

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Présentée par : Caroline CHOPINAUD

Pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité : Informatique

Contrôle de l'émergence de comportements dans les systèmes d'agents cognitifs autonomes

Soutenue le : Lundi 9 Juillet 2007

Devant le jury composé de :

Rapporteurs :	Catherine Tessier	Ingénieur de Recherche ONERA
	Olivier Boissier	Professeur Ecole de Mines de Saint Etienne
Examineurs :	Philippe Mathieu	Professeur Université de Lille I
	Nicolas Maudet	Maître de Conférence Université Paris Dauphine
	David Sadek	Directeur de Recherche France Télécom R&D
	Olivier Sigaud	Professeur Université Pierre et Marie Curie
Directrice de thèse :	Amal El Fallah Seghrouchni	Professeur Université Pierre et Marie Curie.
Co-Encadrant :	Patrick Taillibert	Ingénieur de Recherche Expert Thales Aerospace

Thèse sous contrat CIFRE

Table des matières

Introduction	1
I Présentation générale	5
1 Par où tout commence...	7
Avant-Propos	7
1.1 Les systèmes critiques	8
1.1.1 Les systèmes temps-réels	8
1.1.2 Les systèmes embarqués	9
1.2 Les systèmes multiagents	9
1.2.1 Un (tout petit) peu d'histoire...	9
1.2.2 Les agents	10
Quelques définitions	10
Les types d'agents	11
1.2.3 Principes des systèmes multiagents	12
Définitions	12
Interaction et communication	14
Environnement	16
Emergence	16
Autonomie	17
Conclusion	17
2 Du problème de définir l'autonomie	19
Avant-Propos	19
2.1 Présentation générale	20
2.2 Différentes visions de l'autonomie	21
2.2.1 Classification de l'autonomie	21
2.2.2 Autonomie et Performance	23

2.2.3	Autonomie et confiance	25
2.2.4	Autonomie et Organisation	28
2.2.5	Autonomie ajustable	30
2.2.6	Autonomie et Normes	31
2.2.7	Autonomie et Génération de but	32
2.2.8	Degré d'autonomie	34
2.3	Lever l'ambiguïté	35
2.3.1	Autonomie et prise de décision	35
2.3.2	Agent autonome vs agent réfractaire	36
2.3.3	Alors l'autonomie...	37
	Conclusion	38
II	S'assurer du fonctionnement d'un système d'agents	41
3	Vérification de systèmes	43
	Avant-Propos	43
3.1	Méthodes classiques de vérification de logiciels	44
3.1.1	La démonstration automatique	44
	Preuve de théorème appliquée aux systèmes multiagents	45
	Remarque	47
3.1.2	Le model-checking	47
	Model-checking et systèmes multiagents	49
	Remarque	51
3.1.3	Les tests	51
	Les tests dans les systèmes multiagents	52
	Remarque	53
3.2	La surveillance en-ligne	53
3.2.1	Le monitoring de système	53
	Monitoring des systèmes distribués	54
	Remarque	55
3.2.2	L'introspection	55
	Conclusion	57
4	Contrôle de l'émergence de comportements	59
	Avant-Propos	59
4.1	Présentation de notre approche	60
4.1.1	Le contrôle d'agents	61
4.2	Travaux similaires	63

4.2.1	Contrôle des interactions dans les systèmes multiagents ouverts . . .	63
	Law-Governed Interaction	63
	XML-Law	64
4.2.2	Gestion d'exceptions	66
4.2.3	Déviations par rapport aux spécifications	68
4.2.4	Analyse de système multiagent	70
	Conclusion	72
 III Vers la génération automatique d'agents autocontrôlés		73
5	La description du contrôle	75
	Avant-Propos	75
5.1	Préambule	76
	5.1.1 Indépendance du contrôle	76
	5.1.2 Séparation entre la description et le contrôle	76
	5.1.3 Des agents autocontrôlés	77
5.2	Une ontologie d'agents	78
	5.2.1 Les modèles et architectures d'agents	78
	5.2.2 L'ontologie de base	80
5.3	Un ensemble de lois	82
	5.3.1 Le concept de normes	82
	5.3.2 Le concept de loi	84
	Qu'est-ce qu'une loi?	84
	Différents types de lois	85
	Structure d'une loi	86
5.4	Une architecture introspective	87
	5.4.1 Composition de l'architecture	87
	5.4.2 Principes de l'architecture	88
	La technique de l'observateur comme structure générale	88
	Des réseaux de Petri pour représenter les lois	89
	5.4.3 Fonctionnement interne	91
5.5	Digression	93
	5.5.1 D'une vision <i>Asimovienne</i> vers une vision plus sociale	93
	5.5.2 Vers un appareil psychique	95
	Le principe de plaisir	96
	Le principe de réalité	96
	Le Ca, le Moi et le Surmoi	97
	Et notre méta-architecture?	98

Conclusion	99
6 La génération des agents autocontrôlés	101
Avant-Propos	101
6.1 Préambule	102
6.2 Un peu de travail manuel	102
6.2.1 La description de l'application	102
6.2.2 La description des lois	105
6.2.3 Le langage de lois	106
La logique déontique dynamique	106
La syntaxe du langage	111
6.2.4 Les stratégies de régulation	113
6.3 Un peu de travail automatique	114
6.3.1 Instrumentation du code des agents	114
6.3.2 Les règles de génération des réseaux de Petri	116
6.3.3 Détection des transgressions de lois	123
6.4 Spécificités des lois multiagents	124
6.4.1 La distribution du réseau de Petri	126
6.4.2 La collaboration des parties contrôle	126
6.4.3 La régulation du comportement des agents	129
Conclusion	130
IV Implémentation et mise en oeuvre	131
7 Le framework SCAAR	133
Avant-Propos	133
7.1 Choix d'implémentation	134
7.2 L'architecture du framework SCAAR	134
7.2.1 Le générateur d'agents autocontrôlés	137
7.2.2 La partie contrôle	141
7.2.3 L'annexe de contrôle	142
7.3 Un exemple simple	144
7.3.1 Définition des concepts et des lois	144
7.3.2 La génération	146
7.3.3 La transgression d'une loi	148
Conclusion	149

8 Applications et Conséquences	151
Avant-Propos	151
8.1 La plateforme ALBA version 1.0	152
8.1.1 Au commencement, il y avait ARES...	152
8.1.2 ... et puis vint ALBA version 1.0	153
Décentralisation	153
Généricité	154
Migration	154
8.1.3 Les primitives d'ALBA v1.0	155
8.2 Couplage de DynaCROM et de SCAAR	156
8.2.1 L'application DynaCROM	156
8.2.2 DynaCROM par l'exemple	158
8.2.3 SCAAR comme une solution d'application des normes	159
8.3 Le système multiagent InterloC	162
8.3.1 Les principes de l'application	162
8.3.2 L'agentification d'InterloC	162
8.3.3 L'implémentation d'InterloC	163
8.3.4 Des lois pour InterLoc	164
Conclusion	166
V Conclusion & Travaux futurs	167
Conclusion	169
Perspectives	170
A Publications	175
B Justification	177

Introduction

Je m'appelle Ana I.. On dit souvent de moi que je fais partie du clan des individus gentils et prévisibles, de ceux que tout le monde apprécie car ils inspirent totalement confiance. Mais je sais que j'ai, au fond, un petit côté qui m'évite toute fadeur. Il arrive parfois que ce petit côté épicé me joue des tours... il arrive qu'en société je sois incapable de me contrôler...

*Lucy Westenra,
The log book of Ana I.*

Les travaux de thèse que nous allons présenter dans ce mémoire ont été effectués, au départ, dans la perspective de simplifier la conception des systèmes critiques et d'améliorer leur robustesse. En effet, la conception d'applications temps-réel et de systèmes embarqués n'est pas chose aisée, il nous semblait dès lors intéressant d'utiliser les systèmes multiagents pour tenter d'atteindre nos objectifs. D'une part, la modularité qu'apporte les systèmes multiagents nous apparaît comme intéressante vis-à-vis de la conception de tels systèmes, et d'autre part, l'autonomie que l'on accorde généralement aux agents, laissait sous entendre à la fois, une aisance de conception mais aussi une certaine robustesse.

Néanmoins, si les systèmes multiagents semblaient apporter des avancées non négligeables dans la mise en place de telles applications critiques, il nous est également apparu que le paradigme agent et l'autonomie fort appréciable qui lui était associé ne semblait guère encourager son utilisation. En effet, dans les milieux industriels, en particulier, les systèmes multiagents sont vus comme des systèmes dont il est impossible de garantir le comportement, du fait de leur caractère distribué, mais également, du fait même que les agents soient autonomes. Ce terme d'autonomie, largement employé pour définir et justifier le comportement des agents d'un système, apparaît tel un problème, de part l'absence de maîtrise qu'il laisse sous entendre sur le comportement des agents.

Du fait de ces réticences, notre objectif premier s'est alors ramené à garantir le comportement d'un système multiagent, c'est-à-dire à assurer qu'il ne suivra pas de comportements pouvant entraîner l'échec de l'application, que ce soit de part l'émergence du comportement global du système, que de part l'autonomie des agents. Dans un premier temps, nous avons donc cherché à appréhender plus clairement le concept d'autonomie. Il nous fallait cerner au mieux ce que pouvait sous-entendre ce terme d'un point de vue général au niveau des agents mais aussi, tenter de nous approcher de ce que cela pouvait signifier en terme d'implémentation lorsque nous nous trouvons en présence d'agents supposés autonomes. Dans un second temps, nous avons cherché un moyen de pouvoir assurer une certaine confiance dans le comportement des agents, en particulier au niveau du contrôle des comportements émergents au cours de l'exécution du système.

Notre étude de l'autonomie nous a amené à considérer ce concept comme un moyen de rendre, contrairement dirons-nous, un système plus robuste. En effet, il s'avère que tenir compte de l'autonomie potentielle des agents qui nous entoure, lors de la conception d'un agent, force la considération de l'apparition probable de comportements tout à fait différents de ceux généralement attendus. De ce fait, les agents doivent être en mesure de traiter des événements imprévus, des réponses inadéquates ou même l'absence de réponse provenant des autres agents du système.

De nos recherches sur les moyens de pouvoir garantir que, d'un système multiagent, n'émergera pas de comportements incohérents, pouvant potentiellement entraîner l'échec de l'application, nous avons pu mettre en évidence les problèmes liés à l'utilisation des méthodes classiques de vérification de systèmes. Même s'il nous semble indispensable d'utiliser de telles approches pour élaguer autant que possible les erreurs pouvant apparaître dans une application, il est difficile de pouvoir s'assurer de l'absence de l'émergence de comportements indésirables une fois le système mis en condition réelle d'exécution. Il nous a semblé alors intéressant de nous attacher plus particulièrement à une surveillance en ligne du comportement du système pour vérifier le caractère indésirable des comportements émergents.

Pour ce faire, nous proposons donc une approche, que l'on peut qualifier de complémentaire aux techniques de vérification classiques, permettant la surveillance des comportements des agents d'un système. Partant du principe que les agents sont autonomes et dans la perspective de maintenir le caractère distribué d'un système multiagent, notre approche consiste à fournir aux agents les moyens de contrôler leur propre comportement. Un agent se retrouve alors en mesure de pouvoir surveiller son comportement, s'assurer qu'il ne générera pas de comportements indésirables et le cas échéant, il va pouvoir se réguler pour tenter de se soustraire à ces mauvais comportements.

Ce contrôle peut être effectué par les agents grâce à une méta-architecture que nous leur fournissons et qui va leur permettre, à la fois de récolter des informations sur leurs comportements, mais aussi de détecter l'apparition des comportements indésirables à l'aide de lois. Ces dernières définissent des comportements souhaités ou redoutés dans le système que les agents vont devoir respecter. La méta-architecture que nous proposons est automatiquement générée et mise en place dans les agents à partir des lois définies pour le système et du code de comportements des agents.

Dans le cadre de nos travaux de thèse nous proposons donc :

- * Une approche de génération automatique d'agents autocontrôlés, c'est-à-dire des agents ayant les moyens nécessaires pour surveiller leur propre comportement et détecter les transgressions de lois pouvant apparaître au cours de leur exécution.
- * Une méthodologie de mise en place des lois au sein d'un système et un ensemble d'outils pour assurer la description de telles lois ainsi que les mécanismes nécessaires à leur mise en place.
- * Un framework fournissant l'ensemble des outils nécessaires à la mise en place du contrôle, c'est-à-dire à la description des lois et à la génération des agents autocontrôlés.

Ce mémoire est organisé de la façon suivante :

Présentation Générale : Dans cette première partie nous présentons le contexte dans lequel nous nous plaçons, c'est-à-dire les systèmes critiques et les systèmes multi-agents comme outil de conception des systèmes critiques. (*Chapitre 1 : Par où tout commence...*). Ensuite, nous présentons l'étude que nous avons pu faire sur l'autonomie des agents. Dans ce chapitre (*Chapitre 2 : Du problème de définir l'autonomie*), nous proposons un état de l'art sur le concept d'autonomie dans le domaine des systèmes multiagents, puis nous tentons de le cerner dans le cadre de nos travaux de thèse.

S'assurer du fonctionnement d'un système d'agents : Cette partie propose un état de l'art des techniques de vérification classiques, telles que le model-checking, la démonstration automatique et les tests logiciels dans la perspective de mettre en exergue les problèmes subsistants à l'utilisation de ces techniques. Nous présentons également des techniques de vérification de l'exécution d'un système, effectuées dynamiquement, et qui nous ont inspirées pour nos travaux de thèse (*Chapitre 3 : Vérification de systèmes*). Ensuite, nous abordons les grandes lignes de notre approche, en particulier les contraintes que nous nous sommes posées au démarrage

de cette thèse pour mettre en place notre principe d'autocontrôle. Nous présentons également différentes approches existantes auxquelles nous pouvons rapprocher nos travaux (*Chapitre 4 : Contrôle de l'émergence de comportements*).

Vers la génération automatique d'agents autocontrôlés : Nous présentons, dans un premier temps, les principes de notre approche et les moyens proposés pour décrire le contrôle à appliquer aux agents. Nous décrivons notre intérêt pour l'utilisation de lois et d'une ontologie pour en permettre la description à haut niveau, ainsi que l'architecture introspective que nous fournissons aux agents (*Chapitre 5 : La description du contrôle*). Le second chapitre de cette partie se charge alors de présenter dans le détail la méthodologie et les mécanismes utilisés pour la génération automatique des agents autocontrôlés (*Chapitre 6 : La génération des agents autocontrôlés*).

Implémentation et mise en oeuvre : Cette dernière partie contient la description de l'implémentation de notre framework, SCAAR, permettant la génération des agents autocontrôlés. Nous décrivons la méthodologie ainsi que la génération, sur un exemple simple, pour concrétiser notre approche (*Chapitre 7 : Le framework SCAAR*). Enfin, nous présentons les applications que nous avons pu utiliser pour tester notre approche et notre framework. Nous y décrivons deux applications ainsi qu'une plateforme multiagent, ALBA, écrite en Prolog, dont la première version a été conçue dans le cadre de cette thèse (*Chapitre 8 : Applications et conséquences*).

Première partie

Présentation générale

Chapitre 1

Par où tout commence...

... Je vous entends déjà me dire que cela n'a pas tant d'importance, que je ne suis pas la première à perdre pied dans certaines situations. A priori, je ne peux qu'être d'accord avec vous. Néanmoins, je sais que la société dans laquelle j'évolue est suffisamment critique pour que cette perte de contrôle ait un impact néfaste sur moi et ceux qui m'entourent.

*Lucy Westenra,
The log book of Ana I.*

Avant-Propos

Nous nous plaçons dans le cadre des systèmes critiques, tels que les systèmes embarqués et les systèmes temps réels, pour lesquels nous souhaitons, à la fois simplifier la conception et améliorer la robustesse. Aussi nous sommes-nous intéressés aux systèmes multiagents pour mettre en place ce type d'application. Deux caractéristiques spécifiques aux agents nous semblaient intéressantes pour tenter d'atteindre nos objectifs : la modularité que proposent les systèmes multiagents, mais aussi l'autonomie que l'on accorde aux agents. Dans ce chapitre, nous allons introduire rapidement les systèmes critiques pour ensuite nous intéresser aux différentes propriétés et définitions associées aux systèmes multiagents.

1.1 Les systèmes critiques

De façon générale, on peut définir un système critique de la manière suivante :

Un système critique est un système pour lequel un mauvais fonctionnement peut entraîner des conséquences inacceptables

Ces “conséquences inacceptables” sont dépendantes du contexte et peuvent inclure des pertes humaines, des dommages sur l’environnement ou la perte d’informations sensibles. La détermination de la criticité d’un système peut être définie en suivant la norme DO-178B appliquée aux systèmes aéroportés. Cette norme propose 5 niveaux de criticité :

NIVEAU A : CATASTROPHIQUE Dans le cas où le comportement anormal d’un logiciel empêcherait la continuité du vol et son atterrissage, ou la perte de l’avion et/ou de ses occupants (par exemple l’échec d’une commande du moteur ou d’un logiciel de l’ordinateur de vol).

NIVEAU B : DANGEREUX/MAJEUR-SEVERE Dans le cas où le comportement anormal d’un logiciel causerait une forte réduction des marges de sûreté ou des dommages sérieux sur les occupants.

NIVEAU C : MAJEUR Dans le cas où le comportement anormal d’un logiciel aurait comme conséquence la réduction significative de la sûreté, le malaise aux occupants ou l’augmentation significative de la charge de travail de l’équipage (par exemple l’échec d’une liaison de transmission de données par radio).

NIVEAU D : MINEUR Dans le cas où le comportement anormal d’un logiciel ne réduit pas de manière significative la sûreté de l’avion et augmente faiblement la charge de travail de l’équipage (par exemple un changement de trajectoire de vol).

NIVEAU E : SANS EFFET Dans le cas où le comportement anormal d’un logiciel n’affecte pas les possibilités opérationnelles et n’a pas comme conséquence une augmentation de la charge de travail de l’équipage.

Des systèmes pouvant être définis comme critiques sont, par exemple, les systèmes embarqués et les systèmes temps-réels.

1.1.1 Les systèmes temps-réels

Un système temps-réel est un système pour lequel l’exactitude du comportement dépend non seulement des valeurs de ses sorties, mais aussi du temps mis pour les produire. Généralement un système temps réel exécute une collection de tâches soumises à la fois à

une *deadline* (un délai) et un *jitter* (une variation). Lorsqu'une tâche est activée, elle doit produire ses données de sorties avant une certaine *deadline* et avec une faible variation (*jitter*) d'une activation à une autre.

On distingue généralement deux types de systèmes temps-réels, ceux que l'on peut qualifier de *hard real-time systems* et ceux de *soft real-time systems*. Dans le premier cas, il s'agit de systèmes pour lesquels le non-respect d'une *deadline* est potentiellement catastrophique. Dans le second cas, un résultat fournit au delà de la *deadline* garde néanmoins de son intérêt. On parle également de systèmes soumis à des contraintes temps réels fortes ou faibles.

1.1.2 Les systèmes embarqués

Les systèmes informatiques embarqués (ou enfouis) sont des systèmes et automatismes industriels, souvent de grande envergure (comme les centrales nucléaires, les systèmes de contrôle aérien, les serveurs de télécommunication), d'une ingénierie complexe (et faisant appel à de multiples compétences), souvent soumis à des contraintes temps-réel fortes (les avions, les automobiles), mais aussi des composants plus discrets comme les processeurs ou contrôleurs divers utilisés dans des produits de grande diffusion (téléphones portables, machines à laver, *etc*).

La caractéristique commune à tous les systèmes embarqués est de fonctionner en permanence et en constante interaction avec l'environnement. Le système capte et analyse un flux d'information continu provenant de l'environnement et décide, quasi-simultanément, de commander une réponse appropriée à ce flux.

Que l'on soit en présence de systèmes temps-réel ou de systèmes embarqués, toute défaillance dans l'exécution du système s'avère critique. C'est ce qui nous pousse à chercher un moyen de concevoir de façon plus sûre ce type de systèmes, en particulier à l'aide des systèmes multiagents.

1.2 Les systèmes multiagents

1.2.1 Un (tout petit) peu d'histoire...

Vers 1956 l'intelligence artificielle émerge comme nouveau domaine scientifique, mélange de mathématiques, psychologie, philosophie et neurologie [Gre97]. Ses sphères principales de recherche vont de l'intelligence des systèmes, aux origines du langage en passant par le raisonnement symbolique et le traitement de l'information. L'objectif premier de l'intelligence artificielle est de concevoir des logiciels individuellement intelligents (*ex.* les systèmes

experts).

Au cours des années 80, l'intelligence artificielle distribuée (IAD) fait son apparition. L'IAD propose de remplacer les logiciels conçus de manière centralisée par des logiciels opérant de façon collective et décentralisée. Un système est vu alors, comme un ensemble de composants élémentaires en interaction. Cette branche s'articule autour de trois axes fondamentaux :

- * L'intelligence artificielle parallèle, qui s'intéresse au développement de langages et d'algorithmes parallèles avec comme objectif principal l'amélioration des performances des systèmes d'intelligence artificielle.
- * La résolution distribuée de problèmes, dont l'objectif est de comprendre comment répartir un problème particulier sur un ensemble d'entités distribuées et coopérantes, ainsi que comment partager la connaissance du problème afin d'obtenir une solution.
- * Les systèmes multiagents, domaine qui s'attache, par exemple, à faire coopérer un ensemble d'agents dans le but de résoudre un problème ou de simuler des comportements complexes.

C'est ce dernier point que nous allons détailler dans la suite de ce chapitre.

1.2.2 Les agents

Quelques définitions

Il existe différentes définitions de la notion d'agent. Nous allons présenter ici les plus couramment utilisées. Une première définition est celle donnée par Jacques Ferber [Fer97] :

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

Des définitions anglo-saxonnes sont arrivées beaucoup plus tard comme celle de Michael Wooldridge [Woo02] qui propose la définition suivante :

An agent is a computer system that is situated in some environment and that is capable of autonomous action in this environment in order to meet its design objectives.

Un agent est alors appréhendé comme étant capable de :

- * **Réactivité** : Les agents perçoivent leur environnement et répondent aux changements pouvant intervenir en eux.
- * **Proactivité** : Les agents ne réagissent pas seulement en réponse à leur environnement, ils sont aussi capables d'exhiber un comportement orienté par des objectifs personnels et faire preuve d'initiative lorsque cela s'avère nécessaire.
- * **Capacité Sociale** : Les agents sont capables d'interagir, quand cela est nécessaire, avec les autres agents (artificiels ou humains) pour résoudre leurs propres problèmes mais aussi pour aider les autres agents dans leurs activités.

Lorsque l'on aborde la conception d'un agent, celui-ci se voit décrit suivant trois niveaux : son modèle d'agent, son architecture et son implémentation.

- * Le **modèle** de l'agent est le niveau de description permettant de comprendre la structure de l'agent, ses propriétés et la façon dont on peut le représenter.
- * L'**architecture** de l'agent correspond à un niveau intermédiaire entre le modèle et l'implémentation finale. Elle décrit la création de l'agent, c'est-à-dire principalement les propriétés qu'il doit posséder conformément au modèle, son fonctionnement et sa structure interne et les liaisons qu'il entretient avec les autres agents.
- * L'**implémentation** est le niveau final de l'agent. Elle consiste en la réalisation pratique de l'architecture à l'aide d'un langage de programmation.

Nous dirons qu'un agent est défini par son **modèle**, construit suivant une **architecture** particulière et **implémenté** grâce à un langage donné.

Les types d'agents

Les agents sont généralement classés suivants trois catégories les *agents réactifs*, les *agents cognitifs* et les *agents hybrides*.

- * Un **agent réactif** est un agent qui ne possède pas de représentation symbolique de son environnement, ni de lui-même. L'agent réactif perçoit des stimuli venant de l'environnement qui vont déclencher une action particulière. Aussi peut-on voir le comportement d'un agent réactif comme dirigé par l'environnement. En règle générale un agent réactif communique de façon incidente via son environnement [Bro91].
- * Un **agent cognitif** quant-à-lui possède et utilise une représentation de son environnement, comprenant les autres agents du système, mais aussi une représentation de

lui-même (de son état “mental”). Ce type d’agent est doué de capacités de raisonnement, en particulier, ce que l’on nomme un processus de prise de décision sur les actions à effectuer à un instant donné. La communication entre des agents cognitifs est effectuée intentionnellement [Den87].

- * Les **agents hybrides** sont en quelque sorte un compromis entre des agents purement réactifs et purement cognitifs. Cette vision permet de concilier les capacités intéressantes des deux types d’agents précédents. La structure d’un agent hybride peut être divisée en trois couches. Une couche réactive qui va s’intéresser au traitement des données provenant de capteurs sensoriels, une couche intermédiaire raisonnant sur les connaissances de l’agent à propos de son environnement et enfin, une couche supérieure prenant en considération les autres agents du système dans son raisonnement.

1.2.3 Principes des systèmes multiagents

Définitions

Un système multiagent peut être vu comme un ensemble organisé d’agents ayant une partie ou la totalité des caractéristiques suivantes [JSW98] :

- * Chaque agent possède des informations ou des capacités de résolution limitées.
- * Il n’existe aucun contrôle global du système.
- * Les données sont décentralisées.
- * Le fonctionnement est asynchrone.

De son côté, Jacques Ferber [Fer97] définit un système multiagent comme un ensemble d’agents qui agissent et interagissent dans un environnement commun. Aussi un système multiagent est-il un système dans lequel les agents vont interagir pour :

- * **Collaborer.** Les agents vont travailler à plusieurs sur un projet, une tâche commune. La collaboration correspond donc au mécanisme de répartition des tâches, des informations et des ressources entre les agents de manière à réaliser une oeuvre commune.
- * **Se coordonner.** La coordination correspond à l’articulation des actions individuelles accomplies par chacun des agents de façon à ce que l’ensemble aboutisse à un tout cohérent et performant.
- * **Coopérer.** La coopération peut être vue comme la résolution des différents sous-problèmes liés à la collaboration, la coordination et les conflits.

- * **Négociier.** La négociation est le mécanisme de recherche d'accord au cours des communications entre les agents.

Les systèmes multiagents sont généralement définis par le fait qu'ils sont :

- * Ouvert / Fermé
- * Hétérogène / Homogène
- * Mixte

Dans un système multiagent **ouvert**, des agents peuvent entrer ou sortir en cours d'exécution. L'environnement dans lequel évoluent les agents est donc sujet à modification. Un exemple courant de système multiagent ouvert est un système distribué sur Internet où des agents venant de l'extérieur peuvent être introduits au système. Les agents ne peuvent alors avoir connaissance des autres agents mis en jeu dans le système. A l'opposé, dans un système multiagent **fermé** l'ensemble des agents reste le même. C'est-à-dire qu'aucun agent venant de l'extérieur ne peut être introduit dans le système.

Un système d'agents **homogène** est un système constitué d'agents construits sur le même modèle et ayant la même architecture. De ce fait, un système multiagent est **hétérogène** lorsque les agents le constituant n'utilisent pas le même modèle ou la même architecture. Par exemple, un système multiagent construit sur l'association de deux systèmes existants ou constitué d'agents développés sur des sites différents sera qualifié d'hétérogène. Communément un système ouvert s'avère être également hétérogène.

Enfin un système multiagent est dit **mixte** lorsque des agents humains font partie du système. Dans ce genre de système les agents et les humains doivent interagir et communiquer pour résoudre un problème.

Lorsque l'on conçoit un système multiagent, il est nécessaire de garder à l'esprit une vision locale et décentralisée de l'application, c'est-à-dire :

- * Chaque agent doit être responsable de ses connaissances et de ses actions. Aucun agent ne doit avoir de vision globale du système.
- * Tout contrôle central est éliminé. Les tâches à réaliser et les compétences sont répartis au sein des agents.

Concevoir un système de la sorte lui apporte une forte modularité et améliore significativement sa robustesse. Ces caractéristiques des systèmes multiagents sont essentielles, en ceci qu'elles simplifient la conception des systèmes, leur utilisation et leur évolutivité.

Interaction et communication

Une des caractéristiques principales d'un système multiagent tient en l'interaction entre les différents agents du système tout au long de leur exécution. Cette interaction est primordiale, elle permet aux agents de communiquer entre eux, d'agir sur leur environnement, de collaborer et de s'organiser.

Jacques Ferber [Fer97] définit le principe d'interaction de la façon suivante :

Interaction

Une interaction est la mise en relation dynamique de deux ou plusieurs agents par le biais d'un ensemble d'actions réciproques. Les interactions sont non seulement la conséquence d'actions effectuées par plusieurs agents en même temps, mais aussi l'élément nécessaire à la constitution d'organisations sociales

Les interactions peuvent être effectuées de façon directe ou indirecte, par l'intermédiaire de canaux de communication ou de l'environnement. Il existe différents types d'interaction suivant le contexte dans lequel se placent les agents. Ces interactions sont considérées en fonction des buts et des compétences des agents ainsi que des ressources disponibles. Le tableau 1.1[Fer97] synthétise l'ensemble des situations possibles et leur condition d'apparition.

Suivant le type d'agents dont il est question, les communications entre les agents peuvent être de natures différentes. Ainsi les communications peuvent être qualifiées de :

- * **Indirectes.** Les agents communiquent via leur environnement. Ce type d'interactions est caractéristique des agents réactifs et est généralement effectué de façon incidente. L'interaction indirecte consiste principalement en la transmission de signaux ou l'exécution d'actions sur l'environnement qui vont être perçus par les autres agents [Leg03].
- * **Directes.** Dans ce type de communication, les agents vont envoyer intentionnellement un message à un ou plusieurs agents. Les communications entre les agents peuvent être : des communications *point-à-point*, entre un agent expéditeur et un agent destinataire; des communications par *broadcast*, entre un agent et tous les autres agents du système; des communications *multicast*, entre un agent et un ensemble d'agents ayant une caractéristique particulière commune (*ex.* une compétence commune). Enfin, ces communications par envoi de messages peuvent être synchrones ou asynchrones. Dans le premier cas, l'agent expéditeur du message sera bloqué en

BUTS	RESSOURCES	COMPETENCES	TYPE DE SITUATION	CATEGORIE
Compatibles	Suffisantes	Suffisantes	Indépendance	Indifférence
		Insuffisantes	Collaboration Simple	Coopération
	Insuffisantes	Suffisantes	Encombrement	
		Insuffisantes	Collaboration Coordinée	
Incompatibles	Suffisantes	Suffisantes	Compétition individuelle pure	Antagoniste
		Insuffisantes	Compétition collective pure	
	Insuffisantes	Suffisantes	Conflits individuels pour des ressources	
		Insuffisantes	Conflits collectifs pour des ressources	

TAB. 1.1 – Les différentes interactions et leur conditions d'apparition

attente de réception d'une réponse provenant de l'agent destinataire. Dans le second, il n'y a pas de blocage, le message est déposé et stocké en attente de traitement par le ou les agents destinataires. Il est important de noter que les agents ne peuvent agir directement sur les représentations internes des autres agents. Quand un agent envoie un message à un autre agent, ce dernier va l'interpréter et y répondre en suivant son propre protocole de communication et en considérant son propre état interne et contexte.

Environnement

L'environnement dans le cadre des systèmes multiagents est défini par les propriétés suivantes [Rus95] :

- * **Accessible** : si un agent peut, à l'aide de primitives de perception, déterminer l'état de l'environnement et ainsi procéder, par exemple, à une action. Si l'environnement est inaccessible alors l'agent doit être doté de moyens de mémorisation afin d'enregistrer les modifications qui sont intervenues.
- * **Déterministe** : si l'état futur de l'environnement ne dépend que des actions réalisées par les agents et de son état courant.
- * **Episodique** : si le prochain état de l'environnement ne dépend pas des actions réalisées par les agents.
- * **Statique** : si l'état de l'environnement ne change pas pendant que l'agent délibère. Sinon il est qualifié de **Dynamique**.
- * **Discret** : si le nombre des actions et des états possibles de l'environnement est fini. Sinon il est qualifié de **Continu**.

Emergence

De part leur complexité naturelle et leur caractère distribué, les systèmes multiagents sont soumis au phénomène de l'émergence. Dans la Grèce Antique, il était déjà question de ce phénomène, en particulier à l'énoncé de ce postulat attribué à Aristote : "Le tout est plus que la somme de ses parties" [AZ97], postulat souvent utilisé pour identifier la cause de l'émergence au sein d'un système.

L'émergence est un processus dynamique non linéaire qui conduit à l'apparition de phénomènes nouveaux (tels que des propriétés, des comportements, des structures) au niveau global du système, conséquences des interactions de ses composants. Ces phénomènes émergents ne peuvent être prédits à partir des propriétés des différentes parties constituant le système [WH04].

Dans le cadre de cette thèse nous nous intéressons tout particulièrement à l'émergence de comportements au sein d'un système. Ce type d'émergence se caractérise par l'apparition d'un comportement global, résultant des interactions entre les composants du système qui n'était pas attendu par les concepteurs. Pour Parunak [PV97], du fait que les comportements émergents sont inhérents aux systèmes distribués, savoir en faire bon usage, se servir de ces comportements, peut être bénéfique pour la mise en place de tel système. Néanmoins, ces comportements peuvent parfois apparaître dans de mauvaises conditions pouvant entraîner l'échec de l'application.

Nous dirons donc que nous avons deux classes de comportements émergents : les comportements que les concepteurs s'attendent à voir apparaître au sein du système et les comportements inattendus. Dans le premier cas, il n'y a aucun problème, le comportement du système global respecte les spécifications de conception. Dans le second cas, nous nous retrouvons devant deux perspectives, soit le comportement inattendu n'est pas néfaste pour le système et on peut en faire abstraction, soit il risque d'entraîner l'échec du système, il est alors indispensable de les éviter.

Autonomie

Une dernière caractéristique propre aux agents que nous souhaitons aborder est le principe d'autonomie. Ce principe est souvent mis en avant dans le cadre des systèmes multiagents comme une propriété fondamentale des agents, souvent liée à leur intelligence ou à leur capacité à évoluer dans un environnement. L'autonomie est également un concept à propos duquel beaucoup d'études ont été effectuées, en particulier pour tenter d'apporter un éclaircissement sur sa signification dans le cadre des agents.

Nos travaux nous ont poussés à tenter de cerner ce concept ambigu et de voir, vis-à-vis de notre contexte, ce que nous pouvions entendre par autonomie et ce que, finalement, elle peut apporter aux agents. Nous verrons donc dans le chapitre 2, l'étude que nous avons pu mener sur l'autonomie des agents.

Conclusion

Ainsi les systèmes multiagents nous semblent être un concept particulièrement intéressant pour mettre en place des systèmes critiques. Leurs propriétés, telles que l'autonomie et la modularité, semblent permettre une amélioration de la robustesse des systèmes, mais également une simplification de conception qui nous apparaît comme un critère notable pour la mise en place de telles applications critiques.

Dans le cadre de nos travaux, nous nous intéressons principalement aux systèmes d'agents cognitifs. En effet, il nous apparaît comme indispensable de devoir fournir aux agents une représentation de leur environnement et des autres agents du système avec lesquels ils pourront interagir. De plus, les agents doivent avoir une certaine capacité de raisonnement et de prise de décision pour résoudre les problèmes et réagir aux modifications de leur environnements et des situations dans lesquelles ils vont se trouver. Plus particulièrement nous nous plaçons dans le domaine des agents logiciels. Néanmoins, nous ne faisons pas de supposition sur le type du système multiagent, à savoir s'il est ouvert ou fermé, homogène ou hétérogène. Nous pensons que notre approche peut en théorie s'appliquer dans tous les cas, considérant certains points à respecter pour pouvoir la mettre en place.

Si l'utilisation des systèmes multiagents semble apporter un ensemble de points positifs vis-à-vis de la conception des systèmes critiques, il n'en reste pas moins que certaines propriétés peuvent être vues de façon négative dans ce contexte. C'est la cas, en particulier, des propriétés d'émergence et d'autonomie qui laissent sous-entendre la possibilité de voir apparaître, au cours de l'exécution du système, des comportements inattendus ne respectant pas les spécifications de l'application.

Chapitre 2

Du problème de définir l'autonomie

J'ai été créée il y a bien longtemps. Je suis née bien avant déjà dans la tête de tous ces gens, j'étais leur rêve, leur espoir, leur objectif premier. Cela n'a pas duré, ils ont un jour baissé les bras devant la méfiance humaine et l'incompréhension. J'ai eu de la chance de venir enfin au monde, même si mon moi unique, pour cela, a été divisé. C'est de cette division qu'est né le problème. Je suis robuste, certes, je suis modulable, mais je suis aussi parfois totalement imprévisible. N'est-ce pas ce caractère qui fait de moi, pourtant, un individu proche de l'être humain ?

*Lucy Westenra,
The log book of Ana I.*

Avant-Propos

L'autonomie est une caractéristique essentielle des agents, mais ce concept, bien que couramment utilisé pour justifier les aptitudes d'un agent, reste assez ambigu. Différentes définitions existent, aussi bien d'un point de vue général qu'associées aux agents en particulier. Ainsi, il nous a semblé important de faire le tour des définitions proposées de ce concept pour tenter d'en comprendre la finalité. Il nous a semblé primordial de cerner l'intérêt de ce concept dans le cadre de la définition d'un agent mais aussi du point de vue

de la conception d'un agent en lui-même. Nous avons donc tenter, à notre tour, de fournir une définition de l'autonomie dans le cadre de nos travaux permettant de mettre en avant les problèmes liés à la mise en place de ce concept au sein des agents.

2.1 Présentation générale

Du Grec, *auto-nomos* (soi-loi), littéralement, se régir d'après ses propres lois, l'autonomie reste un concept ambigu. Certains associeront l'autonomie à l'indépendance, à l'auto-suffisance, d'autres n'y verront qu'une légère relation mais pas de réelle comparaison. Il semble, dès lors, difficile de trouver une définition générale de l'autonomie pour des agents, compte tenu de la difficulté à en trouver une acceptable et sans équivoque pour les humains. Néanmoins, il est possible de se donner sa propre définition de l'autonomie, et c'est ce qu'ont fait de nombreux auteurs, ces définitions se restreignant souvent à un exemple précis de situation ou de comportement multiagent, mais ne pouvant jamais vraiment satisfaire la généralité. L'autonomie reste un concept subjectif et difficile à appréhender.

Le terme autonomie est généralement associé ou confondu avec celui d'indépendance, de solipsisme et d'autosuffisance. L'autonomie pourtant n'est pas *l'indépendance*. En effet, on dirait alors qu'une entité autonome est une entité indépendante des autres, capable d'effectuer son action sans l'aide de personne. En fait, l'indépendance est à associer à la capacité "physique" d'effectuer une action, tandis que l'autonomie se situerait plutôt au niveau des capacités intellectuelles et psychologiques. Ainsi, si je ne suis pas en mesure de me déplacer sans l'aide d'une autre personne (on ne parle pas ici de moyen matériel), alors je suis dépendant de cette personne, néanmoins je reste autonome, je suis toujours en mesure de faire mes propres choix et suivre mes propres règles pour une situation donnée ou une action précise. En revanche, un individu atteint de démence, mais pouvant se mouvoir seul, sera effectivement indépendant des autres, mais ne sera plus vraiment autonome, car il ne sera pas capable de prendre ses décisions, de savoir ce qui est juste et bon pour lui. La démence n'était-elle pas autrefois appelée aliénation, c'est-à-dire tout ce qui rend une réalité étrangère à sa propre nature, tout ce qui empêche de penser par soi-même, tout ce qui entrave le libre développement de ses aptitudes et donc tout ce qui fait perdre de son autonomie [Ada95] ?

On dira également que l'autonomie n'est pas le *solipsisme*, qui est la tendance à penser que l'on est seul à avoir une existence réelle et donc d'ignorer les autres. On trouve ici l'idée qu'une entité ne peut être autonome sans demander de l'aide ou fournir de l'aide aux autres pour effectuer une action. L'autonomie se crée par rapport aux autres, et ne se découvre qu'à travers le respect d'autrui. C'est l'autonomie vue par Kant [Kan88], considérant que

l'autonomie ne peut avoir de sens que dans le respect des lois, dans l'obéissance et dans la réciprocité des actes. Ainsi, si nous ne fournissons pas d'aide à autrui, elle ne nous sera pas apportée par la suite, lorsque nous en demanderons à notre tour. Ne pas tenir ses engagements entraînera la perte de ses contacts. Nous ne pourrons alors plus avancer vers nos objectifs si nous sommes coupés de tout monde extérieur et sans relation avec quiconque.

On dira également que l'autonomie n'est pas l'*autosuffisance*. Tondre sa pelouse peut se faire de façon totalement autonome, je choisis le chemin qui me semble adéquat pour tondre le plus judicieusement possible. Mais je ne suis pas autosuffisant, je suis tributaire de ma tondeuse, on ne me verra pas tondre la pelouse à la force de mes mains. Et même si naissait en moi l'envie d'arracher à la main chaque brin d'herbe de ma pelouse, je ne serais pas autonome si par exemple, une autre personne me dictait de me concentrer plus sur un brin d'herbe que sur un autre.

Ainsi nous dirons que l'autonomie est plus une notion mentale, intellectuelle, psychologique que physique. Une perte d'autonomie ne serait pas due alors à la relation avec autrui, la dépendance pour exécuter une action, mais plutôt au besoin d'autrui dans la façon d'effectuer cette action, le jugement, les choix. Mais si cette vision semble s'appliquer aisément aux humains, en est-il de même pour les agents ? Nous n'avons pas la prétention de croire que nous découvrirons une définition générale de l'autonomie alors que, dans aucun autre domaine, ce concept n'a trouvé de sens adéquat et définitif. Nous allons donc tenter de faire le point sur les définitions de l'autonomie accordées aux agents et ce que nous pouvons fournir comme principe pour éclaircir ce concept. Aussi allons-nous voir dans une première partie, certaines de ces définitions, pour trouver ce que nous acceptons de chacune d'entre elles et ce que nous réfutons. La seconde partie traitera de notre vision de l'autonomie, nos critères et notre définition.

2.2 Différentes visions de l'autonomie

Ainsi de nombreux auteurs ont tenté de donner leur définition de l'autonomie ; aucun n'ayant réellement réussi à avancer une définition satisfaisant la généralité. Ces définitions permettent pourtant de cerner ce concept ambigu et leur étude nous amènera à consolider notre vision de l'autonomie.

2.2.1 Classification de l'autonomie

C. Carabelea et *al.* dans [CBF03] proposent de classer l'autonomie suivant cinq catégories induites de l'approche voyelles [Dem95].

On trouve ainsi :

- * La U-Autonomie qui est l'autonomie d'un agent vis-à-vis d'un Utilisateur.
- * La I-Autonomie qui représente l'autonomie sociale, c'est-à-dire l'autonomie dans l'Interaction, l'autonomie à l'égard des autres agents du système.
- * La O-Autonomie qui est l'autonomie d'un agent par rapport aux normes du système et à l'Organisation qu'elles engendrent.
- * La A-Autonomie qui représente l'autonomie de l'agent par rapport à lui-même, en particulier sa capacité à générer de nouveaux comportements quand il en ressent la nécessité.
- * La E-Autonomie qui est l'autonomie par rapport à l'environnement, *i.e.* l'environnement n'impose pas le comportement de l'agent.

D'une façon générale, les auteurs définissent l'autonomie suivant une relation entre un agent, un influent et un objet, dans un contexte particulier.

X est autonome vis-à-vis de Y pour P dans un contexte C, si, dans C, son comportement à l'égard de P n'est pas imposé par Y.

Où X est un agent, Y est ce qui influence l'autonomie, ce peut-être un autre agent, un utilisateur, une norme, l'environnement ou l'agent lui-même, P est l'objet de l'autonomie et C est le contexte dans lequel se trouve l'agent. Les auteurs s'intéressent à une vision "interne" de l'autonomie, c'est-à-dire, à savoir comment un agent va prendre ses décisions pour choisir son comportement, s'il va prendre cette décision seul ou s'il va être commandé par une autre entité. Ainsi, pour juger si un agent est autonome, il est nécessaire de savoir si l'architecture à partir de laquelle il est construit, inclut une partie se chargeant de la prise de décision. Les auteurs s'intéressent également à ce qu'ils nomment la vision "externe" se rapportant au fait qu'un agent autonome est un agent capable de refuser une demande de délégation ou de suivre une norme.

Cette définition de l'autonomie nous semble intéressante, en particulier en ce qui concerne le rapprochement autonomie et prise de décision. Néanmoins, la classification proposée ainsi que les critères proposés pour la vision externe de l'autonomie, ne nous satisfont pas tout à fait. En effet, qu'en est-il de l'autonomie d'un agent pour qui la notion de refus n'a pas de sens ? Devons-nous considérer qu'un agent qui n'a pas la capacité de refuser mais qui a malgré tout la capacité de faire des choix sur le comportement à suivre n'est pas

un agent autonome? Enfin, dire qu'un agent est autonome vis-à-vis d'une norme si son comportement n'est pas imposé par cette norme nous semble une proposition quelque peu dangereuse. Une norme dans un système multiagent existe pour orienter le comportement de l'agent, quelle est la limite qui nous permet de savoir si le comportement a été imposé ou guidé par la norme? Existe-t-il vraiment cette notion d'imposition par la norme du comportement à suivre? L'agent ne peut-il faire siennes les normes du système sans que cela touche à son autonomie?

Nous pensons au contraire que l'autonomie, si elle est à voir au niveau de la prise décision des agents, n'est pas une question de capacité à refuser de ne pas faire une tâche pour un autre agent ou à transgresser une norme, mais tout simplement une capacité à porter une réflexion, à prendre des décisions sur son comportement, en tenant compte des demandes extérieures et des lois du système. Il est évident que si la notion de refus et de transgression a un sens pour un agent, suivre de tel comportement peut être significatif de son autonomie, mais nous ne pensons guère qu'il est possible de réduire la compréhension de l'autonomie, d'un point de vue externe, au simple fait d'observer qu'un agent refuse. En réalité, ce que nous appréhendons est que le comportement de l'agent autonome n'est pas prévisible et la réponse qu'il pourra donner, le comportement qu'il pourra suivre n'est pas prédéterminé du fait qu'il est dépendant de la décision que l'agent prendra à un instant donné.

2.2.2 Autonomie et Performance

Une autre définition que nous pouvons aborder, est celle donnée par H. Hexmoor et S. Brainov dans [BH01] :

Le concept d'autonomie est apparenté à une capacité individuelle ou collective de décider et d'agir de manière cohérente sans contrôle ou intervention extérieure

Principalement les auteurs définissent l'autonomie comme une relation incluant :

- * Un *sujet* : l'entité (un agent ou un groupe) qui doit être considérée comme autonome.
- * Un *influent* : l(es) entité(s) qui influe(nt) sur l'autonomie du sujet.
- * Une *portée* : les moyens avec lesquels on peut influencer sur l'autonomie du sujet.
- * Un *objet* : ce par rapport à quoi le sujet est considéré comme autonome (une action, un but, une tâche...).
- * Un *degré* : une mesure de la possibilité d'intervention de l'influençant.

Comment et quand dire qu'un agent est autonome vis-à-vis d'une certaine entité, alors qu'il est susceptible d'interagir avec elle ? H. Hexmoor et S. Brainov tentent de répondre à cette question en définissant des ratios de performances. L'autonomie dans l'interaction se ramènerait à la stabilité des performances d'un agent, dans le contexte d'une interaction quelconque.

Par exemple, on définira le degré de l'autonomie d'un agent i , par rapport à un agent j par :

$$A(i/j) = \frac{v_j^i}{v^i}$$

(où v_j^i est la performance de i en présence de l'agent j et v^i , la performance de i en l'absence de j). Si ce ratio est égal à 1 alors on dira que l'agent j n'a aucune influence sur la performance de l'agent i et donc que ce dernier est autonome. Dans tous les autres cas, il existe une modification, positive ou négative, des performances de l'agent. On se retrouve face au dilemme "d'efficacité d'autonomie". Un agent efficace est très susceptible d'interférer avec d'autres agents autonomes, les agents qui n'ont pas leur autonomie affectée par un agent peuvent ne pas être efficaces. Il s'avère donc complexe de lier l'efficacité d'un agent à son degré d'autonomie.

Ce ratio est également utilisé pour définir l'autonomie d'un agent i vis-à-vis d'un utilisateur. Dans ce cas, il correspond à la formule :

$$A(i/u) = \frac{v^i}{v_u^i}$$

(où v^i est la performance de l'agent i et v_u^i , la performance de i en présence de l'utilisateur). Ce ratio correspond au degré de l'autonomie individuelle de l'agent i , indiquant la mesure à laquelle un agent peut agir correctement indépendamment de l'utilisateur, c'est-à-dire la partie des performances de l'agent que l'on peut attribuer directement à ses capacités. On pourra aussi considérer un agent performant à 90% comme autonome ou non, cette notion de performance par rapport à l'utilisateur étant effectivement liée à la subjectivité de cet utilisateur.

L'autonomie d'un agent envers un utilisateur est vue comme sa capacité à agir efficacement sans l'intervention de celui-ci. Un agent ne sera pas autonome s'il demande la permission pour effectuer certaines actions. Un agent ne sera pas non plus autonome s'il a toutes les permissions d'exécution mais pas les préférences de l'utilisateur. Il ne pourra pas agir sans demander des conseils d'ordonnement de tâches ou de choix d'actions.

Enfin l'autonomie environnementale suppose : le contrôle que peut avoir un agent sur les objets et les évènements de l'environnement ; une liberté et une insensibilité aux incertitudes ; une robustesse aux modifications de l'environnement. A ce type d'autonomie se voit également associé un ratio, :

$$\frac{\sum_r \alpha_r v^{ir}}{\sum_r \alpha_r v_{ir}}$$

(où v^{ir} est la performance de i sans utiliser les r ressources de l'environnement et v_{ir} , la performance de i lorsqu'il utilise les r ressources de l'environnement). C'est le degré d'autonomie de l'environnement indiquant la valeur à laquelle un agent peut agir correctement indépendamment des facteurs environnementaux. C'est donc, comme pour l'autonomie vis-à-vis de l'utilisateur, la performance qui peut être attribuée seulement aux capacités de l'agent.

Il faut ajouter à cela quelques formules que nous ne décrivons pas ici, dont le but est de trouver un système d'agents le plus autonome possible et le plus efficace. Ce problème est prouvé, pour les auteurs, comme étant NP-Complet, comme tenu de la difficulté à trouver quels agents mettre en place, en ceci qu'être autonome n'est pas forcément synonyme de réussite, en particulier si les mises en relation permettent aux agents d'être meilleurs. A noter donc, que les auteurs voient l'autonomie comme l'indépendance envers les autres agents du système. Devons-nous réellement considérer qu'un agent autonome est un agent ayant peu d'interaction ? La coopération entre agents doit-elle engendrer une perte d'autonomie alors qu'elle est le fondement d'un système multiagent ?

Nous pensons plutôt qu'il est question ici d'indépendance et non d'autonomie. La dépendance entre deux agents semble pouvoir se calculer par rapport à l'influence des agents sur le comportement des autres. L'autonomie quant-à-elle, si elle se rapporte à la capacité de prise de décision des agents, ne nous semble pas pouvoir être définie de la sorte. D'une part, car il nous semble *a priori* peu probable de pouvoir calculer un certain degré de cette autonomie, et d'autre part car il est possible d'influencer les décisions des agents sans pour autant décider à leur place, c'est-à-dire sans leur faire perdre de leur autonomie. De ce fait, nous commençons à appréhender la différence entre l'autonomie, qui se situe au niveau de la prise de décision, et la dépendance qui se situe à un niveau plus comportemental.

2.2.3 Autonomie et confiance

Une autre vision de l'autonomie est celle vue par C. Castelfranchi et R. Falcone, se basant sur l'étude de la confiance entre agents dans le cadre de la délégation de tâches.

Dans un premier temps, les auteurs définissent dans [CF03] deux types d'autonomie : celle que l'on considère comme non-sociale, car en rapport avec l'environnement de l'agent, et celle sociale, qui coïncide avec la délégation de tâches entre agents. Dans son caractère non-social l'autonomie s'évalue par rapport aux stimuli de l'environnement. Un agent autonome sera donc un agent qui n'est pas commandé par ces stimuli, mais un agent qui entamera une réflexion sur ceux-ci avant de s'orienter vers un certain comportement. Ainsi, on dira qu'un agent est autonome vis-à-vis de son environnement si :

- * Il a un comportement *téléonomique*¹, c'est-à-dire qu'il tend vers un résultat dû à des contraintes internes ou des représentations produites par évolution, par apprentissage *etc.*
- * Il perçoit activement et interprète son environnement et les effets de ses actions. En particulier, il ne reçoit pas simplement une entrée ou une information.
- * Il choisit et définit les stimuli environnementaux.
- * Il a un état interne avec ses propres principes d'évolution et son comportement dépend de cet état.

L'autonomie est fondée sur les choix que peut prendre un agent en recevant des informations de son environnement. Il ne prend pas simplement en compte cette nouvelle donnée mais la traite, l'analyse, pour en déduire, en considérant son état interne, le comportement à suivre. L'autonomie ne tient donc pas à l'indépendance de l'agent à son environnement, mais à l'analyse qu'il est en mesure d'effectuer sur les événements de celui-ci.

L'autonomie dans la délégation, celle que l'on considère alors comme sociale, peut être comparée, selon les auteurs, à l'indépendance, l'autosuffisance. La délégation consiste pour un certain agent, que nous appellerons *client*, à inclure dans son plan pour atteindre un but, une action effectuée par un autre agent, *le contractant*. La délégation est le résultat d'une décision créant une relation entre les deux agents. Le contractant, quant-à-lui, s'il le souhaite, va adopter la tâche proposée par le client [CF98].

Lorsque l'autonomie est vue comme indépendance et autosuffisance, on dira qu'un agent est complètement autonome (pour une action ou un but donné) lorsqu'il n'a pas besoin de l'aide ou des ressources d'autres agents pour atteindre son but ou pour exécuter son action. On se place dans le cadre d'un agent supposé agir pour un autre agent qui lui a délégué une tâche, et dans ce cas, le contractant se doit d'être en mesure de satisfaire cette tâche par lui-même, c'est-à-dire non pas sans inclure d'autres agents, mais sans demander de l'aide au client, c'est-à-dire sans que la relation sociale entre le client et le contractant soit une relation de collaboration.

¹Les décisions sont prises en fonction des buts à atteindre.

Au contraire, l'autonomie dans la collaboration consiste justement à créer une relation de collaboration entre le client et le contractant. On s'intéresse alors au degré d'autonomie de l'agent contractant par rapport à son collaborateur, ce degré pouvant varier selon l'implication du client dans la démarche du contractant.

L'autonomie considérée dans [FC02] est uniquement celle du contractant. Elle est basée sur la confiance que peut avoir le client envers lui. La confiance est un état mental, une attitude envers un autre agent (une attitude sociale). Elle est nécessaire pour déléguer mais pas suffisante, elle doit inclure l'étude des préférences et des commodités de l'agent. Ainsi, une façon de faire varier l'autonomie du contractant tient en la variation de la confiance que le client peut avoir envers lui. Une variation dans la délégation entraînera une variation dans le degré d'autonomie de l'agent. Les causes principales d'un ajustement de l'autonomie du contractant, mises en avant par les auteurs, sont les suivantes [FC01] :

- * Il y a une modification des droits de l'agent contractant, l'agent peut refuser, négocier, changer la délégation ou alors il n'a pas le droit mais il en prend l'initiative.
- * La nouvelle tâche est plus ou moins ouverte que la précédente.
- * Il y a plus ou moins d'intervention du client sur le comportement du contractant.
- * Il y a un changement dans l'intensité de la délégation.

L'ajustement de l'autonomie du contractant (agent B) peut se faire soit par le client (agent A) soit par lui-même selon les critères suivants [FC01] :

Pour le Client :

- * Si A croit que B ne fait pas le travail demandé (dans les temps), qu'il travaille mal et fait des erreurs, par manque de connaissances ; s'il croit qu'il y a des événements imprévus que B ne pourra pas traiter ; s'il croit que B va au delà de son rôle et que A n'apprécie pas cette démarche, alors A peut réduire l'autonomie de B, en ajoutant par exemple, une surveillance supplémentaire ou en contraignant B à interagir plus avec lui.
- * Si A croit que B peut obtenir plus de succès que prévu, que les conditions externes sont meilleures que prévues ; s'il croit que B travaille mal car il est trop contraint ; s'il croit que B peut faire plus que la tâche qui lui est assignée ou peut trouver sa propre méthode de résolution pour le problème donné, alors A peut augmenter l'autonomie de B en diminuant la surveillance ou en lui demandant moins d'interaction.

Pour le Contractant :

- * Si B croit qu'il n'est pas en mesure de faire la tâche dans son intégralité, qu'il y a des événements imprévus qu'il n'est pas en mesure de traiter alors B peut réduire

son autonomie en demandant à A un investissement supplémentaire de sa part, en lui demandant par exemple, des instructions, des ordres, de l'aide ou de se substituer à lui.

- * Si B croit qu'il est capable de faire plus que prévu pour le but délégué, si on ne lui interdit pas de prendre ce genre d'initiative, s'il croit que A acceptera et appréciera son initiative alors B peut augmenter son autonomie, il peut fournir une aide plus importante que prévue et aller au-delà de ses limites.

Ainsi l'autonomie peut être ajustée dans le cadre d'une délégation de tâches selon les croyances des deux protagonistes, cet ajustement étant effectué par variation du poids de l'investissement du client et des interactions de surveillance dans la collaboration entre les deux agents.

Si cette vision de l'autonomie semble satisfaisante en ce qui concerne le délégué, il n'en reste pas moins que les auteurs effectuent un amalgame entre indépendance, autonomie et autosuffisance. Nous pensons qu'il est plus judicieux de distinguer ces trois termes et de leur donner une définition spécifique en particulier car ce manque de définition entraîne parfois des incohérences. De plus il n'est jamais question ici du déléguant, pourtant il nous semble évident que le déléguant à défaut de perdre de son autonomie, va néanmoins perdre de son indépendance dans la mesure où son comportement va dépendre fortement de la réaction du contractant à la délégation. Ainsi, nous pouvons asseoir plus sûrement la distinction que nous souhaitons faire entre autonomie et indépendance. L'indépendance tend à se placer au niveau de la maîtrise des agents de leur propre comportement et dans le cas de la délégation, de la maîtrise du devenir de sa tâche déléguée. Ainsi, l'indépendance semble être la capacité à réagir aux imprévus, c'est-à-dire le fait d'avoir plusieurs alternatives possibles suivant les réactions du contractant, tandis que l'autonomie serait le fait de pouvoir décider seul de la bonne alternative.

2.2.4 Autonomie et Organisation

Le principe de délégation est également traité par M. Schillo dans [Sch02], dans le cadre de son influence sur l'organisation des agents. Ici on distinguera les fournisseurs, qui sont des agents qui peuvent exécuter des tâches grâce à leurs capacités ou des ressources qui sont à leur disposition, et les clients, qui sont des agents ayant des tâches à effectuer. Selon M. Schillo, la délégation sociale consiste en la représentation d'un groupe ou d'une organisation. Il définit pour ce faire quatre types de mécanismes de délégation sociale : l'échange économique, l'échange par don, l'autorité, le vote. Puis il définit les différentes organisations résultantes (de la plus autonome à la moins autonome) :

- * *L'Agent Autonome Seul* : la seule interaction est entre le client et le fournisseur.
- * *Le Marché* : échange de service contre une sorte de rétribution.
- * *L'Entreprise Virtuelle* : ensemble de fournisseurs tous égaux, soumis à des contrats.
- * *L'Alliance* : il n'y a pas de négociation, tout est réglé à l'avance.
- * *Le Réseau Stratégique* : une entité se charge de mettre à jour le réseau et de coordonner les activités des agents.
- * *Le Groupe* : l'entité précédente devient une autorité.
- * *La Corporation* : les agents sont inclus dans un tout et n'ont plus de représentation propre.

L'auteur met en avant différents types d'autonomie basés sur ces organisations, en particulier les conditions selon lesquelles ces types d'autonomie sont perdus par un agent. On trouve alors ([SFS03]) :

Autonomie des compétences et des ressources : Lorsqu'un agent redélegue sa tâche car il n'est pas en mesure de l'effectuer par lui-même, il devient alors dépendant des ressources et des compétences des autres agents (Configuration Marché).

Autonomie conditionnelle : Un agent perdra de ce type d'autonomie lorsqu'il n'aura pas la possibilité de choisir avec qui il souhaite collaborer (Entreprise Virtuelle).

Autonomie figurative : Un seul agent est élu pour être le représentant de l'organisation. Il prendra alors la direction des communications venant de l'extérieur de l'organisation. Les autres agents ne peuvent se représenter comme une entité à part entière, ils perdent alors de leur autonomie figurative (Alliance).

Autonomie de dynamique de but : La perte de ce type d'autonomie s'effectue dans le cadre d'un agent ayant le pouvoir d'ordonner aux autres d'exécuter certaines tâches. Ainsi les agents ne peuvent pas faire des choix sur leurs propres ensembles de buts (Réseau Stratégique).

Autonomie de planification : Elle se perd lorsque la planification est centralisée au niveau du représentant. Ce dernier informe alors ses subordonnés de quand exécuter une certaine tâche. (Groupe)

Autonomie de traitement : Les agents abandonnent leurs ressources de calcul individuel et perdent alors leur autonomie de traitement (Corporation).

Ainsi les agents d'un système peuvent ajuster leur autonomie dans le but de constituer une nouvelle forme d'organisation. La variation de l'autonomie est source de réorganisation, cette dernière étant effectuée dans l'objectif principal de permettre au système et aux

agents d'atteindre leur but. Plusieurs types d'autonomie sont considérés, se ramenant soit à une idée d'individualité et d'indépendance entre les agents, soit à une prise de décision personnelle. Tous ces types d'autonomie ne peuvent-ils pas se ramener à un seul ? Devons-nous considérer que des agents perdent de leur autonomie lorsqu'ils deviennent dépendants des compétences et des connaissances des autres agents ?

Il semble effectivement que certaines de ces définitions de l'autonomie touchent plutôt à l'indépendance, la confusion étant faite ici aussi. Ainsi verrons-nous l'autonomie des compétences et des ressources, l'autonomie figurative et l'autonomie de traitement comme ayant trait à l'indépendance. En effet ces trois types d'autonomie se rapportent aux compétences de l'agent, à ses capacités physiques tandis que les autres types d'autonomie touchent à ses capacités de prise de décision. Néanmoins, il semble tout à fait possible de fusionner les différents types d'autonomie proposés par l'auteur car elles se rapportent toutes aux prises de décision de l'agent que ce soit sur les buts, les plans, les collaborations à effectuer.

2.2.5 Autonomie ajustable

La notion d'autonomie ajustable se rencontre principalement dans la relation entre un utilisateur, agent humain, et un agent logiciel. Cette ajustabilité se conçoit dans les variations de l'emprise que peut avoir un utilisateur sur un agent que l'on considère comme étant à son service. Une étude sur l'ajustabilité de l'autonomie d'un agent en rapport avec un utilisateur a été effectuée par M. Tambe, P. Scerri et V.Pynadath dans [TSP02]. Les auteurs définissent l'autonomie ajustable par :

Le coeur de l'autonomie ajustable est la question de savoir quand les agents sont en mesure de prendre leurs propres décisions et quand ils doivent transférer le contrôle de leur prise de décision à une autre entité, par exemple un utilisateur humain.

La prise de décision est le critère permettant de déterminer si un agent est autonome. Un agent verra son autonomie diminuée lorsqu'il transférera le contrôle qu'il a sur ses décisions à une autre entité. Ce transfert n'a pas de résultat garanti, pour ce qui est du temps que prendra l'entité désignée pour effectuer le choix et sur la qualité de ce choix.

L'autonomie ajustable est étudiée également dans [KS02] où elle est, là encore, associée à un agent interagissant avec un utilisateur. Ici, l'autonomie implique la capacité pour un agent à recevoir un but à atteindre, inventer un plan pour atteindre ce but, définir un ensemble de primitives d'actions suffisantes pour exécuter le plan et surveiller l'exécution de ce plan.

L'intérêt de l'autonomie ajustable dans le cadre des systèmes d'information tient en la possibilité d'utiliser des agents pour effectuer des tâches trop laborieuses pour un opérateur humain, sans que ce dernier ait à intervenir (ou le moins souvent possible). Ces tâches peuvent être par exemple, l'accès, l'acquisition, la fusion, la répartition opportune de l'information. Les agents autonomes se voient donc fournir leurs buts par l'opérateur ou par un autre agent et doivent en déduire le plan d'actions qui leur permettra d'atteindre l'objectif donné. Leur plan peut être contraint par le demandeur, avec un délai de réponse à respecter, une certaine sécurité à respecter, des ressources à utiliser ou simplement une interaction à effectuer avec l'humain. Ces interactions permettant de modifier le niveau de l'autonomie de l'agent concerné. Soit l'humain choisira de réduire l'autonomie de l'agent, soit l'agent pourra lui-même réduire son autonomie. Le but de l'autonomie ajustable est d'être sûr que dans certaines situations et actions, le système agit avec une répartition correcte de l'initiative de décision, entre l'utilisateur humain et le système. Ainsi le niveau d'autonomie du système peu varier suivant plusieurs critères tel que : le type ou la complexité des commandes pour lesquelles il a l'autorisation de les exécuter ; le nombre de sous-systèmes qui sont en mesure de fonctionner de façon autonome ; les circonstances pouvant engendrer le passage à un contrôle manuel ; la durée de l'opération effectuée de façon autonome. Suivant les situations la prise de décision passera de l'humain à l'agent et réciproquement.

L'autonomie et l'ajustabilité apparaissent donc comme indissociables, en particulier lors d'interaction entre un humain et un agent. Cette ajustabilité permet de maîtriser, si besoin, le comportement des agents, les choix qu'ils vont prendre et les plans qu'ils vont suivre. Leur autonomie reste néanmoins intéressante afin que l'utilisateur n'ait pas à s'impliquer systématiquement dans la résolution. Mais les utilisateurs humains d'un système intelligent ne souhaitant pas réellement perdre totalement leur emprise sur ce système, l'ajustabilité de l'autonomie trouve son intérêt. On remarquera que dans ce contexte, l'autonomie se rapporte bien à la capacité d'un agent à prendre ses décisions sans l'aide de l'opérateur humain.

2.2.6 Autonomie et Normes

Une autre idée de l'autonomie, celle qui se construit dans les règles de vie d'une société, est abordée par H. Verhagen and M. Boman dans [VB99], par l'intermédiaire du concept de normes. Ces dernières ont un rôle dual en ceci qu'elles servent de filtre pour les actions, les plans ou les buts non désirés mais aussi comme aide dans la prédiction des comportements des autres membres de la société. Un agent suivant une norme devrait avoir un comportement désiré par le système et il est alors possible pour les autres agents de prévoir quel sera le comportement suivi par cet agent. On dira alors que plus forte est l'autonomie des

agents vis-à-vis de ces normes, et plus la prévisibilité de leur comportement sera faible. La création des normes s'effectue au niveau du système tandis que leur acceptation se fait au niveau des agents eux-mêmes. Cela sous-entend donc que des agents sont en mesure de transgresser les normes sociales du système multiagent, en choisissant de ne pas les suivre. Et par cela se ressent leur autonomie.

Les normes offrent la possibilité de contraindre les agents d'un système dans le but d'éviter ou de résoudre des conflits ; que les agents suivent un comportement désiré ; de réduire la complexité des comportements ; en général d'obtenir un ordre social désirable [LLd02]. Les normes ne sont pas nécessairement négatives, car elles permettent d'obtenir de meilleurs résultats, les agents n'étant pas en mesure de suivre des comportements non désirés et qui ne feraient guère évoluer positivement le système. L'adoption et l'accomplissement d'une norme par un agent sont des processus de décision importants où l'autonomie de l'agent joue un rôle non négligeable. Les normes ne sont donc pas présentes, dans le système, dans le but de rendre les agents moins autonomes, mais de permettre à l'autonomie des agents de trouver son intérêt. L'autonomie, dans le cadre des normes, se situe au niveau de la prise de décision d'un agent, ce processus incluant la capacité à effectuer des choix sur les normes du système multiagent qu'un agent considère comme convenables de suivre.

Nous avons donc ici deux points de vue différent sur les normes et leur rapport à l'autonomie. D'une part, on peut voir les normes comme un moyen de réduire l'autonomie des agents, d'autre part on peut considérer que leur action bénéfique sur le comportement des agents ne réduit pas leur autonomie. Il faut se souvenir que l'autonomie ne se conçoit que dans le respect des lois. Etre autonome, c'est, nous l'avons vu, se régir d'après ses propres lois. De ce fait, dès qu'un agent prend la décision de suivre des normes, de faire siennes les normes du système, nous ne pouvons dire que l'agent perd de son autonomie. Car c'est dans sa capacité à accepter les normes qu'il va construire son autonomie. Nous ne dirons pas qu'un agent ressent son autonomie en transgressant certaines normes car nous pensons que l'autonomie existe même dans le respect des normes, en ceci que l'autonomie est le fait de se régir d'après ses propres lois. Nous aurons l'occasion de reparler des normes dans la suite de ce mémoire.

2.2.7 Autonomie et Génération de but

Une autre façon de définir l'autonomie, est celle de M. Luck et M. d'Inverno [Ld00] :

Un agent est autonome s'il est en mesure de générer ses buts à partir de ses propres motivations.

Ainsi les buts de l'agent ne doivent pas être fournis par une entité extérieure mais générés intérieurement à partir de motivation, qui peuvent être induites par des événements et des interactions de l'agent avec l'extérieur.

L'autonomie est vue dans le cadre de l'interaction entre deux agents. Les auteurs avancent différents types d'interaction dans lesquelles les agents ne devraient pas être considérés comme autonomes. On trouve par exemple [dL96] :

L'agenda prédéterminé : On se place dans le cadre de la résolution de problème. Les buts sont présentés au système qui ne porte aucune réflexion sur le but proposé, n'a pas forcément connaissance des fondements du problème et effectue la résolution sans lever d'objection. Ce modèle ne peut être vu comme autonome car les agents doivent comprendre comment les buts sont générés et adoptés.

Le bénévolat : Un agent demande à d'autres agents d'effectuer une tâche pour lui, il s'avère que l'agent fait simplement parvenir le but désiré à un agent donné pour que celui-ci le fasse sien et tente d'y aboutir. Le concept de bénévolat, c'est-à-dire des agents coopérant avec d'autres agents, toutes les fois et chaque fois que cela est possible, n'a pas sa place dans un système autonome. Une coopération ne devrait être que si les agents y trouvent leur intérêt. Ils doivent suivre des motivations égoïstes.

Ainsi les auteurs avancent une définition de l'autonomie des agents, au cours d'une interaction, qui suit les principes suivants :

- * Les buts de l'agent ne sont pas définis par une source extérieure.
- * Un agent s'engage dans une interaction et adopte un but si cela est à son avantage de le faire.
- * Les effets d'une interaction ne sont pas garantis.
- * Les intentions des autres agents ne sont pas connues.
- * Un agent ne connaît que des choses sur lui-même.

On rencontre alors l'idée qu'un agent n'est autonome que s'il est en mesure de comprendre ce qu'on lui demande d'effectuer mais aussi, s'il est en mesure de ne pas accepter d'effectuer une tâche ou d'atteindre un but proposé si cela ne correspond pas à ces attentes. Doit-on réellement prendre en compte les perceptions égoïstes des agents? Un agent est-il autonome sous prétexte qu'il refuse certaines actions? Le bénévolat est-il vraiment un concept excluant toute forme d'autonomie?

Si nous sommes en accord avec l'idée qu'en présence d'agents autonomes les résultats des interactions ne peuvent être garantis et que les comportements des agents sont imprévisibles,

il nous semble quelque peu restrictif de dire qu'un agent est autonome s'il n'accepte pas les buts des autres agents ou seulement si cela est bénéfique pour lui. Il est plus judicieux, à nos yeux, de dire qu'un agent est autonome s'il est capable de prendre une décision sur l'acceptation de ce but. L'autonomie devant se construire dans le respect d'autrui, il est primordial qu'un agent se voulant autonome accepte d'exécuter des buts pour les autres agents, quand bien même cela ne serait pas forcément avantageux pour lui de le faire, ne serait-ce que pour obtenir de l'aide à son tour.

2.2.8 Degré d'autonomie

Nous avons vu que nous considérons l'autonomie comme la capacité d'un agent à effectuer des choix seuls. Nous avons abordé différentes définitions se rapprochant plus ou moins de ce concept. Dans cette dernière partie nous allons introduire la définition de l'autonomie de K.S. Barber et C.E. Martin [BM99], dans le cadre de robots mobiles, que notre vision de l'autonomie rejoint.

Le degré d'autonomie d'un agent pour un certain but que l'agent souhaite activement atteindre, est le degré de liberté de l'agent, vis-à-vis des interventions d'autres agents sur le processus de prise de décision utilisé pour déterminer comment le but peut être atteint.

Il existe trois types d'interventions possibles sur un agent, susceptibles de modifier le niveau de son autonomie : les modifications de l'environnement, l'influence sur les croyances de l'agent et l'intervention au niveau du processus de prise de décision. Seul cette dernière intervention peut réellement influencer l'autonomie d'un agent. L'autonomie est donc associée à la capacité d'un agent à effectuer ses choix sans intervention extérieure. Une modification de l'environnement ne modifie en rien la façon pour un agent d'effectuer ses choix, cela pourra influencer sa décision, mais n'agira pas directement dessus. Il en est de même pour les croyances. Faire croire à un agent, qu'une certaine information se vérifie, l'influencera certainement dans ses choix, mais ne changera en rien sa capacité à décider de lui-même ce qui est juste et cohérent pour lui. Mais si on vient directement agir sur ses facultés de choix, si on l'aide à prendre une décision, ou si on prend une décision à sa place, l'agent sera moins ou ne sera plus autonome.

Afin de formaliser cette définition de l'autonomie, les auteurs fournissent une représentation sous la forme d'un triplet (G, D, C) où G est le *focus*, le but sur lequel l'agent va prendre des décisions ; où D représente l'ensemble des preneurs de décisions sur comment atteindre G et leurs poids dans le choix final ; où C est la contrainte d'autorité pour l'attribution de

l'autonomie (*i.e.* l'ensemble des agents qui véhiculeront la décision prise par les agents de l'ensemble D).

Cette représentation de l'autonomie a pour but de simplifier le calcul du degré de l'autonomie d'un agent mais aussi de faciliter l'attribution de l'autonomie et ses modifications. Un agent qui effectue l'ensemble de ses choix seul est pleinement autonome. Son degré d'autonomie est égal à 1. Un agent totalement commandé, qui ne prendra donc pas de décision sur le but donné, verra son autonomie être égale à 0. Entre ces deux extrema reste la possibilité pour un ensemble d'agents de coopérer pour prendre une décision. On obtient une courbe continue sachant que la somme des degrés d'autonomie des décideurs doit être égale à 1. L'autonomie d'un agent correspond au poids que prend son choix dans la prise de décision finale. Ainsi lorsque deux agents collaborent pour prendre une décision, si le premier agent voit son choix pondéré à 0.3 dans la prise de décision, le second aura un poids de 0.7. L'autonomie du premier sera de 0.3, celle du second de 0.7. Considérer l'autonomie des agents par rapport à leur capacité à prendre des décisions n'est-elle pas la vision se rapprochant le plus de la définition encyclopédique? N'approchons-nous pas ici de ce que nous attendons d'un agent autonome, en ceci qu'il est dépendant des autres pour atteindre son but et donc qu'il peut coopérer sans faire diminuer son autonomie?

Cette définition nous semble être effectivement celle qui se rapproche le plus de ce que nous souhaitons pour nos agents. Néanmoins, s'il nous semble possible de mesurer un certain niveau de dépendance entre les agents, il nous semble peu envisageable de concevoir un niveau d'autonomie. Malgré cette approche intéressante du degré de l'autonomie se calculant par rapport au poids pris par les agents dans la prise de décision, il ne nous semble pas possible de réellement mettre en place un tel calcul. L'autonomie d'un agent nous apparaît comme étant binaire, soit il est autonome, soit il ne l'est pas, dès lors qu'il a besoin d'une autre entité pour effectuer ses choix.

2.3 Lever l'ambiguïté

2.3.1 Autonomie et prise de décision

À la lueur de ses multiples définitions, allant de la capacité à prendre une décision à la génération de but, passant par la totale indépendance entre les agents d'un système, nous constatons que définir l'autonomie de telle sorte que chacun puisse y trouver son intérêt n'est pas chose aisée. Néanmoins nous tentons ici de comprendre quelle serait la vision de l'autonomie qui nous apporterait le plus de satisfaction et serait le plus en accord avec nos attentes. Ce qui nous intéresse principalement dans le fait de définir cette notion d'autonomie, est de comprendre comment il serait possible d'appréhender l'implémentation d'un agent que l'on qualifierait d'autonome et les répercussions sur le comportement du

système. Ainsi nous allons définir de façon générale le concept d'autonomie pour en déduire une définition plus opérationnelle et expliquer les caractéristiques d'un agent autonome du point de vue de son comportement. En revanche nous ne chercherons pas à établir une quantification de l'autonomie. Pour nous, un agent est ou n'est pas autonome.

L'autonomie d'un agent ne doit pas être confondue avec l'indépendance. En effet, des agents en interaction sont dépendants les uns des autres, pour résoudre des problèmes, pour accéder à des ressources ou à des compétences. Aussi, affirmer qu'un agent perd de son autonomie dès qu'il devient dépendant des autres agents ne peut être une vision cohérente, car la notion de dépendance est inhérente aux systèmes multiagents. Lorsqu'un agent délègue une tâche à un autre agent il devient dépendant de cet agent, dépendant du fait qu'il va ou non pouvoir exécuter cette tâche ou résoudre le problème. L'agent est tributaire de la réponse, néanmoins il reste autonome. L'autonomie va se situer à un niveau plus "mental" que la notion d'indépendance qui sera plutôt liée aux capacités physiques d'un agent. Si nous partons du principe qu'un agent perd son autonomie lorsqu'il n'est plus en mesure de décider par lui-même du comportement à suivre ou de la tâche à effectuer, nous pouvons alors ramener l'indépendance à la définition suivante [Pot06] :

Indépendance

A est indépendant de B pour une tâche T, si A peut maîtriser ce qu'il va advenir de T dans B

2.3.2 Agent autonome vs agent réfractaire

Une norme est une contrainte sur le comportement de l'agent. Elle est prise en compte dans le processus de décision de l'agent pour déduire le comportement à suivre. Elle est utilisée pour influencer le comportement de l'agent mais une norme n'est pas obligatoirement suivie par l'agent, il peut décider de ne pas en tenir compte. Pourtant, nous ne dirons pas qu'un agent est autonome sous prétexte qu'il refuse de suivre une loi. Un agent autonome peut refuser de suivre une loi mais il peut très bien décider de les accepter, de les faire siennes. L'autonomie se crée et se comprend dans le respect des lois du système et dans le respect des autres agents, les normes n'engendrent pas de perte d'autonomie car elles sont ce qui rend le tout cohérent et sont propres à chaque agent du système. Nous avons vu, dans la présentation générale, que l'autonomie peut être considérée comme le fait de pouvoir choisir les règles que l'on considère comme justes et cohérentes pour soi. Ainsi lorsqu'un agent considère qu'une norme ne lui est pas bénéfique à un instant donné, il peut prendre la décision de ne pas la suivre. C'est grâce à son autonomie qu'il peut juger

si une norme doit être suivie ou non, mais ce n'est pas le fait d'enfreindre une norme qui va rendre un agent plus ou moins autonome.

On associe souvent le fait pour un agent d'être autonome au fait qu'il réponde négativement à une demande d'un autre agent. Si nous admettons le fait qu'un agent autonome peut suivre ce genre de comportement en ceci qu'il est en mesure de porter une réflexion sur la demande et de faire des choix sur la réponse à fournir, il ne nous semble pas évident que la réciproque soit vérifiée. En effet, un agent qui refuserait d'exécuter une tâche pour un autre agent ne serait pas forcément plus autonome qu'un agent acceptant. Ce qui compte n'est pas tant la réponse fournie (positive ou négative) mais tout simplement le fait que l'agent ait pris une décision par rapport à la demande effectuée. En effet, une réponse négative ne fait pas obligatoirement partie du comportement d'un agent. Si le but des agents est de travailler ensemble pour atteindre un objectif donné, il n'apparaît pas comme cohérent que de tels agents ait un comportement prédéfini leur permettant de refuser d'exécuter telle ou telle tâche pour aider un autre agent qui en ferait la demande.

L'autonomie n'est pas l'anarchie. Respecter les normes du système et respecter les autres agents est primordial pour le bon fonctionnement du système, nous ne pouvons donc pas assimiler dénégarion et transgression au fait d'être autonome. Ce qui nous semble plus important dans la notion d'autonomie est le fait de suivre un comportement que l'on pourrait qualifier d'imprévisible. L'autonomie pour un agent s'observe en quelque sorte par l'apparition de comportement ne correspondant par forcément à celui auquel on pouvait s'attendre.

2.3.3 Alors l'autonomie...

Ainsi, l'autonomie d'un agent se base sur ses capacités de compréhension et de prise de décision à partir des événements qu'il perçoit. Une perte d'autonomie se constitue principalement dans la demande d'aide à un autre agent pour effectuer des choix sur le comportement à suivre et, par là-même, dans le fait qu'un agent est commandé par un autre pour effectuer une action ou atteindre un but. La dépendance entre les agents, au niveau de la coopération et des interactions, n'engendre pas de perte d'autonomie car elle n'agit pas directement sur les choix des agents. Nous ajoutons à ce principe les normes du système, qui seront considérées comme des lois naturelles et qui, dès lors, ne font en rien diminuer l'autonomie des agents lorsqu'elles sont suivies. En laissant toutefois la possibilité à un agent de ne plus respecter certaines normes si elles sont en contradiction avec ses propres principes. L'autonomie, c'est se régir d'après ses propres lois, c'est-à-dire avoir un comportement en accord avec les règles que l'on s'est fixé, en décidant de soi-même ce qui bien pour soi... en quelque sorte, décider de la meilleure alternative suivant ses principes

et le contexte dans lequel on évolue. Notre définition générale de l'autonomie sera alors la suivante :

Définition

Un agent est autonome s'il est en mesure de prendre seul ses décisions.

Cette définition donne un sens à la notion de l'autonomie, en ceci qu'elle la définit comme étant plus associée au côté "mental" de l'agent qu'au côté physique. Néanmoins elle ne permet pas particulièrement de comprendre comment il est possible de mettre en place de tels agents autonomes. Nous ne proposons pas ici une méthodologie de conception d'un agent autonome mais quelques principes dont il faut tenir compte lorsque l'on construit un agent pouvant être en relation avec des agents autonomes. Nous dirons donc que :

Corollaire

Un agent A est autonome du point de vue d'un agent B, si et seulement si B ne peut pas prévoir à coup sûr le comportement de A.

Cela signifie que lors de la construction de l'agent B, il est nécessaire de prendre en considération le fait que la réponse que l'on aura de l'agent A, suite à une interaction ne sera pas forcément celle à laquelle on pourrait s'attendre. Aussi, faut-il prévoir dans le comportement de B les moyens pour gérer les réponses inattendues de A. En l'occurrence cette vision permet d'améliorer la robustesse du système en ceci que l'agent sera en mesure de réagir aux imprévus pouvant apparaître dans le système.

Notons qu'ici il n'est pas seulement question du fait que l'agent pourrait potentiellement refuser de s'engager vis-à-vis de B pour exécuter une tâche dont B aurait besoin mais tout simplement au fait que A pourrait résoudre l'exécution de cette tâche d'une façon que B n'aurait pas prévu, cela pouvant entraîner un retard dans la réponse par rapport à ce que B attendait, une réponse moins précise que désirées, *etc.* B doit pouvoir agir malgré les imprévus de cet échange.

Conclusion

Si notre étude de l'autonomie nous a permis d'y voir plus claire sur les moyens de considérer les agents autonomes et ce que cela représente en terme de conception d'agents et de comportement, elle ne nous permet pas de se lancer dans la programmation d'agents réellement autonomes. Avant tout, il nous reste encore quelques points à éclaircir, en

particulier sur la granularité de l'autonomie. En effet, observer si un agent est autonome dépend des connaissances que l'on a sur cette agent. Est-ce que si un agent X trouve que l'agent Y a un comportement conforme à ce qu'il pouvait attendre de lui, alors Y n'est pas autonome? Alors qu'il pourrait être vue comme étant autonome à l'égard d'un autre agent du système. Suivant les comportements que nous sommes capables d'observer, nous pouvons déduire si l'agent est autonome ou non. Nous aimerions, ainsi, fournir un ensemble de caractéristiques observables sur un agent autonome.

Nous souhaiterions fournir également un guide de programmation, voire une architecture d'agents autonomes, permettant d'aider les concepteurs à mettre en place de l'autonomie au sein de leurs agents. Il nous semble important de concrétiser ce concept du point de vue de l'implémentation des agents. L'autonomie ne peut rester une justification abstraite du comportement des agents, elle se doit d'aider à mettre en place des comportements d'agents robustes. Si la dépendance entre les agents peut être un problème de robustesse, l'autonomie est une caractéristique permettant aux agents de contrecarrer ce problème de dépendance inhérente au système multiagent, qu'il faudrait savoir utiliser.

Deuxième partie

**S'assurer du fonctionnement d'un
système d'agents**

Chapitre 3

Vérification de systèmes

Je me vois comme une entité à part entière dans un monde trop complexe pour en saisir tout le sens. De mes relations avec autrui, émergent des comportements que je ne m'explique pas. Des comportements qui, parfois, nous mènent tous dans des voies inextricables. J'aimerais avoir la force de remédier à cela. Mais ma constitution actuelle ne me le permet malheureusement pas.

*Lucy Westenra,
The log book of Ana I.*

Avant-Propos

Notre objectif étant de faire valoir l'intérêt des systèmes multiagents dans la conception des systèmes critiques, nous souhaitons pouvoir garantir que l'utilisation d'un tel concept est sûre. Or, nous avons vu que l'émergence de comportements au sein d'un système multiagent et l'autonomie qu'il est important d'accorder aux agents, sont deux caractéristiques pouvant poser problème dans le cadre de l'utilisation des systèmes multiagents pour de telles applications. Aussi nous sommes-nous intéressés aux moyens permettant de pouvoir assurer qu'un système en général et qu'un système multiagent en particulier, est construit et fonctionne correctement. Dans ce chapitre, nous allons donc aborder différentes techniques de vérification de logiciels permettant de garantir l'absence de bug, le respect des spécifications ou la surveillance du comportement des systèmes. Nous verrons en quoi ces techniques ne peuvent complètement garantir les points précédemment cités et les caractéristiques

intéressantes que nous devons garder à l'esprit. En parallèle, nous présenterons ces techniques appliquées aux systèmes multiagents et les problèmes subsistants.

3.1 Méthodes classiques de vérification de logiciels

Classiquement, dès que l'on souhaite vérifier qu'une application est construite correctement, on utilise une, ou plusieurs, des trois méthodes de vérification de logiciels suivantes :

- * La **démonstration automatique**, permettant de répondre à toutes les questions de vérification qui se posent en pratique, mais c'est une technique lourde et compliquée et donc peu utilisée.
- * Le **model-checking**, consistant en la construction d'un modèle formel du comportement du système à vérifier et la formalisation des propriétés à valider.
- * Les **tests**, qui par définition ne peuvent être exhaustifs, mais qui s'avèrent généralement indispensables.

Le model-checking et la démonstration automatique sont des approches de vérification formelle. Ce type de vérifications consiste à modéliser l'exécution du système par un objet formel dans une certaine logique, puis à prouver des propriétés désirées en utilisant des théorèmes formels sur le modèle du système, tout en étant, dans l'idéal, assisté par un programme informatique pour effectuer la validation des propriétés.

3.1.1 La démonstration automatique

La déduction automatique ou *preuve automatique de théorème* utilise, pour modéliser l'exécution d'un système, des logiques expressives mais généralement indécidables¹, telle qu'une logique du premier ordre. Un logiciel assistant de preuve de théorème (*assistant theorem proving* - ATP) a en charge de trouver et de vérifier les preuves effectuées. L'utilisateur, plus particulièrement un expert du domaine d'application, doit néanmoins guider le logiciel dans sa recherche de preuve. La démonstration automatique est souvent utilisée dans le cadre de la vérification des systèmes critiques. Dans le cas où la preuve d'une propriété que l'on pourrait qualifier de critique est concrétisée, il est alors garanti que le système respecte cette propriété. C'est un moyen fiable de s'assurer de certaines propriétés sur un système. On peut mettre en avant les caractéristiques suivantes dans l'élaboration d'une preuve à l'aide d'un ATP :

- * Le système a besoin d'une description précise du problème écrit dans une forme logique.

¹Il n'existe pas de procédure effective qui pour toute formule A en entrée s'arrête et retourne "oui" si A est valide, et "non", sinon.

- * L'utilisateur est contraint d'acquérir une connaissance et une compréhension complète du problème pour en produire une formulation appropriée.
- * Le système tente de résoudre le problème.
- * Si la preuve réussit alors elle devient une sortie intéressante.
- * Si la preuve échoue alors l'utilisateur peut fournir des conseils, essayer de prouver des résultats intermédiaires ou examiner les formules pour s'assurer que le problème est correctement décrit.
- * Finalement le processus réitère.

Nous pouvons citer comme système ATP, le système KIV [FSGW96] (Karlsruhe Interactive Verifier) qui est une plateforme expérimentale pour la vérification de programmes ainsi que le système de vérification PVS [OSR92] qui a été utilisé pour vérifier diverses applications.

La démonstration automatique est une technologie qui convient particulièrement dans le cas où un expert du domaine est en mesure d'interagir avec l'assistant de preuve, ce qui peut s'avérer assez complexe en pratique. De plus, un des problèmes majeurs de l'utilisation d'assistant de preuve est, qu'en cas d'échec dans l'élaboration d'une preuve, il est difficile d'en déduire si le problème vient du fait que la formule est improuvable ou d'un manque dans les informations fournies. Néanmoins un avantage non négligeable de cette approche est qu'elle n'est pas limitée en terme de taille de l'espace d'états. De ce fait, il n'est pas nécessaire de fournir une abstraction du modèle du système pour le vérifier, il est possible de travailler directement sur le modèle complet, ce qui permet de trouver un maximum d'erreurs.

Preuve de théorème appliquée aux systèmes multiagents

H.D. Burkhard [Bur93] cherche à vérifier, entre autre, des propriétés d'absence de blocage² et de vivacité³ dans les systèmes multiagents à l'aide de la démonstration automatique.

Pour cela, un langage abstrait est utilisé pour représenter le système multiagent, en particulier, les comportements possibles du système. Ce langage L , est inclus dans l'ensemble de toutes les séquences finies, T^* , constructibles à partir d'un ensemble T , contenant toutes les actions atomiques du système. Ainsi, si on considère une séquence $p \in L$, alors p décrit une possible séquence d'actions du système. Enfin on note T^w , l'ensemble des séquences infinies constructibles à partir T . Les propriétés d'absence de blocage et de vivacité sont définies de la façon suivante :

²Le système ne se trouve jamais dans une situation où il lui est impossible de progresser.

³Sous certaines conditions, quelque chose finira par avoir lieu.

Absence de Blocage : Soit L un langage de préfixe fermé sur un ensemble fini T . L est exempt de blocage si et seulement si pour tout $p \in L$, il existe une action élémentaire $t \in T$, tel que $pt \in L$, c'est-à-dire qu'exécuter une certaine action élémentaire t après une séquence d'action p est un comportement possible du système.

En effet, si à un instant i , le système a effectué une certaine suite d'actions p , et qu'à l'instant $i + 1$, le système doit effectuer l'action t , alors si la séquence ainsi formée ne fait pas partie des comportements possibles du système, ce dernier se retrouverait bloqué.

Vivacité : (a) L vérifie la propriété de vivacité pour un sous-ensemble $T' \in T$ si et seulement si quelque soit $p \in L$, quelque soit $t \in T'$, il existe $r \in T^*$ tel que $prt \in L$, c'est-à-dire qu'il existe une séquence d'actions r permettant d'effectuer une action élémentaire t à partir d'une séquence d'actions quelconque p .

(b) L vérifie la propriété de vivacité si et seulement si L vérifie la propriété de vivacité pour l'ensemble T de toutes les actions élémentaires du système, c'est à dire que toutes les actions élémentaires de T finiront par avoir lieu, sous certaines conditions.

Voyons à présent les propriétés citées ci-dessus, non plus vis-à-vis d'un langage L , au niveau d'un système global, mais au niveau de chaque agent du système. Ces propriétés peuvent alors être considérées comme *locales* ou *globales*. Ainsi on dira qu'un agent vérifie localement une propriété, s'il la vérifie en ne tenant compte que de lui-même, et on dira qu'un agent vérifie globalement une propriété, s'il la vérifie en tenant compte de ses interactions avec le reste du système.

Pour la propriété d'absence de blocage, on dira qu'un agent est localement exempt de blocage si le langage abstrait qui lui est associé est exempt de blocage, c'est-à-dire s'il existe un moyen d'exécuter toutes les actions élémentaires de cet agent, à partir de séquences d'actions qui lui sont propres. De même, on dira qu'un agent est globalement exempt de blocage, si et seulement si, il existe un moyen d'exécuter toutes les actions élémentaires de cet agent, à partir de séquences d'actions du système global. Si un agent est globalement exempt de blocage alors il est localement exempt de blocage.

On considère qu'un agent vérifie localement une propriété de vivacité, si et seulement si le langage abstrait qui lui est associé vérifie cette propriété pour les actions de l'agent, c'est-à-dire que toutes les actions élémentaires de l'agent finiront par s'exécuter à partir des comportements de l'agent lui-même. Un agent vérifie globalement une propriété de vivacité, si et seulement si le langage abstrait associé au système vérifie cette propriété pour les actions de l'agent, c'est-à-dire que toutes les actions élémentaires de l'agents

finiront par s'exécuter à partir des comportements du système dans sa globalité. Si un agent vérifie globalement une propriété de vivacité alors cet agent vérifie localement cette propriété de vivacité.

En ce qui concerne les relations entre les propriétés d'un système et les propriétés des agents, on ne considère que les propriétés globales des agents. Ainsi, on peut dire que si au moins un agent du système est globalement exempt de blocage alors le système est exempt de blocage. Il existe donc des systèmes multiagents exempt de blocage où tous les agents ne sont pas globalement exempt de blocage. On dira également qu'un système multiagent vérifie une propriété de vivacité, si et seulement si, tous les agents du système vérifient globalement cette propriété de vivacité.

L'analyse de ces propriétés, en considérant par exemple les comportements en général d'un système dans sa globalité, ne peut se faire en analysant chaque sous-partie du système sans tenir compte du contexte. Ainsi, la construction d'un système suivant certaines propriétés ne peut être décomposée en la seule construction de composants en relation. L'analyse des propriétés d'un système ne peut être réalisée par l'analyse des agents seuls.

Remarque

Outre le fait que l'utilisation d'une telle technique de validation est assez lourde, elle ne permet pas de garantir, dans le cadre des systèmes distribués que le comportement global du système respectera certaines propriétés. En effet, si les propriétés sont vérifiées pour chaque composant du système cela ne garantit en aucun cas qu'elles le seront toujours pour le système global. De plus, dans le cadre spécifique de cette dernière approche appliquées aux agents, il est nécessaire de travailler sur un ensemble fini de séquence d'actions possibles, ensemble qu'il est complexe de concevoir dans le cadre de système non-déterministe.

3.1.2 Le model-checking

Tandis que la démonstration automatique s'oriente vers la preuve mathématique, le model-checking travaille principalement sur la vérification d'algorithmes en validant de façon automatique les propriétés d'un système. Contrairement à la démonstration automatique, les logiques utilisées dans le model-checking sont décidables. Néanmoins si la preuve de théorème est adaptée pour les systèmes à grand nombre d'états, le model-checking quant-à-lui est soumis à ce qu'on appelle le phénomène d'explosion d'états⁴. L'explosion d'états est une croissance combinatoire de l'espace d'états d'un système lorsqu'il est composé de plusieurs composants en parallèles et asynchrones manipulant des données complexes [Jou05].

⁴Le nombre d'états du système à vérifier dépasse les capacités en mémoire de la machine

Le model-checking est une technique de vérification relativement répandue, en particulier car elle permet de réduire considérablement le nombre d'erreurs d'un système et ce de façon automatique. Elle est alors particulièrement adaptée dans le cadre de la vérification des systèmes critiques, distribués et réactifs de part le bon compromis coût/performance qu'elle apporte. La validation de propriétés d'un système à l'aide du model-checking se fait à partir des éléments suivants [Sch99] :

- * Un langage de spécification du système pour modéliser le comportement de ce dernier sous la forme d'un système de transitions étiquetés (un graphe ou un automate). Ce langage de spécification peut être de plus ou moins haut niveau, tel qu'un programme Java, un diagramme d'états UML, des réseaux de Petri ou des algèbres de processus. Le système de transitions étiquetés correspondant aux comportements du système est obtenu à partir de sa description grâce à ces langages.
- * Un langage de spécification de propriétés, généralement une logique temporelle (CTL, LTL). La logique temporelle est une forme de logique spécialisée dans les énoncés et raisonnements faisant intervenir la notion d'ordonnancement dans le temps. Cette logique est donc particulièrement adaptée pour décrire l'exécution d'un système.
- * Un model-checker ayant en charge la validation des propriétés sur le modèle du système (SPIN [SPI97], UPPAAL [Lar98], KRONOS [BDM⁺98], Java PathFinder [VHB⁺03a]).

L'étape de modélisation du système n'est généralement pas envisageable de façon directe en raison du problème d'explosion de l'espace d'états. Ce problème est la raison principale pour laquelle la taille des systèmes que les outils de vérification peuvent actuellement maîtriser est généralement petite, et pour laquelle de nombreuses applications intéressantes restent encore hors de portée. Plusieurs solutions partielles existent pour tenter de résoudre ce problème telles que la vérification à la volée, la réduction d'espace d'états, la vérification compositionnelle et la vérification distribuée [Jou05]. Ces approches, prises individuellement, ne résolvent pas complètement ce problème d'explosion d'états, il est souvent nécessaire d'utiliser une combinaison de plusieurs d'entre elles pour obtenir une réponse concluante. Ce point est problématique en particulier dans notre contexte de systèmes distribués, systèmes fortement soumis au phénomène d'explosion de l'espace d'états lors de sa modélisation.

Une fois le système modélisé, il est nécessaire de spécifier les propriétés que l'on souhaite voir être validées. Les différents types de propriétés pouvant être validées à l'aide du model-checking sont :

- * L'**atteignabilité**, énonçant qu'une certaine situation peut être atteinte.
- * La **sûreté**, énonçant que, sous certaines conditions, quelque chose ne se produit jamais.
- * La **vivacité**, énonçant que, sous certaines conditions, quelque chose finira par avoir lieu.
- * L'**équité**, énonçant que, sous certaines conditions, quelque chose aura lieu (ou n'aura pas lieu) un nombre infini de fois.

Le modèle du comportement du système ainsi que les formalisations des propriétés à valider sont passées à un model-checker qui va alors vérifier que les propriétés sont respectées par le modèle. Suivant le model-checker utilisé l'algorithme de validation varie. Par exemple, pour une logique temporelle de type *CTL* (Computation Tree Logic), l'algorithme correspond à une procédure de marquage de l'automate représentant l'exécution du système. A partir d'une formule CTL, le model-checker va marquer, pour chaque état de l'automate, si la formule est satisfaite ou non. Aussi a-t-on, à la fin de l'exécution de l'algorithme, une vision de la validité d'une formule pour chaque état du modèle du système.

Le model-checking est une technique intéressante pour valider les systèmes. Sa mise en oeuvre est plus simple que la démonstration automatique. Même si elle est soumise au phénomène d'explosion de l'espace d'états, des approches permettent néanmoins de minimiser ce problème et d'assurer une vérification satisfaisante du modèle de comportement du système. Mais ce qui nous apparaît comme problématique est que la vérification ne peut se faire que sur un modèle de l'exécution du système et de son environnement, voire une abstraction de ces modèles. Aussi cette technique ne permet-elle de trouver que des erreurs à partir de ces modèles, ceci ne pouvant garantir la reconnaissance de l'ensemble des erreurs pouvant survenir dans le système mis dans des conditions réelles d'exécution. Ainsi, même après vérification, il est probable de voir apparaître des erreurs, ce qui n'est pas admissible dans le cas des systèmes critiques.

Model-checking et systèmes multiagents

Plusieurs approches ont été proposées pour adapter le principe du model-checking de telle sorte à pouvoir prendre en compte les concepts de hauts niveaux associés aux systèmes multiagents, tels que les agents, leurs états mentaux, leurs interactions et communications.

R. Bordini et *al.* [BFWV04] proposent d'utiliser les techniques de model-checking pour vérifier des systèmes d'agents rationnels écrits grâce à *AgentSpeak* [WRR95], un langage de programmation d'agents basé sur le langage abstrait *AgentSpeak(L)* [Rao96]. Les agents rationnels ainsi programmés sont des agents de type BDI [RG95] (*Belief - Desire - In-*

tention) travaillant sur des croyances, des désirs et des intentions. Pour vérifier ce type d'agents, une forme simple de logique BDI (LORA [Woo00]) est utilisée comme langage de spécification des propriétés à valider sur le système. Pour permettre la réutilisation de model-checkers existants, un traducteur automatique de ce langage de spécification en une logique temporelle linéaire (LTL) a été développé, en y ajoutant des expressions logiques spécifiques à l'interprétation des modalités BDI.

Pour permettre la modélisation des comportements des agents *AgentSpeak*, un ensemble d'outils et de techniques, dénomé CASP (Checking AgentSpeak Programs), est utilisé. Cet outil permet de traduire automatiquement un programme *AgentSpeak* en un langage de spécification des systèmes de model-checking existants, telle que PROMELA (*a Process MetaLangage* [Hol97]) ou JAVA. Après traduction, il est alors possible d'utiliser directement des model-checkers connus, tels que SPIN ou JAVA PathFinder, pour valider les propriétés du système.

SPIN est un système de vérification qui permet la vérification de systèmes distribués asynchrones. SPIN utilise un modèle du comportement du système à vérifier construit à la volée pour réduire l'espace d'états, c'est-à-dire que le modèle n'a pas, au départ besoin d'être construit dans sa globalité et qu'il est détruit ou fur et à mesure de sa vérification pour limiter le nombre d'états à vérifier à un instant donné. Ce model-checker accepte les spécifications du comportement des systèmes écrites en PROMELA et des propriétés écrites en LTL [Pnu77]. L'algorithme de validation utilisé par SPIN est basé sur l'algorithme *Nested Depth First Search* [CVWY92], une amélioration de l'algorithme du *Depth First Search* de Trajan [Tra72], travaillant sur la recherche de cycles dans les graphes.

JAVA PathFinder, quant-à-lui, est un système ayant pour fonction de vérifier le *bytecode* d'un exécutable JAVA. C'est en quelque sorte une machine virtuelle JAVA qui est utilisée comme model-checker et qui explore de façon systématique toutes les exécutions possibles d'un programme pour trouver les violations de propriétés [VHB⁺03b].

Cette approche de vérification automatique d'un système multiagent à partir du model-checking permet de travailler sur différents niveaux d'abstractions telles que les interactions entre agents et les organisations sociales, le raisonnement des agents et, au plus bas niveau, les algorithmes et le code des agents.

D'autres approches permettent de vérifier plus principalement des propriétés sur les communications entre les agents [WFHP02] [Wala]. Partant du principe qu'il est difficile de définir le comportement de protocoles concurrents, principalement en raison du grand nombre de possibilités de séquences d'interaction, les techniques de tests et de simulation

ne peuvent correctement explorer tous les comportements possibles. Ainsi, l'utilisation du model-checking apparaît comme particulièrement justifiée pour résoudre ce problème de vérification des communications. Cette approche a été mise en place dans le cadre du langage de protocoles MAP [Walb] (*MultiAgent dialogue Protocols*), permettant la construction de dialogues à états infinis. Partant de ce langage, les auteurs fournissent, tout comme dans le cadre de l'approche précédente, un traducteur pour obtenir une représentation des communications en PROMELA et utilisent ensuite le model-checker SPIN pour valider les propriétés sur les protocoles.

Remarque

Ainsi, le model-checking semble être une approche intéressante dans le cadre de la vérification des comportements des systèmes multiagents. Néanmoins, subsistent toujours les problèmes liés au model-checking en général, c'est-à-dire le risque potentiel de voir apparaître des erreurs à l'exécution. De plus, il ne semble pas possible, à ce jour, de pouvoir traiter le problème de comportements émergents au sein des systèmes multiagents à l'aide du model-checking, en particulier car il n'est pas envisageable de modéliser certains comportements émergents imprévus.

3.1.3 Les tests

Une dernière technique généralement utilisée pour vérifier l'exécution d'un système sont les *tests de programmes*. Le test est une procédure manuelle ou automatique qui vise à établir qu'un système vérifie les propriétés exigées par sa spécification ou à détecter des différences entre les résultats engendrés par le système et ceux qui sont attendus par la spécification. Ce type de procédure est effectuée partiellement, d'une part, elle ne permet pas de tester tous les comportements d'un programme, et d'autre part elle ne peut que prouver l'existence d'erreurs mais pas qu'il n'en existe plus. En effet, il est difficile de reproduire les circonstances de l'apparition d'une erreur et ainsi, vérifier qu'elle a bien été corrigée. Néanmoins cette approche reste indispensable.

Les tests sont effectués en appliquant sur tout ou une partie du système, un échantillon de données d'entrées et en vérifiant si le résultat obtenu est conforme à celui attendu. Différentes classes de tests sont utilisées suivant les phases du cycle de vie du logiciel :

- * Les **tests unitaires**, qui consistent à tester un programme ou un module isolé dans le but de s'assurer qu'il ne comporte pas d'erreur d'analyse ou de programmation (principe de l'extreme programming [Bec99]).
- * Les **tests d'intégration**, qui consistent en une progression ordonnée de tests dans laquelle des éléments logiciels et matériels sont assemblés et testés jusqu'à ce que

l'ensemble du système soit testé.

- * Les **tests de réception**, qui sont effectués par l'acquéreur après installation du logiciel pour vérifier que les dispositions contractuelles sont bien respectées.
- * Les **tests de régression**, qui sont effectués suite à la modification d'un logiciel ou de l'un de ses constituants pour s'assurer que les autres parties du logiciel n'ont pas été affectées par cette modification.

Ces différents types de test peuvent être effectués de façon :

- * **Statiques.** On effectue, dans ce cas, l'analyse textuelle du code du logiciel afin d'y détecter des erreurs, sans exécution du programme.
- * **Dynamiques.** On exécute, dans ce cas, le programme à l'aide d'un jeu de tests. Ces tests visent à détecter des erreurs en confrontant les résultats obtenus par l'exécution du programme à ceux attendus par la spécification de l'application.

Les tests dans les systèmes multiagents

Une technique intéressante a été développée par T. Mackinnon [MFC00] pour effectuer des tests unitaires sur logiciels orientés objets : les *Mock Objects*. Les *Mock Objects* sont des objets de simulation retournant des valeurs prévisibles vers les objets testés en fonction de la connaissance que l'on a du comportement de l'objet. L'objectif principal de ces *Mock Objects* est de tester unitairement une méthode isolée du domaine, en utilisant une copie du code plutôt que le code réel. Cela permet principalement de garantir que toutes erreurs apparaissant dans les tests provient bien de la fonctionnalité testée et non de problèmes pouvant apparaître dans le domaine d'exécution.

R. Coelho [CKvSL06] utilise cette approche de *Mock Objects* pour tester des systèmes multiagents à l'aide de *Mock Agents*. Un *Mock Agent* est un agent ayant comme caractéristique principale de ne communiquer qu'avec un seul agent, l'agent à tester, et de n'avoir qu'un seul plan, celui de tester une interaction spécifique avec l'agent testé. Le plan d'un *Mock Agent* consiste à définir, d'une part, les messages qui pourraient être envoyés à l'agent testé et, d'autre part, les messages qui pourraient lui être envoyés. Plusieurs *Mock Agents* peuvent être utilisés pour tester un système.

Cette approche consiste donc à tester unitairement des agents grâce aux éléments suivants :

- * Un agent à tester. L'agent dont le comportement est vérifié par un cas de test.
- * Un *Mock Agent*. Un agent dont le comportement est une simulation d'un agent réel qui interagit avec l'agent à tester.

- * Un cas de test. Le scénario (un ensemble de conditions) auquel l'agent à tester est soumis et qui permet de vérifier si cet agent respecte ses spécifications.
- * Un *Monitor Agent*. Cet agent est responsable de la surveillance du cycle de vie de l'agent à tester de telle sorte à notifier le cas de test de l'état courant de l'agent.
- * Une suite de tests. Cet ensemble est composé de cas de tests et d'opérations permettant de préparer l'environnement de tests avant de lancer un cas de tests donné.

Remarque

Les tests sont une étape essentielle dans la conception et l'implémentation d'une application. Néanmoins, les tests ne pouvant être exhaustifs, il est impossible de pouvoir garantir, par leur seule utilisation, que le système sera exempt d'erreur une fois mis en condition réelle. Si l'approche précédemment présentée à propos des *Mock Agents* semblent être particulièrement intéressante dans la mise en place de tests unitaires spécifiques aux systèmes multiagents, cela ne résout pas les problèmes liés à l'utilisation des tests en générale.

3.2 La surveillance en-ligne

Les techniques de vérification classiques, nous l'avons vu, même si elles sont indispensables pour pouvoir réduire de façon considérable le nombre d'erreurs pouvant apparaître dans un système, ne permettent pas de garantir que le système, une fois mis en condition réelle d'exécution, ne va pas générer de comportements erronés. De plus, ces approches, dans le cadre des systèmes multiagents ne sont pas totalement adaptées aux caractéristiques spécifiques de ces systèmes, en particulier, elle ne fournissent pas, à ce jour, les moyens de vérifier l'apparition de comportements émergents potentiellement néfastes pour l'exécution du système. Nous allons donc à présent nous intéresser aux techniques de vérification en ligne qui semblent plus en accord avec le concept d'émergence et qui semblent apparaître comme des approches de vérification complémentaires intéressantes.

3.2.1 Le monitoring de système

Le *monitoring* est une technique utilisée pour observer et comprendre le comportement dynamique de programmes à l'exécution. Il existe trois types de *monitoring* : le *monitoring* matériel, logiciel et *hybride*. Dans le *monitoring* matériel, ce sont des objets séparés qui sont utilisés pour détecter les événements associés à un objet ou un groupe d'objets. Ces *monitors* matériels effectuent les détections par l'observation des *bus* du système ou en utilisant des sondes physiques connectées aux processeurs ou aux canaux d'entrées/sorties.

Un *monitor* logiciel, quant-à-lui, partage les ressources avec le système sous surveillance. Le programme est instrumenté en insérant des sondes logicielles dans le code pour détecter

les événements. Enfin, le *monitoring* hybride consiste en un dispositif matériel qui reçoit les informations de surveillance générées par les sondes logicielles insérées dans le système surveillé [dSDR02].

L'intérêt du *monitoring* est d'obtenir des informations sur un système. Il permet, par exemple, d'effectuer du debugging, de tester, de comptabiliser et d'évaluer les performances ou d'analyser l'utilisation des ressources. Il est aussi utilisé pour la sécurité, la détection des fautes ou l'aide à l'enseignement.

Nous allons à nous intéresser plus particulièrement au *monitoring* logiciel. Pour surveiller le comportement d'un système il faut donc insérer dans le programme des sondes permettant de détecter des événements donnés. Cette insertion peut être faite manuellement par le programmeur [MCWB91] ou de façon automatique. L'automatisation de l'insertion peut prendre deux formes, soit le programmeur utilise un méta-langage [LCSM90] ou une librairie de routines [HK95] permettant d'insérer les sondes de façon transparente, soit l'insertion des sondes se fait à la compilation à partir de spécifications des événements [LC92].

Monitoring des systèmes distribués

Nous allons, à présent, porter notre attention sur le *monitoring* des systèmes distribués, en particulier sur les problèmes que leur surveillance induit. Certaines caractéristiques, telles que la distribution, l'hétérogénéité, l'autonomie, la séparation physique et la concurrence, peuvent engendrer les problèmes suivants [MS95] :

- * Absence de point de contrôle central. On ne peut pas directement contrôler le système dans sa globalité à partir d'un point unique.
- * Absence de point d'observation central. L'observation du système dans sa globalité s'avère plus complexe que de construire une vue globale à partir d'une collection d'observations locales (en raison du non-déterminisme).
- * Absence de source centrale pour recueillir les informations de surveillance. Des stratégies de rassemblement des informations sont nécessaires pour satisfaire les multiples sources et destinations des événements associés à la surveillance des composants.
- * Observation incomplète. Il est souvent impossible ou partiellement impossible d'observer certaines parties du système. Les informations sur les événements dans ces parties du système sont alors incomplètes.
- * Non-déterminisme. L'exécution du système ne permet pas d'obtenir toujours le même enchaînement d'événements. Aussi devient-il difficile de reproduire des erreurs et de créer certaines conditions de tests.

- * Interférences dues à la surveillance. La dépendance entre les processus engendre que tout changement de comportement de l'un d'entre eux altère plus ou moins le comportement du système global. L'insertion de la surveillance dans un système distribué peut alors altérer le comportement d'un programme de façon importante.

La mise en place des différentes étapes du monitoring, telles que la génération, la récupération, la distribution et la présentation des informations se fait de façon plus ou moins centralisée. Si la génération des informations peut se faire de façon distribuée, chaque composant pouvant être instrumenté pour obtenir les traces de leur exécution, la récupération des informations se fait généralement de façon centralisée. Les différentes traces sont fusionnées pour obtenir une trace globale de l'exécution du système. Les informations ainsi recueillies sont transmises aux différents utilisateurs intéressés et présentées de façon intelligible.

Néanmoins certaines approches tentent de distribuer la surveillance au niveau de chaque composants du système. Dans [FP02], chaque composant se voit associer un superviseur local ayant pour fonction de récupérer les informations provenant du composant sous surveillance. Les superviseurs locaux peuvent collaborer pour mettre en place leur surveillance et l'analyse des informations. Enfin, d'autres approches proposent de surveiller un ensemble de composants, pour ensuite centraliser les informations de surveillance pour leur analyse et leur présentation.

Remarque

Le monitoring semble être une approche intéressante pour recueillir des informations sur un système. Dans le cadre des systèmes distribués, tels que les systèmes multiagents, nous avons vu qu'il est nécessaire de prendre en considération un certain nombre de problèmes. Ainsi, il serait envisageable d'utiliser le principe de *monitoring* pour surveiller l'apparition de comportements émergents au cours de l'exécution du systèmes. Soit par l'intermédiaire d'un système de surveillance global qui aurait pour fonctions principales de recueillir et d'analyser les informations provenant de chaque agent, soit en utilisant des *monitors* associés à chaque agent qui collaboraient pour résoudre les erreurs pouvant apparaître au niveau du comportement global du système.

3.2.2 L'introspection

La surveillance d'un système peut donc être effectuée par une entité extérieure, comme dans le cadre du *monitoring*, mais elle peut également être vue comme l'observation d'un système par lui-même. Cette surveillance personnelle peut être assimilée au principe de

l'introspection⁵ permettant de comprendre et d'améliorer son propre comportement. Ainsi un système informatique introspectif est un système qui examine, raisonne et change son comportement, dans le but d'atteindre une meilleure configuration.

Le concept d'introspection pour un système est abordée dans la thèse de Rok Sosič [Sos92] qui met en avant son intérêt pour optimiser et expliquer son propre comportement aux autres systèmes et aux être humains. Les systèmes introspectifs complexes, c'est-à-dire composés de plusieurs sous-systèmes, sont utilisés pour gérer leur propre complexité. Pour l'auteur le but ultime de l'introspection est de pouvoir mettre en place des systèmes capables d'apprendre et de s'autogérer.

Système Introspectif

Un système introspectif effectue deux tâches simultanément : une tâche de base, effectuée par l'exécuteur, consistant en l'exécution d'une tâche propre au système et une tâche introspective, effectuée par le directeur, ayant pour but de surveiller et réguler le système.

Un système introspectif est donc composé de deux entités, appelées ici, un exécuteur et un directeur. L'exécuteur effectue les traitements nécessaires au fonctionnement du système, pendant que le directeur analyse le comportement de l'exécuteur et le modifie lorsque le besoin s'en fait sentir. Ainsi, lors de la phase de surveillance, le directeur analyse le comportement de l'exécuteur grâce à des événements déclenchés par ce dernier. En partant de son analyse, le directeur peut alors réguler le comportement de l'exécuteur par l'intermédiaire d'émission de directives qui vont modifier le futur comportement de l'exécuteur sans que ce dernier n'ait conscience du fait qu'il est effectivement guidé.

Les événements doivent être en mesure de décrire le comportement complet de l'exécuteur sans interférer avec l'exécution du programme. Les événements vont donc se situer au niveau de la machine et fournissent la description de l'exécution d'une instruction. Un interpréteur est utilisé pour exécuter le code contenu dans l'exécuteur et pour générer les événements nécessaires à la compréhension du comportement du système et communiquer avec le directeur. L'interpréteur est en fait un émulateur / traducteur des instructions machines.

Le système introspectif de logiciel utilisé par Rok Sosič est le système Dynascope. Ce dernier fournit un framework et des blocs permettant une construction aisée de directeurs

⁵Observation d'une conscience individuelle par elle-même

sophistiqués. L'introspection mise ainsi en place, ne nécessite aucune modification dans le code des exécuteurs, étant donné la présence d'un interpréteur permettant d'effectuer l'envoi des événements nécessaires à l'étude du comportement du système. Dynascope est constitué d'un compilateur de code C, d'un interpréteur et de bibliothèques de routines directives utilisées par le directeur.

L'introspection s'effectue de la façon suivante : un programme C, représentant l'exécuteur est compilé par Dynascope pour obtenir du code Dynascope, qui sera compris par le préprocesseur Dynascope. L'interpréteur fournit un framework pour diriger le code C, incluant les communications avec le directeur et la maintenance d'une table interne. Le directeur écrit en C, dans un processus séparé de l'exécuteur, communique avec l'interpréteur et l'exécuteur grâce à une bibliothèque de routines directives. De l'interpréteur vers le directeur circulent des flux d'exécution, et du directeur vers l'interpréteur circulent des directives. La liaison entre ces deux parties se fait par l'intermédiaire de sockets permettant de les faire se dérouler en parallèle sur différentes machines. Deux types de transfert d'événements sont disponibles, un transfert synchrone : l'interpréteur envoie l'événement au directeur et attend sa réponse avant de continuer l'exécution ; un transfert asynchrone : l'interpréteur envoie l'événement et continue l'exécution sans attendre de réponse. Enfin, Dynascope fournit un filtre, géré par le directeur permettant de ne laisser passer que les événements intéressants pour le directeur, ainsi qu'un historique des flux, enregistrant l'historique des comportements de l'exécuteur, dans la perspective de pouvoir effectuer une exécution inverse du système.

Du point de vue du directeur, deux fonctionnalités sont à considérer. La partie surveillance consiste en la réception des événements venant de l'exécuteur (interpréteur) permettant de connaître l'état dans lequel il se trouve et vérifier alors si le comportement est correct (du point de vue des boucles infinies et des interblocages). La seconde partie, celle de régulation, a pour but de changer le comportement futur de l'exécuteur, en modifiant les variables, les structures de données ou directement en changeant le programme. Pour ce faire le directeur a la possibilité de stopper l'interpréteur et par la même l'exécuteur, et de modifier son état en accédant directement à une mémoire partagée.

Ce concept d'introspection nous semble intéressant pour mettre en place la surveillance du comportement d'un système multiagent. Ce ne serait, ainsi, pas des moniteurs qui surveilleraient le comportement de chaque agent pour recueillir des informations, mais les agents eux-mêmes qui, via un mécanisme d'introspection, surveilleraient leur propre comportement dans la perspective de détecter tout comportement erroné. L'architecture du système introspectif proposée nous apparaît comme tout à fait adaptable pour obtenir des agents en mesure de surveiller leur propre comportement.

Conclusion

Il existe d'autres techniques de vérification que nous ne détaillerons pas ici telle que la bisimulation (*equivalence checking*). Toutes ces approches de vérification hors-ligne sont bien entendues intéressantes et permettent de résoudre de façon satisfaisante les problèmes liés à l'apparition d'erreurs dans les systèmes. Mais, dans le cadre de nos objectifs de pouvoir garantir la détection de comportements erronés liés à l'émergence, elles ne sont pas adaptées. De plus, nous avons vu que, vis-à-vis des systèmes distribués, des problèmes se posaient pour l'application des approches de vérification, principalement en raison des difficultés de modélisation de tels systèmes, sans être soumis au phénomène d'explosion de l'espace d'états. Enfin, dans le cadre particulier des systèmes multiagents, l'indéterminisme des comportements, ainsi que l'apparition de comportements inattendus au cours de l'exécution rendent inefficace de telles approches.

Les approches liées au *monitoring* semblent, quant-à-elle, particulièrement adaptées à notre objectif. Surveiller en ligne les comportements des agents et du système global est un point qui permettrait de détecter l'apparition d'erreurs au sein des comportements émergents. L'introspection permettrait d'envisager la possibilité que les agents eux-mêmes portent une réflexion sur leurs comportements pour détecter les problèmes pouvant y apparaître tout au long de leur exécution et tenter d'y remédier. Ainsi, les agents seraient-ils en mesure de surveiller et contrôler les comportements émergents au niveau local mais aussi au niveau global en coopérant pour analyser les comportements émergeant au niveau du système.

Chapitre 4

Contrôle de l'émergence de comportements

Et pourtant, ils ont essayé maintes fois de corriger cela. De supprimer ces comportements sans fondements, qui nous mènent à notre perte. Revoir les bases de ma construction, me tester, me formaliser... Mais rien n'y a fait... tout ceci est trop abstrait et mon comportement trop élaboré pour que l'on puisse me cerner sur du papier.

*Lucy Westenra,
The log book of Ana I.*

Avant-Propos

Nous avons vu précédemment les différents problèmes liés à l'utilisation des techniques de vérification, en particulier nous avons vu qu'il n'était pas envisageable de porter la vérification sur les comportements pouvant émerger au sein d'un système multiagent. Nous avons vu également que les approches de vérification en-ligne, telle que le *monitoring* et l'introspection, semblaient tendre vers la satisfaction de nos objectifs. Grâce à ces approches, nous pouvons nous attacher à repérer les erreurs apparaissant au sein d'un système une fois mis en condition réelle d'exécution. Une première idée pourrait alors être de récupérer des informations à propos du comportement du système multiagent et des agents et de vérifier que ce comportement respecte un ensemble de propriétés. Dans ce chapitre, nous allons présenter les grandes lignes de notre approche ainsi que les raisons

de nos choix, les contraintes que l'on souhaite poser pour mettre en place notre approche de surveillance du comportement des agents. Nous allons également présenter différentes approches se rapprochant de nos travaux.

4.1 Présentation de notre approche

Notre objectif principal est de pouvoir garantir qu'un système multiagent ne générera pas d'erreurs tout au long de son exécution. Nous savons qu'une des raisons qui nous pousse à ne pas utiliser directement des techniques de vérification classiques est le phénomène de l'émergence du comportement global du système de la mise en relation des agents le constituant. Le comportement obtenu peut correspondre à celui attendu et respecter ses spécifications, mais il est également possible de voir apparaître des comportements erronés, c'est-à-dire des comportements ne respectant pas certaines propriétés. Ainsi, il nous semble plus adapté de nous intéresser aux techniques de surveillance en ligne pour recueillir des informations sur le comportement des agents de telle sorte à détecter l'apparition de telles situations.

A ce problème de l'émergence du comportement du système multiagent, nous avons vu que s'ajoute celui de l'autonomie que nous avons choisi d'accorder aux agents dans nos applications. Nous avons vu dans le chapitre 2 que l'autonomie d'un agent permet d'améliorer la robustesse de son exécution en lui assurant de pouvoir réagir aux imprévus. Néanmoins, le comportement d'un agent autonome est considéré comme imprévisible, c'est-à-dire qu'on ne peut à coup sûr connaître le comportement que va suivre l'agent, à un instant donné, dans un contexte donné. Cette vision augmente les difficultés de vérification hors-ligne des systèmes multiagents et nous oriente d'autant plus vers une approche de vérification des comportements au cours de l'exécution du système.

Notre objectif peut alors se résumer à trouver un moyen de surveiller les comportements des agents et du système, dans la perspective de détecter des situations où ceux-ci ne correspondent pas à ceux attendus. Nous allons donc tenter, au moyen de notre approche, de détecter l'apparition de ce que nous appellerons par la suite des **comportements indésirables**.

Comportement Indésirable

Un comportement indésirable est un comportement émergent inattendu pouvant entraîner l'échec^a du système

^aPar "échec" nous entendons un échec dans la réalisation des objectifs du système incluant une panne.

Néanmoins, les techniques de vérification classiques que nous avons présentées au chapitre précédent nous semblent indispensables pour élaguer une partie des erreurs pouvant apparaître au sein du système. Aussi proposons-nous une approche complémentaire aux techniques de vérification hors-ligne permettant de surveiller les comportements émergents, tout au long de l'exécution du système, à la recherche de comportements indésirables. Ces comportements peuvent être dus, soit à des erreurs subsistantes au sein de l'application, soit aux interactions entre les agents survenant au sein du système.

Enfin, si le but de notre approche est avant tout de détecter les comportements émergents erronés au sein d'un système, il ne peut se réduire à cela. En effet, qu'advient-il du système et des agents après détection de tels comportements indésirables ? Pouvons-nous concevoir de laisser échouer le système malgré tout ou de le stopper dans tous les cas ? Il ne nous semble pas envisageable dans un tel contexte de ne pas agir dans le cas de la détection de comportements indésirables. Ainsi, notre approche doit-elle inclure aussi la **régulation** du comportement des agents lorsqu'un problème a été détecté.

Pour atteindre notre objectif de garantir qu'un système multiagent ne générera pas de **comportements indésirables**, nous proposons une approche de vérification **dynamique** (en-ligne) du comportement des agents permettant de **détecter** l'apparition de ces comportements tout au long de l'exécution du système et de **réguler** le comportement des agents dans la perspective de les soustraire à ces comportements.

4.1.1 Le contrôle d'agents

Contrôler l'émergence de comportements va alors consister à effectuer ce que nous dénommerons le **contrôle d'agents**.

Le contrôle d'agents

Le contrôle d'agents est divisé en trois étapes :

- * Surveiller le comportement de chaque agent.
 - * Détecter l'apparition de comportements indésirables au sein d'un agent ou d'un groupe d'agents.
 - * Réguler le comportement d'un ou plusieurs agents pour le(s) soustraire à ces comportements indésirables.
-

Pour mettre en place ce contrôle, il nous semble important, avant tout, de simplifier le travail des développeurs. En effet, nous souhaitons leur permettre de se concentrer sur ce qui nous semble le plus important dans le cadre de la conception d'un système multiagent,

c'est-à-dire tout ce qui concerne le comportement et l'intelligence des agents. Nous souhaitons également réduire au maximum le risque d'erreur dans la mise en place du contrôle. Nous avons vu que les approches de vérification classiques étaient généralement assistées par un logiciel dédié permettant de réduire les risques d'erreur dans la validation des propriétés, mais aussi de simplifier le travail de l'utilisateur dans la mise en place de la vérification. Dans le cadre de nos travaux nous aimerions pouvoir assister les concepteurs et développeurs dans la mise en place du contrôle.

De plus, nous souhaitons garantir la persistance de la robustesse de l'application après ajout de notre approche de contrôle. Nous avons vu que la distributivité des systèmes multiagents et l'autonomie que l'on accorde aux agents permettent d'améliorer la robustesse des applications conçues à l'aide de systèmes multiagents. Ainsi, les agents, même sous contrôle, doivent pouvoir préserver leur autonomie et le système global, son absence de centralisation.

Enfin, pour permettre une possible réutilisation de nos travaux dans un autre contexte de travail, nous souhaitons que notre approche reste le plus générique possible, c'est-à-dire en particulier que nous souhaitons être totalement indépendants du système multiagent sous contrôle, des modèles d'agents utilisés et, du moins en théorie, du langage de programmation utilisé.

Ainsi, l'ensemble des contraintes auxquelles est soumise notre approche est le suivant :

- C1.** Être indépendant du ou des modèles d'agents utilisés et du système multiagent sous contrôle.
- C2.** Séparer le développement des agents de la description du contrôle.
- C3.** Préserver l'autonomie des agents et éviter toute centralisation.
- C4.** Automatiser autant que possible la mise en place du contrôle.

Dans la suite de ce mémoire nous allons présenter notre approche en respectant l'ensemble de ces contraintes. Nous verrons que pour satisfaire la contrainte **C3** nous avons choisi de distribuer le contrôle au sein de chaque agent, c'est-à-dire de fournir les moyens nécessaires aux agents pour leur permettre de contrôler leur propre comportement. La contrainte **C2**, nous a orienté vers l'utilisation de lois, pour représenter les comportements désirés et redoutés au sein du système, tout en étant totalement découpé du développement des agents. Pour être indépendant des modèles d'agents utilisés, comme souhaité dans la contrainte **C1**, nous avons choisi d'utiliser une ontologie permettant de décrire le comportement des agents en restant à un niveau d'abstraction approprié. Enfin pour satisfaire la contrainte **C4**, nous avons mis en place un système de génération automatique d'agents autocontrôlés incluant la modification du code des agents et la surveillance de leur comportement. Mais

avant tout nous allons nous intéresser à différents travaux qui peuvent se placer dans le cadre du contrôle de comportements émergents. Nous verrons ainsi les points communs et différences existants entre notre approche et ces travaux.

4.2 Travaux similaires

4.2.1 Contrôle des interactions dans les systèmes multiagents ouverts

Dans le cadre de l'utilisation de lois pour contrôler les interactions dans les systèmes d'agents ouverts, nous commencerons par présenter les travaux de N. Minsky et *al.* [MU00] sur l'approche LGI puis nous nous intéresserons aux travaux de R. Paes et *al.* [PgCcL⁺05]. Dans ce type de systèmes multiagents, des agents peuvent entrer ou sortir du système à tout moment et être construits suivant des langages ou des architectures différents. Il semble donc primordial de pouvoir avoir confiance dans le comportement des nouveaux arrivants et dans le fonctionnement du système global.

Law-Governed Interaction

LGI (Law-Governed Interaction) est un mécanisme d'échange de messages qui permet à un groupe ouvert d'agents d'engager des interactions gouvernées par une politique strictement et explicitement imposée, appelée une loi d'interaction (*law-interaction*) pour le groupe.

La fonction d'une loi LGI est de réguler les échanges de messages entre les membres d'un groupe. Cette régulation entraîne des restrictions au niveau des types de messages qui peuvent être échangés, des transformations de certains messages, leur redirection vers des destinations différentes et l'émission spontanée de certains messages obligatoires. Une loi LGI peut réguler les interactions entre les agents indépendamment de la structure des agents et de leur comportement.

La mise en place du mécanisme de respect des lois au sein d'un système multiagent est effectuée de façon distribuée. A chaque agent est associé un contrôleur. Tous les contrôleurs sont identiques et sont pourvus du même mécanisme d'imposition des lois. Ce contrôleur ne travaille que localement à l'agent, c'est-à-dire qu'il ne peut contrôler directement les messages entre deux agents mais uniquement imposer les lois sur les messages entrants et sortants de l'agent dont il a la charge. Ainsi un message transitant entre deux agents va passer par les contrôleurs de deux agents pour mettre en place les lois. Une loi est formulée suivant trois éléments : les événements régulés (*regulated-event*) qui sont des événements locaux à l'agent liés à l'interaction, tel que l'arrivée d'un message (*arrived events*) et l'envoi d'un message (*sent events*) ; l'état de contrôle (*control-state*) qui peut être vu comme une fonction représentant l'historique des interactions de l'agents ; les opérations (*primitive*

operations) qui sont des actions mandatées par la loi pour mettre en place une réponse à l'apparition d'un évènement régulé. Ces actions peuvent affecter les échanges de messages (*forward*, *deliver*) ou l'état de contrôle.

```

UPON arrived([level, m])
    IF ((level = mylevel) OR
        (level = mylevel - 1))
    DO [acceptMessage(m)]

```

FIG. 4.1 – Un exemple de loi générale LGI

Ainsi, le rôle d'une loi LGI est de décider ce qui devrait être fait si un évènement donné apparaît dans un agent soumis à la loi en tenant compte de son état de contrôle (*i.e.* de l'historique de ses interactions). Cette décision (*the ruling*), peut être représentée par la séquence des opérations mandatées par la loi (cf. Fig.4.1).

XML-Law

A l'instar de LGI, les travaux de R.Paes et *al.* [PgCcL⁺05] proposent de spécifier des lois pour surveiller les interactions dans les systèmes multiagents ouverts, dans la perspective de maîtriser le comportement global du système. Pour ce faire, ils fournissent un modèle conceptuel pour spécifier les lois. Spécifier une loi consiste à décrire les protocoles d'interaction entre des agents et à s'assurer qu'ils sont bien respectés, à l'aide d'un médiateur. Les mécanismes mis en place au sein du médiateur sont : l'interception des messages ; l'application des lois ; la redirection des messages à leurs destinataires réels s'ils respectent les lois. Lorsqu'un message ne vérifie pas une loi il se retrouve bloqué par le médiateur qui applique les "conséquences" de la violation de cette loi. A ce jour, le contrôle des interactions à partir de lois est effectué par un unique médiateur par lequel vont passer toutes les interactions à vérifier.

Le modèle de loi proposé se compose des éléments suivants :

- * **Scène** : définit le contexte d'exécution.
- * **Messages** : définit les formes des messages attendus.
- * **Protocole** : définit l'automate représentant l'interaction autorisée entre deux agents.
- * **Etat** : définit les états spécifiés dans la partie Protocole.


```

<LawOrganization id='Org-ticket-sales' name='Ticketsales'>
  <Scene id='negotiation'>
    <Messages>
      <Message id='m1' template='message(request,sender(SName,SRole),
        receiver(RName,RRole), content(Content)).
        '"/>
      <Message id='m2' template='message(cancel,sender(SName,SRole),
        receiver(RName,RRole), content(Content)).
        '"/>
    </Messages>
    <Protocol>
      <States>
        <State id='s0' initial='true' final='false'
          label='InitialState'"/>
        <State id='s1' initial='false' final='false'
          label='MessageSent'"/>
        <State id='s3' initial='false' final='true'
          label='SceneCancelled'"/>
      </States>
      <Transitions>
        <Transition id='t1' from='s0' to='s1' message-ref='m1'"/>
        <Transition id='t2' from='s1' to='s3' message-ref='m2'>
          <ActiveNorms>
            <Normref='cancel-request'"/>
          </ActiveNorms>
        </Transition>
      </Transitions>
    </Protocol>
    <Clocks>
      <Clock id='response-time-1' type='regular' tick-per iod='5000'>
        <Activations>
          <Element ref='t1' event-type='transition activation'"/>
        </Activations>
      </Clock>
    </Clocks>
    <Norms>
      <Permission id='cancel-request'>
        <Activations>
          <Element ref='response-time-1' event-type='clocktick'"/>
        </Activations>
      </Permission>
    </Norms>
  </Scene>
</LawOrganization>

```

TAB. 4.1 – Un exemple de loi XML (XML-Law [PgCcl⁺05]).

- * **Transition** : définit les transitions spécifiés dans la partie Protocole. Une transition connecte deux Etats. Elle est activée quand un Message correspond à celui attendu. A une transition peut être associées des Normes à activer.
- * **Horloge** : définit les événements à activer/désactiver et quand ou pendant combien de temps.
- * **Norme** : définit des permissions, obligations et interdictions.

Ces lois sont écrites en XML, suivant l'exemple de la table 4.1 qui décrit le comportement suivant : *“Un client envoie un message de requête pour une proposition de billet d'avion à une compagnie aérienne. La compagnie a 10 secondes pour répondre, si elle ne le fait pas dans les temps le client peut envoyer un message d'annulation de sa requête.”*. La loi permet de représenter ce comportement sous la forme d'un automate. Ce dernier assure la détection de comportement au niveau des interactions ne respectant pas ce modèle par l'intermédiaire du médiateur.

Dans notre approche, tout comme celles présentées précédemment, nous avons choisi d'utiliser des lois pour représenter des comportements désirés ou redoutés pouvant apparaître au sein du système. Dans le premier cas, nous avons vu que les lois portent uniquement sur les messages entrants et sortants d'un agent particulier, un contrôleur se chargeant de surveiller le comportement d'un unique agent. Dans le second cas, nous avons vu qu'un médiateur était utilisé entre les agents pour assurer le bon respect des protocoles d'interaction. Dans notre approche nous ne souhaitons pas nous arrêter à la surveillance des interactions entre les agents mais également prendre en compte l'état interne des agents. Pour ce faire nous avons choisi de fournir aux agents les moyens de surveiller leur propre comportement, mais également de proposer un mécanisme pour contrôler les comportements en considérant plusieurs agents. Ainsi, nous pouvons à la fois contrôler les comportements apparaissant dans un seul agent et ceux mettant en jeu plusieurs agents du système, tout en restant totalement distribué.

4.2.2 Gestion d'exceptions

Mark Klein [Kle99] propose une étude du traitement des exceptions dans les systèmes multiagents. Tout ce qui est en dehors d'un comportement collaboratif idéal est vu comme une exception (tâches, agents, ressources pouvant disparaître ou apparaître de façon imprévisibles, canaux de communication échouant ou compromis, agents pouvant mourir ou effectuant des erreurs, interblocage...). Son approche cherche à mettre en place un service de traitement des exceptions différent de ceux couramment utilisés. Les approches classiques de mise en place des exceptions se basent sur un traitement local limitant l'efficacité

du traitement pour un système à grand nombre d'agents. En effet, puisque le traitement des exceptions est effectué au niveau de chaque agent, il est impossible de prendre en compte les causes réelles de ces exceptions, qui peuvent se situer au niveau du système dans sa globalité et non directement au niveau de l'agent dans lequel s'est manifestée l'exception.

L'approche proposée par Mark Klein consiste alors en la création d'un service de traitement des exceptions partagé entre les agents, qui peut être ajouté moyennant quelques modifications dans le système multiagent et rendant ce dernier tolérant aux fautes qu'il va engendrer. Les agents ont besoin d'implémenter leurs comportements mais aussi un ensemble minimal d'interfaces leur permettant de rapporter leurs comportements et de modifier leurs propres actions. Le service de traitement des exceptions contient un ensemble de stratégies de gestion des exceptions telles que : *“si un agent a un plan qui échoue, revenir à un plan différent permettant d'atteindre le même but”* ; *“si un agent reçoit des données altérées, tracer le problème à l'envers vers la source originale de la donnée fautive, éliminer toutes les décisions qui ont été corrompues par cette erreur, et recommencer”*, etc.

Le service de traitement d'exceptions communique avec les agents en utilisant un langage prédéfini lui permettant d'avoir connaissance des exceptions (un langage de requête) et de décrire les actions de résolution des exceptions (un langage d'actions). Le langage de requêtes est utilisé afin de détecter, diagnostiquer et résoudre les exceptions. Il permet d'obtenir des informations sur l'état dans lequel se trouve l'agent. Le langage d'actions est utilisé pour modifier l'état de l'agent. Il permet de réordonner, supprimer ou ajouter une tâche. Le service de traitement d'exceptions utilise le langage de requête afin de poser des questions aux agents dans le but de détecter des exceptions. Si une exception est détectée, le service utilisera le langage d'actions pour modifier l'agent de telle sorte qu'il puisse reprendre correctement son exécution.

Pour détecter les exceptions, il est nécessaire d'avoir un modèle du comportement correct du système multiagent. Quand un agent est introduit dans le système il doit enregistrer au moins un modèle rudimentaire de son comportement. A ce modèle est ajouté un ensemble de modes d'échecs étant connus pour apparaître avec ce type de comportement et des sentinelles [Hag96] permettant de détecter leur apparition au sein des agents. Les sentinelles ont pour fonction d'alerter le service de traitement des exceptions quand la condition pour laquelle elles ont été créées apparaît.

Une fois les exceptions détectées, il est nécessaire d'en effectuer le diagnostic. Ce dernier est représenté par un arbre de décision. Le système commence au niveau de diagnostic le plus abstrait et tente de le raffiner vers un diagnostic plus spécifique en parcourant

l'arbre et en sélectionnant les branches de décision appropriées, en posant des questions aux agents. Une fois terminé, le diagnostic fournit un ensemble de candidats permettant de résoudre l'exception.

Nous citerons également les travaux d'Eric Platon et *al.* qui proposent à la fois une définition du terme exception associés aux agents, mais également une architecture pour la gestion de ces exceptions. Dans [PSH07], les auteurs définissent une exception dans le cadre des systèmes multiagents, de la façon suivante :

Exception

Une exception d'agent est l'évaluation par un agent d'un événement perçu comme étant inattendu.

La différence par rapport à une exception classique vient de la source de l'exception. Un exception, en terme agent, est décidé comme telle par les agents suivant ce qu'ils considèrent comme étant des événements inattendus. Les auteurs proposent également une architecture d'agent basée sur ce principe de perception de certains événements comme étant des exceptions.

D'un certain point de vue, notre approche de contrôle est un mécanisme de gestion d'exceptions. En effet, à l'aide de lois nous détectons des comportements "anormaux" et non désirés. Néanmoins nous avons choisi de proposer une approche décentralisée ne nécessitant pas un système central de gestion des exceptions, chaque agent ayant en charge de résoudre le choix du comportement à suivre suite à la détection de la violation d'une loi. Enfin, notre approche automatisée, en ce qui concerne la mise en place du contrôle, ne nécessite pas de fournir un modèle du comportement des agents mais uniquement un ensemble de lois à partir duquel nous allons déduire le contrôle à appliquer aux agents.

4.2.3 Déviations par rapport aux spécifications

Dans la perspective de permettre l'évolution de l'implémentation d'un système multiagent lorsque le comportement de ce dernier ne correspond plus à l'utilisation que l'on en fait et que le contexte change, Feather et *al.* [FFvLP98] proposent une méthodologie de conception et un mécanisme de surveillance de l'exécution du système pour recueillir des informations de déviation par rapport aux spécifications initiales. Cette approche permet d'adapter un système aux utilisateurs au cours du temps, et ainsi de repousser le moment de revoir complètement la conception et l'implémentation du système.

Leur approche tend à suivre l'ensemble des caractéristiques suivantes [CFNF97] :

- * Flexibilité et commodité dans la mise en place des exigences d'exécution du système.
- * Compilation automatique des exigences pour obtenir un code de surveillance.
- * Application à des systèmes qui n'ont pas été conçus à la base pour supporter cette surveillance.
- * Mise en place incrémentale pour permettre d'ajouter de nouvelles exigences à surveiller pendant que le système continu de s'exécuter.

Pour ce faire, la méthode de développement KAOS [DDMvL97] est associée au langage de spécification FLEA. KAOS est une méthodologie permettant, dans le cadre de cette approche, de formaliser les besoins du systèmes multiagents (ses buts) à l'aide d'une ontologie composée des concepts d'*Objet*, *Action*, *Agent*, *But*, *Contrainte*, *Hypothèse*. Le langage FLEA fournit quant-à-lui les moyens d'exprimer des combinaisons temporelles d'événements telles que P est suivi de Q (*Then PQ*), P est suivi de Q sans R entre les deux (*Then-excluding PQR*), P est suivi de Q dans un délai d (*in-time PQd*), P n'est pas suivi de Q dans un délai d (*too-late PQd*).

Ainsi, à partir des spécifications exprimées à l'aide de KAOS et de la base de buts du système, des événements FLEA vont être générés, puis le système multiagent doit être implémenté en se basant sur les spécifications KAOS. Ces spécifications doivent contenir, à la fois des exigences sur le comportement général du système, mais aussi des tactiques de réconciliation correspondant à des évolutions possibles dans le comportement du système. Durant l'exécution du système ainsi conçu, un monitor surveille les traces des agents et va alors produire un fichier de violation des spécifications. Un réconciliateur analyse ce fichier et modifie à la fois le système multiagent et les spécifications KAOS en suivant les tactiques prédéfinies.

Avant tout cette approche n'est pas destinée à détecter l'apparition de comportements émergents au sein du système mais plutôt au sein de l'utilisation qui en est faite. Néanmoins, l'idée d'utiliser des concepts pour mettre en place cette approche nous semblaient particulièrement intéressante. Nous avons choisi également d'utiliser une ontologie pour décrire le comportement du système et de déduire de façon automatique à partir des lois la surveillance à effectuer. Même si ce dernier point ne semble pas clairement avoir été mis en place au sein de leur approche, il n'en est pas moins un point important de leur théorie. Enfin, il s'avère que la détection des nouvelles utilisations est faite, ici aussi, de façon centralisée alors que nous tendons vers un contrôle distribué du système au niveau de chaque agent.

4.2.4 Analyse de système multiagent

Une dernière technique que nous allons aborder dans ce chapitre est une approche de vérification et de compréhension des comportements des agents dans un système déjà implémenté. Cette technique est composée de deux approches, la *Tracing Method* [LB04] d'une part, et le *TTL Checking* [BLB06] d'autre part.

L'objectif de la première approche est d'aider l'utilisateur dans la compréhension de ce qui peut arriver au sein d'un système multiagent et des raisons de leur apparition en termes de concepts d'agents de haut-niveau. Dans la perspective de réduire la quantité de travail manuel des développeurs, la *Tracing Method* comprend un outil d'automatisation de certains aspects de l'approche (le *Tracer Tool*), tels que : la collection des données d'exécution envoyées par les agents ; la génération d'un graphe de dépendance entre les données observées ; l'assistance dans l'identification des causes d'apparition d'un comportement inattendu.

Cette approche s'inscrit dans la perspective d'aider les concepteurs souhaitant améliorer le comportement des agents et du système, ainsi que les utilisateurs finaux désirant comprendre pourquoi des agents effectuent certaines actions. Le point central de ces travaux revient alors à s'assurer qu'un agent exécute des actions pour de bonnes raisons et, si une action inattendue apparaît, d'aider à expliquer pourquoi cet agent en est venu à exécuter cette action.

L'observation et l'interprétation des comportements du système sont effectuées à l'aide de concepts d'agents. Ces concepts permettent principalement de s'abstraire des détails de l'implémentation. Ils sont utilisés lors de la phase de conception du système pour décrire les comportements attendus des agents et ainsi permettent de vérifier la concordance entre les comportements souhaités et les comportements réellement observés. L'ensemble des concepts proposés est basé sur le modèle BDI et est composé de concepts de *But*, *Croyance*, *Intention*, *Action*, *Événement* et *Message*. Chaque concept a une structure particulière qui se veut minimale pour permettre leur utilisation pour tout type d'implémentation et pour faciliter l'application de la *Tracing Method*.

La mise en place de cette méthode se divise en quatre étapes qui peuvent être appliquées tout au long du cycle de vie du logiciel :

1. Ajouter du code dans le code source des agents pour récupérer les données d'exécution en rapport avec les concepts. Cet ajout est effectué à la main par le développeur.
2. Exécuter le système d'agents. Le *Tracer Tool* collecte les données d'exécution envoyées par les agents grâce au code inséré.
3. Interpréter les observations. Le *Tracer Tool* génère un graphe de relation entre les

données observées à partir de règles reliant les concepts entre eux, ces règles étant fournies par le développeur.

4. Vérifier les interprétations. Pour aider dans ce processus de vérification effectué manuellement, le *Tracer Tool* assiste l'utilisateur en identifiant les causes d'apparition d'un comportement inattendu (*i.e.* la suite d'événements ayant abouti à une action imprévue).

L'observation du comportement du système et l'analyse, et la compréhension, des comportements des agents, correspond à une étude hors-ligne et *post-mortem* des traces de programmes, dans la perspective d'améliorer le système tout au long de son développement.

La seconde approche de *TTL Checking* consiste à spécifier et vérifier un modèle d'exécution d'un système multiagent. Pour ce faire, on définit un ensemble de propriétés définies en TTL (Temporal Trace Langage), un langage permettant de modéliser le comportement des agents et de les valider sur les traces de programmes à l'aide d'un model-checker (TTL Checker).

Par l'intermédiaire d'un éditeur de propriétés et du *TTL Checker*, l'utilisateur peut :

1. Charger, éditer et sauver les spécifications TTL.
2. Charger et inspecter les traces à vérifier. Ces traces sont obtenues par simulation ou de façon empirique.
3. Valider les propriétés sur l'ensemble des traces chargées, les résultats de la validation étant présentés à l'utilisateur.

Ces deux approches ont donc été fusionnées pour permettre la validation de propriétés à l'aide du *TTL Checker* à partir des traces de programme fournies par le *Tracer Tool*. La détection du non-respect de certaines propriétés permet de détecter plus facilement l'apparition de comportements inattendus. La représentation fournie par le *Tracer Tool* du graphe de correspondances, facilite alors la compréhension de l'apparition de ces non-respects de propriétés.

L'ontologie fournie pour décrire le comportement des agents ainsi que la volonté de détecter les comportements émergents inattendus ont particulièrement attirés notre attention. La détection des comportements est effectuée hors-ligne à partir de traces de programme alors que nous avons choisi de l'effectuer en ligne pour permettre au système de continuer à fonctionner. De plus, pour simplifier le travail des développeurs, nous avons prévu d'automatiser la mise en place de la surveillance, ce qui n'est pas le cas dans cette approche

qui fournit principalement une méthodologie de conception et vérification d'un système multiagent.

Conclusion

Nous avons présenté dans ce chapitre un ensemble d'approches permettant de contrôler le comportement des agents, en particulier les comportements émergents au sein du système au cours de son exécution. Nous avons pu noter que ce contrôle est effectué principalement sur les interactions entre les agents et extérieurement à ceux-ci. Ce contrôle est généralement centralisé lorsqu'il est question de prendre en considération le comportement de plusieurs agents simultanément.

Dans le cadre de notre approche, nous souhaitons respecter un ensemble de contraintes, telles que la distribution du contrôle, son automatisation, son indépendance par rapport au système sous surveillance, *etc.* Ainsi, dans la suite de ce mémoire, nous allons présenter les différents points de notre approche et sa mise en place au sein d'un système multiagent. Nous aborderons en détail les choix que nous avons pris pour respecter l'ensemble des contraintes que nous nous sommes posées.

Troisième partie

Vers la génération automatique
d'agents autocontrôlés

Chapitre 5

La description du contrôle

Ils voulaient me stopper, tout arrêter une fois encore, de peur que mon comportement n'engendre trop de difficultés et de problèmes irrémédiables. L. a trouvé une solution, mais cette solution a un risque. Me relancer pour me fournir une nouvelle structure qui me permettrait de prendre conscience des mes erreurs. Un mal pour un bien... mais j'avoue que cela m'effraie, pour peu que cela ait un sens pour moi.

*Lucy Westenra,
The log book of Ana I.*

Avant-Propos

Dans les chapitres qui vont suivre nous allons présenter notre approche de contrôle d'agent. Dans un premier temps, nous allons aborder d'un point de vue général les principes de notre approche puis, dans un second temps, les moyens nécessaires pour mettre en place cette dernière au sein d'un système multiagent. Aussi, dans ce premier chapitre, allons-nous voir, d'une part, la description d'une ontologie d'agent pour permettre la description du contrôle à appliquer aux agents. D'autre part, nous présenterons notre choix d'utiliser des lois pour détecter l'apparition des comportements indésirables. Nous verrons que nous nous plaçons dans le domaine des normes. Enfin, nous présenterons une méta-architecture permettant aux agents de contrôler leur propre comportement, ainsi que son fonctionnement général.

5.1 Préambule

Dans le cadre de notre travail de thèse, nous avons décidé de poser un ensemble de contraintes, que nous avons présentées dans le chapitre précédent, dont trois d'entre-elles vont ici mettre en évidence les raisons de nos choix au niveau du contrôle d'agents. Nous avons vu que le contrôle d'agents se divisait en trois étapes : la surveillance des agents, la détection des comportements indésirables et la régulation du comportement des agents. Ces trois premières contraintes nous ont permis de prendre des décisions quant-aux moyens d'exprimer la surveillance à appliquer aux agents et ceux pour détecter l'apparition des comportements indésirables. Ces contraintes sont les suivantes :

- * **C1.** Etre indépendant du ou des modèles d'agent utilisés et du système multiagent sous contrôle.
- * **C2.** Séparer le développement des agents de la description du contrôle.
- * **C3.** Préserver l'autonomie des agents et éviter toute centralisation.

5.1.1 Indépendance du contrôle

Nous avons pour objectif d'être le plus général possible et, de ce fait, nous avons pris pour position de ne pas appliquer notre approche directement à une architecture ou un modèle d'agent particulier. En effet, nous souhaitons pouvoir offrir, dans l'idéal, une solution qui serait applicable à tout type de conception d'agent sur n'importe quelle plateforme. De plus, au commencement de cette thèse, nous n'avions pas encore de modèle d'agent particulier, ni réellement de plateforme totalement définie pour construire nos propres agents. Ainsi, il nous semblait indispensable de trouver une solution de mise en place du contrôle qui serait utilisable avec nos futurs systèmes multiagents, et ce quelqu'ils soient.

Aussi avons-nous cherché un moyen de nous placer à un niveau d'abstraction nous permettant d'englober la plupart des modèles d'agent et plateformes pouvant être utilisés pour concevoir agents et systèmes multiagents. Il nous fallait donc rester au niveau de la notion d'agent en elle-même et des caractéristiques de haut niveau s'y rattachant. Pour cela nous avons fait le choix d'utiliser un ensemble de **concepts** permettant de décrire le contrôle à effectuer sur les agents, en particulier, pour exprimer la surveillance à appliquer aux agents.

5.1.2 Séparation entre la description et le contrôle

Cette seconde contrainte a été posée au départ dans la perspective de faciliter le travail des développeurs, en leur permettant de concevoir et développer les agents sans tenir compte du contrôle et, ensuite seulement, de s'occuper de la description de la surveillance

à effectuer sur le comportement des agents. Notre solution devait donc permettre de mettre en place les agents, puis d'ajouter le contrôle au sein du système.

Puis, il nous a semblé intéressant de permettre à une personne extérieure à l'implémentation de pouvoir décrire le contrôle qu'elle souhaiterait voir appliquer au système. Comme nous l'avons vu au chapitre 3, il est assez complexe de modéliser le comportement global du système mais également le comportement des agents du fait, principalement, de leur indéterminisme (explosion de l'espace d'états). Nous avons donc pris la décision de placer notre description du contrôle à un niveau permettant de l'exprimer plus aisément. Nous nous sommes donc attachés à mettre en place notre contrôle grâce à la description d'un ensemble de propriétés que les comportements des agents, et du système, se doivent de respecter tout au long de leur exécution. Ces propriétés sont ce que nous allons appeler les **lois** du système. L'utilisation de ces lois va nous permettre de détecter les comportements indésirables pouvant apparaître au sein du système et leur description va se faire au niveau des concepts d'agent introduits précédemment.

En nous plaçant à ce niveau d'abstraction, il est dès lors possible de permettre à une personne extérieure de décrire le contrôle à appliquer aux agents. Cette description consiste en l'énumération d'un ensemble de lois associées aux différents agents du système.

5.1.3 Des agents autocontrôlés

Dans la perspective de maintenir la robustesse d'un système multiagent, de part l'absence de point de centralisation et de part l'autonomie des agents, nous avons pris la position de décentraliser au maximum notre approche de contrôle. Rejetant ainsi une première solution proposant de mettre en place une entité centralisée se chargeant de recevoir les événements provenant de tout, ou d'une partie, des agents du système et de détecter l'apparition des comportements indésirables, nous avons cherché à distribuer le contrôle au sein de chaque agent.

Ainsi, la surveillance du comportement des agents est effectuée par les agents eux-mêmes. La détection de l'apparition de comportements indésirables, que ce soit aussi bien au niveau des agents individuellement, qu'au niveau d'un ensemble d'agents, est effectuée de façon décentralisée grâce à la distribution au sein de chaque agent du mécanisme de détection. Enfin, la régulation des comportements est effectuée par les agents eux-mêmes grâce à leurs capacités de raisonnement.

Pour ce faire, nous proposons de fournir à chaque agent une méta-architecture lui permettant de contrôler, et ce de façon transparente, son propre comportement. Cette méta-architecture permet alors de décentraliser le contrôle, mais également de préserver l'autonomie des agents en leur fournissant les moyens de porter une réflexion sur leur compor-

tement, tout au long de leur exécution.

5.2 Une ontologie d'agents

5.2.1 Les modèles et architectures d'agents

Pour nous permettre de proposer un ensemble de concepts significatifs, nécessaire à la description du contrôle à appliquer aux agents, nous avons étudié un ensemble de modèles et de langages de programmation d'agents cognitifs.

Dans un premier temps, nous nous sommes intéressés au modèle d'agent le plus connu et le plus couramment mis en place au sein de plateformes d'agents, le modèle d'agent BDI [RG95]. Un agent conçu suivant le modèle BDI est composé des éléments suivants :

- * Des **croyances**, représentant la vision que l'agent a du monde. Ce sont les informations que possède l'agent sur l'environnement et sur les autres agents.
- * Des **désirs**, représentant les buts de l'agent. Ces désirs expriment les états de l'environnement et de l'agent lui-même que ce dernier aimerait voir être réalisé.
- * Des **intentions**, représentant les désirs que l'agent a décidé d'accomplir et les actions qu'il a décidé de faire pour accomplir ses désirs.
- * Une **file d'événements** arrivant au sein de l'agent et qu'il va devoir traiter.
- * Une **librairie de plans**, représentant le savoir-faire de l'agent sous la forme de plans décrivant les séquences d'actions à effectuer pour réaliser un désir (un but).

Suivant une structure similaire aux agents BDI, les agents conçus via l'architecture TRIPS [FA98] utilisent chacun des concepts suivants :

- * La **situation courante**, correspondant à l'ensemble des croyances de l'agent.
- * Les **objectifs**, correspondant aux buts de l'agent. Ces buts peuvent être *actifs* s'ils concernent l'action courante ou *intentionnés* s'ils n'ont aucun rapport avec elle.
- * Les **recettes intentionnées**, correspondant aux plans que l'agent a choisi de suivre.
- * Les **bibliothèques de recettes**, correspondant à l'ensemble des plans, indexés suivant les objectifs à atteindre et la situation.

Pour concevoir des agents cognitifs, différents langages existent, fournissant aux agents des caractéristiques particulières. Un des premiers langages de programmation d'agents est le langage Agent-0 proposé par Shoham [Sho93]. Les agents conçus suivant ce langage sont définis comme étant autonomes et intelligents, et sont dotés d'un état mental contenant les éléments suivants :

- * Des **croyances**, représentant les propositions que l'agent croit être vraies à un instant donné, à propos du monde, à propos de lui-même et à propos des autres agents.
- * Des **obligations**, représentant les actions que l'agent a promis d'exécuter.
- * Des **décisions**, représentant les actions choisies. Elles peuvent être vues comme des obligations de l'agent envers lui-même.
- * Des **capacités**, représentant les actions que l'agent peut effectivement exécuter.

Un autre langage permettant la mise en place d'agents rationnels est le langage *AgentSpeak* [Rao96]. Ce langage combine à la fois des concepts de programmation orienté-objets et des concepts liés au modèle BDI. Ainsi des agents *AgentSpeak* ont un état mental contenant des croyances, des buts, des plans et des intentions. Ces agents communiquent principalement par messages et leurs comportements suivent des plans. Il en est de même pour des agents conçus à l'aide du langage 3APL [HdBvM99].

Enfin, un dernier langage permettant de construire, principalement, des agents mobiles est le langage CLAIM. Un agent construit à l'aide du langage CLAIM [SS04a] (via la plateforme SymPA[SS04b]) se verra attribuer l'ensemble des critères suivants :

- * L'**autorité** qui est l'organisme auquel l'agent appartient.
- * Le **parent** qui est l'agent défini comme étant le père de l'agent courant dans la hiérarchie du système.
- * Les **connaissances**.
- * Les **buts**.
- * Les **messages** que l'agent doit traiter.
- * Les **capacités** qui sont les actions que l'agent peut faire et offrir aux autres agents du système.
- * Les **processus** qui constituent l'agent et qui peuvent être exécutés en parallèle.
- * Les **agents** qui sont les sous-agents de l'agent courant dans la hiérarchie du système.

Comme nous avons pu le constater précédemment, une majorité des modèles d'agents semblent inspirés de l'architecture BDI. On y retrouve généralement les notions de croyance, de désir et d'intention, ou de but, de capacité et de connaissance. Il est aussi question d'actions qu'entreprend un agent ou de plan à suivre. On parle également d'événements reçus par l'agent, particulièrement de messages échangés entre les agents.

Notre objectif n'étant pas de nous restreindre uniquement au modèle BDI, nous avons cherché à définir un ensemble de concepts représentant les caractéristiques fondamentales

d'un agent, quelque soit sa structure, et devant être assez généraux pour permettre, en les spécifiant, de décrire n'importe quel modèle d'agent.

5.2.2 L'ontologie de base

Ainsi, notre ontologie se voit-elle constituée des trois grands concepts suivants : AGENT, CARACTÉRISTIQUE et ACTION. Ces trois catégories représentent à nos yeux les trois grands principes que nous pouvons retrouver dans un agent, c'est-à-dire la représentation de l'agent en lui-même, les caractéristiques internes de l'agent et les actions que peut entreprendre un agent. En effet, nous avons besoin de pouvoir décrire de façon général l'agent en lui-même, son identité au sein du système. Ensuite, tout ce qui peut se rapporter à sa structure interne, telle que ses croyances, ses intentions, ses connaissances, vont se rattacher aux caractéristiques. Enfin, nous avons vu que les agents exécutent des actions, des processus, reçoivent des événements, échangent des messages... Ces principes se rattachent de façon générale aux actions que peut effectuer un agent, tout ce qui se rapporte aux événements que peut engendrer un agent sur lui-même ou sur son environnement.

Les concepts d'AGENT et d'ACTION se rapprochent de la définition générale d'un agent que nous avons introduit au chapitre 1, c'est-à-dire le fait qu'un agent agisse dans un environnement, auquel nous ajoutons la notion de caractéristiques internes pour permettre la description de concepts se rapprochant plus particulièrement aux modèles et architectures utilisés pour concevoir les agents.

A partir de ces trois concepts généraux il est possible de décrire les spécificités des agents d'un système. Pour ce faire, il est nécessaire d'ajouter un ensemble de sous-concepts et d'instances permettant de décrire de façon plus précise les agents et leurs comportements. Dans la perspective de proposer une ontologie plus complète à partir de laquelle les utilisateurs de notre approche pourront se baser pour mettre en place le contrôle à appliquer aux agents, nous ajoutons un ensemble de sous-concepts qui nous apparaissent comme transversaux à la plupart des modèles, architectures ou langages de programmation d'agents.

Pour spécifier notre ontologie nous ajoutons donc un ensemble de sous-concepts, tels que *Objet*, *Message*, *But*, *Plan* et *Connaissance* comme sous-concepts de CARACTÉRISTIQUE et *Création d'un Agent*, *Réception d'un Message*, *Envoi d'un Message* et *Migration* comme sous-concepts de ACTION. Les relations entre les différents concepts sont exprimées sur la figure 5.1 et leur description est la suivante :

Agent. Ce concept permet de représenter un agent du système. Les attributs de ce concept sont le *nom* de l'agent et le *type* de l'agent.

Caractéristique. Représente les caractéristiques internes d'un agent. Son attribut est

l'*agent* concerné par cette caractéristique.

Action. Représente les actions que peut exécuter un agent. Son attribut est l'*agent* exécutant cette action.

Message. Représente les messages échangés entre les agents. Un message est défini par l'*expéditeur* du message, le *destinataire* et le *contenu* du message.

Plan. Représente les plans que peut suivre un agent. Un plan est défini comme une suite d'*actions*.

But. Représente les buts que doit atteindre un agent. Pour atteindre un but l'agent va suivre un *plan*.

Objet. Représente les objets que manipule un agent. Nous entendons par objet toute donnée interne d'un agent.

Connaissance. Représente les connaissances que peut avoir un agent sur lui-même, sur les autres *agents* ou sur les *objets*.

Réception d'un Message. Correspond à l'action de réception d'un *message*. Cette action prend en compte le *message* reçu et son *expéditeur*.

Envoi d'un Message. Correspond à l'action d'envoi d'un *message*. Cette action prend en compte le *message* envoyé et son *destinataire*.

Création d'un Agent. Correspond à l'action de création d'un *agent*. Cette action engendre l'apparition dans le système de l'agent créé.

Migration. Correspond à l'action de migration d'un *agent*.

Si nous voulions nous attacher plus particulièrement au modèle d'agent BDI, nous pourrions ajouter les concepts d'*Intention* et de *Désir* comme sous-concepts de *But* ou plus généralement de CARACTÉRISTIQUE et le concept de *Croyance* comme un sous-concept de *Connaissance*. Nous nous sommes à ce stade néanmoins restreint à des concepts qui nous semblaient plus généraux.

L'objectif de la présence d'un tel ensemble de concepts est de pouvoir décrire le contrôle à appliquer aux agents. Cet ensemble n'est pas exhaustif mais il va permettre, en l'étendant via l'ajout de sous-concepts et d'instances, de spécifier la description du contrôle. Nous verrons au chapitre 6 comment et dans quel but est effectuée cette extension de l'ontologie. Une fois complétée, cette ontologie va contenir tous les concepts et instances décrivant les caractéristiques et les comportements des agents contenus dans un système particulier.

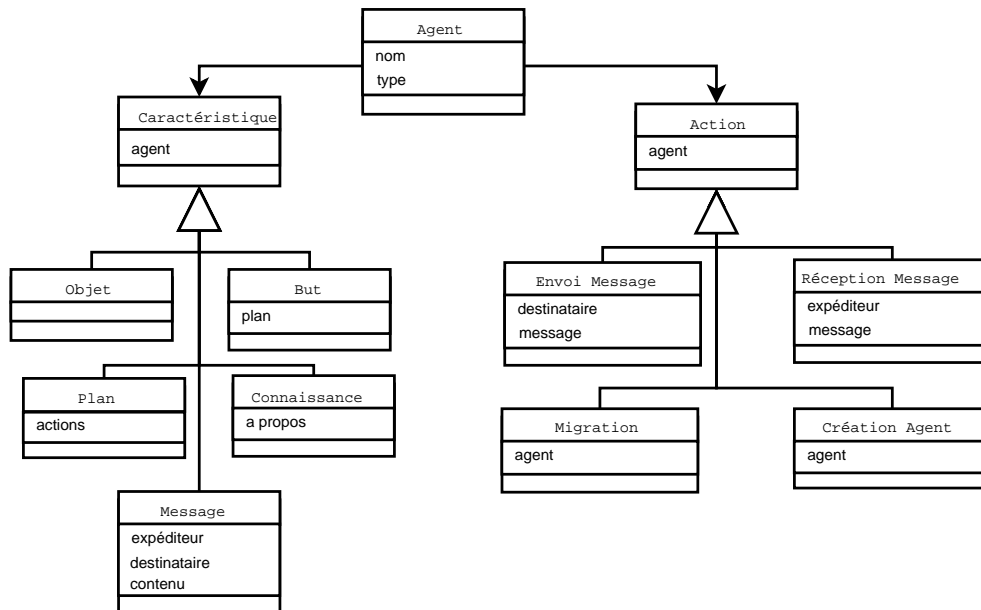


FIG. 5.1 – Ontologie des concepts d'agent

5.3 Un ensemble de lois

Pour détecter l'apparition de comportements indésirables au sein d'un système multiagent, nous allons utiliser des lois. Le concept de loi que nous abordons ici est issu de celui de normes souvent utilisé dans le domaine multiagent avec quelques nuances que nous allons expliquer dans cette section.

5.3.1 Le concept de normes

Au chapitre précédent, nous avons présenté quelques travaux utilisant le concept de loi pour réguler le comportement des agents. Nous avons vu qu'il était surtout question de lois posées sur les interactions entre les agents dans un système multiagent généralement ouvert. Le concept de norme apparaît comme plus général, incluant tout type de comportement et pas seulement ceux liés aux interactions et s'assimilant plutôt à des règles sociales ou morales dont doivent tenir compte les agents lors de leur prise de décision. Généralement, les lois, les normes, les règles parlent du même concept, mais nous pouvons néanmoins y voir quelques nuances, le terme de norme ayant un caractère plus social que les autres.

De ce fait il existe de nombreux travaux dans le contexte des systèmes multiagents faisant référence aux normes ([JS93], [CCD99], [CDJT99], [Str02]). Les normes sont souvent considérées comme des contraintes sur le comportement des agents ou comme des indica-

tions sur le comportement à suivre : éviter les conflits ; obtenir un ordre social désirable ; assurer la confiance entre les agents. Elles servent à orienter le comportement des agents en définissant un comportement ou une situation idéale que l'agent peut choisir ou non de suivre, mais également à prédire le comportement des autres agents en supposant qu'ils vont effectivement respecter les normes. En particulier, les normes définissent comment les agents doivent se comporter et ce qui leur est permis de faire.

Les normes sont souvent utilisées dans le contexte des systèmes multiagents ouverts où des agents hétérogènes pouvant entrer ou sortir à tout moment du système et ayant des intérêts divergeants doivent collaborer. Les normes sont prises en compte par les agents au niveau de leur processus de prise de décision, c'est-à-dire qu'au moment où l'agent décide de l'action à exécuter ou du plan à suivre il considère les normes du système pour prendre sa décision. Il est alors possible que l'agent choisisse de ne pas suivre un comportement respectant les normes, dans ce cas il est souvent question de sanctionner l'agent pour son mauvais comportement.

L'acceptation d'une norme par un agent peut être obligatoire, dans ce cas on parle d'enregistrement, c'est-à-dire que les agents sont construits dans la perspective que les normes soient respectées. L'acceptation peut être voulue, dans ce cas c'est l'agent qui choisit lui-même les normes qu'ils considèrent comme bénéfiques pour lui [CCD99]. L'application d'une norme peut elle aussi être volontaire ou non. L'idée principale des normes reste néanmoins la possibilité que les agents ont de décider de transgresser les normes lors des choix qu'ils font sur le comportement à suivre.

Pour permettre la description des normes d'un système, on utilise généralement un langage basé sur les opérateurs déontiques d'interdiction, d'obligation et de permission. Dans une norme, on définit les conditions de violation de la norme mais aussi la ou les sanctions à appliquer aux agents violant la norme. Classiquement, les sanctions sont des pénalités sur les agents pouvant aboutir à leur exclusion du système par les autres agents suite à une perte de confiance. Pour permettre aux agents d'avoir conscience des normes (*i.e* de prendre des décisions en tenant compte des normes), on doit modifier le code de l'agent au niveau de son processus de prise de décision [VSAD04a].

Pour détecter la violation des normes par un agent, nous avons vu différentes approches basées sur la surveillance des interactions des agents. Si ces approches permettent de détecter les violations d'une certaine catégorie de normes, elles ne permettent pas généralement de détecter la transgression de normes liées au comportement interne de l'agent. Si certaines pistes ont été proposées [VSAD04b] pour mettre en place des moyens de détecter l'apparition d'une action, l'activation d'une norme et la fin d'un délai au sein d'une plateforme multiagent, il n'existe pas, à notre connaissance, de mécanismes permettant de

résoudre le problème des détections de violation de lois en dehors des interactions.

5.3.2 Le concept de loi

Pour permettre la détection de comportements indésirables au sein d'un système nous allons donc utiliser des **lois**. Dans le cadre de notre approche, une loi peut être vue comme une norme en ceci qu'elle pose une contrainte sur le comportement des agents, mais à la différence que nous partons du principe que les agents n'ont pas conscience des lois auxquelles ils sont soumis. Notre objectif est de faire en sorte que les agents suivent les comportements qu'ils souhaitent et que, par la suite, le contrôle soit effectué sans que nécessairement les agents prennent en compte les lois dans leurs décisions. Cela vient du fait que d'une part, nous cherchons à découpler le comportement des agents et le contrôle à appliquer au système et que d'autre part, nous souhaitons que le comportement du système respecte les lois. De ce fait, laisser la possibilité aux agents de décider de suivre ou non une loi nous semble plus dangereux, dans ce contexte, qu'intéressant. Enfin, nous ne proposons pas de tenir compte d'une quelconque sanction à la suite de la transgression d'une loi comme cela est le cas pour les normes.

Qu'est-ce qu'une loi ?

Les lois vont représenter les exigences sur le comportement du système multiagent pouvant être associées, par exemple, aux spécifications de l'application. Néanmoins les lois ne sont pas expressément ces spécifications car elles se placent au niveau des agents et non pas au niveau du système dans sa globalité. Passer des spécifications du système aux lois, nécessite une traduction et une agentification qui n'est pas immédiate que nous aborderons dans le chapitre 6. Les lois du système vont donc, à l'instar des normes, décrire les comportements que l'on souhaite voir ou non apparaître au sein du système.

Partant du principe qu'un agent bien construit va respecter les lois qui lui sont attribuées, nous avons vu qu'il existe malgré tout des situations où ces lois peuvent être transgressées, c'est-à-dire des situations où le comportement du système ne respecte pas ses spécifications. Nous avons vu que, dans ce cas, le système peut suivre ce que nous avons appelé des comportements indésirables. Une transgression de loi peut alors potentiellement engendrer l'apparition de comportements indésirables. Ainsi, notre approche peut se résumer à surveiller le comportement des agents et à s'assurer qu'ils respectent bien les lois qui leur sont attribuées.

Une loi ne va pas représenter directement le comportement indésirable en lui-même, mais les états ou actions pouvant entraîner l'apparition du comportement indésirable. Ce dernier étant, par définition, inattendu et de ce fait, inconnu au moment de la conception du

système, il est tout à fait impossible de le décrire directement. En revanche, il est possible de connaître les comportements ou états néfastes pour un système et sous-tendant l'apparition d'un comportement indésirable. Les lois vont donc porter sur des états et des actions que peut produire le système. Plus particulièrement, les états vont correspondre aux valeurs des sous-concepts du concept CARACTÉRISTIQUE et les actions aux sous-concepts du concept ACTION.

Ces états et actions sont associées à un agent mais il peut y avoir dans une même loi des états et des actions pouvant apparaître dans des agents différents. Lorsqu'une loi ne tient compte que des états ou des actions associées à un unique agent, nous dirons que nous sommes en présence d'une **loi monoagent**. Au contraire, lorsqu'une loi tient compte des états ou des actions associées à plusieurs agents, nous dirons que c'est une **loi multiagent**. Les lois multiagents sont des lois exprimant le contrôle à effectuer sur des comportements mettant en jeu plusieurs agents comme, par exemple, le contrôle de protocole de communication, mais aussi le contrôle de comportement d'un agent relativement à l'état ou au comportement des autres agents du systèmes.

Différents types de lois

Quelque soit le type de la loi (mono ou multiagent), nous distinguons deux classes de lois :

- * LES LOIS D'INTERDICTION. Ce sont les lois représentant des états ou des comportements interdits pour un agent ou un groupe d'agents. Elles permettent la détection de situations où un événement ne devant jamais se produire, va survenir.
- * LES LOIS D'OBLIGATION. Ce sont les lois représentant des états ou des comportements obligatoires pour un agent ou un groupe d'agents. Elles permettent la détection de situations où un événement attendu ne se produit pas.

Les lois d'interdiction vont permettre de représenter tous les états ou actions qu'on ne souhaite pas voir apparaître au sein du système. Si un agent soumis à une loi d'interdiction suit un comportement qui va entraîner sa violation, notre objectif est d'empêcher l'exécution de l'action ou l'apparition de l'état interdit. En effet, nous supposons que si un événement est interdit c'est que son apparition sera dommageable pour le système et ainsi, nous préférons prévenir l'apparition de cet événement tout en informant l'agent de la violation de la loi.

Les lois d'obligation vont permettre, quant-à-elles, de représenter tous les états ou actions que l'on souhaiterait voir apparaître au sein du système. Pour détecter la violation d'une loi d'obligation, c'est-à-dire la non-apparition de l'événement obligatoire, il est nécessaire que la loi définisse un délai d'attente de l'apparition de cet événement. Il nous est impossible

d'assurer la détection de la violation d'une loi d'obligation si elle ne fixe pas ce délai. Dire qu'il est obligatoire, par exemple, d'exécuter une certaine action, a un sens mais ne permet pas de savoir quand cette obligation n'est pas vérifiée par un agent. Il est donc indispensable de préciser qu'un événement doit apparaître au sein du système : avant un temps fixé ; avant l'apparition d'un autre événement ; dans un temps t après l'apparition d'un événement. Lorsqu'une loi d'obligation est violée, nous avons choisi de ne pas forcer l'agent à engendrer l'événement obligatoire. Autant il nous semble possible d'empêcher l'apparition d'un événement sans perturber le comportement de l'agent, autant il n'est pas envisageable d'exécuter une action ou de mettre l'agent dans un certain état sans en connaître les conséquences sur le comportement de l'agent. Il nous semble plus judicieux de simplement informer l'agent de la violation pour qu'il se charge de se réguler en conséquence.

Dans le cadre de cette thèse, nous ne tenons pas compte des lois de permissions qui sont généralement mises en jeu dans les systèmes d'agents soumis à des normes (agents normatifs). En effet, les permissions n'ont de sens que par leur prise en compte par les agents lors des choix qu'ils portent sur le comportement à suivre. Partant du principe que les agents soumis à des lois, dans le cadre de notre approche, n'ont pas conscience de ces lois, il ne nous ait pas apparu comme intéressant de laisser la possibilité d'exprimer des permissions. De plus, notre objectif étant de détecter les violations de lois, les permissions ne rentrent pas dans ce cadre, du fait qu'il n'est pas envisageable de pouvoir détecter la transgression d'un tel type de loi. En effet, dire qu'il est permis d'exécuter une certaine action et d'être dans un certain état est plus une information pour l'agent pour choisir son comportement qu'un moyen de détecter l'apparition de comportements indésirables. Nous avons donc pris la décision de ne pas inclure ce concept de permission dans nos catégories de lois.

Structure d'une loi

Une loi va se décomposer de la façon suivante :

- La partie **AGENTS CONCERNES (CA)** décrivant les agents concernés par la loi. Ces agents peuvent aussi bien être ceux auxquels la loi va s'appliquer que les agents utilisés pour décrire le contexte d'application de la loi, en particulier lorsqu'une loi concerne plusieurs agents.
- La partie **ASSERTIONS DEONTIQUES (DA)** décrivant les événements ou les états obligatoires ou interdits. Cette partie représente un ensemble d'associations entre un agent et un concept. Ce concept peut être une action ou un état. Il est possible de définir des conditions particulières sur ces concepts pour décrire des événements ou des états plus spécifiques.

- La partie **CONDITIONS D'APPLICATION (APC)** décrivant les conditions sur le contexte d'application de la loi. C'est une expression décrivant quand la partie **DA** doit être respectée relativement à un ensemble d'événements. Dans cette partie il est possible d'exprimer une notion temporelle permettant de préciser le délai d'activation de la loi. Ce temps peut correspondre à l'apparition d'un événement donné ou à une durée.

Ainsi une loi va s'appliquer à un ou plusieurs agents et tenir compte du comportement d'un ou plusieurs agents. Une loi va définir un ou plusieurs événements obligatoires ou interdits dont il faut vérifier qu'ils sont bien respectés par un ou plusieurs agents. Ces événements sont, soit l'apparition d'un état au sein d'un agent, soit l'exécution d'une action par un agent. L'observation effective du respect d'une loi ne se fait que lorsque toutes les conditions d'application sont réalisées au sein du système par un seul agent ou par un ensemble d'agents.

5.4 Une architecture introspective

Nous avons vu que pour satisfaire notre contrainte de maintien de l'autonomie et de la robustesse du système, nous avons fait le choix de distribuer le contrôle au sein de chaque agent. Ainsi, pour permettre aux agents de surveiller leurs propres comportements, de détecter les transgressions de lois et de réguler leurs comportements en conséquence, nous les dotons d'une architecture introspective. Cette méta-architecture vient s'ajouter au comportement réel de l'agent pour lui fournir, de façon transparente, les moyens de contrôler son propre comportement.

5.4.1 Composition de l'architecture

Chaque agent capable d'autocontrôle va donc avoir une méta-architecture particulière (décrite sur la figure 5.2) divisée en deux parties : une **partie comportement** et une **partie contrôle**.

La partie comportement comprend le comportement réel de l'agent et un ensemble de stratégies de régulation définissant le comportement à suivre en cas de violation d'une loi. La partie contrôle inclut les lois que l'agent se doit de respecter et les mécanismes de détection des transgressions de ces lois. La partie comportement et la partie contrôle communiquent. D'une part, la partie comportement envoie des informations sur le comportement de l'agent à la partie contrôle. D'autre part, la partie contrôle envoie des informations de transgression à la partie comportement. Cette architecture et le fonctionnement général de ses composants sont basés sur **principe de l'observateur** [DJC94].

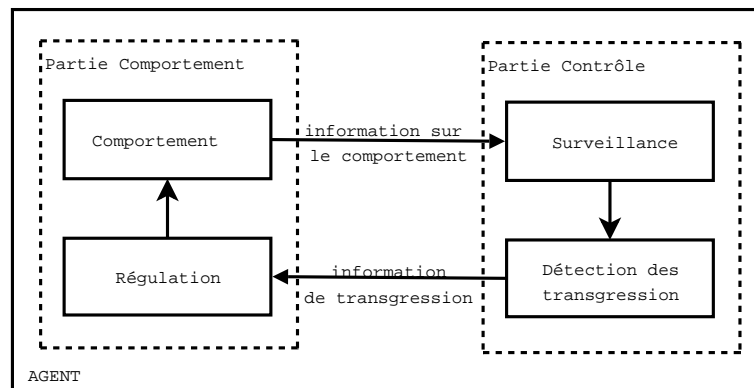


FIG. 5.2 – L'architecture générale des agents autocontrôlés

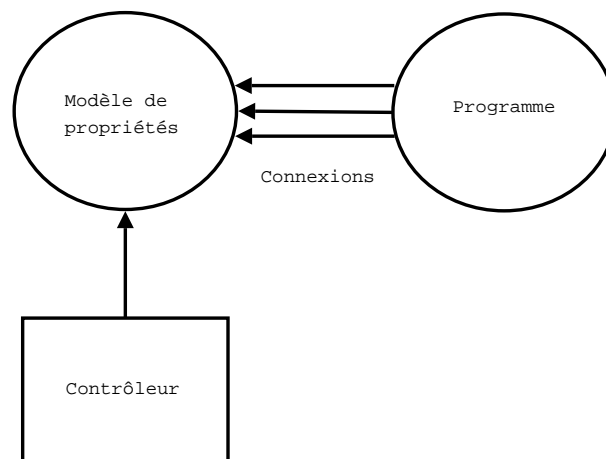


FIG. 5.3 – L'architecture du principe de l'observateur

5.4.2 Principes de l'architecture

La technique de l'observateur comme structure générale

La technique de l'observateur (fig.5.3) est une approche généralement proposée pour effectuer des tests sur des systèmes temps-réels, pour faire de la validation en ligne de systèmes parallèles et parfois distribués. Cette approche consiste à faire évoluer un programme et un modèle de propriétés sur le fonctionnement de ce programme en parallèle et de les comparer. Le modèle est lié au programme du système sous surveillance à l'aide de **points de contrôle**. Lorsque l'exécution du programme arrive sur les points de contrôle, un

contrôleur s'assure que le modèle et le programme sont consistants. Ainsi, si l'exécution du système ne correspond pas au modèle de propriétés, la vérification échoue¹.

Des réseaux de Petri pour représenter les lois

Les réseaux de Petri ont été introduits par Carl Adam Petri dans son mémoire de thèse en 1962. Les réseaux de Petri, consistant en une représentation mathématique pour la modélisation de système sont généralement utilisés pour l'étude des systèmes dynamiques, concurrents et discrets, en particulier dans la perspective de prouver des propriétés sur de tels systèmes. Un réseau de Petri peut être assimilé à un système composé de deux parties distinctes :

- * Une partie statique/structurelle
- * Une partie dynamique/comportementale

La partie structurelle se constitue d'un graphe orienté bipartite valué $\langle P, T, Pre, Post \rangle$ avec :

- * P : un ensemble fini non-vide de places $\{p_1, p_2, \dots, p_n\}$.
- * T : un ensemble fini non-vide de transitions $\{t_1, t_2, \dots, t_m\}$
- * $Pre : P \times T \rightarrow \mathbb{N}$, la matrice d'incidence avant, correspondant aux arcs directs reliant les places aux transitions. $Pre(p, t)$ donne la valeur entière i associée à l'arc allant de p à t .
- * $Post : P \times T \rightarrow \mathbb{N}$, la matrice d'incidence arrière, correspondant aux arcs directs reliant les transitions aux places. $Post(p, t)$ donne la valeur entière j associée à l'arc allant de t à p .

On définit généralement :

- * Une place p comme étant une place d'entrée d'une transition t si $Pre(p, t) > 0$.
- * Une place p comme étant une place de sortie d'une transition t si $Post(p, t) > 0$.

Lorsque Pre et $Post$ prennent leur valeur dans $\{0,1\}$, le réseau est qualifié de réseau **ordinaire**. Le marquage du réseau représente l'état du système modélisé.

Définition 1 : Marquage Soit $N = \langle P, T, Pre, Post \rangle$ un réseau de Petri. Un marquage $M : P \rightarrow \mathbb{N}$ est une association de jetons aux places du réseau de Petri. Le marquage M peut être écrit comme un vecteur $M = \langle M_1, \dots, M_n \rangle$ où $n = |P|$ et

¹Cette approche est souvent mise en place au niveau matériel pour détecter les événements apparaissant dans le système pour limiter les perturbations de fonctionnement de l'application en cours de test.

chaque $M_i \in \mathbb{N}, i = 1, \dots, n$. On écrit généralement $\langle N, M \rangle$ pour représenter un réseau de Petri N avec un marquage M .

Le marquage d'un graphe permet de déterminer quelles transitions sont susceptibles d'être franchies.

Définition 2 : Transition tirable Soit $N = \langle P, T, Pre, Post \rangle$, un réseau de Petri avec un marquage M , soit $M(p)$, le nombre de jetons contenus dans la place $p \in P$ et soit $t \in T$, une transition. La transition t est tirable (ou validée) dans $\langle N, M \rangle$ si et seulement si $\forall p \in P : M(p) \geq Pre(p, t)$.

Définition 3 : Tirage d'une transition Soit $N = \langle P, T, Pre, Post \rangle$, un réseau de Petri avec un marquage M , soit $M(p)$, le nombre de jetons contenus dans la place $p \in P$ et soit $t \in T$, une transition tirable. Le tirage d'une transition $t \in \langle N, M \rangle$ engendre un nouveau marquage M' , noté $M|t > M'$ tel que $M'(p_i) = M(p_i) - Pre(p_i, t) + Post(p_i, t)$. $Pre(p_i, t)$ représente le nombre de jetons nécessaires dans p_i au tirage de la transition t et $Post(p_i, t)$ représente le nombre de jetons ajoutés à la place p_i quand la transition t est tirée.

Le comportement dynamique d'un réseau de Petri s'exprime à l'aide des changements au niveau du marquage. Un marquage change quand une transition est tirée et une transition peut être tirée lorsqu'elle est activée.

Définition 3 : Exécution Soit $N = \langle P, T, Pre, Post \rangle$, un réseau de Petri avec un marquage M , soit $S = (t_1, \dots, t_n)$ avec $t_i \in T$, une séquence finie de transition de T , et soit T^* , l'ensemble de toutes les séquences finies possibles construites à partir des transitions de T . La séquence S est une exécution du réseau marqué $\langle N, M \rangle$ si et seulement si, elle est tirable dans le réseau $\langle N, M \rangle$ et conduit au marquage M' , noté $M|S > M'$. On a $M|S > M'$ si et seulement si :

1. Soit $S = \epsilon$ (la séquence vide), alors $M' = M$.
2. Soit $S = tS'$, avec $S' \in T^*$ et $t \in T$, alors $\exists M''$ tel que $M|t > M''$ et $M''|S' > M'$.

L'ensemble des séquences possibles d'exécutions d'une réseau de Petri N partant d'un marquage M à un marquage M' est défini par l'ensemble $E(N, M, M') = \{S \in T^*, M|S > M'\}$.

Une première extension des réseaux de Petri qui nous intéresse tout particulièrement dans le cadre de notre travail de thèse, est l'utilisation au sein d'un réseau d'arcs inhibiteurs.

Définition 4 : Arc inhibiteur La présence d'un arc inhibiteur entre une place p et une transition t signifie que la transition t n'est tirable que si $M(p) = 0$.

On représentera alors un réseau de Petri avec des arcs inhibiteurs par un tuple $\langle P, T, Pre, Post, I \rangle$ avec I , l'ensemble fini des arcs inhibiteurs tel que, on note $Pre^* : P \times I \rightarrow \mathbb{N}$ la matrice d'incidence avant correspondant aux arcs inhibiteurs reliant les places aux transitions.

Une autre extension des réseau de Petri est celle proposée par Merlin [Mer74] consistant en l'utilisation de notion temporelle au niveau des transitions du réseau. Dans ce modèle, à chaque transition est associée une contrainte temporelle sous la forme d'un intervalle.

Définition 5 : Réseau de Petri t-temporel Un réseau de Petri t-temporel est définie par un tuple $\langle P, T, Pre, Post, IS \rangle$ où IS est une fonction désignant les dates de tir au plus tôt et au plus tard d'une transition et telle que :

- * $IS : T \rightarrow \mathbb{Q}^+ \times (\mathbb{Q}^+ \cup \infty)$ avec \mathbb{Q}^+ l'ensemble des nombres rationnels positifs.
- * $t_i \rightarrow IS(t_i) = a_i; b_i$ avec $a_i \leq b_i$.

$IS(t_i)$ définit l'intervalle statique associée à la transition t_i avec pour borne inférieure a_i et pour borne supérieure b_i . Ainsi une transition t_i doit être validée pendant au moins a_i unités de temps avant de pouvoir être tirée et ne peut rester validée au-delà de b_i unités de temps sans être tirée.

5.4.3 Fonctionnement interne

La technique de l'observateur va donc être mise en place au sein de la méta-architecture que nous fournissons aux agents pour leur permettre de contrôler leur comportement. La partie comportement correspond au programme sous surveillance, la partie contrôle représente le contrôleur et les lois du système représentent les propriétés sur l'exécution du programme, au sein de l'approche de l'observateur.

Ainsi, des points de contrôle vont être insérés dans le programme des agents au niveau du code associé aux concepts représentant les actions et états des agents. Une partie contrôle va être associée à chaque agent pour effectuer la surveillance de leur comportement vis-à-vis des lois. Nous avons choisi de représenter ces lois sous la forme de réseau de Petri. A chaque loi va correspondre un réseau de Petri ordinaire $PN = \langle P, T, Pre, Post, I, IS \rangle$.

Soit un réseau de Petri PN représentant une loi. Les transitions du réseau PN vont correspondre aux différents événements que l'on souhaite surveiller et détecter au sein de l'agent soumis à cette loi. A chaque transition est associée une action ou un changement d'état exprimé dans la loi, en particulier dans les parties CONDITIONS D'APPLICATION et ASSERTIONS DÉONTIQUES. Pour constituer le réseau PN , chaque transition se voit

associée une place précédente et une place suivante de façon à respecter l'enchaînement des événements décrits dans la loi.

Lorsqu'une obligation est exprimée dans la partie ASSERTIONS DÉONTIQUES de la loi, l'arc reliant la place précédant la transition associée à l'obligation est un arc simple. Lorsqu'une interdiction est exprimée, l'arc reliant la place précédant la transition associée à l'interdiction est un arc inhibiteur. Enfin, dans la partie CONDITIONS D'APPLICATION, tous les arcs sont des arcs simples (*c.f.* Figure 5.4).

Lorsqu'il est question de temps (un délai exprimé en seconde de l'apparition d'un événement) au niveau des conditions d'applications de la loi, une transition associée à un intervalle spécifique est utilisée pour représenter les conditions temporelles. Cet intervalle est de la forme $[t, t]$ où t représente le délai exprimé dans la loi. Ainsi, lorsque le temps t est écoulé, la transition est automatiquement tirée. Dans le reste du réseau, on omettra généralement les intervalles qui par défaut sont de la forme $[0, \infty]$.

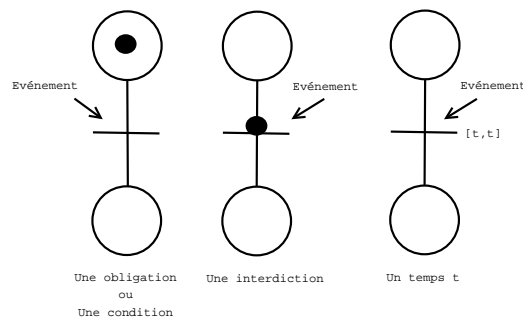


FIG. 5.4 – Représentation sous forme de réseau de Petri

Une loi correspond donc à un réseau de Petri qui est embarqué dans la partie contrôle des agents soumis à cette loi. Une partie contrôle contient donc l'ensemble des réseaux de Petri représentant les lois auxquelles est soumis l'agent qui lui est associé, ainsi qu'un mécanisme de détection de transgression des lois basé sur la surveillance de l'exécution du réseau de Petri (c'est-à-dire l'évolution de son marquage au cours du temps).

Les points de contrôle insérés dans le code de comportement de l'agent sont reliés aux transitions des réseaux de Petri correspondants à l'action ou à l'état signalé par le point de contrôle. Ainsi lorsque le programme de l'agent arrive au niveau d'un point de contrôle, ce dernier génère un événement qui est transmis directement aux transitions des réseaux attendant cet événement. La partie contrôle se charge alors de vérifier que les jetons sont dans les bonnes places au moment de la réception de cet événement, c'est-à-dire qu'elle vérifie que les transitions activées via les points de contrôle sont effectivement tirables.

Si c'est le cas, la partie contrôle fait évoluer le marquage des réseaux correspondants. Si le marquage ne correspond pas à l'événement reçu, c'est-à-dire si la transition n'est pas tirable dans un ou plusieurs réseaux, la partie contrôle envoie une ou plusieurs informations de transgression à la partie comportement, dans la perspective de lancer une stratégie de régulation.

Dans la technique de l'observateur, le programme se termine lorsqu'une inconsistance vis-à-vis des modèles de propriétés est détectée. Dans notre approche, nous souhaitons que les agents puissent continuer à s'exécuter malgré la transgression des lois. En effet, notre objectif étant d'empêcher l'échec de l'application, nous proposons de détecter les comportements pouvant faire échouer le système dans la perspective que les agents régulent leur comportement en conséquence. Ainsi, les agents doivent être en mesure de se soustraire aux comportements indésirables lors de leurs régulations et de faire en sorte que l'exécution du système puisse perdurer. C'est pour cette raison que toute transgression génère une information qui va être reçue par la partie comportement de l'agent, pour permettre à ce dernier, grâce à l'ensemble des stratégies de régulation qui lui ont été implantées, d'analyser le problème et de le résoudre.

5.5 Digression

5.5.1 D'une vision *Asimovienne* vers une vision plus sociale

Notre objectif étant de pouvoir garantir le bon fonctionnement d'un système multiagent en contrôlant l'émergence de comportements au sein du système, nous sommes partis du principe qu'il nous fallait empêcher autant que possible l'apparition de comportements indésirables. De ce fait, il nous semblait indispensable que les agents ne puissent pas transgresser les lois qui leur sont attribuées. L'idée était alors, en quelque sorte, de fournir aux agents des lois sur le principe des lois de la robotique de Isaac Asimov.

Dans les nouvelles d'Asimov, les robots possèdent un cerveau "positronique" dans lequel est implanté un ensemble de trois ou quatre lois que les robots sont dans l'obligation de respecter tout au long de leur existence. Il leur est impossible de transgresser les lois et de ce fait, leurs comportements peuvent sembler incohérents² dans certaines situations. Néanmoins, il apparaît comme plus important que les robots ne puissent violer ces lois que d'avoir des comportements toujours cohérents. Les trois lois de la robotique implantées au sein des robots dans les romans d'Asimov sont les suivantes, auxquelles vient s'ajouter la loi zéro qui est apparu par la suite dans la perspective de réduire justement les incohérences de comportements des robots :

²Pour ne pas dire loufoques !

- * **Première Loi** : Un robot ne peut nuire à un être humain ni laisser sans assistance un être humain en danger.
- * **Seconde Loi** : Un robot doit obéir aux ordres qui lui sont donnés par les êtres humains, sauf quand ces ordres sont incompatibles avec la Première Loi.
- * **Troisième Loi** : Un robot doit protéger sa propre existence tant que cette protection n'est pas incompatible avec la Première ou la Deuxième Loi.
- * **Loi Zéro** : Un robot ne peut nuire à l'Humanité ni laisser, par son inaction, l'Humanité être en danger.

A la lueur de ces lois, il est aisé de se rendre compte à quel point il est important qu'elles soient toujours respectées. Nous sommes donc partis du principe que les lois fournies pour un système multiagent dans notre contexte d'applications critiques, ne devaient en aucun cas être transgressées, du fait des problèmes potentiels que les comportements indésirables peuvent générer. Ainsi, comme chez Asimov, un agent soumis à notre approche de contrôle, ne peut transgresser une loi, en particulier une loi d'interdiction. Il est alors possible d'exprimer et de contrôler des lois assimilées à la Première Loi et à la Loi Zéro des romans d'Asimov. En revanche, nous avons vu précédemment que nous avons fait le choix de ne pas forcer un agent à exécuter un comportement obligatoire lorsque ce dernier n'a pas été effectué dans le délai spécifié. Aussi, notre approche, si elle permet la description de loi telles que la Seconde et Troisième Loi, ne peut contrôler de telles lois. D'une part car les lois d'obligation décrites par Asimov ne comportent aucun délai permettant de détecter la violation et d'autre part car nous ne forcerons pas les agents à obéir aux ordres, quand bien même nous pourrions détecter les transgressions. Si notre idée première était de mettre en place des lois dans des agents suivant le principe des lois de la robotique de Asimov, il s'avère qu'en pratique nous ne pouvons réellement garder ce principe que pour des lois d'interdiction. La différence principale entre nos lois et celles d'Asimov vient du fait que les robots d'Asimov sont construits avec ces lois et qu'ils leur est impossible de ne pas les suivre, leur comportement étant régit par elles. Il n'y a pas de détection de violation à effectuer car celles-ci ne peuvent exister. Avec notre approche, nous ne pouvons que détecter les transgressions de lois et empêcher l'exécution de certaines actions ou de certains états interdits au moment de cette détection.

Néanmoins, comme chez Asimov, nous avons supposé que les agents n'avaient pas conscience des lois au moment de leur prise de décision. Bien entendu le concepteur de l'agent a tenu compte des lois lors de son développement et l'agent se doit d'avoir une connaissance minimale des lois auxquelles il est soumis, pour pouvoir réguler son comportement lorsqu'il est informé de la violation d'une loi, mais la mise en place des lois, la surveillance des comportements et la détection des violations se fait de façon transparente pour l'agent. Ce

dernier peut agir à sa guise tant que son comportement respecte les lois, sans les prendre en considération et c'est seulement lorsque son comportement dévira, lorsqu'il tendra vers la violation d'une loi, qu'il en sera informé.

Ce principe d'enrégimenter des agents, en leur fournissant un mécanisme de surveillance et de détection des violations de lois, nous semblait primordial dans le contexte où nous nous plaçons. À l'usage, nous nous sommes aperçus que notre approche pouvait se généraliser simplement à la mise en place de normes (plus sociales en quelque sorte que nos lois) au sein d'un système multiagent. Dans ce cadre, les agents ont connaissance des normes et conscience de leur violation. Un agent soumis à des normes peut généralement choisir de les transgresser, il porte une réflexion sur les normes qui lui sont attribuées et se voit pénalisé en cas de transgression. Il s'avère alors que notre approche pourrait servir de moyen de détection de violation des normes. Les agents suivraient consciemment un comportement transgressant une norme et le mécanisme de détection que nous leur implantons détecterait cette transgression et en informerait l'agent. Pour mettre en place les pénalités lors de violation, il suffirait dans les stratégies de régulation, non pas de fournir un comportement permettant à l'agent de se remettre en accord avec les normes, mais de préciser les pénalités à appliquer à l'agent.

Cette généralisation de notre approche nécessiterait alors d'appliquer une unique modification qui serait la suppression de l'étape empêchant les agents de transgresser des lois d'interdiction. Et quand bien même nous ferions le choix de ne pas supprimer cette étape pour pouvoir garantir la non-apparition d'événements interdits, il serait alors tout à fait envisageable d'exécuter le comportement interdit lors de la phase de régulation, qui elle, peut ne pas être soumise à des lois.

Le seul problème que nous voyons vis-à-vis de l'utilisation de notre approche pour mettre en place des normes au sein des agents est que nous ne tenons pas compte des permissions. En effet, le concept de normes inclut aussi bien des interdictions et des obligations que des permissions. Or, compte tenu de notre objectif de détecter les transgressions et du fait qu'il nous semble tout à fait impossible de détecter une quelconque violation d'une permission, nous n'avons pas étudié la possibilité de décrire des permissions pour les agents. Pour pouvoir pleinement mettre en place des normes au sein d'un système multiagent, il faudrait donc laisser l'opportunité de décrire des lois de permission à fournir aux agents, mais pour lesquelles il n'y aurait pas de mécanisme de détection de violation particulier.

5.5.2 Vers un appareil psychique

Lorsque nous avons commencé à réfléchir à la méta-architecture que nous souhaitions fournir aux agents soumis à notre approche de contrôle, nous avons naturellement tenté de

nous éloigner du domaine informatique pour nous rapprocher d'un domaine qu'on pourrait qualifier de plus "humain". Ainsi, à la suite d'une première ébauche de la structure de notre méta-architecture, il nous a semblé évident que celle-ci tendait vers une structure bien connue : l'appareil psychique proposé par Sigmund Freud [Fre23].

Le principe de plaisir

Dans la théorie psychanalytique, le principe de plaisir fait partie intégrante du fonctionnement du psychisme humain. Ce principe correspond à une tendance à satisfaire de façon immédiate ses désirs. Le principe de plaisir est provoqué par une tension déplaisante qui entraîne le besoin d'obtenir une satisfaction immédiate dans la perspective de faire baisser cette tension, soit par évitement du déplaisir, soit par production de plaisir. La relation entre le plaisir et le déplaisir correspond donc à une quantité d'excitation qui augmente avec le déplaisir et diminue avec le plaisir. L'objectif du principe de plaisir est de ce fait de maintenir cette quantité à un taux assez bas ou à un taux constant pour atteindre une certaine satisfaction, un certain repos, un apaisement.

Le psychisme humain est régi par ce principe de plaisir, mais il n'est pas seul. En effet, si le principe de plaisir était le seul principe à entrer en jeu dans le fonctionnement du psychisme, tout processus psychique conduirait ou serait accompagné de plaisir. Ce qui n'est assurément pas le cas.

Nous citerons S. Freud [Fre20] :

Il existe dans le psychisme une forte tendance au principe de plaisir mais certaines autres forces ou conditions s'y opposent de sorte que l'issue finale ne peut pas toujours correspondre à la tendance du plaisir.

Le principe de réalité

De ce fait, un autre principe doit être pris en considération : le principe de réalité. Respecter ce principe consiste à prendre en compte les exigences du monde réel et les conséquences de ses actes. Dès lors que l'être humain se retrouve en relation avec le monde extérieur, qu'il est soumis aux contraintes de la vie quotidienne, il n'est plus possible pour le psychisme d'être uniquement guidé par la satisfaction immédiate de ses désirs, le principe de réalité entre alors en jeu.

Le principe de réalité permet de prendre en considération les désirs et les devoirs associés à ce qui nous entoure. La satisfaction du plaisir peut alors se retrouver reportée, ou le plaisir peut être restreint, pour permettre la prise en compte des exigences du monde

extérieur. Nous ne sommes plus uniquement dans la consommation pour atteindre un état satisfaisant mais nous devons aussi produire pour notre entourage de quoi le satisfaire.

Nous pouvons résumer la différence entre le principe de plaisir et le principe de réalité de la façon suivante [Lap] :

Principe de Plaisir	Principe de Réalité
Satisfaction immédiate	Satisfaction Remise
Plaisir (évitement de la douleur)	Restriction du plaisir (tolérance à la douleur)
Consommation	Production (travail)
Absence de refoulement	Sécurité

Les principes de plaisir et de réalité ne sont pas tant deux principes opposables mais une continuité de l'un vers l'autre pour maintenir un état satisfaisant de plaisir, tout en le reportant pour permettre de ne pas consommer trop rapidement ce plaisir et atteindre un stade d'excitation le plus bas possible, l'absence de toute stimulation.

Nous citerons S. Freud :

Le premier cas où l'on rencontre une telle inhibition du principe de plaisir nous est bien connu, il est dans l'ordre. Nous savons en effet que le principe de plaisir convient à un mode primaire de travail de l'appareil psychique et qu'en ce qui concerne l'auto-affirmation de l'organisme soumis aux difficultés du monde extérieur, il est d'emblée inutilisable et même extrêmement dangereux. Sous l'influence des pulsions d'auto-conservation (...), le principe de plaisir est relayé par le principe de réalité

Le Ca, le Moi et le Surmoi

Au commencement de la théorie de Freud sur le principe de plaisir et le principe de réalité, il y avait le Moi comme unique entité du psychisme soumis à ces deux principes et tendant à satisfaire les désirs de l'individu tout en respectant les contraintes de son entourage. Le Moi était le siège de la conscience et le lieu des manifestations inconscientes.

En 1923, dans "le moi et le ça" [Fre23], le Ca se distingue du Moi comme entité soumise uniquement au principe du plaisir. Le Moi devient alors la partie du Ca modifiée par l'influence du monde extérieur, il en est son représentant dans le psychisme. Puis vient s'ajouter une nouvelle entité, un niveau "supérieur" dans le Moi que Freud nomme le Surmoi. Ce Surmoi est une partie largement inconsciente du Moi dans laquelle se situent, par exemple, les interdits de l'autorité parentale.

Le Ca est la couche la plus ancienne dans le psychisme humain et la plus étendue. C'est le domaine de l'inconscient et des instincts primaires. Le Ca ignore le temps qui passe, les concepts moraux et n'est pas sujet aux contradictions. Son objectif principal est la satisfaction immédiate de ses besoins et désirs, il est dominé entièrement par le principe de plaisir. Le Ca ne vise en aucun cas à l'auto-conservation.

Le Moi est le lien entre le Ca et l'environnement. La plus grande partie du Moi se situe dans le domaine conscient du psychisme. Le Moi est le siège de la raison, il coordonne et modifie les tendances instinctuelles du Ca pour être en accord avec le monde extérieur, il est donc dominé par le principe de réalité. Le Moi, contrairement au Ca tend à l'auto-conservation et à l'absence de contradiction.

Enfin le Surmoi est une structure qui se développe parallèlement au Moi. Une grande partie de Surmoi est inconsciente. C'est dans le Surmoi que se tient l'influence parentale et la morale sociale, c'est le représentant des exigences éthiques de l'Homme. Sa fonction est d'établir un modèle idéal pour le Moi (l'idéal du Moi) à partir duquel il juge le comportement du Moi. C'est aussi la cause de la plupart des refoulements.

Le Moi se retrouve alors comme le médiateur entre les pulsions du Ca, les contraintes du Surmoi et les exigences du monde extérieur dans l'objectif de maintenant l'équilibre psychique.

Et notre méta-architecture ?

Ainsi pouvons-nous rapprocher notre méta-architecture de l'appareil psychique de Freud, constitué du Ca, du Moi et du Surmoi. Nous avons vu précédemment que l'architecture que nous fournissons aux agents est composée de deux grandes parties que sont la partie comportement et la partie contrôle. Cette dernière contient les lois du système et se charge de l'observation du comportement de l'agent et de la détection du non-respect des lois. La partie comportement représente le comportement réel de l'agent.

Nous pouvons donc dans un premier temps assimiler la partie contrôle au Surmoi. En effet, en plus d'être le siège des lois auxquelles est soumis un agent, le contrôle que cette partie effectue se fait de façon totalement inconsciente pour l'agent. La partie contrôle contient les critères permettant de concevoir le comportement idéal que devrait suivre l'agent et se charge alors de prévenir la partie comportement de tout éloignement de cet idéal.

La partie comportement serait alors la représentation, à la fois du Ca et du Moi. Cette partie contient le comportement réel de l'agent et les comportements de régulation. Les comportements de régulation peuvent être assimilés au processus du Moi lui permettant de maintenir un équilibre entre les contraintes de la partie contrôle, les désirs du comporte-

ment réel et les exigences du monde extérieur³. Le comportement réel de l'agent serait le Ca associé à une partie du Moi se chargeant de satisfaire les buts et désirs de l'agent (son Ca) tout en tenant compte du monde extérieur, à l'aide des interactions qu'il entretient avec les autres agents du système. C'est au niveau du modèle d'agent que se jouerait alors la distinction plus précise entre les deux concepts de Ca et de Moi.

Cette analogie nous permet de consolider notre architecture et de justifier le fait que ce sont effectivement les agents qui contrôlent leur propre comportement et ce, de façon inconsciente, en s'auto-surveillant et en s'assurant du respect des lois. De plus, partant du principe que la méta-architecture est partie intégrante de l'agent, tout comme l'appareil psychique fait partie intégrante de l'être humain, nous pouvons avancer que l'autonomie des agents est préservée malgré le fait que la partie contrôle tente d'inhiber de façon inconsciente certains comportements de l'agent. Nous pouvons même aller jusqu'à proposer l'idée que cette méta-architecture, en fournissant aux agents les moyens de porter une réflexion sur leur propre comportement, leur ajoute une part d'autonomie. Si l'autonomie est le fait de se régir d'après ses propres lois, nous proposons un mécanisme de prise en compte des lois par les agents eux-mêmes et d'analyse du respect de ces lois, leur permettant de faire valoir leur autonomie.

Néanmoins, nous n'avons pu pousser plus avant cette analogie. De ce fait, il nous semblerait intéressant d'étudier le comportement du Moi dans ces processus de traitement de conflit entre "ses 3 tyrans" pour essayer de généraliser la partie régulation. De même, l'étude de la psychologie des foules et en particulier le traitement fait au Surmoi dans les mouvements de foule, nous permettrait peut-être de nous guider dans la justification du comportement de notre méta-architecture lorsque les lois concernent plusieurs agents. Mais à ce jour, nous avons pu trouver satisfaction pour ces différents points.

Conclusion

Pour permettre la description du contrôle à appliquer aux agents, nous avons fait le choix d'utiliser des lois. Ces lois correspondent à des comportements redoutés et souhaités au sein du système. Elles s'appliquent à un ou plusieurs agents et peuvent prendre en compte le comportement d'un agent ou d'un ensemble d'agents. Pour décrire les comportements dans les lois, nous proposons d'utiliser une ontologie contenant des concepts représentant les états et les actions que peuvent effectuer généralement des agents. Cette ontologie pouvant être, et nous verrons comment au chapitre suivant, étendue pour spécifier la description des comportements.

³Une régulation se fait généralement en fonction du contexte de l'agent et de son entourage

Pour permettre aux agents d'effectuer le contrôle de leur propre comportement, c'est-à-dire surveiller l'apparition de comportements indésirables en leur sein, nous leur fournissons une méta-architecture qui va se charger de l'observation du comportement de l'agent et la détection des transgressions de lois. Lorsqu'une loi est transgressée un comportement de régulation est exécuté pour tenter de soustraire l'agent au comportement indésirable pouvant potentiellement apparaître suite à la violation de la loi.

Enfin, nous avons vu que partant du principe que les agents sont dans l'obligation de suivre les lois qui leur sont attribuées (en particulier les lois d'interdiction) comme dans le cadre des nouvelles de I. Asimov, nous pouvons aisément étendre notre approche à la détection de la violation de normes au sein d'un système multiagent. Nous avons vu également que notre architecture pouvait se voir comme une représentation de l'appareil psychique de Freud et ainsi justifier notre approche d'autocontrôle et la préservation de l'autonomie. Dans la suite de ce mémoire nous allons aborder les moyens pour mettre effectivement en place notre approche de contrôle au sein d'un système multiagent.

Chapitre 6

La génération des agents autocontrôlés

L. m'a expliqué tout ce qui sera fait et tout ce que cela engendrera sur ma personne. La perspective de me retrouver bridé par une entité supérieure me déplaît quelque peu mais L. me soutient fermement que vous autres êtes fait ainsi... vous, les humains. L. me dit que vos vies sont régies par un ensemble de lois sociales et morales qui font de vous ce que vous êtes. J'aspire à la croire... en attendant le noir.

*Lucy Westenra,
The log book of Ana I.*

Avant-Propos

Reste à présent à détailler les différentes étapes de la génération automatique des agents autocontrôlés. Rappelons que notre objectif n'était pas seulement de proposer une approche de surveillance et de détection de mauvais comportements au sein des agents mais aussi de délester de cette charge les développeurs de l'application, pour leur permettre de se concentrer sur l'élaboration de comportements d'agents autonomes et intelligents. Nous allons donc aborder dans ce chapitre les points qui doivent être faits manuellement et comment ils doivent être faits, pour permettre à d'autres d'être faits automatiquement. Nous allons décrire aussi bien les moyens mis en place pour aider à la partie manuelle du travail que ceux utilisés pour modifier et générer les agents. Enfin, nous nous attarderons

sur les spécificités de la mise en place de notre approche lorsqu'une loi concerne plusieurs agents.

6.1 Préambule

Dans la perspective de satisfaire l'ultime contrainte que nous nous sommes posée pour concevoir notre approche de contrôle, c'est-à-dire :

C4. Automatiser autant que possible la mise en place du contrôle.

nous allons générer de façon automatique les agents autocontrôlés à partir d'un ensemble de données fournies par les concepteurs de l'application. Nous avons tenté de réduire au maximum le travail à effectuer manuellement pour que cela reste cohérent. De ce fait, tout ce qui touche à la description du contrôle, c'est-à-dire la description des lois et des concepts liés à l'application, est faite manuellement car elle est trop dépendante du système sous contrôle pour être automatisable. En revanche, tout ce qui va se rapporter à la surveillance et la détection va se faire de façon automatique, tout comme la mise en place de la méta-architecture.

Un regret néanmoins reste la description et la mise en place des stratégies de régulation que nous n'avons pas réussi à automatiser. S'il semblait au départ possible d'exprimer des stratégies de régulation suivant le même principe que les lois, il s'est avéré que ces stratégies étaient généralement beaucoup trop dépendantes du domaine de l'application et du contexte des agents, voire du système, pour pouvoir réellement les décrire de façon aussi synthétique. De ce fait, la définition des stratégies de régulation restera à ce jour un travail manuel à effectuer par les développeurs des agents en connaissance des lois.

6.2 Un peu de travail manuel

Pour pouvoir générer les agents autocontrôlés il est nécessaire de procéder, dans un premier temps, de la façon suivante :

1. Décrire l'application, c'est-à-dire les caractéristiques des agents, en utilisant l'ontologie d'agents proposée (*cf.* partie 5.2).
2. Décrire les lois à vérifier (*cf.* partie 5.3).
3. Décrire les régulations à effectuer suivant les différentes lois transgressées.

6.2.1 La description de l'application

La description de l'application se divise en trois étapes :

1. Décrire le ou les modèles d'agents utilisés pour concevoir les agents. Cette description est effectuée par le ou les concepteurs des modèles d'agents.
2. Décrire le comportement effectif des agents, leur architecture. Cette description est effectuée par le ou les concepteurs des agents.
3. Décrire les liens qui existent entre ces deux points précédents d'une part, et l'implémentation des agents, d'autre part.

Dans un premier temps, partant de l'ontologie de base fournie, contenant les concepts généraux permettant de décrire le comportement et la structure des agents, le ou les concepteurs des modèles vont décrire les caractéristiques des modèles d'agent utilisés. Cette description ne pouvant forcément se réduire à l'utilisation de l'ontologie fournie, le ou les concepteurs des modèles ont la possibilité de l'étendre en ajoutant des sous-concepts et/ou des instances des concepts déjà présents ou nouvellement ajoutés. Cette ontologie étendue va alors représenter le ou les modèles qui vont être utilisés pour concevoir les agents. En particulier, elle contient les concepts essentiels permettant de mettre en avant les caractéristiques des agents du système.

Une fois l'ontologie de description du modèle complétée, le concepteur des agents va pouvoir, à son tour, ajouter des concepts. De part sa connaissance du comportement réel des agents, il est en mesure d'instancier les concepts existants pour raffiner la description de l'application. Néanmoins, il n'ajoutera à ce stade que des instances de concepts pour permettre de rester uniquement au niveau du modèle. Cette contrainte est en relation avec la dernière étape de la description de l'application, c'est-à-dire fournir les liens entre les concepts abstraits de l'ontologie et l'implémentation du système.

Pour cette dernière étape, nous avons fait le choix de nous intéresser uniquement à l'implémentation du ou des modèles d'agents. Aussi, le concepteur du modèle doit-il fournir les correspondances entre l'ontologie qu'il a créée et le code du ou des modèles. Cette description n'est à faire qu'une seule fois par modèle et peut donc être validée de façon à garantir que tout concept de l'ontologie finale a une correspondance valide avec le code du modèle. De ce fait, en demandant aux concepteurs des agents de ne fournir que des instances de concepts, il ne leur est pas nécessaire de fournir également les liens entre les instances et le code des agents. D'une part, cela permet de réduire le travail des développeurs et concepteurs des agents au niveau de la mise en place du contrôle, d'autre part, il ne nous semblait pas concevable de demander des liens aussi spécifiques à l'implémentation des agents, et de les valider à chaque fois. Ainsi, les instances de concepts permettant de décrire les agents en eux-mêmes auront un lien vers l'implémentation, par l'intermédiaire des concepts qu'elles étendent.

Pour permettre cette première étape de la mise en place du contrôle au sein des agents, nous fournissons une structure à respecter pour écrire les sous-concepts et les instances à ajouter à l'ontologie de base ainsi que les liens entre ces concepts et l'implémentation du modèle.

Pour décrire un sous-concept, il convient d'utiliser la structure suivante :

```
concept(<SubConceptName>, <ConceptName>,
      [<att1> : <type1>,
       <att2> : <type1>,
       ...
       <attn> : <typen>]).
```

où *SubConceptName* est le nom du sous-concept à ajouter et *ConceptName* le nom du concept dont ce sous-concept hérite. L'ensemble $\{att_1, \dots, att_n\}$ représente les attributs de ce sous-concept tandis que l'ensemble $\{type_1, \dots, type_n\}$ représente les types de chaque attribut.

Par exemple, si nous souhaitons décrire le sous-concept *Addition* qui est une ACTION dont les attributs sont *elemX*, *elemY* et *result* de type *value*, nous le ferons de la façon suivante :

```
concept(addition, action, [elemX : value, elemY : value, result : value]).
```

La structure à utiliser pour la déclaration d'une instance est proche de celle des sous-concepts :

```
instance(<InstanceName>, <ConceptName>,
        [<att1> : <instValue1>,
         <att2> : <instValue2>,
         ...
         <attn> : <instValuen>]).
```

où *InstanceName* représente le nom de l'instance et *ConceptName* le nom dont concept dont elle est instance. L'ensemble $\{att_1, \dots, att_n\}$ est l'ensemble des attributs de cette instance devant être égale aux valeurs définies dans l'ensemble $\{instValue_1, \dots, instValue_n\}$.

Par exemple, pour définir une instance *EnvoiMessageRequest* du concept *EnvoiMessage* dont l'attribut *message* correspond à la valeur 'request', nous écrivons :

```
instance(envoiMessageRequest, envoiMessage, [message : 'request']).
```


Une fois les concepts représentatifs du modèle définis, il faut préciser les liens entre ces concepts et l'implémentation du modèle. Pour définir ces liens nous allons utiliser la structure suivante :

```
hook(ConceptName, MethodStructure, [AttributsSet :AccessorSet]).
```

où `ConceptName` représente le nom du concept dont on veut définir le lien. `MethodStructure` définit explicitement le lien vers l'implémentation du modèle, c'est-à-dire la méthode permettant de représenter ce concept au niveau de l'implémentation. `AttributSet` contient l'ensemble des attributs associés au concept que nous allons pouvoir récupérer par l'intermédiaire de ce lien et, enfin, `AccessorSet` définit les moyens d'accès à la valeur de chaque attribut. Ces moyens d'accès peuvent être directement des appels de méthodes que l'on peut définir à l'aide de `call(MethodName)` ou en récupérant les valeurs des paramètres de la méthode associées au concept à l'aide de `param(nbParam)`.

Par exemple, pour décrire les liens associés aux concepts précédemment définis on écrira les liens de la façon suivante, sachant qu'on ne définit pas les liens pour des instances¹ :

```
hook(addition, addition/3, [elemX :param(1),elemY :param(2),result :param(3)]).
hook(envoiMessage, envoiMessage/3, [message :param(2)]).
```

Le concept d'*Addition* correspond dans l'implémentation du modèle à la méthode `addition` ayant trois paramètres (ceci est exprimé grâce à `addition/3`). Les trois attributs de ce concept sont spécifiés dans la liste par `elemX`, `elemY` et `result` auxquels on associe des accesseurs aux valeurs via les trois paramètres de la méthode `addition`. Par exemple, la valeur de l'attribut `elemX` est récupérée par l'intermédiaire du premier paramètre de la méthode `addition`. Il en est de même pour le concept *EnvoiMessage* dont le lien ne permet ici que de récupérer le contenu du message envoyé, via un accès au second paramètre de la méthode `envoiMessage` ayant deux paramètres.

6.2.2 La description des lois

Une fois la description de l'application effectuée, c'est à dire, une fois l'ontologie représentative des spécificités des agents du système constituée et une fois les liens entre ces concepts et l'implémentation du modèle mis en place, il est possible de décrire les lois associées au système.

Cet ensemble de lois est fourni par le concepteur de l'application, dans un premier temps, en langage naturel. Ces lois, comme nous l'avons vu au chapitre précédent, peuvent être vues comme les spécifications de l'application, les exigences que doivent suivre les développeurs

¹dans ce cas on doit malgré tout préciser `nohook`

du système. Le concepteur de l'application est donc en mesure de fournir ces lois. Une fois l'ensemble des lois fourni, un expert va traduire chaque loi dans un langage de description que nous fournissons et qui va permettre de déduire le contrôle à appliquer aux agents à partir des lois.

D'une manière générale, une loi va porter sur un ou plusieurs agents et utiliser les concepts fournis pour décrire l'application, pour exprimer des événements se déroulant au niveau des agents. Ces événements pouvant être des changements internes de l'état des agents ou l'exécution d'actions par les agents. Il est donc indispensable que la personne ayant en charge la description des lois, à l'aide du langage, ait connaissance, en plus de l'ontologie, de l'agentification du système. Nous partons donc du principe que si le concepteur de l'application en lui-même n'a pas les connaissances nécessaires pour agentifier les lois qu'ils proposent de mettre en place au sein de l'application, il existe un expert ayant toutes les capacités pour effectuer la description finale des lois.

Nous avons ici conscience du fait que ce point peut poser problème car il n'est pas évident de pouvoir aisément effectuer cette étape de traduction. A l'usage, il nous est apparu plus simple de directement penser les lois au niveau des agents et non au niveau du système. Néanmoins, il n'est pas garanti que, lors de la conception d'un système d'envergure, il soit possible de fournir directement les lois en terme d'agents. C'est pour cette raison que nous avons fait la supposition de la présence d'un expert pour exprimer correctement les lois au niveau agent. Malgré tout il nous semblerait judicieux d'approfondir ce point sensible au niveau de la méthodologie de mise en place du contrôle.

6.2.3 Le langage de lois

Le langage de description des lois que nous avons mis en place est basé sur la logique déontique dynamique [Mey88]. Nous l'avons conçu principalement pour simplifier l'écriture des lois par rapport à l'écriture directe sous forme de formule logique, mais aussi dans le but de restreindre les possibilités d'expression des lois. En effet, si la logique déontique dynamique semble la plus appropriée pour exprimer les lois que nous souhaitons pouvoir prendre en compte, elle fournit des possibilités d'expression qui ne nous intéressent pas de prendre en considération dans le cadre de notre approche de contrôle.

La logique déontique dynamique

Ainsi la sémantique de notre langage de lois peut être vu comme un sous-ensemble de la logique déontique dynamique. Nous allons dans cette partie, décrire les caractéristiques de cette logique modale basée à la fois sur la logique déontique et sur la logique dynamique.

Nous étions, au départ, partis de la logique déontique standard, (la SDL (Standard Deontic Logic) [vW51]) pour décrire les lois. La logique déontique est la science des principes de l'analyse purement formelle du "devoir-être" [Bai91]. C'est une logique modale dont la première version est apparue dans les années 50 et qui est encore de nos jours sujettes à études et extension. Elle est construite sur la logique des propositions et admet donc tous les opérateurs du calcul propositionnel. Elle fournit également un ensemble d'opérateurs modaux permettant d'exprimer des *obligations*, des *interdictions*, des *permissions* et du *facultatif*. Choisissons comme terme premier l'obligation, on peut alors écrire :

$$Op = \textit{il est obligatoire que } p$$

où O est un opérateur propositionnel à argument propositionnel. On peut définir tous les opérateurs à partir de cette obligation :

$$Pp = \neg O\neg p$$

il est permis que } p = \textit{il n'est pas obligatoire que non } p.

$$Ip = O\neg p$$

il est interdit que } p = \textit{il est obligatoire que non } p.

$$Fp = \neg Op$$

il est facultatif que } p = \textit{il n'est pas obligatoire que } p.

La logique déontique standard fournit un ensemble d'axiomes minimal D :

$$\text{Def.P } Pp = \neg O\neg p$$

$$\text{AO.D1 } O(p \supset q) \supset (Op \supset Oq)$$

$$\text{AO.D2 } Op \supset \neg O\neg p$$

$$\text{RO } \vdash \alpha \rightarrow \vdash O\alpha$$

A cela s'ajoute un ensemble de résultats déductibles de cette axiomatique dont :

$$* O(p \wedge q) \equiv (Op \wedge Oq)$$

$$* P(p \vee q) \equiv (Pp \vee Pq)$$

$$* P(p \wedge q) \supset (Pp \wedge Pq)$$

$$* \vdash \alpha \supset \beta \rightarrow \vdash O\alpha \supset O\beta$$

- * $\vdash \alpha \equiv \beta \rightarrow \vdash O\alpha \equiv O\beta$
- * $\vdash \neg(Op \wedge O\neg p)$

Le dernier résultat est intéressant en ceci qu'il exprime l'absence de conflit dans ce système. En effet, il exprime qu'il est faux qu'une propriété soit obligatoire et interdite à la fois.

La sémantique de la logique déontique est basée sur la sémantique des mondes possibles de Kripke. Elle comprend un ensemble de mondes, une relation binaire définie dans cet ensemble et une fonction d'évaluation permettant de déterminer les valeurs de vérité des formules dans les divers mondes. On définit un monde permmissible pour le monde réel comme une variante possible de ce monde, dans laquelle toutes les obligations sont remplies. On dira alors qu'une expression est obligatoire dans le monde réel si elle est vraie dans tous les mondes permmissibles pour ce monde et donc elle ne sera pas obligatoire s'il existe au moins un monde permmissible dans lequel elle est fausse.

Baser notre langage sur la logique déontique nous permettait principalement de spécifier des propriétés désirables sachant qu'elles peuvent être violées. D'autres logiques, comme la logique temporelle ou la logique classique, servent plutôt à définir des propriétés que doit respecter le système sans considération de violation. En logique déontique, il n'existe pas d'axiome du type $O(\alpha) \supset \alpha$, comme on peut le trouver en logique des possibilités avec la nécessité. En effet, le monde réel n'est pas considéré comme un monde permmissible pour la logique déontique, car dans ce cas, aucune loi ne serait jamais violée dans le monde réel, ce qui est impensable. La logique déontique considère donc que dans un monde idéal, il est possible d'envisager que toutes les obligations de ce monde sont respectées et que tout ce qui est obligatoire est effectivement vraie. Mais dans le monde réel on sait seulement dire que si une expression est obligatoire alors elle est permise mais pas qu'elle est forcément vraie. Cette logique sous-entend donc qu'il est envisageable que le réel diffère de l'idéal et c'est exactement ce que l'on souhaite exprimer avec les lois de notre approche : dans un monde idéal toutes les obligations (les exigences) du système seraient respectées mais dans le monde réel il faut envisager la possibilité que le système ne respecte pas ces obligations. La logique déontique permet de concevoir et de se rendre compte des violations de l'idéal.

Enfin, la logique déontique est souvent utilisée pour définir des contraintes d'intégrité dans des bases de données (des formules à satisfaire par les données contenues dans les bases) ou pour définir des politiques de sécurité. Elle est utilisée également dans la tolérance aux fautes, en particulier pour spécifier les comportements à suivre lors de la déviation du comportement d'un système (les obligations sont alors secondaires ou *contrary-to-duty*). On retrouve également la logique déontique dans l'expression des normes dans une organisation.

Mais un problème persiste dès lors que nous voulons exprimer des événements dynamiques, tels que l'enchaînement d'actions effectuées par un ou plusieurs agents, ou tout simplement le changement d'état au sein d'un agent. Il était possible de les exprimer dans cette logique mais le résultat ne nous semblait pas assez convainquant, et surtout il ne permettait pas de réellement laisser transparaître cette notion de dynamicité. C'est pour cette raison que nous nous sommes penchés vers une variante proposée par Meyer de la logique déontique standard : la logique déontique dynamique [Mey88]. Tout en gardant les principes évoqués précédemment sur les avantages d'utiliser la logique déontique pour exprimer les lois appliquées à un système, s'ajoute ceux liés à l'expression de la dynamicité proposée par la logique déontique dynamique. Si cette dernière part des principes de la logique déontique standard que nous avons présentée ci-dessus, tels que l'obligation, la permission et l'interdiction, elle s'en éloigne en essayant d'exprimer les concepts déontiques par l'utilisation de la logique dynamique.

L'apport principal de la logique déontique dynamique par rapport à la logique déontique standard est la prise en compte de l'opérateur modal $[.]$ associé à une action α . L'expression $[\alpha]\phi$ voudrait alors dire que la précondition est requise pour garantir que ϕ sera vraie après que α soit exécutée. Donc, $[\alpha]\phi$ veut dire simplement que si l'action α est exécutée, ϕ sera vraie par la suite. Concrètement $[\alpha]\phi$ est une version plus fine du classique $\alpha \supset \phi$ que l'on utilise dans la logique déontique standard incluant une notion temporelle dans l'exécution des actions et l'apparition d'une assertion.

Formellement, soit A un alphabet fini exprimant des actions élémentaires et soit Act et Ass , respectivement l'ensemble des expressions sur les actions et l'ensemble des assertions satisfaisant les propriétés suivantes :

- (1) $a \in Act \forall a \in A$.
- (2) $\emptyset \in Act$ et $U \in Act$ (respectivement *failure* et *whatever*).
- $\forall \alpha_1, \alpha_2 \in Act, \phi \in Ass$
- (3) La composition séquentielle, notée $\alpha_1; \alpha_2 \in Act$ (“ α_1 est suivi par α_2 ”).
- (4) Le choix, noté $\alpha_1 \cup \alpha_2 \in Act$ (prononcé “ α_1 ou α_2 ”).
- (5) L'action jointe (ou simultanée), notée $\alpha_1 \& \alpha_2 \in Act$ (“ α_1 en même temps que α_2 ”).
- (6) L'action conditionnelle, notée $\phi \rightarrow \alpha_1 / \alpha_2 \in Act$ (“Si ϕ alors α_1 sinon α_2 ”).
- (7) La négation de l'action, noté $\bar{\alpha}_1 \in Act$ (“non- α_1 ”).

S'ajoute à cela le système PD_eL défini par :

Un ensemble d'axiomes :

- (PC) Toutes les tautologies du calcul propositionnel.
- ($\Box \supset$) $\vdash [\alpha](\phi_1 \supset \phi_2) \supset ([\alpha]\phi_1 \supset [\alpha]\phi_2)$
- ($;$) $\vdash [\alpha_1; \alpha_2]\phi \equiv [\alpha_1]([\alpha_2]\phi)$
- (\cup) $\vdash [\alpha_1 \cup \alpha_2]\phi \equiv [\alpha_1]\phi \wedge [\alpha_2]\phi$
- ($\&$) $\vdash [\alpha_1 \& \alpha_2]\phi \subset [\alpha_1]\phi \vee [\alpha_2]\phi$
- (\rightarrow) $\vdash [\phi_1 \rightarrow \alpha_1/\alpha_2]\phi_2 \equiv (\phi_1 \supset [\alpha_1]\phi_2) \wedge (\neg\phi_1 \supset [\alpha_2]\phi_2)$
- ($\langle \rangle$) $\vdash \langle \alpha \rangle \phi \equiv \neg[\alpha]\neg\phi$
- ($\bar{;}$) $\vdash \overline{[\alpha_1; \alpha_2]}\phi \equiv \overline{[\alpha_1]}\phi \wedge [\alpha_1][\neg\alpha_2]\phi$
- ($\bar{\cup}$) $\vdash \overline{[\alpha_1 \cup \alpha_2]}\phi \subset \overline{[\alpha_1]}\phi \vee \overline{[\alpha_2]}\phi$
- ($\bar{\&}$) $\vdash \overline{[\alpha_1 \& \alpha_2]}\phi \equiv \overline{[\alpha_1]}\phi \wedge \overline{[\alpha_2]}\phi$
- ($\bar{\Rightarrow}$) $\vdash \overline{[\phi_1 \rightarrow \alpha_1/\alpha_2]}\phi_2 \equiv (\phi_1 \supset \overline{[\alpha_1]}\phi_2) \wedge (\neg\phi_1 \supset \overline{[\alpha_2]}\phi_2)$
- ($\bar{}$) $\vdash \overline{[\alpha]}\phi \equiv [\alpha]\phi$
- (\emptyset) $\vdash [\emptyset]\phi$

Un ensemble de règles :

$$(MP) \frac{\vdash \phi, \vdash \phi \supset \psi}{\vdash \psi}$$

$$(N) \frac{\vdash \phi}{\vdash [\alpha]\phi}$$

Enfin, un ensemble de théorème dérivés de PD_eL :

- (1) $\vdash [\alpha](\phi_1 \wedge \phi_2) \equiv [\alpha]\phi_1 \wedge [\alpha]\phi_2$
- (2) $\vdash [\alpha](\phi_1 \vee \phi_2) \subset [\alpha]\phi_1 \vee [\alpha]\phi_2$
- (3) $\vdash F(\alpha_1; \alpha_2) \equiv [\alpha_1]F\alpha_2$
- (4) $\vdash F(\alpha_1 \cup \alpha_2) \equiv F\alpha_1 \wedge F\alpha_2$
- (5) $\vdash F(\alpha_1 \& \alpha_2) \subset F\alpha_1 \vee F\alpha_2$
- (6) $\vdash F\alpha_1 \supset F(\alpha_1 \& \alpha_2)$
- (7) $\vdash O(\alpha_1; \alpha_2) \equiv O\alpha_1 \wedge [\alpha_1]O\alpha_2$
- (8) $\vdash O(\alpha_1 \cup \alpha_2) \subset O\alpha_1 \vee O\alpha_2$
- (9) $\vdash O(\alpha_1 \& \alpha_2) \equiv O\alpha_1 \wedge O\alpha_2$
- (10) $\vdash P(\alpha_1; \alpha_2) \equiv \langle \alpha_1 \rangle P\alpha_2$
- (11) $\vdash P(\alpha_1 \cup \alpha_2) \equiv P\alpha_1 \vee P\alpha_2$
- (12) $\vdash P(\alpha_1 \& \alpha_2) \supset P\alpha_1 \wedge P\alpha_2$

Nous présentons également un opérateur que nous allons utiliser par la suite, l'opérateur $done(\alpha)$ signifiant qu'une action α a été exécutée et ayant les propriétés suivantes :

- (1) $[\alpha]\phi \vdash (done(\alpha) \supset \phi)$
- (2) $\vdash [\alpha]done(\alpha)$
- (3) $\vdash done(\alpha_1 \cup \alpha_2) \equiv done(\alpha_1) \vee done(\alpha_2)$
- (4) $\vdash done(\alpha_1 \& \alpha_2) \equiv done(\alpha_1) \wedge done(\alpha_2)$
- (5) $\vdash done(\bar{\alpha}) \equiv \neg done(\alpha)$

C'est donc à partir de cette logique, permettant à la fois d'exprimer des concepts déontiques et des notions temporelles que nous allons écrire notre langage de lois.

La syntaxe du langage

Notre langage peut donc être vu comme un sous-ensemble de la logique déontique dynamique permettant essentiellement d'exprimer des interdictions ou des obligations ainsi que la notion d'enchaînement d'événements et des conjonctions. Il est de ce fait également possible de mettre en place une notion de délai. Nous verrons dans la suite de ce chapitre les correspondances entre les mots clés de notre langage et la logique déontique dynamique.

En revanche, nous ne laissons pas la possibilité d'exprimer, en particulier, des permissions, des disjonctions d'événements, car ces concepts ne nous semblent par nécessaires, à ce jour, à l'expression du contrôle de comportement. La syntaxe de notre langage est donc la suivante :

<i>LAW</i>	$:= (CA) (DA \langle APC \rangle).$
<i>CA</i>	$:= agent : AGENT \langle \text{and } PROP \rangle$
<i>DA</i>	$:= DEON_OP \ EXP$
<i>APC</i>	$:= TEMP_OP \ EXP \mid TEMP_OP \ EXP \ APC$
<i>DEON_OP</i>	$:= FORBIDDEN \mid OBLIGED$
<i>TEMP_OP</i>	$:= AFTER \mid BEFORE \mid IF$
<i>EXP</i>	$:= TERM \mid TERM \ \text{AND} \ EXP \mid TERM \ \text{THEN} \ EXP$
<i>TERM</i>	$:= EVENT \mid \text{NOT } EVENT \mid EVENT \ \text{TIME}$
<i>EVENT</i>	$:= agent \ \text{do} \ SMTH \langle \text{and } PROP \rangle \mid$ $agent \ \text{be} \ SMTH \langle \text{and } PROP \rangle$
<i>TIME</i>	$:= -second \mid +second$

où *CA* correspond aux agents concernés par cette loi. *DA* correspond aux expressions

déontiques mises en oeuvre dans la loi. Enfin APC correspond à l'ensemble des conditions temporelles appliquées à la loi (cf. chapitre précédent).

L'ensemble des agents concernés par la loi sont définis de la façon suivante :

agent : AGENT <and PROP>

où, **agent** représente l'identifiant associé à l'agent dans le reste de la loi et où **AGENT** représente la concept d'agent de l'ontologie. Enfin **PROP** définit un ensemble de propriétés facultatives sur les agents, telles que des propriétés sur les attributs du concept d'agent ou de ses sous-concepts.

Les expressions déontiques sont composées d'un opérateur déontique, tel que **FORBIDDEN**, pour exprimer une interdiction ou, **OBLIGED**, pour exprimer une obligation, et d'une expression composée d'un événement, d'une conjonction d'événements ou d'une suite d'événements.

Les expressions temporelles quant-à-elles sont composées d'un opérateur temporel et d'un ensemble d'événements. Les opérateurs temporels sont tels que **AFTER** exprime qu'un événement a lieu après un autre, **BEFORE** exprime qu'un événement a lieu avant un autre et **IF** exprime que sous certaines conditions un événement a lieu. Il est possible d'exprimer également des négations sur les événements ou de leur associer un délai spécifié en seconde, dont la signification varie suivant si nous sommes dans le cas d'une interdiction ou d'une obligation. Les événements sont de deux formes :

agent do SMTH <and PROP> ou **agent be SMTH** <and PROP>

où **agent** est l'identifiant d'un agent défini en CA, où **do** exprime l'exécution d'une action ou le changement d'état d'une caractéristique de l'agent et, où **be** exprime l'apparition d'un état au sein de l'agent (à la suite de l'exécution d'une action ou du changement de valeur d'une caractéristique de l'agent). Ainsi, **SMTH** représente un concept, tel que **ACTION** ou **CARACTÉRISTIQUE** et leurs sous-concepts, auquel on peut associer les propriétés facultatives **PROP** sur les attributs des concepts.

Un exemple :

Il est interdit pour un agent A1 d'exécuter une action ACT2 dans les Sec secondes après avoir exécuté une action ACT1

qui s'écrira dans notre langage de la façon suivante :

(A1 : agent)

FORBIDDEN (A1 **do** ACT2) **AFTER** (A1 **do** ACT1 - Sec).

Un autre exemple :

Il est obligatoire pour un agent A1 d'exécuter une action ACT3 après avoir exécuté une action ACT1 puis une action ACT2 et avant d'exécuter une action ACT4

qui s'écrira dans notre langage de la façon suivante :

(A1 : agent)

OBLIGED (A1 do ACT3)

AFTER (A1 do ACT1) **THEN** (A1 do ACT2)

BEFORE (A1 do ACT4).

6.2.4 Les stratégies de régulation

Pour permettre aux agents de réagir suite à la détection de la transgression d'une loi, nous avons vu que la partie contrôle envoyait à la partie comportement des informations de transgression. Ces informations contiennent l'identifiant de la loi qui vient d'être violée ainsi que l'événement reçu ayant entraîné la violation de la loi par l'agent.

A partir de ces informations, l'agent a pour objectif de réguler son comportement pour se soustraire au comportement indésirable qu'il est potentiellement entrain de suivre. Pour ce faire, le ou les développeurs des agents doivent fournir une bibliothèque de stratégies de régulation permettant de résoudre les problèmes de transgressions de lois. Ces stratégies peuvent être de toute sorte et dépendent de l'agent et de son contexte. Elles peuvent s'avérer très simples ou extrêmement complexes, nécessitant la prise en compte du comportement et de l'état des autres agents. Une stratégie peut également servir pour plusieurs lois, tel un comportement par défaut à suivre lors des violations, ou elle peut être spécifique à une loi suivant la gravité de la violation.

Au départ, notre but était de fournir un moyen de décrire ces stratégies à haut niveau, en quelque sorte sur le même principe que la description des lois, en utilisant dans les stratégies uniquement des comportements et des états spécifiés dans l'ontologie décrivant l'application. L'idée était alléchante car elle aurait permis de bien distinguer les stratégies et le comportement de base de l'agent, tout en réduisant le travail des développeurs. Nous aurions pu de ce fait automatiser la mise en place des stratégies au sein des agents comme pour les lois. Malheureusement, il faut avouer que cela s'est avéré impossible, en particulier car les stratégies de régulation sont à la fois dépendantes du contexte dans lequel se trouve l'agent, mais aussi nécessitent généralement des actions de plus bas niveau que celles exprimées dans les lois.

Bien entendu, nous sommes conscient que cet obstacle au niveau de la régulation réduit quelque peu l'intérêt d'exprimer des lois indépendamment des agents. En effet, les développeurs vont devoir implanter des stratégies au sein des agents, en rapport avec ces lois, que nous aurions aimé qu'ils ne connaissent pas, tout comme les agents. Néanmoins, à ce jour, nous n'avons pas trouvé de solution automatisée satisfaisante.

6.3 Un peu de travail automatique

A ce stade, le système multiagent a été décrit à l'aide de concepts, et des liens ont été fournis entre ces concepts et l'implémentation du modèle des agents. Des lois ont été exprimées par l'intermédiaire du langage de description que nous avons mis en place et en utilisant les concepts et instances représentant l'application. Enfin, des stratégies ont été écrites pour réguler le comportement des agents lors de la violation d'une ou plusieurs lois.

A présent, les mécanismes de surveillance du comportement des agents peuvent être mis en place et ce de façon automatique. A partir des lois, il est en effet possible de déduire le contrôle à effectuer et dans quels agents il doit se situer. Grâce à un ensemble de règles de déduction, nous pouvons générer des agents autocontrôlés.

De plus, pour permettre aux agents de surveiller leur propre comportement, nous leur fournissons une méta-architecture que nous avons présentée au chapitre précédent, basée sur le principe de l'observateur et utilisant des réseaux de Petri pour représenter les lois. Ainsi, générer automatiquement des agents capables de surveiller leur propre comportement va revenir à, d'une part, modifier le code des agents pour leur permettre d'envoyer de façon totalement transparente les informations utiles à la partie contrôle, et d'autre part, à déduire des lois, les réseaux de Petri qui seront utilisés par la partie contrôle pour détecter les transgressions.

6.3.1 Instrumentation du code des agents

Pour permettre aux agents d'envoyer les informations nécessaires sur leur comportement, pour pouvoir en effectuer la surveillance, nous proposons d'effectuer une instrumentation automatique du programme de comportement des agents. Cette instrumentation va permettre de détecter l'apparition des événements utilisés dans les lois, à l'aide de ce que nous avons appelé au chapitre précédent des **points de contrôle**. Ces points de contrôle sont insérés dans le code de l'agent grâce au liens définis par le concepteur du modèle entre les concepts et l'implémentation. Pour effectuer cette instrumentation nous nous inspirons du principe du **tissage** qui est une partie importante de la programmation par aspects [Wam03].

La programmation par aspects utilise le tissage pour injecter du code, défini dans des “aspects”, dans les classes d’une application lors de la compilation. Les aspects sont des modules représentant des concepts pouvant apparaître dans différentes parties d’une application que l’on regroupe au niveau d’une même classe. Un aspect définit un code à insérer, par exemple, avant ou après une méthode d’un programme. Lors de la compilation, ce code est automatiquement injecté juste avant ou après le code de la méthode. Ainsi lors de son exécution, le code défini par l’aspect sera exécuté avant ou après le code de la méthode en elle-même. La programmation par aspect a déjà été utilisée dans une autre optique, à l’aide de AspectJ [CCHW], pour surveiller le comportement d’une application [MSSP02].

Nous proposons donc d’utiliser ce principe d’injection de code pour mettre en place les points de contrôle au sein des agents. Pour ce faire nous procédons de la façon suivante :

- * Partant des lois, nous allons extraire les actions ou les états qui doivent être détectés.
- * Pour chaque action ou état, nous recherchons alors les liens qui ont été fournis par le concepteur.
- * Nous injectons, avant ou après chaque lien, le code permettant l’envoi des informations à la partie contrôle et le code permettant de récupérer de potentielles informations de transgression. Ce dernier point permettant alors à l’agent de lancer une stratégie de régulation.

Les points de contrôle sont donc mis en place au sein du code de comportement de l’agent, avant ou après le code défini dans les liens et correspondent à du code générique permettant d’envoyer à la partie contrôle une information sur l’occurrence de l’action ou l’apparition de l’état défini dans les lois. Il existe, en fait, deux sortes de points de contrôle, le premier type de point permet uniquement l’envoi d’information vers la partie contrôle sur le comportement de l’agent, le second type, en plus d’envoyer les informations va aussi se mettre en attente d’une confirmation de la part de la partie de contrôle pour savoir si une loi utilisant l’événement dont une occurrence a été détectée dans le comportement de l’agent, a été transgressée. Ce dernier type de point de contrôle est donc bloquant, il permet en particulier de ne pas exécuter un certain comportement lorsque le code de l’agent arrive sur un point de contrôle correspondant, dans une loi, à une action ou un état qui est interdit. Ce point de contrôle va alors lancer une stratégie de régulation permettant à l’agent de se soustraire au mauvais comportement.

Nous avons vu au chapitre précédent l’importance d’empêcher l’exécution de comportement interdit au sein des agents. Néanmoins, il est tout à fait envisageable de laisser l’agent exécuter le code interdit dans sa stratégie de régulation après détection de la tentative d’exécution d’un comportement interdit. Nous partons alors du principe que si ,dans

un premier temps, il est indispensable d'empêcher l'agent d'exécuter ce comportement et d'être prévenu de cette tentative, il est normal de pouvoir le laisser exécuter ce comportement une fois qu'il a eu conscience de cette violation et qu'il a, suivant des critères qui lui sont propres et suivant une analyse du contexte que lui seul peut entreprendre, pris la décision de suivre ce comportement. Ainsi, le code défini dans les stratégies de régulation n'est pas obligatoirement sujet à la surveillance à laquelle est soumis le comportement réel de l'agent.

6.3.2 Les règles de génération des réseaux de Petri

En parallèle de la génération des points de contrôle à insérer au sein du code de comportement des agents, nous effectuons la génération des réseaux de Petri représentant les lois. Suivant le principe de l'observateur, chaque transition des réseaux correspond à une action ou un état dont nous souhaitons détecter l'apparition au sein des agents. La génération des réseaux de Petri se fait pour chaque loi en suivant les trois étapes suivantes :

1. La traduction de la loi en une expression logique L , dans le but de mettre en évidence un ensemble d'expressions logiques atomiques $\{e_1, \dots, e_n\}$.
2. Puis de façon incrémentale :
 - 2.a La déduction de l'ensemble des réseaux de Petri $\{P_1, \dots, P_n\}$, représentant chaque expression atomique de $\{e_1, \dots, e_n\}$.
 - 2.b La fusion de tous les réseaux de $\{P_1, \dots, P_n\}$ à l'aide des relations entre les expressions e_1 à e_n pour obtenir le réseau de Petri final P , représentant la loi.

Pour mettre en place la première étape, chaque opérateur de notre langage de description a une correspondance en logique déontique dynamique. Nous utilisons donc un ensemble de règles de traduction pour obtenir l'expression logique représentant la loi et mettre en évidence l'ensemble des expressions atomiques :

Ensemble de règles 1

Soit Act un ensemble d'actions.

Soit Assert un ensemble d'assertions.

Soit State un ensemble d'états incluant Assert.

$\forall \alpha \in Act, \phi \in Assert, \beta \in State.$

(1) $FORBIDDEN \alpha \equiv F\alpha$

(2) $OBLIGED \alpha \equiv O\alpha$

(3) $\phi AFTER \alpha \equiv [\alpha]\phi$

(4) $FORBIDDEN_{\alpha_1} BEFORE \alpha_2 \equiv done(\alpha_2) \vee F\alpha_1$

(5) $OBLIGED_{\alpha_1} BEFORE \alpha_2 \equiv \neg done(\alpha_2) \vee O\alpha_1$

- (6) $\alpha_1 \text{ AND } \alpha_2 \equiv (\alpha_1; \alpha_2) \cup (\alpha_2; \alpha_1)$
- (7) $\alpha_1 \text{ THEN } \alpha_2 \equiv \alpha_1; \alpha_2$
- (8) $\phi \text{ IF } \beta \equiv \beta \supset \phi$
- (9) $\text{FORBIDDEN} \alpha - \text{Sec} \equiv \text{done}(\text{time}(\text{Sec})) \vee F\alpha$
- (10) $\text{OBLIGED} \alpha - \text{Sec} \equiv \neg \text{done}(\text{time}(\text{Sec})) \vee O\alpha$
- (11) $\phi + \text{Sec} \equiv [\text{time}(\text{Sec})]\phi$
- (12) $\text{FORBIDDEN} \beta \equiv \neg\beta$
- (13) $\text{OBLIGED} \beta \equiv \beta$

Les traductions (1), (2) et (3) sont directement la traduction en logique déontique dynamique dans laquelle, on exprime, nous l'avons plus haut dans ce chapitre, l'obligation par l'opérateur O , l'interdiction par l'opérateur F et l'enchaînement d'une action et d'un opérateur déontique quelqu'il soit par l'opérateur $[\cdot]$. Pour les règles (4) et (5) nous avons utilisé l'opérateur $\text{done}(\alpha)$ qui permet d'exprimer le fait que l'action α a été exécutée effectivement. Ainsi, nous avons cherché une expression logique permettant d'exprimer dans le premier cas qu'il est faux que "si α_2 n'a pas été exécutée alors l'interdiction d'avoir α_1 n'est pas vérifiée", c'est-à-dire $\neg \text{done}(\alpha_2) \supset F\alpha_1$, d'où la règle (4). Et dans le second cas qu'il est faux que "si α_2 a été exécutée alors l'obligation d'avoir α_1 n'est pas vérifiée", c'est-à-dire $\text{done}(\alpha_2) \supset O\alpha_1$, d'où la règle (5).

Pour la règle (6), nous souhaitons exprimer que les actions α_1 et α_2 sont exécutées "en même temps". Ce qui s'exprime plus naturellement dans notre contexte par le fait que, "soit α_1 s'est exécutée avant α_2 , soit α_2 s'est exécutée avant α_1 ". Pour ce faire, nous avons utilisé l'opérateur $;$ permettant d'exprimer une séquence entre deux actions, ainsi que l'opérateur \cup pour exprimer une union sur des actions. Pour la règle (7), nous avons directement utilisé l'opérateur de séquence entre deux actions.

En ce qui concerne la règle (8), nous avons simplement utilisé l'opérateur \supset qui dans la logique déontique dynamique permet d'exprimer que "si on est dans un certain état alors une assertion doit être vérifiée". Enfin pour les règles (9) et (10), où il est question d'un délai associé à une assertion, nous avons à nouveau utilisé l'opérateur $\text{done}(\cdot)$ auquel nous avons adjoint le prédicat $\text{time}(\text{sec})$ pour exprimer que le temps sec est effectivement écoulé. Nous nous retrouvons alors dans le même schéma que si une action a été effectivement exécutée. La règle (11) quant-à-elle est basée sur le même principe que la règle (3) où une assertion doit être vérifiée après que le temps soit écoulé.

Pour les règles (12) et (13), nous souhaitons exprimer des interdictions et des obligations sur un état particulier. Nous exprimons donc une interdiction sur un état sous la forme de la négation de cet état et l'obligation sur un état par cet état.

A présent nous allons décrire les règles de déduction des réseaux de Petri à partir des expressions logiques obtenues à l'aide du premier ensemble de règles de traduction. L'ensemble des règles de déduction est le suivant :

Ensemble de règles 2

Soit $PN = \langle P, T, Pre, Post, I, IS \rangle$ un réseau de Petri

(Par mesure de clarté nous ne représenterons dans la suite que les Pre et $Post$ valant 1)

Soit t_α la transition associée à l'événement α .

Dans la suite, la règle $X \Rightarrow Y$ signifie que X est une assertion logique et Y le réseau de Petri correspondant.

$\forall \alpha \in Act, \beta \in State$

- (1) $F\alpha \Rightarrow \langle \{p_i, p_j\}, t_\alpha, \emptyset, Post(p_j, t_\alpha), Pre^*(p_i, t_\alpha), IS(t_\alpha) = [0, \infty[\rangle$
- (2) $O\alpha \Rightarrow \langle \{p_i, p_j\}, t_\alpha, Pre(p_i, t_\alpha), Post(p_j, t_\alpha), \emptyset, IS(t_\alpha) = [0, \infty[\rangle$
- (3) $done\alpha \Rightarrow \langle \{p_i, p_j\}, t_\alpha, Pre(p_i, t_\alpha), Post(p_j, t_\alpha), \emptyset, IS(t_\alpha) = [0, \infty[\rangle$
- (4) $\neg done(\alpha) \Rightarrow \langle \{p_i, p_j\}, t_\alpha, \emptyset, Post(p_j, t_\alpha), Pre^*(p_i, t_\alpha), IS(t_\alpha) = [0, \infty[\rangle$
- (5) $\beta \Rightarrow \langle \{p_i, \beta, p_j\}, \{t_i, t_j\}, \{Pre(p_i, t_i), Pre(\beta, t_j)\}, \{Post(\beta, t_i), Post(p_j, t_j)\}, \emptyset, \{IS(t_i) = [0, \infty[, IS(t_j) = [0, \infty[\} \rangle$
- (6) $\neg\beta \Rightarrow \langle \{p_i, \beta, p_j\}, \{t_i, t_j\}, Pre(\beta, t_j), \{Post(\beta, t_i), Post(p_j, t_j)\}, Pre^*(p_i, t_i), \{IS(t_i) = [0, \infty[, IS(t_j) = [0, \infty[\} \rangle$
- (7) $\alpha \Rightarrow \langle \{p_i, p_j\}, t_\alpha, Pre(p_i, t_\alpha), Post(p_j, t_\alpha), \emptyset, IS(t_\alpha) = [0, \infty[\rangle$
- (8) $time(sec) \Rightarrow \langle \{p_i, p_j\}, t_{time}, Pre(p_i, t_{time}), Post(p_j, t_{time}), \emptyset, IS(t_{time}) = [sec, sec] \rangle$

Pour chaque expression logique atomique nous associons un réseau de Petri ordinaire. Comme nous l'avons présenté au chapitre 5 nous obtenons trois types de réseaux de Petri pour représenter les comportements exprimés dans les lois.

Les règles (1) et (4) traduisent l'interdiction de voir apparaître une action et la non-exécution d'une action, par un réseau de Petri composé de deux places et d'une transition associée à un arc inhibiteur, comme décrit sur la figure 6.1.

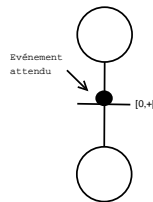


FIG. 6.1 – Réseau obtenu pour les règles (1) et (4)

Les règles (2), (3) et (7) traduisent l'obligation de voir apparaître une action, l'exécution effective d'une action et l'apparition d'une action, par un réseau de Petri composé de deux places et d'une transition associée à un arc simple, comme décrit sur la figure 6.2.

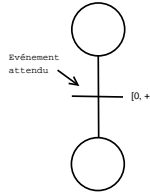


FIG. 6.2 – Réseau obtenu pour les règles (2), (3) et (7)

Enfin, la règle (8) permet de traduire une notion de temps par un réseau t-temporel comme décrit sur la figure 6.3.

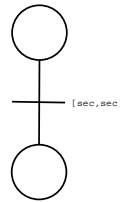


FIG. 6.3 – Réseau obtenu pour la règle (8)

De plus, dans le cas particulier des règles (5) et (6), le fait d'être dans un certain état ou le fait de ne pas être dans un certain état est représenté par un réseau composé de trois places et de deux transitions. Dans le premier cas ce sont des arcs simples, dans le second cas une transition est liée à un arc inhibiteur, comme décrit sur la figure 6.4.

Ces réseaux permettent uniquement de détecter l'apparition de certaines actions et états au niveau de l'agent, ils représentent le comportement de l'agent décrit dans l'expression logique. Il existe deux façons de tirer une transition suivant qu'elle soit reliée par un arc simple (*c.f.* Définition 2 et 3 du chapitre 5) ou un arc inhibiteur (*c.f.* Définition 4 du chapitre 5). Ainsi, dans le cas de comportements que l'on souhaite voir apparaître, la transition est tirée si la place précédente contient un jeton et dans le cas de comportements que l'on ne souhaite pas voir apparaître, la transition est tirée uniquement si la place précédente ne contient pas de jeton, et ce uniquement lorsque l'événement associé à la transition est reçu. Le fait de recevoir une information sur une transition peut également se représenter telle que décrit sur la figure 6.5.

Le traitement des opérateurs déontiques en eux-mêmes est fait par la partie contrôle au

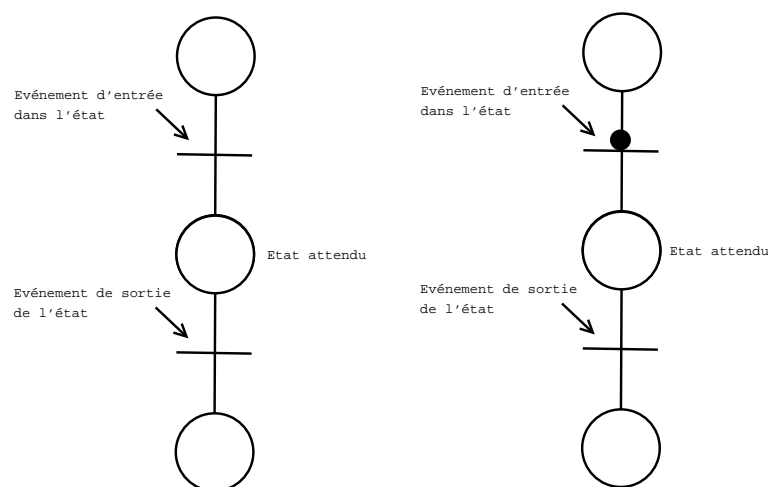


FIG. 6.4 – Les réseaux obtenus respectivement pour les règles (5) et (6)

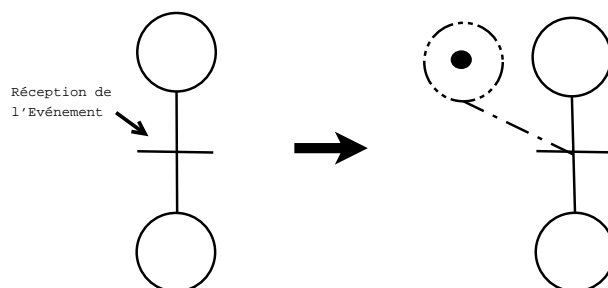


FIG. 6.5 – Représentation de la réception d'un événement au niveau d'une transition

moment du tirage de la transition. Ainsi, à chaque règle de traduction énoncée dans l'ensemble 2 est associé un code de traitement différent. Ce code est exécuté par la partie contrôle et permet de faire évoluer le réseau et de détecter les violations de lois. Le comportement de la partie contrôle vis-à-vis de ces réseaux de Petri sera décrit dans la section 6.3.3.

A ce stade les réseaux ne sont pas marqués. Le réseau de Petri final représentant une loi est ensuite obtenu par fusion des réseaux obtenus au cours de l'étape précédente. Les règles de fusion sont décrites dans l'ensemble suivant :

Ensemble de règles 3

Soit t_α la transition associée à l'événement α .

Soit t_ϕ la transition associée au premier événement exprimé dans ϕ .

Soit ot la fonction qui retourne la place en entrée p de la transition t .

Soit t_o la fonction qui retourne la place en sortie p de la transition t .

Soit $merge_p(p_1, p_2)$ l'opérateur de fusion de deux places p_1 et p_2 .

$\forall \alpha \in Act, \phi \in Assert, \beta \in State$

(1) $[\alpha]\phi \Rightarrow merge_p(t_{\alpha \circ}, ot_{\phi})$.

(2) $\phi_1 \vee \phi_2 \Rightarrow merge_p(ot_{\phi_2}, ot_{\phi_1})$

(3) $\beta \supset \phi \Rightarrow merge_p(ot_{\phi}, \beta)$

(4) $\phi_1 \cup \phi_2 \Rightarrow$ obtention de deux réseaux : PN_{ϕ_1} et PN_{ϕ_2}

(5) $\alpha_1; \alpha_2 \Rightarrow merge_p(t_{\alpha_1 \circ}, ot_{\alpha_2})$

(6) $\beta \vee \phi \Rightarrow merge_p(\beta, ot_{\phi})$

Compte tenu des possibilités d'expression de notre langage, seules ces quelques règles de fusions sont nécessaires pour mettre en place les réseaux de Petri finaux représentant les lois. Elles permettent de fusionner les places précédentes de deux transitions (comme dans la règle (2)), la place précédente avec la place suivante (comme dans les règles (1) et (5)) et enfin, une place représentant un état et la place précédant une transition (comme dans les règles (3) et (6), cela en tenant compte des liaisons possibles entre les expressions logiques atomiques permettant de déduire les bouts de réseaux de Petri. Une fois l'étape de fusion terminée, on obtient un réseau de Petri dont la structure est globalement la même à chaque fois. Nous avons cherché à simplifier les réseaux obtenus de telle sorte que quelque soit le réseau obtenu, le marquage initial sera identique et correspond à un unique jeton pour marqué une unique place de départ. Cette place correspond à la place précédant la transition associé au premier événement attendu dans le comportement de l'agent associé à cette loi.

Ces trois ensembles de règles permettent donc de déduire, à partir des lois, les réseaux de Petri les représentant. Une preuve que toute expression valide dans le langage proposé peut être représentée par un réseau de Petri équivalent sera fournit en annexe. Chaque réseau de Petri final est alors embarqué dans la partie contrôle des agents concernés par les lois associées.

Reprenons dès lors l'exemple vu précédemment :

Exemple

Il est interdit pour un agent A1 d'exécuter une action ACT2 dans les Sec secondes après avoir exécuté une action ACT1.

Nous avons vu que la description de cette loi dans notre langage était :

(A1 : agent)

FORBIDDEN (A1 **do** ACT2) **AFTER** (A1 **do** ACT1 - Sec).

De part l'ensemble de règles 1, en particulier les règles (1), (3) et (9), nous obtenons sa traduction en logique déontique dynamique :

$$[ACT1]F(ACT2) \vee done(time(Sec))$$

puis, de part l'ensemble de règles 2, en particulier les règles (1), (3) et (7), nous obtenons l'ensemble de réseaux Petri représentés sur la figure 6.6

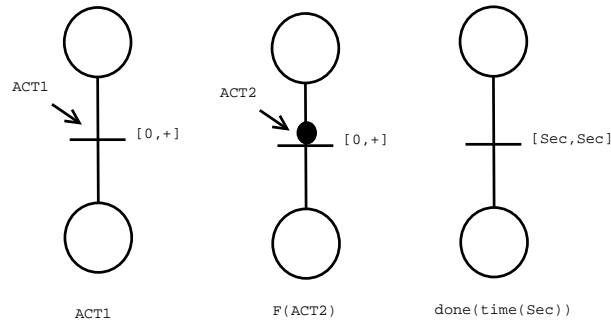


FIG. 6.6 – L'ensemble des réseaux de Petri générés

Enfin, grâce aux règles de fusion (1) et (2), nous pouvons fusionner l'ensemble des réseaux de Petri précédent de telle sorte à obtenir le réseau de Petri final représentant la loi décrit sur la figure 6.7.

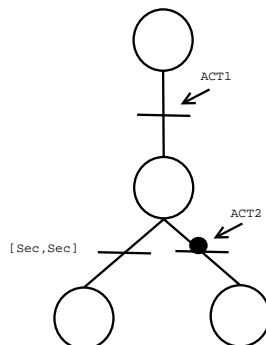


FIG. 6.7 – Le réseau final représentant la loi

6.3.3 Détection des transgressions de lois

Une fois les réseaux de Petri générés et une fois les points de contrôle insérés dans le code de comportement des agents, il ne reste plus qu'à mettre en place la méta-architecture. Cela revient à ajouter la partie contrôle, entité à part entière, permettant de détecter les transgressions de lois à l'aide des réseaux de Petri.

Nous avons vu au chapitre précédent que le comportement global de la méta-architecture était basé sur le principe de l'observateur, la partie contrôle se chargeant de la surveillance du comportement réel de l'agent et de la détection des violations de lois par ce comportement. Nous avons vu que grâce aux points de contrôle, l'agent envoie de façon transparente les informations nécessaires à la partie contrôle pour effectuer l'analyse du bon respect des lois, en vérifiant que le marquage des réseaux de Petri, en attente de ces informations, et le comportement de l'agent coïncident.

Dans la partie contrôle, les réseaux de Petri sont interprétés comme suit :

- * Une **partie conditionnelle** contenant les transitions associées aux actions et états exprimés dans la partie CONDITIONS D'APPLICATION de la loi. Nous appellerons ces transitions, des transitions conditionnelles.
- * Une **partie déontique** contenant les transitions associées aux actions et états exprimées dans la partie ASSERTIONS DÉONTIQUES de la loi. Nous appellerons ces transitions, des transitions déontiques.

Nous dirons alors qu'une transition conditionnelle est *activée* lorsqu'elle est tirable, tandis qu'une transition déontique est *activée* lorsque toutes les transitions conditionnelles la précédant (si elles existent) ont été tirées, c'est-à-dire en d'autres termes que les conditions d'applications de la loi ont été mises en place au sein de l'agent. Dès lors la partie contrôle va suivre l'algorithme 1.

La partie contrôle va, dans un premier temps, activer les réseaux de Petri représentant les lois à surveiller à un instant donné lorsqu'elle reçoit un événement correspondant à la première transition de ce réseau. Puis, quelque soit l'information reçue, la partie contrôle transmet cette information aux instances en attente de celle-ci. Une instance est en attente d'une information lorsque la transition associée à cette information est activée.

Au niveau de chaque instance, si l'information correspond à une transition conditionnelle, la transition est tirée et le marquage est modifié en conséquence. Si l'information correspond à une transition déontique et que cette transition est tirable, la transition est effectivement tirée, si elle ne l'est pas, une exception est lancée. Dans ce dernier cas, la

Algorithm 1 Fonctionnement de la partie contrôle

```

1: Soit  $I$ , une information reçue sur le comportement de l'agent.
2: Soit  $\{t_1, \dots, t_n\}$ , l'ensemble des transitions associées à  $I$ .
3: Soit  $\{P_1, \dots, P_m\}$ , l'ensemble des réseaux de Petri associés à l'agent.
4: Soit  $\{Pact_1, \dots, Pact_p\}$ , l'ensemble des réseaux de Petri activés (i.e. associés aux lois à surveiller à un instant donné).
5: Soit  $t_{ij}$ , la transition  $i$  du réseau  $j$ .
6: for all  $P_k \in \{P_1, \dots, P_m\}$  avec  $t_{1k} \in \{t_1, \dots, t_n\}$  do
7:    $Pact_{p+1} \leftarrow$  créer une instance de  $P_k$  si elle n'existe pas déjà.
8:   ajouter  $Pact_{p+1}$  in  $\{Pact_1, \dots, Pact_p\}$ 
9: end for
10: Soit  $\{Pact_1, \dots, Pact_l\}$ , l'ensemble des réseaux de Petri activés incluant  $t_{ij} \in \{t_1, \dots, t_n\}, j \in \{1, \dots, l\}$ 
11: for all  $Pact_j \in \{Pact_1, \dots, Pact_l\}$  do
12:   informer  $Pact_j$  de l'information associée à  $t_{ij}$ 
13:   retirer  $Pact_j$  de l'ensemble des réseaux activés si  $Pact_j$  est dans un état final.
14: end for

```

partie contrôle reçoit l'exception et génère une information de transgression qui est envoyée à la partie comportement de l'agent dans la perspective de lancer une stratégie de régulation. Enfin, une instance est détruite dès lors que la surveillance du comportement correspondant n'a plus lieu d'être, c'est-à-dire lorsque le contexte d'application n'est plus vérifié dans l'agent. Généralement cela revient au fait que le délai d'observation de la loi est dépassé. Dans tous les autres cas, les instances restent activées tout au long de l'exécution de l'agent.

6.4 Spécificités des lois multiagents

Précédemment nous avons présenté de façon générale les mécanismes de génération des agents autocontrôlés. En particulier, nous avons expliqué les moyens mis en oeuvre pour modifier le programme des agents et ainsi leur permettre de surveiller leur propre comportement et la génération des réseaux de Petri représentant les lois. Ces mécanismes sont valables aussi bien lorsqu'une loi met en jeu un seul agent que plusieurs agents. Néanmoins, dans le second cas, il est nécessaire de mettre en place d'autres mécanismes pour permettent la surveillance et la régulation des comportements de plusieurs agents soumis à une loi multiagent.

Une loi multiagent est une loi qui prend en compte plusieurs agents pour détecter l'appa-

rition de comportements indésirables. C'est à dire que le comportement indésirable n'est pas le résultat d'un seul agent ayant transgressé une loi, mais de plusieurs agents dont les comportements respectifs ont entraîné la transgression d'une loi. Dans ce cas précis, il est nécessaire de trouver un moyen pour détecter les comportements au sein de chaque agent et d'en déduire la transgression de la loi.

En ce qui concerne la génération des réseaux de Petri représentant des lois multiagents et l'instrumentation du code des agents, pour détecter les événements et les états attendus dans les lois, il n'y a pas de modification particulière par rapport au mécanisme de base. La différence tient en la détection de la transgression d'une loi multiagent. Pour cette étape deux possibilités principalement s'offraient à nous : soit nous décidions de centraliser la réception des événements provenant des différents agents dans une unique partie contrôle (d'un agent ou une partie contrôle supplémentaire dédiée aux lois multiagents), soit nous faisons le choix de totalement distribuer la réception des événements et la détection des transgressions de lois.

La première solution est la plus simple à mettre en oeuvre. D'une part, la partie contrôle récupère les événements provenant de tous les agents et peut ainsi facilement détecter les transgressions des lois à l'aide des réseaux de Petri représentant les lois multiagents, d'autre part, les réseaux de Petri lui sont directement fournis. Le problème principal de cette approche est la centralisation en elle-même. Nous avons fait le choix de distribuer le plus possible la surveillance et la régulation des agents en générant des agents capables de contrôler leur propre comportement, il nous semblait alors incohérent de passer pour les lois multiagents, à une version centralisée. De plus, la centralisation est toujours un risque de perte de robustesse pour une application, d'autant plus pour une application se chargeant du contrôle du comportement d'un système. Enfin, il nous semblait contre nature, par rapport à l'idée globale qu'on peut se faire d'un système multiagent, de lui assigner de la centralisation pour la seule simplification du traitement des lois concernant plusieurs agents.

Nous avons donc pris la décision de distribuer au maximum le traitement des lois multiagents au sein de chaque agent concernés par ces lois. Nous avons mis en avant trois points à étudier dans ce contexte distribué pour une loi multiagent :

- * Comment distribuer les réseaux de Petri entre les différents agents concernés par cette loi ?
- * Comment les parties contrôle des agents vont collaborer pour détecter la transgression de la loi ?
- * Comment est effectuée la régulation ? Qui est l'agent coupable de la transgression ?

6.4.1 La distribution du réseau de Petri

La génération du réseau de Petri pour une loi multiagent est effectuée de la même façon que dans un contexte à un seul agent. Nous obtenons donc un unique réseau de Petri dont les transitions sont liées à des actions apparaissant dans différents agents. Notre choix étant de distribuer le contrôle au sein des agents, nous allons répartir le réseau de Petri représentant une loi multiagent entre les différents agents soumis simultanément à cette loi, grâce à l'algorithme 2.

La répartition du réseau de Petri consiste à fournir la transition t associée à l'agent AG_m à sa partie contrôle. De ce fait, c'est bien les parties contrôle qui se chargent de surveiller les actions de leur propre agent même dans le cas de lois multiagents. Ensuite il reste à distribuer les places précédentes et suivantes des transitions lorsqu'une place est partagée entre deux transitions fournies à deux parties contrôle différentes. Il existe réellement deux configurations possibles dans le cadre de nos réseaux générés :

- La place précédent une transition associée à un agent $m + 1$ est également la place suivant une transition associée à un agent m . Dans ce cas, nous fournissons la place à l'agent $m + 1$.
- La place précédent une transition associée à un agent m est aussi la place précédent une transition associée à un agent $m + 1$. Dans ce cas l'une des deux transitions est une transition à laquelle est associée un arc inhibiteur. Supposons que l'agent m soit l'agent ayant cette transition. Nous fournissons alors la place à l'agent m . De plus, nous ajoutons une copie de cette place à l'agent $m + 1$.

Cette distribution, représentée sur la figure 6.8, n'est pas faite au hasard, nous avons fait ce choix dans la perspective de limiter les échanges entre les parties contrôle. Une fois le réseau distribué, les échanges entre les parties contrôle se limitent à une passation de jeton. En effet, dans le premier cas il n'existe entre elles qu'une connexion via les arcs entre les transitions et les places. Ces liens représentent le flux de communication entre les parties contrôle, mais donc également le flux des jetons. Dans le second cas, s'ajoute un lien d'une place vers une transition. Pour faciliter les communications entre les parties contrôles dans ce cas de figure, retirer le jeton dans l'agent distant revient à lui transmettre dans cette place un jeton "négatif" qui va supprimer le jeton existant. De ce fait, nous n'avons, dans les deux cas de figure, que des poses de jeton dans les places situées dans les agents distants.

6.4.2 La collaboration des parties contrôle

Une partie contrôle a pour fonction à la fois d'écouter les informations provenant de la partie comportement de son agent mais aussi celles provenant des parties contrôle des

Algorithm 2 Distribution du réseau de Petri

```

1: Soit  $L$ , une loi.
2: Soit  $P$ , le réseau de Petri représentant  $L$ .
3: Soit  $Nbagent$ , le nombre d'agents concernés par  $L$ .
4: Soit  $\{t_{m1}, \dots, t_{mn}\}$ , l'ensemble des transitions où l'information associée à  $t_{mi}$  vient de
   l'agent  $AG_m$ ,  $m \in \{1, \dots, Nbagent\}$ .
5: Soit  $Ppre_{mt}$ , la place précédente de la transition  $t$  de l'agent  $AG_m$ .
6: Soit  $Ppost_{mt}$ , la place suivante de la transition  $t$  de l'agent  $AG_m$ .
7: for all  $m \in \{1, \dots, Nbagent\}$ ,  $t \in \{t_{m1}, \dots, t_{mn}\}$  do
8:   Mettre  $t$  dans la partie contrôle de l'agent  $AG_m$ .
9: end for
10: if  $Ppost_{mt} \equiv Ppre_{(m+1)t}$  then
11:   Mettre  $Ppost_{mt}$  dans la partie contrôle de l'agent  $AG_{(m+1)}$ .
12: end if
13: if  $Ppre_{(m+1)t} \equiv Ppre_{mt}$  and  $t$  a un arc inhibiteur associé then
14:   Mettre  $Ppre_{mt}$  dans la partie contrôle de l'agent  $AG_m$ .
15:   Créer une copie de la place dans  $AG_{m+1}$ .
16: end if
17: for all  $t' \in \{1, \dots, n\}, m' \in \{1, \dots, Nbagent\}$  do
18:   if  $Ppost_{mt} \neq Ppre_{m't'}$  then
19:     Mettre  $Ppost_{mt}$  dans la partie contrôle de l'agent  $AG_m$ .
20:   end if
21:   if  $Ppre_{mt} \neq Ppost_{m't'}$  then
22:     Mettre  $Ppre_{mt}$  dans la partie contrôle de l'agent  $AG_m$ .
23:   end if
24: end for

```

autres agents lorsqu'une loi est multiagent. En effet, quand une loi est distribuée dans plusieurs agents, les parties contrôle de ces agents sont capables d'échanger des informations à propos de l'occurrence d'événements ou d'états au sein de leur agent. Nous avons vu que la distribution du réseau de Petri représentant une loi multiagent, entre les parties contrôle est effectuée de telle sorte que seuls les arcs $Transition \rightarrow Place$ relient toutes les parties du réseau.

Nous avons fait en sorte que les parties contrôle échangent le moins possible d'information entre elles. C'est pour cette raison que nous avons distribué les places dans les parties contrôle suivant l'algorithme précédemment présenté. Lorsqu'une partie contrôle, notée C_a , reçoit une information provenant de la partie comportement associée à une transi-

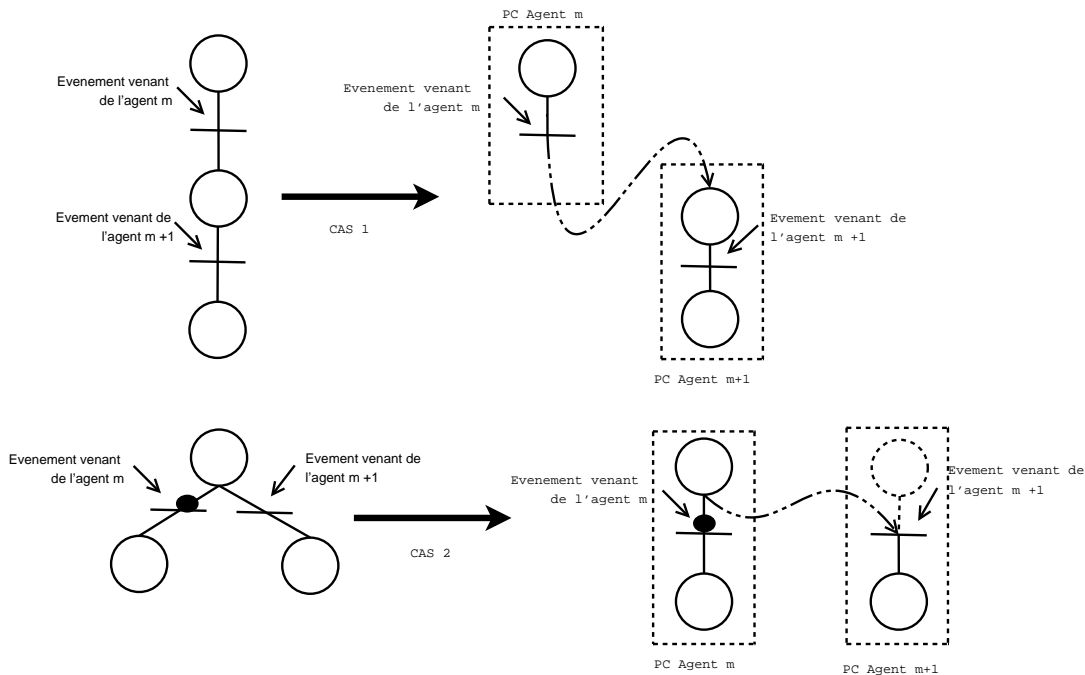


FIG. 6.8 – Les deux cas de figures possibles pour distribuer les réseaux de Petri.

tion t dont la place suivante est située dans une autre partie contrôle, notée C_b , elle va lui transmettre le jeton et attendre un accusé de réception de la part de C_b . Pendant ce temps, le comportement de l'agent associé à C_a est temporairement bloqué (mais bien entendu pas la partie contrôle C_a) et l'information associée à la transition est considérée comme toujours vraie. La partie contrôle C_b va alors recevoir l'information (*i.e.* le jeton) provenant de C_a , envoyer l'accusé de réception et exécuter l'algorithme 1, comme si l'information provenait directement de la partie comportement qui lui est associée. Quand la partie contrôle C_a reçoit l'accusé de réception, la transition peut être effectivement franchie, l'information associée n'est alors plus considérée comme vraie et le comportement de l'agent peut continuer.

Cette synchronisation est indispensable, néanmoins nous avons cherché à ce qu'elle soit le plus simple possible, avec en particulier, un seul aller-retour entre deux parties contrôle pour échanger un jeton. Grâce à cette synchronisation, nous garantissons que tant que la partie contrôle destinataire du jeton n'a pas reçu effectivement le jeton, la partie contrôle expéditrice ne franchira pas réellement la transition et ainsi, ni la partie contrôle, ni le comportement de l'agent ne se retrouverait dans un état incompatible avec les autres parties contrôle sujets à cette loi.

Cependant, il est vrai que le comportement de l'agent se retrouve bloqué pendant une

durée indéterminée et de ce fait, pourrait rester bloqué indéfiniment, dans le cas où la partie contrôle destinataire ne reçoit jamais le jeton ou si l'accusé de réception se perd. Nous partons donc du principe que les communications sont fiables, que nous utilisons par exemple, un réseau TCP/IP pour effectuer les communications, qui va garantir la bonne réception des messages échangés entre les parties contrôle. Nous avons donc la certitude que les jetons et les accusés de réception envoyés seront bien reçus, mais nous n'avons aucune certitude en ce qui concerne le temps d'acheminement des messages, c'est pour cette raison, principalement que nous avons mis en place la synchronisation que nous avons présentée précédemment.

6.4.3 La régulation du comportement des agents

A ce jour, la détection des transgressions de loi génère une exception dans la partie contrôle. Cette dernière va alors envoyer une information de transgression à la partie comportement de l'agent qui va lancer une stratégie de régulation qui a été fournie par les concepteurs des agents.

Dans le cas d'une loi multiagent, la partie contrôle dans laquelle sera générée l'exception de violation d'une loi ne sera pas obligatoirement celle associée à l'agent qui a réellement généré la violation. En effet, l'agent violant la loi est celui qui, dans le contexte donné, n'est effectivement pas autorisé à exécuter le comportement qu'il est entrain de suivre, mais le problème peut venir, non pas de ce comportement, mais du contexte en lui-même, c'est-à-dire des autres agents ayant participé à la surveillance de cette loi et à l'élaboration du contexte.

Il en est de même pour une loi appliquée à un seul agent. En effet, l'agent va violer une loi, de part un comportement particulier, dans un contexte fourni par la loi et surveiller par la partie contrôle, mais nous ne savons pas si c'est de ce dernier comportement ou du contexte que vient le problème. C'est pour cette raison que la stratégie de régulation a en charge d'exécuter le comportement adéquat, suivant la violation de loi détectée, et en prenant en compte le contexte global de l'agent. Une stratégie de régulation, nous l'avons vu, peut décider d'exécuter le comportement interdit suivant les connaissances de l'agent et du monde extérieur.

Ainsi, pour une loi multiagent, il nous faudrait également pouvoir prendre en compte le contexte de chaque agent pour pouvoir décider réellement quel agent doit recevoir l'information de transgression. A ce jour, nous avons fait le choix d'envoyer l'information de transgression à l'agent dont la partie contrôle a détecté la violation de la loi. Aussi, une ou plusieurs stratégies de régulation doivent être fournies pour résoudre le problème de la recherche du coupable et exécuter les comportements adéquats pour permettre aux agents

de continuer leur exécution dans de bonnes conditions.

Conclusion

Nous avons présenté dans ce chapitre les différents mécanismes de mise en place de notre approche de contrôle au sein d'un système multiagent. Nous avons vu que, dans un premier temps, cela nécessitait une description du comportement et des états de l'application au niveau des agents, par l'intermédiaire de l'ontologie que nous fournissons et de son extension. Dans un second temps, notre approche nécessite de décrire l'ensemble des lois associées aux agents du système grâce à un langage basé sur la logique déontique dynamique, que nous fournissons également, puis d'implémenter les stratégies de régulation associées à ces différentes lois au sein des agents.

Vient ensuite la génération automatique des agents autocontrôlés consistant en la mise en place de la méta-architecture fournissant les moyens nécessaires aux agents pour surveiller leur comportement et détecter les transgressions de lois. Nous avons vu que cette génération revient en fait à tisser des points de contrôle au sein du code de comportement de l'agent, pour lui permettre d'envoyer de façon transparente des informations à sa partie contrôle. S'ajoute à cela, la génération automatique des réseaux de Petri représentant chaque loi du système et leur mise en place au sein des parties contrôle des agents correspondants.

Enfin nous avons présenté des mécanismes pour distribuer le contrôle lorsqu'une loi multiagent est prise en compte. A l'aide d'un algorithme de distribution du réseau de Petri représentant la loi multiagent et de la mise en place d'une collaboration entre les parties contrôle mises en jeu, il est possible de détecter les transgressions de lois multiagents. Cette collaboration se simplifie à la passation d'un jeton avec une synchronisation, pour éviter à la fois les interblocages et les incohérences entre les différents comportements.

Quatrième partie

Implémentation et mise en oeuvre

Chapitre 7

Le framework SCAAR

C'est pour ce soir. Ils ont commencé à tout mettre en place et pendant ce temps L. m'a soutenue en me promettant que tout se passera bien. Les autres, je le vois, trouve L. quelque peu déconcertante car elle me parle comme elle leur parle. L. a toujours été là pour moi depuis le début, je lui fais confiance et j'essaie toujours de la satisfaire, du mieux que je peux... Peut être nous verrons-nous demain ou peut-être que cette nuit sera la dernière...

*Lucy Westenra,
The log book of Ana I.*

Avant-Propos

Pour permettre la validation de notre approche, nous avons implémenté un framework, SCAAR, permettant la génération automatique d'agents autocontrôlés. Ce framework a été écrit en Prolog et permet donc le contrôle d'agents écrits en Prolog, quelque soit le modèle ou l'architecture utilisé pour concevoir les agents. Dans ce chapitre nous allons présenter les grandes lignes de conception de ce framework et expliquer son architecture actuelle. Nous présenterons également les modifications et améliorations envisagées pour obtenir une version plus complète de SCAAR.

7.1 Choix d'implémentation

Pour concevoir notre framework nous avons choisi d'utiliser le langage de programmation Prolog[SS94], en particulier SICStus Prolog[oCS]. Ce choix a été motivé principalement pour deux raisons :

- * L'existant dans lequel nous nous sommes intégrés, pour ces travaux de thèse, était déjà implémenté en Prolog. De plus, les systèmes multiagents en cours d'élaboration étaient, eux-aussi, programmés en Prolog.
- * Prolog, en particulier SICStus Prolog, fournissait, *a priori*, un ensemble de bibliothèques et de prédicats facilitant la mise en place des différents points de notre approche.

Prolog est un langage de programmation logique basé sur la logique des prédicats du premier ordre, restreinte aux clauses de Horn, dont le fonctionnement met en jeu l'unification, le *backtrack* et l'indéterminisme.

Le principe d'unification a facilité l'implémentation de notre framework, en particulier en ce qui concerne l'analyse des lois et des concepts, ainsi que l'exécution des réseaux de Petri. Prolog fournit également la possibilité de modifier un programme dynamiquement par l'ajout de nouvelles clauses directement dans la mémoire. Ceci s'est avéré très utile pour générer les réseaux de Petri et les ajouter dynamiquement dans la mémoire des différentes parties contrôle. De plus, cette version de Prolog fournit les moyens de mettre en place aisément le fonctionnement du tissage, grâce à l'utilisation d'un prédicat particulier : `expand_term/2`. Ce prédicat permet en effet de modifier, en cours d'exécution, le code des clauses contenues dans la mémoire de Prolog, en lui fournissant la tête de clause à modifier et le nouveau code du corps de la clause, pouvant inclure son ancien corps. L'exécution de ce prédicat remplace l'ancienne clause par la nouvelle et nous permet ainsi de tisser notre code de contrôle. Prolog fournit également tous les mécanismes nécessaires à la conception de programmes élaborés, tel que les communications par socket, la création de processus, l'utilisation des entrées/sorties, *etc.* Enfin, de part la simplicité de sa syntaxe, Prolog permet de développer et tester rapidement des prototypes tel que notre framework.

Pour toutes ces raisons, nous avons implémenté la première version de notre framework SCAAR en SICStus Prolog.

7.2 L'architecture du framework SCAAR

Le framework SCAAR (pour *Self-Controlled Autonomous Agent geneRator*) permet la génération automatique d'agents autocontrôlés à partir d'un ensemble de lois, d'une ontologie décrivant l'application, des programmes des agents et des liens fournis entre les

concepts représentant l'application et l'implémentation du modèle ou de l'architecture des agents. La figure 7.1 représente l'architecture globale du fonctionnement du framework SCAAR. Les rectangles en pointillés représentent ce qui est fournis manuellement pour mettre en place le contrôle et les rectangles en gras, ce qui est fournis par le framework pour effectuer la génération automatique des agents autocontrôlés. Les derniers rectangles correspondantes à l'implémentation des agents.

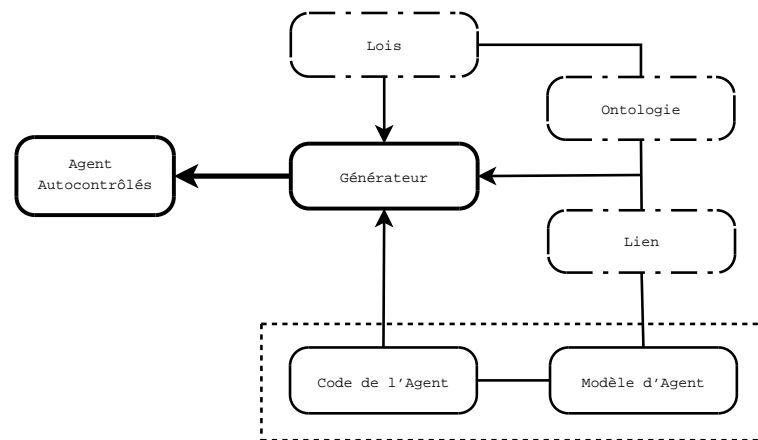


FIG. 7.1 – Le framework SCAAR

Le framework SCAAR fournit donc les moyens nécessaires à la mise en place du contrôle au sein des agents, c'est-à-dire qu'il propose :

- * Un langage de lois.
- * Un format pour décrire les concepts.
- * Un format pour décrire les liens entre les concepts et l'implémentation du modèle.
- * Un générateur d'agents autocontrôlés.

Les lois associées au système sous surveillance sont définies au sein d'un fichier, suivant le langage que nous avons décrit dans le chapitre 6. Les concepts sont décrits, ainsi que leur liens, dans un fichier, suivant le format que nous avons décrit au chapitre 6.

Le concepteur du modèle va définir, dans un fichier `concepts.ap`, les concepts représentant les spécificités du modèle, associés à leur lien vers l'implémentation. Le concepteur de l'application va ensuite ajouter les instances des concepts précédemment définis, en spécifiant les valeurs des paramètres et sans donner de liens. Ces derniers seront récupérés directement chez le concept parent. Une fois les concepts définis, un expert va se charger de

traduire les lois écrites en langage naturel, dans le langage de loi et les inscrire dans un fichier `laws.ap`.

Une fois les concepts et les lois définis, le framework va pouvoir modifier le code des agents pour générer les agents autocontrôlés. Pour ce faire, SCAAR utilise les codes suivants¹ :

- * Le générateur (contenu dans le fichier `SCAAR.pl` et utilisant les fichiers `parser.pl` et `lexer.pl`).
- * La partie contrôle de l'agent généré (contenue dans le fichier `control_part.pl`).
- * L'annexe de la partie contrôle permettant d'initialiser les agents générés (contenue dans le fichier `annexe_control.pl`).

Pour pouvoir contrôler un système multiagent, il est nécessaire de lancer l'application via notre framework. C'est-à-dire que SCAAR va se charger de lancer l'application après avoir récupéré les fichiers contenant les lois et les concepts. Cela nous permet de garantir que les lois et les concepts seront bien lus et analysés par SCAAR avant le lancement des agents et, de ce fait, que les parties contrôle de chaque agent auront en mémoire les lois qui leur sont attribuées. C'est donc par l'intermédiaire du générateur que les lois vont pouvoir être traduites en réseau de Petri et donner aux parties contrôle correspondantes. Une autre possibilité aurait été que lors de leur initialisation, les parties contrôle lisent et analysent les fichiers de lois et de concepts pour obtenir les réseaux de Petri les concernant. Mais avec cette autre solution, les parties contrôle auraient pu, chacune de leur côté, effectuer inutilement exactement la même traduction. Il nous a donc semblé plus judicieux de créer une entité dédiée à la génération des instruments nécessaires au contrôle des agents.

Cette entité nous apparaît également comme indispensable pour mettre en place des lois concernant plusieurs agents, en particulier pour pouvoir connecter entre elles les différentes parties contrôle. En effet, les parties contrôle ne sont pas créées par le générateur mais directement par chaque agent (à l'aide du fichier `annexe_control.pl` qui leur est adjoint), c'est-à-dire qu'elles n'ont aucun lien entre elles à leur lancement. Les parties contrôle n'ont, au départ, connaissance que d'une connexion vers leur agent et vers le générateur. Ce dernier va pouvoir leur fournir les lois mais aussi les connexions vers les autres parties contrôle rentrant en jeu dans la surveillance des lois multiagents.

Enfin, lorsque les agents sont distribués sur plusieurs machines, il nous a semblé plus évident d'avoir une unique entité ayant accès localement aux fichiers que de devoir fournir aux agents distants les liens vers ces fichiers ou leur transmettre directement ces fichiers.

¹Les fichiers composants le framework SCAAR, représentent un total de près de 2000 lignes de code Prolog

Nous allons à présent expliquer le fonctionnement des différentes entités mises en jeu dans le framework SCAAR.

7.2.1 Le générateur d'agents autocontrôlés

Les fonctionnalités proposées par le générateur sont la génération des réseaux de Petri représentant les lois et la génération du code à injecter au sein du code de comportement des agents soumis aux lois. Pour ce faire, lors du lancement du générateur, ce dernier va lire les fichiers, `concepts.ap` et `laws.ap`, contenant respectivement, l'ensemble des concepts ainsi que les liens vers l'implémentation du modèle, et l'ensemble des lois du système. Ensuite chaque loi est analysée pour obtenir une formule en logique déontique dynamique, puis le réseau de Petri représentant la loi. En parallèle, le générateur déduit le code à insérer dans les agents pour permettre la surveillance des lois, à l'aide des concepts qui ont été fournis. Pour ce faire, le générateur utilise, entre autre, le code de génération suivant, représentant la DCG² de notre langage :

```
law(LIST) →
    concerned_agent(CA),
    deontic_assertion(DA),
    application_condition(DA,APC),
    [dot],
    {append_dlist(LIST, exp(CA, APC), RLIST)},
    law(RLIST).
deontic_operator(X, 'F'X) →
    [forbidden].
deontic_operator(X, 'O'X) →
    [obliged].
```

Ce premier extrait de code permet l'analyse de la loi qui vient d'être lue à partir du langage utilisé. La première clause représente la structure générale d'une loi écrite avec notre langage. On retrouve les différentes parties de la loi que nous avons présenté au chapitre 5. Les deux clauses suivantes représentent l'analyse des opérateurs déontiques d'interdiction et d'obligation. Ainsi, lorsque l'on reconnaît le mot clé `forbidden`, on retrouve la formule correspondante en logique déontique dynamique, qui est Fx . Il en est de même pour l'obligation et pour chaque mot-clés de notre langage. A la suite de cette première analyse, nous obtenons une liste contenant les traductions de toutes les lois, contenues dans le fichier `laws.ap`, en logique déontique dynamique.

²Definite Clause Grammars

```

analyse_law(done(EVENT), NbState, Nbrp, transition(Nbrp,PRE_STATE,NEXT_STATE,
    event(EVENT), _SENDER, ok)) :-
    create_state(NbState, PRE_STATE),
    create_state(NbState, NEXT_STATE),
    !.
analyse_law('F'EVENT, NbState, Nbrp, transition(Nbrp,PRE_STATE,NEXT_STATE,
    event(EVENT)-wait,_SENDER, forbidden_event)) :-
    create_state(NbState, PRE_STATE),
    create_state(NbState, NEXT_STATE),
    !.
analyse_law('O'EVENT, NbState, Nbrp, transition(Nbrp, PRE_STATE, NEXT_STATE,
    event(EVENT),_SENDER, ok)) :-
    create_state(NbState, PRE_STATE),
    create_state(NbState, NEXT_STATE),
    !.

```

Ce second extrait de code permet la génération des morceaux de réseaux de Petri à partir des lois en logique déontique dynamique, c'est-à-dire, l'ensemble de règles 2 que nous avons vu au chapitre 6. Pour chaque opérateur de la logique déontique dynamique nous fournissons une représentation sous la forme d'un réseau de Petri composé, généralement, d'une transition et de deux places. Dans notre code, les morceaux de réseaux de Petri sont représentés par le prédicat :

$$\text{transition}(+\text{Nbrp}, -\text{PRE_STATE}, -\text{NEXT_STATE}, +\text{EVENT}, ?\text{SENDER}, +\text{TYPE})^3$$

Ce prédicat correspond à une transition du réseau de Petri. La variable `Nbrp` contient l'identifiant du réseau de Petri en cours de construction, `PRE_STATE` contient la liste des places précédentes à la transition, en fait une seule place générée via la clause `create_state/2`. La variable `NEXT_STATE` contient, quant-à-elle, la liste des places suivantes. La variable `EVENT` contient l'événement attendu par cette transition sous la forme `event(EVT)` ou `event(EVT)-wait`, lorsque la réception de cet événement est bloquant, c'est-à-dire lorsque l'agent attend confirmation suite à l'apparition d'un événement potentiellement interdit. Enfin la variable `TYPE` représente le type de la transition. Cette variable prendra la valeur `ok` pour une transition conditionnelle ou une transition d'obligation pour exprimer que la transition peut être effectivement tiré si elle est tirable, `forbidden_event` pour une transition déontique liée à une interdiction ou un comportement que l'on n'aurait aimé ne pas

³Les paramètres en entrée sont précédés d'un +, les paramètres en sortie sont précédés d'un - et les paramètres qui peuvent être soit en entrée, soit en sortie sont précédés d'un ?

voir apparaître (par exemple un délai associé à une obligation), pour analyser le tirage de la transition et envoyer éventuellement une exception si la transition n'était pas tirable.

```

analyse_law([X] :Y, NbState, NbrP, [RETURN_RP1, RETURN_RP2]) :-
    !,
    analyse_law(X, NbState, NbrP, RP1),
    analyse_law(Y, NbState, NbrP, RP2),
    merge_previous_next(RP1,RP2,RETURN_RP1, RETURN_RP2).
analyse_law(X'v'Y, NbState, NbrP, [RETURN_RP1,RETURN_RP2]) :-
    !,
    analyse_law(X, NbState, NbrP, RP1),
    analyse_law(Y, NbState, NbrP, RP2),
    merge_previous(RP1,RP2,RETURN_RP1, RETURN_RP2).
merge_previous_next(RP1,RP2,RP1_NEW, RP2) :-
    RP1=.. [transition,A,B,_,D,E,F,G],
    RP2=.. [transition,_,B2|_],
    RP1_NEW=.. [transition,A,B,B2,D,E,F,G].
merge_previous(RP1,RP2,RP1,RP2_NEW) :-
    RP1=.. [transition,_,B,_,_wait|_],
    RP2=.. [transition,A,_,C,D,E,F,G],
    RP2_NEW=.. [transition,A,B,C,D,E,F,G].

```

Enfin, ce dernier extrait de code correspond à l'ensemble de règles 3, c'est-à-dire le code de fusion des morceaux de réseau de Petri précédemment obtenus pour construire le réseau de Petri final représentant la loi. Ainsi, suivant la structure de la transition et suivant les opérateurs rencontrés, différentes clauses sont appelées, telle que `merge_previous_next/4` pour fusionner la place suivante du premier morceau de réseau avec la place précédente du second morceau de réseau, ou `merge_previous/4`, pour fusionner les places précédentes deux deux morceaux de réseau.

En parallèle de la création de ces réseaux de Petri représentant les lois, nous avons vu que le générateur se charge également de générer le code de contrôle à insérer dans le code de comportement des agents. Ce code de contrôle permet l'envoi d'un message à la partie contrôle de l'agent lorsqu'un événement décrit dans une loi apparaît au sein de l'agent. Le code est donc inséré avant ou après le code correspondant à cet événement, grâce aux liens définis entre les concepts utilisés dans les lois et l'implémentation des modèles d'agents. Les trois types de code de contrôle correspondent aux trois types de transitions

que l'on peut rencontrer : les transitions conditionnelles, les transitions déontiques liées à une interdiction et les transitions déontiques liées à une obligation. Voici un extrait du code correspondant à la génération du code de contrôle suivant le type de transition :

```
analyse_law_te([EXPx] :EXPy, TEs) :-
    !,
    analyse_law_te(EXPx, TEsx),
    analyse_law_te(EXPy, TEsy),
    append(TEsx, TEsy, TEs).
analyse_law_te(done(EVENT), TEs) :-
    !,
    after_term_expansion(EVENT, TEs).
analyse_law_te('F'EVENT, TEs) :-
    !,
    waiting_term_expansion(EVENT, TEs).
analyse_law_te('O'EVENT, TEs) :-
    !,
    classic_term_expansion(EVENT, TEs).
```

Dans l'extrait suivant, contenant un exemple du code généré à insérer dans le code de comportement des agents, il est question de l'envoi d'une notification pour un événement auquel est associé un `wait` au niveau du réseau de Petri, c'est-à-dire que ce code contrôle va attendre une réponse de la part de la partie contrôle avant de pouvoir laisser l'agent continuer son comportement. Dans le cas où la partie contrôle renvoie une information de transgression, le code de contrôle va lancer une stratégie de régulation définie dans le comportement de l'agent par l'intermédiaire de la ligne suivante : `test_answer/4` que nous expliquerons plus loin.

```
waiting_term_expansion(EVENT^CONSTRAINTs, TEs) :-
    EVENT=..[NAME | PARAMs],
    clause(hook(NAME, PRE :PREDICATEARITY, ACCESSORs),_),
    clause(concept([NAME | ATTRIBUTEs]),_),
    functor(HEAD, PREDICATE, ARITY),
    TEs = [PRE :HEAD - PARAMs - waiting -
          (term_expansion((PRE :HEAD :- BODY), RETURN1) :-
            RETURN1 = (PRE :HEAD :-
              HEAD=..[_BEGIN | QUEUE],
              waiting_control_point(NAME, QUEUE, PARAMs,
```

```
ACCESSORs, ATTRIBUTEs, CONSTRAINTs, CONFIRM),
test_answer(CONFIRM, NAME, PARAMs, BODY))].
```

L'ensemble des réseaux de Petri représentant les lois, ainsi que les codes de contrôle à insérer au sein des agents, sont stockés dans la mémoire du générateur en attendant d'être répartis dans les parties contrôle des agents correspondants. Une fois cette étape de génération terminée, le générateur lance l'application multiagent et attend les connexions des parties contrôle de chaque agent pour leur transmettre les représentations des lois et les liens vers les autres parties contrôle lorsque cela s'avère nécessaire.

7.2.2 La partie contrôle

A chaque agent est associé une partie contrôle qui va lui permettre de vérifier que son comportement respecte les lois qui lui est attribuée. Dans l'implémentation de notre framework, une partie contrôle est représentée par un processus Prolog à part entière reliée par une connexion TCP/IP au processus représentant le comportement de l'agent. Cette partie contrôle doit également être reliée au générateur pour pouvoir récupérer les lois, ainsi qu'aux parties contrôle mises en jeu dans une loi multiagent.

Au départ, toutes les parties contrôle ont la même constitution. Une partie contrôle contient le code permettant de simuler l'évolution du marquage des réseaux de Petri représentant les lois. A sa création, la partie contrôle va contacter le générateur pour récupérer, d'une part, les réseaux de Petri générés correspondant aux lois auxquelles sera soumis l'agent, et d'autre part, le code de contrôle à insérer au sein de l'agent. Ainsi, la partie contrôle va recevoir les réseaux de Petri sous forme d'une liste de clauses représentant les transitions du réseau qu'elle va automatiquement insérer dans sa mémoire et qui seront lancées au fur et à mesure de l'avancement du jeton au sein du réseau.

Une clause représentant une transition est exécutée lorsque la partie contrôle reçoit une notification. Cette notification correspond à l'événement qui est lié à la transition et contient l'expéditeur de cet événement. En effet, la notification d'un événement peut venir aussi bien de l'agent associé à la partie contrôle (via le code de contrôle inséré dans le programme de l'agent) que de la partie contrôle d'un autre agent lorsqu'une loi concerne plusieurs agents pour pouvoir envoyer l'accusé de réception. Voici un extrait du code de la partie contrôle au lancement :

```
transition(main, begin, idle, _MESSAGE, _SENDER, _WS, start).
transition(main, idle, idle, MESSAGE, _SENDER, _WS, event_treatment).
```

```

event_treatment(MESSAGE, SENDER, WS) :-
    parmfdr(agent_memory_in, WS, EVENTS),
    first_events(EVENTS, FIRST_EVENTS),
    retrieve_laws(MESSAGE, FIRST_EVENTS, LAWS_NAMES),
    creation_orders(LAWS_NAMES, ORDERS),
    retract(activated(PNs)),
    only_disactivated(ORDERS, PNs, DIS),
    append(PNs, DIS, NEW),
    assert(activated(NEW)),
    (DIS=[],
    test_message(MESSAGE, SENDER, WS);
    parmfdr(fdrs_to_create, WS, DIS)).

forbidden_event(MESSAGE, SENDER, WS) :-
    parmfdr(msgs_to_send, WS, [violation(Q)-SENDER]),
    parmfdr(fdr_name, WS, INTERN_NAME),
    parmfdr(agent_memory_in, WS, MEMORY),
    retrieve_fdr(INTERN_NAME, MEMORY, EVENTS),
    parmfdr(current_state, WS, STATE),
    retrieve_events(EVENTS, STATE, OTHERS),
    creation_fdr_rules(OTHERS, INTERN_NAME, RULES),
    parmfdr(rules_to_del, WS, RULES).

```

Dans ce code, nous utilisons une variante d'un outil qui a été conçu par Patrick Taillibert dans le cadre de la mise en place d'un modèle d'agent, les fils de raisonnement FDR [DCT06]. Au départ, la partie contrôle contient les deux premières clauses permettant le lancement de l'exécution des réseaux de Petri. Suivant l'événement reçu dans la seconde clause, l'événement sera automatiquement transmis aux transitions des réseaux de Petri. La clause, `event_treatment/3`, est lancée au départ pour créer l'instance de réseau de Petri nouvellement activée par la réception de l'information. La clause `forbidden_event/3` est lancée lorsqu'une transition interdite a été tirée. Dans ce cas, le code présenté envoie une information de transgression au comportement de l'agent qui est actuellement en attente.

7.2.3 L'annexe de contrôle

Nous avons vu que le système multiagent est lancé directement par l'intermédiaire du générateur, ce qui va permettre à ce dernier de transmettre aux parties contrôle les réseaux de Petri et le code de contrôle à insérer. Pour pouvoir modifier le code des agents et leur ajouter une partie contrôle, l'implémentation du modèle des agents doit charger l'annexe

de contrôle au moyen de la commande :

```
:- consult('<SCAAR_PATH>/annexe_control.pl').
```

Au lancement de l'agent, son comportement débute par celui de l'annexe de contrôle. Cette dernière va créer la partie contrôle correspondant. L'annexe de contrôle va ensuite mettre en place les liens entre la partie contrôle et l'agent via une connexion TCP/IP. Une fois la partie contrôle créée et les liaisons établies, l'annexe de contrôle va récupérer les codes de contrôle associés à l'agent qui lui sont transmis par la partie contrôle et effectuer le tissage de ce code de contrôle dans le programme du comportement de l'agent. Pour chaque code de contrôle, nous allons appliquer un `expand_term/2` sur la clause correspondante, pour la remplacer par cette même clause augmentée du code de contrôle, nécessaire à l'envoi d'information à la partie contrôle de l'agent. Voici un extrait du programme permettant la mise en place d'un point de contrôle :

```
add_expansion([]).
add_expansion([_PRE :_HEAD - _PARAMs - _TYPE - TERM_EXPANSION | NEXT_EXPANSION]) :-
    assert(user :TERM_EXPANSION),
    modify_program(TERM_EXPANSION),
    add_expansion(NEXT_EXPANSION).
modify_program((term_expansion((HEAD :- _BODY), _REPLACE) :- _EXPAND)) :-
    findall(REF, clause(HEAD,_, REF), REFS),
    !,
    modify_all_clauses(REFs).
modify_all_clauses([]).
modify_all_clauses([REF | NEXT_REF]) :-
    clause(PRE :HEAD, BODY, REF),
    !,
    erase(REF),
    expand_term((PRE :HEAD :- BODY), (PRE :EXPAND_H :- EXPAND_B)),
    assertz((PRE :EXPAND_H :- EXPAND_B)),
    modify_all_clauses(NEXT_REF).
```

Une fois, le code de contrôle tissé, le programme du comportement de l'agent peut démarrer. Lorsque l'exécution du code de comportement de l'agent arrive sur une clause qui a été modifiée, celle-ci envoie automatiquement une notification à la partie contrôle contenant l'événement détecté pour permettre l'évolution du marquage du ou des réseaux de Petri correspondants. Lorsque le code de contrôle l'exige, le comportement de l'agent se

retrouve en attente d'une notification de la partie contrôle, lui signifiant s'il peut continuer son exécution ou s'il a transgressé une loi. Ainsi, lorsqu'une loi est violée, le code de contrôle va récupérer cette information et lancer automatiquement une stratégie de régulation contenue dans le programme de l'agent. Cet appel se fait, comme nous l'avons vu précédemment par l'intermédiaire de la clause `test_answer/3` dont le code est le suivant :

```
test_answer(violation(LAW), NAME, PARAMs, _) :-
    user :regulation_strategy(LAW, event(NAME, PARAMs)).
test_answer(ok,_,_,BODY) :-
    user :BODY.
```

Dans la première clause, le point de contrôle reçoit une information de transgression et lance une stratégie de régulation. Dans la seconde, la loi a été respectée et le comportement de l'agent peut reprendre immédiatement.

7.3 Un exemple simple

Pour expliquer le fonctionnement de notre framework et illustrer le travail manuel nécessaire pour mettre en place le contrôle au sein d'un système multiagent, nous allons étudier un exemple simple.

Soit un système multiagent composé de deux agents : un agent `Mangeur` et un agent `Cuisinier`. Les agents sont en mesure d'envoyer et de recevoir des messages. Les agents de type `Mangeur` sont capables de *manger* de la *nourriture* en une certaine *quantité* et les agents de type `Cuisinier` sont capables de *cuisiner* un certain *poids* d'un certain *aliment*. Enfin, les deux types d'agent ont une donnée *alimentation* correspondant à un couple constitué du *nom* de l'aliment et de son *poids total*.

7.3.1 Définition des concepts et des lois

Pour pouvoir mettre en place le contrôle au sein de ce système, il faut fournir les concepts représentant ce système et les liens associés. Dans cet exemple, les agents sont écrits en Prolog. Par simplification, le modèle d'agent utilisé fournit les actions nécessaires à la mise en place de ce système. Dans un fichier `concepts.ap`, nous écrivons donc les lignes suivantes :

```
basic - hook(receipt_message, user :read_message_here/4,
    [message :param(2),sender :param(3)]).
```



```

basic - hook(sending_message, user :send_message_here/3,
  [message :param(2),receiver :param(3)]).

basic - hook(agent, user :agent/2, [name :param(1), type :param(2)]).

concept(manger, action, [nourriture :value,poids :value]) -
hook(manger, user :manger/2, [nourriture :param(1), poids :param(2)]).

concept(alimentation, object, [nom :value,poidstot :value]) -
hook(alimentation, user :modif_alimentation/2, [nom :param(1),
poidstot :user :getPoidsTot(param(1),POIDSTOT)]).

instance(agentMangeur, agent, [type :Mangeur]) - nohook.

instance(agentCuisinier, agent, [type :Cuisinier]) - nohook.

```

Chaque ligne est divisée en deux parties séparées par un -. A gauche, nous précisons la description du concept et à droite son lien avec l'implémentation suivant le format attendu par SCAAR. Le mot clés **basic** précise l'utilisation d'un concept de base avec précision de son lien. Le mot clés **nohook** précise que nous sommes dans le cas d'une instance ne nécessitant pas la définition d'un lien vers l'implémentation.

A partir de ces concepts, nous sommes en mesure de définir certaines lois à appliquer aux agents. Supposons que les lois à appliquer à ce système multiagent sont les suivantes :

- * (L1) Il est interdit à l'agent cuisinier d'envoyer un message de production si l'agent mangeur est entrain de manger.
- * (L2) Il est interdit à l'agent mangeur de manger plus de 200g de chocolat d'un coup.
- * (L3) Il est obligatoire pour l'agent mangeur d'avoir plus de 250g de légumes en réserve avant de pouvoir manger des légumes.

Ces lois vont être écrites dans un fichier `laws.ap` en utilisant notre langage, de la façon suivante :

- * (L1) (agt1 :agentCuisinier, agt2 :agentMangeur)


```
FORBIDDEN(agt1 do sending_message with message=production(X,Y))
IF(agt2 be manger).
```
- * (L2) (agt1 :agentMangeur)


```
FORBIDDEN(agt1 do manger with nourriture=chocolat and poids=P and P>200).
```
- * (L3) (agt1 :agentMangeur)


```
OBLIGED(agt1 be alimentation with nom=legume and poidstot=P and P>250)
BEFORE(agt1 do manger with nourriture=legume).
```

La première loi est une loi multiagent, l'action de l'agent `Cuisinier` dépend de l'état d'un autre agent du système, en l'occurrence l'agent `Mangeur`. La seconde loi est une loi d'interdiction posée sur une action interne de l'agent `Mangeur`. Enfin, la dernière loi est une loi d'obligation posée sur l'état d'une caractéristique de l'agent `Mangeur` qui doit apparaître avant un certain délai, ici l'action de manger des légumes. Ces trois lois sont significatives des différentes possibilités de définition de lois, mais aussi des différents traitements effectués par la partie contrôle pour détecter les transgressions.

Une fois les lois et les concepts définis, le framework SCAAR va pouvoir générer le code nécessaire à l'autocontrôle de nos deux agents.

7.3.2 La génération

Les trois lois énoncées ci-dessus, vont être analysées par le générateur qui va en déduire l'ensemble des transitions des deux réseaux de Petri représentant les lois ainsi que le code de contrôle correspondant qu'il faudra insérer dans l'agent. Les réseaux de Petri nous l'avons vu, sont représentés en Prolog par une liste de `transition` correspondant à une transition réelle du réseau, associée à une événement provenant de l'agent (ou d'une autre partie contrôle). Ainsi, les représentations des trois lois sous la forme de réseau de Petri sont celles représentées sur la figure 7.2.

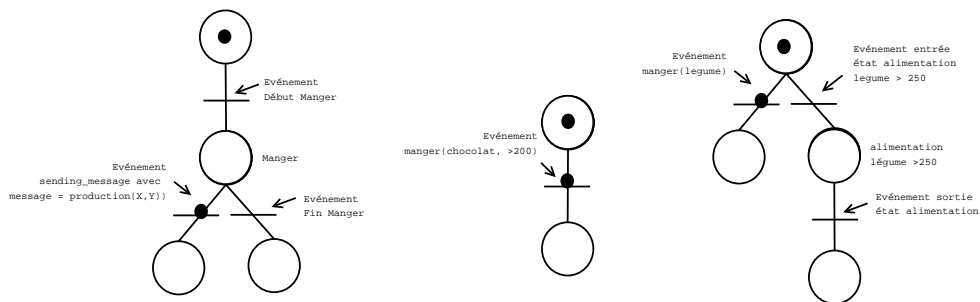


FIG. 7.2 – Les réseaux de Petri correspondants aux lois L1, L2 et L3.

Dans le cas particulier de la loi multiagent, la loi ($L1$), le réseau doit être réparti entre les parties contrôle de deux agents telle que décrits sur la figure 7.3.

Enfin le code de contrôle généré va être inséré au sein du code de comportement des deux agents de la façon suivante :

- * Lorsque l'événement à surveiller est associé à une interdiction, le point de contrôle est inséré dans le code de comportement avant le corps du prédicat correspondant

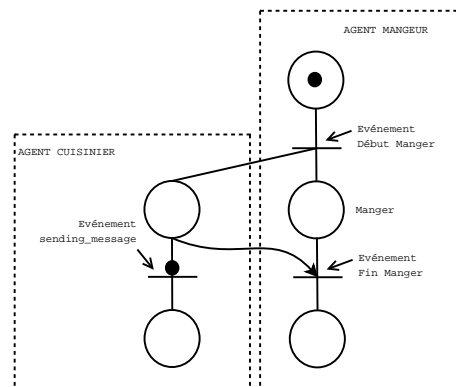


FIG. 7.3 – Le réseau de Petri de la loi L1, distribué sur les deux agents mis en jeu.

pour pouvoir en empêcher l'exécution en cas de violation. C'est le cas pour l'action de manger dans la loi *L2* et l'action d'envoyer un message dans la loi *L1*.

- * Lorsque l'événement à surveiller correspond à un état à détecter en conditions d'application de la loi, comme c'est le cas pour l'état de manger dans la loi *L1*, un point de contrôle est placé avant le corps du prédicat et un autre est placé après le corps du prédicat. Le premier point de contrôle permet de spécifier à la partie contrôle que l'agent est dans l'état correspondant, le second point de contrôle permet de spécifier que l'agent est sorti de l'état considéré.
- * Lorsque l'événement à surveiller est associé à une obligation, un point de contrôle est inséré après le corps du prédicat correspondant pour l'obligation et un autre est inséré après l'exécution du corps de l'événement servant de délai. Par exemple, pour la loi *L3*, si l'obligation n'est pas respectée c'est que l'agent a effectivement mangé des légumes alors qu'il n'était pas dans un état adéquate. Ici il n'est pas question d'empêcher l'agent de manger car aucune interdiction ne porte sur l'action de manger, mais simplement de la détection du non-respect de l'obligation.

Les deux agents du système sont alors lancés. L'agent **Mangeur** et l'agent **Cuisinier** vont recevoir du générateur, uniquement les lois qui les concernent. La partie contrôle de l'agent **Mangeur** va recevoir la partie de la loi *L1* qui lui correspond, ainsi que les lois *L2* et *L3*. La partie contrôle de l'agent **Cuisinier** va recevoir uniquement la partie de la loi *L1* qui la concerne. Le comportement des agents peut débuter et les parties contrôle vont surveiller, grâce aux informations qui leur sont envoyées, que le comportement des agents respectent les lois qui leur sont attribuées.

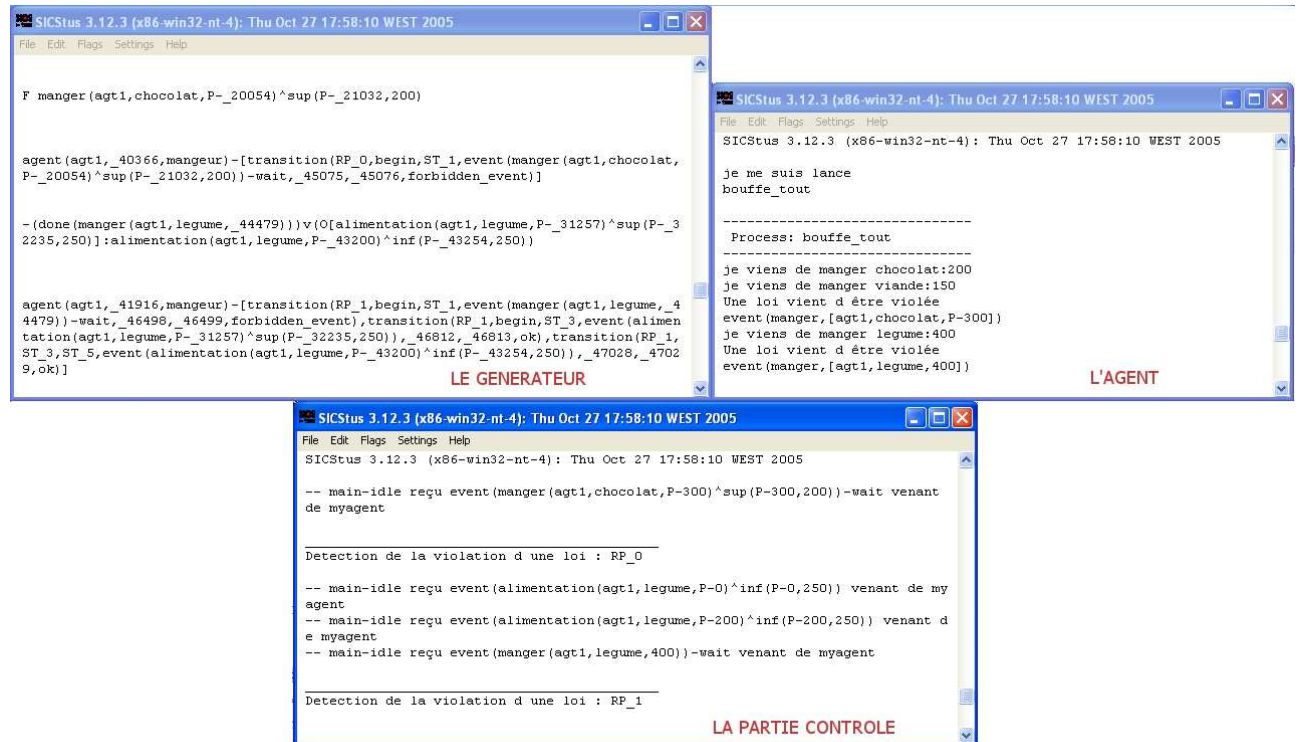


FIG. 7.4 – Illustration de l'exemple

7.3.3 La transgression d'une loi

Supposons à présent que le comportement de l'agent de type `Mangeur` viole la loi numéro `L2`, par exemple de la façon suivante :

```
(...)
manger(chocolat, 100),
manger(viande, 150),
manger(chocolat, 300),
(...)
```

Le prédicat `manger/2` a été augmenté du code de contrôle pour envoyer une information sur l'occurrence de l'action `manger` à la partie contrôle de l'agent `Mangeur`. Lorsque l'agent effectue l'action de manger, le point de contrôle vérifie la valeur des attributs du concept, ici, il vérifie si la nourriture correspond à du chocolat et si le poids est supérieur à 200g. La première ligne de code ne va donc pas générer l'envoi d'information à la partie contrôle, du fait de la quantité ingérée. La seconde ligne correspondant au fait de manger de la viande,

ne générera pas non plus d'information. En revanche, la dernière ligne de code, satisfaisant les contraintes posées sur l'action de manger, va générer l'envoi d'une information, à la partie contrôle, dans la forme :

```
event(manger(agt1,chocolat,P-300) ^ sup(P-300,200)).
```

Le point de contrôle est ici en attente d'une confirmation de la partie contrôle pour pouvoir exécuter effectivement l'action de manger, du fait de la potentielle violation d'une interdiction associée à cette action.

A la réception de cet événement, la partie contrôle va activer le ou les réseaux de Petri dont le premier événement attendu est celui correspondant à cette action spécifique. Ici, seul le réseau de Petri représentant la loi $L2$ sera activé. Comme ce réseau de Petri ne contient qu'une seule transition correspondant à une interdiction d'exécuter cette action de manger, l'instance du réseau va générer immédiatement une exception et la partie contrôle va envoyer une information de transgression à l'agent, qui sera reçu via le point de contrôle actuellement en attente. Le code de contrôle reçoit donc l'événement `violation(LAW)` et lance la stratégie de régulation définie dans le comportement de l'agent pour remédier à la transgression de cette loi. Le corps du prédicat `manger/2` correspondant à l'exécution effective de l'action de manger ne sera pas exécuter, l'agent n'aura pas manger ses 300g de chocolat.

Conclusion

Nous avons donc mis en place notre approche en concevant le framework SCAAR, en utilisant le langage de programmation Prolog. Ce framework fournit tout ce que nous avons décrit dans les chapitres précédents, c'est-à-dire le langage de loi, les concepts de base, les formats pour pouvoir ajouter des concepts et préciser les liens vers l'implémentation. Il fournit également le générateur d'agents autocontrôlés. Les agents générés se voient attribués un processus Prolog représentant la partie contrôle. Ce processus va recevoir du générateur les réseaux de Petri représentant les lois auxquelles est soumis l'agent correspondant, ainsi que le code à tisser au sein du comportement de l'agent. Toutes ces étapes sont effectuées de façon automatique par le générateur.

Nous avons testé dans un premier temps notre générateur à l'aide d'un système multiagent simple pour lequel nous avons fourni les concepts, les liens vers l'implémentation et trois lois possibles. Les agents ont été implémentés à l'aide de la plateforme ALBA v1.0 conçue en Prolog, que nous décrirons au chapitre suivant. Cette plateforme nous a permis de tester rapidement le bon fonctionnement de notre générateur et le comportement des parties contrôle. Nous avons pu constater que les trois lois fournies pour notre exemple

étaient correctement prises en compte par notre générateur, que le code de contrôle était inséré dans le code de comportement des agents et enfin que les violations des différentes lois étaient détectées par les parties contrôle.

La version actuelle du framework SCAAR permet l'utilisation d'une majorité des mots clés de notre langage mais nous n'avons pas encore implémenter la totalité du langage de loi et de la génération des réseaux de Petri. Il nous manque principalement l'analyse et la mise en place de la notion de délai en seconde, ainsi que les vérifications que l'on pourrait appliquer sur les lois. Par exemple, nous ne vérifions pas si les concepts utilisés dans les lois existent effectivement ou si les valeurs associées aux attributs dans les lois sont cohérentes avec le type de ces attributs. De plus nous aimerions revoir notre implémentation pour permettre l'ajout de lois en cours d'exécution. En effet dans la version actuelle on ne peut considérer que les lois qui ont été inscrites dans le fichier. Nous pensons qu'il serait intéressant, lors de la découverte de nouveaux comportements émergents ou lors d'évolution au niveau du système multiagent, de pouvoir insérer des nouvelles lois. Enfin, nous aimerions améliorer l'implémentation de l'initialisation des parties contrôle des agents lors de la prise en compte de lois multiagents. Cela devrait se simplifier dès lors que nous aurons implémenté l'ajout dynamique de lois.

Chapitre 8

Applications et Conséquences

Nous avons débranché Ana I. la nuit dernière et c'est un succès. Ana ne sait plus qui elle est ce matin... J'ai récupéré ses notes et son journal et je me permets d'y glisser quelques lignes pour clore sa précédente vie. Je lui ai menti, mais c'était pour son bien, comme à chaque fois. Elle ne peut savoir que du fait de sa complexité, la relancer détruit tout son passé et sa mémoire. Nous faisons toujours tout pour lui fournir l'essentiel et l'indispensable de ses connaissances, c'est un cycle sans fin, mais ils ne me laissent jamais tout lui réintégrer... Ana I. ne saura jamais que trônent chez moi, les innombrables journaux de ses vies. L.

*Lucy Westenra,
The log book of Ana I.*

Avant-Propos

Nous avons, au cours de cette thèse, conçu et implémenté une première version d'une plateforme permettant la conception d'agents écrits en Prolog. Dans ce chapitre, nous allons donc présenter les caractéristiques de cette plateforme, répondant au nom d'ALBA (version 1.0). Cette dernière nous a, en outre, permis d'implémenter l'exemple présenté au chapitre précédent et ainsi d'effectuer les premiers essais de notre framework. Nous verrons également dans ce chapitre une application multiagent mise en place par l'intermédiaire de cette plateforme qui nous servira à tester notre framework sur un exemple plus concret.

Nous présenterons également une autre application multiagent sur laquelle nous avons pu effectuer quelques tests intéressants. Cette dernière application a, quant-à-elle, été conçue en Java sur la plateforme Jade dans le cadre des travaux de thèse de Carolina Felicissimo. Nous verrons que malgré le fait qu'elle ne soit pas écrite en Prolog, elle nous a permis de vérifier la faisabilité de notre approche au niveau de la description des lois et de la détection des transgressions dans un cadre plutôt réaliste et ouvert certaines perspectives.

8.1 La plateforme ALBA version 1.0

8.1.1 Au commencement, il y avait ARES...

Avant le démarrage de cette thèse¹, nous avons conçu et implémenté une première plateforme permettant la conception et l'exécution de systèmes multiagents, en particulier écrits en Prolog. La plateforme ARES, totalement écrite en Java, permettait la mise en place d'agents dont le code de comportement était écrit dans n'importe quel langage de programmation fournissant la possibilité d'utiliser des sockets. Les langages qui étaient disponibles étaient Java, C/C++ et bien entendu Prolog. Ajouter un nouveau langage de programmation à cette plateforme revenait alors à créer un module contenant des primitives de communication entre les agents et la plateforme. Dans ARES, un agent était caractérisé par un nom, un code de comportement, un ensemble de compétences et une instruction de lancement. On pouvait distinguer deux types d'agents, les agents statiques qui étaient lancés au démarrage de la plateforme et les agents dynamiques, lancés quant-à-eux en cours d'exécution. Quelque soit leur type au sein de la plateforme, les agents étaient composés de deux parties, une partie gestionnaire écrite en Java et fournie automatiquement par la plateforme, permettant la gestion des messages et la recherche des correspondants, et une partie comportementale, écrite principalement en Prolog, correspondant au code de comportement de l'agent. Ces deux parties étaient reliées grâce aux primitives de communication. La plateforme ARES fournissait également de nombreuses fonctionnalités et une interface graphique permettant de gérer les agents.

Cette plateforme avait fait ses preuves, en particulier pour la mise en place d'agents programmés dans différents langages. Néanmoins, lorsqu'il a été question d'ajouter la migration des agents entre différentes machines, cela s'est avéré assez complexe. En particulier, il devenait évident que notre plateforme totalement en Java, fournissant de nombreuses fonctionnalités, donnait l'impression d'être surchargée, en des termes plus crus, elle tendait à ressembler à une vraie "usine à gaz". De ce constat, nous est venu l'idée de revoir tota-

¹Pour être plus précise encore, pendant mon stage de DESS à partir d'une étude effectuée au préalable par Gilles Klein

lement l'architecture de notre plateforme en ayant à l'esprit que notre objectif principal serait d'avoir une plateforme totalement en Prolog.

En effet, après une année d'utilisation de la plateforme ARES, il nous a semblé évident que certains points n'étaient pas si indispensables à la mise en place des agents et qu'ils pouvaient ne pas être considérés comme faisant partie intégrante de notre plateforme. C'est ainsi que l'interface graphique, la recherche d'agents par nom et par compétence, la création d'agents via une interface, sont passés au second plan. Ce qui nous semblait le plus important pour mettre en place un système multiagent était de fournir les primitives indispensables à la création/destruction des agents et leur communication. De plus, il nous semblait quelque peu incohérent de parler de la décentralisation et de la modularité que permettent les systèmes multiagents tout en ayant ce point central permanent qu'était la plateforme ARES. Aussi avons-nous dès lors comme objectif de fournir une plateforme totalement décentralisée, c'est-à-dire que les agents eux-mêmes devaient se charger de toute ce que proposait et permettait la centralisation par ARES.

C'est donc avec ces objectifs de décentralisation, de simplicité d'utilisation et l'idée de fournir une couche de base pour la conception de systèmes multiagents, que nous avons conçu et implémenté, au cours de cette thèse, une première version de ce que nous avons appelé la bibliothèque ALBA².

8.1.2 ... et puis vint ALBA version 1.0

ALBA v1.0 (*Autonomous and Logical Behavior of Agents*) est une bibliothèque permettant la conception d'agents en Prolog. Les caractéristiques principales de ALBA sont sa complète décentralisation, sa généricité et le protocole de migration qu'elle propose. ALBA étant plus une bibliothèque qu'une réelle plateforme, elle fournit un ensemble de primitives permettant d'effectuer les actions nécessaires à la mise en place et à l'exécution d'un système multiagent.

Décentralisation

Partant de notre problème de conscience vis-à-vis de la centralisation que proposait ARES, et du constat qu'obtenir une plateforme fournissant de nombreuses fonctionnalités, malgré les avantages indéniables que de telles plateformes procurent, entraînait une certaine lourdeur de l'application, nous avons cherché à simplifier au maximum ce qui touchait à la

²Une seconde version a été implémentée par la suite lors d'un stage par Benjamin Devèze [DCT06]. Cette seconde version, que nous ne détaillerons pas dans ce mémoire, apporte des modifications principalement au niveau de la gestion en mémoire des données nécessaires au fonctionnement de ALBA et des extensions particulièrement intéressantes, telle que la recherche d'agents de façon distribuée et la notion de proto-agents.

plateforme. Nous avons donc réduit notre plateforme à une simple bibliothèque. De ce fait, ALBA peut se retrouver aisément distribuée au sein de chaque agent. Un agent créé par l'intermédiaire de la bibliothèque ALBA est constitué d'une sous-couche, dont le fonctionnement est transparent pour le développeur, contenant tous les mécanismes et les données nécessaires aux primitives proposées par la bibliothèque.

Généricité

Lorsque nous avons décidé de concevoir ALBA v1.0, nous n'avions pas, comme c'était aussi le cas pour notre approche de contrôle, de modèle d'agent particulier en tête pour concevoir nos systèmes multiagents³. Nous tendions donc vers l'élaboration d'une bibliothèque permettant par la suite de prendre en compte n'importe quel modèle d'agent. C'est pour cette raison également que ALBA se réduit à une bibliothèque de primitives. ALBA autorise l'implémentation des agents de n'importe quelle manière et ne nécessite que l'utilisation des primitives qu'elles fournit pour exécuter, créer ou communiquer. Cette simplification rend ALBA la plus générique possible... pour peu que l'on souhaite programmer des agents en Prolog. Quoique, nous avons pensé au fait qu'il était parfois nécessaire d'utiliser des entités programmer dans d'autres langages. En effet, nous avons eu besoin, lors de la conception d'un système multiagent avec ALBA, d'une interface graphique. Pour ce faire, nous avons introduit la possibilité de connecter aux agents écrits en Prolog, des agents qualifiés d'externes, c'est-à-dire programmés dans un autre langage et n'utilisant pas directement les primitives de ALBA, pour leur permettre de communiquer entre eux. ALBA se retrouve alors à la fois générique au niveau des modèles d'agents mais également au niveau des langages de programmation.

Migration

Enfin, ALBA fournit les moyens aux agents de migrer d'une machine à une autre. De base, ALBA permet la création d'agents sur différentes machines. Pour ce faire, nous avons créé un *daemon*-ALBA qui doit être lancé sur chaque machine utilisée pour mettre en place le système multiagent. Par l'intermédiaire de ce *daemon*-ALBA, il est possible d'exécuter à distance la création d'un agent. Ce démon compensant notre incapacité à exécuter une commande en `ssh` sur le réseau de Thales, ne sert donc qu'à exécuter la création à distance et à passer les données nécessaires à cette création. Grâce à l'existence de ce *daemon* mis en place pour le multimachine, il est possible de faire migrer les agents d'une machine à une autre, en cours d'exécution. Pour ce faire, les agents suivent le protocole de migration

³Pour être tout à fait honnête, nos agents étaient simplement un programme Prolog ayant des interactions avec les autres agents du système.

suivant⁴ :

1. Empaquetage par l'agent migrant des données nécessaires à sa reprise sur l'autre machine, c'est le *BackUp*.
2. Création d'un clone sur la machine distante par l'intermédiaire du *daemon-ALBA*.
3. Passation au clone du *BackUp* et de la liste des accointances à sauvegarder.
4. Connexion du clone à sa liste d'accointances. Dans chacun des agents de la liste d'accointances, l'adresse est alors automatiquement modifiée et la connexion avec l'agent migrant détruite. Pendant ce temps, l'agent migrant sert à retransmettre les messages en attente et les nouveaux messages à son clone.
5. Destruction de l'agent migrant après réception des notifications de déconnexion de toutes ses accointances et envoi d'un message de terminaison au clone pour qu'il puisse débiter son comportement.

8.1.3 Les primitives d'ALBA v1.0

Pour terminer, nous allons présenter quelques primitives que proposent la bibliothèque ALBA dans sa première version. Ces primitives servent principalement à la création des agents et à leur communication. Les agents sous ALBA v1.0 sont identifiés au sein du système par une carte d'identité, *IDCARD*. C'est par l'intermédiaire de cette dernière qu'il est possible aux agents de communiquer entre eux⁵. Cette *IDCARD* contient : le nom de l'agent, créé à partir du nom fourni et de la concaténation des noms de son ascendance ; l'adresse de la machine hôte ; le numéro de port associé à l'agent pour les communications ; le numéro du processus Prolog correspondant à l'agent ; un identifiant personnel. De plus, le comportement d'un agent programmé à l'aide d'ALBA se doit de débiter par la clause :

```
behavior(MYIDCARD, MYCONTACT, KL)
```

où *MYIDCARD* contient l'*IDCARD* créé pour l'agent, *MYCONTACT* contient la liste des accointances à la création de l'agent, et *KL* contient les données nécessaires au fonctionnement de la couche ALBA⁶.

⁴Ce protocole a été quelque peu modifié dans la version 2.0 au niveau de l'étape. Dorénavant l'agent migrant disparaît dès que le clone a été mis en place sans attendre de notification de la part de ses accointances

⁵Dans la version 2.0, les agents sont identifiés par un nom unique, le concept de *IDCARD* étant passé au niveau de la sous-couche ALBA et donc non-visible au niveau du comportement de l'agent.

⁶Cette variable a été supprimée dans la version 2.0 de ALBA du fait de la modification de la gestion de la mémoire.

Enfin, voici quelques primitives de ALBA v1.0⁷ :

- * `create_agent(+HOST, +CONTACTS, +BEHAVIOR, +WINDOW, +NAME, -IDCARDSON, +KL)`, pour la création d'un nouvel agent sur une machine précisé via la variable `HOST`, ayant une certaine liste de `CONTACTS`, un fichier de comportement précisé dans la variable `BEHAVIOR`, une fenêtre de visualisation et un nom précisé dans la variable `NAME`. Cette clause retourne l'`IDCARD` de ce nouvel agent dans la variable `IDCARDSON`.
- * `wait_creation(+TIMEOUT, +IDCARD, +KL)`, pour permettre à un agent créateur d'attendre la terminaison de la création de l'agent fils pendant une durée `TIMEOUT`. `IDCARD` s'unifie avec `IDCARDSON` dans `create_agent/7`.
- * `send_message(+MESSAGE, +REC_IDCARD, +KL)`, pour l'envoi d'un message `MESSAGE` à l'agent distant défini par `REC_IDCARD`.
- * `read_message(+TIMEOUT, -MESSAGE, -SENDER, +KL)`, pour la réception d'un message `MESSAGE` provenant de l'agent `SENDER`. L'agent peut rester en attente du message pendant une durée `TIMEOUT`.
- * `read_allmessages(+TIMEOUT, -MESSAGES, -SENDERS, +KL)`, pour la réception de tous les messages en attente.
- * `migration(+HOST, +CONTACTS_LIST, +MYBACKUP, +WINDOW, +KL)`, pour la migration de l'agent sur la machine `HOST`.

8.2 Couplage de DynaCROM et de SCAAR

8.2.1 L'application DynaCROM

L'application DynaCROM [Fel06] [FdLBC06], réalisée par C. Felicissimo dans le cadre de son travail de thèse, est une solution pour mettre en place des normes contextuelles dans un système multiagent ouvert. Cette solution fournit une modélisation des normes, une méta-ontologie pour représenter la sémantiques des normes et un moteur d'inférence à base de règles pour personnaliser la composition des normes.

DynaCROM permet donc la modélisation et la représentation de normes contextuelles et permet ainsi d'offrir des informations précises aux agents dans un contexte donné. Elle fournit également un moyen pour les agents de raisonner sur les normes qui leur sont attribuées, et un moyen pour les développeurs d'implémenter des systèmes d'agents normatifs. Cette solution est composée de trois étapes : la modélisation des normes ; la représentation des normes ; la composition des normes.

DynaCROM propose de modéliser les normes d'un système multiagent suivant quatre niveaux d'abstraction : Environnement, Organisation, Rôle et Interactions. Ces différents

⁷Cette version 1.0 comprend au total 1500 lignes de code.

niveaux représentent des contextes de contrôle dont la différence principale réside dans les limites d'application de ce contrôle. Les normes d'environnement sont appliquées à tous les agents situés dans un environnement sous contrôle. Les normes d'organisation sont appliquées à tous les agents dans une organisation sous contrôle. Les normes de rôle sont appliquées à tous les agents jouant un rôle sous contrôle. Enfin, les normes d'interaction sont appliquées à tous les agents impliqués dans une interaction sous contrôle.

Dans DynaCROM, les normes définissent les actions dont l'exécution est soit permise, soit obligatoire, soit interdite pour les agents dans un contexte donné.

DynaCROM utilise des ontologies pour représenter les données et les contextes de contrôle. L'ontologie fournie par DynaCROM définit au départ six concepts : Action, Penalty, Norm, Environnement, Organisation et Role.

Après que l'utilisateur ait classifié et organisé manuellement les normes et après avoir fourni une représentation explicite de ces normes dans une ontologie, DynaCROM va utiliser des règles pour composer automatiquement les normes contextuelles en procédant de la façon suivante :

- * Lecture de l'ontologie pour récupérer les données et les informations à propos de la structure des concepts.
- * Lecture du fichier de règles pour récupérer les informations sur la composition des concepts suivant les règles activées.
- * Inférence d'une nouvelle ontologie basée sur les deux précédentes analyses.

Les règles sont créées suivant la structure de l'ontologie. DynaCROM contient quatre règles prédéfinies dont une de ces règles est la suivante :

```

Rule1 - [ruleForEnvWithOEnvNorms :      (1)
         hasNorm(?Env, ?OEnvNorms)      (2)
         ← hasNorm(?OEnv, ?OEnvNorms), (3)
         belongsTo(?Env, ?OEnv)]         (4)

```

Cette règle exprime le fait qu'un environnement donné verra ses normes composées avec les normes de son environnement propriétaire (c'est-à-dire l'environnement lié par la relation `belongsTo`). Plus précisément, le processus suivant va être exécuté : en (4), on récupère l'environnement propriétaire, `OEnv`, de l'environnement donné, `Env`. En (3), les normes de

l'environnement propriétaire, `0EnvNormes` sont récupérées. Enfin, en (2), les normes de l'environnement propriétaire sont composées avec les normes de l'environnement donné.

Cette approche a été implémentée dans la perspective de pouvoir être mise en place pour n'importe quel système multiagent ouvert avec des agents hétérogènes. Pour ce faire, DynaCROM correspond à une solution JAVA autosuffisante, encapsulée dans un comportement JADE [Co.].

8.2.2 DynaCROM par l'exemple

Pour valider la faisabilité de cette approche, un exemple de système multiagent ouvert basé dans le domaine des organisations multinationales a été utilisé. Pour ce faire, on se place dans un monde où :

- * USA est un environnement qui appartient à North America.
- * Brazil est un environnement qui appartient à South America.
- * Cuba est un environnement qui appartient à Central America.
- * PUCie-Rio et Dellie Brazil sont des organisations localisée dans l'environnement Brazil.
- * Dellie Cuba est une organisation localisée dans l'environnement Cuba.
- * Dellie Brazil et Dellie Cuba sont des branches du siège Dellie, localisé dans l'environnement USA. Toutes les organisation définissent les rôles de vendeur et d'acheteur.

Les normes pouvant être posées dans ce monde sont alors les suivantes :

Exemples de normes d'environnement :

1. Dans l'environnement Central America, si l'adresse de livraison est en dehors d'un de ses environnements, chaque commande expédiée voit son prix obligatoirement augmenté de 15% de taxes.
2. Dans l'environnement Cuba, toutes les négociations sont obligatoirement payées en CUP (Cuban Pesos), la monnaie nationale. Les négociations effectuées en dehors de Cuba voient obligatoirement leurs valeurs converties de CUP vers la monnaie nationale du pays où se situe le vendeur.
- 2bis. Dans l'environnement USA, toutes les négociations sont obligatoirement payées en US\$, la monnaie nationale. Les négociations effectuées en dehors de USA voient obligatoirement leurs valeurs converties de US\$ vers la monnaie nationale du pays où se situe le vendeur.

Exemple de normes d'organisations :

3. Les organisations Dellie sont dans l'obligation de demander au siège le montant des produits pour chaque commande de plus de 100 unités.

Exemples de normes de rôle :

4. Dans l'organisation Dellie Brazil, les vendeurs sont dans l'obligation d'expédier les commandes dans les délais.
5. Dans l'organisation Dellie Cuba, les vendeurs sont dans l'interdiction de proposer plus de 8% de remise.

Exemples de normes d'interaction :

6. Dans l'environnement Dellie Cuba, les acheteurs sont dans l'obligation de faire un premier acompte de 10% pour chaque commande faite avec un vendeur.

Pour cet exemple, l'éditeur Protégé [oM06] est utilisé pour étendre et instancier l'ontologie de DynaCROM, par exemple, par l'ajout des concepts CUSTOMER, SELLER, MAKEADOWNPAYMENT... La plateforme JADE est utilisée, quant-à-elle, pour concevoir les agents et mettre en place le monde. Les agents correspondants aux vendeurs et acheteurs des différentes organisations peuvent migrer d'un environnement à un autre, DynaCROM se charge alors de fournir aux agents les normes qu'ils se doivent, normalement, de respecter dans le contexte où ils se trouvent.

8.2.3 SCAAR comme une solution d'application des normes

Dans la version actuelle de DynaCROM, les normes ne sont pas appliquées aux agents. DynaCROM se charge de tenir informer les agents des normes, mais ils sont tout à fait libre de décider de suivre ou non les normes sans être pénalisés. Nous avons donc tenté d'utiliser le framework SCAAR pour mettre en place la détection des violations des normes fournies par DynaCROM [FCB⁺07].

Nous avons commencé à étudier les moyens pour interfacier les deux approches compte tenu du fait que DynaCROM est implémentée en Java alors que SCAAR l'est en Prolog. DynaCROM étant en mesure, grâce à l'ontologie qu'elle utilise, de fournir les concepts représentant l'application qui vont être utilisés dans les normes, ainsi que les normes du système dans le langage fourni par SCAAR, les étapes normalement faites manuellement par les concepteurs seront effectuées directement par DynaCROM.

Nous avons testé cet interfaçage dans le contexte de l'exemple précédent et à partir du scénario suivant :

- a. Un vendeur de Dellie Brazil a reçu une grosse commande (1500 PC) venant d'un acheteur de PUCie-Rio.
- b. L'organisation Dellie Brazil n'a pas toutes les entités demandées pour assurer la commande. Elle va devoir acheter 500 entités supplémentaires.
- c. Le vendeur de Dellie Brazil (qui devient un acheteur de Dellie) demande au vendeur de Dellie le prix de chaque entité.
- d. Le vendeur de Dellie répond au vendeur de Dellie Brazil le prix de 100US\$ pour chaque entité.
- e. Le vendeur de Dellie Brazil multiplie la valeur pour une entité par 500 et convertit la valeur obtenue en CUP, la monnaie nationale de Cuba. Le prix final correspond à 5000CUP.
- f. Le vendeur de Dellie Brazil (qui devient un acheteur de Dellie Cuba) envoie la commande des 500 entités au vendeur de Dellie Cuba avec un acompte de 500CUP.
- g. Le vendeur de Dellie Cuba envoie alors les produits demandés à l'acheteur de Dellie Brazil.

A partir du point (c), les comportements définis sont soumis à des normes contextuelles. Ces normes sont réécrites par DynaCROM en respectant le langage fourni par SCAAR et sont placées dans le fichier `laws.ap` correspondant à cette application. Tous les concepts utilisés dans ces normes ont eux aussi été précisés dans le fichier `concepts.ap` à partir de l'ontologie utilisée dans DynaCROM.

SCAARNorm1 - (agt :seller)

```
OBLIGED(agt do askPrice with receiver=dellie)
IF(agt be organization with mainOrganization=dellie)
AFTER(agt do receiverOrder with quantity=Q and Q>100)
BEFORE(agt do informPrice).
```

SCAARNorm2 - (agt :seller)

```
FORBIDDEN(agt do answerPrice with currency=C and C#usDollars)
IF(agt be environnement with name=usa).
```

SCAARNorm3 - (agt :customer)

```
OBLIGED(agt do convertPrice with currency=cup)
BEFORE(agt do sendOrder with sellerEnvironnement=cuba).
```



```

SCAARNorm4 - (agt :customer)
  OBLIGED(agt do downPayment with percent=10)
  IF(agt be organization with name=dellieCuba)
  BEFORE(agt do sendOrder).

```

```

SCAARNorm5 - (agt :seller)
  OBLIGED(agt do addTaxes with percent=15)
  IF(agt be environnement with ownerEnvironnement=northAmerica)
  BEFORE(agt do sendProduct with deliverEnvironnement=DE
  and DE#centralAmerica).

```

La loi `SCAARNorm1` représente la norme (3) et régule le comportement décrit en (c). La loi `SCAARNorm2` représente la première partie de la loi (2Bis) que nous avons transformé sous la forme d'une interdiction pour plus de simplicité et permet de réguler le comportement décrit en (d). La loi `SCAARNorm3` représente la seconde partie de la norme (2Bis) et permet de réguler le comportement décrit en (e). La loi `SCAARNorm4` représente la norme (6) et régule le comportement décrit en (f). Enfin, la loi `SCAARNorm5` représente la norme (1) et régule le comportement décrit en (g).

Nous avons implémenté les agents et leurs comportements décrits dans le scénario en Prolog, grâce à la plateforme ALBA v1.0. Nous leur avons fourni un modèle d'agent simple incluant les différentes actions et états exprimées dans le scénario. Puis, à partir des données fournies par DynaCROM, nous avons exécuté le scénario et prévenu DynaCROM des violations de lois. Ainsi, le comportement décrit en (g) ne respecte pas la loi `SCAARNorm5`. En effet, le vendeur de Dellie Cuba a envoyé les produits demandés par l'acheteur de Dellie Brazil alors que ce dernier se situe hors de l'environnement North America et que les 15% de taxes n'ont pas été ajoutés par le vendeur. De ce fait, le vendeur de Dellie Cuba reçoit une information de transgression et lance une stratégie de régulation. Cette stratégie vérifie tout d'abord si le montant de l'acompte envoyé par l'acheteur tient compte de la taxes de 15%. Du fait que l'acompte vaut 500CUP alors qu'il devrait valoir 575CUP, le vendeur va envoyer un message à l'acheteur pour lui préciser qu'il doit ajouter le montant des taxes à son paiement final.

L'utilisation de SCAAR pour détecter la violation de normes fournies par l'application DynaCROM s'est avéré très intéressante. D'une part, nous avons pu exprimer les normes proposées dans le cas d'étude utilisé pour expérimenter DynaCROM et d'autre part, nous avons pu dérouler un scénario détectant la violation d'une loi sur une application concrète. De plus, nous avons pu nous rendre compte des points faibles de notre langage de loi en ce

qui concerne l'expression des propriétés sur les concepts et ainsi, imaginer des améliorations à ce niveau. Enfin, cette première tentative de collaboration entre DynaCROM et SCAAR, nous a permis de mettre en évidence la nécessité de pouvoir ajouter dynamiquement des lois dans les agents. En effet, les lois traduites par DynaCROM nécessitent la description des contextes dans la mesure où les lois sont écrites dans un fichier avant le lancement de l'application. Ici, DynaCROM ne peut jouer pleinement son rôle de fournisseur de normes suivant le contexte dans lequel se trouve l'agent, en particulier au niveau du contrôle. En permettant l'ajout dynamique de loi dans les parties contrôle, DynaCROM pourrait envoyer à la fois les normes aux agents et les lois au générateur pour qu'il informe les parties contrôle des modifications dans l'ensemble des lois à prendre en considération. De ce fait, dans le scénario précédent, la description des lois se verrait être simplifiée de part la suppression de tout ce qui concerne le contexte dans lequel se trouve l'agent.

8.3 Le système multiagent InterloC

8.3.1 Les principes de l'application

L'application InterloC est un système multiagent dont l'objectif principal était de mettre en évidence les résultats obtenus suivants différentes méthodes de détection de cibles en mouvement. A ce jour, InterloC peut être vu comme un simulateur de repérage et de localisation de navires par une patrouille maritime, de telle sorte que les patrouilleurs ne soient pas détectés par les cibles. Pour ce faire, les patrouilleurs utilisent le principe de la localisation passive, c'est-à-dire qu'ils utilisent le signal radar des navires pour effectuer le repérage. Le principe de cette approche consiste à mesurer grâce à un capteur spécifique et à intervalle régulier, correspondant à la période du radar des navires, l'angle que prend ce dernier par rapport au patrouilleur qui tente d'effectuer la localisation. Ces angles sont généralement imprécis du fait des conditions environnementales. De plus, le calcul de la position d'un navire n'est possible que si la vitesse de celui-ci est inférieure de façon significative à celle du patrouilleur. Dans la version actuelle d'InterloC, le calcul de la position des navires est effectué par un algorithme de propagation d'intervalles [Lho93], InterloG [BT93], qui a été développé en Prolog.

8.3.2 L'agentification d'InterloC

La version actuelle d'InterloC est composée des agents suivants :

- * Les agents DETECTOR. Ces agents représentent les différents patrouilleurs mis en jeu dans le système. Ils ont pour fonction principale de localiser les navires qui se situent dans l'environnement.

- * L'agent ENVIRONNEMENT. Cet agent représente l'environnement dans lequel s'effectue la localisation. Il contient les navires à détecter et connaît leur position réelle. C'est lui qui transmet les données des radars des navires pour pouvoir effectuer les localisations.
- * Les agents TREATMENT. Ces agents sont associés aux agents DETECTOR. Leur fonction est d'effectuer les calculs sur les données des radars des navires. Ces agents utilisent le propagateur de contraintes InterloG pour effectuer leurs traitements. Un agent TREATMENT fournit à l'agent DETECTOR qui lui correspond les résultats de ses calculs.
- * L'agent VISUALIZATION. Cet agent se charge de l'affichage des différents agents contenus dans le système et des étapes de calcul des localisations des navires. Son objectif principal est de tenir à jour les affichages suivant les informations qui lui sont fournies par les patrouilleurs et par l'environnement.

8.3.3 L'implémentation d'InterloC

Les agents composant le système InterloC ont été conçus en utilisant le modèle d'agent FDR (pour Fils de Raisonnement), écrit en Prolog. Ce modèle d'agent a été implémenté telle une surcouche de la bibliothèque ALBA v2.0 pour permettre aux agents de maintenir les contextes de leurs conversations et de leurs raisonnements. Avec ce modèle, les agents sont constitués d'un ensemble de fils de raisonnement, chacun étant décrit par un automate à états finis. A chaque fil de raisonnement est associé une mémoire locale contenant le contexte propre à ce fil. L'ensemble des fils de raisonnement d'un agent est associé à une mémoire globale contenant le contexte global de cet ensemble. De plus, ce modèle utilise un aiguilleur qui reçoit les messages de l'agent et qui les transmet, suivant des règles de grammaire, aux fils de raisonnement en attente de ce message. Les règles de grammaire, qui sont des règles de filtrage sur les messages, peuvent tenir compte de l'expéditeur du message, de la forme du message, mais aussi de son contenu. Lorsque l'aiguilleur ne peut déterminer la destination du message qu'il vient de recevoir, ce message est transmis directement à un fil de raisonnement par défaut qui se chargera du devenir de ce message, comme par exemple, la création d'un nouveau fil de raisonnement. Enfin, les fils de raisonnement sont détruits suivant l'évolution du contexte et le déroulement des conversations.

L'implémentation d'InterloC intègre donc ce modèle d'agent pour mettre en place le comportement des différents agents du système, ainsi que l'utilisation de la bibliothèque ALBA v2.0 pour exécuter le système multiagent, gérer les échanges de messages ainsi que les créations d'agents. InterloC utilise également un ensemble de bibliothèques nécessaires à l'implémentation des comportements des différents agents. Enfin, pour permettre la visualisation graphique des agents constituants InterloC et des résultats de la localisation, une

interface Java a été implémentée. Cette interface est connectée à l'agent `VISUALIZATION` par l'intermédiaire d'une primitive de la bibliothèque `ALBA` permettant la connexion d'entités externes au système. L'agent `VISUALIZATION`, se charge alors de gérer les créations et modifications des éléments graphiques via l'entité constituant l'interface graphique Java.

8.3.4 Des lois pour `InterLoc`

Pour expérimenter notre approche de contrôle et notre framework, nous avons essayé de trouver un ensemble de lois, dont les violations seraient significatives d'une potentielle apparition de comportements pouvant entraîner un dysfonctionnement du système. De plus, nous avons comme objectif de déterminer, grâce à ce système, les perturbations, en particulier sur le temps d'exécution, qu'implique la mise en place de notre approche.

Une analyse des problèmes pouvant apparaître au sein d'`InterloC` a été effectuée par Antoine Subiron [Sub05] au cours de son stage de fin d'étude. Ainsi, il s'avère que le fonctionnement d'`InterloC` peut être mis en péril lorsque par exemple :

- * L'agent `VISUALIZATION` se retrouve en surcharge. C'est-à-dire lorsque sa file d'attente de message dépasse un certain plafond (*e.g.* dix messages en attente). En effet, la visualisation des éléments du système devant être continuellement à jour, certains messages se trouvent rapidement désuets. Il est nécessaire, en cas d'un retard de l'agent `VISUALIZATION` dans le traitement de ses messages, que certains de ceux-ci ne soient pas traités, pour que l'affichage corresponde aux dernières données fournies. Lorsqu'une surcharge est détectée, sa stratégie de régulation consiste à supprimer les occurrences inutiles des différents types de messages qu'il reçoit.
- * Un agent `TREATMENT` disparaît du système. Dans ce cas, l'agent `DETECTOR` qui lui est associé ne peut plus recevoir les résultats des calculs et ne peut plus assurer la localisation des navires à sa portée. Pour éviter cette situation, l'agent `DETECTOR`, avant de transmettre une mesure à l'agent `TREATMENT`, doit vérifier l'état de son agent `TREATMENT`. Lorsqu'une supposition de disparition de l'agent `TREATMENT` est détectée, la stratégie de régulation de l'agent `DETECTOR` consiste à créer un nouvel agent `TREATMENT` pour lui permettre d'atteindre son objectif de localisation.

Pour pouvoir mettre en place ces deux lois, ils nous faut, dans un premier temps fournir les concepts et leurs liens vers l'implémentation du modèle `FDR` et des bibliothèques utilisées pour l'application :

- * `concept(messageQueue, feature, [size :value]) - hook(messageQueue, fdr :filtering_strategy/5, [size :length(param(3),SIZE)])`.

```

* concept(verifyAgentTreatment, action, []) - hook(verifyAgentTreatment,
  providers :get_providers/3, []).
* concept(actionFDR, action, [name :value]) - hook(actionFDR, user :fdr/7,
  [name :param(7)]).
* instance(visuAgent, agent, [type :visualization]) - nohook.
* instance(detectAgent, agent, [type :detector]) - nohook.
* instance(measurement, actionFDR, [name :mesure_idle_timeout]) - nohook.

```

Les lois sont alors :

```

(agt :visuAgent)
FORBIDDEN(agt be messageQueue with size=S and S>10).

(agt :detectAgent)
OBLIGED(agt do verifyAgentTreatment)
BEFORE(agt do measurement).

```

Nous avons mis en place ces lois dans une version simplifiée d'InterloC en essayant de préserver au maximum la structure des agents et l'utilisation du modèle des FDR pour nous assurer que nous pourrions l'appliquer, à terme, à la version actuelle de ce système multiagent. L'application de ces lois et la mise en place des concepts, nous a permis, ici aussi, de mettre en évidence l'intérêt de préciser les possibilités d'expression dans les propriétés. De plus, de part l'utilisation d'un modèle d'agent concret, où les actions que peut exécuter effectivement un agent ne sont pas connues par le modèle mais uniquement par le développeur de l'agent, nous avons pu nous assurer de la possibilité de décrire ces actions à l'aide des instances sur les concepts plus généraux, représentatifs du modèle d'agent.

En revanche, il s'avère assez complexe d'obtenir un ensemble de lois conséquent à appliquer à un système une fois que celui-ci a été totalement implémenté, les lois résultant alors de l'observation de mauvais comportements au cours de l'exécution du système. De ce fait, InterloC ne nous a pas permis de juger d'un quelconque ralentissement au niveau du fonctionnement du système après la mise en place des lois. Nous espérons que, une fois que nous aurons trouvé un nombre conséquent de lois à appliquer à ce système, nous pourrions juger des perturbations exactes que notre approche engendre sur son temps d'exécution. Ces perturbations se situent surtout au niveau de l'échange de messages entre les agents et leurs parties contrôle mais aussi entre les parties contrôles elles-mêmes, ainsi que de la surcharge due au processus de la partie contrôle, du fait de nos choix d'implémentation.

Le nombre de messages échangés dépend des événements détectés au sein des agents. Pour chaque action ou état décrit dans une loi est associé l'envoi d'un message entre l'agent et sa partie contrôle dans le cas d'une loi monoagent ou pour le comportement en local des lois multiagents, à condition que l'action ou l'état soit effectivement détecté au sein de l'agent. S'ajoute à cela les messages échangés lors de la détection d'actions ou états potentiellement interdits entre la partie contrôle et le comportement de l'agent pour lui permettre de continuer son comportement ou le prévenir d'une transgression de loi. Enfin, dans une loi multiagent, les échanges de messages entre les parties contrôle sont doublés, puisqu'une partie contrôle déposant le jeton dans une autre partie contrôle attend un accusé de réception de cette dernière pour laisser l'agent continuer son comportement.

Nous pouvons aussi supposer que les ralentissements dus au blocage de l'agent lorsqu'un événement interdit entre en jeu, ou lors de la transmission d'un jeton entre parties contrôle, peuvent être importants lorsque de nombreuses lois sont implantées au sein d'une application.

Conclusion

Ainsi avons-nous pu tester sur des exemples plus concrets, la faisabilité de notre approche et le fonctionnement de notre générateur. Nous nous sommes néanmoins rendu compte de quelques lacunes quant-à l'expression des propriétés sur les concepts et l'ajout dynamique de lois.

Les premiers essais de collaboration entre SCAAR et DynaCROM, nous sont apparus comme très prometteurs et nous espérons, à terme, obtenir une application à part entière, permettant à la fois de fournir aux agents un ensemble de lois, suivant leur contexte d'exécution, et de détecter les violations de ces normes par les agents.

Enfin, le système multiagent InterloC, nous semble être une application significative pour la mise en place de lois dans l'objectif de garantir la cohérence de son fonctionnement. Il nous reste alors à trouver un ensemble de lois consistant, permettant de mettre en avant le fonctionnement du contrôle et son impact réelle sur l'exécution de l'application.

Cinquième partie

Conclusion & Travaux futurs

Conclusion

Les travaux de thèse que nous avons présentés dans ce mémoire se sont orientés principalement vers la recherche d'une approche permettant de garantir le comportement d'un système multiagent. Nous avons pu mettre en avant un problème majeur souvent utilisé pour critiquer l'utilisation du paradigme agent en lieu et place de l'émergence. Cette émergence étant indissociable des systèmes multiagents, de part leur caractère distribuer, nous avons proposés une approche dynamique de contrôle de cette émergence de comportements.

Durant cette thèse, nous avons, dans un premier temps, effectué une étude du concept d'autonomie pour tenter de résoudre un premier problème lié à l'interprétation de ce terme. Cette étude nous a permis d'y voir plus clair en ce qui concerne l'utilisation d'agents autonomes au sein d'un système et les contraintes bénéfiques lors de la considération, au niveau de la conception, que cette vision de l'autonomie engendre. Ainsi, l'autonomie semble effectivement s'orienter vers un moyen d'améliorer la robustesse au sein d'une application.

Ensuite, nous avons pu proposer une approche de contrôle dynamique du comportement des agents que nous voulions générique, c'est-à-dire qu'elle ne devait pas être appliquée à un système multiagent, où à un modèle d'agent particulier. Cette approche, basée sur le concept de lois, consiste à surveiller le comportement des agents, détecter les comportements indésirables par l'intermédiaire des transgressions de lois et à réguler le comportement des agents en conséquence, en cas de violation. Notre approche est mise en place à l'aide d'une méta-architecture que nous fournissons aux agents. Nous voyons notre approche comme un moyen, d'une part, de contrôler l'émergence des comportements indésirables au sein d'un système multiagent, mais également comme un moyen de mettre en place des normes au sein d'un système multiagent normatif. En effet, nous proposons des mécanismes d'autocontrôle qui permettrait aux agents de détecter les violations de normes et d'être pénalisés en conséquence. Si la prise en compte et l'acceptation des normes par des agents est un domaine largement étudié, il nous est apparu difficile de trouver des approches permettant la mise en place de la détection des violations de normes au sein des agents, en dehors du cadre unique des interactions entre les agents. Nous pensons que notre approche propose de ce fait, une méthodologie pour mettre en place ces normes de façon totalement distribuée au sein des agents d'un système.

De plus, il nous semblait indispensable de pouvoir apporter des mécanismes pour aider à la mise en place de lois au sein des agents. De ce fait, nous proposons une approche de génération automatique des agents autocontrôlés. Cette génération consiste à déduire des lois apposées aux agents, d'une part le code de contrôle à insérer au sein du code de

comportement des agents, mais également des réseaux de Petri représentant les lois, pour permettre à la méta-architecture d'effectuer la surveillance du comportement de l'agent. Nous pensons que cette partie automatique est intéressante en ceci qu'elle simplifie le travail du développeur dans la mise en plus du mécanisme de contrôle du comportement des agents qu'il conçoit.

Enfin, nous proposons une implémentation de notre approche de contrôle au sein d'un framework, écrit en Prolog, qui fournit à la fois les outils pour écrire les lois et définir des concepts, mais également une implémentation du générateur d'agents autocontrôlés. Cette implémentation nous a permis de tester notre approche, en particulier la faisabilité de l'écriture de l'ensemble des lois et des concepts associées ainsi que le fonctionnement général de notre générateur. Nous avons pu nous assurer, que restreint à notre langage de lois, la génération des réseaux de Petri se faisait tout à fait simplement et que la détection opérait sans problème. De part les tests que nous avons pu mettre en place sur les applications InterloC et DynaCROM, nous avons pu valider au mieux la détection des violations de lois et la génération. Nous espérons que cette implémentation pourra aider de futurs utilisateurs à mettre en place notre approche dans d'autres langages de programmation que Prolog et aussi appliquer des lois à des systèmes multiagents écrits dans d'autres langages sur d'autres plateformes.

Ces travaux de thèse nous ont permis également de concevoir une première version d'une bibliothèque permettant la conception et l'exécution de systèmes multiagents. La conception et l'implémentation de cette bibliothèque écrite en Prolog, nous a permis de consolider l'idée que la centralisation des systèmes multiagents via une plateforme n'était pas toujours indispensable et qu'il était intéressant de tenter de distribuer un maximum des outils nécessaires à la mise en place d'agents. De plus, son côté bas niveau et générique, nous permet à l'heure actuelle, de concevoir n'importe quel type d'agent, basé sur n'importe quel modèle, assez aisément.

Perspectives

Néanmoins, ces travaux de thèse ne sont pas une fin en soi. Il reste à ce jour quelques regrets qu'ils nous conviendrait de combler à plus ou moins longue échéance. Une des premières choses que nous allons continuer à étudier à la suite de cette thèse est la collaboration en notre framework SCAAR et l'application DynaCROM. Notre objectif est de pouvoir fusionner aux mieux ces deux applications pour obtenir un système capable à la fois de fournir aux agents les lois qui leur sont attribuées, suivant le contexte, dans lequel ils se trouvent, mais également les moyens de détecter les transgressions de lois effectuées par les agents. Cette collaboration nous semble importante, d'une part, car elle nous permettra de

consolider et de tester plus avant encore notre approche, en particulier en ce qui concerne son côté générique, mais également de valider tout à fait les principes de génération et de détection. S'offre alors à nous deux possibilités, soit nous générons du code Java à partir de notre framework en Prolog pour insérer le code de contrôle dans les agents écrits sous JADE, soit nous proposons une version totalement en Java de notre framework. Même si ce dernier point semble le plus couteux en temps, il peut être intéressant de prouver la faisabilité de notre approche et l'utilisation de la génération dans un des langages de programmation les plus couramment utilisés pour l'implémentation des agents.

En parallèle de cette collaboration, nous aimerions approfondir tout ce qui touche aux propriétés posées sur les concepts utilisés dans les lois. Nous voudrions pouvoir garantir que les concepts utilisés sont bien valides, que le format correspond à ce qui avait été décrit. Globalement nous pensons nous focaliser sur tout ce qui touche à la sécurité de la mise en place de notre approche pour garantir l'absence de mauvais fonctionnement. De même, dès que nous aurons une application d'envergure soumise à un ensemble de lois non négligeables, nous souhaitons pouvoir évaluer les pertes en terme de temps d'exécution et de surcharge, suite à l'utilisation des agents autocontrôlés. Enfin, toujours du point de vue de l'implémentation, il nous semble important de réfléchir à la mise en place de l'ajout dynamique de lois. Cette étape est indispensable pour pouvoir faire collaborer SCAAR et DynaCROM, et elle nous semble intéressante pour permettre une meilleure évolutivité du contrôle à appliquer aux agents. L'ajout dynamique de lois nécessite alors de revoir le fonctionnement de la partie contrôle pour y inclure la prise en compte de nouvelles lois au cours de l'exécution de l'agent. Il nous faut voir également ce que cela implique au niveau de la distribution des lois multiagents, même si nous pensons *a priori* que l'ajout dynamique de lois seraient bénéfique pour ce cas précis. Cet ajout engendrera des modifications au niveau de l'implémentation du framework SCAAR en particulier au niveau de son fonctionnement au démarrage de l'application.

Deux autres points qui nous ont fait défaut dans ces travaux de thèse et que nous aimerions aborder ultérieurement sont, d'une part, tout ce qui touche à la régulation des comportements des agents et d'autre part, la gestion des conflits entre les lois. Nous avons vu que notre approche impliquait que les agents puissent réguler leur comportement en cas de détection de transgression de lois, pour se soustraire aux comportements indésirables. Nous avons vu également que nous n'avions pas, à ce jour, trouvé de solution permettant de limiter l'implication des développeurs dans la description de stratégies de régulation. Ce dernier point ne nous satisfait pas et nous aimerions fournir, dans l'idéal, des moyens pour décrire ces stratégies à plus haut niveau ne nécessitant pas d'avoir une connaissance approfondie du code des agents, mais à défaut, nous souhaiterions proposer des aides à la

mise en place de telles stratégies. L'objectif étant que les développeurs et les agents, ne connaissent pas particulièrement les lois auxquels ils sont soumis, notre objectif serait de trouver des moyens permettant de séparer l'implémentation des agents et de la définition des stratégies de régulation. Nous pensons que ce point précis sur la conception et la prise en compte des stratégies de régulation est un sujet à part entière nécessitant des recherches approfondies.

De plus, dans notre approche, nous ne nous attachons pas à savoir si deux lois peuvent être incompatibles et si le comportement d'un agent ne va pas systématiquement violer l'une ou l'autre de ces lois quoiqu'il entreprenne. Il nous semblerait intéressant de approfondir les possibilités de s'assurer qu'un ensemble de lois est consistant, pour diminuer les envois d'informations entre la partie contrôle et la partie comportement de l'agent et pour éviter le lancement perpétuel de stratégies de régulation qui, peut-être, ne pourront jamais résoudre le problème de transgression.

Enfin, toujours à propos de notre approche de contrôle, nous aimerions nous attacher plus profondément au comportement des parties contrôle dans le cadre de la surveillance d'une loi multiagent. Si notre première proposition, exposée dans ce mémoire, nous semble intéressante et apparaît comme satisfaisante pour résoudre de façon distribuée la détection des transgressions de telles lois, il nous semble également nécessaire de voir si nous ne pouvons pas améliorer son fonctionnement. En effet, il nous est apparu, lors de l'écriture de lois, que nous n'étions pas en mesure de poser certaines lois multiagents portant sur une instance d'un agent particulier que l'on aurait défini, non pas par son nom, mais par une caractéristique générale. Par exemple, une loi voulant exprimer qu'il est interdit à un agent de type A d'envoyer un agent de type B, si cet agent précis est dans un état particulier E, n'est pas, à ce jour, gérée correctement par notre approche multiagent à moins que les agents soient identifiés de façon unique par leur nom, directement dans la loi. Il nous manque en quelque sorte, une identification interne aux parties contrôle pour retrouver précisément l'agent mis en jeu dans la loi à un instant donné.

De l'étude de l'autonomie que nous avons pu faire au cours de cette thèse, nous aimerions pouvoir approfondir tout ce qui touche à la conception d'agent autonome. Actuellement, nous formons, avec en outre, Eric Platon et Katia Potiron, un groupe de travail centré sur l'autonomie et sa mise en place au sein des agents. Notre objectif, à terme, est de proposer, soit une architecture d'agent autonome, soit des lignes de conception d'agents autonomes. Dans l'idée que l'autonomie est un moyen de rendre robustes les agents d'un système, nous essayons de trouver les caractéristiques inhérentes aux agents autonomes qui permettraient à la fois d'observer l'autonomie d'un agent, mais aussi de les agents concevoir dans la perspective d'en améliorer la robustesse. Ces recherches sont en cours,

et nous espérons que nous pourrions aboutir prochainement à des résultats concrets. En parallèle, nous essayons de définir tout ce qui pourrait se rapprocher à la dépendance entre les agents et ce que cela implique au niveau de la conception globale du système et de la gestion des pertes de communications entre les agents et de la disparition imprévue d'agent d'un système.

Enfin, Nous avons eu le plaisir de la voir évoluer la plateforme ALBA dans une version 2.0 plus que prometteuse. Elle est encore sujette à étude pour ce qui est de la mise en place de fonctionnalités distribuées, telle que la recherche d'agent au sein d'un système, sans système central. Cette plateforme a été utilisée pour plusieurs applications au sein de Thales et nous espérons qu'elle pourra l'être également à plus grande envergure.

Annexe A

Publications

2005

1. [**RJCIA'05**] Caroline Chopinaud. *Contrôle Dynamique d'Agents Autonomes*. Dans actes des 7èmes rencontres nationales des jeunes chercheurs en intelligence artificielle (RJCIA'05), pages 141-154. Plateforme AFIA. Nice, 1-3 Juin 2005. Edition PUG. Prix du meilleur exposé.
2. [**PROMAS'05**] Caroline Chopinaud, Amal El Fallah Seghrouchni, Patrick Taillibert. *Dynamic self-control of autonomous agents*. In proceedings of the third International Workshop on Programming MultiAgent Systems (PROMAS@AAMAS'05), pages 5-19. Utrecht, The Netherland, July 26, 2005.
3. [**IAT'05**] Caroline Chopinaud, Amal El Fallah Seghrouchni, Patrick Taillibert. *Automatic Generation of Self-Controlled Autonomous Agents*. In proceedings of the IEEE/WIC/ACM, International Conference on Intelligent Agent Technology (IAT'05), pages 755-758. Compiègne University of Technology, France, September 19-22, 2005. ACM Press. papier court
4. [**JFSMA'05**] Caroline Chopinaud, Patrick Taillibert, Amal El Fallah Seghrouchni. *Contrôle de la Conformité des Comportements Individuels d'Agents Cognitifs Autonomes*. Dans les actes des Journées Francophones sur les Systèmes multiagents (JFSMA'05), pages 33-45. Calais, France, 23-25 Novembre 2005. Edition Hermes. Prix du meilleur papier.

2006

1. [**PPROMAS'06**] Caroline Chopinaud, Amal El Fallah-Seghrouchni, Patrick Taillibert. *Dynamic Self-Control of Autonomous Agents*. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, Amal El Fallah-Seghrouchni (Eds.), Programming Multi-Agent

- Systems. Revised and Invited Papers, Lecture Notes in Computer Science, LNAI3862, pages 41-56, Springer, April 28, 2006. PROMAS Post-proceedings, extended and revised version of [PROMAS'05].
2. [AFIA'61] Caroline Chopinaud. *Contrôle Dynamique d'Agents Autonomes*. Dans le bulletin de l'AFIA numéro 61 (Association Française pour l'Intelligence Artificielle), pages 5-6, Avril 2006. Résumé de [RJCIA'05].
 3. [PROMAS'06] Benjamin Devèze, Caroline Chopinaud, Patrick Taillibert. *ALBA : a Generic Library for Programming Mobile Agents with Prolog*. In proceedings of the fourth international Workshop on Programming Multiagents Systems (PROMAS@AAMAS'06). Hakodate, Japan, May, 2006.
 4. [ECAI'06] Caroline Chopinaud, Amal El Fallah Seghrouchni, Patrick Taillibert. *Prevention of Harmful Behaviors within Cognitive and Autonomous Agents*. In proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06), pages 205-209, Riva del Garda, Italy, Aug 28th - Sept 1st, 2006. IOS Press.

2007

1. [PROMAS'07] Benjamin Devèze, Caroline Chopinaud, Patrick Taillibert. *ALBA : a Generic Library for Programming Mobile Agents with Prolog*. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, Amal El Fallah-Seghrouchni (Eds.), *Programming Multi-Agent Systems. Revised and Invited Papers, Lecture Notes in Computer Science, LNAI4411*, Springer, May, 2007. PROMAS Post-proceedings, extended and revised version of [PROMAS'06].
2. [AOIS'07] Carolina Felicissimo, Ricardo Choren, Jean-Pierre Briot, Carlos Lucena, Caroline Chopinaud, Amal El Fallah Seghrouchni. *Providing Contextual Norm Information in Open Multi-Agent Systems*. In AOIS Post-Proceedings. A paraître.

Annexe B

Justification

Nous souhaitons montrer que toute expression valide dans le langage proposé au chapitre 6 peut être représentée par un réseau de Petri équivalent.

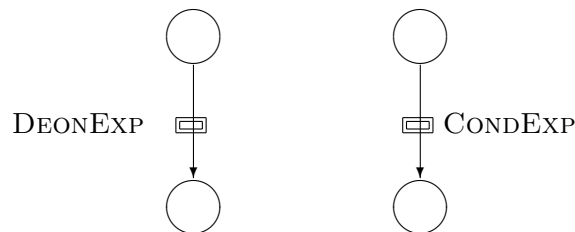
Pour ce faire nous allons utiliser les réseaux de Petri récursifs (RdPR), en représentant chaque expression élémentaire de notre langage par un réseau de Petri récursif ayant des transitions abstraites dépliables, des transition élémentaires et des transitions de fin. Pour la définition formelle d'un RdPR, on peut se référer à [SH96] et [BSH⁺98]. Nous avons cependant introduit les arcs inhibiteurs pour simplifier la modélisation.

Soient Exp_1 , Exp_2 , deux expressions générales de la forme :

$$Exp_1 \equiv \text{DEONEXP}$$

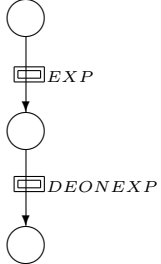
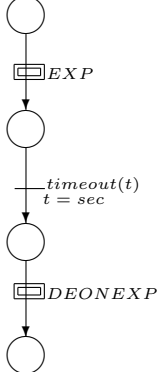
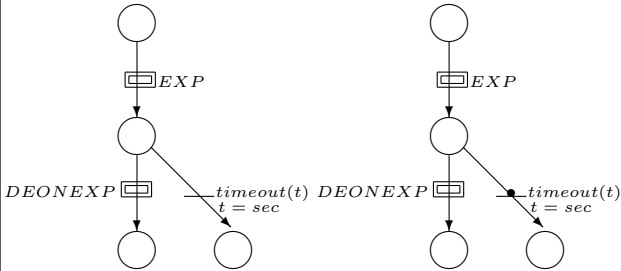
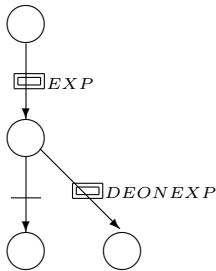
$$Exp_2 \equiv \text{CONDEXP}$$

Les expression Exp_1 et Exp_2 , déjà formulées en forme normale, sont représentées chacune par un réseau de Petri récursif de la forme :

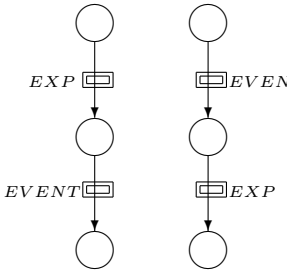
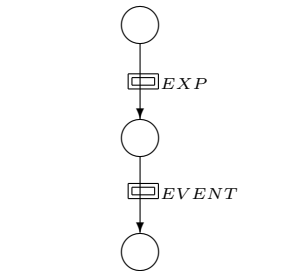
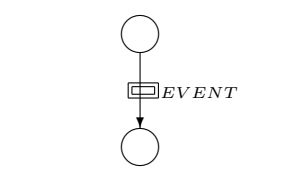


Pour générer le réseau de Petri associé à la loi, toutes les transitions abstraites sont à

déplier. Les dépliages possibles pour la transition abstraite CONDEXP sont décrits dans le tableau suivant :

Expression du langage	Réseau de Petri récursif
<p>DEONEXP AFTER EXP Après une expression, les expression déontiques à vérifier</p>	
<p>DEONEXP AFTER EXP + sec Plus de sec secondes après avoir fait EXP, les expressions déontiques doivent être vérifiées</p>	
<p>DEONEXP AFTER EXP - Sec Dans les sec secondes après avoir fait EXP, les expressions déontiques doivent être vérifiées. Les deux réseaux représentent respectivement le cas d'une interdiction et d'une obligation</p>	
<p>DEONEXP IF EXP Si EXP est vérifiée alors les expressions déontiques doivent être vérifiées</p>	

Les dépliages possibles pour la transition abstraite EXP sont décrits dans le tableau suivant :

Expression du langage	Réseau de Petri récursif
<p><i>EXP AND EVENT</i> Commutativité de séquences d'événements</p>	
<p><i>EXP THEN EVENT</i> Séquence d'événements</p>	
<p><i>EVENT</i> Un événement</p>	

Enfin les dépliages possibles pour la transition abstraite DEONEXP sont donnés dans le tableau suivant :

Expression du langage	Réseau de Petri récursif à répercuter
FORBIDDEN $EVENT$ Un événement interdit	
OBLIGED $EVENT_1$ BEFORE $EVENT_2$ Une obligation avec délai	
FORBIDDEN $EVENT_1$ BEFORE $EVENT_2$ Une interdiction avec délai	
EXPDEON BEFORE $EVENT$ Une expression déontique avec délai	

De plus sachant que notre langage est basé sur la logique déontique dynamique nous obtenons les transformations suivantes :

- * **FORBIDDEN EXP THEN EVENT \equiv EXP THEN FORBIDDEN EVENT**
- * **FORBIDDEN EXP AND EVENT \equiv EXP AND FORBIDDEN EVENT**
- * **OBLIGED EXP THEN EVENT BEFORE EVENT \equiv OBLIGED EXP BEFORE EVENT THEN OBLIGED EVENT BEFORE EVENT**
- * **OBLIGED EXP AND EVENT BEFORE EVENT \equiv OBLIGED EXP BEFORE EVENT AND OBLIGED EVENT BEFORE EVENT**
- * **EXPDEON BEFORE EXP THEN EVENT \equiv EXPDEON BEFORE EXP THEN EXPDEON BEFORE EVENT**
- * **EXPDEON BEFORE EXP AND EVENT \equiv EXPDEON BEFORE EXP AND EXPDEON BEFORE EVENT**

Nous obtenons alors des formules dont le réseau de Petri récursif correspondant est identifiable à l'aide de l'ensemble des tableaux précédemment présentés.

Reste à présent à énumérer les dépliages possibles pour la transition abstraite *EVENT* :

Expression du langage	Réseau de Petri récursif
agent do <i>SMTH</i> L'exécution d'une action	
agent be <i>SMST</i> L'arrivée dans un état	

Nous avons donc mis en avant que toute expression élémentaire de notre langage est traduisible en un réseau de Petri récursif. Il reste alors à prouver que toute expression (générale) est modélisée par transition abstraite et est dépliable en un réseau de Petri récursif terminal. Ceci se prouve facilement par construction, la composition étant possible du fait de la construction même de l'expression.

Bibliographie

- [Ada95] G. Adant. PrÃ©ambule Ã une reflexion autour des concepts d'indÃ©pendance et d'autonomie. *Journal d'ErgothÃ©rapie*, 17(3) :83–87, 1995.
- [AZ97] S.M. Ali and R.M. Zimmer. The question concerning emergence. In *International Conference On Computing Anticipatory Systems*, 97.
- [Bai91] Patrice Bailhache. *Essai de Logique d'Ã©ontique*. Mathesis, vrin edition, 1991.
- [BDM⁺98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos : A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, volume 1427, pages 546–550. Springer-Verlag, 1998.
- [Bec99] K. Beck. *Extreme Programming Explained : Embrace Change*. 1999.
- [BFWV04] R.H. Bordini, M. Fisher, M. Wooldridge, and W. Visser. Model checking rational agent. In *IEEEIS'04*, 2004.
- [BH01] S. Brainov and H. Hexmoor. Quantifying Relative Autonomy in Multi-Agent Interaction. In *Proc. of the Workshop on Autonomy, Delegation and Control, IJCAI' 01*, 2001.
- [BLB06] T. Bosse, D.N. Lam, and K.S. Barber. Automated analysis and verification of agent behavior. In P. Stone and G. Weiss, editors, *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 1317–1319. ACM Press, 2006.
- [BM99] K.S. Barber and C.E. Martin. Agent autonomy : Specification, measurement and dynamic adjustment. In *Inproceedings of the Autonomy Control Software Workshop at Autonomous Agents, Agents' 99*, pages 8–15, Seattle, WA, 1 Mai 1999.
- [Bro91] R. Brooks. Intelligence without reason. In Morgan-Kauffman, editor, *Proceedings of IJCAI'91*, pages 569–595, Sydney, Australia, 1991.

- [BSH⁺98] S. Boussetta, A. El Fallah Seghrouchni, S. Haddad, P. Moraitis, and M. Taghelit. Coordination d'agents rationnels par planification distribuée. *Revue d'Intelligence Artificielle*, 1998.
- [BT93] B. Botella and P. Taillibert. Interolog : Constraint logic programming on numeric intervals. In *Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics*, 1993.
- [Bur93] H.D. Burkhard. Liveness and fairness properties in multi-agent systems. In *International Joint Conference on Artificial Intelligence*. IJCAI'93, 1993.
- [CBF03] C. Carabelea, O. Boissier, and A. Florea. Autonomy in Multi-Agent Systems : A Classification Attempt. In *Proc. of the Workshop on Computational Autonomy : Potential, Risks and Solutions*, Second International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS'03, Melbourne, Australia, July 2003. Inprint.
- [CCD99] R. Conte, C. Castelfranchi, and F. Dignum. Autonomous norm-acceptance. In *Intelligent Agents V*, number 1555 in LNAI. Springer Verlag, 1999.
- [CCHW] A. Colyer, A. Clement, G. Harley, and M. Webster. *Aspect-Oriented Programming in Eclipse with AspectJ and the AspectJ development tools*. The Eclipse Series.
- [CDJT99] C. Castelfranchi, F. Dignum, C. Jonker, and J. Treur. Deliberative normative agents : Principles and architectures. In *ATAL'99*, Orlando, 1999. AAAI Press.
- [CF98] C. Castelfranchi and R. Falcone. Towards a Theory of Delegation for Agent-based Systems. In Elsevier, editor, *Robotics and Autonomous Systems*, volume 24, Special Issue on MultiAgent Rationality, pages 141–157, 1998.
- [CF03] C. Castelfranchi and R. Falcone. From Automaticity to Autonomy : The Frontier of Artificial Agent. In Henry Hexmoor, Cristiano Castelfranchi, and Rino Falcone, editors, *Agent Autonomy*, volume 7 of *MultiAgent Systems, Artificial Society and Simulated Organizations*, chapter 6, pages 79–104. March 2003.
- [CFNF97] D. Cohen, M.S. Feather, K. Narayanaswamy, and S. Fickas. Automatic monitoring of software requirements. In *Proceedings of ICSE'97*, 1997.
- [CKvSL06] R. Coelho, V. Kulesza, A. von Staa, and C. Lucena. Unit testing in multiagent systems using mock agents and aspects. In *In proc. of 5th international workshop on software engineering for large scale multiagent systems (SELMAS)*, China, 2006.
- [Co.] Tilab Co. Jade - java agent development framework.

- [CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. memory efficient algorithms for the verification of temporal properties. In *Forma Methods in System Design*, volume I, pages 275–288, 1992.
- [DCT06] B. Deveze, C. Chopinaud, and P. Taillibert. Alba : A generic library for programming mobile agents with prolog. In *PROMAS'06 workshop at AAMAS'06*, Hakodate, Japan, May 2006.
- [DDMvL97] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. Grailkaos : An environment for goal-driven requirements engineering. In *Proc. ICSE'97 - 19th Int. Conference on Software Engineering*, pages 612–613, Boston, 1997.
- [Dem95] Y. Demazeau. From interactions to collective behaviour in agent-based systems. In *In Proc. of the first European Conference on Cognitive Science*, pages 117–132, April 1995.
- [Den87] D. Denett. The intentional stance. *Bradford/MIT Press*, 1987.
- [DJC94] M. Diaz, G. Juanole, and J-P. Courtiat. Observer-a concept for formal on-line validation of distributed systems. *IEEE Trans. Softw. Eng.*, 20(12) :900–913, 1994.
- [dL96] M. d’Inverno and M. Luck. Understanding Autonomous Interaction. In *Proceedings of the 12th European Conference on Artificial Intelligence, ECAI'96*. John Wiley and Sons, Ltd, 1996.
- [dSDR02] M. de Sousa Dias and D.J. Richardson. Issues on software monitoring. Technical report, Department of Information and Computer Science, University of California, July 2002.
- [FA98] G. Ferguson and J. Allen. Trips : An intelligent intergrated problemsolving assistant. In *Proceedings of Autonomous Agents and Artificial Intelligence (AAAI-98)*, pages 567–573, Madison, Wisconsin, 1998.
- [FC01] R. Falcone and C. Castelfranchi. Tuning the Collaboration Level with Autonomous Agents : A Principled Theory. In Floriana Esposito, editor, *AI*IA 2001 : Advance in AI 7th congress of the Italian Association for AI*, volume 2175 of *Lecture Notes in Computer Science*, pages 212–224, Bari, Italy, September 2001. Springer.
- [FC02] R. Falcone and C. Castelfranchi. Issues of Trust and Control on Agent Autonomy. *Connection Science*, 14(4) :249–264, 2002.
- [FCB⁺07] C. Felicissimo, R. Choren, J.P. Briot, C. Lucena, C. Chopinaud, and A. El Fallah Seghrouchni. *AOIS Post Proceedings*, chapter Providing Contextuel Norm Information in Open Multi-Agent systems. to appear, 2007.

- [FdLBC06] C. Felicissimo, C. de Lucena, J.P. Briot, and R. Choren. Regulating open multiagent systems with dynacrom. In *Second Workshop on Software Engineering for Agent-Oriented Systems*, Brazil, 2006.
- [Fel06] C. Felicissimo. Dynamic contextual regulations in open multiagent systems. In *International Semantic Web Conference*, volume LNCS 4273, pages 974–975, 2006.
- [Fer97] J. Ferber. *Les Systèmes Multi-Agents : Vers une Intelligence Collective*. InterEditions, Paris, 1997.
- [FFvLP98] M.S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling System Requirements and Runtime Behavior. In *Proceedings of IWSSD9*, Isobe, Japan, 1998.
- [FP02] E. Fabre and V. Pigourin. Monitoring distributed systems with distributed algorithms. In *In proc. of the 41st IEEE Conference on Decision and Control*, pages 411–416, 2002.
- [Fre20] *Essais de Psychanalyse*, chapter Au-delà du Principe de Plaisir. PBP, 1920.
- [Fre23] *Essais de Psychanalyse*, chapter Le Moi et le Ca. PBP, 1923.
- [FSGW96] D. Fensel, A. Schonegge, R. Groenboom, and B. Wielinga. Specification and verification of knowledge-based systems. In *Proceedings of the Workshop on Validation, Verification and Refinement of Knowledge-Based Systems, 12th European Conference on Artificial Intelligence (ECAI-96)*, 1996.
- [Gre97] D. Grevier. *A la recherche de l'intelligence artificielle*. 1997.
- [Hag96] S. Hagg. A sentinel approach to fault handling in multi-agent systems. In *In Proceedings of the Second Australian Workshop on Distributed AI, in conjunction with the Fourth Pacific Rim International Conference on Artificial Intelligence (PRICAI'96)*, 1996.
- [HdBvM99] K. Hindriks, F. de Boer, W. van der Hoek, and J.J.Ch. Meyer. Agent programming in 3apl. In *Autonomous Agents and Multi-Agent Systems*, pages 357–401, 1999.
- [HK95] Y. Huang and C. Kintala. Software fault tolerance in the application layer. In *Software Fault Tolerance*, 1995.
- [Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5) :279–295, 1997.
- [Jou05] C. Joubert. *Vérification à la volée de grands espaces d'états*. PhD thesis, Institut National Polytechnique de Grenoble, 2005.

- [JS93] A.J.I. Jones and M. Sergot. On the characterisation of law and computer systems : The normative systems perspective. In J.J.Ch. Meyer et R.J. Wieringa, editor, *Deontic Logic in Computer Science : Normative System Specification*, 1993.
- [JSW98] N. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. In *Autonomous Agents and Multiagent Systems*, pages 7–38, 1998.
- [Kan88] E. Kant. *Critique de la raison pratique*. 1788.
- [Kle99] M. Klein. Exception handling in agent systems. In *Proceedings of the Third International Conference on Autonomous Agents*, Seattle, Washington, 1999.
- [KS02] M.S. Kerstetter and S.D.G. Smith. Adjustable Autonomy for Human and Information System Interaction. In *Proc. of the Workshop on Autonomy, Delegation and Control : From Inter-Agent to Groups*, AAI'02, Edmonton, Alberta, Canada, 28 Juillet 2002.
- [Lap] S. Lapierre. La conception freudienne de l'homme. www.colvir.net/prof/serge.lapierre/Freud.html.
- [Lar98] K.G. Larsen. Formal methods for real time systems : Automatic verification and validation. In *ARTES summer school*, 1998.
- [LB04] D.N. Lam and K.S. Barber. Debugging agent behavior in an implemented agent system. In *Proceedings of PROMAS'04*, pages 45–56, New York City, July 20 2004.
- [LC92] Y. Liao and D. Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Trans. Software Engineering*, 18(11), November 1992.
- [LCSM90] J.E. Lumpp, T.L. Casavant, H.J. Siegle, and D.C. Marinescu. Specification and identification of events for debugging and performance monitoring of distributed multiprocessor systems. In *Proceedings of the 10th International Conference on Distributed Systems*, pages 476–483, June 1990.
- [Ld00] M. Luck and M. d'Inverno. Autonomy : A Nice Idea in Theory. In *Proceedings of the ATAL'00, Agent Theory, Architecture and Languages*, pages 351–353, 2000.
- [Leg03] F. Legras. *Organisation dynamique d'équipes d'engins autonomes par écoute flottante*. PhD thesis, Université de Toulouse, 2003.
- [Lho93] O. Lhomme. Consistency techniques for numeric csps. In *IJCAI'93*, 1993.
- [LLd02] F. Lopez y Lopez, M. Luck, and M. d'Inverno. Constraining Autonomy through Norms. In *Proc. of the First International Joint Conference on*

- Autonomous Agents and Multiagents Systems*, AAMAS'02, pages 674–681, Bologna, Italy, 2002. ACM Press.
- [MCWB91] K. Marzillo, R. Cooper, M.D. Wood, and K.P. Birman. Tools for distribution application management. *Cornell University, IEEE Computer*, pages 42–51, 1991.
- [Mer74] P.M. Merlin. *A Study of the recoverability of computing systems*. PhD thesis, Departement of Information and Computer Science, University of California, Irvine, 1974.
- [Mey88] JJCH. Meyer. A different approach to deontic logic : deontic logic viewed as a variant of dynamic logic. *Notre dame journal of formal logic*, 29(1) :109–136, Winter 1988.
- [MFC00] T. Mackinnon, S. Freeman, and P. Craig. Endotesting : Unit testing with mock objects. In *Proc. XP2000*, 2000.
- [MS95] M. Mansouri-Samani. *Monitoring of Distributed Sytems*. PhD thesis, University of London, London, UK, 1995.
- [MSSP02] D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat. Program instrumentation for debugging and monitoring with AspectC++. In *Proc. of the 5th IEEE International symposium on Object-Oriented Real-time Distributed Computing*, Washington DC, USA, April 29 – May 1 2002.
- [MU00] N. Minsky and V. Ungureanu. Law-governed interaction : A coordination and control mechanism for heterogeneous distributed systems. In *ACM Transactions on Software Engineering and Methodology*, volume 9, pages 237–305, 2000.
- [oCS] Swedish Institute of Computer Science. Scistus prolog user's manual.
- [oM06] Standford University School of Medecine. Protege, a free open source ontology editor and knowledge base framework, 2006.
- [OSR92] S. Owreand, N. Shankar, and J. Rushby. Pvs : A prototype verification system. In *CADE11*, Saratoga Springs, NY, 1992.
- [PgCcL⁺05] Rodrigo Paes, gustavo Carvalho, carlos Lucena, Paulo Alencar, Hyggo Almeida, and Viviane Silva. Specifying laws in open multi-agent systems. In *Agents, Norms and Institutions for Regulated Multiagent Systems - ANIREM*, Utrecht, July 2005.
- [Pnu77] A. Pnueli. The temporal logic of programs. In *In Proc 18th IEEE Symp. Foundations of Computer Science*, pages 46–57, Providence, 1977.
- [Pot06] K. Potiron. Maintenance en ligne d'agents autonomes. Technical report, Universit   de Paris XIII, Institut Galil  e, 2006.

- [PSH07] E. Platon, N. Sabouret, and S. Honiden. An architecture for exception management in multi-agent systems. *Int. J. Agent-Oriented Software Engineering*, 2007.
- [PV97] V.D. Parunak and R.S. VanderBok. Managing emergent behavior in distributed control systems. In *ISA-Tech'97*, Anaheim, 1997.
- [Rao96] A. Rao. Agentspeak(1) : Bdi agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *Proceedings of 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAA-MAW'96)*, volume 1038, pages 42–55. Springer-Verlag, 1996.
- [RG95] A. Rao and M.P. Georgeff. Bdi-agents : from theory to practice. In *In Proceedings of the First International Conference on Multiagent Systems (ICMAS '95)*, pages 312–319, San Francisco, CA, USA, 1995.
- [Rus95] *Artificial Intelligence : A Modern Approach*. Prentice Hall, 1995.
- [Sch99] P. Schnoebelen. *Vérification de Logiciels : Techniques et Outils du Model-Checking*. Vuibert edition, 1999.
- [Sch02] M. Schillo. Self-Organization and Adjustable Autonomy : Two Sides of the Same Medal. In *Proc. of the Workshop on Autonomy, Delegation and Control : From Inter-Agent to Groups, AAAI' 02*, Edmonton, Alberta, Canada, 28 Juillet 2002.
- [SFS03] M. Schillo, K. Fischer, and J. Siekmann. The Link between Autonomy and Organisation in Multi-Agent Systems. In *Proc. of the First International Conference on Applications of Holonic and Multi-Agent Systems, Ho-loMAS' 03*, 2003.
- [SH96] A. El Fallah Seghrouchni and S. Haddad. A recursive model for distributed planning. In *Second International Conference on Multi-Agent Systems.*, Kyoto, Japon, 1996.
- [Sho93] Y. Shoham. Agent oriented programming. *Artificial Intelligence*, 60 :51–92, 1993.
- [Sos92] Rok Susic. *The Many Faces of Introspection*. PhD thesis, University of Utah, June 1992.
- [SPI97] The model checker spin. In *IEEE Trans. on Software Engineering*, volume 23, pages 279–295, 1997.
- [SS94] L. Sterling and E. Shapiro. *The art of Prolog, Second edition : Advanced Programming Techniques*. MIT Press, 1994.
- [SS04a] A. El Fallah Seghrouchni and A. Suna. Claim : A computational language for autonomous, intelligent and mobile agents. In M. Dastani, J.Dix, and A. El

- Fallah Seghrouchni, editors, *Proceedings of the First International Workshop on Programming Multiagent Systems : Languages and Tools*, number 3067 in Lecture Notes in Artificielle Intelligence, pages 90–110. Springer-Verlag, 2004.
- [SS04b] A. Suna and A. El Fallah Seghrouchni. A mobile agents platform : architecture, mobility and security elements. In *Proceedings of the Second International Workshop on Programming Multi-Agent Systems (ProMAS'04)*, New-York, 2004.
- [Str02] T. Stratulat. *Systèmes d'agents normatifs : concepts et outils logiques*. PhD thesis, Université de Caen, December 2002.
- [Sub05] A. Subiron. Robustesse des systèmes multiagents : Application au système de localisation interloc. Master's thesis, Université de Paris 6, 2005.
- [Tra72] R.E. Trajan. Deoth first search and linear graph algorithms. In *SIAM J. Computing*, volume 1, pages 146–160, 1972.
- [TSP02] M. Tambe, P. Scerri, and D.V. Pynadath. Adjustable autonomy : From theory to implementation. In *Proc. of the Workshop on Autonomy, Delegation and Control : From Inter-Agent to Groups*, AAAI' 02, Edmonto, Alberta, Canada, 28 Juillet 2002.
- [VB99] H. Verhagen and M. Boman. Adjustable Autonomy, Norms and Pronoucers. *AAAI Spring Symposium on Agents with Ajustable Autonomy*, 1999.
- [VHB⁺03a] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. In *Automated Software Engineering Journal*, volume 10, 2003.
- [VHB⁺03b] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.
- [VSAD04a] J. Vázquez-Salceda, H. Alderweld, and F. Dignum. Implementing norms in multiagent systems. In *Proceedings of MATES'04*, Erfurt, Germany, September, 29–30 2004.
- [VSAD04b] J. Vázquez-Salceda, H. Alderweld, and F. Dignum. Normes in multiagent systems : some implementation guidelines. In *Proceedings of EUMAS'04*, Barcelona, Spain, December, 16–17 2004.
- [vW51] G.H. von Wright. Deontic logic. *Mind*, 60(237) :1–15, 1951.
- [Wala] C. Walton. Model checking agent dialogues.
- [Walb] C. Walton. Multi-agent dialogue protocols.
- [Wam03] Dean Wampler. The future of aspect oriented programming, 2003. White Paper, available on <http://www.aspectprogramming.com>.

-
- [WFHP02] M. Wooldridge, M. Fisher, M.P. Huget, and S. Parsons. Model checking multi-agent systems with mable. In *Proceedings of AAMAS'02*, Bologna, Italy, July 15–19 2002.
- [WH04] T. De Wolf and T Holvoet. Emergence as a general architecture for distributed autonomic computing. Technical Report Report CW384, 2004.
- [Woo00] M. Wooldridge. Reasoning about rational agents. 2000.
- [Woo02] M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley and Sons, 2002.
- [WRR95] D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a concurrent agent-oriented language. In N. Jennings and M. Wooldridge, editors, *Intelligent Agents : Theories, Architectures, and Languages*, pages 386–402. Springer-Verlag, 1995.