles

La première série de tests que nous présentons a pour but d'évaluer le mécanisme de gestion de conflits de rôles que nous avons présenté au chapitre 6. Pour ce faire, nous avons développé un système MOCA dont la structure générale est représentée à la figure 10.1.

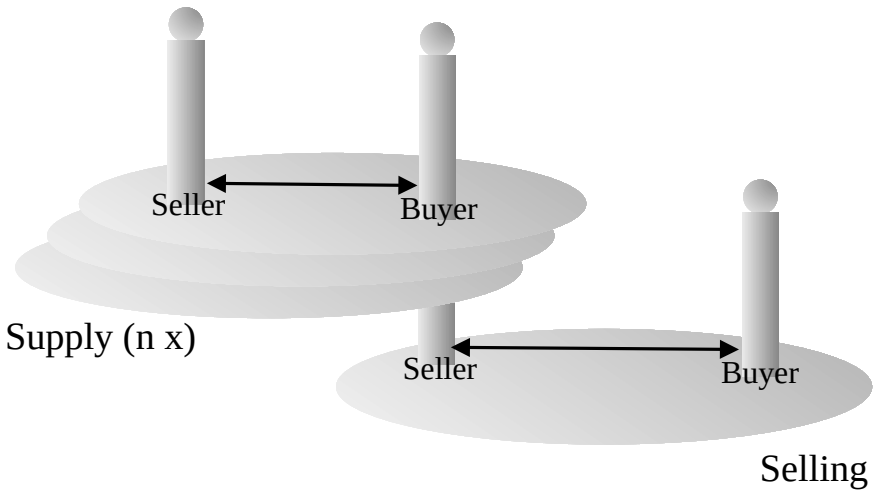


Fig. 10.1 – La structure du système de test

Le système contient deux organisations de même structure (cf. figure 10.2) ; par contre, leur fonctionnement diffère :

L'organisation *Supply* Les deux descriptions de rôles de l'organisation *Supply* sont représentés aux figures 10.3 et 10.4 : le *Buyer* se renseigne auprès du *Seller* sur le prix d'une marchandise, puis lui en achète jusqu'à ce qu'il n'ait plus d'argent.

L'organisation *Selling* Les deux descriptions de rôles de l'organisation *Selling* sont représentés aux figures 10.5 et 10.6 : le *Buyer* va, à intervalles aléatoires, commander une certaine quantité de marchandise au *Seller*. S'il dispose d'un stock suffisant, ce dernier va envoyer la marchandise et attendre son paiement ; dans le cas contraire, il va refuser la livraison.

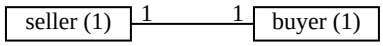


FIG. 10.2 – La structure des organisations *Supply* et *Selling*.

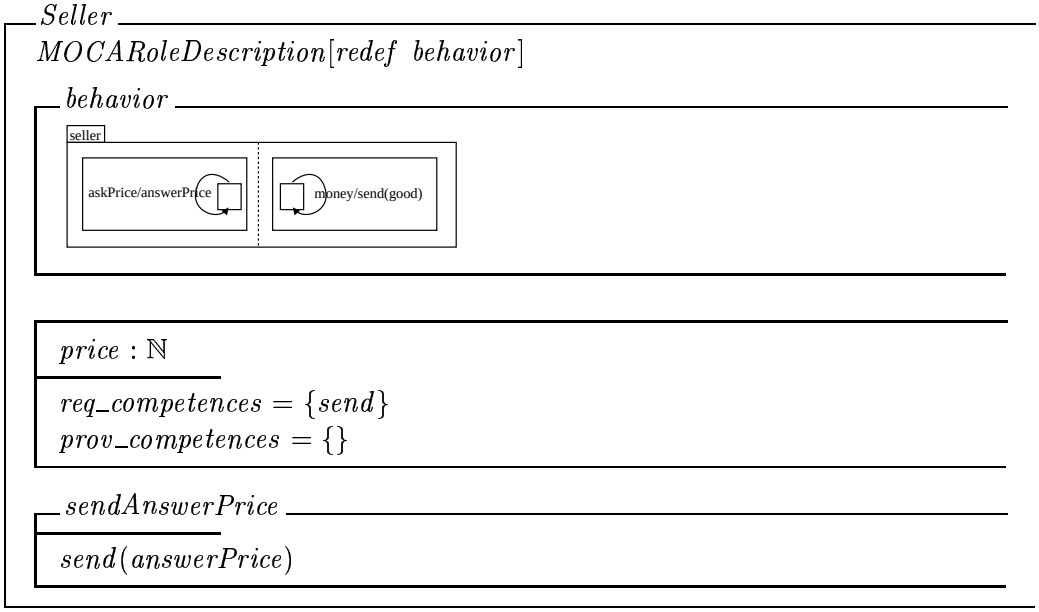


FIG. 10.3 – La description du rôle *Seller* dans l'organisation *Supply*.

On trouvera à l'annexe A des détails sur l'implémentation effective de ces organisations et descriptions de rôles.

Dans les tests que nous avons effectués, trois types d'agents étaient présents :

Le fournisseur a pour comportement de créer une nouvelle organisation *Supply* et d'y prendre le rôle de *Seller*.

L'intermédiaire a pour comportement de demander le rôle de *Buyer* dans chaque groupe instanciant *Supply* ainsi que de créer une nouvelle organisation *Selling* en y prenant le rôle de *Seller*.

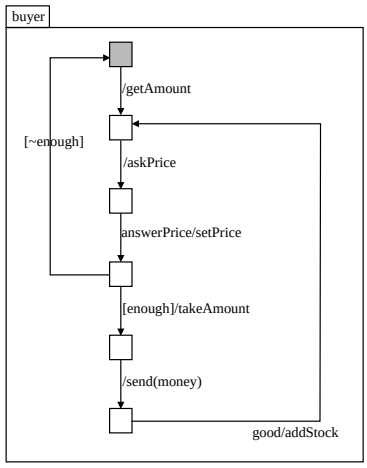
Le client a pour comportement de demander le rôle de *Buyer* dans chaque groupe instanciant *Selling*.

Pour l'étude du mécanisme de gestion des conflits de rôles, nous nous intéressons particulièrement à l'*intermédiaire* : celui-ci doit fournir à ses rôles une compétence *Purse* représentant un porte-monnaie. Le système est conçu de manière à ce que si l'intermédiaire ne fait partie que

Buyer

$MOCARoleDescription[redef\ \ behavior]$

behavior



$purseAmount, price : \mathbb{N}$

$req\_competences = \{send, getAmount, takeAmount, addStock\}$
$prov\_competences = \{\}$

askPrice

$send(askPrice)$

setPrice

$\Delta price$
$p?$

$price' = p?$

enough

$result! : \mathbb{B}$

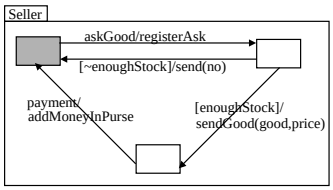
$result! \Leftrightarrow price < purseAmount$

FIG. 10.4 – La description du rôle *Buyer* dans l'organisation *Supply*.

Seller

$MOCARoleDescription[redef\ \ behavior]$

behavior



$askedQuantity, unitPrice : \mathbb{N}$

$req\_competences = \{send, putAmount, removeStock, getStock\}$
$prov\_competences = \{\}$

registerAsk

$\Delta askedQuantity$
$quantity?$

$askedQuantity' = quantity?$

enoughStock

$result! : \mathbb{B}$

$result! \Leftrightarrow askedQuantity \leq getStock$

SendGood

$send(askedQuantity, askedQuantity * unitPrice)$

addMoneyInPurse

$amount?$

$putAmount(amount?)$

FIG. 10.5 – La description du rôle *Seller* dans l'organisation *Selling*.

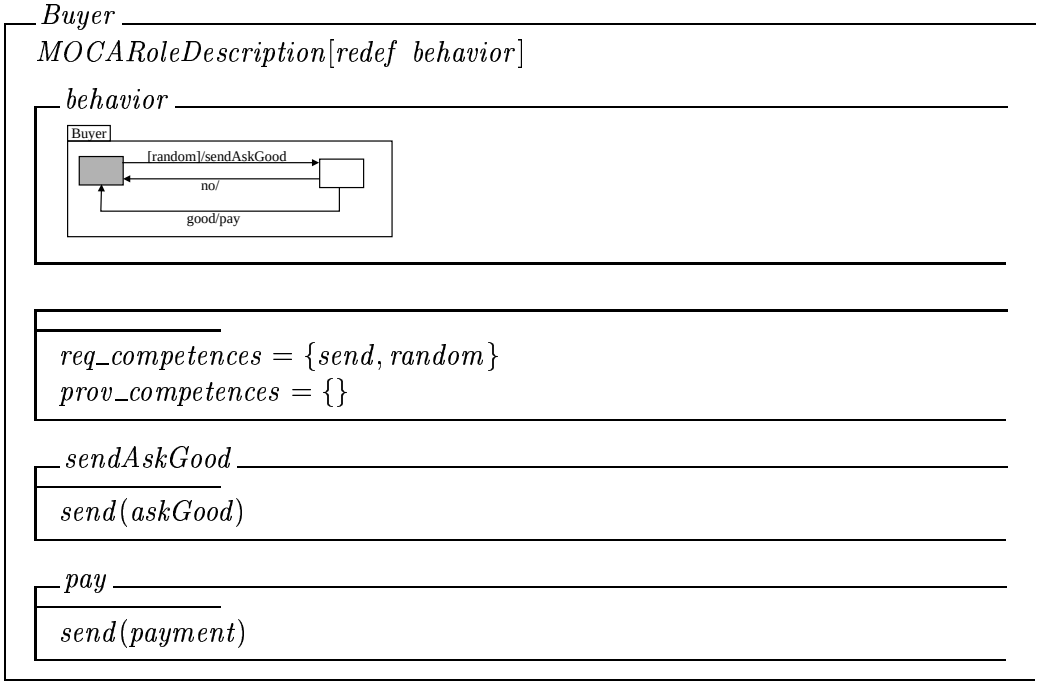


FIG. 10.6 – La description du rôle *Buyer* dans l'organisation *Selling*.

d'un groupe instanciant *Supply*, son avoir restera toujours supérieur ou égal à zéro. Cependant, s'il communique avec plusieurs fournisseurs, un conflit peut survenir entre ses différents besoins d'argent.

Pour gérer ce conflit, nous avons étudié quatre politiques d'acceptation distinctes pour la compétence *Purse* :

1. Une acceptation systématique des appels (pas de gestion des conflits).
2. Une réservation stricte (SR) : le porte-monnaie est réservé pour un composant depuis son premier appel jusqu'à un *release* de la compétence de débit.
3. Une réservation étendue (ER) : comme la réservation stricte, mais en acceptant toute entrée d'argent.
4. Une gestion au dernier moment (LM) : ne sont refusé que les appels de débit provoquant un passage de l'avoir en dessous de zéro.

Ces politiques sont implémentées en modifiant le code des méthodes *require* et *release* du composant *Purse* (cf. annexe A, page 136). Les mécanismes de gestion des conflits de rôles décrits au chapitre 6 peuvent alors entrer en jeu ; nous allons ici étudier leur effet sur différents aspects de l'exécution du système.

Entrelacement des rôles La politique d'acceptation influence de la manière suivante l'entrelacement des rôles :

Sans Tous les rôles s'exécutent en parallèle

SR Il y a toujours au plus un rôle actif à un moment donné (exclusion mutuelle sur les rôles)

ER Les rôles *Buyer* s'excluent mutuellement, mais les rôles *Seller* s'exécutent sans exclusion (car ils ne font que créditer de l'argent)

LM Tous les rôles s'exécutent en parallèle, mais avec blocage lorsque le montant disponible n'est pas suffisant

Évolution de l'avoir La figure 10.7 représente l'évolution de l'avoir en fonction du temps. On constate que pour SR, l'avoir descend toujours à zéro avant de remonter ; ceci est dû à l'exclusion mutuelle des rôles évoquée ci-dessus. Pour les autres politiques, un crédit peut survenir à tout moment. On remarquera que sans gestion des conflits, l'avoir fini par passer en-dessous de zéro (fig. 10.7(a)).

Taux de refus La figure 10.8 présente les taux de refus moyens résultant des différentes politiques. Le taux le plus bas est bien sûr atteint par la première, qui ne refuse aucun appel. Par contre, elle n'assure pas une exécution correcte, puisque l'avoir peut être négatif. La politique LM assure également des taux assez bas, tout en gardant l'avoir non négatif ; cependant, elle n'assure qu'une version faible de la correction de l'exécution. En effet, si elle assure que l'avoir reste positif, elle n'assure pas à un rôle qu'il disposera du montant qu'il attend. Les deux autres politiques sont plus fortes : un rôle se renseignant sur l'avoir peut compter sur le montant obtenu jusqu'à ce qu'il envoie un *release*.

Temps de blocage La figure 10.9 représente la répartition des temps de blocages résultant des différentes politiques. Pour l'obtenir, les temps de blocages ont été arrondis à la demi-seconde la plus proche ; les données ont ensuite été représentées avec en abscisse le temps de blocage (en secondes) et en ordonnée le nombre de blocage de cette durée.

Cette petite série de tests permet donc de constater l'importance de la politique d'acceptation dans le processus de gestion des conflits. C'est cette politique qui va déterminer le type d'entrelacement des rôles, le niveau d'ignorance mutuelle, le taux d'acceptation et les temps de blocage.

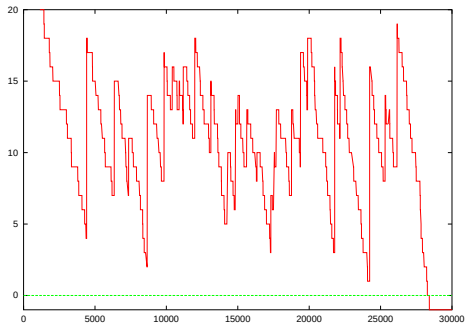
On relèvera que ces tests ont été réalisés en ne changeant que le composant *Purse* dans l'agent *intermédiaire*. En particulier, aucun changement n'a été effectué au niveau de la spécification organisationnelle. Cela remplit donc le but que nous nous étions fixé pour ce mécanisme de gestion des conflits : le concepteur d'une organisation n'a pas besoin de connaître son contexte d'utilisation. Par contre, l'exécution des rôles dans une instance de cette organisation s'adaptera au contexte en fonction des politiques d'acceptation des différents composants en jeu.

10.2 Simulation d'épizootie

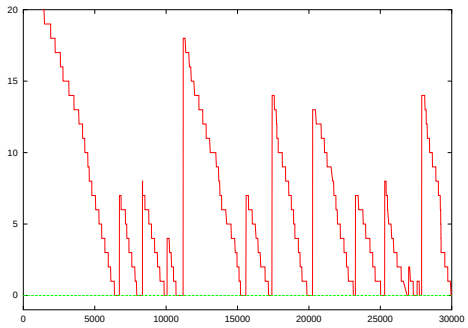
Le test que nous présentons dans ce paragraphe a pour but de démontrer l'utilisabilité de notre plate-forme. Pour ce faire, nous avons choisi de reprendre une spécification développée par Vincent Hilaire [Hil00] ; en effet, la facilité de développement que nous revendiquons pour notre plate-forme repose en grande partie sur l'utilisation de son formalisme.

La spécification que nous reprenons est celle d'une simulation d'épizootie de fièvre aphteuse. Proposée à l'origine par Benoît Durand [Dur96], elle a fait l'objet d'une reformulation par Hilaire [Hil00]. Nous allons rapidement présenter cette dernière.

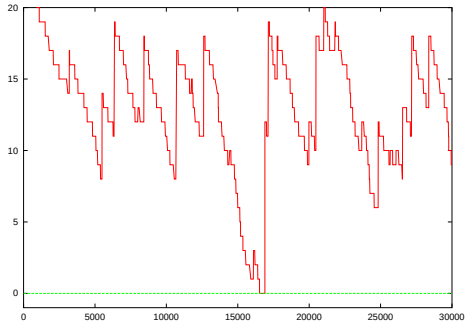
L'épizootie se déroule au sein d'un cheptel ; celui-ci est mis à l'étable l'hiver et à l'herbage l'été. La contagion ne peut se produire qu'à l'étable.



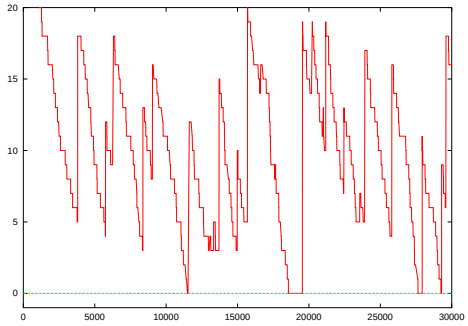
(a) Sans



(b) SR



(c) ER



(d) LM

FIG. 10.7 – Évolution de l'avoir au cours du temps pour les différentes politiques d'acceptation

La figure 10.10 représente la structure du système résultant, composé de trois types d'agents et de deux organisations. La signification des agents *Éleveur* et *Cheptel* est claire ; on notera que la *Maladie* a également été agentifiée.

Nous allons maintenant décrire rapidement les deux organisations.

L'organisation *Production* est composée de deux rôles ; le rôle *Gestion Troupeau* est responsable de la vente des animaux et de leur déplacements vers le pâturage au printemps et vers l'étable en automne. Le rôle *Production* est responsable de simuler la naissance des veaux.

L'organisation *Maladie* est composée de trois rôles ; le rôle *Suivi Sanitaire* est responsable de détecter la présence de la maladie et d'effectuer un traitement ; le rôle *Site Contaminant* s'occupe de l'effet du traitement et le rôle *Anadémie* simule la contagion et la guérison.

Les statecharts correspondant à chacun de ces rôles sont représentés aux figures 10.11 et 10.12. Il s'agit des statecharts que nous avons implémentés, qui sont de légères adaptations de ceux d'Hilaire :

– Dans la version d'Hilaire, les rôles *Production* et *Gestion Troupeau* ne communiquent pas explicitement ; par contre, ils possèdent des attributs en commun. La politique

Politique	Taux de refus		Correct ?
	2 Suppliers	5 Suppliers	
Sans	0%	0%	non
SR	30%	30%	oui
ER	6%	10%	oui
LM	4%	14%	oui, mais

FIG. 10.8 – Taux de refus des appels de compétences

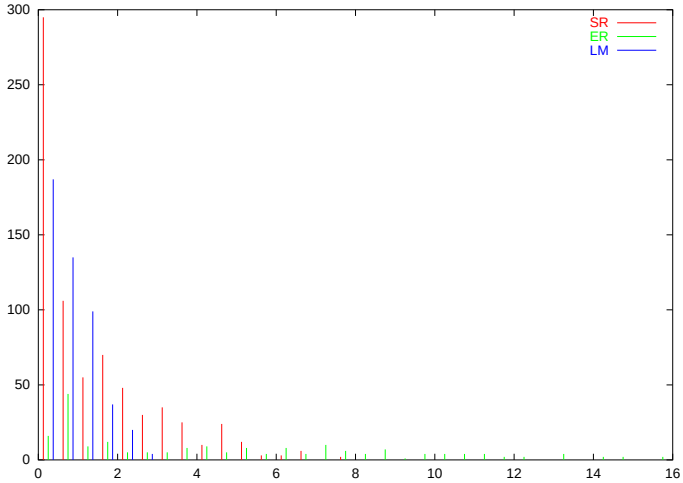


FIG. 10.9 – Temps de blocages

beaucoup plus stricte de MOCA en ce qui concerne la séparation des rôles nous a donc forcé à rajouter l'état O2 du rôle *Production*.

– Des raisons similaires de prise en compte explicite de la communication sont à l'origine de quelques adaptations des statecharts des rôles *Anadémie* et *Site Contaminant*.
– Le rôle *Suivi Sanitaire* a été simplifié sous une forme fonctionnellement équivalente.

Nous avons traduit la partie Object-Z de la spécification d'Hilaire en Java pour en faire des compétences MOCA.

Enfin, nous avons adapté le mécanisme d'ordonnancement des agents : chez Hilaire, celui-ci est réalisé par un rôle environnemental alors que dans MOCA, nous utilisons un agent ordonnanceur de MadKit [Gut01].

La figure 10.13 donne le résultat d'une exécution typique du système MOCA obtenu. Les résultats sont qualitativement semblables à ceux obtenus par Hilaire (figure 10.14).

Nous avons donc obtenu avec un effort minimum une réimplémentation d'un système d'Hilaire. Les adaptations nécessitées sont minimes et ne modifient pas la structure générale du système. Par contre, nous avons produit un SMA complet et opérationnel, alors qu'Hilaire se limite à l'animation de sa spécification.

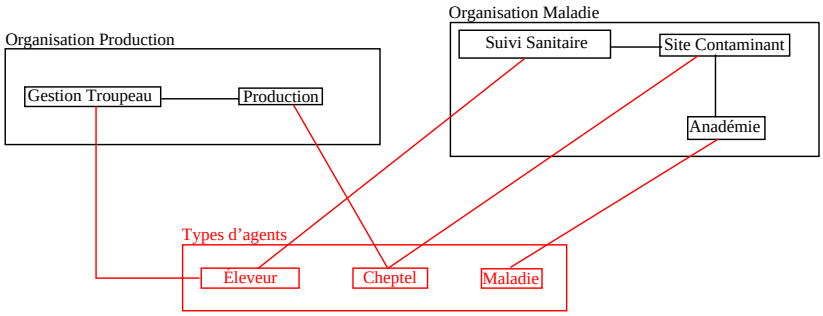


FIG. 10.10 – La structure de la simulation d'épizootie

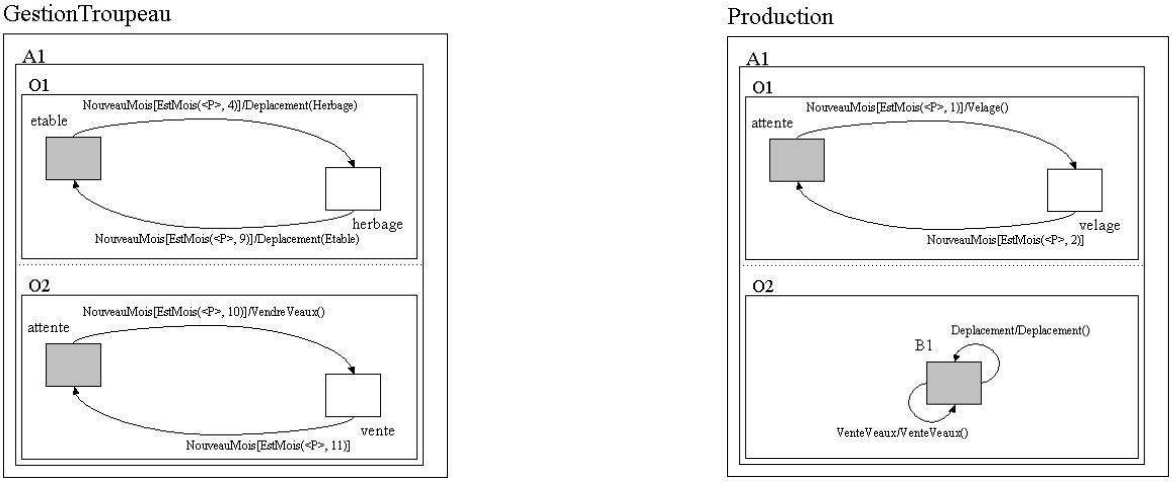


FIG. 10.11 – Les rôles de l'organisation *Production*

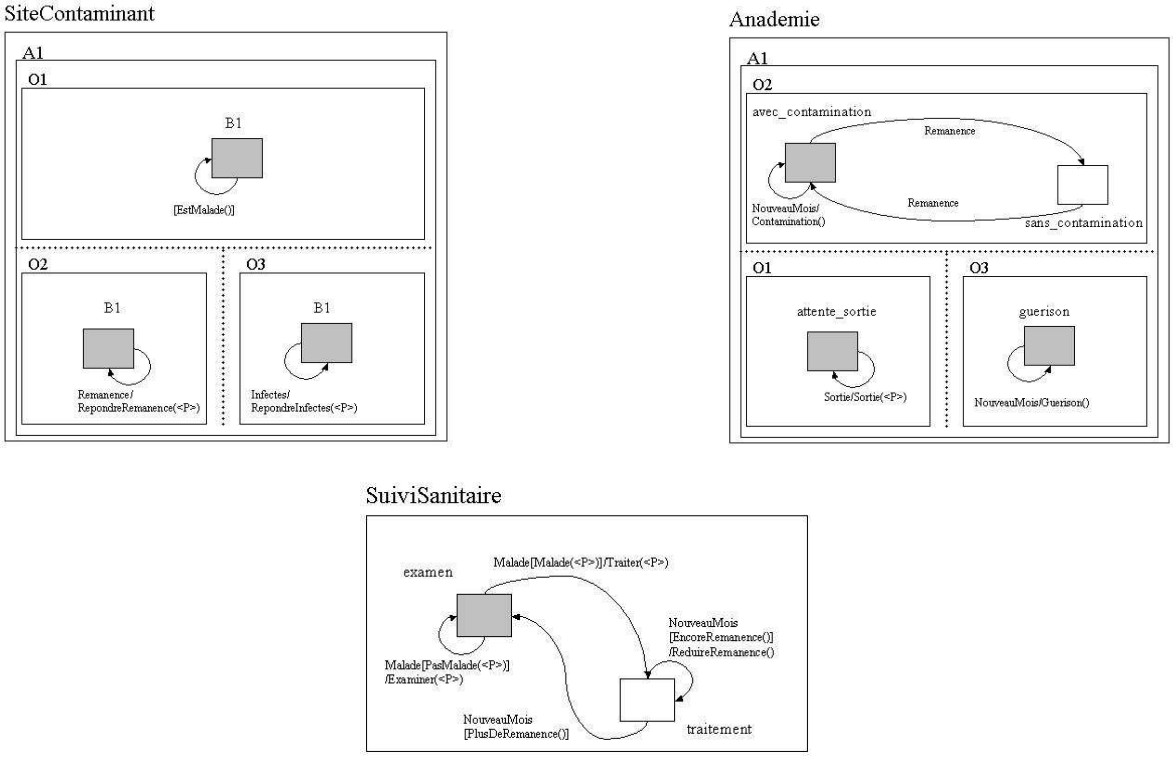


FIG. 10.12 – Les rôles de l'organisation *Maladie*

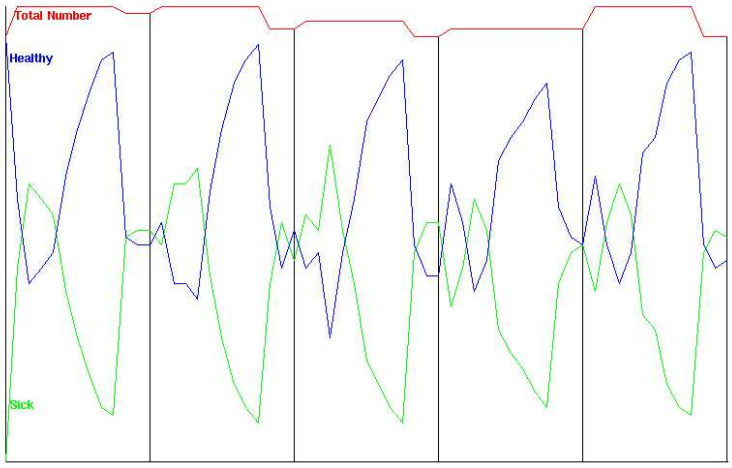


FIG. 10.13 – Les résultats de la simulation par MOCA

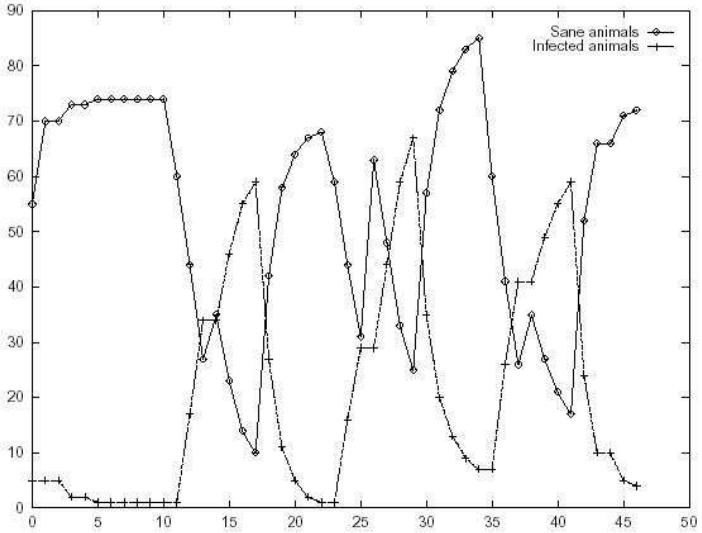


FIG. 10.14 – Les résultats de la simulation d'Hilaire

Quatrième partie

Conclusion et annexes

Chapitre 11

Conclusion et perspectives

La question initiale de cette thèse était celle de la prise de rôles multiples dans un modèle de SMA organisationnel. Pour que cette question ait un sens non trivial, il faut se placer dans un cadre *comportementaliste* — c'est-à-dire qu'il associe des comportements aux rôles — et *dynamique* — à savoir qu'il admet l'évolution des organisations en cours d'exécution.

Nous avons vu au chapitre 2 qu'aucun modèle existant ne pouvait réellement servir de cadre à notre question. Il manque à Cassiopée [CPD96] la structuration en niveaux qui permet une bonne articulation de la description de la structure organisationnelle d'une part et de sa réalisation d'autre part ; Aalaadin [FG98] propose un modèle organisationnel dynamique tenant compte de cette structuration en niveaux, mais n'associe pas réellement de comportements aux rôles ; MOISE+ [HSB02b] et les modèles de Durand [Dur96] et d'Hilaire [Hil00] se trouvent dans la situation inverse ; [PO01] combine ces deux aspects, mais ne propose aucune opérationnalisation de ses concepts ; [Dur00] propose lui une combinaison opérationnelle des aspects comportementalistes et dynamiques, mais nous avons vu que son approche présente plusieurs faiblesses par rapport à celles citées ci-dessus ; enfin, Gaia [WJK00] propose un cadre méthodologique intéressant, mais la disparition des concepts organisationnels au moment de la conception le rend inadéquat pour notre démarche.

Nous avons donc présenté dans la deuxième partie notre modèle, nommé MOCA, qui hérite d'Aalaadin pour le cadre général, des travaux d'Hilaire pour la représentation des rôles, et de quelques éléments des autres approches. De plus, des considérations de souplesse et de réutilisabilité nous ont poussé à adopter une architecture componentielle pour nos agents. Nous rappelons brièvement ici les caractéristiques résultant de cette approche :

- Il est possible de décrire des *organisations*, qui sont des vues partielles sur le fonctionnement du SMA. Ces organisations sont formées de *descriptions de rôles* et de *relations* entre elles (chapitre 5).
- Une organisation peut être instanciée sous forme d'un *groupe* ; les descriptions de rôles y deviennent des *rôles* et les relations des *accointances* (chapitre 5).
- Les agents peuvent *prendre* un rôle et le *quitter en cours d'exécution* (chapitre 7).
- La prise d'un rôle est *contraignante* : l'agent doit suivre les comportements prescrits par les rôles qu'il adopte (chapitre 5).
- Si les comportements associés aux rôles sont décrits par des statecharts, il existe une manière automatique de *gérer les conflits* entre rôles (chapitre 6).
- Les *communications entre agents* consistent en des échanges d'*influences* qui sont limités à celles qui se passent *à l'intérieur des groupes*. Ceci permet d'assurer une bonne

indépendance conceptuelle et une réutilisabilité accrue des organisations. De plus, cela contribue à fournir un cadre adéquat pour la validation (chapitre 8).
- À l'intérieur de l'agent, les communications sont assurées par le Module de Gestion des Composants. Ce module est responsable d'une part de lier le comportement d'un rôle aux compétences de l'agent ; d'autre part, il est indirectement et localement responsable de la coordination inter-groupes au niveau du système. C'est également à ce niveau que sont gérés les conflits entre rôles (chapitre 6).

Ce modèle a été opérationnalisé par une plate-forme organisationnelle, construite au-dessus de MadKit (chapitre 9). Le chapitre 10 a présenté quelques tests du mécanisme de gestion de conflits de rôles ainsi que la réimplémentation d'une simulation de Vincent Hilaire pour démontrer l'utilisabilité de notre plate-forme.

MOCA constitue donc une approche complète, munie d'un pan conceptuel et d'un pan opérationnel. La plate-forme que nous avons réalisée est utilisable et semble bien convenir à la simulation ou au prototypage. Il est clair cependant que le présent travail n'est en rien définitif et que de nombreuses ouvertures restent encore à explorer.

Dans l'immédiat, nous voyons deux pistes qui pourraient compléter et consolider notre approche :

Outillage Actuellement, la spécification organisationnelle des systèmes doit se faire par des fichiers XML. Nous envisageons de fournir à l'utilisateur une interface graphique pour la conception des organisations et des descriptions de rôles. De cette manière, le code à taper serait réduit au minimum (les compétences spécifiques des rôles), permettant ainsi au concepteur de se concentrer sur les fonctionnalités du système qu'il développe.

Tests Parallèlement, il est nécessaire de poursuivre les tests en implémentant d'autres exemples que ceux présentés au chapitre 10. Nous sommes convaincus que notre approche permet d'augmenter considérablement la réutilisabilité lors du développement d'un SMA, mais cette affirmation ne sera bien sûr vérifiable qu'après l'implémentation d'un nombre non négligeable de systèmes. Notons que ces tests seront d'autant plus facilités que l'outillage évoqué ci-dessus sera avancé.

A moyen terme, nous désirons reprendre quelques points de notre modèle qui nous semblent présenter des faiblesses. Nous relèverons en particulier :

Redondance des compétences Il est théoriquement possible dans notre modèle qu'un agent possède plusieurs composants fournissant la même compétence. Cette situation peut se présenter dans au moins deux cas :
- L'agent peut être conçu avec cette redondance ; par exemple, si un agent doit pouvoir gérer séparément deux sommes d'argent, il peut être plus simple de le munir de deux composants gérant chacun une somme, plutôt que d'un composant gérant les deux sommes.
- Un agent peut prendre deux fois le même rôle dans des groupes différents mais instanciant la même organisation. Si ce rôle fournit des compétences, celles-ci seront alors redondantes.

Cependant, nous ne proposons pour l'instant aucun mécanisme général pour gérer une telle situation, qui doit être traitée au cas par cas dans le type d'agent. Il serait donc intéressant d'étendre notre modèle de MGC pour fournir une gestion générique de l'appel de compétences redondantes.

Redondance des rôles Le modèle d'adressage des influences dans MOCA ne permet pas qu'un agent prenne deux fois le même rôle dans le même groupe. Si ceci n'est pas gênant dans la plupart des situations, on peut rencontrer des cas où cela constitue une limitation (par exemple un agent désirant faire deux offres différentes dans le même réseau contractuel). Une extension simple de l'adressage permettrait de supprimer cette contrainte.

Décentralisation de la gestion des groupes L'idée de faire passer la gestion des groupes par un gestionnaire est un héritage d'Aalaadin. Ceci permet d'assurer une certaine sécurité dans le système. Cependant, ce gestionnaire peut devenir un goulot d'étranglement dans une application exigeante, ainsi qu'une source de faiblesse en cas de panne. En particulier, le gestionnaire du Groupe de Gestion occupe une position extrêmement centrale qui semble peu dans l'esprit de décentralisation qui gouverne les SMA. Il serait donc nécessaire de proposer un modèle alternatif et décentralisé de la gestion des groupes. Il est à noter que ce changement n'implique pas de modification des concepts centraux de MOCA ; il suffit de modifier l'Organisation de Gestion.

Validation En exprimant des conditions sous lesquelles la validité d'un protocole peut être ramenée à des conditions au bord sur le comportement d'un agent, le chapitre 8 propose des nouvelles pistes pour la validation des SMA. Cependant, la démarche complète inclurait la validation des protocoles, la localisation de cette validation aux agents et une validation de ces derniers tenant compte de ces nouvelles conditions ; il est clair que tous les problèmes inhérents à cette approche ne sont pas encore résolus.

Extension du formalisme Le formalisme de Vincent Hilaire pour la représentation des comportements des rôles propose un bon compromis entre puissance et facilité d'utilisation ; cependant, il se heurte à des limites d'expressivité dans certains cas. Par exemple, il est difficile de spécifier dans ce formalisme la réaction à une combinaison d'influences concurrentes (telles des forces).

Ceci est visible de manière plus générale dans la spécification des composants : le seul outil formel que nous proposons, les statecharts, se révèle parfois insuffisant. Dans la plate-forme MOCA, nous autorisons l'expression de cette spécification directement en Java ; mais au niveau du modèle formel, il y a là une nette carence d'expressivité.

Une piste particulièrement prometteuse nous paraît être l'utilisation du formalisme DEVS [DRG02], qui permet une encapsulation de différents formalismes (automates, réseaux de Petri, équations différentielles) dans des objets formels présentant de nombreuses similitudes avec notre notion de composant.

Les ouvertures ci-dessus représentent des compléments ou modifications relativement modestes de notre approche. Nous désirons finir sur des perspectives plus larges sur lesquelles ouvre ce travail ; il s'agit là de pistes de recherche à plus long terme nécessitant des modifications profondes ou des ajouts conséquents à notre modèle.

Raisonnement sur les compétences Nous avons relevé plusieurs fois au cours de cette thèse que les motivations qui poussent un agent à prendre ou abandonner un rôle dépendent du type d'agent et ne font donc pas partie de MOCA. Cependant, les choix d'architecture de MOCA pourraient fournir un moteur efficace pour cette dynamique : le raisonnement sur les compétences. En effet, un agent désirant prendre un rôle dans un groupe peut se rendre compte qu'il n'a pas les compétences nécessaires ; dans ce cas, il peut chercher un ou des autres rôles qui les lui fourniraient ; ce mécanisme permet de

retrouver la traditionnelle décomposition de buts en sous-buts. De plus, avant de quitter un rôle, l'agent devrait vérifier qu'il ne perd pas ainsi des compétences dont il a besoin ailleurs.

Agents "désobéissants" Nous avons évoqué au paragraphe 6.4 un cas dans lequel il serait utile que l'agent intervienne activement dans le déroulement de ses rôles au lieu de se contenter d'en gérer l'exécution. En l'occurrence, nous proposions d'intercepter et de modifier le retour d'un appel de compétence, mais on pourrait aussi imaginer que l'agent déroute des influences vers un autre rôle que leur destinataire ou qu'il modifie le contenu de certaines influences entrantes ou sortantes.

Une étude approfondie de la marge de manoeuvre de l'agent, c'est-à-dire des actions qu'il peut entreprendre de son propre chef pour influencer l'exécution de ses rôles — mais sans provoquer d'erreur dans le déroulement de ceux-ci — permettrait d'étendre encore l'autonomie de l'agent dans notre modèle.

Séparation des aspects internes et externes des rôles Nous avons vu au chapitre 5 que le rôle possède un statut un peu ambigu au sein de l'agent, dans le sens qu'il est à la fois interne et externe. Cela permet une sémantique opérationnelle assez naturelle de la prise de rôle, mais simultanément cela limite l'indépendance entre la spécification organisationnelle du système et l'architecture des agents. Il pourrait être intéressant d'adopter une démarche plus "pure" dans laquelle la spécification organisationnelle s'arrêterait à l'*interface* du rôle (au sens du chapitre 8), laissant à l'agent le soin de l'implémentation.

On notera cependant que cette approche, bien que conceptuellement séduisante, doublerait le travail du concepteur : en plus de la spécification organisationnelle, il doit aussi s'occuper de sa réalisation dans les agents. De plus, il peut être délicat de vérifier efficacement au moment de l'exécution si un agent est capable de prendre un rôle (cette vérification revient au calcul du langage associé à l'automate implémentant l'interface de rôle).

Évolution des organisations La représentation explicite dans le système lui-même des concepts organisationnels permet aux agents d'y avoir accès, de raisonner sur leur base et éventuellement même de les manipuler. Le calcul des zones d'exclusion mutuelle présenté au chapitre 6 en est un exemple, mais on pourrait imaginer aller beaucoup plus loin, et permettre aux agents de modifier les organisations présentes dans le système ou d'en créer des nouvelles. Techniquement, cela est déjà possible dans MOCA, mais nous ne possédons aucun outil conceptuel pour le faire de manière contrôlée.

Le chemin à parcourir est donc encore long et MOCA n'en constitue qu'une modeste étape. Mais si notre approche pouvait apporter sa contribution, aussi minime fût-elle, à une théorie permettant de comprendre les rapports entre interactions locales et comportement global d'un système organisé, nous en serions comblés.

Épilogue

Le progrès scientifique suit parfois les chemins les plus inattendus. On sait qu'un des problèmes fondamentaux de l'approche émergentiste des SMA est celui de l'observation :

> « En ce qui concerne l'observation, nous avons vu qu'il faut un observateur du phénomène global pour qu'il y ait émergence. Ce point soulève une ambiguïté car l'observateur peut rester extérieur au système auquel cas on lui demande un effort d'interprétation dans une direction qui n'est pas intrinsèquement contenue dans la dynamique du système ou l'observateur peut lui-même être partie prenante de la dynamique d'ensemble de par ses possibilités d'interaction avec le système auquel cas il est le moteur même de l'émergence. Dans ce second cas, l'interaction de l'utilisateur/observateur peut faire émerger plus que ce que l'utilisateur ou le système ne pourraient faire isolément. » [Jea97]

Notre approche n'avait au départ aucune prétention de s'attaquer à ce problème difficile. Et pourtant, c'est en allant présenter MOCA à MABS'02 [AMBN] que nous sommes tombés sur un dispositif ingénieux qui pourrait y apporter une solution élégante (cf. figure 11.1).

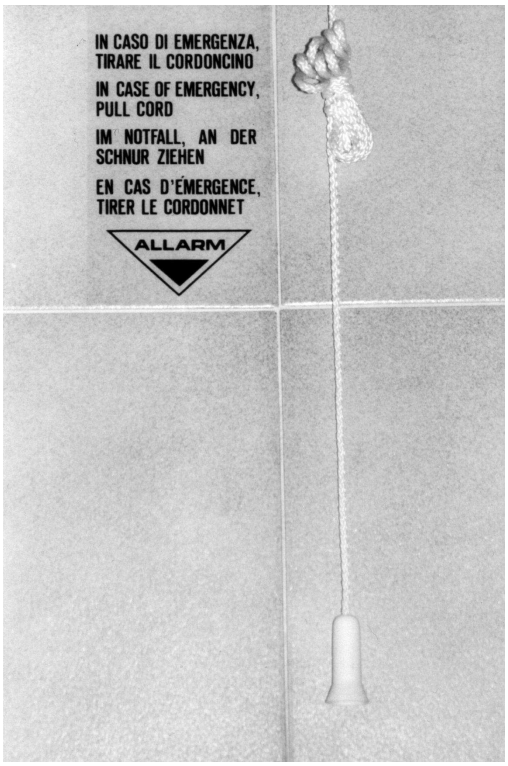


Fig. 11.1 – Un dispositif d'observation de l'émergence

En munissant un tel appareil d'une connexion USB, cela rendrait la résolution de problèmes par émergence accessible à une grande majorité des machines sur le marché actuellement. Une demande de brevet est à l'étude.

Annexe A

La spécification organisationnelle d'un système MOCA et son implémentation

A.1 Le format de document

Les spécifications organisationnelles de la plate-forme MOCA doivent suivre la structure définie par la DTD suivante :

```
<!ELEMENT MOCA_FILE ((IMPORT | PERTURBATIONTYPE | STATECHART |  ↩
   ORGANISATIONSCHEME)*)>
        <!ATTLIST MOCA_FILE version CDATA #REQUIRED>
        <!ATTLIST MOCA_FILE date CDATA #IMPLIED>
<!ELEMENT IMPORT EMPTY>
        <!ATTLIST IMPORT file CDATA #REQUIRED>
<!ELEMENT PERTURBATIONTYPE (ATTRIBUTEDESCRIPTION*)>
        <!ATTLIST PERTURBATIONTYPE name CDATA #REQUIRED>
<!ELEMENT ATTRIBUTEDESCRIPTION EMPTY>
        <!ATTLIST ATTRIBUTEDESCRIPTION name CDATA #REQUIRED>
        <!ATTLIST ATTRIBUTEDESCRIPTION type CDATA #REQUIRED>
        <!ATTLIST ATTRIBUTEDESCRIPTION def CDATA #IMPLIED>
<!ELEMENT PERTURBATIONTYPENAME EMPTY>
        <!ATTLIST PERTURBATIONTYPENAME name CDATA #REQUIRED>
<!ELEMENT STATECHART (ORSTATE)>
<!ELEMENT BASICSTATE EMPTY>
        <!ATTLIST BASICSTATE name CDATA #REQUIRED>
<!ELEMENT ORSTATE (((LINKEDSTATE | BASICSTATE | ORSTATE | ANDSTATE) ↩
   *),(TRANSITION*))>
        <!ATTLIST ORSTATE name CDATA #REQUIRED>
        <!ATTLIST ORSTATE start CDATA #IMPLIED>
<!ELEMENT ANDSTATE ((LINKEDSTATE | ORSTATE)*)>
        <!ATTLIST ANDSTATE name CDATA #REQUIRED>
<!ELEMENT LINKEDSTATECHART EMPTY>
        <!ATTLIST LINKEDSTATECHART name CDATA #REQUIRED>
        <!ATTLIST LINKEDSTATECHART newname CDATA #IMPLIED>
<!ELEMENT TRANSITION (SOURCE,DESTINATION,(EVENT?),(CONDITION?),( ↩
   ACTION?))>
```

```
        <!ATTLIST TRANSITION forAllPerturbations CDATA #IMPLIED>
<!ELEMENT SOURCE (STATEPATH*)>
<!ELEMENT DESTINATION (STATEPATH*)>
<!ELEMENT STATEPATH EMPTY>
        <!ATTLIST STATEPATH path CDATA #REQUIRED>
<!ELEMENT EVENT EMPTY>
        <!ATTLIST EVENT perturbationtype CDATA #REQUIRED>
<!ELEMENT ACTION (PARAMETER*)>
        <!ATTLIST ACTION skill CDATA #REQUIRED>
        <!ATTLIST ACTION method CDATA #REQUIRED>
<!ELEMENT CONDITION (PARAMETER*)>
        <!ATTLIST CONDITION skill CDATA #REQUIRED>
        <!ATTLIST CONDITION method CDATA #REQUIRED>
<!ELEMENT PARAMETER EMPTY>
        <!ATTLIST PARAMETER value CDATA #IMPLIED>
<!ELEMENT ORGANISATIONSCHEME ((ROLEDESCRIPTION*),(RELATIONSIDE*))>
        <!ATTLIST ORGANISATIONSCHEME name CDATA #REQUIRED>
        <!ATTLIST ORGANISATIONSCHEME manager CDATA #IMPLIED>
<!ELEMENT ROLEDESCRIPTION ((PUBLICSKILL*),(PRIVATESKILL*),( ↩
   SKILLNEEDED*))>
        <!ATTLIST ROLEDESCRIPTION name CDATA #REQUIRED>
        <!ATTLIST ROLEDESCRIPTION cardinality CDATA #REQUIRED>
        <!ATTLIST ROLEDESCRIPTION behavior CDATA #REQUIRED>
<!ELEMENT PUBLICSKILL EMPTY>
        <!ATTLIST PUBLICSKILL class CDATA #REQUIRED>
<!ELEMENT PRIVATESKILL EMPTY>
        <!ATTLIST PRIVATESKILL class CDATA #REQUIRED>
<!ELEMENT SKILLNEEDED EMPTY>
        <!ATTLIST SKILLNEEDED class CDATA #REQUIRED>
<!ELEMENT RELATIONSIDE (PERTURBATIONTYPENAME*)>
        <!ATTLIST RELATIONSIDE fromRole CDATA #REQUIRED>
        <!ATTLIST RELATIONSIDE toRole CDATA #REQUIRED>
        <!ATTLIST RELATIONSIDE cardinality CDATA #REQUIRED>
```

A.2 Exemple

À titre d'exemple, nous donnons ici la définition complète de l'organisation *Selling* du chapitre 10 :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MOCA_FILE SYSTEM "MOCA.dtd">
<!-- MOCA 2.1 XML specification: Tests for the conflict management ↩
   algorithm, Selling Organization-->
<!-- Amiguet Matthieu   : matthieu.amiguet@unine.ch -->
<!-- Baez Jose          : jose.baez@unine.ch -->
<!-- Muller Jean-Pierre : jean-pierre.muller@cirad.fr -->
<!-- Nagy Adina         : adina.nagy@unine.ch -->
<MOCA_FILE date="december␣02" version="2.1">
  <PERTURBATIONTYPE name="AskGoodAndPrice">
    <ATTRIBUTEDESCRIPTION name="Quantity" type="java.lang.Integer" ↩
       />
  </PERTURBATIONTYPE>
```

```
<PERTURBATIONTYPE name="GoodNotAvailable"/>
<PERTURBATIONTYPE name="Payment">
  <ATTRIBUTEDESCRIPTION name="Amount" type="java.lang.Integer"/>
</PERTURBATIONTYPE>
<PERTURBATIONTYPE name="GoodAndPrice">
  <ATTRIBUTEDESCRIPTION name="GoodAmount" type="java.lang.Integer ↩
     "/>
  <ATTRIBUTEDESCRIPTION name="Price" type="java.lang.Integer"/>
</PERTURBATIONTYPE>
<STATECHART>
  <ORSTATE name="Selling.Seller" start="A">
    <BASICSTATE name="A"/>
    <BASICSTATE name="B"/>
    <BASICSTATE name="C"/>
    <TRANSITION>
      <SOURCE>
        <STATEPATH path="A"/>
      </SOURCE>
      <DESTINATION>
        <STATEPATH path="B"/>
      </DESTINATION>
      <EVENT perturbationtype="AskGoodAndPrice"/>
      <ACTION method="registerAsk" skill="mocaConflicts.skills. ↩
         SellingSellerPrivateSkill">
        <PARAMETER/>
      </ACTION>
    </TRANSITION>
    <TRANSITION>
      <SOURCE>
        <STATEPATH path="B"/>
      </SOURCE>
      <DESTINATION>
        <STATEPATH path="A"/>
      </DESTINATION>
      <CONDITION method="notEnoughStock" skill="mocaConflicts. ↩
         skills.SellingSellerPrivateSkill"/>
      <ACTION method="sendGoodNotAvailable" skill="mocaConflicts. ↩
         skills.SellingSellerPrivateSkill"/>
    </TRANSITION>
    <TRANSITION>
      <SOURCE>
        <STATEPATH path="B"/>
      </SOURCE>
      <DESTINATION>
        <STATEPATH path="C"/>
      </DESTINATION>
      <CONDITION method="enoughStock" skill="mocaConflicts.skills ↩
         .SellingSellerPrivateSkill"/>
      <ACTION method="sendGoodAndPrice" skill="mocaConflicts. ↩
         skills.SellingSellerPrivateSkill"/>
    </TRANSITION>
    <TRANSITION>
      <SOURCE>
```

```
        <STATEPATH path="C"/>
      </SOURCE>
      <DESTINATION>
        <STATEPATH path="A"/>
      </DESTINATION>
      <EVENT perturbationtype="Payment"/>
      <ACTION method="addMoneyInPurse" skill="mocaConflicts. ↩
          skills.SellingSellerPrivateSkill">
        <PARAMETER/>
      </ACTION>
    </TRANSITION>
  </ORSTATE>
</STATECHART>
<STATECHART>
  <ORSTATE name="Selling.Buyer" start="A">
    <BASICSTATE name="A"/>
    <BASICSTATE name="B"/>
    <TRANSITION>
      <SOURCE>
        <STATEPATH path="A"/>
      </SOURCE>
      <DESTINATION>
        <STATEPATH path="B"/>
      </DESTINATION>
      <CONDITION method="randomCond" skill="mocaConflicts.skills. ↩
          SellingBuyerPrivateSkill"/>
      <ACTION method="sendAskGoodAndPrice" skill="mocaConflicts. ↩
          skills.SellingBuyerPrivateSkill"/>
    </TRANSITION>
    <TRANSITION>
      <SOURCE>
        <STATEPATH path="B"/>
      </SOURCE>
      <DESTINATION>
        <STATEPATH path="A"/>
      </DESTINATION>
      <EVENT perturbationtype="GoodNotAvailable"/>
    </TRANSITION>
    <TRANSITION>
      <SOURCE>
        <STATEPATH path="B"/>
      </SOURCE>
      <DESTINATION>
        <STATEPATH path="A"/>
      </DESTINATION>
      <EVENT perturbationtype="GoodAndPrice"/>
      <ACTION method="sendPayment" skill="mocaConflicts.skills. ↩
          SellingBuyerPrivateSkill">
        <PARAMETER/>
      </ACTION>
    </TRANSITION>
  </ORSTATE>
</STATECHART>
```

```
  <ORGANISATIONSCHEME name="Selling">
    <ROLEDESCRIPTION behavior="Selling.Seller" cardinality="1" name ↩
       ="Seller">
      <PRIVATESKILL class="mocaConflicts.skills. ↩
         SellingSellerPrivateSkill"/>
      <SKILLNEEDED class="mocaConflicts.skills.Purse"/>
      <SKILLNEEDED class="mocaConflicts.skills.Stock"/>
    </ROLEDESCRIPTION>
    <ROLEDESCRIPTION behavior="Selling.Buyer" cardinality="0" name= ↩
       "Buyer">
      <PRIVATESKILL class="mocaConflicts.skills. ↩
         SellingBuyerPrivateSkill"/>
    </ROLEDESCRIPTION>
    <RELATIONSIDE cardinality="1" fromRole="Seller" toRole="Buyer">
      <PERTURBATIONTYPENAME name="GoodAndPrice"/>
      <PERTURBATIONTYPENAME name="GoodNotAvailable"/>
    </RELATIONSIDE>
    <RELATIONSIDE cardinality="0" fromRole="Buyer" toRole="Seller">
      <PERTURBATIONTYPENAME name="AskGoodAndPrice"/>
      <PERTURBATIONTYPENAME name="Payment"/>
    </RELATIONSIDE>
  </ORGANISATIONSCHEME>
</MOCA_FILE>
```

A.3 Implémentation

Pour être fonctionnelle, la spécification ci-dessus doit être complétée par l'implémentation des compétences. Par exemple, la classe suivante implémente les compétences de la description de rôle *Seller* dans l'organisation *Selling* ; il s'agit d'une implémentation directe de la spécification Object-Z. La seule subtilité réside dans la manière de créer les objets représentant les compétences (dans le constructeur de la classe) et de les invoquer par la suite.

```
package mocaConflicts.skills;

import java.lang.reflect.Method;

import moca.instanciation.Acquaintance;
import moca.instanciation.Perturbation;
import moca.specific.MKAgent;
import moca.instanciation.RefusedSkillException;

public class SellingSellerPrivateSkill extends ExtendedSkill {

    int askedQuantity, unitPrice;
    Method putAmount, takeAmount, removeStock, getStock, ↩
       addMoneyInPurse;
    Acquaintance clientAc;

    public SellingSellerPrivateSkill() {

        askedQuantity = 0;
```

```
        unitPrice = 1;
        clientAc = null;

        // finding the skills...

        Class            skill;
        Class[]          parameters;

        try {

            skill          = Class.forName("mocaConflicts.skills.Purse" ↩
                );
            parameters     = new Class[1];
            parameters[0] = Class.forName("java.lang.Integer");
            putAmount     = skill.getMethod("putAmount", parameters);

            parameters     = new Class[1];
            parameters[0] = Class.forName("java.lang.Integer");
            takeAmount     = skill.getMethod("takeAmount", parameters);

            skill          = Class.forName("mocaConflicts.skills.Stock" ↩
                );
            parameters     = new Class[1];
            parameters[0] = Class.forName("java.lang.Integer");
            removeStock    = skill.getMethod("removeStock", parameters) ↩
                ;

            parameters     = new Class[0];
            getStock       = skill.getMethod("getStock", parameters);

            skill = this.getClass();
            parameters     = new Class[1];
            parameters[0] = Class.forName("moca.instanciation. ↩
                Perturbation");
            addMoneyInPurse = skill.getMethod("addMoneyInPurse", ↩
                parameters);

        }
        catch(ClassNotFoundException ex) {

            System.err.println("SellingSellerPrivateSkill::<init>␣:␣" ↩
                + ex);

        }
        catch(NoSuchMethodException ex) {

            System.err.println("SellingSellerPrivateSkill::<init>␣:␣" ↩
                + ex);

        }

    }
```

```
    public void registerAsk(Perturbation p) {

        askedQuantity = ((Integer)p.getAttributeValue("Quantity")). ↩
            intValue();
        clientAc = p.getAcquaintance().getOtherSide();

    }

    public Boolean enoughStock() throws ClassNotFoundException {

        boolean result;
        int stock;

        try {
            stock = ((Integer)owner.invokeSkill(owner, getStock, null) ↩
                ).intValue();
            result = (askedQuantity <= stock);
        }
        catch (RefusedSkillException e) {
            result = false;
        }

        return new Boolean(result);

    }

    public Boolean notEnoughStock() throws ClassNotFoundException {

        return new Boolean(!(enoughStock().booleanValue()));

    }

    public void sendGoodNotAvailable() {

        Perturbation     perturbation;

        perturbation     = mocaInstanciation.getNewPerturbation(" ↩
            GoodNotAvailable");
        perturbation.setAcquaintance(clientAc);
        sendPerturbation(perturbation);

        clientAc = null;

    }

    public void sendGoodAndPrice() throws RefusedSkillException, ↩
        ClassNotFoundException {

        Perturbation     perturbation;

        perturbation     = mocaInstanciation.getNewPerturbation(" ↩
            GoodAndPrice");
        perturbation.setAttributeValue("GoodAmount", new Integer( ↩
```

```
            askedQuantity));
        perturbation.setAttributeValue("Price", new Integer(unitPrice* ↩
            askedQuantity));
        perturbation.setAcquaintance(clientAc);
        sendPerturbation(perturbation);

        Object[] params = new Object[1];
        params[0] = new Integer(askedQuantity);

        try {
            owner.invokeSkill(owner, removeStock, params);
        }
        catch (RefusedSkillException e) {
            throw new RefusedSkillException(addMoneyInPurse,e);
        }

    }


    public void addMoneyInPurse(Perturbation p) throws ↩
        RefusedSkillException, ClassNotFoundException {

        int amount = ((Integer)p.getAttributeValue("Amount")).intValue ↩
            ();
        Object[] params = new Object[1];
        params[0] = new Integer(amount);

        try {
            owner.invokeSkill(owner, putAmount, params);
        }
        catch (RefusedSkillException e) {
            throw new RefusedSkillException(addMoneyInPurse,e);
        }

        clientAc = null;

    }

}
```

Pour illustrer la manière d'implémenter une politique d'acceptation, nous prendrons la compétence *Purse* ; le code suivant correspond à une acceptation systématique (sauf si l'avoir a passé en dessous de zéro) :

```
package mocaConflicts.skills;

import java.lang.reflect.Method;
import java.util.LinkedHashSet;
import java.util.Iterator;

import moca.managinggroup.DefaultSkill;
import java.lang.ClassCastException;
import moca.instanciation.OrganisationalAgent;
```

```
public class Purse extends ExtendedSkill {

    protected int amount;
    protected LinkedHashSet toFree;
    protected boolean bankrupt;

    public Purse(int a) {

        amount = a;
        toFree = new LinkedHashSet();
        if (a < 0) bankrupt = true; else bankrupt = false;

    }

    public Integer getAmount() {

        return new Integer(amount);

    }

    public void takeAmount(Integer val) {

        amount = amount-val.intValue();
        trace();

    }

    public void putAmount(Integer val) {

        amount = amount+val.intValue();
        trace();

    }

    private void trace() {

        if (amount < 0) {
            agentPrint("BANKRUPTCY !");
            bankrupt = true;
            return;
        }

        String s ="";
        for (int i = 0; i < amount ; i++) s += "#";
        agentPrint(s);

    }

    public boolean require(Object requester, Method theMethod, Object ↩
        [] theParams) {

        if (bankrupt) return false;
```

```
        else return true;

    }

}
```

On peut ensuite affiner la politique d'acceptation pour introduire par exemple un système de réservation :

```
package mocaConflicts.skills;

import java.lang.reflect.Method;
import java.util.LinkedHashSet;
import java.util.Iterator;

import moca.managinggroup.DefaultSkill;
import java.lang.ClassCastException;
import moca.instanciation.OrganisationalAgent;
import moca.instanciation.Role;

public class PurseSimpleReservation extends Purse {

    private Object reservedFor;

    public PurseSimpleReservation(int a) {

        super(a);
        reservedFor = null;

    }

    public boolean require(Object requester, Method theMethod, Object ↩
        [] theParams) {

        boolean result;

        result = super.require(requester, theMethod, theParams);
        if (!result) return result;

        if (reservedFor == null) {
            reservedFor = requester;
            result = true;
        }
        else result = (requester == reservedFor);

        if (!result) toFree.add(theMethod); //IMPORTANT!!

        return result;

    }


    public void release(Object requester, Method theMethod) {
```

```
        if (reservedFor != requester) return;

        if (!theMethod.getName().equals("takeAmount")) return;

        reservedFor = null;
        freeMethods();

    }

    private void freeMethods() {

        Iterator freeIt = toFree.iterator();
        while (freeIt.hasNext())
            notifyFree((Method)freeIt.next());

        toFree.clear();

    }

}
```

Enfin, le code suivant permet à l'agent *Intermédiaire* de demander son entrée en tant que *Buyer* dans chacun des groupes instanciant l'organisation *Supply* :

```
package mocaConflicts.agents;

import java.util.List;
import java.util.Iterator;

import moca.instanciation.Role;
import moca.instanciation.OrganisationIdentity;
import moca.managinggroup.DefaultSkill;
import moca.managinggroup.WantAnswerAskRole;
import moca.managinggroup.WantAnswerGetAllOrganisations;
import moca.managinggroup.WantAnswerAskAgents;
import moca.instanciation.Acquaintance;
import moca.instanciation.AgentReference;
import moca.managinggroup.WantAnswerAskAcquaintance;

public class MiddleAgentSkill extends DefaultSkill
    implements WantAnswerAskRole, WantAnswerGetAllOrganisations,
WantAnswerAskAgents, WantAnswerAskAcquaintance {

    public MiddleAgentSkill() {
    }

    public void init() {

        getAllOrganisations(this, null);

    }
```

```
public boolean answerGetAllOrganisations(List answer, Object ↩
    reference) {

    boolean                 result = false;

    if ((answer != null) && (answer.size() > 0)) {
        OrganisationIdentity oi;
        Iterator orgIt = answer.iterator();
        while (orgIt.hasNext()) {
            oi = (OrganisationIdentity)orgIt.next();
            if (oi.getOrganisationScheme().getName().equals(" ↩
                Supply")) {
                askRole(oi, "Buyer", this, oi);
                result = true;
            }
        }
    }

    return result;

}

public boolean answerAskRole(Boolean answer, Role role, Object ↩
    reference) {

    askAgents(role.getOrganisationIdentity(), "Seller", this, role ↩
        );
    return true;

}

public boolean answerAskAgents(Boolean answer,
                               OrganisationIdentity oi,
                               String rd,
                               List agents,
                               Object reference) {

    boolean                 result;
    Acquaintance            ac;


    ac = mocaInstanciation.getNewAcquaintance("Buyer", rd, oi,
            owner.getAgentReference(),
            (AgentReference)(agents.get(0)));
    result          = askAcquaintance(ac, this, reference);

    return result;

}

public boolean answerAskAcquaintance(Boolean answer, Acquaintance ↩
    ac, Object reference) {
```

```
        boolean                 result;
        Role                    role;

        role            = (Role)reference;
        role.setOwner(owner);
        result          = addRoleToAgent(owner.getAgentReference(),
            role);
        result          = addAcquaintance(ac.getOtherSide()) && result
            ;

        return result;

    }

}
```

Annexe B

Diagrammes de classes

Cette annexe propose un bref survol de la structure des principales classes qui composent la plate-forme MOCA. Pour plus de détails, on se référera à [Báe02] ou directement au code, disponible sur la page internet de MadKit [Mad].

B.1 Niveau descriptif

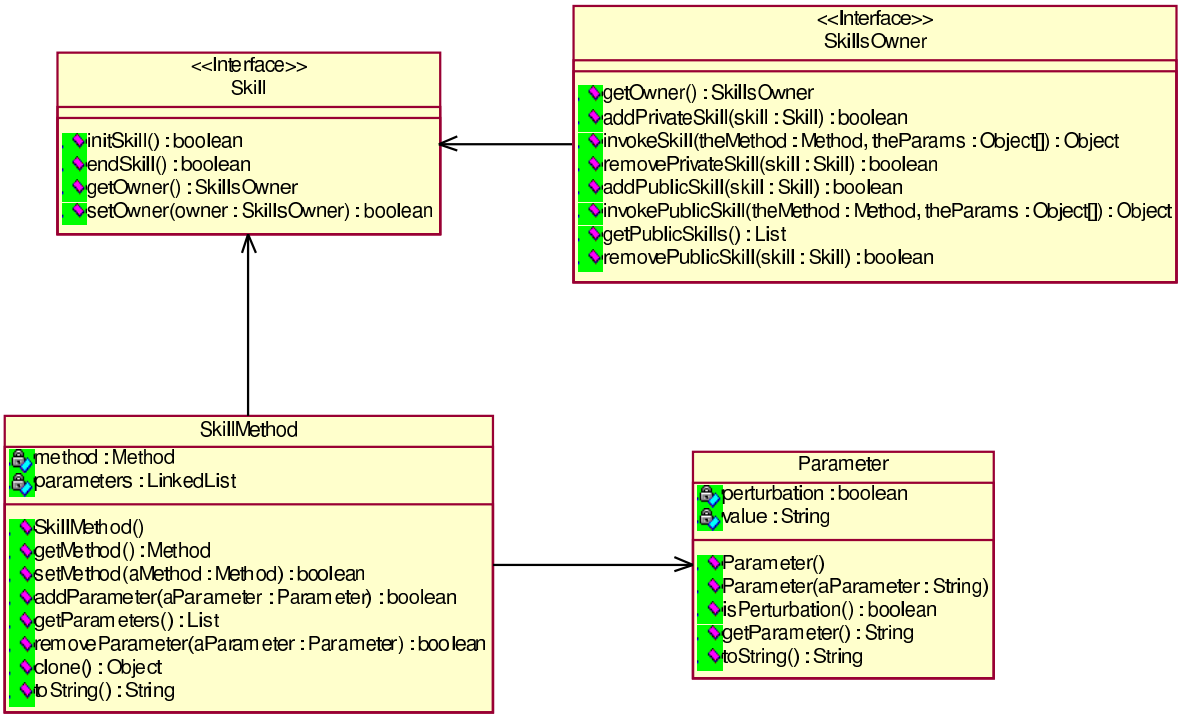


Fig. B.1 – Descriptions de compétences

La figure B.2 montre la structure des organisations, descriptions de rôles et relations. On remarquera la présence au sein du système d'une bibliothèque d'organisations, recensant toutes les organisations disponibles. Elle est utilisée par le rôle *Pages Jaunes* du *Groupe de*

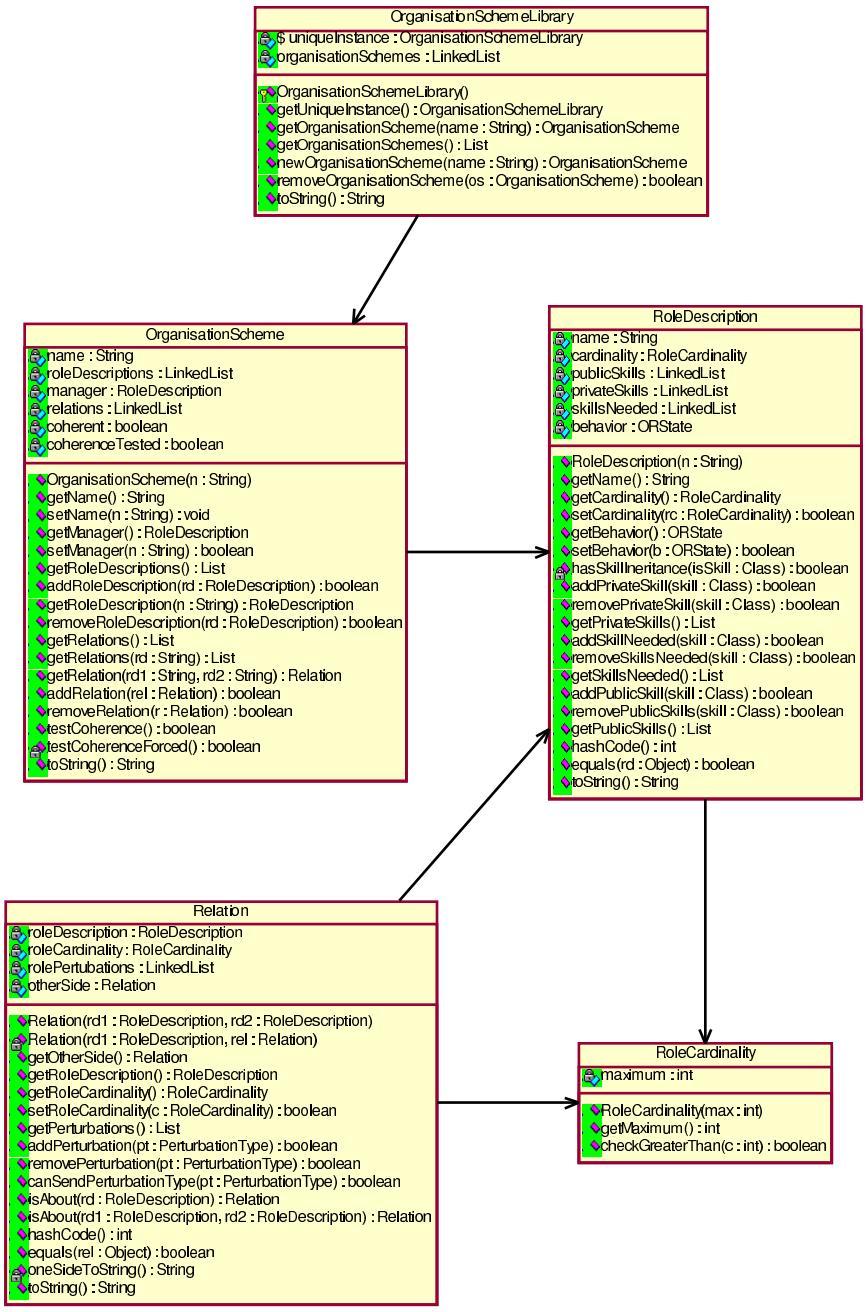


FIG. B.2 – Organisations

Gestion.

La figure B.1 montre la structure des descriptions de compétences (*Skill*) et des classes associées. Les compétences sont représentées par des méthodes, et les appels de compétences deviennent des invocations de méthodes, réalisées à l'aide de la couche réflexive de Java (package *java.lang.reflect*).

B.2 Niveau exécutif

Un agent MOCA doit réaliser l'interface *OrganisationalAgent* dont la structure est représentée à la figure B.3. On peut voir sur cette même figure la structure des classes *Perturbation* et *Acquaintance*, qui instancient conceptuellement les *types de perturbations* et *relations* du niveau descriptif.

Les principales classes liées à l'exécution d'un rôle sont représentées à la figure B.4. On remarquera les différentes méthodes permettant d'ajouter, de retirer, ou d'appeler des compétences, privées ou publiques, ainsi que la présence d'un interpréteur de statecharts.

B.3 Ancrage dans MadKit

La figure B.5 représente les principales classes qui permettent d'ancrer MOCA dans la plate-forme multi-agents MadKit. La classe *MKAgent*, héritant à la fois de l'interface *OrganisationalAgent* et de la classe *Agent* de MadKit, constitue bien sûr le pilier central de cet ancrage. Les classes *MKAgentReference* et *MKPerturbation* bénéficient également d'un double héritage MOCA et MadKit.

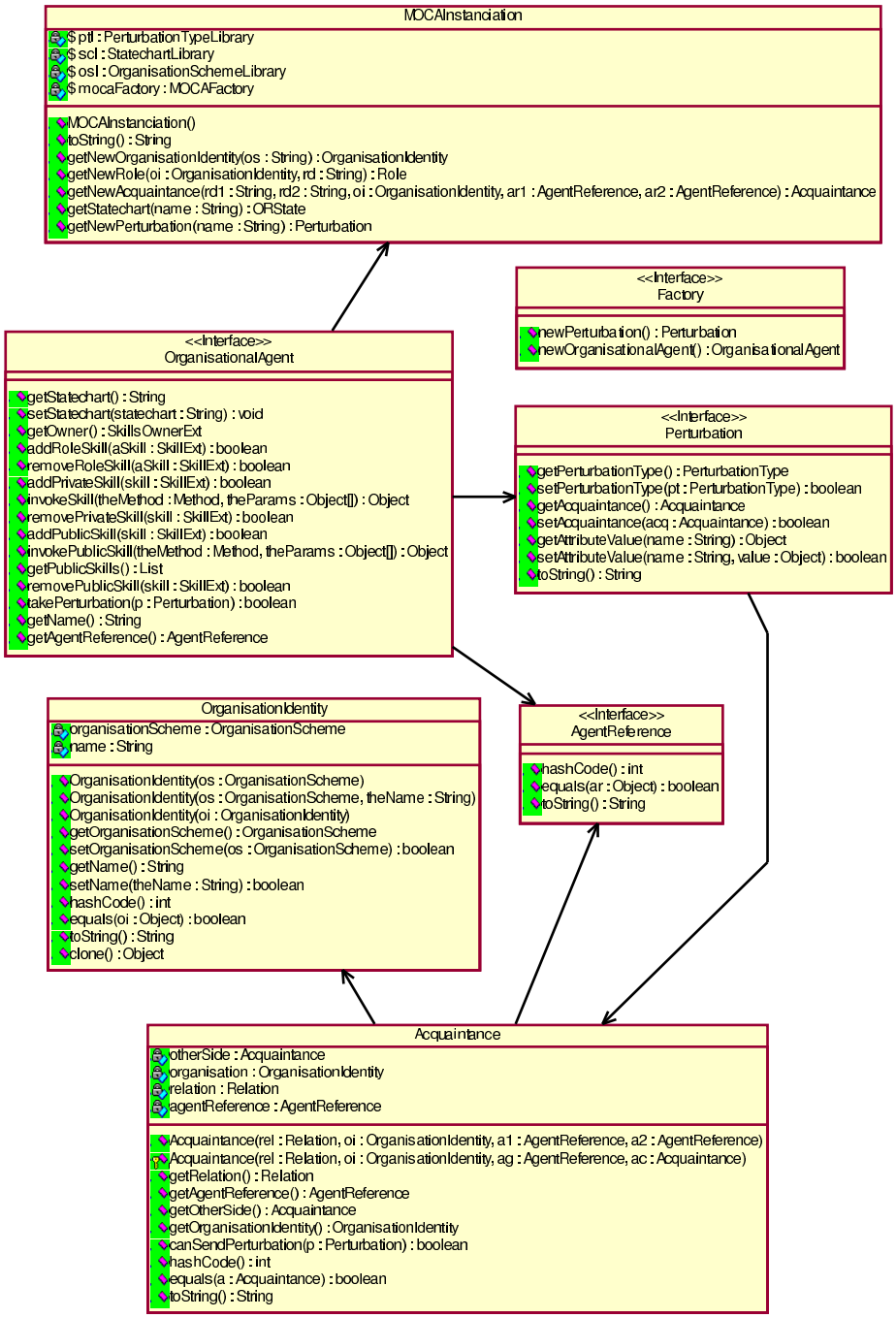


FIG. B.3 – Agents et accointances

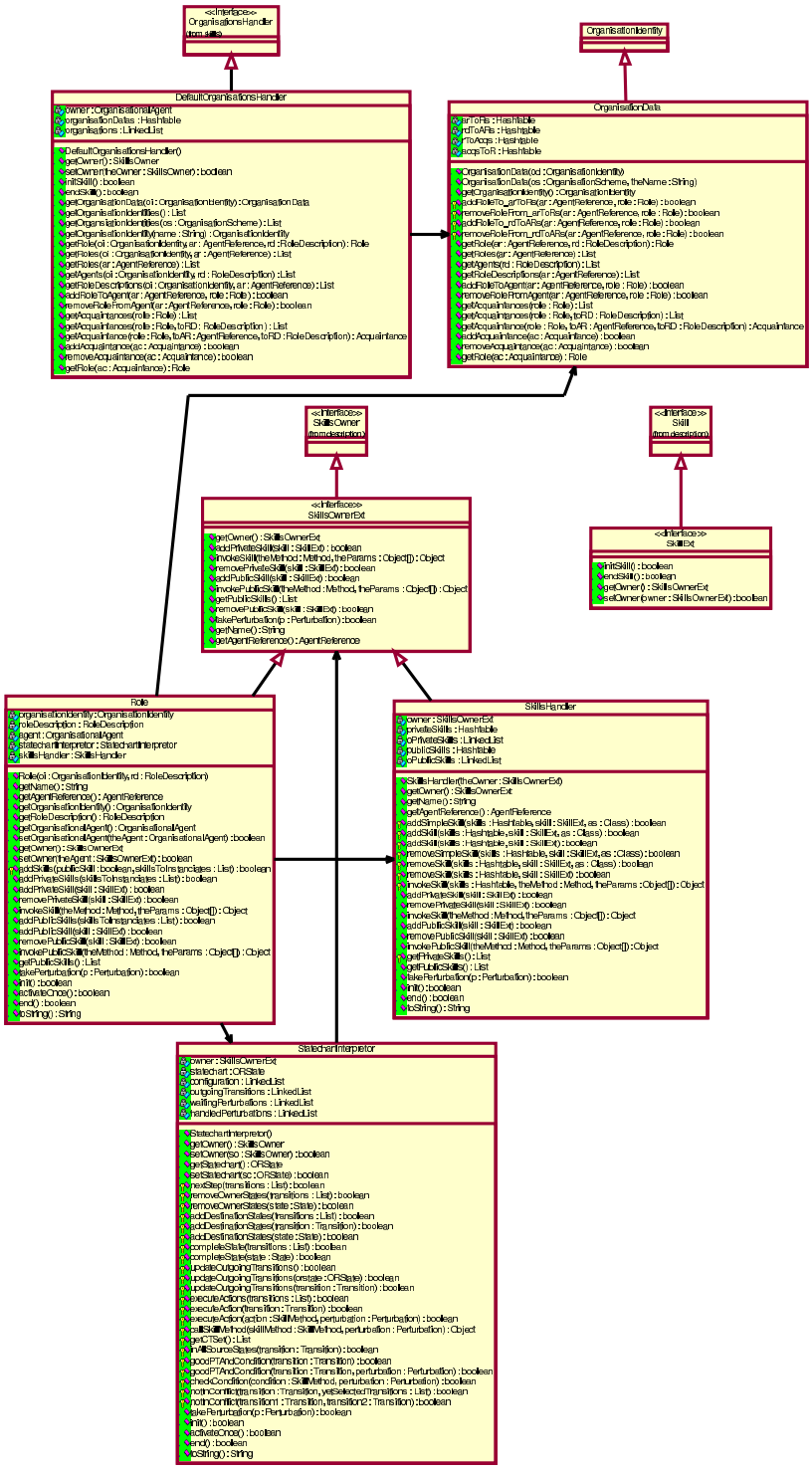


FIG. B.4 – Exécution des rôles

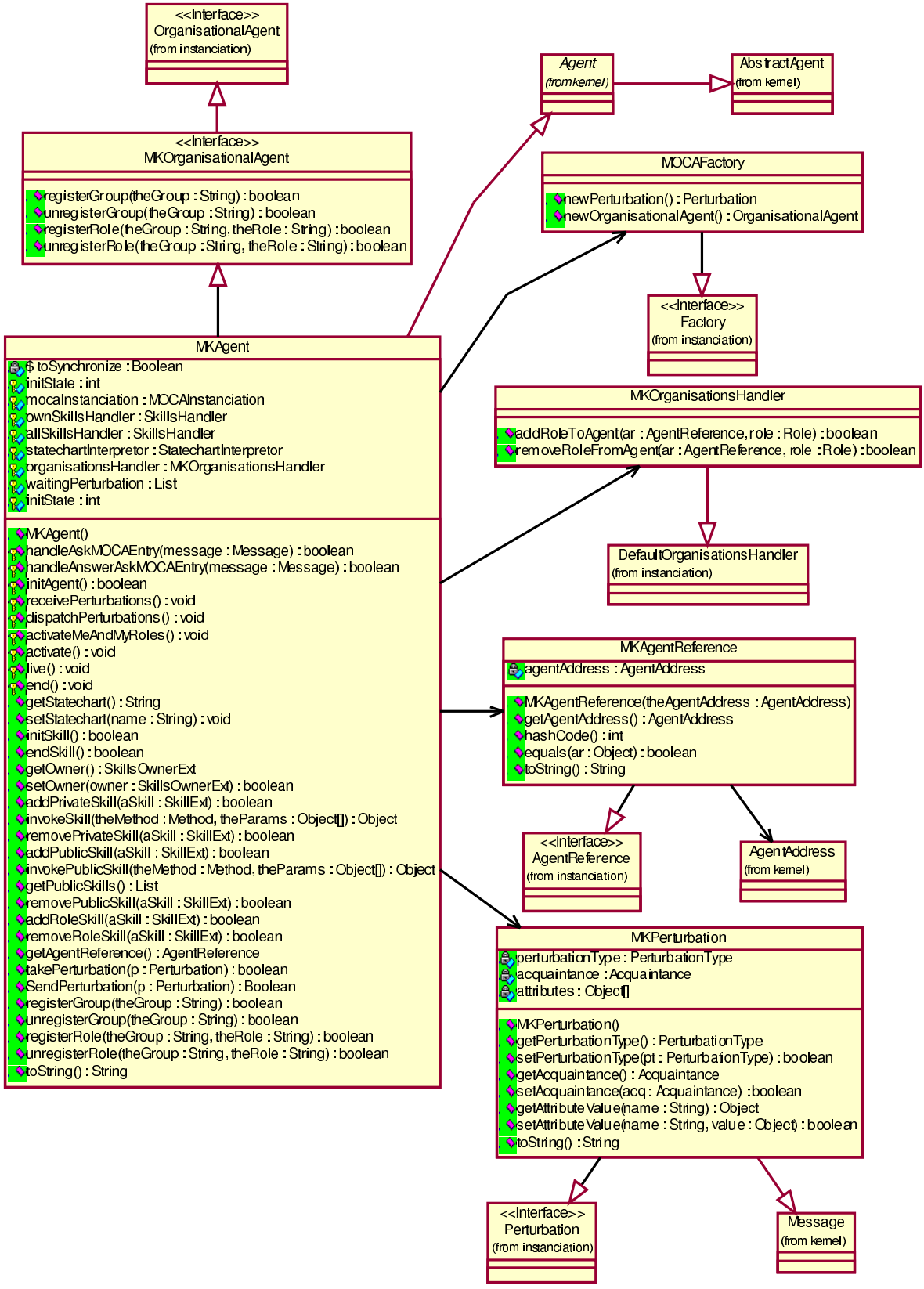


FIG. B.5 – Ancrage dans MadKit

Bibliographie

[Aek99] F. V. Aeken. *Les systèmes multi-agents minimaux*. Thèse de doctorat, Institut National Polytechnique de Grenoble, 1999.

[AHU87] A. V. Aho, J. E. Hopcroft et J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987.

[AMBN] M. Amiguet, J. Müller, J. Báez-Barranco et A. Nagy. The MOCA Platform : Simulating the Dynamics of Social Networks. Dans J. Sichman (éditeur), *MABS'02*. Springer-Verlag. À paraître.

[Ami98] M. Amiguet. Introduction à la théorie des catégories, 1998. Travail de diplôme de l'Université de Neuchâtel.

[And93] L. Andrey. *Protocoles de communication et grammaires attribuées*. Thèse de doctorat, Université de Nancy I, 1993.

[BDJT97] F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings et J. Treur. DESIRE : Modelling Multi-Agent Systems in a Compositional Formal Framework. *Int. Journal of Cooperative Information Systems*, 6(1) :pages 67–94, 1997.

[Báe02] J. Báez. Extension et consolidation de la plate-forme organisationnelle MOCA. Travail de diplôme de l'Université de Neuchâtel, 2002.

[BGM98] M. Barbuceanu, T. Gray et S. Mankovski. Coordinating with Obligations. Dans *Agents'98*, pages 62–69. Minneapolis, mai 1998.

[BJT97] F. M. T. Brazier, C. M. Jonker et J. Treur. Formalization of a cooperation model based on joint intentions. Dans J. P. Müller, M. J. Wooldridge et N. R. Jennings (éditeurs), *Intelligent Agents III (ATAL'96)*, pages 141–155. 1997.

[BMM99] A. Bondavalli, I. Majzik et I. Mura. Automated dependability Analysis of UML designs. Dans *ISORC'99*. IEEE Computer Society, 1999.

[Bra83a] G. Brams. *Réseaux de Petri, théorie et pratique, tome 1*. Masson, 1983.

[Bra83b] G. Brams. *Réseaux de Petri, théorie et pratique, tome 2*. Masson, 1983.

[Bur93] H.-D. Burkhard. Liveness and Fairness Properties in Multi-Agent Systems. Dans *IJCAI'93*. 1993.

[CEMS01] R. Conte, B. Edmonds, S. Moss et K. Sawyer. Sociology and Social Theory in Agent Based Social Simulation : A Symposium. *Computational and Mathematical Organization*, 7 :pages 183–205, 2001.

[Cha90] N. P. Chapman. Defining, Analysing and Implementing Communication Protocols Using Attribute Grammars. *Formal Aspects of Computing*, 2 :pages 359–392, 1990.

[Cor01] K. C. Correa e Silva Fernandes. *Systèmes Multi-Agents Hybrides : Une Approche pour la Conception de Systèmes Complexes*. Thèse de doctorat, Université Joseph Fourier – Genoble 1, 2001.

[CPD96] A. Collinot, L. Ploix et A. Drogoul. Application de la méthode Cassiopée à l'organisation d'une équipe de robots. Dans J.-P. Müller et J. Quinqueton (éditeurs), *JFIADSMA'96*, pages 137–152. Hermes, 1996.

[CSS01] J. Cardoso, C. Sibertin-Blanc et C. Soulé-Dupuy. Une sémantique formelle des diagrammes d'interaction d'UML via les réseaux de Petri. Dans G. Juanolle et R. Valette (éditeurs), *Actes du Colloque Francophone sur la Modélisation des Systèmes Réactifs MSR'01*, pages 497–512. Hermes, 2001.

[DES] The DESIRE Research Programme. http ://www.cs.vu.nl/vakgroepen/ai/projects/desire.

[DRG02] R. Duboz, E. Ramat et N. Giambiasi. Utilisation du Formalisme DEVS pour la Spécification de Systèmes d'Agents Réactifs. Dans J. Müller (éditeur), *JFIAD-SMA'02*. Hermes, 2002.

[DRS94] R. Duke, G. Rose et G. Smith. *Object-Z : a Specification Language Advocated for the Description of Standards*. Rapport technique 94–95, Software Verification Research Centre, Department of Computer Science, University of Queensland, décembre 1994.

[Dur83] D. Durand. *La systémique*. Numéro 1795 dans Que sais-je ? Presses Universitaires de France, 1983.

[Dur96] B. Durand. *Simulation multi-agents et épidémiologie opérationnelle*. Thèse de doctorat, Université de Caen, 1996.

[Dur00] A. Dury. *Modélisation des interactions dans les systèmes multi-agents*. Thèse de doctorat, Université Henri Poincaré, 2000.

[EH96] A. El Fallah Segrouchni et S. Haddad. A Recursive Model for Distributed Planning. Dans *ICMAS'96*, pages 307–314. 1996.

[EJT02] J. Engelfriet, C. Jonker et J. Treur. Compositional Verification of Multi-Agent Systems in Temporal Multi-Epistemic Logic. *Journal of Logic, Language and Information*, 11 :pages 195–225, 2002.

[EW00] R. Eshuis et R. Wieringa. Requirements-Level Semantics for UML Statecharts. Dans S. Smith et C. Talcott (éditeurs), *Formal Methods for Open Object-Based Distributed Systems IV*, pages 121–140. Kluwer Academic Publishers, 2000.

[Fer99] J. Ferber. *Multi-Agent Systems*. Addison-Wesley, 1999.

[FG98] J. Ferber et O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. Dans *ICMAS'98*, pages 128–135. IEEE Computer Society, 1998.

[FG99] J. Ferber et O. Gutknecht. Operational Semantics of a Role-based Agent Architecture. Dans *ATAL'99*. 1999.

[FGJ+] J. Ferber, O. Gutknecht, C. M. Jonker, J.-P. Müller et J. Treur. Organization Models and Behavioural Requirements Specification for Multi-Agent Systems. Dans Y. Demazeau et F. Garijo (éditeurs), *MAAMAW'01*, Lecture Notes in AI. Springer-Verlag. À paraître.

[FGJ+00] J. Ferber, O. Gutknecht, C. M. Jonker, J.-P. Müller et J. Treur. Organization Models and Behavioural Requirements Specification for Multi-Agent Systems. Dans *ICMAS 2000*. IEEE Computer Society Press, 2000.

[FH01] M. Frappier et H. Habrias (éditeurs). *Software Specification Methods*. Springer-Verlag, 2001.

[FIP] The Foundation for Intelligent Physical Agents (FIPA). http ://www.fipa.org.

[FM96] J. Ferber et J.-P. Müller. Influence and reaction : a model of situated multi-agent system. Dans *ICMAS'96*. AAAI Press, Kyoto, 1996.

[FT00] J. L. Fernández et A. Toval. Can Intuition Become Rigorous ? Foundations for UML Model Verification Tools. Dans F. M. Titsworth (éditeur), *International Symposium on Software Reliability Engineering*, pages 344–355. IEEE Press, 2000.

[GH+99] A. Guillemet, G. Haïk *et al.* Mise en œuvre d'une approche componentielle pour la conception d'agents. Dans M.-P. Gleizes et P. Marcenac (éditeurs), *Ingénierie des systèmes multi-agents (JFIADSMA'99)*, pages 53–65. 1999.

[Gut01] O. Gutknecht. *Proposition d'un modèle organisationnel générique de systèmes multi-agents et examen de ses conséquences formelles, implémentatoires et méthodologiques*. Thèse de doctorat, Université des Sciences et Techniques du Languedoc, 2001.

[Haa86] O. Haas. Formal Protocol Specification based on Attribute Grammars. Dans M. Diaz (éditeur), *Proceedings of the IFIP WG 6.1 fifth international workshop*, pages 39–48. 1986.

[Ham01] N. Hameurlain. Composition et substitution d'agents : sémantique et préservation de propriétés. Dans A. El Hallah Seghrouchni et L. Magnin (éditeurs), *Fondements des systèmes multi-agents (JFIADSMA'01)*, pages 135–147. Hermes, 2001.

[Har87] D. Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8 :pages 231–274, 1987.

[Hil00] V. Hilaire. *Vers une approche de spécification, de prototypage et de vérification de systèmes multi-agents*. Thèse de doctorat, Université de Franche-Comté, 2000.

[HL98] B. Horling et V. Lesser. *A Reusable Component Architecture for Agent Construction*. Rapport technique UM-CS-1998-049, UMass Computer Science Dept, University of Massachusetts, 1998.

[HM00] G. Huszerl et I. Majzik. Quantitative Analysis of dependability Critical Systems Based on UML Stetcharts Models. Dans *HASE 2000*, pages 83–92. novembre 2000.

[HN96] D. Harel et A. Naamad. The Statemate Semantics of Statecharts. *ACM Trans. Soft. Eng. Method.*, 4(5), octobre 1996.

[HSB02a] J. F. Hübner, J. S. Sichman et O. Boissier. MOISE+ tutorial. http ://www.inf.furb.br/ jomi/, 2002.

[HSB02b] J. F. Hübner, J. S. Sichman et O. Boissier. Spécification structurelle, fonctionnelle et déontique d'organisations dans les SMA. Dans J.-P. Müller (éditeur), *JFIADSMA'02*. Hermes, 2002.

[Hug01] M.-P. Huget. *Une ingénierie des protocoles d'interaction pour les systèmes multi-agents*. Thèse de doctorat, Université Paris IX - Dauphine, 2001.

[Jea97] M. Jean (nom collectif). Émergence et SMA. Dans *Journées Francophones IAD et SMA*. Nice, 1997.

[Kan96] K. C. Kang. Formalization and Verification of Safety Properties of Statechart Specifications. Dans *Proceedings of Asia-Pacific Software Engineering Conference*, pages 16–27. 1996.

[Ken99] E. Kendall. Role modelling for agents analysis, design and implementation. Dans *1st ASA - 3rd MA*. Palm Springs, 1999.

[Lev99] F. Levi. Compositional Verification of Timed Statecharts. Dans H. Barringer, M. Fisher, D. M. Gabbay et G. Gough (éditeurs), *Advances in Temporal Logic*, Applied Logic Series, pages 47–70. Kluwer Academic Press, 1999.

[Lhu98] M. Lhuillier. *Une approche à base de composants logiciels pour la conception d'agents. Principes et mise en œuvre à travers la plate-forme Maleva.* Thèse de doctorat, Université de Paris VI, 1998.

[LMM99] D. Latella, I. Majzik et M. Massink. Towards a formal operational semantics of UML statechart diagrams. Dans P. Ciancarini et R. Gorrieri (éditeurs), *Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, pages 331–347. 1999.

[MABN01] J.-P. Müller, M. Amiguet, J. Baez et A. Nagy. La plate-forme MOCA : réification de la notion d'organisation au-dessus de MadKit. Dans A. El Fallah Segrouchni et L. Magnin (éditeurs), *JFIADSMA'01*, pages 307–310. 2001.

[Mad] The MadKit Project (a Multi-Agent Development Kit). http ://www.madkit.org.

[Maz01] H. Mazouzi. *Ingénierie des protocoles d'interaction : des Systèmes Distribués aux Systèmes Multi-agents.* Thèse de doctorat, Université Paris Dauphine, 2001.

[MB01] T. Meurisse et J.-P. Briot. Une approche à base de composants pour la conception d'agents. *Technique et science informatique*, 20(4) :pages 538–602, 2001.

[MEH02] H. Mazouzi, A. El Fallah Seghrouchni et S. Haddad. Open Protocol Design for Complex Interactions in Multi-Agent Systems. Dans *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS 02)*. Bologna, 2002.

[PNJ99] P. Panzarasa, T. J. Norman et N. R. Jennings. Modeling Sociality in the BDI Framework. Dans *Proceedings of First Asia-Pacific Conference on Intelligent Agent Technology (IAT'99)*, pages 202–206. 1999.

[PO01] H. Parunak et J. Odell. Representing social structures in UML. Dans *Workshop Agent-Oriented Software Engineering, AOSE2001.* 2001.

[PS98] J. Philipps et P. Scholz. Formal Verification and Hardware Design with Statecharts. Dans B. Möller et J. V. Tucker (éditeurs), *Prospects for hardware foundations*, numéro 1546 dans Lecture notes in computer science, pages 356–389. Springer-Verlag, 1998.

[PS99] R. Pfeifer et C. Scheier. *Understanding Intelligence.* MIT Press, 1999.

[Ric01] P.-M. Ricordel. *Programmation Orientée Multi-Agents : Développement et Déploiement de Systèmes Multi-Agents Voyelles.* Thèse de doctorat, Institut National Polytechnique de Grenoble, 2001.

[Ros92] G. Rose. Object-Z. Dans S. Stepney, R. Barden et D. Cooper (éditeurs), *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer-Verlag, 1992.

[Sau] S. Sauvage. *Conception de systèmes Multi-agents : un thésaurus de motifs orientés agent.* Thèse de doctorat, Université de Caen et Université de Neuchâtel. En préparation.

[Sou01] J.-C. Soulié. *Vers une approche multi-environnements pour les agents.* Thèse de doctorat, Université de la Réunion, 2001.

[SP88] A. Silberschatz et J. L. Peterson. *Operating System Concepts, alternate edition.* Addison-Wesley Publishing Company, 1988.

[SS00] J. A. Saldhana et S. M. Shatz. UML Diagrams to Object Petri Net Model : An Approach for Modeling and Analysis. http ://citeseer.nj.nec.com/296679.html, 2000.

[US94] A. C. Uselton et S. A. Smolka. A Compositional Semantics for Statecharts using Labeled Transition Systems. Dans *International Conference on Concurrency Theory*, pages 2–17. 1994.

[VHL01] R. Vincent, B. Horling et V. Lesser. An Agent Infrastructure to Build and Evaluate Multi-agent Systems : The Java Agent Framework and Multi-agent System Simulator. *Lecture Notes in Computer Science*, 1887, 2001.

[WFHP02] M. Wooldridge, M. Fisher, M. Huget et S. Parsons. Model Checking Multi-Agent Systems with MABLE. Dans *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS 02).* Bologna, 2002.

[WJK00] M. Wooldridge, N. R. Jennings et D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3 :pages 285–312, 2000.

[YB01] M.-J. Yoo et J.-P. Briot. *Une approche componentielle pour la modélisation d'agents mobiles coopérants.* Rapport technique 2001/013, LIP6, 2001.

[Yoo99] M.-J. Yoo. *Une approche componentielle pour la modélisation des agents coopérants et sa validation.* Thèse de doctorat, Université de Paris 6, 1999.