La notion d'intentionnalité dans la spécification de comportements d'agents autonomes situés en environnement virtuel

Pierre-Alexandre Favier

▶ **To cite this version:**

Pierre-Alexandre Favier. La notion d'intentionnalité dans la spécification de comportements d'agents autonomes situés en environnement virtuel. Autre [cs.OH]. Université de Bretagne occidentale - Brest, 2004. Français. NNT : . tel-00008452

HAL Id: tel-00008452
https://theses.hal.science/tel-00008452

Submitted on 11 Feb 2005

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

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

La notion d'intentionnalité dans la spécification de comportements d'agents autonomes situés en environnement virtuel.

Thèse
présentée devant

l'Université de Bretagne Occidentale

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE
Mention Informatique

par
Pierre-Alexandre FAVIER

Equipe d'accueil 3883 (UBO, ENIB)
Laboratoire d'Ingénierie Informatique (LI2/ENIB)

Soutenue le 14 décembre 2004 devant la commission d'examen :

Dr. Pierre DE LOOR	Maître de conférences, ENIB, Brest	*(Examinateur)*
Pr. Jean JOUQUAN	Professeur au CHU La Cavale Blanche, Brest	*(Président)*
Pr. Philippe MATHIEU	Professeur à l'Université de Lille 1	*(Rapporteur)*
Dr. Jean-Pierre MÜLLER	Directeur de recherche au CIRAD, Montpellier	*(Rapporteur)*
Dr. Laurent NANA TCHAMNDA	Maître de conférences à l'UBO, Brest	*(Examinateur)*
Pr. Jacques TISSEAU	Professeur à l'ENIB, Brest	*(Directeur)*

Remerciements

Je tiens à remercier de nombreuses personnes, et certaines à plusieurs titres. Je m'excuse de ne pourvoir citer tout le monde, mais je sais ce que je dois à chacun. Je remercie particulièrement :

PHILIPPE MATHIEU et JEAN-PIERRE MÜLLER pour avoir accepté d'être les rapporteurs de ma thèse.
LAURENT NANA TCHAMNDA pour avoir accepté d'en être l'examinateur et JEAN JOUQUAN qui a assuré la présidence du jury.

JACQUES TISSEAU et PIERRE DE LOOR pour les rapports scientifiques et humains que nous avons entretenus toutes ces années, ma gratitude est à la mesure de ce qu'ils ont fait pour moi.

L'ensemble des membres du *LI2* et du *CERV* pour nos discussions aussi bien scientifiques qu'anodines : elles ont toutes participé à me faire avancer.

Les membres des groupes de travail auxquels je participe (*ASA* et *MFI*) et les membres de la communauté que j'ai eu l'occasion de cotoyer lors de conférences pour nos rencontres agréables et nos conversation intéressantes.

La Communauté Urbaine de Brest qui a en partie financé mes travaux.

Les doctorants du *LI2* qui m'ont supporté, dans tous les sens du terme, particulièrement CÉDRIC BOIDIN : nos nombreuses discussions ont souvent été l'occasion pour moi de confronter mes idées ou de les formuler pour la première fois. Cet "écho" à ma démarche a été très précieux et je suis sûr qu'il reconnaîtra dans ce document la trace de certaines de ses idées. J'espère que cet apport a été réciproque.

Les membres des personnels de l'*ENIB* et du *CERV* : leur travail "dans l'ombre" nous facilite la vie au quotidien. Sans eux les trames sur le réseau ou les publications ne trouveraient pas leur chemin, ou de bêtes problèmes d'emploi du temps prendraient le pas sur ce qui est réellement important.

Les élèves de l'*ENIB* : les rapports que nous avons eu lorsque j'étais élève-ingénieur comme ceux que nous avons maintenant que j'enseigne occupe une place importante pour

moi.

Enfin, et plus que tous, je tiens à remercier mes proches : sans leur soutien indéfectible les violentes remises en cause que j'ai vécu auraient peut-être eu raison de ma motivation. J'imagine à quel point l'omniprésence de mes absences a été difficile durant ces années.

PIERRE-ALEXANDRE FAVIER

Table des matières

1 Introduction **13**

2 Agents autonomes et réalité virtuelle **23**
2.1 L'approche multi-agents 24
2.1.1 Le paradigme agent 24
2.1.2 Du problème à l'analyse : l'agent unité conceptuelle 27
2.1.3 De l'analyse à la conception : l'agent unité computationnelle 29
2.1.4 De la conception au développement : l'agent unité d'implémentation 31
2.1.5 Du développement au déploiement : l'agent unité d'exécution . . . 33
2.2 Agents autonomes situés et réalité virtuelle 35
2.2.1 La réalité virtuelle 35
2.2.2 L'émergence de comportement 36
2.2.3 La situation de l'agent : une notion subtile 38
2.2.4 L'autonomie de l'agent : un but ou un moyen ? 39
2.3 Agents autonomes situés et comportements 41
2.3.1 Aspect externe de l'agent : les comportements émergents 41
2.3.2 Aspect interne de l'agent : les modèles comportementaux 42
2.4 Agents autonomes situés et spécification de comportement 52
2.4.1 La spécification : localité *versus* globalité 52
2.4.2 Le syndrome de Frankenstein : contrôle *versus* autonomie 53
2.5 Conclusion : explicitation du *sens* de l'action 55

3 Intentionnalité et agent **57**
3.1 Introduction 58
3.2 L'intentionnalité : un élément du modèle comportemental 58
3.2.1 L'intentionnalité et l'action 59
3.2.2 L'intentionnalité pour la prise de décision 62
3.2.3 L'intentionnalité dans les interactions 66
3.3 L'intentionnalité : un outil pour la conception 68
3.4 Conclusion : l'intentionnalité dans un modèle comportemental 69

4 Le modèle d'agent intentionnel **73**
4.1 Les bases du modèle 74
4.1.1 Les objectifs 74
4.1.2 Une approche duale 76

4.1.3 Définition intensive et définition extensive . . . 78
4.1.4 Dualité impératif / déclaratif . . . 79
4.1.5 De l'acte perceptif à la perception : la construction de l'extériorité . 81
4.1.6 Les principes du modèle . . . 83
4.2 Les 7 composantes du modèle . . . 84
4.2.1 Les méthodes . . . 86
4.2.2 Les attributs . . . 87
4.2.3 Les perceptions . . . 88
4.2.4 Les propriétés . . . 89
4.2.5 Les règles comportementales . . . 90
4.2.6 Les descriptions d'action . . . 92
4.2.7 Les connaissances . . . 94
4.3 Le principe de sélection . . . 95
4.3.1 Exemple de calcul . . . 98
4.4 Conclusion : un modèle et une démarche . . . 100

5 Le modèle d'exécution 103
5.1 Programmation multi-paradigmes . . . 104
5.2 Ordonnancement d'activités hétérogènes . . . 110
5.2.1 ARéVi . . . 110
5.2.2 SWI-Prolog : *thread* et modules . . . 112
5.2.3 Intégration ARéVi - SWI-Prolog . . . 117
5.3 Principes du modèle d'exécution . . . 118
5.3.1 Répartition de la définition d'un agent . . . 118
5.3.2 Le déroulement de la simulation . . . 125
5.3.3 Le prototypage interactif . . . 127
5.4 Conclusion : nécessité d'un outil dédié . . . 133

6 Application : SMAIN 137
6.1 Architecture logicielle . . . 138
6.2 Les services offerts . . . 138
6.2.1 Les services communs aux classes et aux instances . . . 139
6.2.2 Les services de classes d'agents intentionnels . . . 139
6.2.3 Les services d'instances d'agents intentionnels . . . 140
6.3 La répartition de l'information . . . 140
6.3.1 Les explorateurs . . . 141
6.3.2 L'éditeur . . . 144
6.3.3 La fenêtre d'information . . . 148
6.4 Utilisation . . . 149
6.5 Conclusion : un outil en évolution . . . 151

7 Conclusion et perspectives 153
7.1 Conclusion . . . 154
7.2 Perspectives . . . 156

Bibliographie **164**

A La notion d'intentionnalité **165**
A.1 Intentionalité ou intentionnalité ? 165
A.2 Philosophie et intentionnalité : la phénoménologie 166
A.3 Psychanalyse et intentionnalité : l'agentivité 168
A.4 Neurophysiologie et intentionnalité : intentionnalité de la perception 170
A.5 L'intentionnalité dans le rapport à autrui 173

B Détails techniques **179**
B.1 Le fichier `intentionalAgent_pl.pl` 179
B.2 Le fichier `intentionalObject.h` 184
B.3 Le fichier `intentionalClass.h` 188
B.4 Le fichier `intentionalAgent.h` 191
B.5 Le fichier `class_Predator.xml` 194

Table des figures

3.1 Perception - décision - action : une autre vision. 61

4.1 Du modèle vers l'implémentation : un lien bidirectionnel est nécessaire. . . 75
4.2 Du modèle vers l'implémentation : un lien sémantique pour assurer la continuité. 76
4.3 Une architecture multi-paradigmes. 80
4.4 Le double rôle du lien sémantique. 81
4.5 L'architecture d'agent intentionnel. 85

5.1 Première implémentation : une passerelle tcp/ip entre oRis et GNU Prolog. 106
5.2 Seconde implémentation : une intégration de GNU Prolog au moyen de classes natives oRis. 108
5.3 Un exemple simple de manipulation de modules dans SWI-Prolog. 115
5.4 Un exemple de surdéfinition de connaissances entre modules. 116
5.5 Un exemple de prédicat "transparent" aux modules. 117
5.6 Troisième implémentation : une intégration de SWI-Prolog dans ARéVi. . . 119
5.7 Un exemple de méthode : la méthode *turnLeft* de la classe *Predator*. 120
5.8 Des exemples d'attributs : les attributs de la classe *Predator*. 120
5.9 Un exemple de propriété : le calcul de *nearestPreyAngle* de la classe *Predator*. 121
5.10 La méthode *plForeignGetValue* : un pont entre impératif et déclaratif. . . . 123
5.11 Le prédicat *getProperty* : l'interface impératif/déclaratif du côté de Prolog. . 123
5.12 Exemple de règles comportementales : les règles du prédateur. 124
5.13 Exemple de descriptions d'action : les descriptions des actions du prédateur. 125
5.14 Un exemple de réification de classes : diagramme de structure statique. . . . 129
5.15 Exemple d'organisation des librairies dynamiques : le cas du prédateur. . . 132
5.16 Exemple de génération de code : les instructions saisies par l'utilisateur. . . 133
5.17 Exemple de génération de code "statique" : création d'une méthode de classe. 133
5.18 Exemple de génération de code "dynamique" : création d'une fonction externe. 133
5.19 L'architecture finale : diagramme de classe. 135

6.1 Smain : vue générale de l'interface. 141
6.2 Explorateur de composantes : examen d'une instance. 142
6.3 Explorateur de composantes : examen d'une classe. 143
6.4 L'éditeur de classe. 144
6.5 L'éditeur d'instance. 145

6.6 L'éditeur de connaissance. 146
6.7 L'éditeur de règle comportementale. 146
6.8 L'éditeur de description d'action. 147
6.9 L'éditeur de méthode. 147
6.10 L'éditeur de propriété. 148
6.11 Les autres éditeurs : l'exemple de l'éditeur d'attribut. 148
6.12 La fenêtre d'information et le réglage des préférences. 149
6.13 La règle comportementale de prédation initiale. 150
6.14 La règle modifiée pour générer une explication. 151
6.15 L'explication générée. 151

Propos préliminaires

> Maîtrisez le sujet, les mots suivront.
>
> CATON L'ANCIEN

Positionnement de la thèse

La thèse qui fait l'objet de ce mémoire traduit avant tout une démarche, une réflexion. J'ai vécu sa rédaction comme la création de mon chef-d'œuvre, dans le sens premier du terme évidemment ! Loin de moi l'idée de proposer une œuvre majeure, je suis au contraire assez peu satisfait du résultat. L'inverse serait à mon sens inquiétant pour l'avenir de mes activités de recherche. J'emploie ici le mot *chef-d'œuvre* tel qu'il était utilisé autrefois dans l'artisanat : je me suis considéré durant ces trois dernières années comme un apprenti, un compagnon, et ce mémoire constitue l'œuvre que je soumets à l'approbation de mes pairs pour rejoindre la confrérie.
Cette étude constitue au premier chef un travail que j'ai effectué sur moi-même, un rite initiatique, et si l'abord de l'informatique avec le vocabulaire de l'artisanat peut dérouter, il traduit ma démarche personnelle et la manière dont je considère cette activité : c'est pour moi un mélange de technique, de savoir et de création et, plus que tout, un formidable support de réflexion. En suivant un cursus d'ingénieur en formation initiale, j'ai rapidement compris que l'étude de la technique seule ne saurait satisfaire ma curiosité. Je me suis donc tourné vers la recherche dans ce domaine, plein d'interrogations qui me motivent au quotidien : qu'est-ce qu'un langage ? Qu'est-ce que le sens ? Si une machine peut apprendre, peut-elle savoir ?
Une foule d'autres questions plus ou moins pertinentes me taraudent et me conduisent sur la voie dangereuse d'une approche de l'informatique plus épistémologique que ce qui est habituellement attendu. C'est le fruit immature de cette réflexion que j'ai voulu partager dans ce mémoire. Bien sûr, il est difficile de se départir des premiers réflexes acquis et cette étude porte les séquelles de considérations purement techniques. Je ne regrette pas pour autant la dualité de ma formation : il me semble toujours être "en retard" et devoir me perfectionner techniquement autant qu'il me faut apprendre scientifiquement. C'est un sentiment frustrant mais moteur : j'espère ne jamais pouvoir me satisfaire de ce que je sais.

Si ce mémoire conduit à la proposition d'un modèle comportemental et à la présentation d'une application, il faut donc avant tout le considérer comme un accompagnement sur le

chemin de ma réflexion, comme l'exposition de mon point de vue. Comme toute prise de position, cette démarche est mâtinée d'ambiguïtés et d'imperfections : j'ai fait mon possible pour rester didactique tout en étant précis, mais les mots n'ont pas toujours suivis…

Conventions typographiques

La suite de ce document adopte des conventions typographiques très habituelles dans ce type de rédaction, mais il est tout de même bénéfique de les préciser.

Les chapitres débutent par une citation entourée de deux traits horizontaux, comme ceci :

> Il est plus bête qu'un idiot, c'est tout le monde.
>
> GUSTAVE FLAUBERT

Une police italique est utilisée pour noter les expression latines, les acronymes ou pour mettre en évidence un mot, ceci afin de rendre plus lisible des phrases telles que :

Le mot *mot* est un trigramme.

Un mot, ou un groupe de mots, est écrit en italique et entre guillemets quand son emploi est détourné, emprunté à un autre domaine ou qu'il relève d'une expression familière. Ces écarts ont pour but de "*faire passer le message*".
Une graisse plus importante est parfois utilisée pour **insister** sur un terme.

Dans les chapitres décrivant la réalisation (chapitres 5 et 6), les termes techniques sont notés avec une police à chasse constante : `terme technique`.

Pour les noms propres, une police n'utilisant que des caractères de haut de casse est utilisée : ROSENCRANTZ et GUILDENSTERN sont des personnages d'*Hamlet*, de WILLIAM SHAKESPEARE.

Les citations sont mises en évidence par un trait vertical épais en marge gauche. Voici un bref exemple avec *Chantre*, un monostique de GUILLAUME APPOLINAIRE :

> Et l'unique cordeau des trompettes marines

J'espère que ce document se révèlera clair, bonne lecture…

Chapitre 1

Introduction

> Le vrai est qu'il est aussi impossible de concevoir l'esprit que d'imaginer ce qui n'est pas.
>
> EDGAR ALLAN POE

L'informatique est souvent considérée comme un simple outil permettant de réaliser des machines qui se substituent à l'homme pour des tâches fastidieuses telles que de longues suites de calculs. Et si cette approche semble avoir été la motivation première des recherches dans le domaine, les développements actuels de l'informatique en font une science qui, au contraire de cette définition *a priori*, se révèle de plus en plus être une étude de l'homme, et non de la machine. Après avoir longuement œuvré à la conception de machines infaillibles permettant de mener à bien plus rapidement et plus surement que tout être humain des tâches sans cesse plus complexes, certains travaux tendent aujourd'hui à obtenir ce qui paraît être un désaveu de ces efforts : créer des entités artificielles faillibles qui hésitent et se trompent comme tout être humain. Pourquoi cela ? Si la première réponse donnée est souvent pragmatique – certains domaines nécessitant de modéliser des comportements humains aussi crédibles que possible, jusque dans leurs erreurs – la motivation réelle est certainement plus profonde : comme la sociologie, la philosophie, la biologie et de nombreuses autres sciences, l'informatique est désormais un moyen d'étude de l'homme par l'homme.

Cette analogie entre l'informatique et les autres sciences doit toutefois être considérée avec prudence. Premièrement, bien qu'elle repose sur des lois, des théorèmes, qu'elle soit riche d'une histoire et de différents "courants de pensée", l'informatique n'est pas universellement reconnue comme science. Le débat épistémologique ne sera certainement pas mené ici. Deuxièmement, l'étude de l'informatique dans son ensemble ne peut se ramener à cette approche anthropomorphique : **les travaux qui font l'objet du présent monographe concernent la modélisation de comportements d'agents autonomes situés dans un environnement virtuel**. Chacun de ces mots s'accompagne de concepts qui sont à préciser pour guider le lecteur dans son abord des idées développées. L'explication de ces termes-clefs fait l'objet du chapitre suivant. Pour les réflexions exposées ici en introduction il nous suffira d'évoquer le cadre général d'étude. Dans un premier temps, nous retiendrons donc que notre

but est de proposer un modèle informatique de comportements dans le contexte d'une approche individu-centrée.

Dans cette optique, la différence entre l'informatique et les autres sciences s'impose alors rapidement, même si leurs histoires présentent des similitudes notables. Par exemple, si nous comparons l'évolution de la médecine avec celle des langages informatiques, les mêmes mécanismes d'investigation scientifique ont été mis en œuvre, dans le même ordre. Ainsi, les premiers médecins dit animistes (PYHTAGORE et HIPPOCRATE par exemple) croyaient en l'existence de quatre éléments constitutifs du corps humain (l'eau, la terre, l'air et le feu), de quatre "caractères" permettant de qualifier l'état du patient (l'humide, le sec, le froid et le chaud) et de quatre humeurs responsables des maux (le sang, le phlegme, la bile jaune et l'atrabile). Si leur connaissance du corps humain prête aujourd'hui à sourire, elle permettait tout de même d'obtenir des résultats par l'administration de plantes ou la pratique de saignées, cautères... La démarche des médecins se limitait alors à une étude de causalité directe entre l'état du patient et ces actes médicaux élémentaires. Vinrent ensuite les pneumatistes[1] (citons notamment PLATON) qui identifièrent en plus de ces quatre caractères des fonctions vitales : le souffle vital, puis la circulation par exemple. Rapidement, les anatomistes, dont ARISTOTE est considéré comme le précurseur, se sont intéressés à l'étude des organes. L'évolution jusqu'à la médecine moderne telle que nous la connaissons s'est faite ensuite beaucoup plus lentement, les progrès étant liés aux avancées technologiques telles que l'invention du microscope. L'histoire de la médecine est évidemment beaucoup plus riche et complexe que ce résumé volontairement simpliste[2]. L'objectif ici est de mettre en évidence la succession des modèles de l'homme construits par les médecins. Au début de simples relations de causalité ont été identifiées et exploitées. Dans un second temps la vie a été considérée comme le concours de fonctions vitales. Enfin, le corps a été vu comme une composition d'organes, et la médecine actuelle ne se contente pas d'étudier leurs fonctionnement isolément : la compréhension du rôle des ces organes n'est possible que si leurs interactions et la dynamique de ces interactions sont prises en compte. De plus, l'étude des fonctions vitales et les relations causales directes entre l'état du patient et certains actes ou certaines substances n'ont pas pour autant été invalidées.
C'est cette évolution dans le type des modèles élaborés qui peut être mise en parallèle avec l'évolution des langages informatiques. Les premiers langages (FORTRAN, 1956) étaient procéduraux et donc basés sur des relations de causalité directes entre les commandes exécutées et les données traitées. Très rapidement, les langages fonctionnels firent leur apparition (LISP, 1958) offrant un premier niveau d'abstraction du modèle interne des machines. Enfin, les langages orientés objets (SMALLTALK, 1968) ont permis une modélisation par composition d'éléments. Actuellement, les limites de la modélisation par objets semblent atteintes dans les domaines qui nécessitent une modélisation fine d'interactions complexes. La modélisation par objet est insuffisante lorsque la dynamique des interactions entre les objets est difficilement prévisible. L'utilisation des systèmes multi-agents permet de traiter ces

[1] Le mot *pneumatiste* provient du nom *pneuma* qui été alors donné au *souffle vital*, la première fonction vitale identifiée comme telle.

[2] Le lecteur intéressé pourra se tourner vers des ouvrages spécialisés tel que [Sournia, 1991] sur lequel s'appuie ce paragraphe.

problèmes où la dynamique des interactions est primordiale.

Là aussi, le résumé de l'évolution des langages est drastiquement simplifié. Notre but est uniquement de mettre en évidence une similitude de la démarche dans l'abord de la médecine et de l'informatique. Dans les deux cas, la première étape est une compréhension très immédiate du système à travers l'identification de lois de causalité. Le besoin d'une abstraction s'impose alors et conduit à l'étude fonctionnelle du sujet. Enfin, pour certaines considérations l'aspect structurel interne ne peut être ignoré et incite à une modélisation du tout comme une composition d'éléments. Finalement, la dynamique des interactions entre ces constituants semble être le stade d'étude suivant.

Cependant, si une ressemblance peut être soulignée, une différence de fond demeure entre l'informatique et les autres sciences sur le plan de l'observation, et donc de la réfutabilité[3]. En effet en médecine, en biologie, en physique et dans toutes les autres sciences, le sujet d'étude préexiste au modèle. Pour l'informatique, c'est le cas seulement dans le cadre de l'étude des mécanismes de la machine. Par la suite, nous parlerons dans ce cas d'informatique fondamentale. Par contre, dans l'approche qui nous intéresse, la modélisation informatique de comportements d'entités artificielles, comme c'est le cas pour tout artefact informatique, la relation est complètement inversée : le modèle préexiste au sujet. Les observations des résultats obtenus lors d'une simulation ne peuvent donc pas constituer la base d'hypothèses nouvelles, elles ne reflètent qu'un cas d'utilisation programmé dans le modèle. Dans le cas d'applications complexes impliquant de nombreuses entités en interaction, il n'est pas toujours possible d'établir *a priori* un modèle du comportement global du système constitué. L'observateur prêtera alors un sens à une propriété macroscopique du système qui résulte de la superposition des interactions des entités qui le constituent. Ce phénomène est appelé *émergence de comportement* – ce qui sera sujet à discussion dans le chapitre suivant – et constitue ce qui est considéré comme un des principaux atouts de la modélisation multi-agents sur les approches classiques [Wegner, 1997].

La modélisation comportementale pour la simulation est intrinsèquement pluridisciplinaire et cette caractéristique est centrale à son étude : cette approche n'est pas auto-suffisante, dans le sens ou une série d'expériences ne donne pas lieu à de nouvelles hypothèses de travail. L'observation n'est plus alors un moyen de découverte mais uniquement un moyen de validation. Pour reprendre l'exemple de la médecine, l'évolution des théories et techniques médicales s'est principalement basée sur l'observation : la connaissance des organes était impossible aux médecins tant qu'ils ne pratiquaient pas de dissections, et les implications

[3]Selon le philosophe KARL POPPER (1902 - 1994), il doit être possible d'imaginer des contre-exemples à une loi pour qu'elle relève d'une théorie scientifique. Si aucun exemple pouvant *réfuter* une loi n'est imaginable, celle-ci appartient à une théorie non scientifique. L'observation a donc, en plus du rôle de découverte, un rôle de validation d'une théorie : l'observation du sujet est le vecteur de la *réfutabilité*. L'observation peut vérifier un exemple positif d'application d'une loi, pas la loi elle-même. Par contre, l'observation d'un exemple négatif suffit à réfuter une loi. Selon KARL POPPER, une loi qui ne s'expose pas à la réfutation par l'observation n'est donc pas scientifique [Popper, 1959]. Ce raisonnement est souvent utilisé pour distinguer les théories scientifiques de celles qui ne le sont pas. Un contre-exemple classique dans la littérature est la théorie du contrôle de l'univers par un Dieu omnipotent : quel que soit le phénomène observé, il sera toujours possible de prétendre qu'il relève de la volonté divine. Cette théorie irréfutable par l'observation n'est donc pas scientifique.

morales et religieuses pesant sur cet acte ont longtemps interdit tout progrès. Finalement, les pratiques illicites et les guerres, par le nombre de cas d'étude qu'elles ont amené, ont permis de progresser. Ainsi, la médecine s'est développée par l'observation lors de la manipulation et de la deconstruction du sujet d'étude. Dans le cas de la modélisation comportementale, le progrès doit se faire à la construction du modèle, et éventuellement lors de sa manipulation (la pratique du *prototypage interactif* sera évoquée ultérieurement). La compréhension du sujet se fait donc par abstractions successives puisqu'elle relève d'un procédé créatif : il s'agit de concevoir des modèles générateurs, et non de déduire des modèles explicatifs de l'expérimentation. Les observations n'ont plus alors d'intérêt que pour la validation. Les hypothèses de travail doivent provenir d'ailleurs, ce qui condamne la modélisation comportementale à la pluridisciplinarité : les fondements de tout modèle comportemental informatique proviennent nécessairement d'un autre domaine, tel que l'éthologie, la sociologie, la physiologie…

Cette nécessaire pluridiciplinarité engendre une double difficulté.

Premièrement, les modèles qui servent de base à l'élaboration des modèles comportementaux sont eux-mêmes sujets à caution : si les domaines d'application et la validité d'un modèle sont parfois facilement identifiables – notamment pour les sciences dites "dures", dans le cas de modèles physiques par exemple – il arrive que cette identification soit extrêmement délicate. C'est notamment le cas pour les modèles sociaux ou émotionnels qui sont parfois exploités. A cela s'ajoute une inévitable dégradation du modèle lors de son implémentation informatique. Ces approximations impliquent d'établir une procédure d'évaluation pour valider ou non la proposition. Et c'est là que se situe le premier obstacle : le choix des critères de validation appliqués lors de l'observation d'une simulation n'est pas trivial. Si les critères choisis sont ceux du domaine de provenance du modèle - critères sociaux ou émotionnels par exemple – l'altération subie par ce dernier lors de sa "traduction" dans le domaine informatique, et les conditions particulières de la simulation ne permettent généralement pas de valider, ou d'invalider, les hypothèses de travail. Il faut donc distinguer deux catégories très différentes d'objectifs poursuivis dans la modélisation comportementale.

D'une part, la démarche peut être d'exploiter un modèle comportemental d'un domaine connexe en profitant des propriétés des simulations informatiques : répétabilité des expériences, maîtrise totale des conditions initiales… L'observation est alors faite suivant les critères d'évaluation du domaine abordé. Par exemple, lorsqu'un modèle de comportement de foule est utilisé pour simuler les déplacements d'un flux de personnes dans un bâtiment en projet, les déplacements doivent être considérés selon les concepts éthologiques utilisés dans le modèle : la question n'est pas de connaître la distance cartésienne moyenne entre les individus qui constituent la foule, mais de savoir si ces individus se sentiront oppressés dans le couloir modélisé. L'informatique est dans ce cas utilisée comme simple outil de simulation, les résultats obtenus ne peuvent rien apporter d'autre à la modélisation comportementale que des progrès d'ingénierie. Le domaine d'emprunt du modèle reste prédominant.
D'autre part, la seconde approche de la pluridiciplinarité est la libre inspiration d'un autre domaine pour l'établissement d'un modèle informatique. L'objectif n'est pas alors la simple simulation d'un modèle numérique préétabli, mais la constitution d'un nouveau type de mo-

dèle pour des applications informatiques ou de nouvelles démarches de conception. Les résultats de l'observation doivent alors être considérés avec les critères de l'informatique fondamentale : complexité algorithmique, décidabilité,...

Malheureusement, dans le cas de simulation comportementale la séparation entre ces deux cas de figure n'est pas possible. La finalité est non seulement d'établir un modèle informatique efficace, mais en plus d'obtenir un comportement satisfaisant de l'entité artificielle ainsi créée. Les critères d'observation sont donc autant issus du domaine de l'informatique fondamentale que des domaines connexes à la modélisation comportementale. Dans ce type de travaux, il faut alors clairement discerner les éventuels apports d'une nouvelle proposition : les progrès en technique de modélisation risquent d'être confondus avec les bénéfices de la simulation informatique pour les sciences comportementales.
Cette ambiguïté apparente dans les objectifs n'est pas propre à la modélisation comportementale d'agents autonomes. La même confusion règne dans un des domaines piliers de cette discipline : l'intelligence artificielle.

En effet, depuis sa naissance[4] dans les années 50, l'intelligence artificielle s'est divisée en deux courants principaux. Une approche repose sur l'imitation des processus intelligents de l'homme ou de l'animal pour la conception de modèles informatiques. L'autre approche, dite classique, consiste à essayer de reproduire à l'identique ces processus mentaux. Cette deuxième démarche est celle qui introduit le plus d'ambiguïté en posant le problème de la définition de l'intelligence. Face à la nécessité d'établir des critères d'observation permettant de conclure ou non à l'intelligence d'une machine, ALAN TURING proposa un test qui fit consensus, actuellement encore référencé comme *le test de Turing* [Turing, 1950]. L'idée fondatrice de ce test est qu'une machine peut être considérée comme intelligente si il est impossible de la distinguer d'un être humain[5]. Si ce test paraissait satisfaisant au regard de ce qu'était l'intelligence artificielle à l'époque, à savoir principalement la résolution symbolique de problèmes, il amène par son principe même de nombreuses questions.

La première interrogation que soulève ce test concerne évidement l'appréciation de ce qu'est l'intelligence. Il est dérangeant d'accepter qu'il puisse être décidé de l'intelligence d'une entité à la seule lecture de ses réponses à une série de questions. Premièrement, parce que cela fait abstraction d'une grande part des aspects de communication pour se placer au

[4]La naissance de l'*intelligence artificielle* comme dicipline à part entière est généralement attribuée aux conséquences d'une conférence tenue à Dartmouth (Etats-Unis) en août 1956. C'est lors de cette conférence que le terme a été proposé et il fut adopté malgrè l'opposition de certains acteurs du domaine. Cette conférence rassemblait notamment ceux qui allaient devenir les pionniers de cette nouvelle voie de recherche : J. MCCARTHY,M. MINSKY, C. SHANNON, A. NEWELL,...

[5]Le test de TURING se déroule en deux étapes, dont la première qui apporte peu a rapidement été négligée. Le mode opératoire est le même pour ces deux étapes : un opérateur dialogue avec deux entités au moyen d'un terminal informatique, l'échange est donc strictement textuel. La première étape consiste pour l'opérateur à distinguer un homme d'une femme. Cette étape provient du fait que ALAN TURING soutenait qu'il est impossible de discriminer le sexe d'une entité par une modélisation purement symbolique. La deuxième étape, celle qui est désormais seule désignée comme *le test de Turing*, consiste à demander à l'opérateur dans les mêmes conditions de distinguer un être humain d'un programme informatique. La machine passe le test si l'opérateur commet autant d'erreurs vis-à-vis des deux entités : il ne distingue pas la machine, ou désigne les deux entités comme étant des machines, l'être humain compris (ce qui arrive !).

seul niveau de l'esprit : la restriction de la communication entre deux entités au seul mode écrit est une hypothèse très forte. Si les aspects strictement langagiers sont pris en compte, ce qui représente un travail très important sur le plan informatique, il serait abusif de considérer que le *test de Turing* incorpore toutes les facettes de la communication entre les entités impliquées. De plus, à aucun moment il n'est question de chercher à identifier certains processus mentaux mais uniquement leurs effets, les "sorties" du système sous forme symbolique. Ce test se place au niveau du pur raisonnement tout en refusant d'en aborder les mécanismes. Le grand absent du test est alors le sens : il n'est pas nécessaire que les réponses d'une entité aient de sens pour celle-ci pour qu'elle soit désignée comme étant intelligente. Un retour à l'origine étymologique du mot *intelligence* comme *capacité de comprendre* met alors en évidence l'insuffisance du *test de Turing*. L'exemple de *la chambre chinoise* de SEARLE[6] en est une illustration. En poussant ce genre de raisonnement à son extrémité, il est envisageable qu'un dispositif débitant des réponses tirées aléatoirement d'une base de données puisse créer, par un hasard favorable hautement improbable, l'illusion de l'intelligence alors qu'il est avéré que des entités qui *sont* intelligentes, des êtres humains en l'occurrence, ne sont parfois pas reconnues comme telles par ce test.
Cette confusion provient du fait que s'il est facile de déterminer des tâches que l'intelligence permet de mener à bien, cela n'implique en rien que le simple fait d'accomplir ces tâches atteste de l'intelligence du "cobaye". Il est même parfaitement possible qu'une entité intelligente échoue à un tel test. L'idéal de l'intelligence artificielle comme approche antropomorhique serait de reproduire les hésitations et erreurs au même titre qu'elle tend à reproduire les raisonnements.

Une fois encore, le test doit être cohérent avec les objectifs poursuivis. Dans le cadre de la résolution de problème, comme la création de programmes jouant aux échecs par exemple, seule compte la résolution, même si les choix faits par la machine ne correspondent pas au raisonnement qu'aurait tenu un être humain pour arriver à la victoire. Inversement, dans le cas où le but est le mimétisme des décisions humaines, il faut juger des défaillances tout autant que des succès.

C'est cette seconde approche qui nous intéresse particulièrement : le but de la spécification de comportement pour des entités virtuelles immergées dans un environnement virtuel n'est pas d'obtenir des agents "froidement" intelligents. Ces agents doivent être crédibles. Les systèmes d'intelligence artificielle conçus pour satisfaire au *test de Turing* sont de plus en plus perfectionnés et intègrent de nos jours – plus de 50 ans après la création du test – de plus en plus de biais pour "faire illusion" : le programme introduit des aléations et des temps d'attente variables dans sa construction des réponses afin d'humaniser le discours. Il est donc raisonnable de considérer qu'ils atteignent un niveau de crédibilité satisfaisant pour ce test, c'est-à-dire pour une interaction purement textuelle avec l'opérateur. Dans le cas d'agents autonomes dans un environnement tridimensionnel, il apparaît très vite que les

[6]Dans [Searle, 1983] l'auteur imagine un opérateur enfermé dans une chambre où il possède des tables de correspondances entre des séries d'idéogrammes. Lorsque quelqu'un glisse sous la porte une feuille où est inscrite une question en chinois, l'opérateur qui ignore tout de cette langue se content d'inscrire en réponse ce que lui donne les tables de correspondance dont il dispose. Depuis l'extérieur, un observateur peut considérer que l'opérateur enfermé dans la pièce maîtrise le chinois. SEARLE introduit cette métaphore pour exposer son opinion selon laquelle un ordinateur ne fait que manipuler des symboles qui n'auront jamais de signification pour lui, quelle que soit la pertinence du traitement effectué pour un observateur extérieur.

critères d'évaluation vont être beaucoup plus nombreux. Si l'interaction peut se faire par le dialogue, non seulement le discours en lui-même mais également tous les aspects non verbaux de la conversation rentrent en ligne de compte : attitude, variation du ton et du volume de la voix, expression du visage… De plus, la crédibilité du comportement sera aussi évaluée par l'utilisateur sur la base des actions entreprises par les agents, et ce même en l'absence d'interaction directe : il s'agit d'agents autonomes, ils ne doivent pas revenir à l'inactivité dès que l'utilisateur cesse de les solliciter. De plus, ils interagissent aussi entre eux, et l'utilisateur peut être témoin de ces interactions ou de leurs effets. Il est alors aisé d'imaginer la difficulté d'établir un test objectif des résultats obtenus lors de la simulation du comportement d'une telle entité artificielle. Il semble totalement vain de proposer une "échelle de crédibilité" : les résultats à juger sont à confronter à l'application ciblée. Par exemple, un même modèle comportemental peut donner des résultats qui seront jugés acceptables pour un jeu vidéo guerrier, mais insuffisants pour un environnement virtuel de formation destiné à des militaires. La *crédibilité* recherchée dans ce genre d'approche est d'ailleurs relativement difficile à définir, le choix de ce terme n'est sans doute pas innocent : ce flou entretenu reflète l'incapacité à établir des critères objectifs d'évaluation. Imaginons un concepteur de jeu vidéo qui est chargé de définir le comportement d'un dragon. Le résultat de son travail peut être jugé peu *crédible*, si par exemple l'animal vole trop lourdement. Par contre, le modèle ne pourrait en aucun cas être jugé *réaliste*, aucune observation d'un dragon réel ne pouvant corroborer la simulation. Il est possible, et parfois souhaitable, de ne pas croire en ce qui est réaliste et, inversement, de croire en ce qui ne l'est pas. Les applications de jeu vidéo ou de théâtre virtuel illustrent régulièrement ce principe. La notion de crédibilité est subjective et contextuelle. Les modèles de comportements d'agents autonomes doivent donc être évalués en fonction des notions sur lesquelles ils sont fondés : il est inutile de juger le rendu des émotions d'un agent dont le modèle de comportement ne définit pas d'aspect émotionnel.

Cependant, avec toutes ces nuances et ces restrictions à l'esprit, il reste délicat d'apprécier la pertinence d'un modèle de comportement. Cette difficulté est d'autant plus importante que les modèles envisagés sont de plus en plus complets et prennent fréquement en compte des aspects dont l'appréhension nécessite une connaissance avancée d'un domaine connexe : les recherches actuelles traitent de modèles sociaux, conversationnels, émotionels…

Bien souvent, le seul recours pour l'évaluation est alors la comparaison avec l'homme au moyen d'un test basé sur le principe du *test de Turing* : l'entité réussit le test si d'un point de vue externe elle se comporte comme un homme dans la même situation. Cela implique que l'expérimentateur ait une connaissance suffisante des réactions humaines attendues, donc qu'il dispose d'une forme de modèle du comportement de l'homme. Or, si ce modèle existe, ne serait-ce qu'à titre d'expertise, il est certainement possible de l'implémenter comme un nouveau modèle de comportement. La question qui se pose alors, si le raisonnement est poussé à son terme, est celle de l' existence d'un critère qui permettrait en dernier lieu de discerner l'être humain de la machine. C'est ce problème de fond du jugement du caractère d'humanité par l'homme lui-même qui constitue le centre de [Dick, 1968] : dans son roman d'anticipation, PHILIPPE K. DICK imagine des androïdes tellement similaires à l'homme, les répliquants, qu'il est impossible de les distinguer *a priori*. L'auteur introduit alors un test

– le test de Voigt-Kampf, du nom de son inventeur fictif – qui permet d'identifier les machines : il s'agit d'une forme de *test de Turing* qui analyse les capacités empathiques du sujet testé. D'après PHILIPPE K. DICK, ce qui distingue en dernier lieu l'homme de la machine est donc sa capacité à s'identifier à autrui, donc finalement sa capacité à construire un modèle de l'autre et à se projeter dans cette modélisation...

La première difficulté amenée par la pluridisciplinarité de la modélisation comportementale d'entités autonomes en environnement virtuel est donc l'évaluation des résultats obtenus. La deuxième, s'il ne faut en retenir que deux, est assurément la difficulté de communication. En effet, lorsque deux domaines aussi denses que l'informatique fondamentale et la psychologie sont confrontés, chaque mot peut être sujet à débat et doit être clairement défini. Ainsi dans cet exemple, deux spécialistes qui se rencontrent ne seront pas d'accord d'emblée sur la signification de mots tels que *processus*, *message*, *méthode* ou *connaissance*. Comme évoqué précédemment, les modèles comportementaux d'entités artificielles recourent parfois à des modèles neurophysiologiques, psychologiques, sociaux... Chacun de ces "emprunts" peut conduire à des erreurs d'interprétation, voire à des contresens. Cet exercice de la pluridisciplinarité nécessite beaucoup de prudence et de rigueur et constitue souvent une entrave à la progression : si le concepteur du modèle n'est pas lui-même un expert des différents domaines abordés, il doit passer par des phases d'audit d'expertise qui s'avèrent trop souvent délicates.

Après avoir pris conscience de la difficulté de composer avec ces domaines si différents, il peut demeurer un doute sur l'intérêt de s'engager dans une voie de recherche pour laquelle les résultats sont *a priori* délicats à valider, et au sujet de laquelle toute communication sera difficile à mener. Le principal attrait de cette pluridisciplinarité est la volonté de la commuer en transdiciplinarité : si certains principes, certains modèles, resistent à la variation du domaine d'étude ils acquièrent alors une sorte de légitimité *de facto*. L'idée de fond qui anime le modélisateur est que si certaines des idées qu'il manipule perdurent et conservent ou prennent du sens quand elles sont transposées ainsi d'un domaine à l'autre, c'est qu'elles relèvent d'un concept transverse significatif pour le problème sous-jacent : la compréhension de l'esprit humain. La modélisation comportementale, comme la neurophysiologie, la psychologie, la philosophie, et toutes les sciences impliquées dans cette activité, se heurte aux limites de notre compréhension du fonctionnement de nos esprits. La confrontation de toutes ces théories est donc sous-tendue par un unique but : identifier des invariants qui participent de l'explication de ce que nous sommes. Les phénoménologues parleraient d'intersubjectivité : si les spécialistes de ces différentes diciplines s'accordent sur l'existence d'un concept commun, il est raisonnable d'accepter son existence objective. C'est en cela que, comme déclaré en début de cette introduction, la modélisation comportementale d'entités artificielles devient une étude de l'homme par l'homme : l'activité principale du modélisateur est après tout, ou plutôt avant tout, de s'interroger sur lui-même et sur les autres.

Les travaux qui font l'objet de ce mémoire, par leur caractère pluridisciplinaire, n'échappent pas aux difficultés de positionnement et de justification vis-à-vis des domaines abordés. Le chapitre suivant s'attache donc à préciser le sous-domaine de l'informatique qui constitue le

cadre de cette étude : la spécification de comportement d'agents autonomes immergés dans un environnement virtuel. Le chapitre 3, pour sa part, résume la réflexion menée autour de la notion d'intentionnalité et justifie son intérêt dans la conception d'un modèle comportemental. Le développement de cette réflexion est reporté en annexe pour une plus grande lisibilité : seuls les aspects informatiques apparaissent ainsi dans le corps du document.
Les chapitres suivants détaillent notre contribution. Le chapitre 4 présente le modèle d'agent *intentionnel*. Le chapitre 5 détaille le modèle d'exécution adapté à la simulation du comportement d'agents ainsi définis. Le chapitre 6 présente l'application qui a été réalisée sur la base de ces modèles. Enfin, le chapitre 7 conclut sur les apports de la prise en compte de la notion d'intentionnalité dans la spécification de comportements et les perspectives d'études complémentaires ouvertes par cette approche.

Chapitre 2

Agents autonomes et réalité virtuelle

> Il faut se méfier des ingénieurs : ça commence avec une machine à coudre, et cela finit par la bombe atomique.
>
> MARCEL PAGNOL

Les travaux présentés ici ont pour but la spécification de comportements d'agents autonomes situés évoluant dans un environnement virtuel. Comme indiqué en introduction, chacun de ces mots est fortement connoté et cette expression doit être détaillée dans tous ses aspects afin d'expliciter les bases, les limites et les champs d'application de cette étude. C'est l'objet du présent chapitre.

Le point de départ de cet exposé est la définition de ce qu'est un agent, ou plus généralement l'approche multi-agents pour le développement d'applications informatiques. Dans un second temps, la notion de réalité virtuelle est présentée, ainsi que les conséquences de l'immersion d'un système multi-agents dans un environnement virtuel. Enfin, les concepts de *comportement* et de *spécification* de comportement doivent être précisés afin d'exposer clairement les objectifs de notre approche.

2.1 L'approche multi-agents

Au cours des vingt dernières années, les approches multi-agents se sont progressivement imposées comme une voix de recherche au potentiel prometteur. Elles sont même considérées par certains comme *la nouvelle révolution du logiciel* [Guilfoyle and Warner, 1994] et de nombreux articles traitent des apports de ces approches [Wooldridge and Jennings, 1995a, Wegner, 1997, Jennings, 2000, Shen et al., 2001]. Pourtant, il reste délicat de trouver une définition des agents logiciels, et donc des systèmes multi-agents, qui fasse l'unanimité. A ce sujet, les auteurs rapportent dans [Wooldridge and Jennings, 1995a] une remarque faite par CARL HEWITT[1] lors de la 13^{eme} conférence internationale sur l'intelligence artificielle distribuée :

> [T]he question *What is an agent ?* is embarrassing for the agent-based computing community in just the same way that the question *What is intelligence ?* is embarrassing for the mainstream AI community.

Cette section n'a pas la prétention de répondre au problème de *la* définition de l'agent, mais plutôt celle de présenter *une* définition de l'agent qui corresponde à notre cadre d'étude. Pour cela, nous allons examiner les approches multi-agents comme un paradigme d'étude, au même titre qu'il existe le paradigme objet par exemple, en se basant sur une définition généraliste de l'agent logiciel. Dans un second temps, nous examinerons des critères de taxonomie des types d'agents afin de situer notre approche parmi les travaux de la communauté.

2.1.1 Le paradigme agent

Les systèmes multi-agents sont souvent définis comme des systèmes informatiques dont le comportement global est le résultat des interactions des agents logiciels qui le constituent. D'autre part, les agents logiciels sont souvent définis comme les entités constitutives d'un système multi-agents.
Cette explication récursive est naturellement génante et nous renvoie à la difficulté de définition du concept d'agent. Ce floue qui semble entourer ces approches ne provient pas d'un manque d'exactitude du sujet mais de sa complexité. Quand il est question d'approche multi-agent, il s'agit en fait d'une démarche d'analyse et de décomposition qui peut s'appliquer à plusieurs niveaux du cycle de conception d'un logiciel, avec différents outils et des conséquences très distinctes. C'est pourquoi nous abordons les approches multi-agents comme un paradigme d'étude et non comme une technique ou une technologie.

Le terme *paradigme* provient originellement de la linguistique. Il est construit sur la racine grecque *paradeigma* qui signifie *exemple, modèle*. Une des premières définitions précises du terme remonte à 1916 où FERDINAND DE SAUSSURE écrivait dans son *Cours de linguistique générale* :

[1] CARL HEWITT est professeur au MIT, il est un spécialiste reconnu de l'intelligence artificielle.

> Le paradigme permet de distinguer les relations qu'une unité linguistique entretient avec d'autres unités linguistiques présentes dans la chaîne du discours (*in praesentia*) des relations qu'elle entretient avec les unités absentes et qui pourraient occuper sa place (*in absentia*).

Le paradigme peut donc être compris comme la définition des frontières d'une classe de substituabilité. Par exemple, dans la phrase *Je bois mon café dans une tasse*, il est possible de remplacer la mot *tasse* par le mot *bol*. Le sens de la phrase est alors changé, mais la substitution se fait au sein du paradigme du "*contenant de café*" : la classe de substituabilité a été respectée.
Cette définition purement linguistique a ensuite été étendue, passant des unités linguistiques aux classes de concepts. Ainsi, d'un point de vue épistémologique, nous retiendrons ici la définition donnée par le philosophe THOMAS KHUN dans son ouvrage *Structures des révolutions scientifiques* (1962) :

> Paradigme : les règles admises par la communauté scientifique, à un moment donné de son histoire, pour délimiter et problématiser les "faits" qu'elle juge dignes d'étude.

La définition d'un paradigme est donc la démarche consensuelle d'une communauté scientifique qui identifie une classe de problèmes ainsi que les outils – vocabulaire, théorèmes, techniques, technologies – permettant de traiter ces problèmes. C'est en ce sens que les différentes approches multi-agents relèvent d'un même paradigme : toutes ces approches, depuis les méthodologies de conception jusqu'aux plateformes dédiées, sont fondées sur l'exploitation des interactions d'agents. Afin de considérer les différents aspects de la conception logicielle qui peuvent recourir à la modélisation multi-agents, nous allons partir d'une définition généraliste de l'agent logiciel qui sera progressivement affinée dans les sections suivantes. Parmi les nombreuses définitions proposées dans la littérature, nous retiendrons ce que WOOLDRIDGE et JENNINGS qualifient de *notion faible d'agent* (traduction d'un extrait de [Wooldridge and Jennings, 1995a]) :

> De manière générale, le terme agent est utilisé pour dénoter un système informatique matériel ou (plus couramment) logiciel doté des propriétés suivantes :
> – autonomie : les agents agissent sans intervention directe des humains, et possèdent un moyen de contrôle de leurs actions et de leur état interne ;
> – capacité social : les agents interagissent avec les autres agents (et éventuellement avec les humains) *via* un langage de communication *agent* ;
> – reactivité : les agents perçoivent leur environnement (qui peut être le monde physique, un utilisateur au travers d'une interface graphique, un ensemble d'autres agents, l'Internet, ou peut-être une combinaison de tout cela), et répondent de manière opportune à ses changements ;
> – proactivité : les agents n'agissent pas seulement en réponse à leur environnement, ils sont capables de faire montre de comportements dirigés par des buts en *prenant l'initiative*.

Les agents sont donc des entités autonomes, en ce sens qu'il perçoivent leur environnement, réagissent à l'évolution de celui-ci et anticipent sur ces évolutions en agissant de

manière à atteindre un but. Il s'agit de plus d'entités communicantes. La notion clef de cette définition est certainement la capacité d'un agent à *prendre l'initiative*.

Toute démarche de conception visant à décomposer un problème et à proposer une solution basée sur les interactions de telles entités sera donc qualifiée d'*approche multi-agents*. Un tel raisonnement peut être mené à différents niveaux du cyle de conception d'un logiciel. La simulation informatique est usuellement présentée comme la composition de trois activités ([Fishwick, 1994] cité dans [Drogoul et al., 2003]) :

- conception d'un système réel ou théorique ;
- exécution de ce modèle sur un ordinateur ;
- analyse des résultats de cette exécution.

L'analyse des résultats possède un statut particulier : elle relève de l'interprétation et de la synthèse, non de la décomposition analytique. Cette étape du développement ne saurait donc être abordée en suivant une approche multi-agents, elle en étudie les conséquences. De plus, comme exposé en introduction, dans le cadre de simulation comportementale l'analyse des résultats diffère de l'interprétation des autres types de simulations numériques. Dans ce contexte particulier emprunt de pluridisciplinarité, les deux autres étapes identifiées par FISHWICK méritent d'être détaillées.

Ainsi, la conception d'un système est susceptible de faire intervenir un spécialiste du domaine abordé – nous reprendrons pour le désigner le terme *thématicien* employé dans [Drogoul et al., 2003] – et un spécialiste de la conception logicielle : pour les domaines complexes, le concepteur informatique est trop ignorant du domaine d'inspiration pour effectuer cette analyse seul, et ces deux niveaux d'expertise sont requis. Nous distinguerons donc l'analyse du problème et la conception du modèle informatique comme deux aspects distincts, bien qu'intimement liés, de la conception du système.

Enfin, l'étape d'exécution sur ordinateur sera elle aussi séparée en deux étapes : le développement de l'application d'une part, et le déploiement de celle-ci d'autre part. Les enjeux de la répartition et de la distribution d'applications informatiques sont tels que l'étude des techniques de déploiement constitue un sous-domaine à part entière de la simulation informatique. De plus, il est tout à fait possible de déployer comme un système multi-agents un ensemble d'applications non orientées agent. L'étape du déploiement est donc à traiter séparément.

Nous retiendrons donc finalement pour détailler le cycle de développement d'une application logicielle les étapes distinguées par PIERRE-MICHEL RICORDEL et YVES DEMAZEAU dans [Ricordel and Demazeau, 2001] pour leur étude comparative des plateformes multi-agents :

- analyse ;
- conception ;
- développement ;
- déploiement.

Chacune de ces étapes peut être traitée à l'aide des concepts du paradigme multi-agents. Cependant, comme le fait remarquer [Drogoul et al., 2003], le consensus sur ces concepts communs est principalement syntaxique et non sémantique : si les mêmes termes sont ma-

nipulés, la manière dont ces concepts sont exploités diffère d'une étape à l'autre. Ainsi, la plupart des plateformes et méthodologies multi-agents se concentrent sur un sous-ensemble de ces étapes, et il n'existe pas d'approche intégralement multi-agent (de l'analyse jusqu'au déploiement).

Les quatre sections qui suivent examinent ce qu'est l'approche multi-agents au regard de ces quatre étapes :

- les *théories multi-agents* pour l'analyse ;
- les *méthodologies multi-agents* pour la conception ;
- les *langages multi-agents* pour l'implémentation[2] ;
- les *plateformes multi-agents* pour le déploiement.

La bonne réalisation de ces quatre étapes nécessitent des compétences différentes et incombent à des experts *a priori* distincts. Dans la pratique, lorsque les domaines abordés sont peu complexes, la même personne peut endosser plusieurs rôles. Par la suite, nous emploierons les termes suivants pour identifier ces spécialistes :

- le *thématicien* est expert du domaine d'inspiration du comportement simulé, il est en charge de l'analyse du problème ;
- le *modélisateur* est expert en génie logiciel et choisit l'architecture la plus à même de supporter le modèle construit par le thématicien, il est en charge de la conception du modèle informatique ;
- l'*informaticien* est expert du développement logiciel et implémente au mieux l'architecture choisie par le modélisateur, il est en charge de l'implémentation ;
- l'*utilisateur final* est celui qui exécute la simulation obtenue, il est en charge du déploiement de l'application.

En pratique, l'informaticien est très souvent en charge du déploiement de l'application. De même, la modélisation est souvent assurée par l'informaticien et le thématicien conjointement dans une démarche d'audit d'expertise.
Ces différents rôles ne sont pas nécessairement incarnés par des personnes différentes et les étapes correspondantes sont parfois confondues, mais dans chacune de ces activités il est possible de recourir au paradigme multi-agents avec un impact différent.

2.1.2 Du problème à l'analyse : l'agent unité conceptuelle

Le rôle du thématicien dans le cycle de développement d'un logiciel est de transcrire ses connaissances du domaine abordé. Cette opération est souvent délicate : il est parfois difficile de discriminer les informations capitales des détails subalternes lors d'un audit d'expertise. C'est un obstacle très connu des concepteurs de systèmes experts : certains principes paraissent tellement évidents pour un spécialiste qu'il ne juge pas toujours nécessaire de les

[2]Le terme implémentation est un anglicisme : *implementation* signifie *réalisation,ouvrage, exécution*. Il est utilisé dans les publications anglosaxones pour désigner l'étape de programmation au sens strict (traduction en langage informatique). Ce mot est passé tel quel dans la communauté française, les autres termes possibles semblant trop généralistes ou connotés (programmation, implantation, codage, développement…). Nous cédons donc aussi à la facilité du néologisme…

expliciter au modélisateur. Ce phénomène est renforcé par une barrière psychologique : le thématicien refuse parfois, plus ou moins consciemment, de livrer des informations clefs qui permettent la construction d'un modèle efficace qui pourrait remplacer son expertise chèrement acquise.
Une fois ces incontournables difficultés surmontées, le principal problème demeure le choix d'un formalisme exploitable par le modélisateur permettant la transcription des connaissances du thématicien. La notation adoptée doit offrir suffisament d'expressivité au thématicien pour lui permettre de détailler ses connaissances sans concession. Cette même notation doit permettre une analyse aisée par le modélisateur, et le niveau de formalisation doit être suffisant pour valider la "*traduction*" : un aspect du rôle de modélisateur est la validation des transformations qu'il effectue sur les propositions du thématicien dans sa construction d'un modèle conceptuel.
Dans ce cadre, l'approche multi-agents peut se traduire de deux manières différentes : le thématicien peut être amené à exprimer ses connaissances selon un formalisme qui a été étendu pour comporter les concepts liés aux agents, ou il peut recourir à un nouveau formalisme basé totalement sur cette approche.
La première de ces démarches est représentée par exemple par les extensions d'*UML*[3] telles que [Bauer et al., 2000, Odell et al., 2000], les travaux de LUCK et D'INVERNO qui se basent sur le formalisme Z [Luck et al., 1997] ou la méthodologie GAIA [Wooldridge et al., 2000] qui s'appuie sur la méthode de conception orientée objet FUSION. Pour ces méthodes, il ne s'agit pas vraiment d'une analyse orientée agent, mais plutôt d'une analyse *en vue* d'une modélisation multi-agents. Le principal avantage de telles approches est qu'elles sont fondées sur un formalisme éprouvé, pour lequel il existe un savoir-faire répandu et des outils robustes. Le principal inconvénient est que ce formalisme n'étant pas à l'origine fondé sur les concepts utilisés lors de la modélisation (les concepts agents), il introduit un biais supplémentaire dans le cycle de développement et peut conduire à une décomposition inopportune du problème. A ce sujet, WOOLDRIDGE et CIANCARINI écrivent dans [Wooldridge and Ciancarini, 2000] :

> There are obvious advantages to such approach, the most obvious being that the concepts, notations, and methods associated with object-oriented analysis and design (and UML in particular) are increasingly familiar to a mass audience of software engineers. However, there are several disadvantages. First, the kinds of *decomposition* that object-oriented methods encourage is at odds with the kind of decomposition that *agent oriented* design encourages. Put crudely, agents are more coarse-grained computational objects than are objects ; they are typically assumed to have the computational ressources of a UNIX process, or at least a Java thread. Agent systems implemented using object-oriented programming languages will typically contain many objects (perhaps millions), but will contain far fewer agents. A good agent oriented design methodology would encourage developpers to achieve the correct decomposition of entities into either agents or objects.

La seconde catégorie de méthodes d'analyse qualifiée d'orientées agents est constituée de méthodes totalement nouvelles, dont les concepts fondateurs sont ceux des approches multi-agents. Ces méthodes conviennent particulièrement aux domaines pour lesquels l'identifi-

[3] Unified Modelling Language [Rational, 1997]

cation d'entités et la spécification de leurs interactions constituent des décompositions instinctives du problème. Elles ont par exemple été utilisées avec succès pour des simulations sociologiques, économiques, éthologiques... Ces approches sont aussi qualifiées d'individu-centrées : le concept d'entité est central à l'analyse et chaque entité considérée correspond à un individu (éventuellement fictif) inscrit dans une organisation.
Dans ce cadre, le paradigme multi-agents est pleinement exploité et l'analyse du problème repose sur l'expression d'une dynamique d'interaction entre agents exprimée au moyen de notions de haut niveau : agents, groupes d'agents, rôles, responsabilités, buts, protocoles de communication...
Cette démarche est celle qui sous-tend des méthodologies telles que CASSIOPEIA [Drogoul and Zucker, 1998] ou DESIRE [Brazier et al., 1995] par exemple.

Les défauts de ces approches sont principalement des *"défauts de jeunesse"* : il s'agit de méthodologies nouvelles dont la formalisation est ardue, et de nombreux aspects délicats à traiter se révèlent au fur et à mesure des expérimentations. Le lien avec les étapes suivantes du cycle de développement n'est pas toujours validable, faute d'une formalisation suffisante.

Notre dessein est ici de mettre en évidence les différences entre ces méthodologies, pas de détailler certaines d'entre elles : cela dépasserait largement le cadre de cette étude. Le lecteur intéressé pourra se reporter à [Iglesias et al., 1999] ou [Casteran et al., 2000] pour une revue très complète des méthodologies orientées agents existantes, ou à des articles les détaillant individuellement ([Iglesias et al., 1997] pour MAS-CommonKADS, [Glaser, 1996] pour CoCoMAS...).

Pour toutes ces méthodes d'analyse, l'agent est une unité conceptuelle qui guide le thématicien dans la décomposition du problème.

2.1.3 De l'analyse à la conception : l'agent unité computationnelle

Le rôle du modélisateur dans le cycle de développement d'un logiciel est de transcrire le modèle établi par le thématicien en une proposition d'architecture logicielle. Cette activité n'implique pas que le modélisateur définisse les détails de programmation : c'est le rôle de l'informaticien. Par contre, il doit s'assurer que l'architecture qu'il propose supportera le modèle dans tous ses cas d'utilisation, sans introduire de biais. Cette transcription peut être guidée par une méthodologie qui prend en charge le problème depuis son analyse, faisant ainsi le lien entre le thématicien et le modélisateur. Selon le formalisme adopté lors de l'analyse, il se peut aussi que cette étape soit automatisée, au moins partiellement : la production de l'architecture depuis le fruit de l'analyse peut parfois se faire par l'application de règles de réecriture, de transformation de schémas...

L'approche multi-agents peut intervenir dans cette étape de deux manières différentes mais non exclusives :

- le thématicien a effectué une analyse orientée agent ;
- le modélisateur choisit une architecture multi-agents.

Il est à noter qu'une analyse orientée agent n'implique pas une implémentation selon les mêmes préceptes : les simulations orientées agents sont très souvent réalisées au moyen d'architectures orientées objet [Drogoul et al., 2003], les modélisateurs et informaticiens étant plus coutumiers de ce type d'approche. Inversement, il est tout à fait possible de choisir une architecture multi-agents pour modéliser une analyse qui n'a pas été menée sur cette base [Rodin et al., 1998].

Effectuer une modélisation orientée agent revient donc à choisir une architecture logicielle dans laquelle les agents apparaissent en tant qu'unité computationnelle.

Un tel choix n'augure en rien des méthodes d'implémentation : chaque agent pourra être réalisé à l'aide de fonctions, d'objets…
Le paradigme agent en tant que paradigme de modélisation est à rapprocher du paradigme objet. Ce dernier repose sur des notions telles que l'encapsulation de données, l'héritage, le polymorphisme, la communication par envoi de message… Il ne s'agit nullement d'un paradigme d'implémentation : il est possible de faire de la programmation orientée objet à l'aide de langages fonctionnels ou déclaratifs et logiques. Si les langages impératifs orientés objets se sont imposés comme un standard industriel *de facto*, ils ne doivent pas réformer la définition du paradigme objet : il s'agit d'un paradigme organisationnel (au sens de l'organisation des données) avant tout et le recours à une exécution impérative n'est pas corollaire.
C'est dans ce sens que la programmation orientée agent et la programmation orientée objet peuvent être comparées. Le fait que les architectures agents soient souvent implémentées à l'aide de langages orientés objets doit être considéré comme anecdotique au regard de la modélisation, bien que cela nous incite d'autant plus à la comparaison.
Plus particulièrement, les agents sont à rapprocher de ce que la communauté de l'orienté objet nomme *objets actifs* ([Booch, 1994] cité dans [Wooldridge and Ciancarini, 2000]) :

> An active object is one that encompasses its own thread of control[…]. Active objects are generally autonomous, meaning that they can exhibit some behavior without being operated upon by another object. Passive object, on the other hand, can only undergo a state change when explicitely acted upon.

En se référant point par point à la définition de l'agent proposée en début de chapitre, il apparaît que :

- les objets actifs peuvent être réactifs et proactifs ;
- ils possèdent un moyen de contrôle de leur état interne et ainsi présentent une forme d'autonomie ;
- ils interagissent *via* un mécanisme commun d'envoi de message.

Le bilan est presque identique à la description de la *notion faible* d'agent, à ceci près que :

- les objets actifs n'ont pas de moyen de contrôle sur leurs actions (il est toujours possible d'invoquer une méthode publique d'un objet) ;
- les capacités sociales des objets actifs se limitent à l'appel de méthode (sauf implémentation applicative).

La différence entre les objets actifs et les agents tient donc à la notion d'autonomie : un objet ne peut refuser, retarder ou négocier un appel de méthode. WOOLDRIDGE résume cette différence par le slogan :

- Objects do it for free ; agents do it because they want to.

Une architecture orientée agent est donc très proche d'une architecture orientée objets, mais elle doit en plus supporter nativement les interactions sociales entre agents et respecter le principe de contrôle par l'agent de ses actions (autonomie).
Cette ressemblance importante explique l'implémentation d'architectures orientées agent sur la base d'architectures orientées objet : une "*surcouche*" applicative permet aisément d'implémenter les mécanismes manquants de manière suffisamment générique pour être réutilisable.

2.1.4 De la conception au développement : l'agent unité d'implémentation

Le rôle de l'informaticien est de réaliser l'implémentation de l'architecture proposée par le modélisateur. Il doit pour cela choisir un ou plusieurs langages informatiques. Le plus souvent, ce choix est affaire d'habitude ou de contrainte environnementale : dans un souci d'interopérabilité ou de compatibilité ascendante, l'informaticien se voit imposer le langage, ou il conserve ses réflexes de travail et systématise son choix.
Cet état de fait est souvent regrettable : plus qu'une syntaxe, un langage apporte avant tout une sémantique et un modèle d'exécution. Bien qu'il soit pratiquement possible de tout faire avec tout langage, l'expressivité de chacun le prédispose à un champ d'application : il sera certainement plus commode de recourir à PROLOG pour programmer un système expert et d'utiliser JAVA pour concevoir un système client/serveur que l'inverse.

Si les différentes approches multi-agents exploitent les mêmes concepts avec la même syntaxe, la sémantique agent n'a pas de définition qui soit consensuelle. Ainsi, d'une méthodologie à l'autre, les termes *agent*, *groupe*, *croyance*… n'auront pas rigoureusement la même utilisation. Il est donc *a priori* impossible à l'heure actuelle d'écrire un langage orienté agent qui puisse être un support fédérateur de toutes ces approches.

Par langage orienté agent, nous entendons un langage pour lequel l'agent constitue une unité d'implémentation.

La plupart des environnements de développement – le terme *framework* est souvent employé – recourent à des langages impératifs orientés objet. Chaque agent est alors constitué d'un ou, plus fréquemment, de plusieurs objets.

Si l'implémentation d'un système multi-agents à l'aide d'un langage pour lequel les agents sont l'unité d'exécution peut sembler une fin logique, la pratique est sans doute plus complexe. Un seul agent – au sens du modèle – peut nécessiter l'utilisation de plusieurs

agents – au sens de l'implémentation – pour sa bonne réalisation. Ce principe est nommé *principe de récursion* [Demazeau, 1997] et certaines méthodologies se basent même sur une décomposition récursive des systèmes multi-agents [Bernon et al., 2004].

Il n'existe donc pas à l'heure actuelle de langage de programmation agent, dans le sens où les langages nouvellement créés ne permettent pas l'exécution de tous les types d'agent. Ceci ne sera de toute façon pas possible tant qu'un modèle d'exécution commun ne sera pas défini, et cette définition semble utopique – et non nécessairement souhaitable - au regard de la pluralité des modèles d'agent existants (une présentation des principaux modèles est exposée dans la suite de ce chapitre).

Deux voies sont actuellement explorées pour l'implémentation des systèmes multi-agents :
– l'extension de langages existants ;
– la création de nouveaux langages supportant un ou plusieurs types d'agents.

L'extension d'un langage en vue d'une utilisation multi-agents peut se présenter sous la forme d'un ajout de fonctionnalités propres à ce types d'architectures. Le plus souvent, cette modification prend la forme d'une bibliothèque de fonctions et n'impose aucun modèle d'exécution. Par exemple, MADKIT [Ferber and Gutknecht, 1998] est une extension de JAVA supportant l'ordonnancement des activités de plusieurs agents, des protocoles de communication... Il s'agit en fait d'un moteur de simulation multi-agents basé sur un modèle organisationnel, le modèle AALAADIN, mais qui laisse le développeur libre de créer son modèle d'agent. De même, la bibliothèque ARÉVI[4] étend le langage C++ avec des services similaires, en plus de l'intégration d'un moteur de rendu graphique de scènes tridimentionnelles. Ces approches sont très souples d'utilisation mais laissent une part de développement importante à l'informaticien.

De manière plus restrictive, certaines extensions limitent l'informaticien à un ensemble de modèles d'agents. C'est le cas par exemple de `Jack` [Busetta et al., 1999] qui ajoute le support des agents *BDI*[5] à JAVA. La perte de souplesse d'utilisation est compensée par une plus grande homogénéité et une plus grande facilité d'exploitation : le modèle d'agent étant connu *a priori*, il plus aisé d'ajouter efficacement les spécificités multi-agents au langage cible.

La dernière voie explorée est la création d'un nouveau langage, ou plus exactement d'un atelier de développement : généralement ces approches se limitent à un ensemble de modèles d'agent et permettent leur implémentation a travers un outil de conception interactif (et non d'un langage). Là aussi, la maîtrise totale du type d'agent employé permet de prévoir lors de la conception de l'outil l'intégration la plus complète des concepts manipulés. Ainsi, les applications conçues dans cette optique offre souvent une prise en

[4] **A**telier de **Ré**alité **Vi**rtuelle : cette librairie, développée au Centre Européen de Réalité Virtuelle (CERV, le laboratoire hôte de cette étude) sera présentée plus en détail dans la partie consacrée à nos réalisations.

[5] **B**elief **D**esire **I**ntention, il s'agit d'un modèle comportemental basé sur l'exploitation des notions de *croyance*, *désir* et *intention*. Ce modèle est exposé en détail à la fin de ce chapitre.

charge quasi complète du cycle de développement. Toutes les étapes, de l'analyse au déploiement, sont supportées et une validation du franchissement des transitions entre ces étapes est possible : le modèle d'agent étant prédéterminé, la formalisation complète du processus de développement est facilitée. Nous pouvons citer en exemple AGENTBUILDER [Reticular Systems, 1999] qui permet de construire des systèmes multi-agents basé sur le modèle *BDI*, ou ZEUS [Nwana et al., 1998] qui traite les problèmes qui se décomposent comme une organisation d'agents reposant sur un modèle économique.

Ces deux approches différentes d'extension de langages existants ou de création de nouveaux langages se rejoignent progressivement au rythme des processus de standardisation : la pratique permet de cerner les propriétés pertinentes des concepts de développement multi-agents. Cette expérience se concrétise par la définition de standards qui permettent aux différents intervenants du secteur de s'accorder sur l'interprétation de ces concepts en termes de génie logiciel. Ce phénomène est particulièrement remarquable au niveau de la définition de protocoles de communication entre agents (KQML [Finin et al., 1994] ou FIPA ACL [Fipa, 2001] par exemple), certainement parce qu'il s'agit d'un aspect de ces approches aussi crucial que délicat [Labrou et al., 1999].

2.1.5 Du développement au déploiement : l'agent unité d'exécution

L'utilisateur final est *a priori* le dernier maillon de la chaîne de développement : il subit les choix commis par les autres intervenants lors de l'emploi de l'application obtenue. La plupart du temps, il ignore sur quelles technologies le développement s'est appuyé : les détails d'analyse, de modélisation et d'implémentation ne l'intéressent pas. Cependant, ces choix déterminent directement l'ergonomie de l'application qu'il manipule. Nous ne traiterons pas dans cette section de l'interfaçage avec l'utilisateur – bien que là encore le recours à l'approche multi-agents soit d'actualité – mais seulement du déploiement de l'application. La plupart des applications informatiques exploitent désormais des réseaux de machines, que ce soit par nécessité de communiquer à longue distance, pour répartir la charge de l'exécution sur plusieurs postes ou pour permettre à plusieurs utilisateurs d'interagir avec l'application. Ce constat est particulièrement vérifié dans le cadre de la simulation comportementale qui nous intéresse. Ici encore, l'utilisation du paradigme agent permet de modéliser efficacement le problème. Par exemple, pour une répartition de charge de traitement sur un réseau, il est commode de considérer chaque processus de l'application à répartir comme un agent. Ce-dernier est doté de capacités de communication, d'une autonomie d'action et perçoit son environnement. Il peut donc percevoir les problèmes de surcharge et, après négociation avec les autres processus ou les nœuds du réseau, décider de sa migration vers un autre serveur. L'approche multi-agents évite donc de concevoir un algorithme global de répartition de charge et permet de doter chaque processus d'une politique de gestion des ressources. Ceci n'est qu'un exemple d'exploitation de l'approche agent en terme de déploiement. En plus des objectifs de répartition de charges, l'agent peut migrer pour satisfaire à d'autres impératifs : gestion de qualité de service sur un réseau, réplication de processus pour la tolérance aux fautes…

Dans ce cas, la notion d'agent correspond souvent à du code mobile, l'agent est une unité

d'exécution.

Ainsi, l'approche agent permet de gérer le déploiement d'applications qui ne sont pas elles-mêmes basées sur une approche multi-agents. C'est le cas par exemple des applications multi-experts qui fédèrent plusieurs systèmes experts. Cet emploi de l'approche agent est totalement transparent pour l'utilisateur : c'est la plateforme qui gère le déploiement des applications. Cependant, dans certains cas l'utilisateur n'est pas ignorant du cycle de développement : il peut s'agir de la même personne que le thématicien, le modélisateur ou l'informaticien qui cherche à valider son modèle par la manipulation de l'application finale. En effet, le cycle de développement a ici été présenté de manière très abstraite. Dans la pratique, les experts n'interviennent pas à tour de rôle dans une séquence d'étapes indépendantes : les "*aller-retours*" entre ces phases sont nombreux afin d'affiner et de valider la transcription que chacun effectue de l'expertise de l'autre (cycles "*en V*", cycle "*en spirale*"...). Lorsque l'application est conçue comme une organisation d'agents, il est souhaitable de retrouver ces agents au niveau du déploiement : l'utilisateur peut désirer interagir avec un agent précis durant l'exécution même de l'application.
Le cas de figure le plus extrême de ce genre d'interaction est le prototypage interactif : dans le cas de modélisation comportementale par exemple, il est souhaitable de pouvoir modifier le comportement d'un agent alors que la simulation est en cours [Rodin et al., 2000]. En effet, pour certains types de modèles, la crédibilité du comportement obtenu peut dépendre de paramètres impossibles à définir exactement hors situation. Ce n'est donc que pendant la simulation que certains mécanismes peuvent être ajustés avec pertinence. Pour être homogène, une plateforme conçue pour la simulation de systèmes multi-agents doit donc offrir les services nécessaires à ce genre d'interaction (granularité instance, service de nommage, introspection...).
Dans [Drogoul et al., 2003], l'auteur dégage quatre points communs à tout type d'application qui motivent l'emploi de systèmes multi-agents pour l'implémentation de simulations orientées agents :

- la conception participative : il s'agit d'une forme de prototypage interactif où l'expert ne modifie pas directement le modèle de l'agent, mais "*joue*" le rôle que celui-ci doit tenir dans la simulation, l'agent apprenant alors en ligne son comportement à partir de cette "*démonstration*" de l'expert ;
- l'interprétation assistée par agent : chaque agent peut être doté de capacités d'interprétation locale et d'explication, facilitant l'analyse d'interactions complexes au sein d'un système multi-agents ;
- l'adaptation au niveau du système : la distribution physique de l'application peut être gérée par les agents eux-mêmes, permettant l'automatisation de mécanismes d'adaptation à l'état du réseau au cours de la simulation ;
- la calibration assistée par agent : les agents peuvent "*participer*" au calibrage de la simulation, qui se ramène alors au calcul distribué d'une optimisation multi-critères.

2.2 Agents autonomes situés et réalité virtuelle

Depuis l'entité conceptuelle jusqu'au code mobile, nous avons évoqué plusieurs formes possibles du concept d'agent qui sont toutes conformes à la *notion faible d'agent* décrite précédemment. Derrière un vocabulaire et des concepts communs se cachent donc des points de vue très différents et autant d'applications du paradigme agent. Afin de délimiter au mieux notre cadre d'étude, nous allons maintenant considérer ces notions sous l'éclairage particulier de l'utilisation d'agents autonomes situés en réalité virtuelle.

2.2.1 La réalité virtuelle

Nous adopterons du terme *réalité virtuelle*[6] la définition donnée dans [Fuchs et al., 2003] :

> La réalité virtuelle est un domaine scientifique et technique exploitant l'informatique et des interfaces comportementales en vue de simuler dans un monde virtuel le comportement d'entités 3D, qui sont en interaction en temps réel entre elles et avec un ou des utilisateurs en immersion pseudo-naturelle par l'intermédiaire de canaux sensori-moteurs.

Les applications couvertes vont donc du jeu vidéo à la simulation comportementale en passant par le théâtre virtuel ou les environnements virtuels de formation.
Sans détailler complètement cette définition, nous pouvons relever certains points qui devrons être systématiquement pris en compte dans la suite de cette étude :

- utilisation d'*interfaces comportementales* : les techniques de réalité virtuelle exploitent différentes sortes d'interfaces (gants de données, capteurs de position, caméras…) afin de faciliter des interactions complexes avec l'utilisateur ;
- le but est de *simuler dans un monde virtuel le comportement d'entités 3D* : la modélisation comportementale d'entités en interaction est centrale à ce domaine ;
- gestion d'interactions *en temps réel* : l'utilisateur étant impliqué dans la simulation, les réponses des mécanismes comportementaux impliqués doivent survenir à une fréquence suffisante pour rester crédibles ;
- l'utilisateur est en *immersion pseudo-naturelle* : la qualité de rendu doit favoriser l'immersion de l'utilisateur, qu'il s'agisse du rendu visuel, sonore ou de toute autre nature.

La notion de temps réel pourrait ici être qualifiée de *temps réel faible* : l'objectif n'est pas de traiter les données en fonctions de leur pérennité (temps réel "*dur*") mais en fonction de la manière dont elles sont perçues (leur pertinence). Par exemple, un brin d'herbe qui se

[6] Le terme *réalité virtuelle* provient de l'expression anglaise *virtual reality*. Cependant, le passage en français est tombé dans le piège du "*faux ami*" : le terme *virtual* en anglais signifie *de fait, pratiquement* alors que le terme *virtuel* en français signifie *possible, potentiel* ou *latent*. L'adoption de ce qui est parfois considéré (exagérément) comme un oxymoron en français nuit à la lisibilité du domaine, cet écart est certainement aussi regrettable que le choix de l'expression *intelligence artificielle* qui a aussi été critiquée depuis sa création. Les implications de cette traduction maladroite ont été traitées dans la littérature, par exemple dans [Fuchs et al., 2003].

balance dans le vent prend une infinité de positions en une seconde, mais dans le cadre de la réalité virtuelle, la détermination d'une soixantaine de positions en une seconde suffira à générer une animation fluide et crédible pour l'utilisateur. Si ce même brin d'herbe est en arrière-plan de la scène visualisée, une fréquence plus faible ou des approximations de position pourront passer totalement inaperçus. La problématique est donc bien de modifier l'environnement conformément aux moyens de perception de l'utilisateur (de manière crédible), et non dans le respect strict des lois d'évolution (de manière réaliste).

La nature *individu-centrée* de la réalité virtuelle appelle naturellement le paradigme agent, aussi bien pour l'analyse que pour la modélisation, l'implémentation ou le déploiement des applications.
Par la suite, nous nous concentrerons essentiellement sur les phases de modélisation et d'implémentation : quelle que soit la décomposition proposée par le thématicien, elle se ramènera dans le cadre de la réalité virtuelle à une "*distribution*" de rôles à des agents au sein d'une organisation, chaque agent représentant ici une entité de la simulation. Le point central qui demeure est donc la définition de modèles de comportements d'agents adaptés à ce type d'application, et la conception de modèles d'exécution capables de supporter de telles simulations. Cependant, dans l'élaboration de ces modèles une attention particulière doit être portée au maintien d'interfaces efficaces avec les autres étapes du développement : la modélisation et l'implémentation doivent être réalisées avec la conscience des impératifs de l'analyse et du déploiement.

L'objectif principal de notre étude sera donc de proposer un modèle comportemental ainsi qu'un modèle d'exécution pouvant le supporter qui maintiennent tous deux un lien sémantique fort avec les autres étapes du cycle de développement.

Non seulement l'approche multi-agents est appropriée à la réalité virtuelle, mais les autres types d'approche semblent voués à l'échec par manque d'adaptabilité et de dynamisme. Dans une telle application l'environnement est intrinséquement ouvert : des entités peuvent apparaître ou disparaître à tout moment ou perdre leur autonomie pour tomber sous le contrôle d'un ou plusieurs utilisateurs. Le découplage entre les entités constitutives du système doit donc être suffisant pour autoriser de telles modifications dynamiques de l'organisation. De plus, l'environnement lui-même est modifié perpétuellement par ses interactions avec les entités qui le peuplent. Les capacités de perception et d'autonomie d'action sont donc incontournables pour l'implémentation de systèmes ouverts. Tous ces problèmes trouvent une réponse avec l'approche multi-agent.

2.2.2 L'émergence de comportement

Le recours à de telles approches n'est pas uniquement une solution structurelle mais aussi computationnelle. Non seulement les architectures logicielles orientées agent offrent la souplesse et le dynamisme recherchés en réalité virtuelle, mais les modèles comportementaux permettent en plus de doter les entités simulées de comportements qui ne sauraient être définis d'une autre manière. Le paradigme agent, en explorant les effets résultant d'ensembles

complexes d'interactions, permet d'observer au niveau du système des comportements qui ne sont pas programmés explicitement au niveau des agents. C'est cette observation au niveau du système (niveau macro) d'un comportement non explicité au sein des entités qui le constituent (niveau micro) qui est nommée *émergence*. L'exemple traditionnel de ce phénomène dans la litérature est celui du comportement de fourmis : des fourmis qui vont et viennent entre une source de nourriture et leur nid empruntent le même chemin. Ce comportement que nous interprétons au niveau macro en termes topologiques (*le même chemin*) peut être le fruit des interactions de nombreuses fourmis qui "*ignorent*" tout du concept de *chemin* au niveau micro : dans leurs déplacements les fourmis laissent derrière elles un marquage chimique – des phéromones – qu'elles perçoivent et qui les guident à leur tour, renforçant ainsi leur attraction pour un chemin commun qui "*émerge*" des différents chemins possibles entre ces deux points. Un tel mécanisme est employé comme métaphore de programmation dans [Rodin et al., 1998] pour un traitement d'image : des "*fourmis*" se déplacent librement sur une image et laissent un marquage chimique lorsqu'elles sont confrontées à des conditions particulières, notamment de contraste. Chaque entité ne perçoit donc son environnement que localement et en termes de couleurs, sans connaissance de ce qu'est une image. Pourtant, avec un choix judicieux de conditions de marquage et de règles d'exploitation de ces "*phéromones*" pour guider les déplacement des entités, les auteurs parviennent à réaliser un traitement de l'image dans sa globalité : l'observateur constate l'émergence d'un traitement de signal au niveau macro sans qu'aucune notion d'image globale ne soit manipulée au niveau micro.

Ainsi, en reprenant la définition de JEAN-PIERRE MÜLLER dans [Müller, 2002], nous dirons qu'un phénomène est émergent si :

- il y a un ensemble d'entités en interaction dont la dynamique est exprimée dans un vocabulaire ou théorie D distincte du phénomène émergent à produire ;
- la dynamique de ces entités interagissantes produit un phénomène global qui peut être un état structuré stable ou même la trace d'exécution ;
- ce phénomène global peut être observé et décrit dans un vocabulaire ou théorie D' distincte de la dynamique sous-jacente.

Ces propriétés de la dynamique des interactions d'un système complexe sont donc intéressantes lorsque le phénomène observé ne relève pas de la même théorie que celle qui sous-tend la modélisation des comportements des entités impliquées : ceci laisse entrevoir la possibilité d'effectuer des traitements comme résultats d'interactions locales aisément spécifiables, alors qu'il est difficile ou impossible d'analyser le problème dans sa globalité pour le résoudre comme "*un tout*". Cette approche est souvent résumée par l'adage : "*le tout est plus que la somme des parties*". Si les concepts impliqués étaient les mêmes au niveau micro et au niveau macro, il ne s'agirait que d'une forme de décomposition.
Dans le cadre de la réalité virtuelle, l'intérêt des phénomènes émergents n'est pas étudié qu'au niveau du système : comme décrit précédemment, par le principe dit de récursion, une entité d'un environnement virtuel peut elle-même être implémentée à l'aide d'un système multi-agents. Dans ce cas, le comportement de l'entité émerge des interactions des agents qui la composent. Une telle architecture permet de simuler des comportements dont aucun

algorithme global ne peut être donné. Il peut s'agir de concurrence, de compétition, de négociation ou tout autre type d'organisation qui s'instaure entre des éléments constituant le modèle comportemental de l'entité (buts, plans d'action, rôles...). Chaque entité est ainsi vue comme un système complexe. Cette décomposition est souvent utilisée pour la simulation de comportements cognitifs évolués pour lesquels il est impossible de construire un modèle computationnel générateur, les expertises dans ce domaine étant soit insuffisantes, soit impossibles à retranscrire. A ce propos, FETZER évoque les limites du computationnalisme en ces termes ([Fetzer, 1994] cité par [Bringsjord, 1998]) :

> [T]he computational conception appears to have arisen from the almost irresistible temptation to appeal to a domain about which a great deal is known (formal systems and computability theory) as a repository of answers to questions in a domain about which very little is known (the nature of mentality and cognition).

Ainsi les systèmes multi-agents sont parmi les outils les plus appropriés à l'exploration des théories qu'ils exploitent. Cette réflexivité entre l'évolution de l'outil et celle de la théorie ne fait qu'augmenter la difficulté d'investigation scientifique sur la base des simulations obtenues. Toute mesure est perturbée par l'instrument qui l'opère.
Dans [Crutchfield, 1994], l'auteur expose clairement le rôle de l'émergence dans le processus de découverte scientifique et les relations entre ce concept et les notions de mesure de complexité dans la modélisation des phénomènes non linéaires.

2.2.3 La situation de l'agent : une notion subtile

Si le recours aux systèmes multi-agents apporte des réponses à certaines difficultés de la réalité virtuelle, ce champ d'application contraint réciproquement l'implémentation de tels systèmes. En plus des interactions entre les agents qui le constituent, le système doit prendre en charge les interactions entre chaque agent et son environnement. Ces dernières se décomposent en trois catégories :

- les perceptions de l'environnement par l'agent ;
- les actions de l'agent sur l'environnement ;
- les actions de l'environnement sur l'agent.

Nous reprenons pour définition du terme *environnement* celle employée par OLIVIER BOISSIER dans [Boissier, 2001] :

> Nous définirons rapidement l'*environnement* comme le médium commun aux agents du système.

Ce médium est donc le support de communications indirectes (par "*marquage*" de l'environnement). La communication directe entre agents – par envoi de message – est souvent considérée séparément des autres types d'actions, alors que certains modèles de communication se basent sur la théorie des *actes de langages* et sont donc essentiellement des actions [Austin, 1962]. La discrimination entre action et communication est alors souvent ambigüe. Les implications de cette confusion et les différentes formes de communication entre agents sont détaillées dans [Chicoisne, 2002]. Par communication indirecte, nous entendons la prise

en compte par un agent des modifications de son environnement dûes aux actions des autres agents. Nous nous attachons donc dans un premier temps aux perceptions de l'agent : celles-ci sont indispensables à l'autonomie d'action. L'agent peut *prendre l'initiative* parce qu'il perçoit son environnement, ce qui détermine son comportement : il privilégie des actions pour répondre à la dynamique de son environnement. Cette adaptation peut se faire en réaction ou par anticipation sur l'évolution de son environnement, selon le modèle comportemental qui anime l'agent (la notion de *comportement* est détaillée dans la section suivante).

C'est la perception de son environnement – éventuellement partielle ou erronée – qui fonde l'autonomie de l'agent. C'est pourquoi un agent immergé dans un environnement virtuel est qualifié d'*agent situé*.
Il faut comprendre le terme *agent situé* comme *agent en situation* : le mot *situé* est souvent pris dans le sens de *localisé*. C'est une approximation qui cache la complexité de cette notion : un agent peut être *localisé* sans être *situé dans l'espace*. Pour saisir la nuance, considérons un agent simple : une montre. Cette dernière perçoit les interactions simples que sont les manipulations de ses boutons (mise à l'heure, réglage d'une alarme...) et agit en réaction (selon un modèle très simple !). Quel que soit l'endroit où se trouve cette montre dans l'environnement virtuel, elle indiquera l'heure ou sonnera de la même manière : son comportement n'est pas modifié en fonction de sa position, elle n'est pas *située dans l'espace*. Cependant, une entité qui souhaite consulter l'heure ou régler une alarme sur cette montre devra se trouver à proximité : la montre est *localisée* dans l'environnement.

Un agent situé est donc un agent qui prend en compte sa situation, et non pas une entité qu'un observateur peut situer dans l'environnement.

L'exemple proposé ici est d'autant plus trivial qu'il traite de la situation d'un agent par rapport à une notion sur laquelle la définition d'une métrique est naturelle (l'espace). La nuance devient plus subtile quand la situation s'entend par rapport à un concept plus abstrait. Pour illustrer cette affirmation, considérons les modèles d'agents émotionnels : dans le but de simuler des comportements sociaux, certains modèles comportementaux exploitent des notions telles que l'envie, la nervosité, la peur... Un agent qui exploiterait *en interne* de telles notions pour sélectionner ses actions, sans prendre en compte les émotions des agents avec lesquels il interagit, ne serait pas *émotionnellement situé*. A l'inverse, une autre type de comportement pourrait consister pour un agent à prendre en compte les perceptions des émotions des autres entités comme entrées d'un modèle de sélection d'action qui ne repose absolument pas sur la gestion de ses propres émotions. L'agent mû par un tel modèle serait alors *émotionnellement situé*, sans être lui-même un agent *émotionnel*.

2.2.4 L'autonomie de l'agent : un but ou un moyen ?

Ainsi, la perception par l'agent de son environnement local au moyen de capteurs virtuels [Thalman et al., 1997] est le vecteur de son autonomie. Le terme *autonomie* s'entend

ici dans son sens premier, comme *le droit de se gouverner par ses propres lois*[7]. C'est pourquoi l'expression *autonomie d'action* était employée précédemment : l'autonomie de l'agent signifie que celui-ci "*décide*" de ses actions. Ceci n'implique pas une indépendance totale de l'agent, particulièrement l'agent n'a pas d'autonomie d'exécution : il est tributaire de la plateforme qui supporte son exécution. Le mot *autonomie* n'est pas à prendre dans le sens d'*autosuffisance* comme dans le cas de l'autonomie d'un véhicule.

Au début de ce chapitre, l'autonomie des agents a été présentée comme une réponse aux problèmes de flexibilité, de distribution, de réutilisabilité… Cette notion apparaît alors comme la clef de voûte des approches multi-agents et la recherche d'une autonomisation des modèles numériques semble le propos de fond de l'étude de telles approches [Tisseau, 2001].

Si cette analyse s'impose dans l'abord des différentes approches multi-agents, les auteurs de [Aylett and Luck, 2000] proposent un point de vue très différent. Dans cet article ils s'intéressent avant tout à la problèmatique des environnements virtuels, les systèmes multi-agents n'étant alors considérés que comme un outil adéquat à l'implémentation, et non nécessairement comme le centre d'intérêt premier de l'étude. Dans cette optique, l'informaticien chargé des développements dispose d'une maîtrise totale des agents et de l'environnement dans lequel ils sont immergés. Doter un agent de la capacité de manipuler une information de cet univers virtuel est alors à la discrétion de ce spécialiste, la création d'agents omniscients ne pose aucun problème technique. Or, nous avons exposé deux voies principales conduisant à l'utilisation de systèmes multi-agents :

- le problème traité est trop complexe pour être abordé selon une approche globale, les agents sont alors un moyen de décomposition et de distribution ;
- l'application implique la simulation d'entités, cette approche individu-centrée trouve un écho naturel et logique dans la modélisation multi-agents.

Dans le premier cas, la décomposition du problème comme une organisation d'agents est un recours : face à l'incapacité d'établir un modèle global, le développeur utilise un système multi-agents, ce qui le conduit à une décomposition appropriée. Pour la deuxième alternative, la crédibilité souhaitée impose des modèles autonomes : une entité qui perçoit tout et sait tout de son environnement ne peut avoir un comportement crédible, ses perceptions doivent être limitées et faillibles pour sembler "*naturelles*".
Quelles que soient les raisons qui poussent à recourir aux systèmes multi-agents, l'autonomie peut donc être considérée comme un moyen, et non comme un but.

La volonté d'accroître l'autonomie des agents est donc le but de la recherche *sur* les systèmes multi-agents, alors que cette même autonomie est plutôt un moyen pour la recherche *avec* les systèmes multi-agents. Ce second point de vue a pour intérêt de mettre l'accent sur les limitations des développements actuels de la théorie agent. Dans ce même article, il est par exemple question de la prise en compte des lois physiques par les agents : une loi comme la gravité peut être codée au sein du comportement de chaque agent, ou directement au niveau de la plateforme dans la gestion de l'environnement. Cette seconde solution pose

[7] Premier sens de la définition donnée dans *Le petit Robert, dictionnaire de la langue française.*

le problème de l'élaboration de protocoles d'interactions, alors que la première conduit à des compromis de développement qui nuisent à la réutilisabilité : comment devrait se comporter un poisson s'il était placé dans un environnement virtuel qui n'était pas rempli d'eau ?[8]

Malgrè les efforts dans cette direction ([D'Inverno and Luck, 2003] par exemple), le parcours du chemin qui mène à l'unification des systèmes multi-agents et l'interopérabilité totale des différentes applications sera certainement encore long.

2.3 Agents autonomes situés et comportements

Après avoir éclairci les notions d'agents, de systèmes multi-agents, d'autonomie et de situation dans un environnement virtuel, il nous reste à définir les concepts de *comportement* et de *spécification* de comportement pour décoder dans sa totalité l'expression : *spécification de comportements d'agents autonomes situés dans un environnement virtuel*.

2.3.1 Aspect externe de l'agent : les comportements émergents

Ici encore, nous prendrons le temps de clarifier le vocabulaire employé afin de cerner les concepts impliqués dans notre réflexion. Au premier chef, le *comportement* d'une entité désigne ses agissements, sa manière de réagir. L'idée d'*étude du comportement* a donc une connotation behaviouriste : elle relève de l'analyse des actions commises par l'entité observée et non de la considération des mécanismes – physiologiques ou mentaux – qui engendrent ces actions. Il s'agit d'une observation de l'aspect externe de l'agent.
La qualification du comportement d'une entité n'augure normalement en rien de sa structure interne. Par exemple, chacun est libre de considérer que les distributeurs de boissons automatiques sont des entités intelligentes très sympathiques : si vous leur donner suffisamment d'argent pour couvrir les frais de la transaction, elles vous délivrent en échange une boisson fraîche de votre choix et vous rendent le change le cas échéant. Ce modèle, qui doit raisonnablement être considéré erroné, *explique* parfaitement le *comportement* d'un distributeur automatique. Heureusement, ce comportement est *généré* par des moyens électro-mécaniques simples, évitant ainsi l'enfermement d'entités intelligentes sympathiques !

Le fait que le comportement qualifie l'aspect externe d'une entité introduit une ambiguïté dans le cas d'observations de comportements émergents[9]. Dans [Brassac and Pesty, 1996] SYLVIE PESTY et CHRISTIAN BRASSAC montrent qu'un comportement de coopération peut être prété à un système multi-agents composé d'agents qui n'exploitent aucune méthode de coopération. Les approches individu-centrées appellent parfois à tord des interprétations antropomorhiques. Dans cet article, les auteurs discutent les dangers de l'emploi d'un vocabulaire impropre ou imprécis dans les approches multi-agents et suggèrent d'ailleurs l'emploi du mot *conduite* pour désigner les mécanismes internes à l'agent qui génèrent son compor-

[8] Exemple proposé par [West and Hubbold, 1998] cité dans [Aylett and Luck, 2000].

[9] Il faut évidemment comprendre ici l'expression *observer un comportement* comme *être spectateur d'un comportement*. Cette expression pouvant aussi s'entendre dans un autre contexte comme *endosser un comportement*, le doute devait être levé.

tement.

Pour la suite, nous sacrifierons aux habitudes de la communauté en employant l'expression *modèle comportemental* pour désigner le modèle de *génération* de comportement d'un agent (*conduite*) et non un modèle explicatif de ses agissements.

2.3.2 Aspect interne de l'agent : les modèles comportementaux

Le but de cette section n'est pas de proposer un "*bestiaire*" exhaustif des modèles comportementaux. Le lecteur intéressé pourra se tourner vers les panoramas très complets existants [10]. Notre dessein est de reprendre quelques critères de taxonomie parmi les plus consensuels afin de présenter les grands "*courants*" de modélisation et ainsi situer notre proposition dans les chapitres à venir.

Nous avons ainsi retenu les trois catégories les plus courantes et traité en plus le cas particulier des architectures *BDI* séparément.
Ces trois catégorie identifient le modèle de raisonnement utilisé. Seul l'aspect interne de l'agent est considéré ici. Nous étudions ainsi la manière dont les perceptions de l'agent mènent à ses actions. Les trois familles de modèles présentées dans la suite de cette section sont :

- les agents délibératifs ;
- les agents réactifs ;
- les agents hybrides.

Cette classification permet d'exposer les principales approches, mais elle ne prétend pas suffir à qualifier exactement tous les travaux dans le domaine. Un flou subsiste parfois entre le modèle comportemental proposé – par le modélisateur – et l'implémentation qui en est faite – par l'informaticien – qui rend cette taxonomie délicate. De nombreux autres critères devraient être considérés pour un classement exhaustif. Par exemple l'architecture logicielle peut différer pour un même modèle comportemental de base, ce qui génère des comportements différents. La classification proposée par [Müller et al., 1995], et reprise dans [Boissier, 2001], distingue ainsi les architectures logicielles selon la topologie des différentes "*couches*" qui les composent (architectures horizontales, verticales modulaires, verticales en couches ...). De même, dans [Boissier, 2001] OLIVIER BOISSIER prend en compte le type de coordination entre agents comme second axe d'étude pour subdiviser chaque catégorie (agent réactif, délibératif ou hybride) en trois autres. Il distingue ainsi trois modèles de coordination entre agents :

- les agents autonomes ;
- les agents interagissants ;
- les agents sociaux.

Il aurait sans doute été préférable de renommer la catégorie *agent autonome* en *agent indifférent* afin d'éviter les ambiguïtés : tous les modèles comportementaux concernent ici des

[10] ([Wooldridge and Jennings, 1995a, Nwana, 1995, Pirjanian, 1999, Wooldridge and Ciancarini, 2000, Arnaud, 2000, Boissier, 2001, Donikian, 2004] par exemple)

agents *autonomes* au sens décrit dans la section 2.2.4. L'auteur classe dans cette catégorie de coordination les agents qui ne prennent pas en compte les autres entités impliquées dans la simulation, contrairement aux agents interagissants. Quant aux agents sociaux, ils communiquent sur leurs actions et sont capables de collaboration ou de négociation par exemple.

La revue des différentes approches selon ces deux axes orthogonaux par OLIVIER BOISSIER – le modèle de raisonnement de l'agent d'une part, et le type de coordination entre les agents du système d'autre part – apporte une vision claire et synthétique d'un domaine très vaste.

Les agents délibératifs

Le chapitre d'introduction a mis en évidence la parenté entre l'intelligence artificielle et les approches multi-agents. Cette forte connexité trouve son prolongement dans l'étude des modèles d'agents dits *délibératifs*. En effet, la décomposition d'un problème en système multi-agents et sa résolution par les effets combinés des interactions de ces agents correspond à une branche particulière de l'intelligence artificielle : l'intelligence artificielle distribuée. Son propos est l'étude de la distribution d'algorithmes de résolution de problèmes, la définition de comportements d'agents en interaction peut donc se ramener à cette activité.
Cette approche est riche de l'héritage de l'intelligence artificielle classique et reprend les principales théories logiques comme autant de modèles comportementaux. Le principe générique de ces approches est la représentation au sein de l'agent de son environnement sous forme symbolique, et le traitement de ces connaissances pour le déclenchement des actions. Ce traitement peut être arbitrairement complexe et reposer sur différents types de logiques : logique du premier ordre, modale, temporelle, déontique...
Dans [Wooldridge and Jennings, 1995a] WOOLDRIDGE et JENNINGS définissent les agents délibératifs comme suit :

> We define a deliberative agent or agent architecture to be one that contains an explicitly represented, symbolic model of the world, and in wich decision (for example about what action to perform) are made via logical (or at least pseudo-logical) reasoning, based on pattern matching and symbolical manipulation.

Ces approches se heurtent principalement à deux difficultés majeures :
– le problème de transduction ;
– le problème de représentation.

Le problème dit *de transduction* est un problème d'adaptation symbolique. Sans se soucier, dans un premier temps, de la méthode exacte de raisonnement utilisée par l'agent, celui-ci doit prendre ses décisions en fonction de son environnement et agir sur ce-dernier. Il lui faut donc "*traduire*" les informations perçues par ses capteurs virtuels en un ensemble de symboles qui puissent être support de son raisonnement. Cette "*transduction*" doit se faire en temps réel pour que les décisions correspondent à la situation courante de l'agent. Cette difficulté est à la fois d'ordre théorique et d'ordre pratique. Premièrement d'ordre théorique

car il incombe au modélisateur de choisir un mécanisme de conversion opportun entre les informations brutes qui sont des valeurs numériques et un ensemble de symboles. Cette adaptation peut être générique, avec le recours à la logique floue par exemple, ou peut être prévue dans la modélisation de chaque capteur virtuel. Deuxièmement, c'est une difficulté d'ordre pratique : cette conversion doit être faite en temps réel. Ici encore, la notion de temps réel peut être qualifiée de *temps réel faible* : les perceptions de l'agent doivent être traitées dans un délai qui varie en fonction de leur nature. Par exemple, l'interprétation de l'émotion d'un interlocuteur sera pertinente si elle est effectuée dans un délai qui est de l'ordre de la seconde. Par contre, l'interprétation de la position d'un obstacle en vue d'un évitement doit se faire beaucoup plus vite (selon les vitesses respectives de l'agent et de l'obstacle).
Dans [Newell, 1990] l'auteur définit quatre "*bandes de temps*" principales dans l'activité humaine qui sont souvent prises comme points de repère pour situer les contraintes de temps sur les systèmes de raisonnement :

- la bande biologique : ALLAN NEWELL situe entre 100μ s et 10ms l'activité au niveau neuronal (transmission de signaux élémentaires) ;
- la bande cognitive : entre 100ms et 10s, elle correspond aux perceptions et actions simples ou composées d'actions simples (une tâche complexe déjà connue) ;
- la bande rationnelle : de la minute à quelques heures, elle implique l'exploitation par l'individu de sa mémoire, de sa représentation du monde et de ses capacités de raisonnement pour atteindre ses buts ;
- la bande sociale : la gestion par l'individu de ses relations aux autres se fait beaucoup plus lentement (de l'ordre de la journée à plusieurs années).

Bien que ce découpage ait été effectué par analyse du comportement humain, il donne de bonnes indications des contraintes à respecter pour obtenir des comportements crédibles en simulation. Les tâches de la bande biologique ne sont généralement pas simulées directement. Il apparaît de même que la simulation de la couche sociale ne devrait pas poser de problème majeur. Par contre, l'échelle de temps de la bande cognitive illustre la problèmatique de la transduction : quelle que soit la complexité de la scène et le nombre d'agents impliqués, chacun doit pouvoir prendre en compte ses perceptions avec un délai de l'ordre du dixième de seconde pour réagir "*à temps*" de manière crédible. L'emploi d'un modèle de perception ou d'un mécanisme de transduction trop complexe peut donc devenir un facteur limitant du modèle d'exécution. De plus, la détermination des informations pertinentes dans une situation, et surtout leur pérennité, est totalement contextuelle : ceci est connu sous le nom de "*frame problem*". Chaque décision doit être prise "*dans un cadre donné*". Il n'est pas évident de déterminer quelle partie des informations sur l'environnement doit être mise à jour entre deux décisions successives. De même, s'il faut revenir sur une décision qui a mené à un échec, toutes les informations exploitées pour cette décision ne sont pas obligatoirement à remettre en cause. Ici encore, faire la séparation entre ce qui peut être conservé et ce qui ne le peut pas est une difficulté majeure des approches symboliques. En plus de l'aspect syntaxique de la transduction, il faut donc prendre en compte les aspects sémantiques et temporels de la perception pour déterminer quelles informations acquérir et quand.

D'autre part, la considération de la bande d'activité dite *bande rationnelle* nous amène à la deuxième difficulté majeure de ce type d'approche : le problème de représentation. En

partant du principe que la "*traduction*" se fait suffisamment rapidement entre les perceptions et la partie raisonnement, il reste à fixer l'ensemble des symboles à manipuler. Cet "*alphabet*" doit être suffisant pour représenter le plus possible de situations tout en étant suffisamment restreint pour qu'une logique de manipulation soit définissable. Les décisions prises par l'agent ne seront que le fruit des opérations effectuées sur cet ensemble de symboles. Cela revient à définir "*une grammaire*" sur cet "*alphabet*", un *logos* : le langage de la pensée de l'agent. La spécification de cet ensemble de symboles peut se faire *ad hoc*, de manière pragmatique et expérimentale. Mais alors le problème de la transduction se pose dans l'autre sens : il faut être capable de repasser du domaine des symboles au domaine de l'action, en sélectionnant une action qui corresponde à la décision prise. La sémantique des symboles manipulés doit donc être suffisante pour établir un lien de la décision à l'action. C'est pourquoi la plupart des approches symboliques restent dans la lignée de l'intelligence artificielle en exploitant des logiques aussi génériques que possible. Il peut s'agir d'une logique de l'action, par exemple avec le modèle *PRS*[11], ou, plus récemment, des logiques épistémiques[12] ou déontiques[13] avec les systèmes *BDI* par exemple. Les approches basées sur une logique de l'action participent d'un mouvement au sein des approches symboliques : la planification. En effet, beaucoup de modèles génèrent le comportement comme le fruit d'un processus de planification des actions nécessaires à l'accomplissement d'un but.

Chaque modèle possède des limites et se concentre plus particulièrement sur un type de problème. La difficulté est qu'il ne s'agit plus alors de décrire *ce que fait* l'agent mais *ce qu'est* l'agent. La question centrale est donc la définition de l'ontologie de l'agent : selon le domaine abordé, il faut être capable de définir *a priori* la totalité des concepts susceptibles d'être impliqués dans le raisonnement de l'agent afin de délimiter l'ensemble minimal des symboles correspondants.

Certaines approches reposent sur des processus d'apprentissage symbolique afin de limiter le corpus de base et de s'appuyer sur des capacités d'extension automatique de l'ensemble de symboles.

Ces différents type d'approche, qui ne sont pas détaillés ici[14], se heurtent tout de même à un problème de temps : l'accroissement de la complexité des mécanismes de raisonnement symbolique fait exploser les temps de traitement. La bande de temps rationnelle semble à première vue "*confortable*" avec une référence de l'ordre de la minute, pourtant cette contrainte peut être difficile à respecter pour des simulations impliquant de nombreux agents basés sur des modèles symboliques complexes. De plus, prendre la minute comme référence implique que seules les tâches de raisonnement sont prises en charge par le modèle, il faut alors assurer la gestion des actions simples par un autre mécanisme. Si le modèle de raisonnement symbolique n'est pas doublé d'un tel mécanisme secondaire, le raisonnement doit "*descendre*" au niveau de la couche cognitive, et donc s'exécuter dans un temps de l'ordre de la seconde.

[11] **P**rocedural **R**easoning **S**ystem, système de planification basé sur l'expression des pré- et post-conditions associées aux actions [Georgeff et al., 1985]. Ce modèle est la base de nombreux modèles de planification tels que *dMars* par exemple [D'Inverno et al., 1997].

[12] Logiques des connaissances.

[13] Logiques des obligations.

[14] La revue détaillée des approches symboliques sortirait largement du cadre de cette étude, le lecteur intéressé trouvera dans [Wooldridge and Jennings, 1995a] ou [Boissier, 2001] des points de départ utiles à une investigation approfondie.

Les agents réactifs

Pour faire face aux problèmes de transduction et de représentation, certaines approches ont pris un parti radicalement différent en choisissant d'éliminer toute représentation interne de l'environnement. Les modèles issus de cette démarche sont dits *réactifs* : ils sont basés sur une liaison très directe entre les grandeurs perçues en entrée et le déclenchement des actions (l'analogie est souvent faite avec le mécanisme *stimuli/réponses* des être vivants). La figure la plus emblématique de ces approches est sans doute RODNEY BROOKS qui critiqua vivement les approches symboliques dans les années 80, soutenant qu'aucune représentation symbolique n'est nécessaire pour générer des comportements intelligents, l'intelligence étant selon lui une propriété émergente de certains systèmes complexes[15]. En 1985, il proposa une architecture nommée l'*architecture à subsomption* [Brooks, 1986] : ce modèle est basé sur une hiérarchie de couches logicielles dotées de capacités de forçage et d'inhibition de tâches sur les couches inférieures. Cette architecture a permis de réaliser des robots mobiles faisant montre de comportements évolués, que certains croyaient impossibles à implémenter sans représentation de l'environnement (comportements d'exploration avec évitement d'obstacle par exemple).
Ces approches trouvent un enrichissement mutuel avec la recherche en robotique mobile. Pour des questions de robustesse, la plupart des applications de robotique mobile privilégient les techniques de sélection d'action aux raisonnements complexes. Dans le cas d'un robot, de nombreux aléas viennent perturber la prise de décision : défauts de mesure des capteurs, irrégularités du terrain, usure des pièces… Dans ce contexte, il est préférable de recourir à ce genre d'approche où les comportements sont "*câblés*" : les concepteurs relient directement certaines situations (des valeurs de capteurs) à l'exécution de certaines actions (activation des effecteurs). Ces architectures répondent non seulement au besoin de robustesse, mais aussi aux contraintes de temps de réponse : sur un système réel, il faut garantir les temps de sélection d'action en fonction de la mécanique du robot.
Les approches réactives sont donc basées sur un couplage très direct entre les perceptions et les actions, par le biais de relations d'inhibition/forçage dans le cas de l'architecture à subsomption, ou grâce à la gestion de seuils d'activation comme dans l'architecture proposée par PATTIE MAES [Maes, 1989] pour ne citer que les plus connues[16]. Les lois régissant ce couplage étant connues *a priori*, il est plus aisé de valider un modèle de comportement par une méthode formelle (prouver l'absence d'interblocage, la décidabilité de toute situation perçue…).

Evidemment, si les approches réactives présentent des propriétés intéressantes à l'exécution (robustesse et rapidité essentiellement) elles admettent des limitations importantes. La première d'entre elle est la difficulté d'extension : la modification du comportement nécessite d'intervenir sur l'implémentation du modèle la plupart du temps, les liens entre les perceptions et les actions étant définis par le modélisateur, ils n'ont de sens que pour lui et ne sont pas explicités au sein de l'agent. Il est de plus très délicat de concevoir une architecture réactive adaptative : l'obtention du comportement souhaité repose parfois sur le

[15] BROOKS cité dans [Wooldridge and Jennings, 1995a]

[16] Le lecteur intéressé trouvera des revues très complètes des principales architectures réactives dans [Pirjanian, 1999] ou [Arnaud, 2000].

réglage de valeurs qui n'ont pas de signification pour le modélisateur (seuil d'activation, coefficient d'auto-inhibition d'un état...). Tous ces problèmes prennent leur source dans le même défaut majeur des approches réactives : le manque de sémantique. En évitant toute représentation symbolique au sein du modèle, l'informaticien prive celui-ci de toute capacité d'introspection. Il n'est alors pas aisé d'implémenter des mécanismes d'auto-adaptation ou d'apprentissage qui manipulent des valeurs qui n'ont pas de signification dans le domaine de la simulation. De même, la génération d'explication par l'agent sur ses prises de décision est souvent impossible, l'analyse de la trace d'exécution est alors le seul recours pour comprendre une particularité du comportement généré.
Ces lacunes interdisent la résolution systèmatique de problèmes complexes par des architectures réactives, ce qui n'est de toute façon pas leur but.

Les agents hybrides

Afin de combiner les avantages de ces deux approches, symboliques et réactives, certains travaux tendent à proposer des approches hybrides. Bien que ce soit le terme consacré, il serait sans doute plus exact de parler de mixité que d'hybridation : la démarche adoptée est généralement l'utilisation d'une architecture réactive conjointement à un système symbolique [Ferguson, 1992, Müller and Pischel, 1993, Wooldridge and Jennings, 1995b]. La partie réactive prend en charge les opérations de la bande cognitive alors que le système de résolution symbolique assure les tâches de la bande rationnelle (planification, anticipation, mémorisation, apprentissage...). Cependant, le but étant d'obtenir un comportement qui intégre les réactions aux stimuli tout en restant cohérent avec les buts de l'agent, il faut coordonner ces deux parties du dispositif de raisonnement : une loi d'activation d'une action doit pouvoir être inhibée ou forcée selon l'état de la partie délibérative. Ceci implique l'établissement d'un dispositif de contrôle qui exploite les sorties des parties réactives et délibératives pour déclencher les actions en maintenant la cohérence de l'ensemble.

La définition de ce mécanisme de contrôle constitue le point faible de ces approches. Il nécessite d'avoir une vue d'ensemble du comportement : il n'est pas aisé de prendre en considération tous les cas de figure, ne sachant pas *a priori* quels stimuli sont susceptibles de survenir au cours d'un raisonnement. Si l'établissement de tous ces cas d'utilisation était possible, cela signifierait qu'un modèle générateur du comportement est connu... La conception du seul système de coordination peut donc, dans les cas extrèmes, avoir la complexité d'une spécification complète de comportement. De plus, la juste répartition entre les parties réactives et délibératives est aussi problèmatique. Il peut paraître judicieux d'impliciter des raisonnements simples en les intégrant au module réactif, pourtant ceci "*gomme*" la sémantique associée et peut rendre le contrôle global encore plus difficile.
Pour expliquer cela, reprenons l'exemple de l'acquisition de l'heure par une entité. Admettons qu'un agent possède une montre. Lors de la définition de son comportement, il peut être tentant de coder comme un réflexe l'action de consultation de la montre pour satisfaire au besoin de connaître l'heure courante. En faisant cela, la sémantique du geste *regarder la montre* est implicité dans le modèle comportemental : l'agent "*ne sait pas*" pourquoi il regarde sa montre. Si l'agent a besoin de lire l'heure alors qu'une horloge est dans son champ de vision, cette définition va le conduire à regarder sa montre en plus de la consultation de

l'horloge. Pour éviter cela, il faut coder explicitement dans toutes les règles de raisonnement qui peuvent conduire à consulter l'heure l'inhibition du geste *regarder la montre*. L'autre solution serait de ne pas coder ce geste comme un réflexe, mais comme le résultat d'un des raisonnements possibles permettant d'acquérir l'heure. Cet exemple peut paraître assez peu crucial, mais il illustre une limite importante de ces approches hybrides en termes de crédibilité : un humain ne consulte pas toujours sa montre de manière rationnelle. Il nous arrive de porter le regard au cadran alors que nous venons de consulter l'heure juste avant, ou sans être capable l'instant suivant de dire ce que nous avons lu comme indication. Regarder sa montre est bien un geste réflexe parfois, cependant nous ne le faisons pas si une horloge se trouve devant nous (sauf pour vérifier la similitude des indications !). Il est donc des aspects du comportement qui ne sont pas faciles à classer parmi les réflexes ou les raisonnements, les approches hybrides conduisent donc souvent à multiplier les définitions d'un même comportement (définition comme réflexe, comme raisonnement et comme règle de contrôle global).

Cette difficulté est ici présentée sous son aspect pragmatique (implémentation), mais ce n'est sans doute que le symptome d'un problème de fond identifié comme suit par les auteurs de [Wooldridge and Jennings, 1995a] :

> One potential difficulty with such architectures, however, is that they tend to be *ad hoc* in that while their structures are well-motivated from a design point of view, it is not clear that they are motivated by any deep theory.

Les approches hybrides mènent donc à des modèles qui peuvent se révéler très efficaces, mais qui sont malheureusement trop applicatifs pour être pleinement réutilisables et permettre des avancées théoriques importantes.

Les architectures *BDI*

Cette section présente plus en détail les architectures *BDI*. Cet acronyme signifie **B**elief **D**esire **I**ntention, les notions de *croyance*, *désir* et *intention* étant les concepts clefs de ces approches. Cette attention particulière est dûe à l'enjeu évident que représentent ces architectures. De très nombreux travaux s'appuient sur des modèles *BDI* qui ont pris une place prédominante dans les approches multi-agents. Il s'agit d'un modèle intrinséquement symbolique, il est donc exploité pour des agents délibératifs[17] ou comme partie symbolique de modèles hybrides[18].

Le modèle *BDI* trouve ses fondements théoriques dans les travaux de BRATMAN [Bratman, 1987] qui identifient les intentions, conjointement aux croyances et aux buts, comme un élément central au raisonnement pratique. Dans [Bratman et al., 1991] il énonce les propriétés que devrait posséder un agent dont le raisonnement se base sur ces notions. Au cours de cette étude, il identifie deux difficultés majeures à la formalisation et la mise en œuvre de cette théorie :

[17] Par exemple *IRMA* [Bratman et al., 1991], *PRS/dMARS* [D'Inverno et al., 1997] ou *ADEPT* [Jennings et al., 2000].

[18] Par exemple INTERRAP [Müller and Pischel, 1993].

– la thèse d'asymétrie ;
– le problème des effets de bord.

La thèse d'asymétrie est composée de l'inconsistance de la relation des croyances aux intentions et de l'incomplétude de la relation des croyances aux intentions :
– un agent ne peut pas *rationnellement* croire qu'il ne réalisera pas l'acte α si il a l'intention de réaliser α (inconsistance croyance / intention) ;
– un agent peut *rationnellement* avoir l'intention de réaliser l'acte α mais ne pas croire qu'il réalisera α (incomplétude croyance / intention).

Le problème d'effet de bord peut être apparenté à un problème de *transitivité* des intentions : si un agent croit que $\alpha \Rightarrow \beta$, le fait d'avoir l'intention de réaliser α ne doit pas l'obliger à avoir l'intention de réaliser β. Un exemple très classique dans la littérature est le suivant : une personne qui croit qu'aller chez le dentiste va le faire souffrir et qui a l'intention d'aller chez le dentiste *n'a pas* l'intention de souffrir.

Les détails et implications de la thèse d'asymétrie et du problème d'effets de bord sont discuté dans [Rao and Georgeff, 1991a].

Malgrè ces difficultés, plusieurs formalisations des modèles *BDI* ont été proposées et, pour certaines, implémentées. Ces modèles s'appuient sur une logique multi-modale définissant des opérateurs entre les désirs, croyances et intentions de l'agent. Les logiques employées sont souvent complexes et rarement exécutables efficacement puisque non totalement axiomatisées. Nous ne détaillerons pas ici tous les axiomes et théorèmes des différentes approches pour ne pas surcharger la lecture : l'interprétation des formules demande une bonne familiarité avec la formalisation logique. Nous allons donc, dans la suite de cette section, présenter les principaux modèles *BDI* et les comparer qualitativement. Pour celà, nous insisterons principalement sur le rôle de la notion d'intention qui est le principal discriminant entre ces approches.

Cohen et Levesque

Dans [Cohen and Levesque, 1990] COHEN et LEVESQUE sont parmi les premiers à formaliser partiellement la théorie de BRATMAN et POLLACK. Ils utilisent pour cela le concept des *mondes possibles*. Un *monde possible* est un ensemble d'événements possibles situés dans le temps. Les croyances et buts de l'agent constituent des relations d'accessibilité qui déterminent les mondes accessibles en croyance et les mondes accessibles en but. L'ensemble des mondes accessibles en buts doit être un sous-ensemble des mondes accessibles en croyances (l'agent a un but qu'il croit possible). Les auteurs appellent cette notion la propriété de *réalisme*. Les problèmes de la thèse d'asymétrie et d'effets de bord ne sont pas résolus, et l'intention possède un rôle qui se démarque peu de la notion de but : l'intention est la persistance d'un but, ce qui amènera les auteurs à travailler sur le concept d'*engagement*.

L'intention est alors un facteur de stabilisation du système décisionnel, évitant les phénomènes d'oscillation entre différents buts [Dastani et al., 2003] :

> An intention can be interpreted as a previous decision that constrains the set of alternatives from which an agent can choose, and it is therefore a factor to stabilize the decision making behaviour through time.

Ce modèle est difficile à implémenter efficacement, l'ensemble des mondes possibles pouvant atteindre un cardinal très élevé : pour chaque événement non déterministe, il faut générer autant de groupes de mondes possibles qu'il existe d'alternatives à l'issue de cet événement.

Rao et Georgeff

Pour faire face au problèmes de réalisation, et particulièrement à l'impossibilité d'implémenter un système *BDI* temps réel, RAO et GEORGEFF ont proposé une autre formalisation de cette théorie [Rao and Georgeff, 1991b, Rao and Georgeff, 1995].

Leur approche se base sur celle de COHEN et LEVESQUE mais diffère sur les points sensibles :
- les événements non déterministes sont traités de manière probabiliste ;
- l'intention est traitée comme une attitude mentale au même titre que les désirs et les croyances, la notion d'engagement est donc redéfinie.

L'ensemble des situations possibles est manipulé sous forme arborescente. Il est composé de nœuds de décision (ce que décide l'agent) et de nœuds de hasard (ce que "*décide*" l'environnement pour les événements non déterministes). Des règles de réduction transforment cette représentation en arbre des mondes possibles, les seuls nœuds restant étant des nœuds de décision étiquetés par deux valeurs :
- la probabilité d'occurence de la situation ;
- le coût de parcours du chemin jusqu'à cette situation.

L'agent peut donc prendre ses décisions en fonction de la contingence (accessibilité en croyance) et de l'utilité (accessibilité en désir) des mondes possibles.

L'intention devient une attitude mentale de base et les auteurs énoncent la propriété de *réalisme fort* [Rao and Georgeff, 1991b] :
- pour chaque monde accessible en croyance ω, il existe un monde accessible en but ω' qui est un sous-monde de ω ;
- pour chaque monde accessible en but ω', il existe un monde accessible en intention ω'' qui est un sous-monde de ω'.

Ceci peut se vulgariser comme suit : un agent qui désire une situation la considère comme une option.
Les auteurs définissent aussi la notion de *réalisme faible* dans [Rao and Georgeff, 1995] qui pourrait s'énoncer : un agent qui désire une situation la croit possible (il ne croit pas que sa négation est inévitable).

La thèse d'asymétrie de BRATMAN est alors respectée.

Cette modification du statut de l'intention implique une redéfinition de la notion d'engagement : *intention* et *engagement* ne sont plus confondus. L'engagement est alors le processus de révision de l'intention : il détermine sous quelles conditions un agent doit abandonner son but précédent (rupture d'engagement). L'intention, elle, conditionne l'engagement. Les auteurs proposent différents mécanismes de révision des intentions modifiant l'engagement et définissent ainsi trois types d'agents :

- l'agent engagé aveuglément ;
- l'agent étroit d'esprit ;
- l'agent ouvert d'esprit.

Ces variantes sont détaillées dans [Rao and Georgeff, 1991a].

Cette formalisation ne considère pas de relation entre les intentions et se heurte donc elle aussi au problème d'effets de bords.

Konolige et Pollack

Pour échapper à ces écueils, KONOLIGE et POLLACK proposent dans [Konolige and Pollack, 1993] une formalisation pour laquelle les logiques modales normales sont abandonnées pour les intentions. Ils proposent d'utiliser les intentions comme discriminant entre deux sous-ensembles des mondes possibles : les mondes que souhaite l'agent (qu'ils nomment *scénarios*) et les mondes qu'il ne veut pas. L'intention devient un opérateur non fermé pour l'implication qui permet à l'agent de choisir un scénario minimal satisfaisant une situation *s* sans satisfaire l'intention d'obtenir *s* (aller chez le dentiste et être en situation de souffrir sans avoir l'intention de souffrir). L'essentiel de la formalistion reste identique à la proposition de RAO et GEORGEFF.

Seules les principales approches ont été présentées ici, sous un angle assez pragmatique : les principaux défauts mis en évidence concernent la difficulté d'implémentation. Il est écrit à ce sujet dans [Wooldridge and Ciancarini, 2000] :

> In general, models for these logics (typically accesibility relations for each of the modal operators) have no concrete computational interpretation. As a result, it is not clear how such agent specification languages might be executed.

Cette remarque de fond est à relativiser par le nombre croissant de modèles basés sur ces propositions qui sont implémentés. Cependant, il est vrai que ces formalisations comportent toutes des lacunes qui les limitent à des problèmes simples, la complexité algorithmique de leur exécution restant le principal obstacle.

Les recherches sur les agents *BDI* ne se limitent plus désormais aux théories de BRATMAN et abordent d'autres aspects de la prise de décision. Par exemple, la logique déontique est utilisée dans certains travaux afin d'introduire la notion d'obligation [19]. En ef-

[19] ([Boella and van der Torre, 2004] par exemple)

fet, si la notion d'engagement stabilise le mécanisme décisionnel d'un agent, les notions d'*obligations* et de *normes* semblent avoir le même bénéfice à l'échelle des systèmes multi-agents ([Torre, 2003] cité dans [Dastani et al., 2003]) :

> *Norms stabilizing multiagent systems*. It has been argued that obligations play the same role in multiagent systems as intention do in single agent systems, namely that they stabilize its behavior.

Ceci nous amène à un problème connexe de la modélisation de comportement : le contrôle de l'agent. Les notions de *norme* ou d'*obligation* permettent de contraindre un agent à "*respecter ses engagements vis-à-vis d'autrui*". Dans le cadre d'un système multi-agents, bien que l'autonomie de l'agent soit recherchée, il faut tout de même qu'il "*participe*" au but global. Nous regroupons sous le terme de *spécification* l'activité qui consiste à modéliser le comportement d'un agent tout en prévoyant les moyens de contrôle nécessaires à sa direction. La définition de cette notion de *spécification* constitue l'objet de la section suivante.

2.4 Agents autonomes situés et spécification de comportement

Après avoir présenté les principales familles de modèles comportementaux, nous devons revenir au champs d'application qui nous intéresse en premier lieu : l'utilisation de systèmes multi-agents en réalité virtuelle. Nous avons insisté sur la notion d'autonomie pour les agents impliqués dans ce type de simulation. Néanmoins, même dans ce domaine dont le but est parfois purement ludique, dans le cas des jeux vidéo ou du théâtre virtuel par exemple, il ne faut pas chercher à définir des agents totalement autonomes : il serait complètement inutile de créer un acteur virtuel particulièrement crédible qui n'ait pas "*envie de jouer*". La spécification d'un comportement ne se limite donc pas à la conception de son modèle générateur, il faut lui ajouter des moyens de contrôle. Le terme "*harnet*" est parfois employé pour désigner ces moyens de contrôle pour exprimer la manière dont ils doivent contraindre sans excès, suffisamment pour guider, comme le harnais guide le cheval.

Malheureusement, pour ce problème non plus il n'existe pas de solution universelle, et la définition du niveau de contrôle reste souvent une compromission strictement applicative.

2.4.1 La spécification : localité *versus* globalité

Que ce soit jouer une pièce de théâtre virtuel ou résoudre un problème distribué, l'agent doit "*participer*" à un but global pour nous être utile. La question se pose alors de savoir si l'agent doit connaître, totalement ou partiellement ce but. En effet, nous avons présenté en section 2.2.4 la localité de la perception de l'agent comme l'élément déterminant de son autonomie. Ce niveau de localité reste donc à définir pour chaque application.

Un agent qui ne perçoit que son environnement immédiat et ignore le but global du système multi-agents auquel il prend part n'a pas de "*raison*" de participer à cette résolution.

Ceci implique que lors de la définition du comportement, le modélisateur doit impliciter les mécanismes qui contribuent à cet objectif. Par exemple, pour le traitement d'image à l'aide de "*fourmis*" évoqué précédemment, les entités ignorent tout de l'objectif global. Le comportement de ces agents n'a donc pas été conçu pour résoudre cet unique problème, et l'adaptation dynamique de tels agents à une situation nouvelle est immédiat : les agents agissent uniquement en fonction de ce qu'il perçoivent et non en fonction d'une quelconque connaissance de la situation globale. Ce type d'approche pose le problème de l'*arrêt* : si un agent "*ne sait pas ce qu'il fait*", il n'a aucune raison de cesser d'agir. Une entité ne peut pas détecter l'échéance d'un but global dont elle ignore l'existence. De plus, il semble *a priori* peu prometteur en terme d'efficacité de confier la résolution d'une tâche à des entités qui "*ne se soucient pas*" de cet objectif.

L'autre extrême est la conception d'un agent qui perçoit au contraire la totalité de son environnement et "*connaît*" le problème à résoudre. Ceci implique dans la définition du comportement l'explicitation des mécanismes de résolution. L'agent est alors dédié à un type de tâche et ses capacités d'adaptation seront très limitées, ce qui peut être problématique dans un environnement ouvert. Par contre, il semble évident que la résolution sera menée plus efficacement par des agents qui "*savent ce qu'ils font*" et le problème de l'*arrêt* ne se pose plus.

En fait, contrairement à ces préjugés, une connaissance du problème globale peut s'avérer favorable ([Parker, 1993] par exemple) ou défavorable ([Petta and Trappl, 1997] par exemple) selon la tâche abordée.
L'approche la plus générique est donc la conception d'agents auxquels il est possible d'assigner des objectifs dynamiquement et qui peuvent acquérir des connaissances. L'implicitation et l'explicitation totales des mécanismes de résolution sont ainsi toutes deux évitées, et il est possible de "*diriger*" l'agent vers la résolution de l'objectif global, tout en ménageant son autonomie.

Toutefois, cette solution nécessite la création de dispositifs de prise en compte des objectifs. Le niveau d'abstraction de ces objectifs doit donc être défini lors de la conception du modèle comportemental. Cette problématique de choix du niveau de contrôle est discutée dans la section suivante.

2.4.2 Le syndrome de Frankenstein : contrôle *versus* autonomie

Les deux objectifs de création et de contrôle d'une entité autonome sont antithétiques. Dans ses romans de science fiction sur les robots, ISAAC ASIMOV nommait cela le "*syndrome de Frankenstein*" en référence au roman de MARY SHELLEY [Shelley, 1818]. Une partie de ses histoires s'appuie d'ailleurs sur les limites du contrôle de robots intelligents par les humains, ceux-ci étant gouvernés par les trois lois de la robotique[20] :

[20] ISAAC ASIMOV est considéré comme un des maîtres de la science fiction. Il fut le premier à envisager que le développement et l'étude de robot intelligent puisse constituer une science à part entière et inventa le mot *robotique* (sur la base du mot robot, issu du tchèque *robota* qui signifie *travailleur*). Dans son roman *Les robots* [Asimov, 1950] il imagine la parution de la 58ème édition du fictif *Manuel de la robotique* en 2058 qui

> – un robot ne peut porter atteinte à un être humain ni, restant passif, laisser cet être humain exposé au danger ;
> – un robot doit obéir aux ordres donnés par les êtres humains, sauf si de tels ordres sont en contradiction avec la première loi ;
> – un robot doit protéger son existence dans la mesure où cette protection n'est pas en contradiction avec la première ou la deuxième loi.

Ces lois sont séduisantes puisqu'elles semblent maintenir l'autonomie totale du robot tout en garantissant les intérêts des humains. Le fait de recourir à des lois permet de définir non pas *un* comportement, mais plutôt *une classe* de comportements valides. Cette définition déclarative et extensive est très expressive : la définition des limites à respecter est plus synthétique que l'énumération de toutes les actions autorisées. Pourtant, tout un cycle de romans d'ISAAC ASIMOV explore les failles de cet ensemble de lois. Par exemple, dans une de ces histoires un robot accepte de verser un poison dans un verre, et un autre robot accepte d'apporter ce verre à un humain. La distribution de connaissances locales – le contenu du verre d'une part, et son destinataire d'autre part – donne lieu à un comportement global en contradiction avec les lois : un meurtre !
La plupart de ces failles provient du fait que ces lois sont soumises à interprétation de la part du robot. C'est le revers de la médaille des approches de haut niveau d'abstraction, l'avers en étant la possibilité de confier des tâches complexes, impliquant des capacité de raisonnement, à ces entités. Un niveau de contrôle moins abstrait exigerait plus d'intervention de l'utilisateur.

Ces scénarios de science-fiction illustrent parfaitement les compromis auxquels il faut céder dans l'élaboration d'un système multi-agents. Pour certaines applications, telles que le théâtre virtuel par exemple, des procédés de contrôle très directs sont appropriés. Par exemple, l'architecture IMPROV déstinée à ce champ d'application ménage trois niveaux de contrôle [Blumberg and Galyean, 1995] :

– le niveau *direct* qui agit directement sur la géomètrie de l'agent (plier le coude droit de 20°) ;
– le niveau *tâche* qui agit sur l'action accomplie par l'agent (marcher vers un point, saisir un objet…) ;
– le niveau *motivation* qui agit sur le moteur comportemental de l'agent (faim, envie de jouer…).

Cependant, dans ce modèle ces contrôles sont définis dans des scripts qui génèrent une animation. L'exploitation de ces didascalies par les acteurs virtuels n'est pas gérée dynamiquement et en temps réel.

Pour répondre aux besoins de contrôle interactif, les "*harnaiss*" nécessaires doivent être moins intrusifs et donc relever d'un niveau d'abstraction plus élevé. Il existe de très nombreuses propositions de modèles adaptés à ce qui est nommé *autonomie adaptable*, la plu-

---

contient les trois lois de la robotique. Cette incursion dans la littérature de fiction nous autorise le plaisir rare d'une citation anticipée…

part se concentrant sur un aspect particulier. Par exmple, [Sardina and Shapiro, 2003] se concentre sur la priorité des objectifs individuels, [Myers and Morley, 2001] sur la direction d'une équipe d'agents, [Scerri et al., 2001] sur l'opportunité pour des agents de "*demander de l'aide*" à l'utilisateur... D'autres approches insistent plus sur l'utilisateur afin de lui offrir des interactions de haut niveau [Chen et al., 2001] ou de garantir la crédibilité des comportements qu'il observe [Caicedo and Thalmann, 2000].

Toutes ces propositions apportent des éléments pertinents en réponse à la problématique du contrôle, mais aucune ne peut répondre à toutes les situations : les besoins dans ce domaine sont totalement dépendants de l'application finale. La complexité croissante des modèles d'interaction posent un autre problème, soulevé dans [Aylett and Luck, 2000] : les canaux habituels d'interaction avec l'utilisateur ne suffisent plus. En plus des périphériques classiques (clavier, souris, gant de données...), il faut réfléchir à des outils assurant un interfaçage de plus haut niveau. Dans cet article les auteurs proposent par exemple le recours à une caméra pour l'interprétation des émotions de l'utilisateur. Une telle utilisation ne peut pourtant pas fonctionner pour toutes les applications : elle mise d'une part sur l'expressivité de l'utilisateur, qui varie fortement d'une personne à l'autre, et d'autre part sur sa sincérité. Il est tout à fait imaginable, pour une application de jeu ou de théâtre par exemple, que l'utilisateur s'amuse à jouer un personnage colérique : les émotions contradictoires de colère et d'amusement risquent d'être difficiles à distinguer... La surcharge de modalité d'interaction devient un problème d'ergonomie important pour l'utilisateur.

2.5 Conclusion : explicitation du *sens* de l'action

Nous avons présenté l'approche multi-agents comme un paradigme qui peut être exploité au cours des différentes étapes du cycle de conception d'une application :

- pour l'analyse par le thématicien l'agent est une unité conceptuelle ;
- pour la conception par le modélisateur l'agent est une unité computationnelle ;
- pour l'implémentation par l'informaticien l'agent est une unité d'implémentation ;
- pour le déploiement par l'utilisateur final l'agent est une unité d'exécution.

Après avoir exposé les différentes formes que peut prendre une démarche orientée agent au cours de ces étapes, nous nous sommes intéressés plus particulièrement à la modélisation et à l'implémentation qui revêtent une importance particulière dans le cadre de la réalité virtuelle, le but recherché étant l'interaction entre l'utilisateur et les agents de la simulation. Pour de telles applications, l'approche peut être qualifiée d'individu-centrée, chaque agent correspondant à une entité de l'univers virtuel. Les notions de situation et d'autonomie sont alors cruciales, ce qui amène à définir soigneusement la portée des perceptions de l'agent.

Si l'autonomie peut être à la fois un but et un moyen dans de tels développements, offrant la robustesse et l'adaptation requises dans les environnements ouverts, sa préservation est affaire de compromis avec le maintien de possibilités de contrôle de l'agent. Cette nécessité de pouvoir contraindre l'agent à participer à l'accomplissement d'un but global, observé par l'utilisateur comme un comportement émergent du système, s'accommode plus facilement

des modèles comportementaux de haut niveau d'abstraction. Le contrôle à travers l'émission de requêtes ou l'assignation d'objectifs avec une sémantique proche du domaine d'application est plus confortable pour l'utilisateur.

Cependant, de tels dispositifs impliquent l'utilisation d'un modèle d'agent délibératif qui se prête peu à l'exécution en temps réel de simulations complexes. A l'inverse, les modèles réactifs sont très efficaces en termes d'exécution mais le manque total de sémantique limite leur utilisation à des problèmes simples.
Des solution hybrides existent, mais il s'agit plus de solutions mixtes, un système de contrôle global à l'agent se chargeant de coordonner les résultats de différents modules (délibératifs et réactifs). Ces architectures mêlent l'expressivité des architectures délibératives avec l'efficacité des architectures réactives. Malheureusement, les fondements théoriques sont souvent absents et les implémentations sont des solutions *ad hoc* difficilement réutilisables.

Suite à de tels constats, nous avons décidé d'entreprendre la conception d'un modèle comportemental qui soit réellement hybride. Notre architecture se base sur un mécanisme de sélection d'action, c'est donc *a priori* une architecture réactive. Une telle approche semble plus à même de satisfaire aux contraintes d'exécution en temps réel. Cependant, nous choisissons d'ajouter des informations plus abstraites au modèle afin d'augmenter sa sémantique et ainsi d'éviter l'implicitation totale des mécanismes de résolution. En effet, l'idée la plus séduisante des architectures *BDI* est la prise en charge du *sens de l'action*. L'agent possédant des informations sur le sens des actions qu'il entreprend, il peut raisonner sur les conséquences de ses actes et donc les expliquer, apprendre de nouveaux comportements en fonction des résultats obtenus... Les architectures *BDI* se heurtent cependant aux limitations de toutes les approches symboliques, particulièrement au "*frame problem*". Mais si la définition des symboles permettant de raisonner efficacement sur un environnement ouvert semble un problème insurmontable dans le cas général, il est sans doute plus aisé de définir un ensemble fermé de symboles permettant à l'agent de raisonner sur lui-même.

L'idée fondatrice de notre approche est donc l'explicitation au sein de l'agent du *sens* de ses actions sous forme symbolique afin d'en permettre la sélection. Notre but est d'obtenir une architecture réactive qui possède une sémantique suffisante pour évoluer facilement, de manière applicative, vers des modèles plus complexes (apprentissage, génération d'explication...). Une perspective importante est aussi l'examen des apports de cette sémantique au regard des différentes étapes de développement (formalisation, établissement d'une méthodologie de développement...). Le concept de *sens* de l'action est abordé à travers la notion d'*intention*.

Le chapitre suivant s'attache donc à étudier les bénéfices de la prise en compte de la notion d'intention, et plus généralement d'intentionnalité, dans l'établissement d'un modèle comportemental hybride.

Chapitre 3

Intentionnalité et agent

> Sachez vous éloigner, car quand vous reviendrez à votre travail, votre jugement sera plus sûr.
>
> LEONARD DE VINCI

Ce chapitre s'attache à mettre en évidence les bénéfices attendus de la prise en compte de l'intentionnalité dans la conception d'un modèle comportemental. Son but est avant tout de discerner les aspects de cette notion qui peuvent être intégrés dans une démarche de spécification de comportement de ceux qui ne le peuvent pas dans l'état actuel des recherches. Il jette ainsi les bases de notre proposition qui est détaillée dans les chapitres suivants. La revue des aspects philosophiques, psychologiques et neurophysiologiques de l'intentionnalité est reportée en annexe (*cf.* A page 165) afin d'offrir une plus grande lisibilité du corps du document.

3.1 Introduction

L'étude reportée en annexe A considère la notion d'intentionnalité selon trois points de vue différents :

- le point de vue de la philosophie, plus précisemment la phénoménologique (à la lumière notamment des travaux de HUSSERL, MERLEAU-PONTY et SARTRE[1]) et la philosophie analytique (avec les travaux de SEARLE, BRATMAN et AUSTIN[2]) ;
- le point de vue de le psychanalyse (avec les approches de FREUD, LACAN et des travaux plus récents comme [Pachoud, 1997]) ;
- le point de vue de la neurophysiologie[3].

Notre volonté n'est pas de mener une étude approfondie dans ces trois domaines, mais plutôt d'identifier des points communs qui permettent d'aborder la notion d'intentionnalité transversalement au domaine d'application.

Nous allons dans un premier temps dégager des éléments de définition du rôle de l'intention vis-à-vis de la perception, de la décision et de l'action afin d'explorer la prise en compte de cette notion dans l'élaboration d'un modèle comportemental.
Dans un second temps, nous considérerons l'application d'une démarche intentionnelle à l'activité de spécification de comportements dans son ensemble, la notion d'intention étant alors employée comme un outil de conception.

3.2 L'intentionnalité : un élément du modèle comportemental

Nous débuterons cette synthèse par une reprise des phrases clefs de l'étude annexe. Certaines introduisent des réflexions qui semblent applicables à la conception de l'aspect interne de l'agent :

- Il faut restreindre les objets potentiels des intentions de l'agent pour ne pas se heurter au *"frame problem"* (A.2 page 168) ;
- la signification des représentations symboliques est propre au sujet et [...] les notions de *tendance* et d'*intention* participent du lien entre l'état mental d'un sujet et ses actes. De même, la concurrence et la contradiction entre ces tendances nécessitent la mise en place de mécanismes d'inhibition (A.3 page 169) ;
- la capacité de produire une intention est nécessaire au déclenchement d'une action volontaire (A.3 page 169) ;

[1] ([Husserl, 1913, Merleau-Ponty, 1945, Sartre, 1943])
[2] ([Austin, 1962, Searle, 1983, Bratman, 1987])
[3] ([Hollis, 1991, Berthoz, 1997, Bizzi and al., 2003])

- [il est nécessaire] d'établir un mécanisme d'inhibition entre les actions volontaires (initiées par une intention préalable, donc par le raisonnement) et les actions réflexes (comportements réactifs, stimuli/réponse) (A.3 page 170) ;
- l'intention n'est pas uniquement l'élément déclencheur de l'action, elle permet de l'ajuster et de prendre pleinement conscience de l'acte en cours. Il faut donc bien distinguer les intentions *préalables* à l'action des intentions *dans* l'action, qui existent aussi dans l'accomplissement d'actions réflexes (A.3 page 170) ;
- au niveau du contrôle de l'action, l'intention a donc un double rôle : ajuster l'effectuation de l'acte en cours en fonction du but poursuivi et permettre la construction d'un modèle d'anticipation à long terme (A.4 page 173) ;

D'autres remarques concernent les interactions et donc l'aspect externe de l'agent :

- dans l'interprétation des intentions d'autrui, l'hypothèse des limites des capacités de raisonnement de l'autre est [. . .] cruciale (A.5 page 175) ;
- si l'interprétation des intentions d'autrui est nécessaire à l'interprétation de son comportement et à la communication avec lui, nous avons vu qu'elle est aussi la base des comportements sociaux, tels que la collaboration par exemple (A.5 page 175) ;
- si le lien entre l'identification de l'intention et l'apprentissage semble indéniable, sa caractérisation exacte est encore l'objet de nombreux travaux, particulièrement en psychologie (A.5 page 176).

Il ressort donc de cette étude un double rôle de l'intentionnalité : le déclenchement et le contrôle de l'action d'une part (aspect interne de l'agent) et l'importance de l'intentionnalité dans le rapport à autrui d'autre part (aspect externe de l'agent). Nous reviendrons par la suite sur la complexité de ce deuxième aspect, le rapport entre l'intentionnalié et l'effectuation de l'action étant le plus concret et par là même le plus simple à synthétiser dans un premier temps.

A la manière dont nous avons décrit dans le chapitre précédent différentes voies possibles de l'application d'une approche agent aux systèmes multi-agents, il nous faut considérer plusieurs manières d'exploiter la prise en considération de l'intentionnalité dans l'élaboration d'un modèle comportemental. C'est ce à quoi nous nous employons dans la suite de cette section.

Nous allons ainsi explorer les aspects suivants :
- l'intentionnalité et l'action ;
- l'intentionnalité pour la prise de décision ;
- l'intentionnalité dans les interactions.

3.2.1 L'intentionnalité et l'action

L'étude en annexe souligne la complexité des phénomènes mis en jeu lors de l'effectuation d'une action et insiste sur le fait qu'il serait illusoire de vouloir construire un modèle

universel simple de ces mécanismes. Néanmoins, la considération de la notion d'intentionnalité transversalement à la philosophie, la psychologie et la neurophysiologie fait ressortir des traits communs sous ces trois éclairages.

Nous distinguerons trois étapes dans l'exécution d'une action :
- le déclenchement de l'action ;
- l'effectuation ;
- l'arrêt de l'action.

L'intention joue un rôle différent pour ces trois étapes.
L'intention préalable, issu des processus délibératifs, est nécessaire au déclenchement de l'action. Ceci constitue une des deux voies possibles de déclenchement, l'autre étant le réflexe. Il est en outre nécessaire d'établir des liens inhibiteurs entre les processus délibératifs et les processus réflexes pour garantir la cohérence du comportement. Pour reprendre la décomposition *perception - décision - action*, nous pouvons voir dans le déclenchement délibéré d'une action faisant suite à une intention préalable une première boucle de cette nature, les processus de déclenchement réflexes en constituant une seconde.

Quel que soit le moyen de déclenchement à l'origine de l'action, celle-ci doit être ajustée en cours d'effectuation. Ce contrôle de l'action n'est efficace que si l'intention qui motive l'acte persiste dans ce processus. C'est l'intention dans l'action. Les canaux de réafférance, en propageant cette intention vers les processus délibératifs, assurent un retro-contrôle de l'action indispensable à l'agentivité : l'entité *se sait* agissante et sait dans quelle intention elle effectue l'action en cours. Nous pouvons voir dans cette "*boucle d'asservissement*" qui contrôle l'action en cours d'effectuation une troisième boucle du type *perception - décision - action*. Il faut ici comprendre le mot *action* au sens le plus large, nous l'employons pour désigner les actions de perception au même titre que les interactions (le sens le plus commun du terme). En effet, nos perceptions sont fondées par des actions : il n'est pas possible de considérée la perception visuelle sans prendre en compte les mouvements oculaires par exemple. Notre état mental influence et contrôle en permanence l'effectuation de nos actes de perception, lesquels conditionnent à leur tour notre état mental. C'est le principe de perception active [Berthoz, 1997] : l'effectuation de l'action préconditionne les canaux sensoriels et influence la construction des perceptions sur la base des sensations. D'après ALAIN BERTHOZ, le cerveau agit la plupart du temps non pas comme un calculateur résolvant des problèmes mais comme un simulateur anticipant les effets de nos interactions. Par exemple, lors d'un mouvement de préhension les sensations de *toucher* sont amplifiées, avant même que l'objet ciblé ne soit atteint. A l'inverse, lors d'un contact inattendu, dans le noir par exemple, l'information est traitée avec retard et perçue avec une intensité différente. L'anticipation des effets présumés de l'action conditionne donc la construction de nos perceptions, lesquelles sont exploitées pour le contrôle de cette même action. Une modélisation d'un mouvement de préhension qui ne prenne pas en compte les sensations tactiles et kinesthésiques n'est donc pas complète, mais la réciproque est tout aussi vraie. Par la suite, le mot *action* désignera donc aussi les actes perceptifs.

Enfin, l'arrêt d'une action implique une décision d'arrêt qui ne peut être qu'intentionnelle.
La notion d'intention intervient bien à des degrés différents dans les trois étapes de l'effectuation de l'action.

La figure 3.1 résume cette triple boucle *perception - décision - action* régissant l'action. Elle fait apparaître les schèmes sensori-moteurs que sont les actions au sens large, action perceptives incluses (regarder, humer...), les processus délibératifs et réflexes de déclenchement d'action et l'activité de contrôle d'effectuation de cette dernière. Le modèle à court terme (pour les ajustements réflexes) et le modèle intentionnel à long terme (utilisé pour la délibération) qui sont construits à partir des perceptions sont aussi représentés (*cf.* annexe pour plus de détail à ce sujet). Les flèches en trait continu dénotent une relation d'*influence* alors que les flèches en trait interrompu dénotent une relation d'*exploitation*.

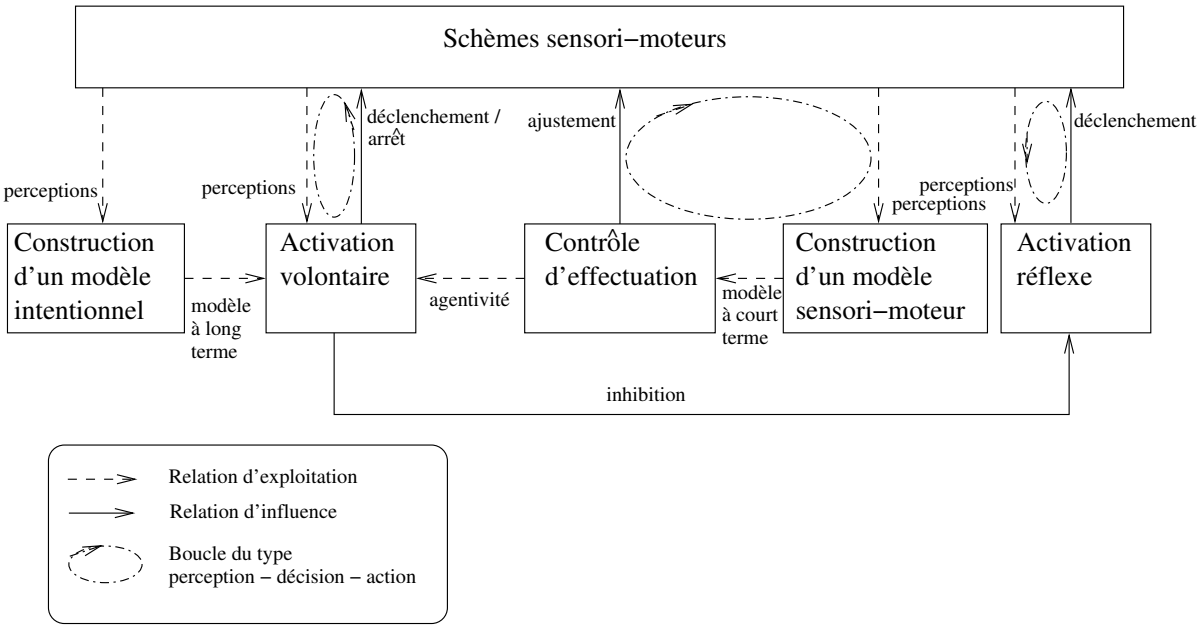

FIG. 3.1 – *Perception - décision - action : une autre vision.*

Dans les approches multi-agents, lorsque la boucle *perception - décision - action* est évoquée, il s'agit d'une fusion incomplète des différents pans de la triple boucle illustrée ici. L'essentiel des modèles se concentrent sur le déclenchement de l'action, qu'il s'agisse d'un déclenchement délibéré (agents délibératifs), d'un déclenchement réflexe (agents réactifs) ou des deux pour les agents hybrides.
Les aspects contrôle et arrêt de l'action sont généralement gérés implicitement ou de façon détournée, lorsqu'ils le sont. En pratique, pour la plupart des modèles l'exécution d'un schème sensori-moteur est traitée comme une action atomique. Ainsi, le problème de l'arrêt ne se pose pas : la méthode ou la fonction correspondante à l'action choisie est appelée et se

termine avec une valeur de retour reflétant le succès ou l'échec de l'action.
Le problème central est alors celui de la granularité de l'action. Si la méthode exécutée correspond à l'exécution complète d'un schème sensori-moteur, il faut recourir à un modèle de l'action pour pouvoir raisonner sur son adéquation avec le but poursuivi, et le contrôle d'effectuation doit être implicité dans la conception de ladite méthode. C'est le fonctionnement des approches délibératives. La difficulté est l'élaboration du modèle de l'action. La notion d'intentionnalité peut apporter des éléments de réponse.
Si par contre la méthode correspond à un fragment de l'action complète, il est possible de l'exécuter cycliquement au sein d'une boucle de contrôle jusqu'à la réalisation complète de l'action. Ces appels successifs peuvent donner lieu à un ajustement des paramètres, simulant ainsi un mécanisme d'ajustement de l'action en cours d'effectuation. C'est le fonctionnement de certaines approches réactives. La difficulté est alors de fixer les fréquences d'activation des méthodes et de détecter les conditions de maintien ou d'arrêt. De plus, il faut pouvoir assurer une certaine stabilité des processus de déclenchement pour obtenir un comportement cohérent, la compétition entre les différentes conduites réflexes pouvant facilement mener à une oscillation. Là encore, la notion d'intentionnalité peut contribuer à cette stabilité au travers de la notion d'*engagement* (*cf.* section suivante).

3.2.2 L'intentionnalité pour la prise de décision

Après avoir considéré le lien entre la notion d'intentionnalité et les actions et perceptions, nous allons maintenant nous intéresser au travaux qui explorent le rôle de cette notion dans la prise de décision. Dans le cadre de la modélisation de comportements d'agents rationnels, les principaux apports proviennent des travaux de DENNETT et BRATMAN.

D'après DENNETT les concepts de *croyance* et de *désir* peuvent être employés pour expliquer un comportement rationnel. Le fait de prêter à une entité de tels états mentaux et de la considérer comme rationnelle est appelé *proposition intentionnelle* (*intentional stance*) et cette entité est alors nommée *système intentionnel* ([Dennett, 1987] p.49). L'auteur distingue deux degrés différents pour les systèmes intentionnels (traduction d'un extrait de [Dennett, 1987], page 243, cité dans [Wooldridge and Jennings, 1995a]) :

> Un système intentionnel de *premier ordre* possède des croyances et des désirs (*etc.*) mais aucune croyance ou aucun désir *sur* les croyances et les désirs. [...] Un système intentionnel du *second ordre* est plus sophistiqué ; il possède des croyances et des désirs (et sans aucun doute d'autres états intentionnels) sur les croyances et les désirs (et les autres états intentionnels) – aussi bien les siens que ceux des autres.

Selon cette proposition, le comportement de toute entité rationnelle peut être prédit : l'observateur identifie les désirs et les croyances de l'entité et peut, en raisonnant sur cette base, anticiper sur les buts qui seront poursuivis. Les désirs sont des préférences à long terme alors que les croyances sont équivalentes à des connaissances, si ce n'est que leur véracité n'est pas nécessaire[4]. Selon cette approche, les *intentions* sont liées à l'*engagement* : une intention

[4]Ce modèle est proche des *Knowledge-Based Systems* (KBS) utilisés dans les systèmes experts pour assurer une sélection d'actions adaptées aux buts fixés par l'exploitation d'une base de connaissances portant sur le

est la rémanence d'un engagement, un but précédemment adopté.

Pour MICHAEL BRATMAN l'intention est bien plus qu'une rémanence des choix commis. Dans [Bratman, 1987] il place l'intention au cœur de la prise de décision. Pour lui, la modélisation au moyen des seules notions de *désir* et de *croyance* est insuffisante :

> [T]he *desire - belief model* sees intentional action that stands in appropriate relations to the agent's desire and beliefs. This is a reductive model : it sees intentions to act as reducible to certain desires and beliefs.

Il replace donc l'intention au même niveau que les autres états mentaux et, plus encore, lui attribue un rôle prédominant dans le lien entre les états mentaux en général et les actions. Cependant, il ne prétend pas lever totalement l'ambiguïté dont est empreint ce lien :

> We do thing *intentionally*, and we *intend* to do things. Our commonsense psychology uses the notion of intention to characterize both our *actions* and our *mental states* : I might intentionally start my car, and I might intend to start it. My intention to start it clearly does not guarantee that I do. But beyond that it is not obvious how these two phenomena *are* related. A central problem for a theory of intention is to provide a plausible account of this relation.

Dans ce même ouvrage, BRATMAN considère la notion d'*intention* comme intrinséquement liée à celle de *plan*. Il dénonce en outre l'ambiguïté de ce second concept dont il donne deux définitions différentes. Selon lui, un plan peut être une structure abstraite identifiant une fonction qui lie un enchaînement d'actions à une situation. C'est le sens le plus communément adopté. Mais il définit aussi le plan comme une structure comprenant en plus des états mentaux. Dans ce second cas, un plan contient, en plus d'une représentation d'un enchaînement d'action, des états mentaux spécifiques à ce plan, tels que des croyances, des désirs… Nous reviendrons dans le paragraphe suivant sur cette double définition.

Pour résumer la proposition de BRATMAN, nous pourrions synthétiser le rôle de l'*intention* dans le processus de prise de décision en trois points :

- l'intention participe à la recherche d'un plan permettant d'atteindre le but fixé selon les ressources disponibles ;
- l'intention contraint les plans envisageables (l'entité ne peut avoir d'intentions contradictoires) ;
- l'intention participe à la remise en cause des croyances.

La seconde manière dont la notion d'intentionnalité a été employée dans l'abord des systèmes multi-agents prend donc sa source dans les travaux de MICHAEL BRATMAN. Bien que certaines approches tendent à exploiter l'intention dans le second sens de la notion de *plan* proposé par ce-dernier[5] (citons par exemple [ChaibDraa and Millot, 1990]), la plupart des études exploitent les théories de BRATMAN à des fins de planification d'action. Il s'agit des modèles *BDI* présentés au chapitre précédent (section 2.3.2 page 48).

---

monde.

[5]Le plan est alors considéré comme une structure mentale comprenant des états mentaux

Hormis les difficultés de réalisation déjà évoquées, les modèles *BDI* présentent des lacunes qui limitent fortement la classe des comportements modélisables. La principale est connue sous le nom de *problème de l'effet de bord* (*side-effect problem*). Un agent qui possède les connaissances suffisantes sur les effets de ses actions peut, lorsqu'il considère un plan, raisonner à l'infini sur les conséquences de ses actes. Ceci peut, par une trop grande "*clairvoyance*", le conduire à abandonner systématiquement ses plans et ne jamais prendre de décision. Un exemple classique dans la littérature est celui d'un agent qui est sur son lieu de travail et qui désire aller à la plage. En considérant ce plan, il peut déterminer les conséquences d'une telle décision :

- pour aller à la plage il doit quitter son lieu de travail ;
- quitter son lieu de travail peut le faire renvoyer ;
- être sans travail implique un manque d'argent ;
- le manque d'argent engendre le désir de travailler ;
- le désir initial d'aller à la plage conduit donc au désir de travailler !

Cette "omniscience" de l'agent met donc en défaut le modèle théorique dans la génération de comportements crédibles. Cette lacune trouve son fondement dans l'absence de limitation des ressources disponibles pour le raisonnement de l'agent. Une entité réelle cesse de réfléchir aux "*conséquences des conséquences*" de ses actes au-delà d'un certain degré. BRATMAN traite en détail ce problème dans [Bratman et al., 1991] où il énonce les propriétés d'un agent aux ressources de raisonnement limitées, l'architecture devant selon lui :

- permettre une analyse moyens / fins pour identifier les buts non contradictoires ;
- permettre une comparaison des options concurrentes par une pondération en fonction de leur pertinence ;
- permettre des interactions entre ces deux premiers modes de raisonnement ;
- répondre au problème des limitations des ressources.

Il identifie donc le problème mais ne propose pas directement de solution. Cette démarche l'amène à formuler la *thèse d'asymétrie* à laquelle les modèles *BDI* répondent par la définition de la notion de *réalisme* (*cf.* section 2.3.2).

L'autre lacune importante des modèles *BDI* provient de la définition du concept de *plan* employée, et non pas directement de l'application des théories de BRATMAN. En effet, en excluant la représentation d'états mentaux spécifiques dans la structure de plan, les modèles *BDI* échouent à la prise en compte des interactions sociales. Nous pouvons par exemple considérer le problème de l'ordre et de l'obéissance. Un agent qui reçoit un ordre et qui obtempère adopte une intention qui est éventuellement contradictoire avec ses intentions personnelles. La notion de réalisme des modèles *BDI* ne permet pas de représenter une telle attitude mentale. La seule issue est donc d'amener l'agent à adopter l'intention "*ordonnée*" comme intention "*personnelle*". Or, une telle acceptation n'a valeur que dans *un* des mondes possibles (où l'ordre est accepté). Pour prendre en compte totalement l'ordre et l'autorité, un agent devrait dans ce cas raisonner à égalité sur deux types de mondes accessibles en possibilités : un ensemble de mondes dans lequel l'ordre est accepté et un autre ensemble dans lequel il ne l'est pas. Ces deux ensembles comprennent alors des états mentaux diffé-

rents. Les plans devraient donc comporter des états mentaux pour supporter les interactions sociales [ChaibDraa and Millot, 1990]. En pratique, les approches existantes ajoutent un protocole de gestion des interactions sociales au modèle *BDI* par le biais de nouvelles attitudes mentales, mais le plan demeure basé sur la première définition : les états mentaux n'en font pas partie.
Cette incapacité du modèle *BDI* de base à prendre en charge complètement les interactions sociales explique les travaux proposant d'ajouter aux états mentaux épistémiques (*connaissances* et *croyance*) et conatifs (*désirs* et *intentions*) des attitudes déontiques (*obligation* ou *engagement* par exemple) [Broersen et al., 2001]. Mais l'enrichissement du modèle par de nouveaux concepts et les opérateurs associés ne semble pas être une solution à long terme : l'ensemble des états mentaux impliqués dans le rapport à autrui est de cardinal bien trop important pour donner naissance à une logique de raisonnement se prétant à une résolution automatique efficace. Ces approches entretiennent le faux espoir qu'il est possible de définir une sorte de logique de la pensée. Un modèle issu de cette démarche ne peut s'appliquer à toutes les situations envisageables. La faille de cet abord du problème prend sa source dans une ambiguïté de la notion d'intentionnalité. Un état mental peut être qualifié d'intentionnel lorsqu'il relève de la *visée*, de la *représentation* : cette intentionnalité n'est pas liée à l'intention (dans le sens avoir l'intention d'une action). Par exemple l'inquiétude est un état mental intentionnel, dans le sens où l'on est inquiet *au sujet de* quelque chose. L'inquiétude ne peut en aucun cas engendrer seule l'action. Les états mentaux qui peuvent être qualifies d'intentionnels sont alors très nombreux et, d'après SEARLE, il n'est pas justifié de considérer que les *désirs* et les *croyances* entretiennent une relation particulière avec l'*intention*. Il écrit à ce sujet [Searle, 1983] :

> Considérer, par exemple, les croyances et les désirs comme des sortes d'intentions participe [d'une] confusion. Croyances et désirs sont des états Intentionnels mais n'intentionnent (*intend*) rien. [. . .]
> Voici quelques exemples d'états qui peuvent être des états Intentionnels : croyance, crainte, espoir, désir, amour, haine, aversion, sympathie, antipathie, doute, questionnement, joie, exaltation, dépression, inquiétude, orgueil, remord, tristesse, chagrin, culabilité, ravissement, irritation, perplexité, approbation, pardon, hostilité, affection, attente, colère, admiration, mépris, respect, indignation, intention, souhait, manque, imagination, fantaisie, honte, convoitise, dégoût, animosité, effroi, plaisir, répulsion, aspiration, amusement et déception.

Le modèle *BDI* ne permet donc pas de réaliser la *visée intentionnelle*, l'intention manipulée est l'intention de commettre une action. Seul le sens téléonomique de la notion est exploité. Ainsi, le modèle *BDI* est un modèle purement interne : il permet d'assurer une tâche de planification mais n'offre aucune prise en compte des interactions de l'agent avec d'autres entités en termes d'intentionnalité.

Enfin, en acceptant le modèle *BDI* comme un modèle comportemental qui gère les aspects strictement internes de la prise de décision et non les interactions sociales, il demeure un biais dans les implémentations qui ont déjà été réalisées. En effet, dans tous ces modèles, les *désirs* jouent un double rôle [Dastani et al., 2003], ce qui n'est pas conforme aux théories de BRATMAN :

– d'abord les désirs permettent de générer l'ensemble des buts non contradictoires de l'agent ;
– ensuite les désirs permettent d'évaluer les effets de bord des actions pour atteindre ces buts.

Or, les effets de bord évalués d'une action menant à un but peuvent être à ce point contradictoires avec les désirs que ce but n'en est plus désirable et n'aurait pas dû être adopté. L'adoption d'un but devrait donc se faire conjointement à l'évaluation de tous ses effets de bord, et non uniquement sur l'intention qu'il représente. Cette division en deux phases permet un traitement selon une résolution du type "*générer - tester*" très pratique à implémenter, mais elle dénature le rôle des *désirs* dans le modèle.

Si les notions de *désir*, de *croyance* et d'*intention* fondent un système de planification très expressif, elles ne suffisent pas, en l'état actuel des approches *BDI*, à constituer un processus de décision complet. D'autres mécanismes doivent leur être adjoints, comme un système de limitation des ressources de raisonnement par exemple. De plus, le modèle *BDI* peine à prendre en compte les interactions sociales de l'agent avec d'autres entités. La section suivante étudie la possibilité de prendre en compte cette dimension de l'intentionnalité dans un modèle informatique.

3.2.3 L'intentionnalité dans les interactions

Au-delà de l'exploitation de la notion d'intentionnalité à des fins de planification, l'apport le plus attendu de la prise en compte de cette notion serait la considération de la *visée intentionnelle* telle que la définit SEARLE [Searle, 1983]. L'explicitation au sein d'un modèle comportemental des états mentaux impliqués dans ces théories serait la porte ouverte à l'émergence de comportements épigénétiques de l'agent. L'intentionnalité, si elle était traitée dans sa pleine définition, deviendrait ainsi un véritable moteur d'interactions pour l'agent, guidant ses décisions non plus seulement par rapport à ses seuls désirs, mais aussi par rapport à l'ensemble des phénomènes perçus dans son environnement et l'interprétation qu'il en ferait en termes intentionnels. Cependant, un tel dessein se heurte aux limites habituelles des approches symboliques en se confrontant au problème de la *représentation* : les symboles manipulés par la machine ne sont pas des *symboles* au sens philosophique du terme. En effet, tenter de représenter les attitudes intentionnelles telles que les définit SEARLE, avec leurs directions d'ajustement et leurs conditions de satisfaction, implique d'expliciter le *sens* des symboles manipulés. Or, SEARLE lui-même énonce l'impossibilité de cette explicitation en identifiant le concept d'*arrière-plan* (*background*) : les représentations intentionnelles ne se définissent pas totalement les unes par rapport aux autres. Le réseau de nos représentations intentionnelles ne se comprend, selon lui, que conjointement à un ensemble de connaissances sur le monde : l'arrière-plan. Ces connaissances sur "la manière dont les choses fonctionnent" ne sont pas représentationnelles mais font partie des conditions de satisfaction de certaines attitudes intentionnelles. Ceci est résumé comme suit dans [Cicourel, 1987] :

> [A]n Intentional Network functions in the context of a Background of 'non-representational mental capacities'. This latter refers to certain sorts of know-how about the way things work.

L'auteur ajoute que selon lui les états mentaux sont difficilement identifiables pour un raisonnement donné, et certainement non énumérables.

SEARLE, par le recours à la notion d'arrière-plan, évite d'avoir à définir comment les symboles manipulés par nos attitudes mentales nous sont signifiants : il n'est pas de sémiotique des actes mentaux, ils héritent leur sens de l'arrière-plan.

Les problèmes soulevés par la modélisation informatique de la notion d'intentionnalité sont donc les mêmes que ceux qu'implique la théorie computationnelle de l'esprit. Les premiers défenseurs de cette théorie, FODOR et PUTMAN en tête, pensaient pouvoir reproduire le fonctionnement de l'esprit par le truchement de systèmes symboliques. Cette approche était particulièrement appréciée par les philosophes matérialistes puisqu'elle proposait une réalisation *physique* de l'intention comme *symbole* représenté et manipulé dans la mémoire d'une machine. Mais ces symboles n'ont que le sens qui leur est prêté, ils ne réalisent donc en rien l'intentionnalité. Tout au plus, ils en sont une représentation qui n'a de signification que pour le concepteur, ou l'observateur. Dans [Horst, 1996] STEVEN HORST résume ce constat comme suit (p. 135) :

> There are really two quite separate descriptions of the computer. On the one hand, there is a functional-causal story ; on the other a semiotic story. The art of the programmer is to find a way to make the functional-causal properties do what you want in transforming the symbols. The more interesting symbolic transformations you can get the functional properties of the computer to do for you, the more money you can make as a computer programmer. So for a computer to be *useful*, the symbolic *features* need to line-up with the fonctional-causal properties. But they *need not in fact* line up, and when they do it is due to excellence in design and not to any a priori relationship between functional descriptions and semiotics.

Pour étayer cette position, nous pouvons considérer les développements effectués à partir de la théorie des actes de langages, ceux-ci étant des actes mentaux intentionnels par essence. L'utilisation en informatique de cette théorie a permis de formaliser un protocole d'échange entre agents. Les mécanismes qui en découlent sont très expressifs et permettent la communication entre des agents hétérogènes, donnant les bases à des processus d'interactions sociales. Pourtant, même si les modèles qui sont issus de ces travaux sont très aboutis, personne n'affirmerait que les agents "comprennent" ce qu'ils se "disent". Ils peuvent vérifier la structuration des messages échangés et en extraire des informations pertinentes, mais ils n'ont pas de compréhension de l'échange au sens linguistique du terme. Les symboles impliqués dans une telle communication n'ont réellement de sens que pour nous, aucunement pour les agents

Ainsi, le problème de représentation symbolique semble condamner d'avance l'exploitation de la notion d'intentionnalité comme *visée intentionnelle* en informatique. Cependant, son étude n'est pas stérile, et le projet de création d'un modèle comportemental intention-

nel ouvre des perspectives intéressantes en matière de génie logiciel. Par exemple, le projet ADAPT [Manzotti et al., 2003] conduit les auteurs à mener une analyse approfondie des architectures informatiques qui pourraient réaliser un tel modèle. Le but de ce projet européen est la création d'entités virtuelles engendrant la sensation de présence. Pour les auteurs, cette sensation passe par la production de comportements intentionnels : nous ne pouvons considérer être "*en présence*" d'un individu que si nous pouvons lui prêter des intentions (c'est une forme d'application de la *proposition intentionnelle* de DENNETT). Si ce rapport ne propose pas de modèle complet, il identifie clairement certains obstacles à cette réalisation. Par exemple, les auteurs énoncent l'inadéquation des langages orientés objets à la modélisation d'états mentaux : si les représentations mentales des objets de la pensée sont eux-même réifiés, l'architecture obtenue confine au dualisme, les représentations étant alors différentes de ce qu'elles représentent.

Il n'existe donc pas à l'heure actuelle de modèle d'agent prenant en charge la notion d'intentionnalité comme *visée intentionnelle*. Au vu du niveau d'abstraction de l'approche qui laisse une part importante à l'interprétation des concepts impliqués, la tâche paraît aussi ardue qu'ambitieuse. Mais, si comme écrit dans [Loeck, 1990] il semble certain que l'informatique n'aidera pas ou peu à explorer la notion d'intentionnalité, la réciproque semble au contraire pleine d'avenir.

3.3 L'intentionnalité : un outil pour la conception

La section précédente explore la possibilité d'employer la notion d'intentionnalité comme composante d'un modèle comportemental. Au prix d'un effort d'abstraction, ce concept peut permettre d'aborder l'activité de spécification de comportement par elle-même. Ainsi, il est intéressant d'exprimer la définition du comportement d'une entité en termes intentionnels, même si le modèle informatique employé pour la réalisation de ce modèle ne manipule pas nécessairement cette notion explicitement. La *proposition intentionnelle* peut être utilisée comme un outil de conception.

L'intérêt majeur de la *proposition intentionnelle* de DENNETT est qu'elle permet d'aborder *a priori* toute entité avec ce même modèle. Bien qu'il semble plus intéressant de l'appliquer à des entités complexes dont le fonctionnement interne n'est pas connu, et qui ne permettent que l'observation de leur comportement et non l'analyse des mécanismes de conduite qui génèrent ces derniers, il est possible de prêter de tels états mentaux à des entités simples, au modèle connu.
Nous pourrions reprendre ici l'exemple du distributeur de boissons du chapitre précédent : il est tout à fait possible de modéliser un tel automate avec des états mentaux (*croyances*, *désirs* et *intentions*). Evidemment, dans un tel cas le recours à la proposition intentionnelle n'apporte rien à la compréhension du comportement de l'entité. Mais l'intérêt d'une telle démarche réside dans la souplesse de modélisation obtenue : depuis les entités les plus simples jusqu'aux agents rationnels, cette approche semble à même d'aborder tous les types de comportements de manière unifiée [Shoam, 1990].

L'ensemble de ces observations amène WOOLDRIDGE à considérer le fait pour un agent d'être un système intentionnel comme une condition nécessaire, mais non suffisante, à l'agentivité [Wooldridge and Jennings, 1995a].

Mais si la *proposition intentionnelle* permet de modéliser les comportements d'entités complexes aux modèles inaccessibles, il faut rester prudent sur l'emploi de cette démarche dans l'observation des résultats obtenus. Considérer le comportement d'un agent artificiel en lui prêtant de tels états mentaux peut se résumer à de l'antropomorphisme inopportun : si le comportement de l'agent n'est pas généré par un modèle reposant sur de tels concepts, leur emploi dans l'analyse des résultats est sans objet. Par exemple dans [Brassac and Pesty, 1996] CHRISTIAN BRASSAC et SYLVIE PESTY considèrent différents modèles de coopération. Ils distinguent les cas où les agents possèdent "une intentionnalité collective de réaliser un but" – ils caractérisent l'organisation résultante des interactions de ces agents de *coopération "cognitive"* – des cas mettant en interaction des agents purement réactifs. Dans ce dernier cas, bien que l'observateur puisse constater ce qu'il interprète comme une coopération – que les auteurs nomment *coopération "réactive"* - il serait abusif de conclure que les agents coopèrent (*sciemment*) :

> [P]eut-on dire que les entités réactives orientent leurs agissements vers un but, de façon intentionnelle ? Sûrement non. Il s'agit donc de ne pas appliquer une terminologie intentionnaliste à un phénomène qui, certes peut être vu avec des lunettes "intentionnalistes" comme tel, mais qui n'est somme toute qu'un comportement à plusieurs de co-action.

Une voie d'application des théories de l'intentionnalité aux systèmes multi-agents est donc le recours à la *proposition intentionnelle* de DENNETT pour analyser les comportements à reproduire. Il reste alors à définir les outils qui permettent de concevoir les agents dont le comportement est spécifié selon cette approche. C'est cette démarche qui a donné lieu à l'élaboration du modèle *BDI* tel qu'il a été présenté précédemment.

Une telle démarche est *a priori* applicable à d'autres modèles. Son apport principal est l'explicitation d'un lien sémantique entre les activités des différents intervenants dans la spécification (informaticien, concepteur. . .) : le comportement de l'agent émerge de la confrontation des points de vue de ces différents spécialistes. Une approche en termes intentionnels peut leur offrir un langage commun : en partant de l'idée que toute entité peut être spécifiée *via* une analyse intentionnelle, un modèle d'exécution capable de réaliser tous comportements spécifiés selon ce langage permettrait de simuler toute entité, rationnelle ou non.

3.4 Conclusion : l'intentionnalité dans un modèle comportemental

A la lumière de cette étude des rapports possibles entre la notion d'intentionnalité et la spécification de comportements d'agents autonomes, il apparaît que seul le lien entre l'inten-

tion et l'action est aisément exploitable au sein du modèle comportemental. L'intentionnalité joue un rôle indéniable dans les perceptions ainsi que dans le rapport à autrui, laissant entrevoir des applications en matière de communication, d'apprentissage ou plus généralement dans une modélisation plus avancée des processus mentaux. Mais de tels champs d'application sont tributaires des travaux connexes (en psychologie ou en neurophysiologie par exemple), et l'état actuel des recherches dans ces domaines ne permet pas d'établir directement un modèle informatique complet.
Cependant, l'étude de l'intentionnalité a déjà été considérée dans le cadre de la modélisation d'agents autonomes [Scaglione, 1997, Cardon, 2000, Müller, 2002, Manzotti et al., 2003], plus particulièrement dans son rôle vis-à-vis des perceptions de l'agent comme outil de construction d'une *extériorité*. Nous reviendrons dans le chapitre suivant sur ce point particulier. Toutefois, à notre connaissance, si ces travaux ont donné lieu à des propositions de modèles exploitant un aspect de l'intentionnalité, aucune architecture à ce jour n'est à même de permettre à un agent de se construire par son seul raisonnement une compréhension de son environnement médié par ses seules perceptions. Ainsi, nous pouvons dire qu'il n'existe pas de modèle d'agent intentionnel dans le sens phénoménologique du terme.

Pour la suite, sans pour autant écarter définitivement les autres aspects de notre réflexion d'ensemble, nous nous concentrerons particulièrement sur l'intentionnalité dans le sens téléonomique du terme : l'intention sera considérée dans son rôle dans le déclenchement, le contrôle et l'arrêt de l'action. Jusqu'à présent, parmi les propositions donnant lieu à des modèles réellement implémentés, les *BDI* sont pratiquement les seuls à explorer cette voie de recherche en matière de spécification comportementale. Or, ils ne donnent lieu qu'à des architectures symboliques. La notion d'intention a pour l'instant été fortement délaissée dans le cadre des approches réactives pour les systèmes multi-agents. Tout au plus, certaines architectures hybrides font cohabiter un module *BDI* avec un processus réactif (*InteRRap* par exemple [Müller and Pischel, 1993, Müller et al., 1995]). C'est du moins le cas pour le modèle d'exécution : la notion d'intention peut aussi être utilisée pour la spécification du comportement[6]. Pourtant, la notion d'intention reste pertinente dans le cadre d'une approche réactive, ne serait-ce qu'au travers du concept d'*intention dans l'action*. Comme nous l'avons vu, l'intention guide et ajuste l'action en cours d'effectuation, et cette notion ne doit pas être cantonée au seul domaine du cognitif. Cette constataion est la base de notre proposition, telle qu'elle est exposée dans les chapitres suivants : nous considérons l'exploitation de la notion d'intention au sein d'un modèle comportemental réactif.

Cette première application de la notion d'intentionnalité concerne le seul modèle comportemental. Le deuxième niveau de réflexion que nous avons mis en évidence consiste à considérer l'ensemble de l'activité de spécification selon une approche intentionnelle. Le comportement de l'agent synthétise les traits qui résistent à la variation de point de vue des différents intervenants impliqués dans la spécification : la définition du comportement constitue le champ d'intersubjectivité de ces différentes personnes. L'ensemble du proces-

[6]Dans [Müller, 1987] par exemple, l'auteur utilise une logique basée sur les notions de *croyance*, de *but* et d'*action* pour la spécification, alors que l'implémentation est réalisée par un automate construit à partir de cette spécification.

sus de spécification relève donc d'une démarche intentionnelle. Une difficulté majeure dans cette collaboration est la définition d'un langage commun. Cette considération nous amènera à proposer un modèle qui favorise l'établissement d'un lien entre les intervenants : chaque spécialiste doit disposer d'outils lui permettant de minimiser l'implicitation du sens de ses apports au modèle comportemental lors de sa construction. L'instauration d'une sémantique qui puisse être commune au concepteur et à l'informaticien au sein du modèle sera un point essentiel de notre proposition. Dans la continuité de la *proposition intentionnelle*, nous utiliserons donc la notion d'intention comme trait d'union explicite entre les activités de ces spécialistes.

Dans les chapitres suivants, nous allons présenter une architecture de sélection d'action. Notre but est de concevoir un modèle réactif qui possède une sémantique suffisante pour permettre au concepteur d'expliciter le *sens* de l'action vis-à-vis de l'agent.
L'idée maîtresse de l'approche est d'exploiter la notion d'intention pour une sélection d'action basée sur un raisonnement qualitatif[7]. L'agent exploite des règles comportementales pour générer l'ensemble des intentions à satisfaire. Il s'agit d'intentions exprimées qualitativement et qui portent uniquement sur l'état interne de l'agent. Pour reprendre la terminologie employée dans le chapitre 2, nous pouvons dire qu'il s'agit d'un modèle d'agent intentionnel (dans le sens où il repose sur la notion d'intention) qui n'est pas *situé* en intention (l'agent mû par ce modèle ne prend pas en compte les intentions d'autrui). Le but du recours à un raisonnement qualitatif est d'effectuer une sélection d'action rapide, sans passer par un processus délibératif complexe. Ainsi, le contrôle continu de l'action est simulé par une succession d'appels à une méthode de faible granularité (qui réalise un fragment de l'action terminale) à une fréquence plus élevée que ne le permettrait une résolution numérique.
Le lien entre les intentions à satisfaire et les méthodes appropriées est fait grâce à une connaissance par l'agent de l'effet de ses méthodes sur son état interne : il possède un modèle qualitatif de l'effet de ses méthodes, c'est le "*sens*" de ses actions.
Les règles comportementales qui génèrent les intentions à satisfaire sont en charge du déclenchement de l'action. Elles peuvent réaliser un déclenchement délibéré ou un déclenchement réflexe, selon le parti adopté par le concepteur. De plus, comme il s'agit de règles logiques, des mécanismes d'inhibition sont aisément exprimés dans les conditions d'éligibilité de ces règles, permettant de concevoir les deux types de déclenchement au sein du même agent.

Il s'agit donc d'une approche hybride, non pas dans le sens où elle fait cohabiter des modèles réactifs et symboliques, mais dans le sens où elle exploite des symboles au sein d'un processus réactif de sélection d'action, apportant la sémantique recherchée (l'explicitation du "*sens*" de l'action). Les perspectives reposent sur l'exploitation de cette explicitation comme base à des processus cognitifs, favorisant ainsi des tâches d'apprentissage ou de génération d'explication : l'agent possède des informations sur les raisons de sa sélection sous forme textuelle et qualitative.

Le modèle proposé explore donc la considération de la notion d'intentionnalité selon les

[7] La méthode proposée repose sur des opérateurs qualitatifs simples, nous ne nous appuyons pas pour autant sur une théorie de la résolution qualitative telle que [Forbus, 1984].

deux axes que nous avons mis en évidence :

- dans l'élaboration du modèle comportemental, l'*intention dans l'action* est exploitée pour le contrôle de cette dernière ;
- dans la démarche de conception, l'enrichissement sémantique d'un modèle purement réactif permet de créer un lien entre les phases d'analyse et de conception.

Au-delà de la définition d'un modèle simple de sélection d'action, il faut donc considérer notre proposition comme relevant d'une démarche plus globale : notre but est d'aborder la définition d'un modèle comportemental avec le souci de son intégration dans le processus de spécification. Nous attacherons donc une attention particulière à l'instauration d'un lien entre les différents intervenants qui leur permette d'aborder la spécification selon leur point de vue tout en employant un vocabulaire commun.

Le chapitre 4 présente en détail le modèle d'agent issu de cette réflexion. Le chapitre 5 pour sa part expose les détails du modèle d'exécution capable de supporter ce modèle comportemental. Enfin, le chapitre 6 présente l'application qui a été réalisée sur la base de ces travaux.

Chapitre 4

Le modèle d'agent intentionnel

> If you do things right, people won't be sure you've done anything at all.
>
> *Dieu, dans un épisode de Futurama*, MATT GROENING

Ce chapitre définit un modèle d'agent sur la base des réflexions menées dans le chapitre précédent sur la prise en compte de la notion d'intention dans la spécification de comportements. Un tel travail n'est exploitable que s'il est effectué avec la conscience de son contexte : il faut garder à l'esprit les différentes étapes de spécification – analyse, conception, développement et déploiement – qui ont été présentées dans le chapitre 2 (pages 23–56). Cette prise en compte est la condition *sine qua non* de l'élaboration d'un modèle qui puisse s'intégrer dans une démarche de développement.

4.1 Les bases du modèle

4.1.1 Les objectifs

Parmi les nombreuses approches et méthodes pour la spécification fonctionnelle et comportementale d'agents adaptées à la réalité virtuelle (*cf.* chapitre 2), l'emploi d'une ou plusieurs d'entre elles dépend souvent des habitudes du concepteur du système multi-agents, et de l'application concernée. Or, le choix de la combinaison d'une approche et d'un outil d'implémentation est problématique : si ce dernier n'offre pas de fonctionnalités qui correspondent directement aux outils conceptuels mis en œuvre pour la spécification, le développement sera difficile et l'extensibilité ainsi que la capacité de réutilisation seront compromises. Il apparaît donc nécessaire d'offrir aux spécificateurs un cadre de pensée uniforme couvrant les besoins d'expressivité depuis l'analyse jusqu'à l'implémentation du comportement dans un formalisme cohérent, et si possible une méthodologie le guidant dans sa démarche de spécification.
L'étude des modèles comportementaux et de l'outil supportant leur exécution ne peut donc pas être scindée si l'on souhaite garder cette cohérence.
La réflexivité entre l'élaboration du modèle et l'outil de conception et de simulation constitue une difficulté majeure.

Or, dans la majorité des approches existantes cette relation est unidirectionnelle : le modèle et l'outil de simulation sont conçus pour permettre *une* instanciation du comportement défini par le modélisateur. Considérons le cas des plateformes exploitant un langage orienté objets. Elles constituent la majorité des plateformes multi-agents existantes. Les actions et les perceptions de l'agent sont réalisées *via* l'appel de méthodes de ce dernier par une activité principale qui réalise la prise de décision (*cf.* figure 4.1 ci-après). Cette activité peut être elle-même une méthode appelée cycliquement, un *thread* ou un processus séparé par exemple, selon les technologies adoptées. Dans ce cas, l'informaticien réalise *une implémentation possible* du modèle conçu par le modélisateur. Il peut justifier, éventuellement prouver, ses choix techniques et ainsi garantir que son implémentation réalise effectivement le modèle. Cependant, la liaison réciproque n'est pas systématique : il n'est pas prévu de "*remonter*" de l'implémentation vers le modèle du concepteur. Beaucoup d'aspects du modèle sont implicités dans l'implémentation qui en est faite, ce qui est très nuisible à l'analyse des résultats obtenus et réserve le prototypage interactif à l'informaticien.

Prenons un exemple concret. Imaginons que le modélisateur établisse le modèle comportemental d'une créature qui, selon les situations auxquelles elle est exposée, peut marcher ou courir. Si l'informaticien décide d'employer une unique méthode *move* pour simuler ces déplacements, il implicite une part importante du sens de cette action. Dans la réalisation de la partie décisionnelle, il peut invoquer cette méthode avec des paramètres de vitesse différents et ainsi réaliser correctement le modèle qui lui est soumis. Par contre, si lors de la simulation l'analyste ou le modélisateur cherchent à comprendre un comportement observé, ils ne pourront pas disposer d'une information claire sur l'action effectivement exécutée sans prendre en compte des détails d'implémentation (la vitesse en l'occurence). Une simple trace d'appel n'est pas suffisante ("*à T=10ms la méthode move a été invoquée*"). Cet exemple est

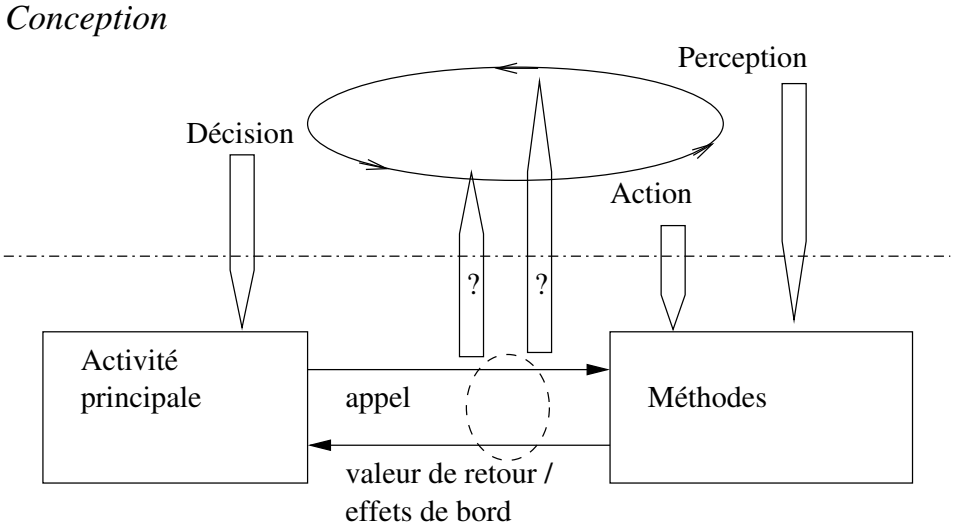

FIG. 4.1 – Du modèle vers l'implémentation : un lien bidirectionnel est nécessaire.

simple, mais les implicitations de ce genre dans l'implémentation compliquent le lien entre les intervenants lorsqu'elles se multiplient. Le lien entre le modèle et son implémentation doit donc être bidirectionnel, et l'établissement de ce lien doit être pris en charge par l'outil pour être systématique.

L'étape d'implémentation doit donc être une étape de transcription et non une simple traduction : il faut minimiser au maximum la perte d'information en restreignant la part d'interprétation faite par l'informaticien. Il est alors nécessaire de prendre garde à maintentir explicitement au sein du modèle les informations qui ont servi à spécifier le modèle comportemental. De plus, l'outil permettant la programmation et la simulation correspondante ne doit pas "gommer" le sens des concepts manipulés au travers des services et des facilités de développement qu'il offre.

Ce constat implique la définition d'une sémantique permettant la spécification. Le choix des concepts manipulables détermine l'ensemble des comportements qu'il sera possible d'exprimer. Il est indispensable de s'imposer une définition stricte du type d'agent que l'on souhaite spécifier afin de délimiter l'ensemble des concepts mis en œuvre. Pour permettre à des utilisateurs non experts de spécifier des comportements, nous préférons restreindre l'ensemble des modèles exprimables et proposer des outils conceptuels plus simples à appréhender, d'un haut niveau d'abstraction qui faciliteront l'intégration de l'ontologie du domaine applicatif.

D'autre part, dans le cadre particulier de l'utilisation d'agents pour les simulations participatives et interactives, la principale caractéristique attendue est la crédibité comportementale. En effet, nous n'attendons pas des agents qu'ils aient un comportement déterministe

défini statiquement et validable formellement. Au contraire, les agents doivent être capables de prises de décision sur des bases imparfaites et dynamiques, voir de faire des erreurs "*cohérentes*" avec le contexte, leur perception de l'environnement pouvant être dégradée et incomplète. Evidemment, ce critère, *a priori* subjectif, ne peut être évalué que par un spécialiste du domaine applicatif abordé. Enfin, rappelons que nous avons pour but de permettre un prototypage "*en ligne*" d'un haut niveau d'abstraction.

Pour résumer nos objectifs, le modèle qui fait l'objet de notre proposition doit :

- s'inscrire au mieux dans le cycle de spécification en intégrant au maximum les concepts métiers de l'analyste et du modélisateur ;
- faciliter l'interprétation des résultats observés au cours de la simulation *via* ces mêmes concepts ;
- permettre un prototypage interactif de haut niveau d'abstraction accessible à l'analyste ou au modélisateur.

Ceci passe d'après nous par l'établissement d'un lien sémantique qui réifie la relation entre la partie *décision* d'une part, et les parties *perception* et *action* d'autre part. Si la même sémantique est employée lors de l'implémentation pour établir le lien entre l'*activité principale* et les *méthodes*, la cohérence recherchée est atteinte : les mêmes concepts sont utilisés dans les différentes phases de spécification. La figure 4.2 résume ce point de vue.

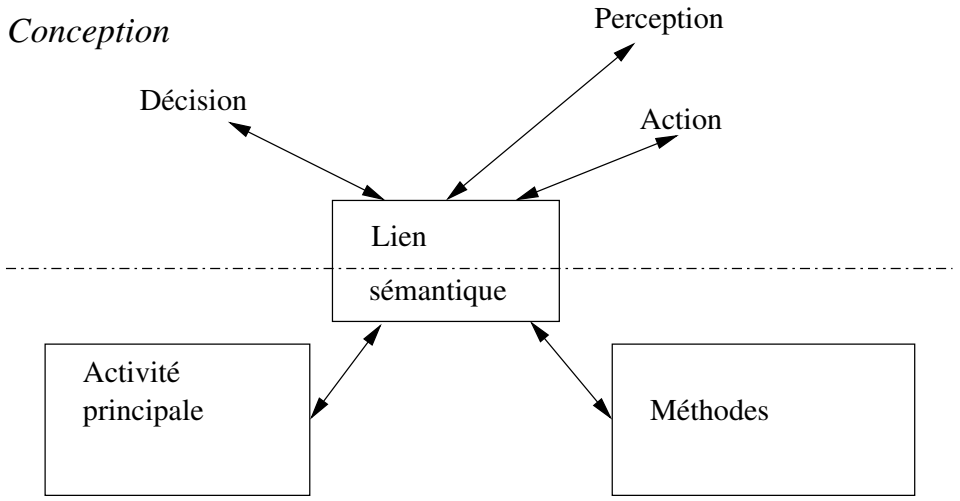

FIG. 4.2 – Du modèle vers l'implémentation : un lien sémantique pour assurer la continuité.

C'est en réponse à ce besoin que la notion d'intention va intervenir dans notre modèle.

4.1.2 Une approche duale

Le chapitre 2 décrit les deux grandes familles de modèles comportementaux : l'approche symbolique et l'approche réactive. Nous commencerons donc par résumer leurs caractéris-

tiques principales dans le cadre de la simulation de systèmes multi-agents avant d'exposer le type d'approche suivi par notre proposition.

Les approches réactives permettent de définir le comportement d'un agent par le biais d'une fonction liant les grandeurs d'entrée et de sortie (principe stimuli/réponses). Cette fonction ne dépendant pas de l'historique du système, ni d'un modèle *a priori* de celui-ci. Les calculs sont souvent aisément automatisables et rapides à traiter, mais ces approches présentent des limitations intrinsèques :

- l'absence de représentation interne de l'environnement limite le comportement à des réflexes de bas niveau : le lien de causalité entre les événements doit être permanent et non contextuel ;
- l'absence d'une sémantique explicitement associée aux mécanismes de sélection d'action dans la plupart de ces modèles interdit toute introspection, ou génération d'explication par l'agent. De plus, l'absence de cette explicitation du modèle au sein du modèle lui-même rend l'apprentissage et le prototypage en ligne délicats.

Quant aux modèles symboliques, ils sont issus des approches symboliques de l'intelligence artificielle, plus particulièrement des techniques de planification dynamique ou de la théorie des actions. Ils permettent d'aborder un niveau conceptuel élevé (notions de croyance, de but…) mais présentent eux aussi des limitations dans leur mise en œuvre :

- la définition de la base de faits relative à un domaine, la détermination de ses règles de mise à jour et de sa taille sont problèmatiques : cette base doit pouvoir représenter l'ensemble des états possibles du monde et l'ensemble des plans susceptibles d'être exécutés ;
- l'utilisation d'un moteur d'inférence logique est particulièrement idoine pour la définition des mécanismes de prise de décision mais inadaptée à l'exécution des actions issues de cette décision : par essence les actions d'un agent fondent son interaction avec son environnement, elles nécessitent donc des effets de bord, ce qui constitue intrinséquement une infraction au paradigme déclaratif logique employé (par *effet de bord* nous entendons modification de l'environnement au sens large).

Ainsi, les approches symboliques apportent un gain en expressivité et en abstraction important mais soulèvent, en simulation, des problèmes de complétude, de gestion des interactions et de cohérence temporelle liés à la dynamicité de l'environnement. D'autre part, les approches réactives permettent un traitement efficace mais sont inadaptées aux systèmes ouverts, donc à la modélisation en ligne, particulièrement par un utilisateur non-expert.

L'approche que nous proposons est donc une hybridation : il s'agit d'une méthode de sélection d'action dont l'idée principale est d'enrichir le modèle d'informations sémantiques sur le sens des actions. Le propos n'est pas de définir une logique de l'action comme en [Pratt, 1990, Lamport, 1994, Baral and Gelfond, 1997], mais d'imposer que chaque action ou perception fournisse une information sur ses effets vis-à-vis de l'agent. C'est ce que JACQUES TISSEAU nomme "le *savoir* sur le *savoir-faire*" pour contribuer à l'autonomisation des modèles [Tisseau, 2001]. Tous les effets de bord susceptibles de concerner l'agent sont

donc décrits selon les préceptes d'une approche réactive, alors que la prise de décision est abordée de la même manière que dans les approches symboliques. Cette mixité d'approche soulève le problème connexe du choix d'un langage d'implémentation pouvant supporter au mieux cette dualité de définition.

4.1.3 Définition intensive et définition extensive

La programmation impérative orientée objet est devenue un standard *de facto*, elle pose pourtant des problèmes d'expressivité pour la définition de comportements : la bonne interprétation d'un programme impératif par l'utilisateur nécessite que celui-ci l'exécute mentalement, au moins partiellement [Dikötter, 1993]. De plus, elle impose une définition intensive : tous les effets de bord doivent être explicités, et l'absence de mécanisme tel que l'unification interdit l'emploi de valeurs indéterminées. Il faut décrire de manière exhaustive tous les cas d'utilisation définissant le comportement et, malgré les possibilités d'héritage et de polymorphisme, il est très délicat de structurer correctement un objet pour qu'il soit extensible et modifiable dynamiquement.
Par exemple, pour une application classique de simulation de comportement proie / prédateur, il faut envisager une architecture logicielle relativement complexe pour pouvoir ajouter ou modifier des comportements de prédation au cours de la simulation. C'est pourtant l'objectif du prototypage en ligne.

Pour sa part, la programmation déclarative logique, telle que l'implémente `Prolog` par exemple, permet une définition extensive : elle facilite le traitement ensembliste de toutes les entités satisfaisant un critère, grâce entre autres mécanismes à l'unification et au retour arrière. Ainsi, il est facile d'écrire une règle qui porte sur *l'ensemble des proies*, pour reprendre l'exemple précédent. De plus, un *programme* logique (la base de faits) est aisément modifiable dynamiquement : si le comportement de notre prédateur est défini par un ensemble de règles comportementales, il suffit d'invalider ou d'ajouter des clauses à ces règles pour modifier instantanément le comportement de prédation.

Ceci tient au fait que pour une implémentation de spécification déclarative logique, et contrairement à l'impératif, le modèle d'exécution et le modèle comportemental ne sont pas confondus : les implémentations des langages déclaratifs logiques reposent sur l'utilisation d'une base de connaissances et d'un moteur d'inférence bien distincts. Le moteur d'inférence traite tous les ensembles de règles de la même façon, modifier la base de faits ne remet donc pas en cause le mécanisme d'exécution du code. A l'inverse, pour une implémentation impérative le comportement est spécifié à l'aide d'instructions de contrôle, modifier ce dernier revient nécessairement à perturber l'ordonnancement du programme. De plus, dans le cadre du prototypage en ligne, la sémantique de haut niveau d'une spécification logique est un atout pour l'utilisateur expert du domaine. Cependant, comme le comportement résulte de l'*extension* des règles comportementales, il est délicat, et parfois même impossible, de prouver formellement les propriétés de ce dernier puisqu'il est utilisé dans le cadre d'un problème ouvert. Ceci n'est pas un obstacle dans notre cadre d'étude : nous ne cherchons pas à générer des simulations à partir du résultat exactement calculé dans une situation, mais plutôt

à visualiser le déroulement de la résolution d'un problème par des agents, même s'il n'existe pas de solution au départ (l'environnement pouvant dynamiquement apporter ou retirer des solutions) [Favier et al., 2001].

En résumé, une méthode écrite dans un langage impératif permettra efficacement de simuler les effets de bord d'une action, par exemple le déplacement d'un agent situé dans l'espace, mais la raison de ce déplacement est beaucoup moins aisée à transcrire dans ce même langage. A l'inverse, un langage relationnel logique est beaucoup plus expressif pour la spécification d'un mécanisme décisionnel, mais très peu efficace à la définition des effets de bord de l'action entreprise.

L'utilisation conjointe de la spécification impérative des actions de l'agent et de la spécification déclarative des règles de comportement semble constituer une réponse possible à la dualité de définition recherchée. Ce choix induit l'architecture du modèle proposé, telle qu'elle est présentée dans la section suivante.

4.1.4 Dualité impératif / déclaratif

Pour résumer les choix introduits dans les sections précédentes, le modèle que nous proposons est fondé sur la sélection d'action basée sur l'explicitation, au sein du modèle, des effets de ces actions. L'idée principale est d'utiliser une définition intensive des attributs de l'agent et de ses *savoir-faire*, conjointement à une définition extensive de sa partie décisionnelle (ses *savoirs*). Les actions, qu'elles soient actions perceptives ou actions d'interaction, sont donc définies de manière impérative, sous forme de méthodes. La prise de décision est alors spécifiée de manière déclarative, elle se ramène à la définition des règles exploitées par un moteur d'inférence, la base de faits utilisée étant renseignée par les méthodes (perception) : en aucun cas l'état de l'agent ou son environnement ne sont modifiés directement par la partie décisionnelle, celle-ci n'a pas d'autre effet de bord qu'une sélection d'action, le paradigme déclaratif logique est enfreint le moins possible.
La figure 4.3 présente une ébauche de l'architecture adoptée faisant apparaître les deux paradigmes retenus : l'impératif et le déclaratif logique.

La possibilité d'une séparation entre la spécification des *savoir-faire* par l'expert des systèmes multi-agents, et la spécification du mécanisme de prise de décision (les *savoirs*) par l'expert du domaine applicatif est ainsi ménagée. De plus, le recours à une définition logique des règles comportementales apporte une sémantique de haut niveau d'abstraction favorable au prototypage en ligne et aux mécanismes d'apprentissage.

Il reste à définir un lien entre ces deux parties qui soit pris en charge par la plateforme elle-même, afin d'assurer la plus grande indépendance possible de ces deux aspects de la spécification. Si l'explicitation de ce lien était à la charge du modélisateur, le modèle ne permettrait pas de s'affranchir des dépendances entre les deux aspects du comportement, donc de séparer les expertises nécessaires. Le modèle proposé impose cette séparation : il n'est pas possible de spécifier le lien entre action et décision de manière applicative, la spécification

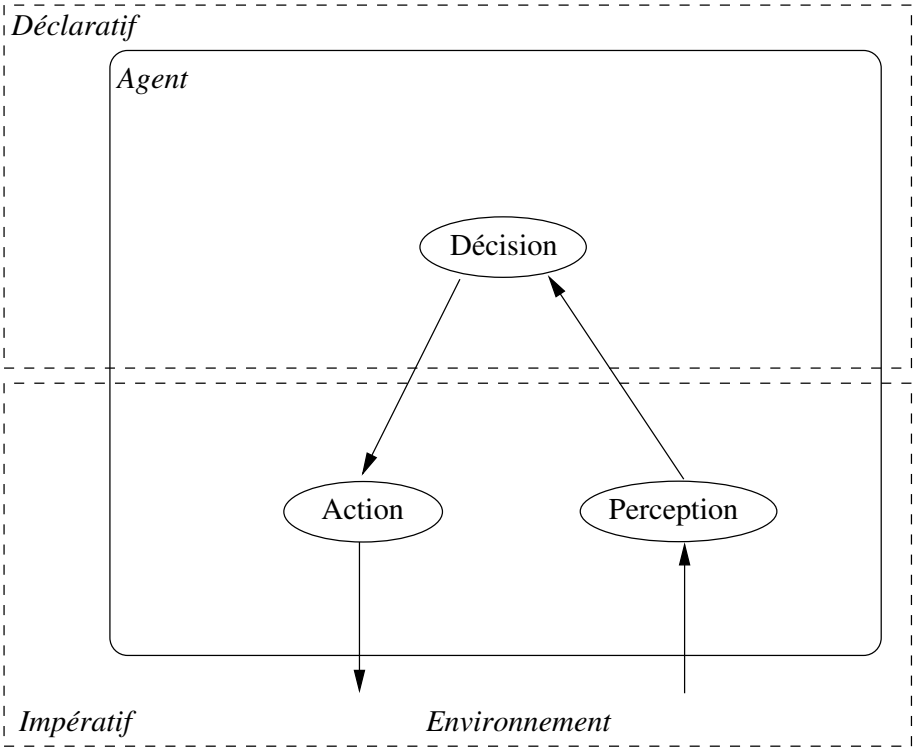

FIG. 4.3 – Une architecture multi-paradigmes.

du comportement doit se faire selon le mécanisme générique du modèle.

Ici encore, le lien sera établi au travers de la notion d'intention. Comme exposé précédemment, notre but est d'utiliser le même lien sémantique entre la partie *décision* et les parties *perception* et *action*, que ce soit lors de la spécification ou lors de l'implémentation. En reprenant la figure 4.2 page 76 à laquelle nous ajoutons cet aspect de lien entre les paradigmes de programmation, nous obtenons le schéma 4.4 page ci-contre qui illustre le double rôle de ce lien sémantique.

Pour résumer ce double rôle, le lien sémantique que nous proposons d'établir doit :
- permettre de manipuler des termes proches des concepts métiers lors de la conception ;
- permettre d'établir un lien entre la partie impérative et la partie déclarative logique du modèle.

Au regard des critiques précédentes sur les approches symboliques (*frame problem*, décidabilité...), l'apport d'un lien sémantique basé sur l'exploitation de symboles au sein d'un modèle réactif semble assez antithétique. Mais il faut garder à l'esprit que les limitations décrites étaient formulées dans le cadre particulier des systèmes ouverts. Or, dans notre cas, nous allons tâcher de restreindre l'ensemble des symboles manipulables au sein de l'agent. Pour expliquer cela, nous devons détailler la manière dont nous traitons les perceptions de l'agent. C'est l'objet de la section suivante.

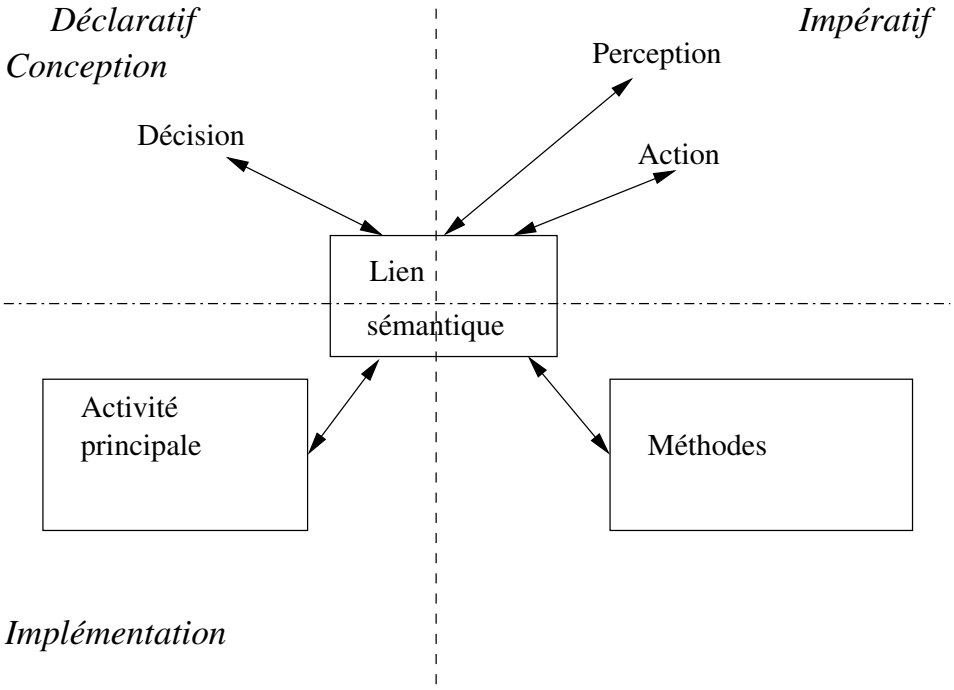

FIG. 4.4 – Le double rôle du lien sémantique.

4.1.5 De l'acte perceptif à la perception : la construction de l'extériorité

Comme nous l'avons exposé en chapitre 2 dans la section 2.2.4 dédiée à la notion d'autonomie (*cf.* page 39), la conception d'un agent omniscient dans un environnement virtuel ne pose aucun problème de mise en œuvre. L'informaticien dispose de la possibilité technique d'introduire des biais dans la réalisation des perceptions d'une entité en lui donnant le moyen d'accéder directement aux propriétés des autres entités de la simulation. Cependant, ceci va à l'encontre de notre velléité de concevoir des modèles comportementaux engendrant des conduites assimilables à celles dont fait montre une entité autonome. Dans cette démarche, il faut favoriser l'acquisition d'informations par l'agent et minimiser au maximum l'implicitation de son emprise avec l'environnement.

Ici encore, une réflexion de fond sur l'intentionnalité constitue un guide intéressant dans cette démarche. En effet, selon les préceptes de la phénoménologie, à l'opposé complet de la doctrine du perceptionnisme, notre esprit n'a aucune forme d'accès à une réalité objective. Nous nous construisons *une représentation* de la réalité médiée par nos perceptions. La variation de point de vue étant notre seul principe de découverte. Il faut ici comprendre le mot *perception* au sens large, en cela qu'il regroupe :

- nos extéroceptions, c'est à dire ce qui nous est extérieur (c'est le sens commun de la perception) d'une part ;
- nos proprioceptions, c'est à dire la perception de notre propre état.

La représentation de la réalité regroupe alors la représentation de ce que nous sommes au

même titre que la représentation de ce qui nous entoure.

La transposition de cette posture à la modélisation d'un agent artificiel conduit à refuser à ce-dernier toute forme d'accessibilité directe au "*sens*" de son environnement : l'agent doit se construire *une représentation* de son environnement. L'intentionnalité de l'agent ne réside pas alors dans cette représentation, mais dans la *construction* de cette représentation : l'enjeu n'est pas de décrire l'aspect interne de l'agent, ni de lui donner accès à ce qui lui est externe, le véritable enjeu est de le doter de mécanismes lui permettant de se *construire* une *extériorité*. La conception d'un agent manipulant explicitement la manière dont il se projette au monde, dont il est entrelacement avec son environnement, d'un agent *situé en intention* serait le franchissement d'une étape essentielle sur le chemin de l'autonomisation des modèles.
Pour notre modèle, nous retiendrons simplement l'idée de restreindre la réflexion à *une représentation* de l'environnement, sans considérer pour notre étude la manière dont l'agent se *construit* cette représentation : elle est acquise au moyen des perceptions.

Ce choix est une des pierres angulaires de notre proposition. Ainsi, l'agent ne doit pas prendre de décision en fonction de son environnement, mais plutôt en fonction de la perception qu'il a de son environnement. C'est une nuance de taille. La décision porte toujours sur quelque chose qui appartient à l'agent, même si la propriété sur laquelle l'agent raisonne peut représenter n'importe quel objet appartenant (ou non) au monde virtuel. Il incombe aux actions perceptives de traduire l'environnement dans l'alphabet symbolique de l'agent. Cet alphabet peut donc être arbitrairement restreint, et les méthodes perceptives plus ou moins efficientes. Nous ne tenterons donc jamais de raisonner directement sur l'état de l'environnement, mais uniquement sur l'état de l'agent, lequel est en partie un reflet de l'environnement par l'entremise des actions perceptuelles.

Selon ce modèle, les perceptions sont en charge de l'adaptation sous forme de symboles des informations pertinentes pour l'agent. C'est pourquoi nous parlons d'actions perceptives. Nous distinguons l'acte de percevoir de la chose perçue[1] : l'action perceptive (regarder) est différente de son éventuelle production (la chose vue). Ainsi, chaque action de perception a pour rôle de produire les symboles utiles au raisonnement. Ce choix permet d'utiliser des méthodes impératives pour l'implémentation de l'action de percevoir et d'établir le lien avec la partie déclarative logique au moyen de la chose perçue. Le fait de recourir à des méthodes impératives est un point important dans le cadre de la simulation d'environnements virtuels en temps réel : l'implémentation de certains "*capteurs virtuels*" requiert la puissance de traitement de ce paradigme. Par exemple, la réalisation de la vision d'une entité spatialement située peut être réalisée par un processus coûteux (en temps de calcul) de lancer de rayons pour simuler au mieux son champ visuel. Le résultat de cette "*vision virtuelle*" sera ensuite filtré selon un modèle de perception strictement applicatif afin de générer les symboles pertinents. Il est important de ménager cette possibilité de filtrage, d'adaptation, pour simuler des perceptions imparfaites ou dégradées, ou pour introduire par la suite une influence des perceptions par l'état interne (le bruit semble plus fort à quelqu'un qui est stressé par exemple).

[1](comme *noèse* et *noème* distinguent l'acte de pensée de la chose pensée, le mot *perception* confondant malheureusement les deux aspects ; il faudrait pour être rigoureux parler d'acte perceptif et de percept)

C'est la différence entre perception et sensation : il est possible de percevoir une chaleur sans avoir de sensation de chaud.

En contrepartie, ce choix implique que les propriétés qui peuvent faire l'objet du raisonnement sont prédéfinies : si l'on dote l'agent de méthodes de perception visuelle, on doit aussi le doter d'un moyen de représentation interne de ce qu'il voit (espace de stockage, de représentation). Et c'est uniquement sur cette représentation que se fait le raisonnement. L'ensemble des représentations symboliques susceptibles d'être manipulées est donc prédéfini. Il est totalement applicatif et reste local à l'agent.
Ceci ne constitue pas un handicap, au contraire : l'écueil de la représentation symbolique de l'environnement est ainsi évité, le raisonnement exploite une représentation locale à l'agent. La prise de décision de ce dernier se base donc sur les connaissances qu'il a de lui-même (ses capacités, ses savoir-faire) et de son état (contexte environnemental, buts poursuivis, humeur...).

4.1.6 Les principes du modèle

En résumé, dans notre modèle nous considérons les perceptions comme des *actions perceptives*, qu'il s'agisse de proprio- ou d'extéroceptions. Les perceptions ont pour rôle de renseigner, éventuellement imparfaitement, l'agent sur son propre état et celui de son environnement. Si ses actions (perceptives ou d'interaction) sont codées en impératif, l'agent doit pouvoir raisonner en fonction de leurs effets, et surtout raisonner pour les déclencher. La programmation relationnelle logique a été proposée pour assurer ce raisonnement. Il faut donc que l'agent possède une interface de transcription symbolique de l'effet de ses actions pour inférer, et ainsi décider. La définition d'une logique de l'action permettrait d'effectuer ce travail de transcription, mais il serait alors nécessaire d'avoir recours à des concepts relativement abstraits pour prendre en compte tous les comportements possibles. Ceci nous éloignerait de la sémantique applicative, ce qui nous écarterait de notre objectif premier : spécifier le comportement selon les termes métier afin de permettre le prototypage en ligne par un expert du domaine.

Le principe du modèle proposé est donc le suivant : chaque action doit fournir les symboles permettant sa prise en compte dans le raisonnement. Le choix et le déclenchement des actions étant fondés par les *intentions* de l'agent, ce modèle est nommé : modèle d'agent intentionnel[2]. Pour les actions perceptives, cela implique de produire des symboles en fonction des valeurs perçues dans le respect de la sémantique du domaine. L'environnement est alors consulté et la base de faits modifiée (renseignée).
Pour les autres actions, la base de faits est consultée (raisonnement pour déclencher l'action) alors que l'environnement est modifié. Ceci implique que chaque action fournisse des symboles permettant la proaction : sans anticipation des effets d'une action, elle ne peut être déclenchée volontairement par l'agent ; elle peut éventuellement l'être par un réflexe pré-

[2]Au regard des nombreuses facettes de la notion d'intention qui ont été présentées, l'appellation d'*agent intentionnel* est ici abusive. Elle est à considérer comme un raccourci pour désigner notre modèle.

programmé, indépendamment de tout raisonnement. Les actions doivent donc produire les symboles indiquant leurs effets présumés. Ces derniers peuvent être codé dans l'action elle-même, ou issu d'un processus d'apprentissage par l'agent. L'effet de l'action concernant soit l'environnement, soit l'agent lui-même, il pourra être représenté par une influence sur une propriété de l'agent puisque ce dernier possède par hypothèse une représentation symbolique de lui-même et de sa perception de l'environnement. L'ensemble des symboles nécessaires est donc défini lors de la conception, dans le respect de la sémantique du domaine abordé.

L'agent raisonne à partir de *savoirs* **sur** ses *savoir-faire* (l'ensemble des symboles produits par les méthodes) en fonction de ses *intentions*.

Les intentions de l'agent sont ici considérées comme sa volonté d'influencer certaines de ses propriétés selon différentes *tendances*. Quatre *tendances* sont définies : *augmenter*, *réduire*, *maintenir* et *indifférent*. Le modèle se base alors sur un mécanisme de résolution qualitative symbolique pour élire la ou les actions satisfaisant au mieux les intentions de l'agent. Ceci permet la sélection d'action même dans le cas ou aucun *savoir-faire* ne correspond exactement au but poursuivi.
Par exemple, si un agent possède un niveau d'énergie faible (par rapport à un seuil défini de manière applicative), sa *proriété faim* est vraie (passage du quantitatif au qualitatif symbolique par les méthodes d'introspection). Ses règles comportementales génèrent alors l'*intention* de *réduire* sa *faim* et il va chercher la ou les actions dont un des effets connus est "*réduire la faim*". Ce mécanisme de résolution est offert par l'outil d'implémentation et est commun à tous les agents : il y a donc bien séparation entre la spécification des *savoirs* et des *savoir-faire*, le lien entre les deux parties étant implicitement pris en charge par le modèle d'exécution. La seule partie applicative est la définition des méthodes de l'agent et la déclaration de leurs effets respectifs. La base de faits impliquée doit pouvoir représenter l'ensemble des perceptions possibles de l'agent, ainsi que l'ensemble des actions qu'il est susceptible d'entreprendre, et non la totalité des états du monde possibles et l'ensemble des plans susceptibles d'être adoptés comme dans les approches symboliques classiques.

La section suivante reprend une à une les différentes parties constitutives du modèle et décrit la manière dont elles se combinent pour réaliser la sélection d'action.

4.2 Les 7 composantes du modèle

Le modèle d'agent intentionnel exploite des règles comportementales conjointement à des connaissances d'arrière-plan pour générer l'ensemble des intentions à satisfaire. La connaissance des effets présumés de ses actions permet à l'agent de sélectionner celles qui satisfont au mieux ses intentions. Ce raisonnement se fait sur la base de ses perceptions. Il a été choisi d'employer l'impératif (C++) pour l'implémentation des actions et perceptions et le déclaratif logique (PROLOG) pour le raisonnement. Les points critiques de cette proposition sont :

– le passage du domaine numérique (les valeurs issues des méthodes de perception) au domaine symbolique (les termes exploités par le processus de décision) ;

– l'explicitation des effets supposés des actions ;
– l'ordonnancement d'activités hétérogènes (impératif / déclaratif).

Si le troisième point relève de choix techniques dans l'implémentation du modèle d'exécution (détaillé au chapitre suivant), les deux premiers ont des implications dans l'architecture du modèle comportemental.

L'architecture proposée est illustrée par la figure 4.5 qui complète l'ébauche présentée en figure 4.3 par les sept composantes du modèle que nous allons maintenant détailler.

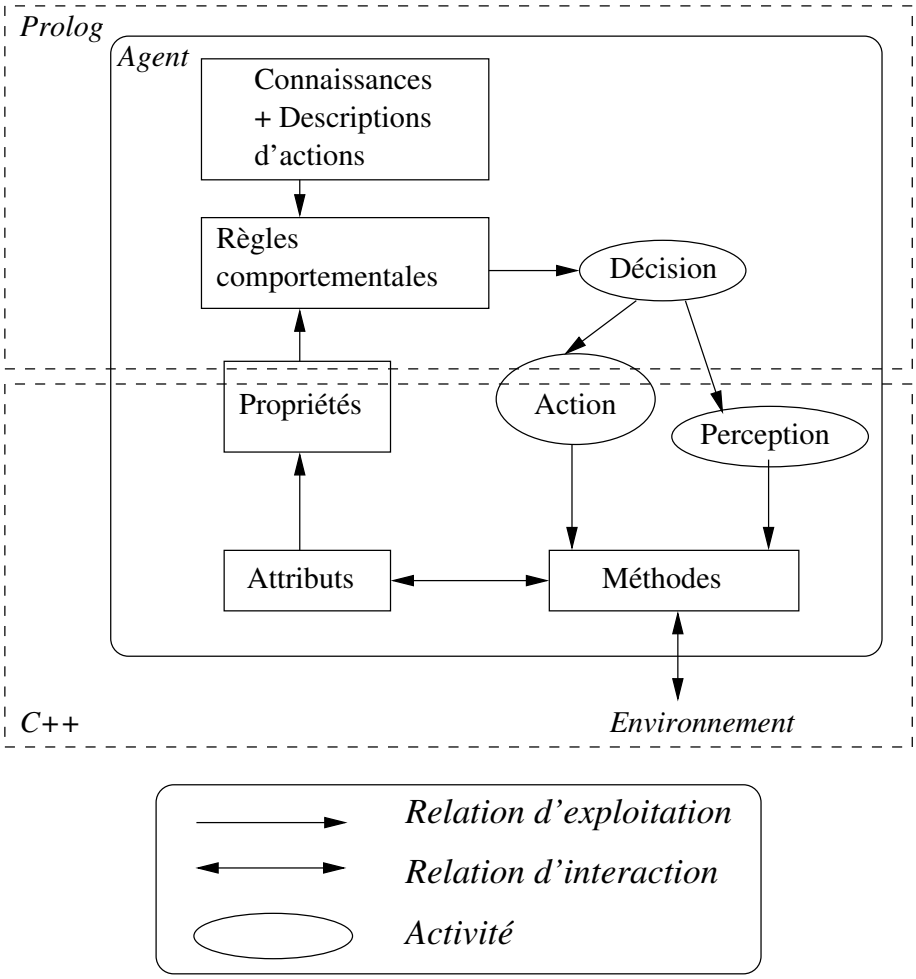

FIG. 4.5 – L'architecture d'agent intentionnel.

Toute la partie rendu étant assurée par le bloc impératif (visualisation de scènes tridimentionnelles, détection de collisions...), l'environnement est noté dans cette partie de la figure. De même, chaque agent sera avant tout un objet, au sens de la programmation orientée objet, ses *méthodes* et *attributs* apparaissent donc dans l'architecture. Les méthodes étant pour l'agent le moyen d'interagir avec l'environnement.

Un des points cruciaux, le passage des valeurs issues des méthodes perceptives au domaine symbolique est effectué par les *propriétés* : il s'agit d'une abstraction des attributs de l'agent. Leur rôle est détaillé dans la suite de cette section.
Enfin, l'architecture fait apparaître trois activités : la *perception*, la *décision* et l'*action*. Notre proposition n'implicite pas un ordonnancement entre ces trois activités (les perceptions permettent une décision qui engendre l'action), nous les considérons comme trois activités se déroulant cycliquement avec leurs fréquences d'évolution propres. Evidemment, sur le plan conceptuel la boucle principale demeure puisque une décision est toujours effectuée sur la base des perceptions déjà acquises, et une action est déclenchée suite à une décision déjà commise. Cependant, nous ne souhaitons pas introduire de point de synchronisme explicite entre les trois activités afin de respecter au mieux les réalités neurophysiologiques : les études tendent à mettre en évidence l'existence de fréquences très différentes pour les processus mentaux impliqués dans le déclenchement et le contrôle de l'action (*cf.* annexe A page 165).
Enfin, la figure fait aussi apparaître les *descriptions d'actions* et les *connaissances* de l'agent qui sont exploitées par l'activité de décision.

Nous allons maintenant détailler les différentes composantes du modèle. Pour cela, nous allons filer un exemple, afin d'illustrer la manière dont elles permettent de spécifier un comportement. L'*exemple jouet* choisi est très classique en réalité virtuelle : il s'agit du comportement d'un prédateur dans un milieu peuplé de proies.

Le comportement retenu pour cette explication sera extrêmement simple, notre but ici étant seulement de situer le rôle des différentes parties du modèle. Ce comportement a ainsi été choisi parce qu'il permet d'illustrer toutes les composantes tout en restant minimaliste. Le prédateur sera caractérisé comme suit :

- l'état de santé du prédateur est défini par un niveau d'énergie ;
- quand il est en pleine santé il se déplace aléatoirement : il se promène ;
- quand il est fatigué il cherche à regagner de l'énergie ;
- il est capable de détecter une proie proche ;
- il est capable de prendre en chasse une proie détectée.

Nous allons maintenant passer en revue les composantes du modèle d'agent intentionnel une à une avec cet exemple comme fil conducteur.

4.2.1 Les méthodes

Les méthodes sont des méthodes dans le sens habituel de la programmation orientée objet. Elles servent à implémenter les actions de l'agent, qu'il s'agisse d'interactions au sens large ou d'actions perceptives. Leur définition incombe à l'informaticien qui doit réaliser les mécanismes correspondant aux spécifications du modélisateur. Elles prennent en charge les spécificités de la plateforme (aspect visualisation par exemple, gestion du son, des collisions…). Afin de satisfaire à l'objectif de prototypage interactif, il est souhaitable d'utiliser un outil permettant l'inspection et la modification dynamique de code. Nous reviendrons sur ce point dans le chapitre suivant.
Dans notre exemple du prédateur, des méthodes de déplacement sont nécessaires pour per-

mettre la réalisation des comportements de marche aléatoire et de poursuite. De plus, il faut réaliser les méthodes correspondant aux perceptions. Ces méthodes ont pour rôle de modifier les attributs de l'agent en fonction de ce qui est perçu (représentation interne de l'environnement). Dans notre cas, une méthode de recherche de proie doit être implémentée. Elle peut par exemple exploiter un mécanisme de lancer de rayons pour la détection des entités de type *proie*. De plus, il faut prévoir les proprioceptions, c'est à dire les méthodes qui permettent à l'agent d'inspecter son propre état.
Enfin, pour des raisons purement techniques il peut être souhaitable de définir des méthodes utilitaires qui ne correspondent pas directement à un comportement de l'agent mais qui sont la base de sa réalisation, afin de factoriser au mieux le code nécessaire.

Ceci conduit dans notre exemple à la réalisation des méthodes suivantes :
- `randomWalk` effectue une marche aléatoire, ce qui influence l'état interne de l'agent (baisse de l'énergie) et l'environnement (déplacement d'une entité spatialement située) ;
- `purchase` effectue un déplacement rapide en direction de la proie la plus proche ;
- `turnLeft` fait pivoter l'agent vers sa gauche, c'est une méthode utilitaire exploitée par les deux précédentes ;
- `turnRight` fait de même vers la droite ;
- `forward` fait avancer l'agent d'un "*pas*", c'est aussi une méthode utilitaire ;
- `findNearestPrey` détecte une proie proche, c'est une méthode de perception (extéroceptive) ;
- `checkEnergy` réduit le "*pas*" de l'agent à 0 si son énergie est trop faible, c'est une méthode de perception (proprioceptive).

Il faut garder à l'esprit que, dans notre modèle, le contrôle de l'action en cours est effectué par la répétition des appels de méthodes. La granularité des méthodes doit donc être faible : la méthode `turnLeft` par exemple fait pivoter l'agent de quelques degrés. Il n'est pas question de faire une méthode qui oriente directement l'agent dans une direction donnée : la méthode `turnLeft` sera appelée de manière répétitive tant que l'intention de l'agent sera de s'orienter plus sur sa gauche.
Le choix de la granularité des actions se pose alors. Il s'agit malheureusement d'un problème intrinsèque en réalité virtuelle : selon la finesse des interactions voulue, les capacités de traitement de la plateforme et la complexité des scènes simulées la granularité sera choisie aussi fine que possible. Dans notre exemple, le pas des rotations sera choisi en regard de la taille des entités : dans le cadre d'une simulation de déplacement, une rotation d'un dixième de degrés est une précision sans doute inutile pour simuler la marche d'un être humain, elle devient critique pour simuler l'orientation d'un outil dans une application de micro-mécanique.

4.2.2 Les attributs

De manière comparable aux méthodes, les attributs de l'agent sont des attributs d'instance au sens habituel de la programmation orientée objet. Ici encore, il est possible de

distinguer deux types d'attributs : ceux qui seront utiles à la définition des comportements et ceux qui sont purement utilitaires. Par exemple, pour pouvoir effectuer le rendu visuel, il est possible que l'objet doive posséder un identifiant de fenêtre, une référence à un arbre de scène ou tout autre valeur propre à la plateforme. L'informaticien doit donc prévoir tout ces attributs, mais en plus il faut prévoir les espaces de représentation pour assurer le stockage des éléments perçus.

Dans notre exemple, nous aurons les attributs suivants[3] :
- `_energy` est un entier qui représente le niveau d'énergie du prédateur ;
- `_step` et `_angleStep` sont des réels correspondant aux pas respectivement d'avance et de rotation de l'entité, il s'agit d'attributs utilitaires ;
- `_nearestPreyX` et `_nearestPreyY` sont les coordonnées de la proie la plus proche perçue dans un repère local à l'agent, il s'agit d'espaces de représentation des perceptions ;
- d'autres attributs peuvent être nécessaires à l'interfaçage avec la plateforme, ils sont à la discrétion de l'informaticien.

4.2.3 Les perceptions

Parmi les différentes méthodes implémentées par l'informaticien, certaines constituent les perceptions de l'agent. Dans notre modèle, l'agent possède une activité de perception dont le rôle est d'invoquer cyliquement ces méthodes à une fréquence donnée. La définition des perceptions se résume donc à la désignation des méthodes qui devront faire l'objet de ces appels.

Dans notre exemple, le prédateur possède deux méthodes de perception :
- `findNearestPrey`
- `checkEnergy`

L'intérêt de cette indirection est de clairement séparer les aspects implémentation du reste de la spécification du comportement : ces mêmes méthodes peuvent éventuellement être exploitées comme méthodes utilitaires par d'autres. Il est donc important de les désigner explicitement comme perception pour éviter les confusions.
La liste des perceptions est donc un des ponts établis entre deux intervenants : le modélisateur n'a pas besoin de connaître les détails de l'implémentation d'une méthode par l'informaticien pour décider qu'elle fait partie des perceptions.

Ici encore, un paramètre demeure applicatif : la fréquence d'appel des perceptions. Selon la lourdeur du traitement et le temps de pertinence de l'information perçue, une action perceptive doit être commise plus ou moins fréquemment. Par exemple, il est admissible qu'un agent mette quelques instants à se rendre compte qu'il est fatigué, par contre sa vision doit

[3](les noms sont ici préfixés par le caractère '_' par convention d'écriture, ceci afin d'être homogène avec le modèle d'exécution présenté dans le chapitre suivant)

être rafraichie beaucoup plus fréquemment pour donner lieu à des comportements crédibles. Nous reviendrons dans le chapitre suivant sur l'ordonnancement des différentes activités au sein de l'agent.

4.2.4 Les propriétés

Les propriétés[4] constituent le point nodal de notre modèle : elles assurent la transcriptions symboliques des perceptions de l'agent.
En plus de permettre le lien entre la partie impérative (perceptions) et la partie déclarative (décision), elles constituent un autre pont entre les rôles de l'informaticien et du modélisateur. En effet, les attributs comprennent des valeurs purement utilitaires et d'autres qui seront utiles à la spécification du comportement. Mais si elles sont utiles, elles ne sont pas pour autant sous une forme aisément exploitable au sein du processus décisionnel. Les propriétés doivent donc assurer une adaptation de ces valeurs d'attributs.

En pratique, nous avons choisi trois types valides de propriété :
– le nombre entier ;
– le nombre réel ;
– la chaîne de caractères.

Dans certains cas, l'adaptation peut être quasi nulle : si une méthode perceptive fournit une information pertinente directement dans l'un de ces types, la propriété est l'image de l'attribut. Dans d'autres cas, les types manipulés par l'informaticien ne se prtent pas du tout au raisonnement. Les propriétés interviennent alors comme une couche d'abstraction entre les attributs et l'activité de raisonnement. Cette indirection est nécessaire car, bien souvent, les attributs "*bruts*" doivent être conservés pour des raisons utilitaires. Une propriété est une combinaison de tout ou partie des attributs.
Considérons notre exemple du prédateur. Pour des raisons techniques, l'informaticien peut avoir intérêt à représenter la position de la proie la plus proche sous la forme de ses coordonnées cartésiennes. Par contre, pour la modélisation du comportement, l'information pertinente pour le prédateur est la distance à la proie et la direction dans laquelle elle se trouve. Il faut donc créer deux propriétés qui donneront la distance et la direction en exploitant l'espace de représentation des perceptions (en l'occurence, les attributs `_nearestPreyX` et `_nearestPreyY`).

Les propriétés sont donc des fonctions écrites dans le langage impératif mais qui peuvent être invoquées à tout moment par la partie déclarative pour prendre en compte les informations perçues sous une forme exploitable. Au sein de ces fonctions, les informations peuvent donc être traitées (mise à l'échelle, seuillage, utilisation de logique floue…) afin d'obtenir

[4]Il faut considérer ici le mot *propriété* dans son sens premier : les propriétés de l'agent sont des représentations de lui-même et de son environnement qui lui sont propres, ce sont des appropriations des états perçus. Le mot pouvant aussi désigner dans un sens plus abstrait une caractéristique (habituellement invariante) il peut prêter ici à confusion. Par exemple, nous dirons d'un agent qui raisonne sur sa température interne qu'il exploite sa propriété température, alors que dans le sens second du terme il s'agirait d'un paramètre et non d'une propriété (*être adiabatique* serait un exemple de propriété).

des symboles directement exploitables par l'activité de décision.
Un autre intérêt de cette architecture est que ce travail d'interrogation et d'adaptation est fait à la demande de la partie décisionnelle, de manière totalement asynchrone avec l'activité de perception. Des perceptions fréquentes n'entrainent donc pas un traitement systématique par le moteur d'inférence. Ce point est détaillé dans le chapitre suivant.

Pour notre prédateur, le comportement décrit est tellement simple que les attributs renseignés par les perceptions sont très proches des propriétés utiles au raisonnement. La position de la proie la plus proche est toutefois adaptée. Les propriétés du prédateur sont donc les suivantes :

- `energy` est l'image de l'attribut `_energy` ;
- `nearestPreyDistance` est calculée à partir des attributs `_nearestPreyX` et `_nearestPreyY` ;
- `nearestPreyAngle` est obtenue sur la même base.

L'ensemble des composantes présentées jusqu'à présent recouvrent la totalité du rôle de l'informaticien. Une fois qu'il a implémenté les actions demandées par le modélisateur selon les spécificités de la plateforme et qu'il a fourni les fonctions permettant de transcrire les valeurs pertinentes dans le domaine symbolique, il a rempli sa part du travail. Il reste au modélisateur à exprimer la génération des intentions de l'agent en fonction de ces valeurs afin de sélectionner l'action la plus appropriée à la situation. Ce choix sera cycliquement consulté est exécuté par l'activité d'*action* (*cf.* chapitre suivant).

4.2.5 Les règles comportementales

Le but des règles comportementales est de générer les intentions de l'agent. Dans ce modèle, les intentions sont simplement une tendance à satisfaire qui porte sur une propriété de l'agent. Par défaut il existe quatre tendances (*increase*, *decrease*, *keep* et *independent*). Dans l'exemple du prédateur, le comportement de pourchasse d'une proie ne peut être déclenché que si le prédateur a l'intention d'augmenter son énergie. Il faut donc qu'il existe un règle comportementale produisant l'intention : `increase energy`.
Une même règle comportementale est susceptible de générer plusieurs intentions qui concernent différentes propriétés. La production d'une telle règle est donc une liste d'intentions. De plus, afin de prendre en compte des comportements plus complexes, chaque intention est accompagnée d'une force qui indique son degré de nécessité. Par défaut, quatre forces symboliques sont proposées dans le modèle :

- `weak`
- `normal`
- `medium`
- `strong`

L'ensemble des forces symboliques manipulées, comme l'ensemble des tendances prises en compte, peut être étendu de manière applicative. Nous reviendrons sur ce point dans la suite de cette section.

Une règle comportementale produit donc une liste de doublets {*intention, force*}, une intention étant elle-même un doublet {*tendance, propriété*}.
pour la suite nous noterons :

- b une règle comportementale (*behavioural rule*) ;
- $\mathcal{B}$ l'ensemble des règles comportementales de l'agent ;
- t une tendance (*trend*) ;
- $\mathcal{T}$ l'ensemble des tendances valides ;
- p une propriété (*property*) ;
- $\mathcal{P}$ l'ensemble des propriétés de l'agent ;
- s une force (*strength*) ;
- $\mathcal{S}$ l'ensemble des forces valides ;
- i une intention (*intention*), donc un doublet $\{t, p\}$;
- $\mathcal{I}$ l'ensemble des intentions de l'agent (liste de doublets $\{i, s\}$).

Ainsi, les règles comportementales génèrent une liste de la forme :

$$\{\{\{t_0, p_0\}, s_0\}, \ldots, \{\{t_n, p_n\}, s_n\}\}$$

Ce qui peut se noter plus simplement :

$$\{\{i_0, s_0\}, \ldots, \{i_n, s_n\}\}$$

Pour une règle comportementale b donnée, nous noterons cette liste $\mathcal{L}_d$ (*list of desired intention*).

Pour déclencher le comportement de prédation, notre prédateur doit posséder au moins une règle comportementale qui produise l'intention suivante :
`{{increase,energy}, normal}`

La production de cette intention est nécessaire au déclenchement volontaire de l'action de pourchasse.
Nous avons ici choisi arbitrairement la force *normal* pour dénoter l'importance de cette intention.

Lorsqu'il doit sélectionner une action, l'agent commence donc par examiner l'ensemble $\mathcal{B}$ de ses règles comportementales afin de générer l'ensemble $\mathcal{I}$ des intentions à satisfaire. Mais toutes ces règles ne sont pas actives à chaque instant : elles possèdent toutes des conditions de validité. Dans notre exemple, le prédateur ne prend une proie en chasse que s'il est fatigué et si il a détecté une proie à proximité. L'intention ne sera produite par la règle comportementale correspondante que si ses prémices sont vérifiées.

En notant c les conditions et $\mathcal{C}$ l'ensemble de ces conditions, la forme d'une règle comportementale est alors :

$$b(\{c_0, \ldots, c_j\}, \{\{\{t_0, p_0\}, s_0\}, \ldots, \{\{t_n, p_n\}, s_n\}\})$$

Ce qui peut se noter :

$$b(\mathcal{C}, \mathcal{L}_d)$$

avec $\mathcal{C} = \{c_0, \ldots, c_j\}$ et $\mathcal{L}_d = \{\{i_0, s_0\}, \ldots, \{i_n, s_n\}\}$

L'expression des conditions d'application d'une règle comportementale implique la capacité de consulter la valeur des propriétés de l'agent et d'effectuer des tests logiques sur ces valeurs. C'est par hypothèse le rôle des propriétés, nous détaillerons dans le chapitre suivant les mécanismes requis pour l'implémentation de cette fonctionnalité.
Pour l'instant, nous noterons simplement les conditions d'application sous forme littérale, ce qui nous donne pour le prédateur une première règle comportementale b_0 :

$$b_0(\{tired, prey_is_near\}, \{\{\{increase, energy\}, normal\}\})$$

Dans le processus de sélection d'action, la première étape est donc d'examiner l'ensemble $\mathcal{B}$ des règles comportementales de l'agent. Les règles de cet ensemble dont les conditions de validité sont vérifiées génèrent l'ensemble $\mathcal{I}$ des intentions de l'agent. Parmi ces intentions, plusieurs peuvent concerner les mêmes propriétés et être éventuellement contradictoires. Ceci va dans le sens de notre démarche initiale : un agent autonome doit être capable de sélectionner une action même lorsqu'il évolue dans un environnement surcontraint. L'agent va devoir faire une compromission et choisir l'action la moins défavorable à la satisfaction de l'ensemble de ses intentions. Pour cela, il doit comparer les forces de chaque intention pour ses différentes propriétés et élire l'action qui répond au mieux à ses besoins.
Mais pour ce faire, l'agent doit posséder la connaissance de l'effet présumé de ses actions. C'est le rôle des descriptions d'action que nous allons maintenant présenter.

4.2.6 Les descriptions d'action

Les descriptions d'action ont pour but d'expliciter au sein du modèle le *sens* de l'action. Nous défendons l'idée qu'une action peut être sélectionnée par l'agent parce qu'il en connaît les effets. Dans notre modèle, un agent raisonne sur une représentation de lui-même et de son environnement. Cette représentation prend la forme de propriétés qui sont la transcription dans le domaine symbolique des productions des actions perceptives. L'effet des actions doit être exprimée en fonction de ces propriétés. Une description d'action est donc une loi d'association entre le nom de l'action et la liste de ses effets présumés sur certaines propriétés. Ces effets sont modélisés par la notion d'intention (doublet {*tendance*, *propriété*}). Cette

liste peut ainsi être directement comparée avec l'ensemble $\mathcal{I}$ des intentions de l'agent afin d'évaluer la pertinence de l'action dans le contexte.

Nous adopterons les notations suivante :
- d est une description d'action ;
- $\mathcal{D}$ est l'ensemble de ces descriptions pour un agent ;
- a est une action de l'agent ;
- $\mathcal{A}$ est l'ensemble de ces actions ;
- $\mathcal{L}_e$ est la liste des effets présumés d'une action dans une description d'action (*list of expected trend*).

Une description prend alors la forme suivante :

$d(a, \{\{t_{a_0}, p_0\}, \ldots, \{t_{a_n}, p_n\}\})$ où t_{a_j} désigne la tendance sur la propriété p_j résultante de l'action a (effet présumé)

ou plus simplement :

$$d(a, \mathcal{L}_e) \text{ avec } \mathcal{L}_e = \{i_0, \ldots, i_n\}$$

Il est important de noter qu'il s'agit ici de l'expression des effets *présumés* de l'action. Un agent peut posséder un modèle erroné de ses propres actions. Il peut même être nécessaire d'introduire un biais au niveau de la description d'une action en fonction de la granularité de cette-dernière. Par exemple, pour notre prédateur nous avons choisi d'utiliser une unique méthode de pourchasse pour simuler le comportement de prédation. Ce n'est pas une action de granularité très fine. Nous allons lui associer l'effet présumé d'augmentation du niveau d'énergie, or il s'agit d'un effet à moyen terme : une poursuite qui n'aboutit pas fatigue le prédateur sans lui permettre de dévorer sa proie. L'effet est alors une baisse d'énergie. Pour éviter ce biais et aboutir à un comportement plus fin, il faudrait raisonner sur le niveau d'énergie et sur la distance à la proie : la pourchasse fatigue mais permet d'approcher la cible. Nous verrons par la suite qu'il est possible de mettre en place des mécanismes d'apprentissage ou de planification sur la base de cette sélection d'action. L'effet présumé des actions peut alors être appris par l'agent au lieu d'être spécifié par le modélisateur. Pour ce premier exposé, nous en resterons au comportement simpliste que nous avons desormais presque entièrement spécifié.

Le prédateur, pour pouvoir choisir l'action de poursuite, doit posséder la description de ses effets, ce qui se traduit par la description d'action d_0 suivante :

$$d_0(purchase, \{\{energy, increase\}, \{preyDistance, decrease\}\})$$

Comme illustré ici, la description d'une action doit être aussi complète que possible : l'effet de la poursuite sur la distance à la proie est indiqué alors que nous n'avons défini aucune règle comportementale portant sur cette propriété. C'est un des intérêts du modèle : les effets des actions sont explicités indépendamment de tout contexte. Il est tout à fait possible

que des règles comportementales soient ajoutées ou modifiées en cours d'exécution. Cela peut être le fait du modélisateur (prototypage en ligne) ou des interactions de l'agent avec son environnement (apprentissage) ou avec d'autre agents (comportements sociaux). Notre but est de fournir un modèle qui puisse servir facilement de base à la mise en place de tels mécanismes de manière applicative. L'apport sémantique que constituent les indirections introduites par les règles comportementales et les descriptions d'actions permettent d'imaginer réaliser aisément de telles extensions. Par exemple, un agent pourrait communiquer avec un autre pour lui demander s'il connaît une action qui produit certains effets.

La définition des règles comportementales et des descriptions d'actions constitue l'essentiel du rôle du modélisateur. Il peut toutefois avoir besoin de recourir à l'ajout de connaissances supplémentaires au sein de l'agent pour spécifier complètement son comportement.

4.2.7 Les connaissances

Pour exprimer des règles comportementales complexes, il peut être utile au modélisateur d'avoir la possibilité de définir des règles de traitement utilitaires. De plus, si nous souhaitons que ce modèle puisse constituer la base de modèles plus évolués comprenant par exemple des capacités d'apprentissage ou de génération d'explication, il faut doter l'agent de capacités de mémorisation. Les connaissances ont été ajoutées à l'architecture pour répondre à ces besoins.
En pratique, il s'agit de faits ou de règles PROLOG, puisque c'est la langage qui a été choisi pour l'implémentation de la partie décisionnelle.
Pour satisfaire aux besoins du prototypage interactif et permettre de créer des comportements plus complexes, ces connaissances doivent avoir une granularité instance. C'est à dire que chaque agent possède ses propres connaissances et que l'altération de sa base de faits ne doit pas affecter les autres agents. De plus, afin d'augmenter le niveau d'expressivité, il est souhaitable de disposer de mécanismes d'héritage et de surcharge : chaque agent doit par défaut posséder les connaissances propres à sa classe et pouvoir surcharger ces connaissances. Les implications techniques d'un tel choix sont détaillées dans le chapitre suivant.

Pour tirer le meilleur parti de ce modèle, les règles comportementales et les descriptions d'actions prennent la forme de connaissances particulières. Ceci implique qu'un agent peut éventuellement enrichir ou modifier ses règles comportementales indépendamment des autres entités de la même classe. De même, certains éléments qui servent à l'évaluation de la pertinence des actions sont aussi modifiables. C'est le cas des forces symboliques, des tendances et de la politique d'évaluation (ce dernier élément est détaillé dans la section suivante). Il est donc possible de modifier de manière applicative le processus de sélection d'action en ajoutant un type de tendance par exemple. Cela peut être fait pour tous les agents, pour une classe d'agent ou pour un agent en particulier.

Nous avons décrit les sept composantes du modèle d'agent intentionnel. L'informaticien est en charge de l'implémentation des actions, il dispose pour cela des méthodes et des attributs. Il doit aussi écrire les propriétés qui sont des fonctions d'abstraction des attributs

permettant de fournir les informations sous forme symbolique à la partie décisionnelle. Le modélisateur quant à lui doit désigner explicitement les actions qui sont des perceptions, énoncer les règles comportementales et les descriptions d'action. Il dispose pour cela des connaissances qui lui permettent d'exprimer plus facilement des traitements complexes.
Ces éléments permettent de générer l'ensemble des intentions de l'agent dans un contexte donné par l'évaluation des conditions de validité des règles comportementales. Ils nous reste maintenant à décrire la manière dont ces intentions sont confrontées aux descriptions d'actions pour assurer la sélection d'action.

4.3 Le principe de sélection

A l'issu de l'examen de l'ensemble $\mathcal{B}$ des règles comportementales valides, le processus de sélection d'action dispose de l'ensemble $\mathcal{I}$ des intentions de l'agent. Nous rappelons que cet ensemble est une liste de doublets $\{i, s\}$. La forme de $\mathcal{I}$ est donc :

$$\mathcal{I} = \{\{i_0, s_0\}, \ldots, \{i_n, s_n\}\}$$

Parmi les différentes intentions i de cet ensemble, plusieurs peuvent concerner la même propriété. Or, la pertinence d'une action va être évaluée en fonction de son effet vis à vis des différentes propriétés qu'elle affecte. La première étape est donc la génération d'une intention à satisfaire unique pour chaque propriété. Nous noterons $\mathcal{P}'$ le sous-ensemble de $\mathcal{P}$ des propriétés effectivement présentes dans $\mathcal{I}$. L'ensemble $\mathcal{I}$ est donc subdivisé en plusieurs listes de doublets $\{t, s\}$ correspondant chacune à une propriété. Nous noterons ces listes $\mathcal{I}_p$. Nous avons alors :

$$\mathcal{I} = \bigcup_{k \in \mathcal{P}'} \mathcal{I}_{p_k} \text{ avec } n = card(\mathcal{P}') \text{ propriétés distinctes dans } \mathcal{I}$$

Chaque ensemble $\mathcal{I}_p$ est de la forme :

$$\mathcal{I}_p = \{\{t_0, s_0\}, \ldots, \{t_{n'}, s_{n'}\}\} \text{ avec } n' \leq n$$

Cet ensemble est ensuite traité afin d'associer une unique tendance à satisfaire pour chaque propriété. Les éventuelles intentions contradictoires sont ainsi fusionnées : la tendance dont la somme des forces associées est la plus forte l'emporte. Nous noterons t_{d_p} cette tendance unique associée à p (*desired trend*).
Nous avons alors :

$$t_{d_p} = \mathsf{t} \; ssi \; \forall \, \mathsf{t}' \in \mathcal{T}, \; \sum_{t_j = \mathsf{t}} s_j > \sum_{t_k = \mathsf{t}'} s_k \text{ avec } \mathsf{t} \in \mathcal{T} \; et \; \mathsf{t} \neq \mathsf{t}'$$

Si il y a égalité entre les sommes des forces associées à différentes tendances, et que ces *ex æquo* possèdent le score le plus élevé, la tendance *independent* est retenue. Ceci évite d'introduire un biais par le choix de l'ordre dans lequel se fait l'évaluation.

A l'issu de ce traitement, l'ensemble $\mathcal{I}$ des intentions, éventuellement contradictoires, de l'agent a été divisé en un ensemble de doublets {*tendance*, *force*} pour chaque propriété présente dans $\mathcal{I}$. Cette transformation de $\mathcal{I}$ ne comprend donc plus d'intentions contradictoires, et chaque intention est pondérée d'une force s_p qui est issue de la sommation des forces correspondantes à cette intention dans $\mathcal{I}$.

Nous noterons cet ensemble $\mathcal{I}'$. Il possède le même type de structure que $\mathcal{I}$ mais chaque propriété n'y apparaît qu'une fois.

$$\mathcal{I}' = \{\{i_0, s_0\}, \ldots, \{i_n, s_{p_n}\}\} \text{ avec } n = card(\mathcal{P}')$$

Cet ensemble construit, il est désormais possible d'évaluer la pertinence de chaque action grâce aux descriptions d'actions. Nous rappelons qu'une description d'action a pour forme :

$$d = \{a, \mathcal{L}_e\} \text{ avec } \mathcal{L}_e = \{i_0, \ldots, i_k\}$$

En comparant la liste $\mathcal{L}_e$ des effets présumés d'une action avec l'ensemble $\mathcal{I}'$ des intentions non contradictoires pondérées par leur nécessité, il est possible de définir une fonction de coût calculant la pertinence de cette action dans ce contexte.
Pour cela il nous faut disposer d'une politique d'évaluation des tendances. Notons-là Π. Son rôle est de donner un facteur f_{a_p} à appliquer à la force qui pondère l'intention selon la tendance voulue par l'agent (t_{d_p}, *desired trend*) et la tendance qui résulte de l'effet présumé de l'action a (t_{a_p}, *action trend*) pour une propriété p donnée.

$$f_{a_p} = \Pi(t_{a_p}, t_{d_p}) \text{ avec } \Pi(\mathcal{T} \times \mathcal{T} \rightarrow \mathbb{R})$$

Le calcul de la valeur v_a d'une action (sa pertinence) au regard des intentions de l'agent s'effectue donc comme suit :

$$v_a = \sum_{p \in \mathcal{P}'} f_{a_p} * s_p$$

Ainsi, une action candidate "*obtient*" des points lorsque sa description garantit une tendance qui abonde dans le sens de la tendance intentionnée par l'agent sur une propriété. Par contre elle "*perd*" des points si son effet va à l'encontre de l'intention de l'agent. L'expression d'un effet sur une propriété qui n'appartient pas à $\mathcal{P}'$ (sur laquelle ne porte aucune intention de l'agent) est neutre dans l'évaluation de la pertinence de l'action. Il en va de même si aucun effet n'est déclaré sur une des propriétés appartenant à $\mathcal{P}'$ (l'effet de l'action sur cette propriété est inconnue, elles sont considérées indépendantes).

La politique Π présente par défaut dans le modèle est la suivante :

	independent	increase	keep	decrease
independent	0	0	0	0
increase	0	1	-1	-1
keep	0	-1	1	-1
decrease	0	-1	-1	1

Comme indiqué dans la section précédente, cette politique d'évalutation est définie par une succession de faits, des connaissances particulières de l'agent. Il est donc possible de modifier ou d'étendre cette matrice pour s'adapter à l'extension de l'ensemble des tendances par exemple.

L'activité *action* consulte cycliquement l'ensemble $\mathcal{V}$ des valeurs de pertinence ainsi calculées et exécute la ou les actions les plus appropriées au contexte. Par défaut, l'action possédant la valeur v_a maximum est exécutée. Il est possible d'affiner cette sélection en filtrant éventuellement $\mathcal{V}$ pour en extraire plusieurs actions compatibles et les éxécuter toutes. Ceci impliquerait d'enrichir le modèle de connaissances sur les actions non contradictoires qui peuvent être exécutées simultanément.

Voici le résumé du processus de sélection d'action ainsi défini :

Etape 1 : L'évaluation de l'ensemble des règles comportementales de l'agent permet, en exploitant l'ensemble des perceptions, de générer l'ensemble des intentions (éventuellement contradictoires) de l'agent.

$$\mathcal{B} \times \mathcal{P} \rightarrow \mathcal{I} = \{\{i_0, s_0\}, \ldots, \{i_n, s_n\}\}$$

Etape 2 : Cet ensemble qui porte sur n propriétés différentes est subdivisé en n sous-ensembles portant chacun sur une propriété distincte

$$\mathcal{I} = \bigcup_{k \in \mathcal{P}'} \mathcal{I}_{p_k} \text{ avec } \mathcal{I}_p = \{\{t_0, s_0\}, \ldots, \{t_{n'}, s_{n'}\}\}\ ,\ n' \leq n$$

Etape 3 : Au sein de chacun de ces sous-ensembles, les tendances éventuellement contradictoires sont fusionnées pour se réduire à une unique tendance pondérée par la sommation des forces impliquées.

$$\mathcal{I}' = \bigcup_{k \in \mathcal{P}'} \mathcal{I}'_{p_k} \text{ avec } \mathcal{I}'_p = \{t_{d_p}, s_p\}$$
$$\text{et } t_{d_p} = \mathsf{t}\ ssi\ \forall\, \mathsf{t}' \in \mathcal{T},\ \sum_{t_j = \mathsf{t}} s_j > \sum_{t_k = \mathsf{t}'} s_k \text{ avec } \mathsf{t} \in \mathcal{T}\ et\ \mathsf{t} \neq \mathsf{t}'$$

Etape 4 : La confrontation de l'ensemble obtenu et de l'ensemble des descriptions d'action permet de déterminer la valeur de pertinence des différentes actions.

$$\mathcal{D} \times \mathcal{I}' \rightarrow \mathcal{V}$$
$$\text{avec } v_a = \sum_{p \in \mathcal{P}'} f_{a_p} * s_p$$
$$\text{et } f_{a_p} = \Pi(t_{a_p}, t_{d_p})\ ,\ \Pi(\mathcal{T} \times \mathcal{T} \rightarrow \mathbb{R})$$

Etape 5 : L'activité *action* exécute la ou les actions qui obtiennent le meilleur score. Par défaut, seule la meilleure action est exécutée.

4.3.1 Exemple de calcul

Nous allons illustrer un cas de calcul d'une sélection en prenant un exemple simple. Nous considérons un aérostier dans une mongolfière. En vol, son comportement est défini par les deux règles suivantes :

- afin de profiter le plus longtemps possible de son vol, il cherche à maintenir une altitude constante en évitant si possible de larguer du lest (la réserve de gaz est une ressource moins critique que le nombre de sacs de sable embarqués) ;
- aux abords d'un relief, il est urgent d'augmenter l'altitude de l'aérostat, en préservant au mieux les réserves de lest et de gaz.

Ces deux propositions donnent lieu à la définition des règles comportementales suivantes :

$$b_0 \quad (\{flying\}, \quad \{\{\{altitude, keep\}, normal\}, \{\{weight, keep\}, weak\}\})$$
$$b_1 \quad (\{flying, \quad near_obstacle\},$$
$$\{\{\{altitude, increase\}, strong\}, \{\{weight, keep\}, weak\}, \{\{gas, keep\}, weak\})$$

Pour contrôler son appareil, l'aérostier dispose de deux actions possibles :

- il peut larguer du lest, nous noterons cette action *ballast*, l'altitude de la mongolfière augmente alors et le poids embarqué diminue ;
- il peut actionner le brûleur, nous noterons cette action *burner*, l'altitude augmente aussi, mais la réserve de gaz diminue.

Les deux descriptions d'action correspondantes sont respectivement d_0 et d_1 :

$$d_0(ballast, \{\{altitude, increase\}, \{weight, decrease\}\})$$

$$d_1(burner, \{\{altitude, increase\}, \{gas, decrease\}\})$$

Nous allons maintenant examiner la sélection dans le cas où l'aéronef se trouve en vol à proximité d'un relief : les deux règles comportementales sont alors valides simultanément. Voici les cinq étapes du processus :

Etape 1 : Les deux règles comportementales étant actives, l'ensemble $\mathcal{I}$ des intentions de l'agent (le pilote) est :

$$\mathcal{I} = \{\{\{altitude, keep\}, normal\}, \{\{weight, keep\}, weak\},$$
$$\{\{altitude, increase\}, strong\}, \{\{weight, keep\}, weak\}, \{\{gas, keep\}, weak\}\}$$

Etape 2 : Certaines propriétés apparaissent deux fois dans l'ensemble $\mathcal{I}$, avec des tendances ou des forces différentes. L'ensemble $\mathcal{I}$ est donc divisé en trois ensembles $\mathcal{I}_p$ concernant chacun une propriété :

$$\begin{aligned}
\mathcal{I}_{p_{altitude}} &= \{\{increase, strong\}, \{keep, normal\}\} \\
\mathcal{I}_{p_{weight}} &= \{\{keep, weak\}, \{keep, weak\}\} \\
\mathcal{I}_{p_{gas}} &= \{\{keep, weak\}\}
\end{aligned}$$

Etape 3 : Pour chacun de ces ensembles, les tendances sont fusionnées. La tendance la plus forte l'emporte, la nouvelle force est obtenue par sommation :

$$\begin{aligned}
\mathcal{I}'_{p_{altitude}} &= \{increase, strong\} \\
\mathcal{I}'_{p_{weight}} &= \{keep, 2 * weak\} \\
\mathcal{I}'_{p_{gas}} &= \{keep, weak\}
\end{aligned}$$

Ce qui donne le nouvel ensemble $\mathcal{I}'$ des intentions du pilote :

$$\mathcal{I}' = \{\{\{altitude, increase\}, strong\}, \{\{weight, keep\}, 2{*}weak\}, \{\{gas, keep\}, weak\}\}$$

Etape 4 : La valeur de pertinence de chaque action pour cette situation peut alors être calculée grâce aux descriptions d'action :

$$\begin{aligned}
v_{ballast} &= \sum_{p \in \{altitude, weight, gas\}} f_{ballast_p} * s_p \\
&= \Pi(increase, increase) * strong \\
&+ \Pi(decrease, keep) * 2 * weak \\
&+ \Pi(independent, keep) * weak \\
&= 1 * strong + (-1) * 2 * weak + 0 * weak \\
&= strong - 2 * weak
\end{aligned}$$

$$\begin{aligned}
v_{burner} &= \sum_{p \in \{altitude, weight, gas\}} f_{burner_p} * s_p \\
&= \Pi(increase, increase) * strong \\
&+ \Pi(independent, keep) * 2 * weak \\
&+ \Pi(decrease, keep) * weak \\
&= 1 * strong + 0 * 2 * weak + (-1) * weak \\
&= strong - weak
\end{aligned}$$

Etape 5 : Avec les valeurs de pertinence calculées, l'aérostier choisira d'actionner le brûleur. Ceci correspond au résultat attendu : même dans le besoin urgent de gagner de l'altitude, le souci latent d'éviter de larguer du lest en vol a influencé le choix.

4.4 Conclusion : un modèle et une démarche

Nous avons proposé un modèle de sélection d'action basé sur l'explicitation des effets présumés des actions au sein du modèle *via* la notion d'intention[5]. Dans le cadre de cette proposition, une intention est une tendance à satisfaire qui porte sur une propriété, donc sur une abstraction de l'état de l'agent et de sa perception de l'environnement. Ces tendances sont exprimées de manière qualitative et permettent d'évaluer la pertinence des actions de l'agent dans un contexte donné. La politique de sélection proposée par défaut dans le modèle est très simple mais elle permet de spécifier des comportements réactifs élémentaires.

Le véritable intérêt de ce modèle réside dans le fait qu'il est extensible très facilement et qu'il constitue ainsi une base à la réalisation de modèles plus complexes. Il ne limite pas le modélisateur à la spécification de comportements purement réactifs : des indirections entre les perceptions et la partie décisionnelle ont été introduites. Ces indirections, à savoir les propriétés, les règles comportementales et les descriptions d'action, introduisent une sémantique permettant de fonder un comportement délibératif. En effet, les concepts manipulés au sein du modèle sont suffisamment proches du domaine abordé pour favoriser l'expression de mécanismes décisionnels complexes : le lien entre les perceptions et les actions n'est plus implicité dans l'implémentation, contrairement aux architectures purement réactives.

Enfin, le modèle d'agent intentionnel proposé ici est plus intéressant pour la démarche qu'il nous conduit à adopter que pour l'architecture effectivement proposée. L'abord de la conception d'un modèle comportemental au travers de l'étude de la notion d'intention nous a conduit à mettre en perspective le rôle des différents intervenants interagissant *via* ce modèle. Le modèle est ainsi le champ de l'intersubjectivité de ces intervenants : il est nécessaire d'identifier un ensemble de concepts permettant d'établir un lien entre leurs activités. Nous avons cherché à faciliter au maximum la coopération des experts du domaine avec l'informaticien et nous avons proposé la notion d'intention, et conjointement de tendance, pour l'établissement de ce lien sémantique.
Ceci nous a amenés à construire un modèle qui sépare le plus clairement possible les interventions des différents experts en explicitant au maximum le sens de leur contribution vis-à-vis de l'agent. Cette démarche constitue selon nous une condition nécessaire à l'établissement d'une architecture d'agent qui permette un prototypage interactif et collaboratif avec un haut niveau d'abstraction, qui facilite l'interprétation des résultats de la simulation et qui s'inscrive au mieux dans le processus global de spécification.

Si ce modèle n'est pour l'instant qu'une proposition et doit être éprouvé et éventuelle-

[5]Cette proposition a fait l'objet de plusieurs publications scientifiques [Favier and De Loor, 2003, Favier, 2003, Favier and De Loor, 2005].

ment affiné, l'effort qui a été effectué pour aller dans le sens de cette intégration des contributions des différents intervenants ouvre des perspectives intéressantes. La principale est sans doute la possibilité de définir une démarche de conception complète qui, de l'analyse à l'implémentation, guide les experts dans la définition des différentes composantes : identification des actions, des perceptions, des propriétés, définition des règles comportementales,... Avant de travailler dans ce sens, il nous faut concevoir un modèle d'exécution capable de supporter la simulation d'agents spécifiés selon ce type de modèle comportemental.

Les principaux choix qui fondent cette proposition soulèvent des difficultés techniques à sa réalisation, et nous allons maintenant étudier les moyens de mise en œuvre d'une telle architecture. C'est l'objet du chapitre suivant qui détaille le modèle d'exécution développé.

Chapitre 5

Le modèle d'exécution

> There are only 10 types of people in the world : those who understand binary and those who don't.
>
> ANONYME

Ce chapitre présente les développements que nous avons effectués dans le but de créer une application de simulation exploitant le modèle d'agent intentionnel tel qu'il a été décrit dans le chapitre précédent. Il détaille les principales difficultés techniques attenantes à l'exécution d'un code mixte (impératif / déclaratif) et la manière dont nous avons implémenté les fonctionnalités requises par le modèle. L'évolution des différentes solutions auxquelles nous avons eu recours est présentée, le modèle d'exécution ayant donné lieu à deux implémentations préliminaires avant sa forme actuelle.

5.1 Programmation multi-paradigmes

Les principales difficultés posées pour l'implémentation du modèle d'agent intentionnel sont :

le besoin de dynamisme : pour répondre aux exigences du prototypage interactif, il faut pouvoir modifier en ligne le comportement de n'importe quel agent. C'est à dire avoir accès non seulement à la base de faits, pour les modifications de comportement par le biais des règles comportementales, mais aussi au code des méthodes implémentant les actions perceptives et d'interaction pour une éventuelle phase de mise au point. Ceci doit pouvoir se faire pendant l'exécution de la simulation et avec une granularité instance (modifier le comportement d'**un** poisson dans un banc de poissons).

l'approche multi-paradigmes : l'architecture du modèle exige de pouvoir inférer sur une base de faits et exécuter des méthodes impératives au sein du même agent. De plus, ces différentes activités peuvent être asynchrones : l'agent peut inférer alors qu'il est en train d'exécuter une ou plusieurs actions. L'outil doit donc être capable d'activer différentes activités en parallèle (entre plusieurs agents et même au sein de l'agent), et des activités de natures hétérogènes (inférence et interaction). De plus, une communication entre ces deux aspects doit être offerte (déclenchement d'une méthode à l'issu d'une réflexion, consultation des valeurs de propriétés par la partie décisionnelle), le plus simplement possible et en dégradant le moins possible chacun des paradigmes (infraction du paradigme relationnel logique par l'ajout de mécanismes extra-logiques par exemple).

le besoin de réactivité : l'agent peut mener plusieurs réflexions en parallèle, et ces réflexions peuvent porter sur des propriétés qui sont elles-mêmes en cours de modification, puisque l'agent peut "réfléchir" pendant qu'il agit. De même, une activité d'inférence peut reposer sur la véracité d'un fait lui-même issu d'une telle activité. Il faut donc pouvoir activer toutes ces activités en parallèle, mais surtout que les éventuels effets de bords d'une activité soient immédiatement pris en compte dans les autres.

L'ensemble de l'étude présentée ici est menée au sein du projet `ARéVi`[1] du Laboratoire d'Ingénierie Informatique (*LI2*) de l'Ecole Nationale d'Ingénieurs de Brest. Ce projet a pour vocation de concevoir des outils adaptés à la simulation de systèmes multi-agents en réalité virtuelle. Notre but était de produire un outil qui soit utilisable par l'ensemble des membres du projet et qui repose sur les développements de cette équipe. Lorsque nous avons débuté ce travail d'implémentation, le laboratoire utilisait le langage `oRis` [Harrouet et al., 2002]. Il s'agit d'un langage interprété orienté agent qui est le fruit du travail de cette équipe. Il offre un mécanisme d'ordonnancement d'activités en parallèle, une granularité instance et des facilités de prototypage en ligne. Cependant, c'est un langage impératif, il fallait donc déterminer la voix d'intégration du paradigme déclaratif afin de satisfaire les besoins du modèle d'agent intentionnel.

Dans un souci de concision, les approches possibles pour la conception d'un outil multi-paradigmes peuvent être classifiées comme suit :

[1] Atelier de Réalité Virtuelle.

communication inter-langages : il est possible de créer un *pont* entre deux langages (fichier, `tcp/ip`,`middleware`,...) qui reposent chacun sur un paradigme différent,

intégration : un outil existant est choisi comme base, des extensions lui sont ajoutées afin d'intégrer le nouveau paradigme,

réification : un outil est créé de toute pièce, il est pensé, depuis la définition de sa syntaxe jusqu'à celle de son modèle d'exécution, pour intégrer les paradigmes ciblés.

Les deux premières approches présentent un défaut commun : elles nécessitent de privilégier un des paradigmes qui est en charge du contrôle du flot d'exécution. La première est particulièrement peu efficace : le mécanisme de liaison est coûteux en temps d'exécution ; de plus il y a par définition un écart, voire une opposition, sémantique entre les langages impératifs et déclaratifs qui rend la mise en place de ce *pont* délicate. Parmi ces difficultés, les plus grandes sont sans doute le typage des données et la contextualisation. En effet, dans notre cas par exemple, il faut mettre en place une communication entre la simulation exécutée sous `oRis` et un serveur de requêtes écrit en `Prolog`. Cette communication ne peut pas tenir compte de l'ordonnancement des activités au sein d'`oRis` et doit prendre en charge la contextualisation des requêtes pour conserver la granularité instance. Ceci implique d'être capable de gérer un espace de clauses différent pour chaque agent, soit par un mécanisme de modules dans `Prolog`, soit par l'utilisation de plusieurs instances du moteur `Prolog`. Ces problèmes seront d'autant plus pénalisants que le nombre d'agents manipulés sera important, cette solution possède donc une très mauvaise capacité de "passage à l'échelle".

Malgré tous ces défauts, c'est la première solution que nous avons retenue dans un premier temps : son principal mérite est d'être facilement mise en œuvre.
Afin d'effectuer simplement une première série de tests pour évaluer la viabilité du modèle, nous avons développé une communication par connexion réseau (`tcp/ip`) entre `oRis` et `Prolog`. Le dispositif obtenu créait donc un processus `Prolog` par agent instancié. Cette architecture était évidemment très coûteuse en temps d'exécution et ne se prétait qu'à des essais simples. Nous ne nous étendrons pas plus sur le détail de son fonctionnement qui est désormais purement anecdotique. Cette première réalisation nous a tout de même permis de mettre à l'épreuve le mécanisme de sélection d'action et de valider sa faisabilité. Le code à produire pour spécifier un comportement était particulièrement abscons, la liaison entre les deux langages étant effectuée de manière applicative.
L'architecture de cette première implémentation est illustrée par la figure 5.1 page suivante.

Nous nous sommes donc logiquement tournés dans un second temps vers une solution d'intégration. Pour cette deuxième étape, nous nous sommes à nouveau appuyés sur le langage `oRis`. Comme c'est un langage orienté objet interprété avec une granularité instance, il offre toutes les fonctionnalités requises d'introspection et de modification dynamique de code. Il restait donc à lui ajouter les capacités de raisonnement logique. Pour cela, nous avons choisi d'intégrer `Prolog` en totalité : le développement d'un moteur d'inférence additionnel au sein d'`oRis` aurait nécessité un développement important sans ajouter beaucoup plus de fonctionnalités. La barrière sémantique entre les deux paradigmes demeurant, une intégration bien menée de `Prolog` nous a parue préférable à la conception d'un moteur d'inférence qui serait resté élémentaire : les versions de `Prolog` existantes comprennent de nombreux

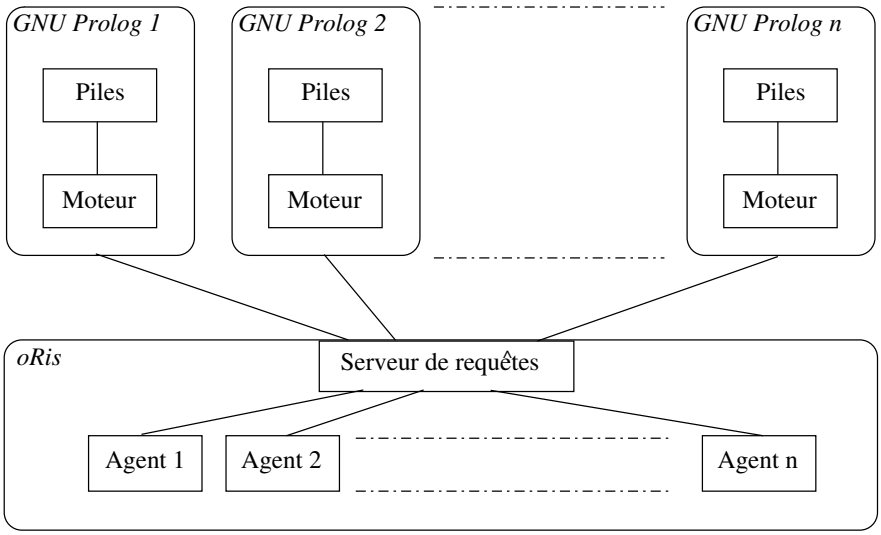

FIG. 5.1 – Première implémentation : une passerelle tcp/ip entre oRis et GNU Prolog.

prédicats prédéfinis qui facilitent grandement le développement. Il était donc intéressant de tirer parti de ces bibliothèques au prix de la conception d'un lien entre les deux langages plus évolué qu'une simple passerelle avec un serveur de requêtes.
Pour ce développement, notre choix s'est porté sur `GNU Prolog`. C'est une version de `Prolog` sous licence GPL, ce qui nous permettait d'accèder à la totalité du code source. De plus `GNU Prolog` permet de génerer du code objet à partir de code `Prolog` (compilation), il est très modulaire et possède une `API`[2] en langage `C`. La modularité de `GNU Prolog` nous intéressait particulièrement : il est possible, lors de la compilation de l'interpréteur, d'inhiber la prise en charge de certaines parties du moteur. Ceci correspondait à l'un de nos besoins : le langage `oRis` étant le support de nos développement, il était inutile de doter en plus le moteur d'inférence de certaines capacités (interpréteur de ligne de commande, gestion de l'affichage, du réseau...) et la modularité de `GNU Prolog` nous a permis d'utiliser un moteur d'inférence minimaliste, donc plus "*léger*" (espace mémoire moindre et plus grande réactivité).
Pour sa part, `oRis` a été écrit en `C++` et possède une `API` avec ce langage. Pour construire un lien qui offre un interfaçage fort entre les deux langages, il était donc nécessaire de développer une surcouche `C++` à l'`API C` de `Gnu Prolog`, puis de créer une classe native[3] `oRis` qui embarque cette surcouche.

[2] *Application Programming Interface*, c'est une interface qui permet d'exploiter une application depuis un programme extérieur. En l'occurrence, il est possible de créer et manipuler des termes et des requêtes `Prolog` depuis un programme écrit en langage `C`.

[3] Le langage `oRis` est constitué d'un ensemble de classes `C++` qui réifient les notions de *classe* et d'*instance* afin de les "*instrumenter*" : cet enrichissement apporte les fonctionnalités nécessaires aux mécanismes d'interprétation, d'introspection du code,... L'`API oRis` permet de définir de nouvelles classes natives. Elles sont reconnues comme des classes du système et permettent de s'interfacer avec le moteur `oRis` à très bas niveau. En contrepartie, elles doivent implémenter une interface (au sens du `C++`) très stricte. Nous ne rentrerons pas plus en détail sur les possibilités et les limitations des classes natives. Pour plus d'explications sur `oRis` voir [Harrouet, 2000].

La première étape fut donc le développement de PrologWrapper[4] : il s'agit d'une librairie dynamique écrite en C++ qui encapsule l'API C de GNU Prolog. En fait, PrologWrapper fait un peu plus que cela : notre but était de n'utiliser qu'une seule instance du moteur Prolog pour ne pas retrouver les mêmes limitations qu'une simple passerelle entre les deux langages. Or, la destination de ce développement, à savoir l'utilisation pour un système multi-agents, imposait de pouvoir lancer des requêtes sur des bases de faits distinctes[5] (au moins une par agent). PrologWrapper introduit donc la notion de contexte : il exploite le moteur de GNU Prolog mais redirige à la volée les espaces de pile[6] de celui-ci en fonction du contexte appelant. Il est possible d'instancier un nombre quelconque de contextes et toutes les transactions (création de terme, lancement d'une requête, récupération d'un résultat...) sont attachées à un contexte.
Pour cette réalisation, nous nous sommes imposés de ne pas modifier le code de GNU Prolog afin d'assurer la maintenabilité de l'ensemble et la possibilité d'évoluer vers les versions ultérieures. Il nous a donc fallu nous livrer à une analyse détaillée du code pour identifier l'ensemble des valeurs à sauvegarder ou à rediriger, et pour déterminer les étapes de la résolution qui supportent une telle modification dynamique de l'état de la machine virtuelle. Le code de GNU Prolog est complexe et extrêmement optimisé. Nous ne ferons pas ici l'exposé des détails techniques de ce développement, une telle présentation sortirait très largement du cadre de cette étude. A l'issu de ce travail, PrologWrapper permet d'instancier un nombre arbitraire de contextes (dans la limite de la mémoire disponible tout de même !) et de "*s'adresser*" à chaque contexte comme s'il s'agissait d'un moteur Prolog à part entière. Lorsque cela est nécessaire, la librairie permute les espaces de résolution (piles et registres de la machine virtuelle) de manière totalement transparente pour l'utilisateur. Cette permutation se fait par la modification de quelques pointeurs vers les principales structures de données, elle introduit donc un surcoût extrêmement faible en termes de vitesse d'exécution. Ces valeurs étant des variables globales dans l'implémentation de GNU Prolog, aucune modification du code de ce dernier n'a été nécessaire : tout le traitement de contextualisation est effectué par PrologWrapper.

---

[4] Le terme *wrapper* signifie *emballage*, *enveloppe*. Il est souvent utilisé en informatique pour nommer une interface qui "*embarque*" une librairie ou un langage afin de traduire ses entrées et sorties en un autre langage ou de les adapter à un protocole.
PrologWrapper est disponible sur http://www.enib.fr/~favier/prologWrapper.

[5] Une alternative possible était l'utilisation d'une version de Prolog offrant un mécanisme de module (chaque module est considéré comme une base de fait séparée des autres). SWI-Prolog offre un tel mécanisme, mais la version de cette époque là se prétait peu de par son implémentation à une intégration : le moteur et le contexte de résolution étaient fortement liés, ce qui laissait peu de possibilités de rendre l'inférence interruptible. Le recours à ce système aurait alors imposé de sérialiser les traitements, ce qui va à l'encontre d'une utilisation pour un système multi-agents.

[6] Le moteur d'inférence de Prolog est une machine virtuelle à pile : il exploite différentes structures de pile (4 en tout) pour représenter les termes, stocker les points de choix au fur et à mesure de la résolution... A un instant donné, l'état de la machine virtuelle est entièrement défini par le contenu de ses registres et l'état de ses piles. La redirection des espaces de pile, conjointement à la sauvegarde et la restauration des registres, permet donc de permuter entre plusieurs espaces de résolution.

Un ensemble de classes natives nommé `LogoRis`[7] a été développé sur la base de `Prolog Wrapper`. Cette extension permet de manipuler les contextes `Prolog` au sein d'oRis tels qu'ils ont été définis dans `PrologWrapper`. Le moteur `Prolog` n'est instancié qu'une seule fois et est lié dynamiquement à `oRis`. Ces classes natives correspondent aux différents objets manipulables au travers de cette interface (`PrologAtom`, `PrologTerm`, `PrologQuery` et `PrologContext`). Cette intégration de `GNU Prolog` dans `oRis` nous a permis de réaliser une seconde implémentation du modèle d'agent intentionnel. Dans cette architecture, chaque agent possède un contexte `Prolog` qui assure la prise de décision et qui permet de stocker les descriptions d'actions et les règles comportementales ainsi que les connaissances de l'agent.
L'architecture de cette seconde implémentation est illustrée par la figure 5.2.

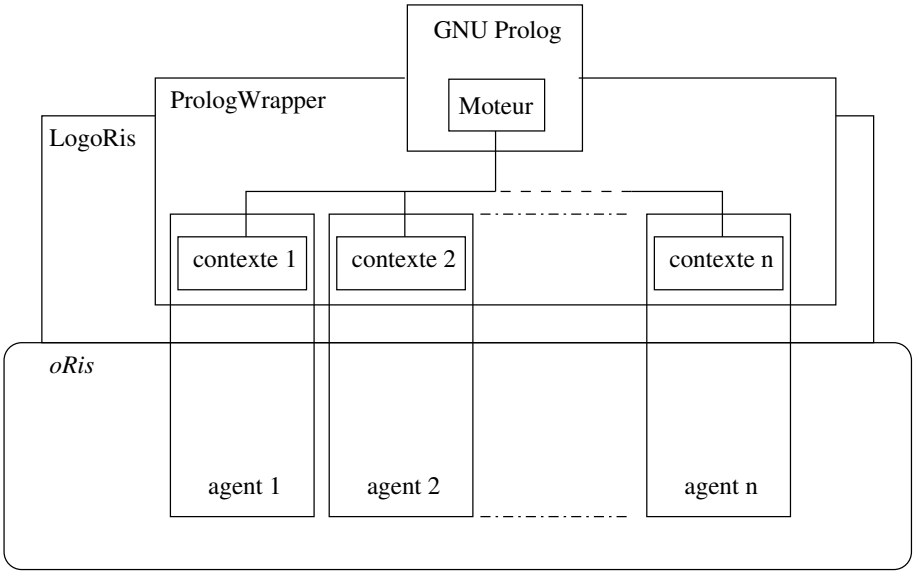

FIG. 5.2 – Seconde implémentation : une intégration de GNU Prolog au moyen de classes natives oRis.

Si la première implémentation était très contraignante, les résultats obtenus *via* la liaison `tcp/ip` devant être interprétés de manière applicative, cette seconde implémentation offrait une plus grande expressivité : les éléments manipulés par `Prolog` étant réifiés dans `oRis` l'interface utilisateur était beaucoup plus agréable.
`LogoRis` n'a d'ailleurs pas été utilisé uniquement pour les travaux sur le modèle d'agent intentionnel. Cet outil a été employé dans le développement d'un modèle d'utilisateur dans le cadre d'une étude d'interfaces multi-modales de systèmes interactifs[8] : plusieurs contextes `Prolog` sont utilisés au sein du même agent ; il servent à réaliser les deux espaces de représentation de l'information impliqués dans le raisonnement de l'utilisateur d'un PDA (mémoire à long terme d'une part et mémoire de travail d'autre part). Il est ainsi possible de

[7](http://www.enib.fr/~favier/logoRis)
[8]([Le Bodic and Favier, 2004])

spécifier pour chaque agent un traitement de l'information dans différents contextes de raisonnement au moyen d'un langage plus accessible aux spécialistes impliqués (en l'occurence des psychologues et des ergonomes).

Malgrè les fonctionnalités intéressantes de cette implémentation, surtout dûes à la grande souplesse d'utilisation du langage `oRis`, nous avons été confrontés à des limitations gênantes pour le modèle d'agent intentionnel. La première était la difficulté de partager les connaissances entre plusieurs espaces de résolution. Les connaissances éventuellement communes à plusieurs agent devaient être explicitement déclarées dans les différents contextes. Ceci représentait un obstacle à l'implémentation de mécanismes d'héritage de connaissances. Les agents intentionnels étant en partie considérés comme des objets, il est souhaitable de retrouver au niveau des faits et règles les même possibilités d'héritage qu'au niveau des méthodes et attributs. L'autre limitation, beaucoup plus gênante, était dûe au fait que le changement de contexte au sein de `Prolog` ne peut se faire qu'à la fin de l'exécution d'une requête ou sur un point de choix (lors d'un retour arrière). Cela signifie qu'il n'est pas possible d'interrompre le moteur `Prolog` en cours de résolution. Il faut attendre au minimum une première solution à la requête traitée. Pour le modèle proposé, une telle architecture n'était utilisable que pour un nombre peu élevé d'agents dotés de comportements simples. En effet, la résolution d'une requête étant un appel blocant, la multiplication du nombre d'agents, donc du nombre d'inférences nécessaires, ne permettait plus de satisfaire les contraintes de rendu en temps réel : il suffisait qu'un agent exploite une règle comportementale complexe pour que la simulation se fige quelques instants.

Nous avons donc abordé la question de l'interruptibilité du moteur d'inférence : notre but était de pouvoir initier une requête `Prolog` avec un nombre de "*pas*". Chaque pas correspondant alors à une instruction de la machine virtuelle `Prolog`, le moteur aurait alors exécuté au plus ce nombre d'instructions avant de sortir de l'appel, la découverte d'une solution ou l'examen d'un point de choix terminant l'étape en cours prématurément. Un tel fonctionnement nous aurait permis une intégration totale à l'ordonnanceur `oRis` : ce langage est interprété et chaque entité, à tour de rôle, voit exécuter n micro-instructions de son code. Ce nombre de pas n peut être réglé de manière applicative, tout comme la politique d'ordonnancement qui est ajustable. Nous envisagions donc, à ce moment là, de mêler le code impératif et le code déclaratif directement au sein de l'ordonnanceur d'`oRis` en faisant cohabiter les deux machines virtuelles. Ceci aurait permis de maîtriser totalement des activités hétérogènes, ce qui constitue le principal obstacle technique à la réalisation de notre proposition.

L'exploration de cette piste nous a conduit à une étude très détaillée de la machine virtuelle de WARREN qui assure le fonctionnement interne de `Prolog`[9]. Malheureusement, le choix initial de `GNU Prolog` nous a confrontés à une difficulté supplémentaire : cette version du langage repose sur une implémentation très optimisée. Parmi les différents mécanismes que DANIEL DIAZ a utilisé dans ce développement se trouve une optimisation au niveau du *run-time* du langage `C`, `GNU Prolog` étant développé avec ce langage : `Prolog`

[9]([Warren, 1983, Aït-Kaci, 1991])

utilisant essentiellement des mécanismes de gestion de pile, l'auteur a décidé d'employer la pile d'appel du `C` comme pile principale. Ceci permet d'obtenir une implémentation très efficace, puisqu'elle exploite au maximum les optimisations du processeur pour la gestion de la pile. C'est l'avers de la médaille, le revers étant pour nous l'impossibilité de sauvegarder l'état de la machine de Warren à un instant quelconque facilement. Une telle contextualisation nécessiterait une sauvegarde de la pile d'appel, ceci peut se faire par des instructions très particulières (`longjmp` par exemple) ou en utilisant un mécanisme de *thread* (chaque *thread* possède sa propre pile d'appel). La première solution s'est révélée trop délicate à mettre en place : un changement de contexte au moyen de l'instruction `longjmp` est déjà utilisé dans `GNU PRolog` pour le dispositif de "*debug*" du langage. Il aurait fallu modifier le code en profondeur pour une telle mise en place, l'effort correspondant aurait sans doute été du même ordre que la création d'un nouveau moteur d'inférence, et la modification du code allait à l'encontre de notre volonté première. La seconde solution aurait conduit à repenser totalement l'architecture et le procédé d'ordonnancement.

Ces deux premières implémentations nous ont donc permis d'évaluer en partie les principes du modèle proposé, tout en mettant en évidence les difficultés techniques attenantes à sa réalisation. Le problème d'interruptibilité du moteur d'inférence conduit logiquement à envisager une solution par réification : la conception d'un nouveau langage basé sur la mixité du code impératif et déclaratif constituerait une réponse adaptée. De tels langages existent, mais à notre connaissance aucun n'est orienté vers la simulation de systèmes multi-agents en environnement virtuel : l'adaptation à un tel usage reste strictement applicative. Cet axe de recherche implique une importante étude préliminaire que nous n'avons pas menée, il a donné lieu à la proposition d'un nouveau sujet de thèse au sein du laboratoire. Pour notre proposition, l'issue technique a pris une autre voie : le constat de l'impossibilité d'adapter `LogoRis` à nos besoins a été fait aux environs du mois de septembre 2003. Or, à cette période les membres du projet ne travaillaient plus au développement du langage `oRis` (qui demeure utilisé) mais à la création d'une bibliothèque dédiée aux simulations de systèmes multi-agents. Cette bibliothèque écrite en `C++` porte le nom du projet : `ARéVi`. Dans le but de participer à cette dynamique de développement, il était préférable pour nous de nous orienter vers ce nouveau projet. De plus, à la même période l'équipe de développement de `SWI-Prolog` délivrait une version multi-thread stable de son interpréteur [Wielemaker, 2003]. Cette conjoncture nous a conduit à étudier l'opportunité d'intégrer `SWI-Prolog` et `ARéVi` en une unique bibliothèque. Cette troisième implémentation, avec laquelle nous travaillons actuellement, est présentée dans la section suivante.

5.2 Ordonnancement d'activités hétérogènes

5.2.1 ARéVi

La bibliothèque `ARéVi`, écrite en `C++`, est conçue pour la réalisation de simulations de systèmes multi-agents en environnement virtuel.
Elle réifie les notions de classe et d'objet (renommés respectivement `ArClass` et `ArObject`) afin de les "*instrumenter*". Les apports sont multiples, parmi eux nous retien-

drons essentiellement :

- l'utilisation de pseudo-pointeurs et la mise en place d'un "*ramasse-miettes*" (*garbage collector*) qui implicite toutes les opérations de désallocation de la mémoire ;
- l'ajout de capacités d'introspection limitée (un objet contient une référence à sa classe, une classe détient des références sur toutes ses instances...) ;
- des facilités de rendu (visualisation *via* `OpenGL`, son tridimentionnel *via* `OpenAL`...) ;
- un dispositif d'interfaçage avec des modules externes (*plugins*) permettant de recourir à des bibliothèques existantes (traitement d'images *via* `ImageMagick`, traitement de flux XML *via* `LibXML2`...) ;
- un ordonnanceur d'activités.

`ARéVi` offre donc de nombreuses fonctionnalités indispensables à la réalisation de telles simulations. Par contre, cette bibliothèque ne fournit pas de modèle comportemental : sa vocation est de servir de plateforme à de tels développements. Un des points clefs de la liste précédente est la présence d'un ordonnanceur : `ARéVi` est capable d'orchestrer l'exécution de différentes activités ayant chacune leur fréquence d'activation. Pour `ARéVi`, une activité est une méthode non bloquante d'un `ArObject`. Deux politiques d'ordonnancement sont fournies par défaut :

- l'ordonnancement en temps virtuel exécute toutes les activités en attente avant de "*sauter*" à l'instant suivant. Le temps simulé n'est alors pas en rapport avec le temps physique, l'intérêt de ce mode est surtout l'application à la simulation numérique en l'absence d'interaction avec l'utilisateur ;
- l'ordonnancement temps réel exécute les différentes activités conformément à la fréquence de celles-ci, le temps correspond alors au temps physique. Ce mode de fonctionnement convient aux simulations impliquant un utilisateur (interactions ou nécessité d'un rendu "temps réel").

Dans notre cadre applicatif, `ARéVi` est exploitée avec l'ordonnanceur temps réel. Il est entendu que l'ordonnanceur ne peut administrer plus de traitements que le système ne peut exécuter : lorsque la charge de calcul est trop importante, les activités peuvent être déclenchées avec retard ou, dans les cas extrêmes, ne pas être déclenchées lors d'une itération de la boucle de simulation. Une activité reçoit en paramètre l'écart temporel depuis sa dernière invocation, le traitement peut donc être adapté au sein de chaque activité pour compenser une éventuelle dérive temporelle. Pour de plus amples détails sur le fonctionnement de la bibliothèque `ARéVi` voir la page du principal développeur[10].

La volonté d'implémenter le modèle d'agent intentionnel avec `ARéVi` se heurte donc principalement à deux difficultés :

- l'intégration de `Prolog` est à repenser avec les contraintes d'ordonnancement ;
- contrairement à `oRis`, `ARéVi` n'offre pas de possibilité d'inspection ou de modification dynamique du code (le code `C++` est compilé).

Nous allons par la suite exposer la manière dont nous avons résolu ces deux problèmes.

[10] http://www.enib.fr/~harrouet

5.2.2 SWI-Prolog : *thread* et modules

Les versions courantes de SWI-Prolog comprennent une gestion stable des *thread*[11]. De plus, cet interpréteur offre la possibilité de définir des modules. Ces deux fonctionnalités nous ont permis de concevoir une architecture répondant aux besoins du modèle intentionnel.

Les *thread*

Les *thread* utilisés dans SWI-Prolog sont des *thread* C natifs (leur ordonnancement est donc à la charge du système d'exploitation au même titre que les autres *thread* ou processus). Ils ne comprennent pas le moteur Prolog : chaque *thread* possède un "*tas*" (*heap*). Il s'agit de la pile qui permet de stocker et manipuler les termes utilisés par la résolution en cours. La pile globale qui contient les faits et règles est par contre partagée entre les *thread*. L'architecture utilisée permet d'instancier un nombre arbitraire de moteurs d'inférence, chaque *thread* pouvant être à loisir attaché à ou détaché d'un moteur. Cette architecture a été conçue afin de faciliter la mise en place de systèmes *client/serveur* : chaque client exécute sa requête au sein d'un *thread* qui est attaché au premier moteur disponible. Il est ainsi possible de gérer la charge du serveur en contrôlant le nombre de moteurs créés. Le système comporte alors autant de *thread* qu'il a de client en demande, les *thread* en attente d'un moteur libre ne consommant qu'une très faible ressource.

L'implémentation de la gestion des *thread* dans SWI-Prolog offre des fonctionnalités qui nous sont très utiles :

- les *thread* sont complétement indépendants et peuvent être créés ou détruits à volonté ;
- un système de "*boîte à message*" facilite la communication entre *thread* ;
- un *thread* peut émettre et recevoir des signaux POSIX, ce qui permet de déclencher des traitements réflexes même si une résolution est en cours ;
- il est possible de définir des prédicats locaux à un *thread* (le prédicat n'est alors visible que depuis ce *thread*, il est dynamique et volatile) ;
- l'API C de SWI-Prolog permet de manipuler les *thread* Prolog.

L'utilisation des *thread* est donc une première réponse aux problèmes d'ordonnancement. En instanciant un unique moteur d'inférence lié à l'application, il est possible de créer autant de *thread* qu'il faut gérer d'agents intentionnels : ils correspondent tous à un espace de résolution différent. L'ordonnanceur ARéVi séquentialise les appels aux différentes activités. Donc, en créant une activité de *décision* pour chaque agent intentionnel qui soit en interface avec le *thread* dédié à l'agent, les résolutions Prolog sont *a priori* appelées à tour de rôle. Mais, si une de ces résolutions s'avère longue, le système suspend le *thread* correspondant

[11] Bien que les premières modifications en vue d'une implémentation multi-thread de SWI-Prolog datent de la version 2.9.1, le processus fut long et la version 5.1.10 (14 mars 2003) fut la première à séparer le moteur d'inférence des *thread*. La possibilité d'utiliser des prédicats locaux aux *thread* (ce qui est nécessaire à notre architecture) est venue avec la version 5.2.8 (13 septembre 2003). C'est à partir de cette version que nous avons pu utiliser SWI-Prolog. Nos développements reposent finalement sur la version 5.2.13 (29 janvier 2004) qui a résolu des problèmes de fuites mémoire lors de la destruction d'atomes dans les applications multi-thread.

au profit des autres (autres *thread* `Prolog` ou processus `ARéVi`). Il faut pour que cette organisation ne soit pas susceptible de figer la simulation que les appels à `Prolog` effectués dans les activités `ARéVi` ne soient pas des appels bloquants.
Un nombre élevé d'agents n'est plus un obstacle, par contre le recours à des règles comportementales complexes en grand nombre peut augmenter la charge du système au point que le temps processeur alloué à `ARéVi` ne soit plus suffisant pour respecter un ordonnancement et un rendu en temps réel. Cette limitation est incontournable, le matériel n'offre pas une ressource de traitement infinie !
Mais, si l'activité de décision qui initie la résolution n'effectue pas un appel bloquant, il n'est pas facile d'en traiter l'issue : il faut exploiter l'éventuel résultat de la requête de manière asynchrone. Ceci correspond justement au modèle proposé : les règles comportementales sont évaluées cycliquement à l'initiative de l'activité de *décision* et la dernière décision en date est exécutée par l'activité d'*action* qui a sa fréquence propre. Ainsi, il est possible d'exécuter plusieurs fois une action à l'issue d'un unique raisonnement, ou au contraire de prendre plusieurs décisions qui ne donnent pas lieu à une action. L'asynchronisme entre la décision et l'action est selon nous vecteur de réalisme. Concrètement, cela signifie pour l'agent intentionnel qu'au moment où son activité d'*action* consulte la dernière décision prise, il se peut que l'activité de décision soit en train de sélectionner une action totalement contraire, le contexte ayant changé pour l'agent. Cela n'est pas un comportement que nous souhaitons éviter : il arrive fréquemment que nous réalisions être en train de commettre une erreur alors même que nous sommes en pleine action, ce type d'"*incohérence*" génère donc des comportements qui restent crédibles. Dans les cas particuliers où cette situation est à proscrire, il suffit de fixer judicieusement les fréquences et les priorités des activités de *décision* et d'*action*.

Cette première analyse préfigure l'architecture définitive et répond à nos exigence en matière d'ordonnancement d'activités hétérogènes. Il reste à surmonter les problèmes de partage de connaissances, afin de mettre en place un mécanisme d'héritage, et de prototypage interactif.

Dans un premier temps, nous aborderons le partage de connaissances. Le fait que chaque *thread* corresponde à un espace de résolution nous permet d'effectuer des résolutions séparées pour chaque agent, mais sur une base de faits et de règles commune. La seule spécialisation offerte passe par la directive `thread_local` qui permet de spécifier que les clauses d'un prédicat ou des faits particuliers sont dynamiques et non partagés entre les *thread*. Cette fonctionnalité va nous être d'un grand secours pour certaines composantes du modèle. Par exemple, chaque agent intentionnel nécessite la définition de règles comportementales. Ces règles auront toutes la même forme et seront manipulées de la même manière. Par contre, les règles elles-mêmes restent propres à chaque agent.
En déclarant que les règles comportementales sont locales aux *thread*, il est possible de déclarer les prédicats de traitement de ces règles dans la base commune : chaque agent exploite alors le même mécanisme de traitement des règles comportementales, mais au cours de cette résolution il ne peut utiliser que les règles définies au sein de son thread.
Ce mécanisme recouvre nos besoins pour l'ensemble du processus de sélection d'action : les

prédicats utilitaires qui assurent l'évaluation des règles comportementales et des descriptions d'actions sont définis dans la base commune. Par contre, les règles comportementales et les descriptions d'actions sont déclarées locales aux *thread*, ce qui assure la séparation des spécifications de chaque agent.

Cependant, le recours à cette directive ne résoud pas tous nos problèmes. En effet, dans certains cas une séparation aussi forte n'est pas souhaitable. Par exemple, il serait plus confortable qu'un agent possède par défaut les règles comportementales définies pour sa classe et qu'il puisse les surcharger (dans le sens de la surcharge de méthode en programmation orientée objet) ou en définir d'autres. Or, la directive `thread_local` ne fait que déclarer les propriétés du foncteur sur laquelle elle porte : il faut déclarer le fait concerné dans tous les *thread* où il sera nécessaire. De plus, et c'est beaucoup plus gênant, la directive porte sur une règle ou un type de fait particulier. Il faut donc prévoir le nom et l'arité de tous les éléments susceptibles d'être déclarés au sein du *thread* au cours de l'exécution. Si cela est possible pour des éléments faisant partie du modèle, comme les règles comportementales ou les descriptions d'actions, c'est contradictoire avec notre volonté de laisser la possibilité de définir des connaissances "libres" au sein de chaque agent.
Heureusement, l'usage des modules, conjointement à celui des *thread* et à la possibilité de définir des prédicats locaux, nous permet d'effacer toutes ces difficultés.

Les modules

La notion de module a été introduite en `Prolog` pour permettre la définition d'espaces de nommage. L'ensemble des faits et prédicats qui sont déclarés au sein d'un module ne sont visibles que depuis ce module. Seuls les prédicats qui sont explicitement exportés seront accessibles depuis un autre module. Ceci introduit un mécanisme de résolution de portée équivalent à l'utilisation des sections *public* et *private* en programmation orientée objet, ou à l'export de symboles depuis une bibliothèque par exemple.
Ainsi, un module peut en importer un autre, cela signifie simplement qu'il déclare recourir aux prédicats publiques de celui-ci. En contrepartie, un module doit déclarer explicitement les prédicats qu'il exporte. L'avantage principal est la possibilité d'utiliser des prédicats utilitaires sans se soucier de l'unicité des noms : tous ces "*sous-prédicats*" peuvent utiliser un foncteur déjà déclarer dans un autre module, ceci ne donne lieu à aucun conflit de nommage.
Au niveau global, un foncteur exporté est caractérisé par la forme :

```
Module:Nom/Arité
```

Ainsi, il est possible d'invoquer explicitement un prédicat publique d'un module tierce même si il n'a pas été importé.

Dans l'exemple figure 5.3 page suivante, le fait `unFait` est déclaré dans le module `mondeA`. La vérification de son existence échoue dans le module `mondeB` sauf s'il est appelé explicitement avec le préfixe correspondant. Au passage, il est à noter que la gestion des modules est dynamique : il suffit de s'adresser à un module pour qu'il soit créé s'il n'existe pas déjà.

De plus, la liste des modules importés est manipulable. Par défaut, tous les modules importent un module commun : `user` (c'est le module qui est utilisé par toutes les requêtes qui ne s'adressent pas explicitement à un module particulier). Le prédicat `import_module` permet de consulter cette liste alors que les prédicats `add_import_module` et `delete_import_module` permettent de la modifier. Dans notre exemple d'illustration, le module `mondeA` est ajouté dynamiquement à la liste d'import de `mondeB`, ce qui permet de vérifier `unFait` dans `mondeB`. Cet ajout peut se faire en début ou en fin de liste : les clauses d'un prédicat qui serait éventuellement défini dans plusieurs modules importés se surchargent entre elles.

```
 ?- mondeA:assert(unFait).
Yes
?- mondeB:unFait.
No
?- mondeA:unFait.
Yes
?- module(mondeB).
Yes
mondeB:  ?- unFait.
No
mondeB:  ?- mondeA:unFait.
Yes
mondeB:  ?- import_module(mondeB, Module_import).
Module_import = user ;
No
mondeB:  ?- add_import_module(mondeB, mondeA, start).
Yes
mondeB:  ?- import_module(mondeB, Module_import).
Module_import = mondeA ;
Module_import = user ;
No
mondeB:  ?- unFait.
Yes
mondeB:  ?-
```

FIG. 5.3 – Un exemple simple de manipulation de modules dans SWI-Prolog.

Ces fonctionnalités permettent de mettre en place le niveau de séparation des connaissances requis par le modèle : en utilisant un module par classe et par agent, il suffit de déclarer qu'un agent importe le module de sa classe et qu'une classe importe le module de ses classes parentes pour obtenir un mécanisme d'héritage. Par contre, il se pose alors un problème : il faut faire la différence entre surdéfinition et surcharge.
Comme illustré par la figure 5.4 page suivante, le comportement par défaut est une surdéfinition : la déclaration d'un fait `connaissance/1` dans le module de la classe fille masque les faits du même type déclarés dans la classe parente. Cette occlusion demeure tant que le foncteur n'est pas aboli : un simple retrait ne suffit pas.

Le cas où une surcharge est souhaitée doit être traité de manière applicative : si la déclaration d'une nouvelle clause doit compléter et non masquer les clauses du même type présentes dans la hiérarchie d'import, il faut prévoir soit de recopier dans le module courant les clauses existantes, soit de rechercher dans l'arbre d'héritage les différentes clauses lors de leur exploitation.

L'utilisation de modules pose aussi problème pour l'écriture de méta-prédicats. En effet, lorsqu'un prédicat est invoqué, il est par défaut exécuté dans l'espace de nommage où il est défini. Par exemple, si un prédicat est défini dans le module `user`, il est visible depuis tous

```
?- classeParente:assert(connaissance(classique)).
Yes
?- module(classeFille).
Warning: classeFille is not a current module (created)
Yes
classeFille:  ?- add_import_module(classeFille, classeParente, end).
Yes
classeFille:  ?- connaissance(C).
C = classique ;
No
classeFille:  ?- assert(connaissance(moderne)).
Yes
classeFille:  ?- connaissance(C).
C = moderne ;
No
classeFille:  ?- retractall(connaissance(_)).
Yes
classeFille:  ?- connaissance(C).
No
classeFille:  ?- abolish(connaissance/1).
Yes
classeFille:  ?- connaissance(C).
C = classique ;
No
classeFille:  ?-
```

FIG. 5.4 – Un exemple de surdéfinition de connaissances entre modules.

les modules, mais il ne peut accéder *a priori* qu'aux faits et prédicats eux-mêmes définis dans `user`. Donc, l'écriture d'un prédicat commun à plusieurs modules pose problème s'il doit exploiter des prédicats définis dans le module appelant, et non dans le module de définition de ce méta-prédicat. Pour résoudre ce problème, `SWI-Prolog` permet de spécifier qu'un prédicat doit être exécuté dans le contexte appelant et non dans son module de définition. Cette déclaration se fait au moyen de la directive `module_transparent`. Cette relation est transitive : si plusieurs prédicats s'exploitent mutuellement et qu'ils sont tous déclarés "*transparents*", le prédicat le plus imbriqué dans le traitement sera exécuté dans le contexte d'appel du prédicat englobant.
La figure 5.5 page suivante illustre l'utilisation de cette directive, elle reproduit une session `Prolog` au cours de laquelle sont menés les tests suivants :

- un fichier de test est chargé et son contenu est listé : il définit deux versions différentes d'un prédicat dont le but est de trouver toutes les occurences du fait `solution/1` ;
- deux faits de ce type sont déclaré dans un module ;
- la version simple du prédicat est testée, elle renvoie une liste vide : le prédicat `toutes` appartenant au module `user`, il ne peut accéder aux faits `solution/1` du module `un_module` ;
- la version "*transparente*" du prédicat peut accéder aux faits déclarés dans `un_module` bien qu'il soit défini dans le module `user` : la transparence au module fait qu'il est exécuté dans le contexte appelant, en l'occurence dans le module `un_module`.

Dans cet exemple, la définition des prédicats est faite dans un fichier car ces directives ne sont pas gérées rigoureusement de la même manière en mode interactif. Nous ne rentrerons pas plus dans les subtilités de la gestion des modules `SWI-Prolog`. Ce mécanisme permet de définir un ensemble de prédicats communs aux modules qui effectueront un traitement contextualisé au module appelant. Pour notre modèle, il est donc possible de définir

```
?- [test_transparent].
% test_transparent compiled 0.00 sec, 1,236 bytes
Yes
?- listing.
:- dynamic solution/1.

:- module_transparent toutes_transparent/1.
toutes_transparent(A) :-
        findall(B, solution(B), A).

toutes(A) :-
        findall(B, solution(B), A).

Yes
?- un_module:assert(solution(1)).
Yes
?- un_module:assert(solution(2)).
Yes
?- module(un_module).
Yes
un_module:  ?- toutes(Liste).
Liste = [] ;
No
un_module:  ?- toutes_transparent(Liste).
Liste = [1, 2] ;
No
un_module:  ?-
```

FIG. 5.5 – Un exemple de prédicat "transparent" aux modules.

dans le module `user` les prédicats assurant la sélection d'action. Ces prédicats sont déclarés "*transparents*", ils exploitent donc les règles comportementales, descriptions d'action et connaissances propres à un agent s'ils sont invoqués depuis le module de cet agent.

L'utilisation conjointe des modules et des *thread* apporte donc des réponses techniques à la plupart des problèmes soulevés lors des deux premières implémentations. L'architecture finalement mise en place pour la troisième implémentation est détaillée dans la section suivante.

5.2.3 Intégration ARéVi - SWI-Prolog

L'architecture logicielle mise en place repose sur l'utilisation d'`ARéVi` et de `SWI-Prolog`. Ce dernier offre une `API C` qui permet de créer des requêtes et d'exploiter leurs résultats depuis un programme externe. Pour assurer la communication entre `ARéVi` et `Prolog`, nous avons utilisé[12] une interface `C++` qui est une simple encapsulation de l'`API C`. Par la suite l'emploi de cette interface sera implicitée, ses détails de fonctionnement étant sans intérêt pour exposer une vue d'ensemble de l'architecture.

La section précédente a mis en évidence l'interêt d'employer un module différent pour chaque agent afin de maîtriser le niveau de séparation des connaissances. De plus, un module sera utilisé pour chaque classe, un agent important le module de sa classe et une classe important le module de sa classe parente. L'héritage ne peut être multiple avec `ARéVi`, chaque

[12] Cette interface a originellement été conçue par WOLKER WYSK , nous l'avons simplement modifiée pour l'actualiser : sa maintenance n'étant plus assurée, les dernières modifications de `SWI-Prolog` n'étaient pas prise en compte et le code n'était pas compatible avec les compilateurs `C++` récents.
(`http://www.volker-wysk.de/swiprolog-c++/index.html`)

agent étant aussi un objet ARéVi la question de l'héritage multiple entre modules ne se pose pas.
En plus d'être un objet ARéVi auquel est associé un module, un agent intentionnel doit initier ses activités de raisonnement dans un *thread* séparé pour les problèmes d'ordonnancement que nous avons exposés.

Les inclusions de modules rendent la représentation de l'architecture délicate, la figure 5.6 page ci-contre illustre un cas simple : deux classes A et B sont schématisées. La classe B est une sous-classe de la classe A et deux instances ont été créées pour chacune d'elle (a1, a2, b1 et b2).
Cette figure montre les points suivants :

- le module de la classe B importe le module de la classe A (les modules sont nommés avec le nom de la classe préfixé par le caractère *m*) ;
- les modules mA_1 et mA_2 (respectivement mB_1 et mB_2) importent le module mA (respectivement mB) ;
- chaque agent possède un module associé pour lequel les requêtes sont exécutées au sein d'un *thread* dédié ;
- les agents intentionnels sont aussi des objets ARéVi liés à Prolog par l'identifiant du *thread* qui leur est dédié.

La moteur Prolog n'est donc instancié qu'une seule fois, de plus il est compilé sans l'interpréteur interactif (*top level*) afin d'obtenir un système plus léger et plus réactif. Enfin, un fichier Prolog définissant les prédicats de traitement pour la sélection d'action est précompilé et lié avec le reste du moteur et les classes ARéVi afin de générer un bibliothèque dynamique dédiée aux agents intentionnels (libIntentionalAgent.so).

La section suivante reprend les sept composantes du modèle d'agent intentionnel et détaille leur implémentation. A la lumière de ces explications, elle décrit l'architecture complète an donnant le détail des classes ARéVi mises en place et les mécanismes de liaison avec Prolog.

5.3 Principes du modèle d'exécution

5.3.1 Répartition de la définition d'un agent

Vue l'architecture adoptée, la définition d'un agent intentionnel est répartie entre l'instance ARévi qui l'implémente et l'espace de résolution Prolog qui lui est dédié (son module exploité dans son *thread*). Nous allons passer en revue les différentes composantes du modèle pour identifier clairement la localisation et la forme de leurs implémentations respectives.

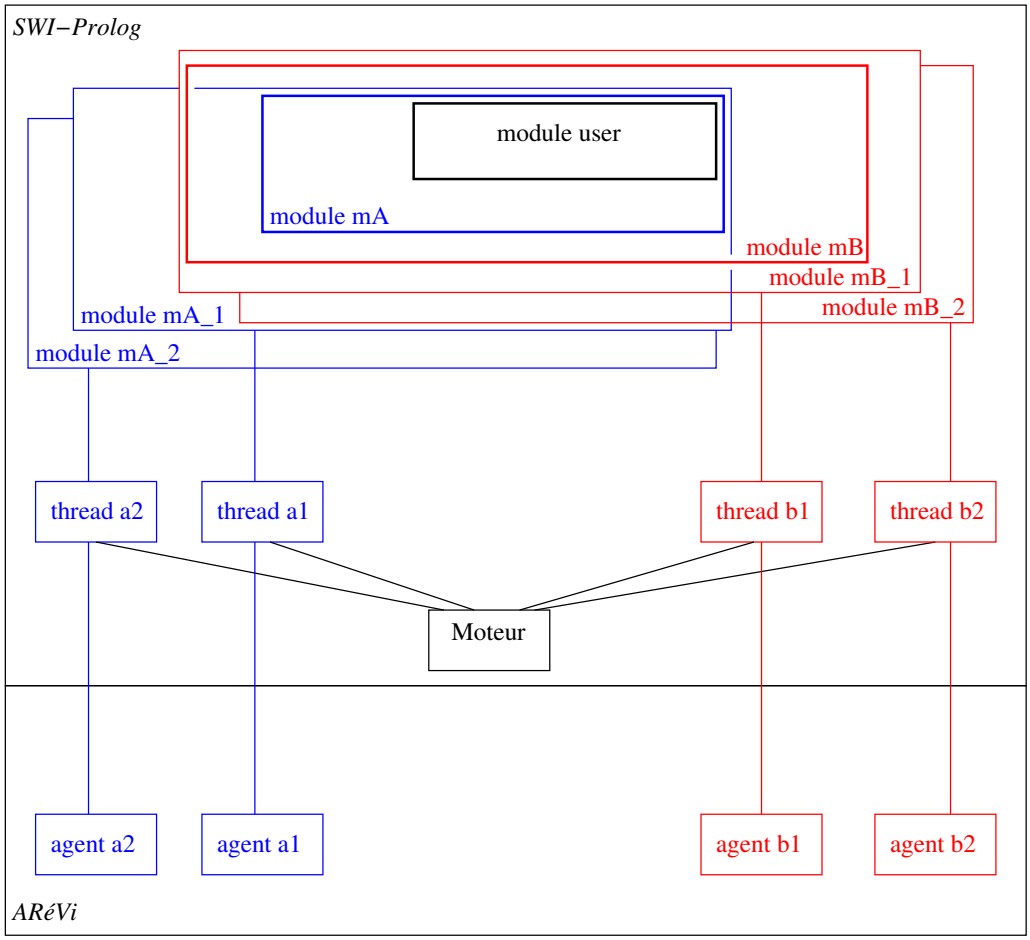

FIG. 5.6 – Troisième implémentation : une intégration de SWI-Prolog dans ARéVi.

Les méthodes

Les méthodes d'un agent intentionnel sont ses méthodes d'instance `ARéVi`. Ces méthodes ne sont toutefois pas invoquées directement, mais au travers d'un pointeur de méthode, ceci afin de permettre le prototypage interactif. Nous reviendrons sur ce point dans la suite de la section. Pour respecter l'approche proposée, les méthodes doivent effectuer un traitement atomique de granularité faible au regard des comportements spécifiés. Il s'agit de méthodes qui doivent être non bloquantes et qui ne prennent aucun paramètre : leur contextualisation se fait par la consultation des attributs de l'instance, toutes les valeurs pertinentes pour le contrôle de l'action doivent donc être explicitées. De même, une méthode n'a aucune valeur de retour : les effets de l'action se concrétisent par une interaction avec l'environnement ou par la modification de l'état interne (modification des attributs). Une action n'a pas de "résultat" : l'agent doit percevoir l'effet de ses actions, il ne le "*connaît*" pas. Par le mécanisme d'héritage, l'instance peut accéder aux méthodes de ses classes parentes : toutes les méthodes sont déclarés dans la section "*public*" de la classe (pour des contraintes concernant le prototypage interactif exposées ultérieurement, l'utilisation de la section "*private*" n'a pas été retenue).

Les méthodes sont écrites par l'informaticien : elles prennent en charge les spécificités de la plateforme. La figure 5.7 illustre la méthode `turnLeft` empruntée à l'exemple du prédateur (nous allons continuer à analyser le modèle d'exécution à travers cet exemple).

```
void
Creature::turnLeft(void)
{
double roll, pitch, yaw ;
self->_object3D->extractOrientation(roll, pitch, yaw ) ;
self->_object3D->setOrientation(roll- self->_angleStep, pitch, yaw ) ;
self->_energy-- ;
}
```

FIG. 5.7 – Un exemple de méthode : la méthode *turnLeft* de la classe *Predator*.

Le code fait apparaître des méthodes utilitaires de la classe `Object3D`, il s'agit typiquement de spécificités de la plateforme `ARéVi` qui relèvent du rôle de l'informaticien.
Dans cet exemple se trouve aussi employé à plusieurs reprises un auto-pointeur (`self`). Cette particularité introduite dans notre architecture peut être ignorée pour le moment et considérée transparente : son but est d'assurer une partie du mécanisme de prototypage interactif tel qu'il est présenté dans la section 5.3.3 page 127.

Les attributs

Les attributs d'un agent intentionnel sont ses attributs d'instance `ARéVi`. Par le mécanisme d'héritage, l'instance peut accéder aux attributs de ses classes parentes : tous les attributs sont déclarés dans la section "*public*" de la classe. Le type de ces attributs est totalement libre, l'informaticien peut donc déclarer des attributs utilitaires.

La figure 5.8 rassemble la totalité des attributs de la classe prédateur.

```
//----- Behaviour
int    _energy ;
double _step ;
double _angleStep ;
double _nearestPreyY ;
double _nearestPreyZ ;
//----- 3D
ArRef<Object3D> _object3D ;
ArRef<Cone3D>   _cone ;};
```

FIG. 5.8 – Des exemples d'attributs : les attributs de la classe *Predator*.

Dans cette liste se trouvent les attributs nécessaires à la spécification du comportement ou à l'exploitation des perceptions (`_energy`, `_step`, `_anglestep`, `_nearestPreyY` et `_nearestPreyZ`) et les attributs purement utilitaires. Dans ce cas il s'agit d'attributs nécessaires à la représentation (`_object3D` et `_cone`).

Les perceptions

La définition des perceptions est la déclaration des méthodes devant être exécutées cycliquement afin de renseigner certains attributs (espace de représentation des perceptions). Dans notre architecture, il s'agit d'une simple liste qui contient les noms de ces méthodes : toutes les classes qui implémentent un agent intentionnel héritent de la classe `IntentionalAgent`. Cette classe offre divers services dont un système de résolution de nom (*look up*) qui porte sur les méthodes : une méthode peut être retrouvée dans un annuaire dédié par son nom. Ainsi, le simple fait de compléter la liste des perceptions permet à l'activité de perception d'invoquer les méthodes correspondantes, qu'elles soient définies au sein de la classe ou de ses classes parentes.

Dans notre exemple, le prédateur déclare les méthodes `findNearestPrey` et `checkEnergy` dans sa liste de perceptions.

Les propriétés

Les perceptions sont des méthodes d'inspection et d'abstraction des attributs qui doivent être consultables depuis le *thread* `Prolog` afin d'évaluer les règles comportementales.
Il s'agit donc avant tout de méthodes `ARéVi` qui sont, elles-aussi, manipulables par leur nom. Elles doivent implémenter une interface précise : elles n'ont pas de paramètre mais possèdent une valeur de retour correspondant au type de la propriété. Les types valides sont :

- `double`;
- `int`;
- `string`.

Leur nom est normalisé : il se compose de la concaténation de la chaîne '`get`', du type débutant par une majuscule et du nom de la propriété préfixé par le caractère '_'.
Par exemple, la méthode qui calcule la valeur instantanée de la propriété `nearestPreyAngle` (de type double) a pour signature :
`double getDouble_nearestPreyDistance(void);`
Cette méthode est en charge de l'adaptation des valeurs pertinentes des attributs pour un traitement par la partie décisionnelle.
Dans notre exemple, il s'agit d'un simple calcul de trigonométrie, la figure 5.9 illustre la manière dont il est effectué grâce aux méthodes utilitaires de la plateforme. L'abstraction pourrait être plus fine (arrondi, seuillage...).

```
double
Predator::getDouble_nearestPreyAngle(void)
{
double nearestPreyAngle;
nearestPreyAngle = cart2DToAngle(self->_nearestPreyY,self->_nearestPreyZ)-M_PI_2;
return nearestPreyAngle;
}
```

FIG. 5.9 – Un exemple de propriété : le calcul de *nearestPreyAngle* de la classe *Predator*.

Il subsiste un point plus délicat : ces méthodes doivent être invoquées depuis la partie décisionnelle écrite en Prolog.
Cette fonctionnalité nécessite la mise en place d'un mécanisme d'introspection du côté d'ARéVi d'une part, et la mise en place d'un dispositif permettant de recourir à cette introspection du côté de Prolog d'autre part.

Pour l'introspection, la classe IntentionalAgent dont hérite toute classe d'agent intentionnel offre ce service par l'intermédiaire de trois méthodes :

- `bool getIntPropertyValue(const StlString & propertyName, long int & value);` tente d'affecter à `value` la valeur d'une propriété de type `int` dont le nom est spécifié (`propertyName`). Cet appel est susceptible d'échouer (la propriété demandée n'existe pas par exemple) ;
- la méthode `getDoublePropertyValue` fait de même pour une propriété de type `double` ;
- la méthode `getStringPropertyValue` fait de même pour une propriété de type `string`.

Il suffit donc d'invoquer cette méthode avec le nom de la propriété sur la bonne instance pour obtenir la valeur souhaitée.
Pour ce faire, un prédicat externe (donc écrit en C++ *via* l'API) est ajouté au moteur Prolog. Il s'agit d'une fonction globale qui prend en paramètre le nom de la classe, le numéro de l'instance et le nom de la propriété et qui unifie le résultat obtenu avec la valeur de la propriété correspondante ou échoue (les paramètres sont des termes Prolog). Sa signature est donc :

```
bool plForeignPredGetValue(Term className,Term instanceId,
Term property, Term value);
```

La figure 5.10 page suivante détaille cette fonction :

- récupération de la classe dont l'agent est une instance ;
- récupération de cette instance *via* les services de classe grâce au numéro d'instance (service ARéVi) ;
- appel de la méthode d'introspection adaptée au type requis.

Le nom de la classe et le numéro d'instance sont déclaré dans le module de l'agent à sa construction. Il reste donc à rendre cette fonction accessible à Prolog. Ceci est fait grâce à la macro suivante (fonctionnalité de l'API C++) :

```
foreign( plForeignPredGetValue, "getValue", 4)
```

Ainsi, la fonction devient accessible comme n'importe quel autre prédicat (`getValue/4`).
Enfin, un prédicat (`getProperty(+PropertyName,?Value)`) est déclaré dans l'extension au moteur afin de dispenser l'utilisateur d'avoir à manipuler le nom de la classe et le numéro d'instance : un agent ne consulte que ses propres propriétés. La figure 5.11 page ci-contre donne un extrait du fichier Prolog compilé avec le moteur d'inférence.

Ce mécanisme est un point fondamental de notre architecture : il permet la consultation des valeurs de propriétés par la partie décisionnelle de manière asynchrone. Le calcul de la valeur est effectué au moment où la règle Prolog le requiert. Dans l'hypothèse d'un traitement relativement complexe dans une règle comportementale, le spécificateur a deux solutions :

- soit il consulte la valeur de propriété au début de la règle principale : la totalité du traitement est effectuée sur la base de cette valeur, il peut arriver qu'elle ne soit plus pertinente lorsque la

```
bool
plForeignPredGetValue(Term className,Term instanceId, Term property, Term value)
{
  ArRef<ArClass> ia = ArClass::find((const char *)className) ;
  if (!ia)
    return FALSE ;
  ArRef<IntentionalAgent> agent ;

  ia->getInstance((long)instanceId,agent) ;
  if (!agent)
    return FALSE ;


  // property is a double
  double d ;
  if (agent->getDoublePropertyValue((const char *)property,d))
    {
     return value.unify(d) ;
    }
  // property is a long
   long l ;
   if (agent->getIntPropertyValue((const char *)property,l))
     {
       return value.unify(l) ;
     }
   // property is a string
   StlString s ;
   if (agent->getStringPropertyValue((const char *)property,s))
     {
      return value.unify(s) ;
     }
   return false ;
}
```

FIG. 5.10 – La méthode *plForeignGetValue* : un pont entre impératif et déclaratif.

```
% An intentional agent knows its class and instance id
% (see intentionalAgent.cpp)
:-dynamic(this_class/1).
:-dynamic(this_id/1).

% getProperty(+PropertyName, ?PropertyValue)
%-----------------------------------------
%
% retrieve the Value of a Property declared in AReVi
%
:-module_transparent  getProperty/2.
getProperty(Property,Value):-
    context_module(M),
    M:this_class(Class),
    M:this_id(Id),
    M:getProperty(Class,Id,Property,Value).
% getProperty(+AgentClass, +AgentId, +PropertyName, ?PropertyValue)
%-----------------------------------------------------------------
%
% retrieve the Value of a Property declared in AReVi
% from the class Id and the instance Id of an agent
% througth a call to getValue which is foreign (see intentionalAgent.cpp)
:-module_transparent  getProperty/4.
getProperty(Class,Id,Property,Value) :-
     getValue(Class,Id,Property,Value).
```

FIG. 5.11 – Le prédicat *getProperty* : l'interface impératif/déclaratif du côté de Prolog.

résolution arrive à son terme ;

– soit il invoque le prédicat `getValue/2` au plus près du test qui concerne cette valeur, éventuellement aux différents endroits où cette propriété est prise en compte : la décision prend alors en considération l'évolution de la valeur.

La deuxième solution peut s'avérer particulièrement intéressante : un agent rationnel qui raisonne tout en agissant peut au bout de quelques instants de réflexion s'engager dans une voie qui n'était pas valide au début de cette réflexion : le dynamisme de son environnement est pris en compte. A l'inverse, il est possible qu'il s'engage dans une voie dont les conditions de validité ne se vérifient plus au bout de quelques instants. La gestion de différentes fréquences de perception, de décision et d'action prend alors tout son sens.
L'asynchronisme de cette consultation ne peut créer d'effets de bord nuisibles : le mécanisme de *thread* garantit que l'invocation d'une méthode d'introspection ne peut se faire pendant une modification des attributs.

Les règles comportementales

Les règles comportementales sont de la forme :

$$b(\{c_0, \ldots, c_j\}, \{\{\{t_0, p_0\}, s_0\}, \ldots, \{\{t_n, p_n\}, s_n\}\})$$

où les c_i sont les conditions de validité et les t_i, p_i et s_i désignent respectivement les tendances, propriétés et forces générées.

Cette forme est parfaitement adaptée à l'expression d'une règle `Prolog`. Les règles comportementales sont donc des prédicats du type `behaviouralRule/1`. Ces règles sont vraies sous les conditions de validités spécifiées et génèrent alors une liste de triplets de la forme *propriété-tendance-force*.
La figure 5.12 liste les règles comportementales du prédateur.

```
behaviouralRule([happiness-increase-weak]) :-
                                            getProperty('energy',E),
                                            E > 5000.
behaviouralRule([energy-increase-normal]) :-
                                            getProperty('energy',E),
                                            E < 7000,
                                            getProperty('nearestPreyDistance', D),
                                            D \== 0,
                                            D < 20.
```

FIG. 5.12 – Exemple de règles comportementales : les règles du prédateur.

Une règle comportementale supplémentaire a été introduite par rapport à l'explication du chapitre précédent : en l'absence de connaissances des effets présumés de l'action `randomWalk` et de règle comportementale correspondante, l'agent n'a jamais de "*raison*" de se promener. Nous avons donc introduit la première règle qui incite l'agent à accroître sa joie tant qu'il n'est pas fatigué. De même, nous déclarerons une description d'action indiquant que se promener rend joyeux !
Des seuils sur l'énergie ont été choisis arbitrairement : l'agent cesse la marche aléatoire quand son énergie est inférieure à 5 000 et commence à chasser dès qu'elle est inférieure à 7 000. De plus, la distance maximum au-delà de laquelle une proie n'est plus considérée proche est fixée à 20 (dans un exemple aussi simple, nous n'avons pas fixé d'unités : les méthodes de perception seraient à adapter selon la plateforme le cas échéant).

Ces règles comportementales illustrent l'utilisation du prédicat `getProperty/2` pour interroger les valeurs instantanées de propriétés. Le deuxième argument est dans ce cas toujours une variable libre (demande de valeur), le prédicat peut aussi être exploité avec deux valeurs (vérification de la valeur présumée d'une propriété).

Les descriptions d'action

De manière analogue aux règles comportementales, les descriptions d'action peuvent se transcrire aisément en prédicats `Prolog`. Rappelons que les descriptions d'action sont de la forme :
$d = \{a, \{\{t_{a_0}, p_0\}, \ldots, \{t_{a_n}, p_n\}\}$ où t_{a_j} désigne la tendance sur la propriété p_j résultante de l'action a (effet présumé).

Les prédicats correspondant seront donc du type `action/2` où le premier argument est le nom de l'action et le deuxième est une liste de doublets *propriété-tendance*.
La figure 5.13 liste les descriptions d'action du prédateur.

```
action(randomWalk, [happiness-increase]).
action(purchase, [energy-increase, nearestPreyDistance-decrease]).
```

FIG. 5.13 – Exemple de descriptions d'action : les descriptions des actions du prédateur.

Les connaissances

Par définition, les connaissances sont constituées de code `Prolog` laissé à la discrétion du spécificateur. Dans l'exemple simple du prédateur, il n'est pas nécessaire de recourir à des prédicats utilitaires ou à la mémorisation de résultats. Ce-dernier ne possède donc aucune connaissance.

Les connaissances peuvent aussi bien prendre la forme de faits que celle de prédicats. Le seul impératif est de les déclarer dans le module correspondant à l'agent.

5.3.2 Le déroulement de la simulation

Après avoir détaillé la forme et la localisation des composantes du modèle d'agent intentionnel dans notre architecture, il nous reste à préciser la manière dont elle exploite ces composantes pour générer le comportement de l'agent.

L'instanciation d'un agent intentionnel se décompose comme suit :
- l'objet `ARéVi` correspondant est créé ;
- un *thread* dédié à l'exécution des requêtes de l'agent est créé *via* le moteur `Prolog`, son numéro d'identification est enregistré comme attribut de l'objet `ARéVi` ;
- au moyen de ce *thread*, le module dédié à l'agent est créé dans l'espace `Prolog` et le nom de la classe ainsi que le numéro d'identification de l'instance `ARéVi` sont déclarés dans ce module (ces informations sont exploitées par les prédicats assurant la communication impératif/déclaratif) ;

- le module de la classe parente est ajouté à la liste des modules importés ;
- trois activités ARéVi sont créées pour cet agent : une activité de perception, une d'action et une de décision.

Une fois cette phase de création terminée, la "*vie*" de l'agent est entièrement définie par l'exécution cyclique de ces trois activités (chacune possède sa fréquence d'activation).

Nous allons donc exposer le fonctionnement de ces trois activités.

L'activité de perception

Le rôle de l'activité de perception est d'exécuter toutes les méthodes déclarées comme telles. Son fonctionnement est donc relativement simple : lorsqu'elle se déclenche, elle parcourt la liste des perceptions de l'agent et invoque ces méthodes.
Comme expliqué précédemment, les actions peuvent être invoquées dès que leur nom est défini : un mécanisme de recherche défini dans la classe `IntentionalAgent` se charge de retrouver la méthode correspondante. Pour permettre le prototypage interactif nous évitons de nous reposer sur le mécanisme d'appel de méthodes du `C++`. Les raisons sous-jacentes de ce choix sont exposées dans la section suivante.

Ces actions étant exécutées, elles renseignent les attributs en fonction de l'état interne de l'agent et de son environnement.

L'activité d'action

Le rôle de l'activité d'action et son fonctionnement sont assez similaires à ceux de l'activité de perception.
Le but est ici d'interroger la partie décisionnelle pour connaître le nom des actions à exécuter. Par convention, la liste de ces actions est enregistrée sous la forme d'un fait (`decision/1`). Pour l'activité, il suffit donc d'exécuter une requête `Prolog` au sein du module dédié à l'agent pour obtenir la dernière décision en date. Selon les fréquences respectives des trois activités, il se peut que l'activité d'action manque des décisions, ou au contraire exécute plusieur fois la même décision. Une fois cette liste récupérée en résultat de la requête, il reste à la parcourir élément par élément et à invoquer systématiquement les actions correspondantes. Ici encore, le dispositif de résolution de nom identifie les méthodes correspondantes.

C'est l'exécution de ces actions qui réalise concrètement les interactions de l'agent.

L'activité de décision

Le rôle de l'activité de décision est d'effectuer la sélection d'action en exploitant les règles comportementales, les descriptions d'action et les éventuelles connaissances. Or, ce traitement est réalisé au sein de la partie `Prolog` de l'architecture. L'activité ARéVi doit donc se contenter de déclencher ce processus. De plus, ce déclenchement doit être non bloquant : le but est de laisser le mécanisme d'ordonnancement des *thread* par le système d'exploitation équilibrer la charge entre les différentes parties (le processus ARéVi d'une part et les *thread* `Prolog` d'autre part).

Pour cela, nous avons eu recours au mécanisme de boîte à message dont sont dotés les *thread* `Prolog`. L'activité de décision exécute donc une requête très rapide qui se contente de "*poster*" un message au *thread* dédié à l'agent. Ce message n'a aucun effet si le *thread* est déjà en train de résoudre une sélection d'action. Par contre, au repos le *thread* est en attente passive de message. Cet envoi de la part de l'activité `ARéVi` "*réveille*" donc le *thread* qui procède à la sélection (interrogation des propriétés, évaluation des règles comportementales...).
Si cette résolution est brève, le *thread* se remet en attente sur sa boîte à message après avoir mis à jour le fait `decision/1` qui permet à l'activité d'action de faire le lien déclaratif/impératif. Sinon, au bout de quelques instants le *thread* est préempté par le système qui rend la main à `ARéVi`. Ce mécanisme assure donc l'interruptibilité du moteur d'inférence qui faisait défaut aux précédentes implémentations.

Le processus de sélection en lui-même n'est pas détaillé ici. Il est implémenté conformément à la spécification exposée au chapitre précédent par une suite de prédicats "*transparents*" aux modules. L'intégralité du fichier qui réalise l'extension de `Prolog` pour notre architecture est reporté en annexe B.1 page 179.

5.3.3 Le prototypage interactif

Le dernier point important de notre approche que nous n'ayions pas développé jusqu'à présent est l'établissement de mécanismes de prototypage interactif.

Pour la partie déclarative, ce but est très facile à atteindre : l'ensemble des traitements nécessaires à la sélection d'action est défini sous forme de faits ou de prédicats. Par surdéfinition, il est donc possible de modifier intégralement le mécanisme de sélection qui est au cœur du modèle. De manière moins radicale, les règles comportementales, les descriptions d'action et les connaissances peuvent être modifiées dynamiquement.

Les perceptions sont énumérées dans une liste qui est un attribut d'instance. La modification des actions à exécuter pour assurer les perceptions n'est donc pas un obstacle.

La grande difficulté provient donc de la volonté de rendre les attributs, les propriétés et les méthodes modifiables. Le choix d'`ARéVi` imposant le recours au `C++`, le code correspondant à ces composantes du modèle est compilé.
Notre propos est de proposer une démarche de développement facilitant les interventions des différents spécialistes impliqués dans la spécification, nous nous devons alors de proposer des solutions à ces problèmes de prototypage interactif.

L'architecture que nous avons mise en place pour répondre à cette problèmatique repose sur la réification des objets manipulés, à savoir les classes d'agents intentionnels, les instances d'agents intentionnels, mais aussi les sept composantes du modèle.

La réification des classes et des instances permet d'implémenter les services suivants :

- déclaration et retrait dynamique des sept composantes du modèle ;
- sauvegarde et chargement de la définition complète de l'agent (parties impérative et déclarative) dans un format unique (`XML`) ;

- introspection complète (examen de la définition de toute composante, explicitation de la hiérarchie de classe...) ;
- modification dynamique des fréquences des trois activités d'un agent (perception, décision et action) ;
- mise en place d'une granularité instance (modification d'*un* agent, création d'une nouvelle classe à partir d'une instance spécialisée) ;
- évaluation de code libre "*à la volée*" (code `Prolog` ou code `C++`).

Le but de l'instauration de ces services était de se rapprocher au plus de la souplesse d'utilisation d'un langage interprété comme `oRis`.

Le premier point important de cette approche est la réification des notions de classe et d'agent. En effet, une méthode peut être une méthode de classe ou une méthode d'instance. Il en va de même pour les propriétés, les connaissances et toutes les autres composantes du modèle. Les classes et les instances d'agents intentionnels possèdent donc pratiquement la même structure. Ce constat conduit à définir la classe `IntentionalObject` qui offre les services de base :

- manipulation des composantes du modèle (déclaration, modification et retrait) ;
- évaluation dynamique de code `Prolog` ou `C++` dans le contexte de l'objet ;
- sauvegarde et chargement de la définition *via* `XML`.

Cette classe doit s'inscrire dans la hiérarchie de classes d'`ARéVi` afin de permettre l'exploitation des fonctionnalités de cette bibliothèque (ordonnanceur, facilités de représentation graphique...). La classe `IntentionalObject` hérite donc de la classe `ArObject`, c'est le point d'entrée normal pour exploiter `ARéVi`.

A partir de cette classe abstraite (`IntentionalObject`) dérivent deux classes implémentant les notions de classe et d'instance d'agent intentionnel :
`IntentionalClass` et `IntentionalAgent`.
Ces deux classes implémentent complètement l'interface de la classe `IntentionalObject` en prenant en compte les différences entre les deux notions[13]. Par exemple l'appel de la méthode d'inspection du parent renvoie le nom de la classe intentionnelle correspondante s'il elle est invoquée sur une instance de `IntentionalAgent` alors qu'elle renvoie le nom de la classe parente si elle est invoquée sur une instance de `IntentionalClass`. L'exception à ce comportement est l'instance de `IntentionalClass` qui implémente la classe `IntentionalAgent` : elle est son propre parent. Le lecteur coutumier de `SmallTalk` aura immédiatement reconnu le mécanisme des méta-classes de ce langage.
Ainsi, dès qu'une classe d'agent intentionnel est manipulée, une instance de `IntentionalClass` correspondante est créée si elle n'existe pas déjà. Il n'est donc pas nécessaire de créer une instance de cette classe d'agent intentionnel pour qu'elle soit réifiée. Prenons un exemple concret. Nous partons du principe que des mécanismes communs seront mis en place pour les prédateurs et pour les proies (déplacement, marche aléatoire...). Les classes `Predator` et `Prey` héritent donc toutes les deux de la classe `Creature`, une sorte d'agent intentionnel. Le simple fait de déclarer les relations d'héritage, ou de déclarer les attributs par exemple entraîne l'instanciation des classes correspondantes. Imaginons que les classes `Creature` et `Predator` soient entièrement définies, alors que la classe `Prey` ne l'est que partiellement (aucune instance ne peut être créée). Le concepteur crée alors une

[13](une instance de la classe `intentionalClass` est nécessairement unique, cette classe implémente le *design patern* `singleton`)

instance de la classe `Predator` : les trois classes ont été réifiées et il existe une instance de la classe `Predator`. La classe `Prey` n'existe pas encore, bien qu'une instance de `IntentionalClass` contenant sa définition partielle existe. La figure 5.14 représente l'état du système à ce moment précis par un diagramme de structure statique.

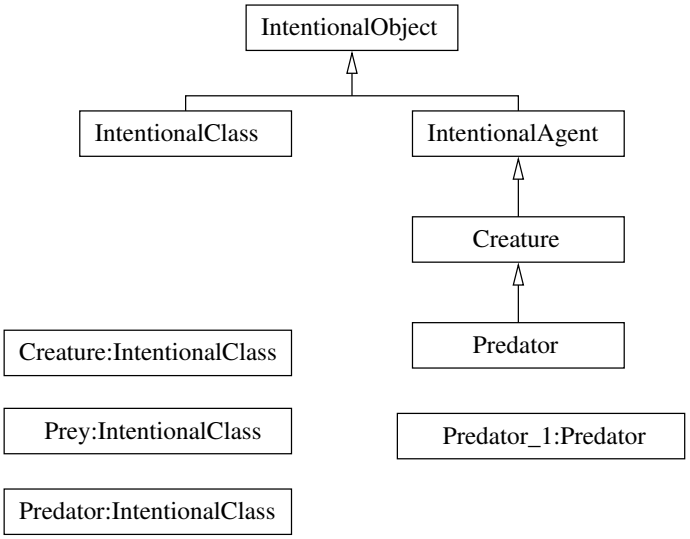

FIG. 5.14 – Un exemple de réification de classes : diagramme de structure statique.

Cette architecture permet ainsi de manipuler les classes et les instances d'agents intentionnels pratiquement de la même façon. Ces deux classes possèdent évidemment des interfaces légèrement différentes.

La classe `IntentionalAgent` permet de :
- régler les fréquences des trois activités du modèle (perception, décision et action) ;
- générer une nouvelle classe à partir de l'instance.

Pour sa part, la classe `IntentionalClass` permet la gestion de tout le code de classe :
- les préambules aux fichiers d'inclusion (`.h`) et d'implémentation (`.cpp`) permettent d'inclure des fichiers utilitaires ;
- le code du constructeur permet d'effectuer des initialisations propres à la bibliothèque `ARéVi` ;
- le code du destructeur permet d'effectuer des traitements spécifiques à la destruction d'une instance de la classe.

Le but de la mise en place de cette organisation est de permettre aux classes d'agents intentionnels et aux instances d'agents intentionnels de générer l'ensemble du code (`Prolog` et `C++`) nécessaire à leur implémentation.

C'est par ce moyen que nous satisfaisons aux exigences du prototypage interactif : l'explicitation au sein des instances de classes d'agent et au sein des instances d'agents du code nécessaire à leur définition leur permet de se regénérer à chaque modification. Il nous faut donc distinguer à nouveau la part de la définition exprimée en `Prolog` et la part définie en `C++`.

Du point de vue de la classe `IntentionalObject`, cinq des sept composantes sont traitées de manière uniforme. En effet, nous ne parlerons plus pour la suite de cette section de la liste des perceptions : toutes les opérations sont triviales (déclaration, modification, sauvegarde...) puisqu'il s'agit d'une simple liste de noms d'action. De même, les attributs ne sont pas réifiés dans cette implémentation du modèle : ils sont définis à la compilation de la classe ou de l'instance d'agent intentionnel. Il n'est donc pas possible d'ajouter ou de retirer dynamiquement des attributs (l'intérêt d'une telle possibilité paraît faible au regard du surcoût en matière de conception).

Il reste donc à considérer :
- les composantes définies dans la partie `Prolog` (règles comportementales, descriptions d'action et connaissances) ;
- les composantes définies dans la partie `C++` (les méthodes et les propriétés).

Pour que ces composantes soient manipulées uniformément par les instances du type `IntentionalObject`, elles sont toutes réifiées par des classes dérivant d'une unique interface : la classe abstraite `DynamicComponent`.
Cette classe offre les services suivant :
- création de la composante depuis une définition `XML` ;
- production d'une représentation `XML` de la composante ;
- génération du code correspondant à une implémentation "dynamique" ;
- génération du code correspondant à une implémentation "statique".

Les deux premiers points sont triviaux : en passant en revue l'ensemble de ses composantes et en invoquant leur méthode de production d'une représentation `XML`, une instance du type `IntentionalObject` peut générer une représentation `XML` de sa définition complète (il suffit d'insérer les informations qui ne sont pas des composantes pour obtenir un description complète : nom, nom du parent, attributs, liste de perceptions...). Réciproquement, une instance peut être construite par l'examen d'un fichier de description `XML` valide.

Les deux derniers points appellent par contre un commentaire détaillé. En effet, la génération de code ne pose aucun problème pour les composantes de la partie `Prolog` : il suffit de produire la chaîne de caractères à déclarer dans `Prolog` pour construire la règle ou le fait concerné. Le réalisation de l'introspection est ici triviale. Les deux types de génération de code indiqués ici ("dynamique" et "statique") sont dans ce cas rigoureusement équivalents. L'avantage de la réification de ces composantes est qu'elles deviennent ainsi des variables automatiques : il suffit de détruire l'objet `C++` pour que la clause correspondante soit retirée du module `Prolog` de l'agent (*via* le mécanisme du destructeur de la composante). La gestion du retrait de connaissance (au sens large) qui aurait pu paraître problématique est ainsi automatique : la connaissance a la même durée de vie que l'objet `C++` qui la réifie. L'autre avantage majeur est l'échange d'information : pour générer une nouvelle classe à partir d'une instance spécialisée par exemple, il suffit de copier les instances de `DynamicComponent`. Il n'est pas nécessaire de passer par une procédure de "*listing*" du module de l'agent.
Pour les composantes de la partie `C++` par contre (méthodes et propriétés), les deux types de génération de code sont différenciés. En effet, l'idée de régénérer le code des classes ou des instances d'agents intentionnels ne pose aucun problème pour la partie `Prolog` : il suffit de retirer les clauses modifiées pour déclarer les nouvelles versions. Pour la partie `C++`, le défi est plus important. Nous avons décidé de recourir à un système de compilation en arrière-plan de l'application et de charge-

ment à la volée : les instances de `IntentionalClass` et de `IntentionalAgent` sont toutes compilées dans des librairies dynamiques différentes. Lorsque le code d'une telle instance est modifié, la librairie correspondante est recompilée et, si la compilation n'échoue pas, la version courante de la librairie est déchargée au profit de la nouvelle version.
Ce mécanisme présente un inconvénient majeur : si il n'est pas affiné il n'apporte qu'une sorte d'aide à la compilation. Le déchargement de la librairie impliquant la destruction des toutes les entités reposant sur la classe définie dans la librairie, toute modification de la classe entraîne la destruction de toutes les instances. Il n'est pas alors vraiment question de prototypage interactif !
Nous avons donc "*doublé*" ce mécanisme : la définition d'une méthode ou d'une propriété pourra donner lieu à la génération d'un code "statique" ou "dynamique". Pour notre architecture, ces adjectifs ne sont pas à interpréter dans leur sens habituel en informatique. En fait, ces deux générations seront traitées avec une légère différence :

- la génération de code "statique" produit le code tel qu'il doit être écrit dans le fichier de définition de la classe `C++` ;
- la génération de code "dynamique" produit le code tel qu'il doit être écrit dans une fonction `C++` externe à la définition de la classe.

Pour une même instance d'`IntentionalObject`, deux librairies partagée sont susceptibles d'être chargées simultanément : la librairie qui contient la définition de la classe est obligatoirement chargée, une librairie qui contient la surcharge de certaines méthodes ou propriétés est éventuellement chargée en plus.
Ce doublement de la librairie offre la souplesse désirée pour l'informaticien. Les méthodes et propriétés sont définies une première fois, ce qui donne lieu à la compilation initiale de la librairie de base. En cours de simulation, l'informaticien peut modifier ces composantes ou en ajouter d'autres. Tant qu'il considère être en phase de mise au point, son intérêt est de déclarer ces composantes "dynamiques". Ainsi, chaque modification entraîne la compilation de la librairie additionnelle en arrière-plan et son chargement à la volée en lieu et place de l'ancienne version. Mais la librairie principale, qui contient la définition de la classe, n'a pas été modifiée. Elle n'est donc pas déchargée et les entités qui dépendent de la classe qu'elle implémente ne sont pas détruites. Quand l'informaticien considère la mise au point terminée, il peut déclarer que les nouvelles versions de ces composantes sont désormais "statiques" : ceci nécessite de recompiler la librairie principale (ce qui détruit les instances), mais la définition de ces nouvelles composantes fait désormais partie intégrante de la classe d'agent intentionnel.

L'avantage de ce système est double : non seulement la simulation en cours continue alors que certaines méthodes sont modifiées, mais en plus la compilation en arrière-plan est relativement rapide parce que la librairie additionnelle ne contient que les méthodes et propriétés modifiées.

Le dispositif technique à mettre en place pour cette fonctionnalité n'est pas très compliqué. L'essentiel du fonctionnement réside dans la méthode de résolution de noms. Lorsqu'une instance de `IntentionalAgent` invoque une méthode par son nom, le mécanisme de résolution de noms cherche dans l'ordre :

- dans la librairie additionnelle de l'instance (code "dynamique") ;
- dans la librairie additionnelle de la classe (code "dynamique" de classe) ;
- dans la librairie de base de la classe (code "statique" de classe) ;
- ...

Cette méthode est récursive : si le nom de la méthode n'est pas trouvé au sein de l'instance, la méthode de résolution s'appelle récursivement sur les classes parentes. La hiérarchie de classe est ainsi "*remontée*" jusqu'à ce que la méthode soit trouvée, ou échoue.

La figure 5.15 illustre l'organisation des différentes librairies partagées dans l'exemple du prédateur. Le cas de figure représenté est le plus complet qui puisse exister : pour les deux classes certaines méthodes ou propriétés de classe (`Creature.so` et `Predator.so`) sont surchargées par des versions "dynamiques" (`Creature_class.so` et `Predator_class.so`). De plus, l'instance `Predator_1` est spécialisée : sa définition surcharge certaines méthodes ou propriétés (`Predator_1.so`). Le trait interrompu représente le parcours d'une résolution de nom (*lookup* ascendant) : ce chemin est suivi jusqu'à ce qu'une version de la méthode ou de la propriété invoquée soit trouvée.

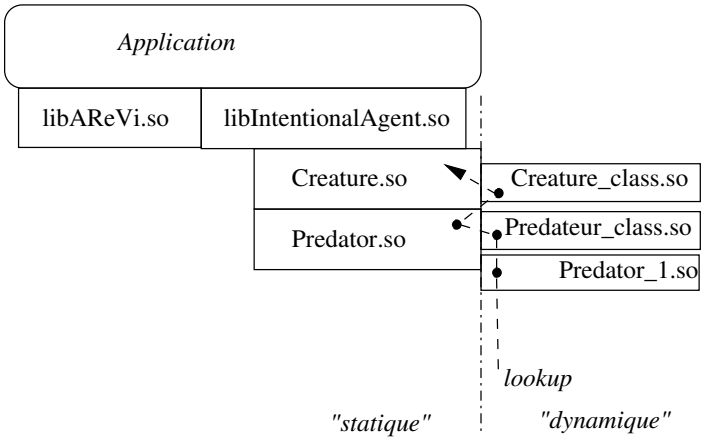

FIG. 5.15 – Exemple d'organisation des librairies dynamiques : le cas du prédateur.

Le point techniquement délicat est de pouvoir définir dans une fonction extérieure à la classe un traitement qui soit effectué dans le contexte de la classe. De plus, la génération de ces deux types de code ("statique" dans la librairie de base et "dynamique" dans la librairie additionnelle) doit être totalement transparente pour l'utilisateur, sinon il n'est plus question de prétendre lui faciliter la tâche ! C'est ici qu'intervient l'auto-pointeur `self` rencontré dans l'exemple de méthode du prédateur. La seule chose qui est demandée à l'utilisateur de notre modèle est d'écrire l'accès aux attributs de l'agent au moyen de ce pointeur. Ce mécanisme permet d'écrire le code généré indifféremment dans une méthode de classe ou dans une fonction externe. En effet, chaque instance d'`IntentionalObject` possède un pointeur sur elle-même `self` (initialisé à la construction de l'objet `C++`). Cette indirection est donc transparente lorsqu'elle apparaît dans une méthode de classe. Par contre, lors de la génération de code "dynamique", la fonction créée dans la librairie additionnelle reçoit ce pointeur en paramètre. Comme tous les attributs sont déclarés dans la section "`public`", la fonction externe peut accéder à n'importe quel attribut de l'instance grâce à ce pointeur. Ce subterfuge permet d'utiliser le même code pour la définition d'une méthode de classe ou d'une fonction externe.

Prenons l'exemple de la définition de l'action `checkEnergy` de la classe `Creature`. Le but de cette action perceptive est de vérifier que le créature n'a pas atteint un niveau d'énergie nul, le cas échéant son "*pas*" est mis à zéro : la créature ne peut plus se déplacer. La figure 5.16 page ci-contre

montre le code tel que le saisit l'utilisateur. La figure 5.17 illustre le code obtenu par une génération "statique" : ce code sera employé pour la génération du code de la classe. La figure 5.18 enfin permet de voir l'intérêt de l'auto-pointeur `self` : lors de la génération de code "dynamique", le pointeur sur l'instance invoquant la méthode est passé à la fonction externe ainsi définie. Ce mécanisme permet donc d'exploiter le code défini par l'utilisateur indifféremment au sein de la classe ou dans une fonction externe.

```
if (self->_energy <= 0)
  self->_step = 0 ;
```

FIG. 5.16 – Exemple de génération de code : les instructions saisies par l'utilisateur.

```
void
Creature::checkEnergy(void)
{
if (self->_energy <= 0)
  self->_step = 0 ;
}
```

FIG. 5.17 – Exemple de génération de code "statique" : création d'une méthode de classe.

```
// This file was generated by Creature
#include "./dynamicSrc/Creature.h"
extern "C"
void
Creature_checkEnergy(IntentionalAgent * owner)
{
 Creature * self  ; self = (Creature*) owner ;
if (self->_energy <= 0)
  self->_step = 0 ;
}
```

FIG. 5.18 – Exemple de génération de code "dynamique" : création d'une fonction externe.

La méthode `eval` qui permet d'exécuter du code `C++` dans le contexte d'une classe ou d'une instance d'agent intentionnel repose sur le même principe : une librairie dynamique contenant une fonction composée du code passé en paramètre est créée et chargée à la volée. La fonction est alors invoquée avec `self` en paramètre. Ce mécanisme revient pratiquement à de l'interprétation de code `C++`, au délai de compilation près...

Les méthodes et les propriétés de l'agent sont donc déclinées en deux versions : les versions "statiques" et "dynamiques". De plus les propriétés sont encore divisées selon leur type. Enfin, les connaissances, règles comportementales et descriptions d'action font appel à des mécanismes très proches. Les classes qui les réifient héritent donc d'une base commune.
Tout ceci nous a conduit à définir l'arborescence de classes illustrée par la figure 5.19 page 135.

5.4 Conclusion : nécessité d'un outil dédié

Nous avons montré comment la progression de nos essais successifs nous a amenés à réaliser l'implémentation de notre proposition. L'architecture mise en place repose sur des mécanismes avancés

de SWI-Prolog et l'utilisation de la bibliothèque ARéVi. Le fait de vouloir participer au développement de cette bibliothèque et ainsi de bénéficier de son fort potentiel à la réalisation de simulations d'environnements virtuels nous impose toutefois des contraintes de développement. Ainsi, le fait d'utiliser le C++ nous a conduit à mettre en place un dispositif de compilation en arrière-plan et de chargement à la volée de librairies dynamiques.
La mixité de paradigme de programmation exigée par notre proposition introduit nécessairement ce genre de complication. Une approche par réification, c'est à dire la création d'un nouveau langage dédié à ce genre de modèle permettrait de résoudre plus efficacement ces problèmes. L'implémentation présentée ici est considérée avant tout comme une étude de faisabilité, bien qu'elle soit fonctionnelle. Ses défauts majeurs sont :

- la lenteur des compilations en arrière plan ;
- la redondance d'information.

Le premier point est à relativiser : un langage interprété serait globalement plus lent mais permettrait une modification dynamique beaucoup plus facilement. Dans cette implémentation, une fois la phase de prototypage terminée, l'application est plus rapide puisqu'elle est compilée. Le second point est plus dérangeant : chaque instance du type DynamicComponent comprend son code sous forme de chaîne de caractères, alors que l'équivalent est présent soit dans l'espace de représentation de Prolog, soit dans l'espace mémoire d'ARéVi sous forme compilée.

Si les aspects techniques exposés dans ce chapitre peuvent paraître complexes, ils ne faut pas perdre de vue que cette complexité demeure interne au modèle d'exécution. Tout a été fait pour offrir en dernier lieu une forte expressivité aux différents intervenants avec simplicité, la totalité des fonctionnalités du modèle d'agents intentionnels étant mise en œuvre par cette implémentation.

Toutefois, si les mécanismes existent, il manque à ce stade un outil pour exploiter ce modèle : le code à écrire est en partie en C++ et en partie en Prolog, et la définition complète d'un agent est répartie en une suite de composantes. Un simple éditeur ne peut suffire : les capacités de modification dynamique de la quasi totalité des composantes du modèle implique de mêler l'édition de la définition des agents intentionnels avec la simulation de leurs comportements. Il est donc indispensable de concevoir un outil dédié à ce modèle pour en tirer parti.

Cette constatation nous à logiquement incités à développer un tel outil : le chapitre suivant présente l'application Smain (Système Multi-Agents INtentionnels).

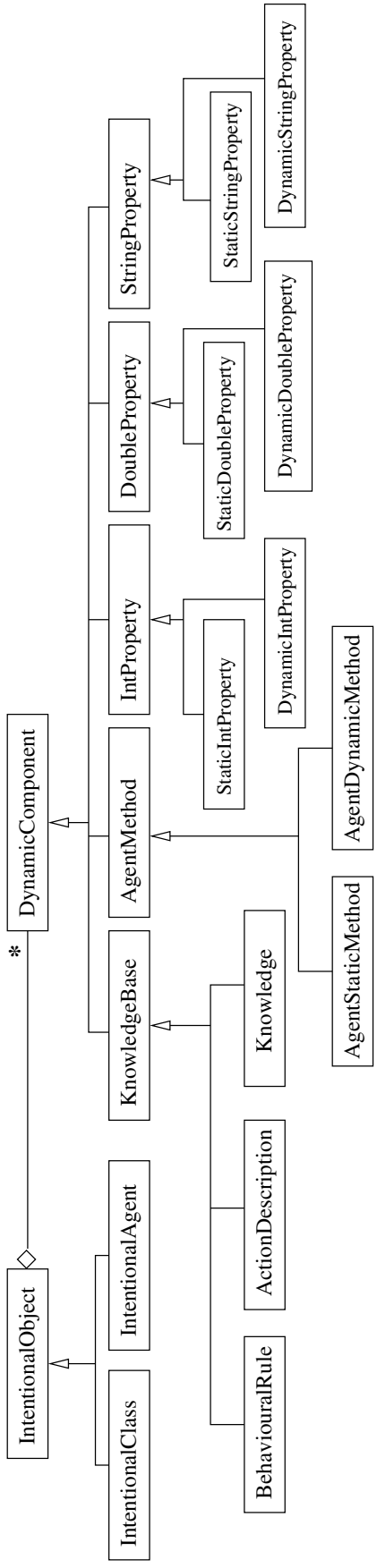

FIG. 5.19 – L'architecture finale : diagramme de classe.

Chapitre 6

Application : SMAIN

> L'homme qui ne tente rien ne se trompe qu'une fois.
>
> LAO-TSEU

Ce chapitre présente l'application que nous avons développée pour exploiter l'architecture décrite au chapitre précédent. Il s'agit d'un outil dédié qui ne permet de définir que des agents intentionnels et qui prend en charge la simulation résultant de l'exécution de leur comportement. Il se nomme `Smain`, ce qui est l'acronyme de SYSTÈME MULTI-AGENTS INTENTIONNELS.
Nous décrivons la structure de l'application dans un premier temps, puis nous résumons les capacités de l'outil. Le fonctionnement de l'ensemble est illustré par l'exemple du prédateur qui nous a jusqu'à présent servi de fil conducteur pour nos explications.

6.1 Architecture logicielle

Smain est une application écrite en C++. Elle est composée d'un ensemble de classes ARéVi qui implémente une interface graphique permettant la pleine exploitation de la librairie dédiée aux agents intentionnels. Le contrôle d'exécution global est donc assuré par l'ordonnanceur d'ARéVi, le traitement des interactions de l'utilisateur avec l'interface étant effectué au sein d'activités ARéVi. Ainsi, il est possible de définir aisément des moyens d'interaction entre la simulation et l'utilisateur final : l'interface graphique ne perturbe pas l'ordonnancement et la gestion des événements. Au final, une fois l'éventuelle phase de prototypage interactif terminée, l'application obtenue est une application ARéVi à part entière : elle est totalement indépendante de Smain. Cet outil s'utilise donc comme un éditeur d'agents intentionnels.

Nous rappelons que la librairie dynamique dédiée aux agents intentionnels est composée d'un ensemble de classes ARéVi et du moteur de SWI-Prolog enrichi par quelques extensions assurant la communication entre les deux langages. Il n'est donc pas nécessaire d'installer une version modifiée de SWI-Prolog, il peut même être absent du système une fois la librairie construite. Ainsi, Smain ne dépend que d'ARéVi et de la librairie libIntentionalAgent. Evidemment, il possède aussi des dépendances vis-à-vis du système (librairie graphique par exemple).

Le fonctionnement de Smain repose sur l'utilisation de trois répertoires de fichiers particuliers :

- le répertoire de sauvegarde des définitions de classes et d'agents au format XML (par défaut ./xml);
- le répertoire dédié à la génération dynamique de code (par défaut ./dynamicSrc);
- le répertoire dédié au stockage des librairies dynamiques créées à la volée (par défaut ./dynamicBin).

Seul le répertoire de stockage XML est utile pour garder la trace des définitions : les fichiers sources et les librairies partagées sont générés à partir des fichiers de définition XML. Toutefois, le fait de conserver les fichiers objets entre deux sessions Smain permet de gagner du temps : les fichiers de définition sont datés et les fichiers sources et objets ne sont générés que s'ils n'existent pas, ou s'ils sont antérieurs à la définition. Ainsi, si une définition n'est pas modifiée entre deux sessions l'utilisateur économise le temps de compilation lors du chargement d'une classe ou d'une instance.
Un exemple de fichier de définition en XML (celui de la classe Predator) est joint en annexe (*cf.* B.5 page 194).

6.2 Les services offerts

Le but de Smain est d'offrir une interface complète avec la librairie développée afin d'exploiter pleinement le modèle proposé. Il doit donc reprendre la totalité des services offerts par les classes (interface de IntentionalClass) et par les agents (interface de IntentionalAgent). Nous allons donc rappeler les différents point d'entrée de la librairie.

6.2.1 Les services communs aux classes et aux instances

Les services communs aux classes et aux instances sont définis dans la classe `IntentionalObject`. L'interface complète est détaillée dans l'annexe B.2 page 184. En voici le résumé :

- modification du nom et de la classe parente de l'objet ;
- évaluation de code `C++` libre dans le contexte de l'objet ;
- évaluation de code `Prolog` libre dans le contexte de l'objet ;
- déclaration et retrait de connaissances ;
- déclaration et retrait de règles comportementales ;
- déclaration et retrait de descriptions d'action ;
- déclaration et retrait de perceptions ;
- déclaration et retrait de propriétés (de type `int`, `double` ou `string`) ;
- déclaration et retrait de méthodes ;
- mise à jour de la partie "statique" de l'objet ;
- mise à jour de la partie "dynamique" de l'objet ;
- création d'une nouvelle classe à partir de la définition courante de l'objet ;
- sauvegarde et chargement de la définition complète de l'objet (format `XML`).

L'outil doit donc permettre d'accéder à la totalité de ces services, aussi bien pour les classes que pour les instances.
De plus, les composantes peuvent toutes être déclarées "statiques" ou "dynamiques", la modification de ce statut pouvant être faite à tout moment. La modification d'une composante se fait par une combinaison retrait/déclaration (ceci assure la cohérence de l'espace de représentation `Prolog`).

Nous verrons dans la suite de ce chapitre la manière dont `Smain` permet de réaliser toutes ces opérations.

6.2.2 Les services de classes d'agents intentionnels

Parmi les services spécifiques aux classes, seule une partie concerne l'utilisateur final. En effet, certains mécanismes sont mis en place au sein des classes pour la gestion du dispositif de chargement à la volée (mise à jour de la table des symboles, déclaration de composantes depuis une librairie externe…). Pour la clarté de l'exposé du chapitre précédent ces aspects de pure ingénierie n'ont pas été détaillés. L'interface complète de la classe `IntentionalClass` est toutefois reportée en annexe (*cf.* B.3 page 188).

Voici les services auquel l'utilisateur doit pouvoir accéder :

- déclaration et consultation d'un éventuel préambule au fichier d'en-tête (inclusion de fichiers d'en-tête de librairies utilitaires) ;
- déclaration et consultation des attributs de la classe ;
- déclaration et consultation d'un éventuel préambule au fichier d'implémentation (comme précédemment pour la partie implémentation) ;
- déclaration et consultation du code de constructeur ;
- déclaration et consultation du code de destructeur.

Les services particuliers aux classes sont donc tous à destination de l'informaticien : il s'agit de la manipulation de détails d'implémentation spécifique à `ARéVi`.

Ces services doivent donc être accessibles *via* `Smain`.

6.2.3 Les services d'instances d'agents intentionnels

Les instances d'agents intentionnels ne possèdent que quelques services supplémentaires dédiés à leur exécution. L'interface complète de la classe `IntentionalAgent` est elle-aussi incluse dans l'annexe B.4 page 191.

Ces services sont :

- exécution d'une action désignée par son nom (*via* le mécanisme de *lookup*) ;
- modification de la fréquence de l'activité *perception* ;
- modification de la fréquence de l'activité *décision* ;
- modification de la fréquence de l'activité *action*.

6.3 La répartition de l'information

D'après le bilan des services offerts dressé dans la section précédente, il apparaît que la grande majorité des fonctionnalités concerne autant les classes que les instances. La réalisation d'une interface relativement claire est donc envisageable. La principale difficulté tient à la répartition de l'information : le modèle impose une séparation de la définition d'une classe ou d'une instance en de nombreuses composantes. Il faut donc offrir un moyen de parcourir les différentes composantes qui reste lisible. De plus, bien que certaines parties soient écrites en `C++` et d'autres en `Prolog`, le but du modèle proposé est de considérer les différentes composantes de manière uniforme. En effet, grâce au mécanisme de prototypage interactif mis en place pour la partie impérative, toutes les composantes sont définies de manières déclarative par l'utilisateur : il peut à loisir consulter et modifier le code impératif. L'interface doit donc contribuer à abroger la séparation entre ces différentes parties en proposant un mécanisme d'édition relativement similaire.
Cependant, certaines composantes comme les règles comportementales et les descriptions d'action doivent être saisies selon une forme très stricte.

Pour répondre au mieux à ces objectifs, l'interface de `Smain` est divisée en trois parties :

- la partie supérieure est composée de trois explorateurs : un explorateur de classes, un explorateur des instances de la classe sélectionnée et un explorateur des composantes de cette instance ;
- la partie médiane est un éditeur contextuel : cette partie de l'interface s'adapte à l'objet examiné (classe, instance ou composante) pour offrir une interface avec les services spécifiques ;
- la partie inférieure est une fenêtre d'information : les messages d'erreurs ou d'information défilent dans une zone dédiée.

Cette organisation générale est représentée en figure 6.1 page ci-contre.

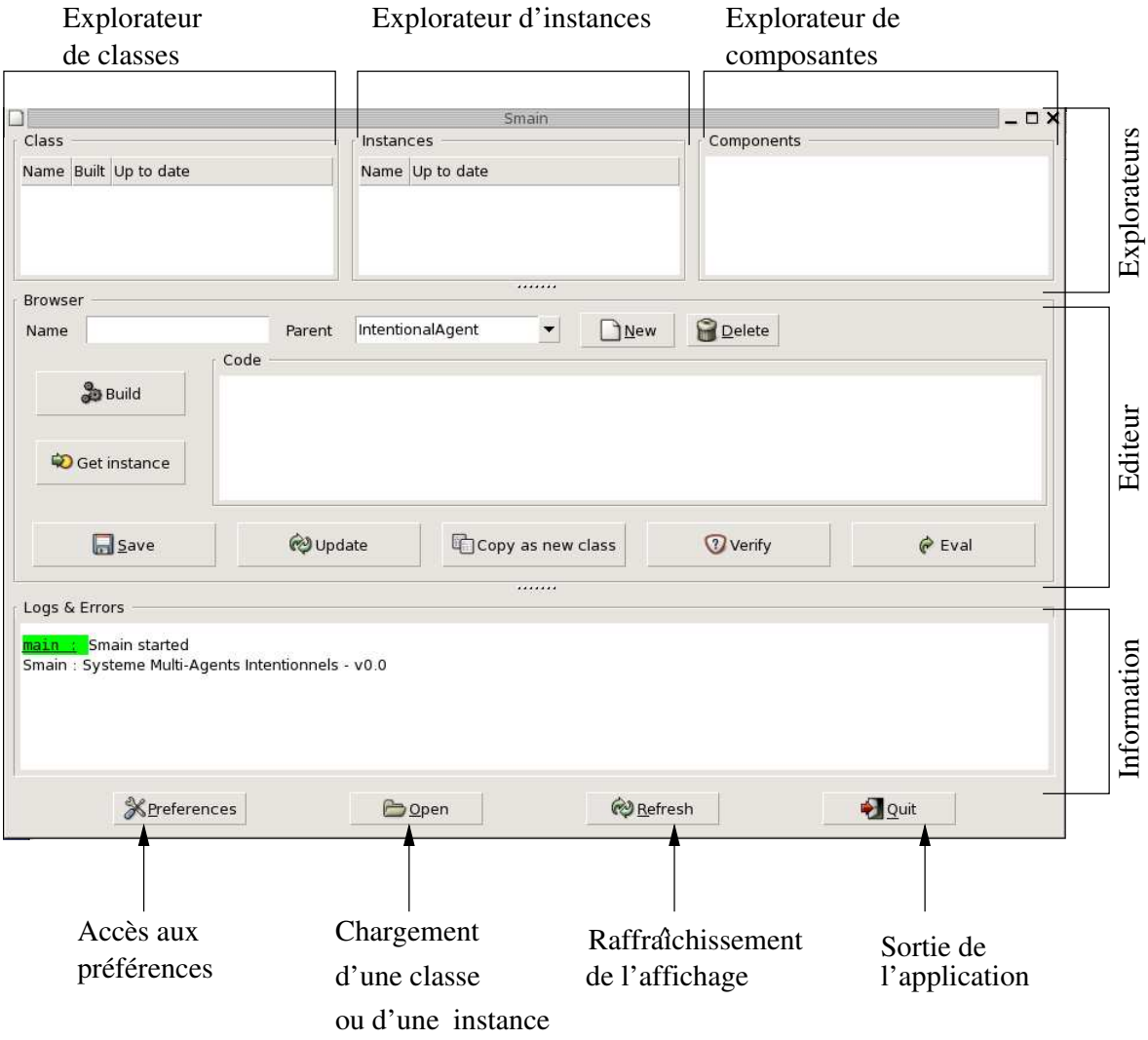

FIG. 6.1 – Smain : vue générale de l'interface.

6.3.1 Les explorateurs

Lorsqu'au moins une classe est chargée, les explorateurs de classe et d'instance permettent d'explorer l'arborescence formée. L'explorateur de composantes affiche alors les composantes de l'objet sélectionné. Il comprend donc toujours :

- les connaissances ;
- les descriptions d'action ;
- les règles comportementales ;
- les perceptions ;
- les propriétés ;
- les méthodes.

Si l'objet sélectionné est une classe, il s'enrichit alors des entrées suivantes :

– préambule du fichier d'en-tête ;
– attributs ;
– préambule du fichier d'implémentation ;
– code du constructeur ;
– code du destructeur.

La figure 6.2 montre la partie supérieure de l'interface lorsqu'une instance est sélectionnée. Dans cet exemple, l'instance `Predator_1` n'est pas spécialisée : aucune composante n'est surdéfinie par rapport à la classe `Predator`. Les différentes catégories de composantes sont donc vides. Cette figure montre aussi la présence d'indicateurs de mise à jour : la case à cocher "*built*" indique que la classe a été compilée, alors que la case à cocher "*up to date*" indique que la partie "dynamique" du code est chargée. Dans cet exemple, il est donc possible que `Predator_1` surdéfinisse une partie de ses composantes mais que ces définitions soient déclarées "dynamiques" et que la mise à jour n'a pas encore été faite. En effet, pour éviter les compilations intempestives à chaque modification de code, la construction des parties "statiques" et "dynamiques" du code se fait à la demande de l'utilisateur.

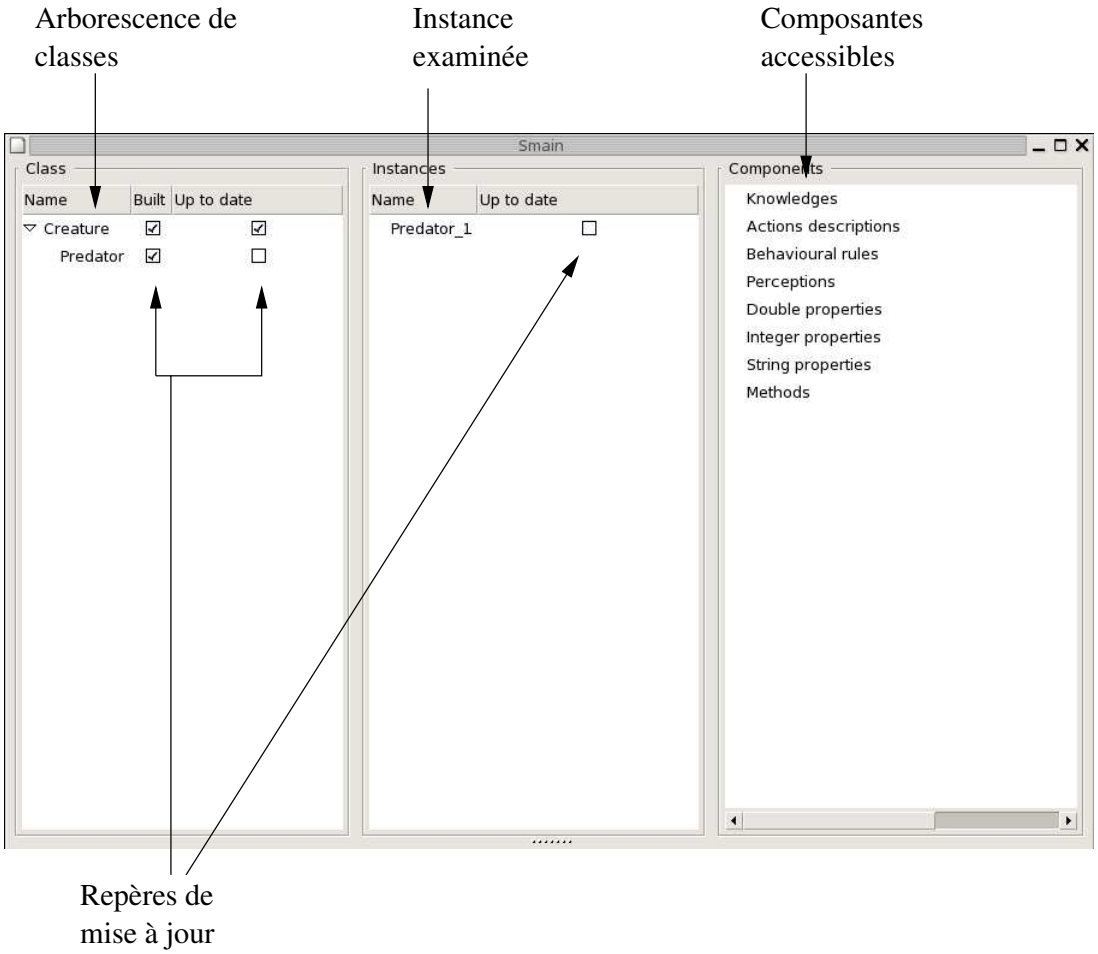

FIG. 6.2 – Explorateur de composantes : examen d'une instance.

La figure 6.3 pour sa part montre la même partie de l'interface lorsqu'une classe est sélectionnée. Différentes composantes apparaissent ici dans l'explorateur.

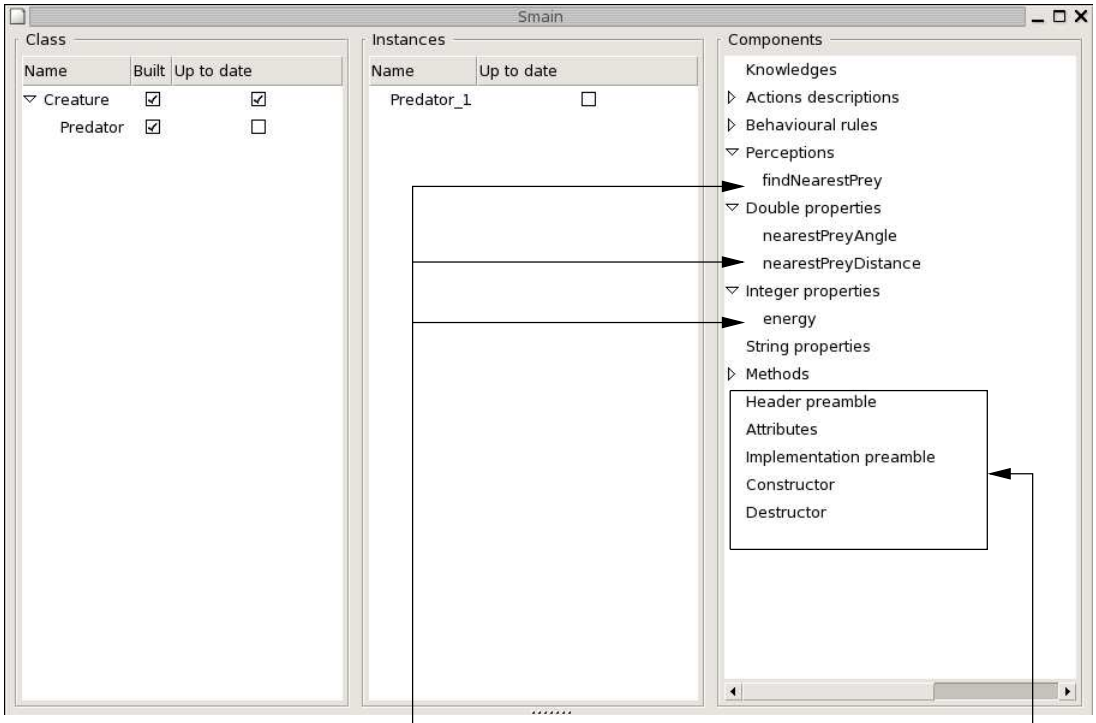

FIG. 6.3 – Explorateur de composantes : examen d'une classe.

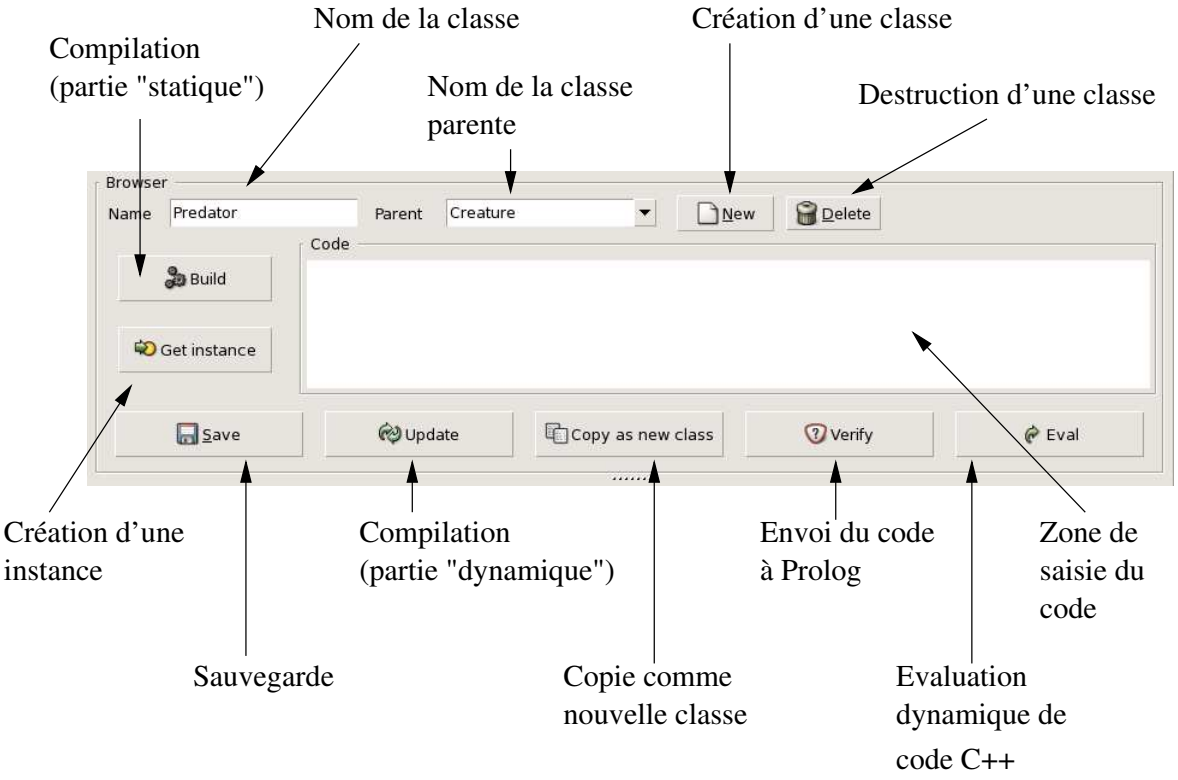

FIG. 6.4 – L'éditeur de classe.

6.3.2 L'éditeur

La partie médiane de l'interface s'adapte au contexte : elle permet l'accès aux services spécifiques à l'objet sélectionné. Nous allons passer en revue les différentes formes qu'elle peut prendre.

L'éditeur de classe

L'éditeur permet la création d'une nouvelle classe après avoir saisi son nom et sa classe parente (`IntentionalAgent` par défaut). La modification dynamique de la classe parente est aussi possible. Ceci peut être utile lorsqu'une entité spécialisée est transformée en nouvelle classe. Un choix aberrant à ce niveau rend la classe inutilisable évidemment (perte des attributs des classes parentes par exemple).
Une classe peut aussi être détruite au moyen de l'éditeur. Sur la gauche de la zone de saisie du code se trouvent les boutons spécifiques aux classes : compilation de la partie "statique" et création d'une instance.
En dessous-de la zone de saisie du code se trouvent les boutons communs aux classes et aux instances : sauvegarde, mise à jour (compilation et chargement de la partie "dynamique"), copie comme nouvelle classe et évaluation de code. Il est à noter que la zone de saisie de code est commune : l'utilisateur emploie soit le bouton *verify* soit le bouton *eval* selon la nature du code saisi (respectivement `Prolog` ou `C++`). La figure 6.4 représente cet éditeur.

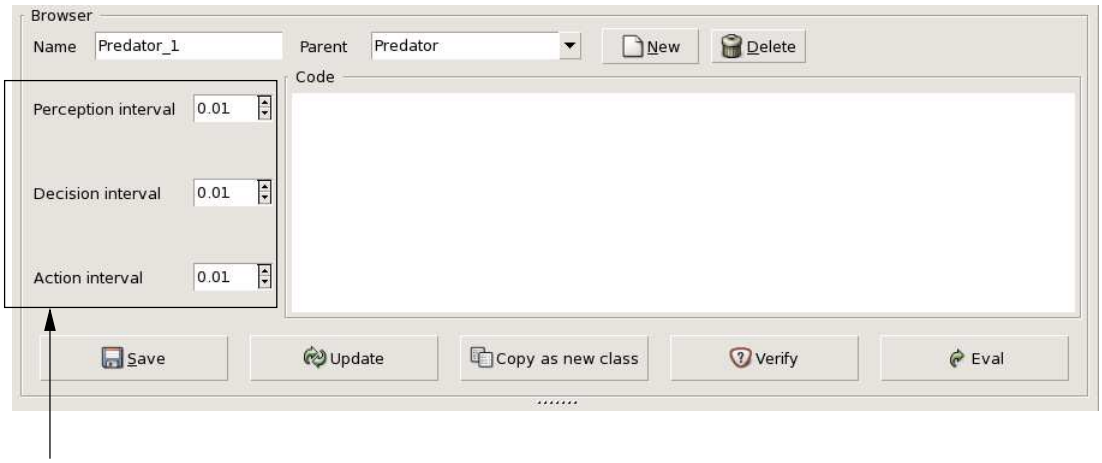

FIG. 6.5 – L'éditeur d'instance.

L'éditeur d'instance

L'éditeur d'instance est très semblable à l'éditeur de classe. Les boutons *Build* et *Get instance* ont évidemment disparu au profit de boîtes de saisie permettant d'ajuster les fréquences des trois activités du modèle (perception, décision et action). La figure 6.5 représente cet éditeur.

L'éditeur de connaissance

Chaque composante du modèle possède un éditeur dédié. Celui qui est consacré à la saisie de connaissance est très simple : il se résume à une zone de saisie de code dans laquelle l'utilisateur peut indiquer un ou plusieurs faits ou prédicats. L'éditeur possède en plus une case à cocher *Built in* qui permet d'indiquer si la connaissance doit appartenir à l'espace de définition "dynamique" ou "statique". Cela peut sembler inutile pour les composantes stockées dans l'espace de représentation de `Prolog`, mais cela intervient lors de certaines opérations. Les composantes déclarées dynamiques ne sont prises en compte que suite à une mise à jour de l'objet (bouton *update*). Ceci est aussi vrai pour les composantes écrites en `Prolog` : la déclaration des composantes "statiques" est effectuée lors du chargement de la classe, alors que la déclaration des composantes "dynamiques" est faite lors de la mise à jour. De plus, lorsqu'une nouvelle classe est créée à partir d'un objet spécialisé, seules les composantes "statiques" de ce-dernier sont importées. Les composantes "dynamiques" sont toujours considérées comme non définitives : elles relèvent de la phase de prototypage.

La figure 6.6 page suivante représente l'éditeur de connaissance.

L'éditeur de règle comportementale

Les règles comportementales doivent avoir une forme très précise pour être exploitées par le processus de sélection d'action. L'éditeur correspondant fournit donc une interface qui génère la règle sous la bonne forme. Celle-ci se compose d'une zone de saisie dans laquelle l'utilisateur peut exprimer les conditions de validité de la règle. Le ou les triplets {*tendance-propriété-force*} correspondant sont exprimés à l'aide de menus déroulants. Ce système contraint l'utilisateur à entrer une forme de règle

FIG. 6.6 – L'éditeur de connaissance.

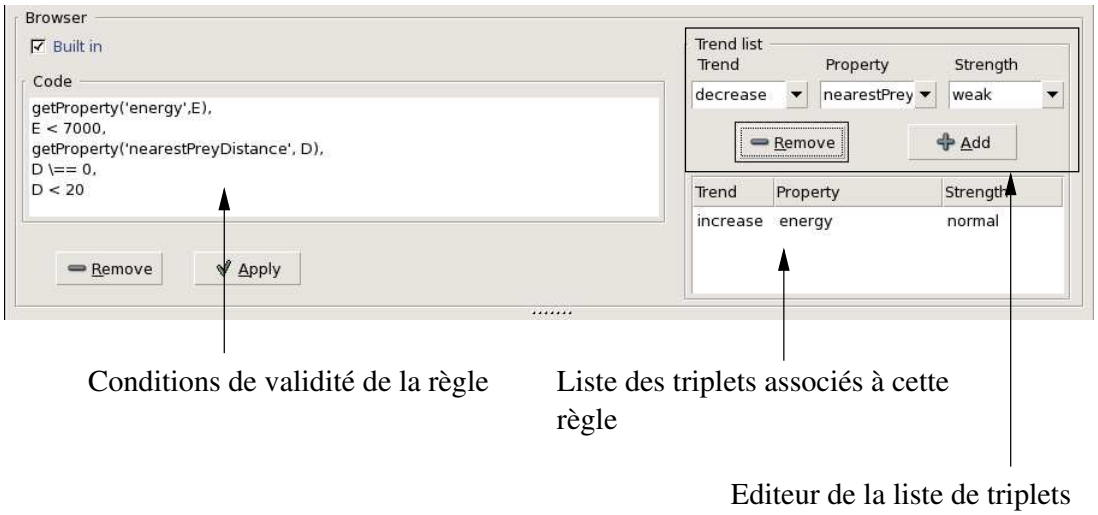

FIG. 6.7 – L'éditeur de règle comportementale.

valide tout en lui facilitant la saisie. La figure 6.7 représente cet éditeur lors de l'édition d'une règle comportementale du prédateur.

L'éditeur de description d'action

Comme les règles comportementales, les descriptions d'action doivent respecter une forme précise. L'éditeur dédié est donc basé sur un principe très similaire à celui des règles comportementales. La figure 6.8 page suivante illustre la saisie de la description de l'action de pourchasse du prédateur.

L'éditeur de méthode

Une méthode est entièrement définie par son nom et le code de son implémentation. L'éditeur de méthode comprend donc simplement une zone de saisie pour le nom de la méthode et une autre pour le code. Dans cette dernière, l'utilisateur doit respecter les conventions imposées par le mécanisme de prototypage interactif : tous les attributs sont adressés *via* l'auto-pointeur `self`. De plus, les actions ne peuvent pas être invoquées autrement qu'au moyen de la méthode *perform* qui assure le *lookup*. Enfin, le code étant explicité au sein du modèle sous forme de chaînes de caractères, il était plus

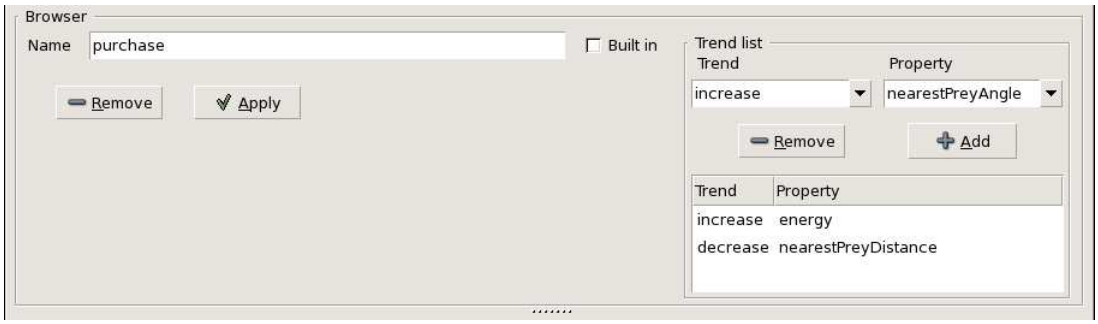

FIG. 6.8 – L'éditeur de description d'action.

FIG. 6.9 – L'éditeur de méthode.

commode de changer de délimiteur par rapport au `C++` : le code saisie dans `Smain` doit utiliser l'apostrophe simple (*quote* : `'`) en lieu et place du guillemet habituel (*double quote* : `"`). Ces trois restrictions mises à part, le code saisi est du `C++` et la bibliothèque `ARéVi` est pleinement exploitable. La figure 6.9 illustre la saisie du code de la méthode *turnLeft* de la classe `Creature`.

L'éditeur de propriété

Une propriété est une méthode particulière, l'éditeur correspondant est donc très semblable à l'éditeur de méthode. Nous avons toutefois ajouter une fonctionnalité : il est possible d'évaluer la valeur d'une propriété à tout moment. Ceci s'avère particulièrement pratique pour le prototypage interactif. Un tel dispositif n'est pas nécessaire pour les méthodes. En effet, il suffit de saisir un code de la forme :

```
self->perform('une_methode') ;
```

Ensuite, la méthode est invoquée à chaque appui sur la touche *eval* de l'éditeur d'instance (ou de classe). Pour les perceptions, l'existence d'une valeur de retour (contrairement aux méthodes dans notre modèle) rendait la tâche plus délicate. Le bouton d'évaluation a donc été ajouté à l'éditeur de propriété pour pallier cette difficulté. La figure 6.10 page suivante illustre la définition de la propriété `nearestPreyDistance` du prédateur.

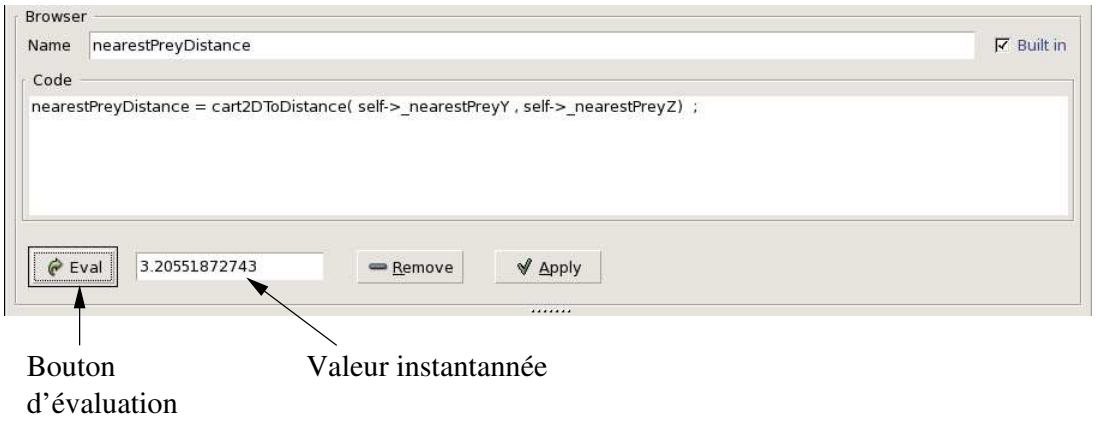

FIG. 6.10 – L'éditeur de propriété.

FIG. 6.11 – Les autres éditeurs : l'exemple de l'éditeur d'attribut.

Les autres éditeurs

Les autres composantes (préambules, attributs, code du constructeur...) nécessitent des éditeurs extrêmement simples (une zone de saisie suffit). La figure 6.11 montre l'exemple de la saisie des attributs de la classe `Predator`. Certains attributs ont disparu par rapport à l'explication du chapitre précédent : l'existence de la classe `Creature`, parente de la classe `Predator`, avait été omise alors pour simplifier l'exposé. Une partie des attributs est donc héritée.

6.3.3 La fenêtre d'information

La partie inférieure de `Smain` est une zone de défilement où sont affichés les messages (*logs*) et les erreurs. Du fait du prototypage interactif, de nombreuses opérations sont susceptibles d'échouer : invocation d'une méthode inexistante, compilation d'une classe dont la définition est erronée...
Les différents cas d'erreur sont interceptés par la librairie `libIntentionalAgent` et donnent lieu à l'émission de messages sans pour autant conduire à l'interruption de la simulation. Le rôle de cette fenêtre est de communiquer toutes ces informations à l'utilisateur. La figure 6.12 page suivante montre l'état de cette fenêtre après le chargement et la compilation de la classe `Creature`, le message d'erreur correspond à la tentative de charger un fichier `XML` non valide. La quantité d'informations pouvant être assez importante, les messages sont hiérarchisés et les messages et erreurs peuvent être filtrés par niveau. Ce réglage est disponible dans la fenêtre des préférences qui permet

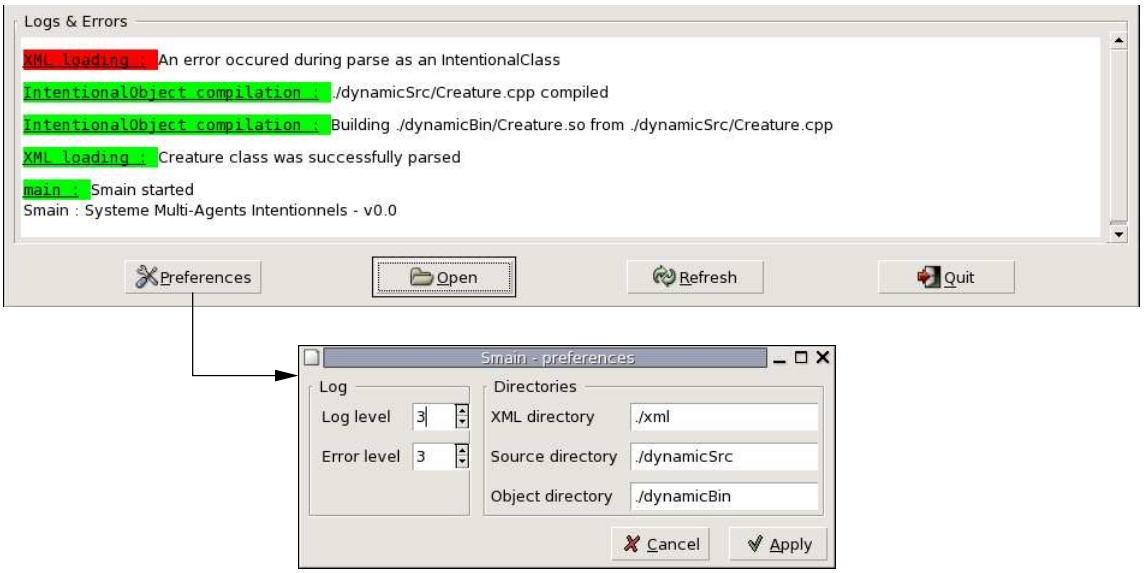

FIG. 6.12 – La fenêtre d'information et le réglage des préférences.

aussi de modifier les répertoires exploités par l'application.

6.4 Utilisation

L'application `Smain` telle qu'elle a été présentée permet d'exploiter totalement le modèle d'agent intentionnel.

Les différentes composantes du modèle sont clairement séparées et il et possible d'utiliser cet outil à différents niveaux :

- l'informaticien peut utiliser `Smain` pour définir toute la partie spécifique à `ARéVi`, il n'a besoin de recourir à aucun outil externe ;
- le spécificateur peut modifier à loisir les règles comportementales et les descriptions d'action d'une classe ou d'une instance en particulier à tout moment ;
- un simple utilisateur peut se contenter de charger des définitions de classes et de créer des instances pour démarrer une simulation.

Evidemment, l'évaluation de l'ergonomie est purement qualitative. Depuis la proposition du modèle d'agent intentionnel jusqu'à l'implémentation de `Smain`, le but que nous poursuivions était la mise en place d'un fort dynamisme et de moyens d'interaction expressifs avec les intervenants du cycle de spécification. L'aspect dynamique étant intrinsèque, il est délicat d'illustrer le fonctionnement de l'ensemble : `Smain` a pour intérêt la possibilité de modifier les comportements d'agents en cours de simulation. La présentation d'une suite de saisie d'écrans ou de traces d'exécution serait ici fastidieuse. Nous ne pouvons donc démontrer la souplesse d'utilisation de l'application, mais l'exposé détaillé que nous avons fait du modèle sous-jacent permet de cerner l'étendue des possibilités de celle-ci.

Lors de la présentation du modèle d'agent intentionnel, nous avons défendu l'opinion selon la-

quelle l'introduction d'une indirection sémantique entre la partie décisionnelle et la gestion des actions augmente l'expressivité. Les perspectives d'applications annoncées étaient, entre autres, la mise en place de mécanismes d'apprentissage ou la génération d'explications. Pour illustrer ce propos, nous allons donner l'exemple d'une génération d'explication en langage naturel du comportement de prédation.

La génération d'explication n'est pas aisée à mettre en œuvre avec la plupart des architectures. Il ne s'agit pas d'expliquer l'action commise, mais le lien entre la décision et l'action : pourquoi l'action est commise. Dans notre modèle, ce lien est explicité par l'intermédiaire des règles comportementales. Il n'est donc pas nécessaire de modifier le code implémentant les actions, ni le processus de sélection de ces dernières : la seule exploitation d'une règle comportementale suffit à faire ressortir au niveau utilisateur des éléments sémantiques suffisamment explicites pour générer une explication en langage naturel.

Considérons la règle qui régit le comportement de prédation telle qu'elle a été présentée jusqu'ici (figure 6.13).

```
behaviouralRule([energy-increase-normal]) :-
    getProperty('energy',E),
    E < 7000,
    getProperty('nearestPreyDistance', D),
    D =\= 0,
    D < 20.
```

FIG. 6.13 – La règle comportementale de prédation initiale.

Cette règle se lit comme suit :
L'agent cherche à augmenter son énergie si :

- son énergie est faible ;
- il a détecté une proie ;
- cette proie n'est pas trop loin.

Elle contient donc la sémantique suffisante à une génération d'explication. Il suffit de modifier légèrement la règle pour générer des phrases. Celles-ci peuvent alors être stockées dans un fichier par exemple, pour la démonstration nous nous contenterons de les afficher dans la console. La figure 6.14 page suivante montre la modification de la règle comportementale à l'aide du prédicat `write`.

Avec un peu d'apprentissage de la syntaxe, ce genre de modification peut être accessible à un intervenant qui n'est pas expert des systèmes informatiques. La modification se résume à afficher une phrase différente pour chaque alternative. Aucune intervention n'est nécessaire dans le code implémentant les actions correspondantes. Cette version de la règle permet d'obtenir des explications simples du comportement du prédateur telles que celle proposée par la figure 6.15 page ci-contre.

L'utilisation de `Prolog` apporte l'expressivité qui était recherchée. La possibilité de déclarer des prédicats ou des faits dynamiquement permet de mettre en place des mécanismes beaucoup plus riches, sans avoir à modifier le fonctionnement de base. A l'instar de `Prolog` qui sépare le moteur d'inférence de la base de faits, notre application sépare complètement le mécanisme de sélection d'action de la définition du comportement. `Smain` offre donc un moteur de simulation à part

```
behaviouralRule([energy-increase-normal]) :-
        getProperty(energy, A),nl,
        (   A<7000 -> ,write('je suis fatigue')
                      ,
                       write('je suis en forme, inutile de chasser'),
                       nl,
                       fail
        ),
        getProperty(nearestPreyDistance, B),
        (   B==0 -> write('mais aucune proie a l horizon '),
                    nl,
                   ;fail
                   ,
                    write('et je vois une proie a '),
                    write(B),
                    write(' metres'),
                    nl
        ),
        (   B<20 -> write('c'est assez pres pour vouloir chasser'),
                    nl
                   ;
                   ,write('c est trop loin...'),
                    nl,
                    fail
        ).
```

FIG. 6.14 – La règle modifiée pour générer une explication.

```
je suis en forme, inutile de chasser
je suis en forme, inutile de chasser
je suis en forme, inutile de chasser
je suis fatigue mais aucune proie a l horizon
je suis fatigue mais aucune proie a l horizon
je suis fatigue et je vois une proie a 4.03219 metres
c'est assez pres pour vouloir chasser
je suis fatigue et je vois une proie a 3.93336 metres
c'est assez pres pour vouloir chasser
je suis fatigue et je vois une proie a 32.75633 metres
c est trop loin...
```

FIG. 6.15 – L'explication générée.

entière : l'utilisateur ne se soucie que de la spécification du comportement de manière déclarative, l'odonnancement global est assuré par l'application. Il peut donc ignorer complètement les détails du modèle d'exécution : tout comportement spécifié *via* `Smain` est nécessairement conforme au modèle sous-jacent.

6.5 Conclusion : un outil en évolution

Le modèle d'exécution présenté au chapitre précédent offre des services permettant la pleine exploitation du modèle d'agent intentionnel. Son utilisation nécessitait toutefois la création d'un outil dédié, le simple fait de devoir employer deux langages différents pour définir un agent justifiant une telle démarche. `Smain` reprend la totalité de ces services et propose une interface graphique relativement claire. A l'usage, la librairie `libIntentionalAgent` s'avère robuste, la majorité des erreurs étant interceptées. Les fausses manipulations amènent la plupart du temps l'utilisateur à modifier son code ou, dans les cas défavorables, à détruire certaines instances en rechargeant leur définition.

La première prise en main est relativement déroutante : le fait de pouvoir modifier radicalement l'implémentation de certaines méthodes alors que l'agent concerné est impliqué dans une simulation en cours offre de grandes libertés de prototypage. Et, comme toute liberté nouvelle, trouver le bon

moyen d'employer cette expressivité demande un peu de temps. L'ergonomie de `Smain` peut être améliorée sur plusieurs points. Nous considérons cette application comme la première étape d'un développement en spirale : l'aspect intrinséquement dynamique du modèle d'agent intentionnel le rend impossible à exploiter sans un outil qui soit lui-même dynamique. `Smain` constitue ainsi avant tout un moyen d'explorer les possibilités du modèle proposé. Au fur et à mesure des utilisations, le modèle sera affiné et, par effet de bord, `Smain` devra lui-même être modifié pour prendre en compte cette évolution. Le but de cette circularité est de permettre la maturation progressive du modèle afin de jeter les bases d'un nouveau langage. En effet, les difficultés d'utilisation de `Smain` proviennent essentiellement du mélange des langages. De plus, la bibliothèque `ARéVi` présente des spécificités qu'il faut assimiler pour exploiter toutes les possibilités de l'outil. La solution à long terme serait donc la création d'un langage dont la sémantique repose sur des concepts du même ordre que ceux manipulés dans le modèle d'agent intentionnel, et dont le modèle d'exécution soit proche de celui que nous avons développé.

Avant de débuter une telle entreprise, il faut stabiliser l'ensemble des concepts et leur sémantique. La prochaine étape consiste donc à multiplier les applications, et surtout les intervenants autour de `Smain` : un point fondamental est la prise en compte des difficultés à spécifier un comportement que peuvent rencontrer des personnes qui ignorent les détails techniques du modèle d'exécution.

L'outil que nous avons développé débute donc, nous l'espérons, une lente évolution vers la conception d'une application stable, d'un accès aisé et permettant une grande expressivité dans la spécification de comportements.

Chapitre 7

Conclusion et perspectives

I have not failed, I have merely found 10,000 ways it won’t work

THOMAS EDISON

7.1 Conclusion

L'étude que nous avons présentée consitue une contribution à la modélisation de comportements d'agents autonomes situés dans un environnement virtuel. Nous avons souligné l'aspect intrinséquement pluridiciplinaire de cette activité et les nombreuses ambiguïtés rencontrées dans son abord (chapitre 1) : une traduction maladroite ou l'emprunt à un autre domaine d'un terme fortement connoté sans précaution suffit à introduire des approximations néfastes. Un vocabulaire imprécis engendre des raisonnements erronés, et la dérive dans la reflexion menée se révèle souvent suffisamment incidieuse pour ne se révéler que lors de la confrontation avec les théories des domaines connexes. Nous nous sommes donc employés à clarifier autant que possible les concepts nécessaires à notre étude.

La présentation de notre cadre d'étude nous a conduits à préciser les notions d'agent, de spécification de comportements, d'autonomie et de situation dans un environnement virtuel (chapitre 2). La spécification de comportements a été décomposée en quatre étapes qui se prètent toutes, de manière différente, à une application des principes du paradigme multi-agents :

- pour l'analyse par le thématicien l'agent est une unité conceptuelle ;
- pour la conception par le modélisateur l'agent est une unité computationnelle ;
- pour l'implémentation par l'informaticien l'agent est une unité d'implémentation ;
- pour le déploiement par l'utilisateur final l'agent est une unité d'exécution.

Pour les simulations interactives et participatives, notre domaine applicatif, nous avons mis en évidence des objectifs *a priori* antithétiques : la volonté de produire un rendu en temps réel et de contrôler les agents impliqués dans la simulation semble difficilement conciliable avec la volonté de maximiser l'autonomie et la crédibilité des modèles comportementaux utilisés. Ces objectifs sont d'autant plus problématiques qu'ils paraissent totalement subjectifs : les notions d'autonomie et de crédibilité sont affaire de point de vue.

Cette dernière remarque constitue le point nodal de notre réflexion d'ensemble : la spécification de comportements d'agents situés en environnement virtuel relève systématiquement de l'adoption d'un point de vue. Ceci se vérifie à différentes échelles :

- le principe même de la réalité virtuelle est d'offrir un point de vue maîtrisé sur un environnement qui peut éventuellement être impossible dans le réel (environnement inaccessible ou à une échelle différente, représentation métaphorique ou d'objets qui n'existent pas…) ;
- à son niveau, l'agent possède un point de vue sur son environnement et sur lui-même ;
- les agents peuvent se construire un point de vue sur leur situation et sur les attitudes des autres agents *via* leurs interactions ;
- enfin, les différents intervenants impliqués dans l'activité de spécification (thématicien, modélisateur…) élaborent le comportement de l'agent par la confrontation de leurs points de vue respectifs.

Cette approche de la spécification dans son ensemble nous a incité à aborder ces différents points à la lumière de la notion d'intention (chapitre 3 et annexe A) : ce concept traduit la visée, la directionnalité d'une attitude mentale et permet d'aborder les différents points de vue mis en évidence selon une démarche commune.
L'étude transdiciplinaire de l'intentionnalité que nous avons menée repose sur des travaux en philosophie, en psychologie et en neurophysiologie. Bien qu'il s'agisse d'une étude superficielle, la variation de point de vue selon ces trois diciplines nous a permis de dégager des traits de cette notion que nous

avons tenté de retrouver dans notre problématique. Nous avons retenu deux axes de recherche :

- l'exploitation du rapport entre l'intention et l'action pour l'élaboration d'un modèle comportemental ;
- l'abord de la spécification de comportements comme une démarche intentionnelle : le comportement de l'agent est façonné par la variation de point de vue entre les intervenants, il est le champ de leur intersubjectivité.

Le rapport entre l'intention et l'action peut se résumer comme suit :

- l'intention préalable est nécessaire au déclenchement délibéré d'une action ;
- l'intention dans l'action permet le contrôle de l'action en cours d'effectuation, qu'elle ait été déclenchée délibéremment ou par réflexe, et constitue le vecteur de l'agentivité (l'entité se sait agissante autant qu'elle se sent agissante) ;
- l'arrêt d'une action ne peut être qu'intentionnel.

Nous avons fédéré les notions d'action et de perception, une perception étant construite au moyen d'un acte perceptif. L'intention qui sous-tend l'acquisition d'information influence donc cet acte. C'est le principe de perception active.

A la suite de cette étude, nous avons proposé un modèle comportemental (chapitre 4). Notre objectif était d'exploiter la notion d'intention selon ces deux mêmes axes :

- l'intention fonde un mécanisme de sélection d'action basé sur une résolution qualitative ;
- l'apport sémantique à ce modèle réactif constitué par l'explicitation du *sens* de l'action au sein du modèle permet d'offrir un ensemble de concepts communs aux différents intervenants, il est donc le support de leur intersubjectivité.

Pour satisfaire aux contraintes d'efficatité du cadre applicatif, tout en maximisant l'expressivité offerte par l'introduction de ce lien sémantique dans le modèle, nous avons proposé un modèle d'exécution multi-paradigmes (chapitre 5) : un langage impératif implémente les actions de l'agent, alors que les règles régissant son comportement sont spécifiées à l'aide d'un langage déclaratif logique.
Cette dualité amène des contraintes d'implémentation fortes et nous avons détaillé les implications en termes de développement, particulièrement l'importance de l'interruptibilité d'un moteur d'inférence et de l'ordonnancement de tâches hétérogènes.

La distribution de l'information, la mixité de code impératif et déclaratif et la très forte dynamicité du modèle imposaient la création d'une application dédiée. Un des objectifs poursuivis étant le prototypage interactif, l'étape de spécification confond totalement l'étape de simulation : la définition du comportement peut être complètement modifiée alors que l'agent est impliqué dans une simulation en cours. L'application `Smain` a donc été développée sur la base du modèle comportemental et du modèle d'exécution proposés (chapitre 6).

L'application finale exploite pleinement le modèle d'agent intentionnel et sépare clairement les tâches de spécification qui incombent à l'informaticien de celles qui sont de la responsabilité du modélisateur. Malgré cette séparation, des concepts communs forment l'interface entre ces deux intervenants instaurant alors le lien désiré.

Nous avons ainsi montré l'intérêt de la prise en compte de la notion d'intentionnalité dans la spécification de comportements d'agents autonomes situés en environnement virtuel.

7.2 Perspectives

Les réflexions menées au cours de cette étude ouvrent de nombreuses perspectives quant au recours à la notion d'intentionnalité dans la spécification de comportements. Le point le plus délicat, mais aussi le plus attirant, est l'étude de la visée intentionnelle. Un agent qui serait capable de se construire une extériorité, de se considérer lui-même commme projeté au monde, entrelacé avec son environnement, serait alors réellement autonome. Mais cette approche se heurte très rapidement aux problèmes de la représentation et du sens. Si les théories de l'intentionnalité ne fournissent pas immédiatement un modèle, elle nous donnent au moins le vocabulaire et les outils de réflexion nécessaires à l'abord de problèmes qu'elles identifient clairement. En cela, nous pouvons considérer que l'approche intentionnelle constitue un paradigme d'étude à part entière.

Un autre point soulevé est le contrôle en continu de l'action. Nous avons résolu ce problème par des appels répétés de méthodes impératives de granularité faible. Il ne s'agit donc pas réellement de "continuité", le contrôle de l'action est discrétisé. Pour explorer plus avant cette problématique, l'étude de la programmation fonctionnelle réactive peut très certainement apporter des éléments pertinents à notre réflexion.

De manière plus pragmatique, et à plus court terme, nous orientons nos travaux vers l'exploration du modèle intentionnel. Tel qu'il est proposé à la base, ce modèle de sélection d'action est très simple. Mais son intérêt réside dans le soin que nous avons apporté à favoriser l'intégration au cycle de spécification et dans ses très grandes possibilités d'évolution : le modèle d'agent intentionnel constitue pour nous une base d'étude. Notre but est d'explorer les possibilités offertes par l'apport sémantique du modèle, en le faisant évoluer vers des modèles délibératifs (mécanismes d'apprentissage, communication de connaissances entre agents...). Pour cela, il nous faut multiplier les cas d'utilisation de `Smain` : du fait de la dynamicité de l'ensemble, cet outil constitue le meilleur moyen d'exploration du modèle qui le fonde. Nous nous engageons donc dans un développement en spirale au cours duquel `Smain` va nous conduire à enrichir le modèle d'agent intentionnel et réciproquement.
Une autre démarche importante est de confronter des utilisateurs novices à l'utilisation de `Smain` afin d'identifier les éventuels manques du modèle. Le dessein de ces expériences est de stabiliser l'ensemble des concepts nécessaires à la spécification de ce type d'agent et de mettre au point les mécanismes d'exécution correspondant.

A long terme, nous espérons être en mesure de proposer une démarche de spécification d'agent intentionnel conjointement à l'outil support de leur exécution, ce qui constitue les premier pas vers une méthodologie de développement.

Bibliographie

[Arnaud, 2000] Arnaud, P. (2000). *Des moutons et des robots*. Collection META. Presses Polytechniques et Universitaires Romandes.

[Asimov, 1950] Asimov, I. (1950). *Les robots*. Editions J'ai lu, Paris.

[Austin, 1962] Austin, J. L. (1962). *How to Do Things With Words*. Oxford University Press : Oxford, England.

[Aylett and Luck, 2000] Aylett, R. and Luck, M. (2000). Applying artificial intelligence to virtual reality : Intelligent virtual environments. *Applied Artificial Intelligence*, 14(1) :3–32.

[Aït-Kaci, 1991] Aït-Kaci, H. (1991). *Warren's Abstract Machine : A Tutorial Reconstruction*. MIT Press.

[Baral and Gelfond, 1997] Baral, C. and Gelfond, M. (1997). Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31(1-3) :85–117.

[Bauer et al., 2000] Bauer, B., Müller, J., and Odell, J. (2000). Agent UML : A formalism for specifying multiagent interaction. In *Proc. of the 1st Int. Workshop on Agent-Oriented Software Engineering, AOSE'00*, pages 91–104. Ciancarini and Wooldridge.

[Bernon et al., 2004] Bernon, C., Camps, V., Gleizes, M.-P., and Picard, G. (2004). Tools for self-organizing applications engineering. In Serugendo, G. D. M., Karageorgos, A., Rana, O., and (Eds), F. Z., editors, *The First International Workshop on Engineering Self-Organising Applications (ESOA'03)*, volume 2977. LNCS, LNAI, Springer Verlag.

[Berthoz, 1997] Berthoz, A. (1997). *Le sens du mouvement*. editions Odile Jacob.

[Bizzi and al., 2003] Bizzi, E. and al. (2003). *Taking Action, Cognitive Neuroscience Perspectives on Intentional Acts*. Scott H. Johnson editor, A Bradford Book, MIT Press.

[Blumberg and Galyean, 1995] Blumberg, B. and Galyean, T. (1995). Multi-level direction of autonomous creatures for real-time virtual environments. *Computer Graphics*, pages 47–54.

[Boella and van der Torre, 2004] Boella, G. and van der Torre, L. (2004). The social delegation cycle. In *LNAI n.3065, Procs. of EON'04*, pages 29–42.

[Boissier, 2001] Boissier, O. (2001). Modèles et architectures d'agents. *Principes et architectures des systèmes multi-agents*, pages 71–107.

[Booch, 1994] Booch, C. (1994). *Object-Oriented Analysis and Design (second edition)*. Addison-Wesley : Reading.

[Brassac and Pesty, 1996] Brassac, C. and Pesty, S. (1996). La pelouse fourmilière, de la coaction à la coopération. In Quinqueton Muller, editor, *4èmes Journées Francophones sur l'Intelligence Artificielle Distribuée et Systèmes Multi-Agents (JFIADSMA'96)*, pages 250–263. Hermes.

[Bratman, 1987] Bratman, M. E. (1987). *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA.

[Bratman et al., 1991] Bratman, M. E., Israel, D., and Pollack, M. (1991). Plans and resource-bounded practical reasoning. In Cummins, R. and Pollock, J. L., editors, *Philosophy and AI : Essays at the Interface*, pages 1–22. The MIT Press, Cambridge, Massachusetts.

[Brazier et al., 1995] Brazier, F., Keplicz, B. D., Jennings, N. R., and Treur, J. (1995). Formal specification of multi-agent systems : a real-world case. In *First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 25–32, San Francisco, CA, USA. AAAI Press.

[Bringsjord, 1998] Bringsjord (1998). Computationalism is dead ; now what ? response to fetzer's 'minds are not computers : (most) thought processes are not computational procedures'. *JETAI : Journal of Experimental and Theoretical Artificial Intelligence*, 10 :393–402.

[Broersen et al., 2001] Broersen, J., Dastani, M., Hulstijn, J., Huang, Z., and van der Torre, L. (2001). The boid architecture : conflicts between beliefs, obligations, intentions and desires. In *Proceedings of the fifth international conference on Autonomous agents*, pages 9–16. ACM Press.

[Brooks, 1986] Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1) :14–23.

[Busetta et al., 1999] Busetta, P., Ronnquist, R., Hodgson, A., and Lucas, A. (1999). Jack intelligent agents - components for intelligent agents in java. *AgentLink News Letter*, 2 :2–5.

[Caicedo and Thalmann, 2000] Caicedo, A. and Thalmann, D. (2000). Virtual humanoids : Let them be autonomous without losing control. In *Proc. 3IA2000*.

[Cardon, 2000] Cardon, A. (2000). *Conscience artificielle et systèmes adaptatifs*. Editions Eyrolles.

[Casteran et al., 2000] Casteran, J., Gleizes, M., and Glize, P. (2000). Des méthodologies orientées multi-agent. In *Systèmes multi-agents, méthodologies, technologies et expériences, JFIAD-SMA'00*. Hermès.

[ChaibDraa and Millot, 1990] ChaibDraa, B. and Millot, P. (1990). A framework for cooperative work : an approach based on the intentionality. In *Artificial Intelligence in Engineering*, volume 5. Computational Mechanics Publications.

[Chen et al., 2001] Chen, L., Bechkoum, K., and Clapworthy, G. (2001). A logical approach to high-level agent control. In *Proceedings of the fifth international conference on Autonomous agents*, pages 1–8. ACM Press.

[Chicoisne, 2002] Chicoisne, G. (2002). *Dialogue entre agents naturels et agents artificiels, une application aux communautés virtuelles*. PhD thesis, LEIBNIZ/IMAG, Institut National Polytechnique de Grenoble.

[Cicourel, 1987] Cicourel, A. V. (1987). On John R. Searle's intentionality. In *Journal of Pragmatics*, volume 11. Cambridge University Press.

[Cohen and Levesque, 1990] Cohen, P. and Levesque, H. (1990). Intention is choice with commitment. *Artificial Intelligence*, 42(3) :213–261.

[Crutchfield, 1994] Crutchfield, J. P. (1994). Is anything ever new ? - considering emergence. In *Sciences of Complexity*, volume 9. Cowan, Pines and Melzner editors, Santa Fe Institute Studies, Addison-Wesley.

[Dastani et al., 2003] Dastani, M., Hulstijn, J., and van der Torre, L. (2003). How to decide what to do ? *European Journal of Operational Research, In Press, Corrected Proof*.

[Delahaye and Mathieu, 1996] Delahaye, J. P. and Mathieu, P. (1996). The iterated lift dilemma or how to establish meta-cooperation with your opponent ? In *Complexity, Society and Liberty Conference*.

[Demazeau, 1997] Demazeau, Y. (1997). Steps towards multi-agent oriented programming. *In 1st International Workshop on Multi-Agent Systems (IWMAS'97)*.

[Dennett, 1987] Dennett, D. C. (1987). *The intentional stance*. MIT Press.

[Dick, 1968] Dick, P. K. (1968). *Do androïds dream of electric sheep ? (Blade Runner)*. Del Rey Book.

[Dikötter, 1993] Dikötter, M. (1993). *Les accepteurs, Un nouveau modèle de programmation*. Presses Polytechniques et Universitaires Romandes.

[D'Inverno et al., 1997] D'Inverno, M., Kinny, D., Luck, M., and Wooldridge, M. (1997). A formal specification of dMARS. In *Agent Theories, Architectures, and Languages*, pages 155–176.

[D'Inverno and Luck, 2003] D'Inverno, M. and Luck, M. (2003). Unifying agent systems. *Annals of Mathematics and Artificial Intelligence*, 37(1–2) :131–167.

[Donikian, 2004] Donikian, S. (2004). Modélisation, contrôle et animation d'agents virtuels autonomes évoluant dans des environnements informés et structurés. *mémoire d'Habilitation à Diriger des Recherche, Université de Rennes 1*.

[Drogoul et al., 2003] Drogoul, A., Vanbergue, D., and Meurisse, T. (2003). Simulation orientée agent : où sont les agents ? In *Actes des Journées de Rochebrune, Rencontres interdisciplinaires sur les systèmes complexes naturels et artificiels*.

[Drogoul and Zucker, 1998] Drogoul, A. and Zucker, J. (1998). Methodological issues for designing multi-agent systems with machine learning techniques : Capitalizing experiences from the robocup challenge. *Technical Report LIP6 1998/041, Laboratoire d'Informatique de Paris 6*.

[Favier et al., 2001] Favier, P., De Loor, P., and Tisseau, J. (2001). Programming agent with purposes : Application to autonomous shooting in virtual environment. *Lecture Notes In Computer Sciences*, 2197 :40–43.

[Favier, 2003] Favier, P.-A. (2003). Cross-paradigm programming for behavioural specification in multi-agent systems,intentionality : From decision to action. *in First Indian International Conference on Artificial Intelligence Proceedings (IICAI-2003)*, pages 302–314.

[Favier and De Loor, 2003] Favier, P.-A. and De Loor, P. (2003). Intentionality : a needed link between decision and action for intelligent's agent behaviour. *in Forth International Conference on Virtual Reality and its Application in Industry Proceedings (VRAI'2003)*, pages 166–172.

[Favier and De Loor, 2005] Favier, P.-A. and De Loor, P. (2005). From decision to action : intentionality, a guide for the specification of intelligent agents' behaviour. *International Journal of Image and Graphics (IJIG), special issue, to appear*.

[Ferber and Gutknecht, 1998] Ferber, J. and Gutknecht, O. (1998). A meta-model for the analysis and design of organizations in multi-agent systems. *In Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS98)*, pages 128–135.

[Ferguson, 1992] Ferguson, I. (1992). *TouringMachine : An Architecture for dynamic, Rational, Mobile Agents*. PhD thesis, Clare Hall, University of Cambridge, Great Britain.

[Fetzer, 1994] Fetzer, J. (1994). Mental algorithms : Are minds computational systems ? *Pragmatics and Cognition*, 2 :1–29.

[Finin et al., 1994] Finin, T., Fritzson, R., McKay, D., and McEntire, R. (1994). KQML as an Agent Communication Language. In Adam, N., Bhargava, B., and Yesha, Y., editors, *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, MD, USA. ACM Press.

[Fipa, 2001] Fipa (2001). Fipa communicative act library specification.

[Fishwick, 1994] Fishwick, P. (1994). Computer simulation : Growth through extension. In *Proc. European Simulation Multiconference, Modelling and Simulation of Uncertain Dynamics*, pages 3–17. SCS.

[Forbus, 1984] Forbus, K. D. (1984). Qualitative process theory. *Artificial Intelligence*, 24 :85–168.

[Fuchs et al., 2003] Fuchs, P., G. Moreau, B. A., Burkhardt, J., Chauffaut, A., Coquillart, S., Donikian, S., Duval, T., Grosjean, J., Harrouet, F., Klinger, E., Lourdeaux, D., d'Huart, D. M., Paljic, A., Papin, J. P., Stergiopoulos, P., Tisseau, J., and Viaud-Delmon, I., editors (2003). *Le traité de la réalité virtuelle*, volume 1. Les Presses de l'École des Mines de Paris, deuxième edition. ISBN : 2-911762-47-9.

[Georgeff et al., 1985] Georgeff, M., Lansky, A., and Bessiere, P. (1985). A procedural logic. In *Proceedings of the 9th IJCAI*.

[Glaser, 1996] Glaser, N. (1996). *Contribution to Knowledge Modelling in a Multi-Agents Framework (the CoMoMAS approach)*. PhD thesis, Université Henri Poincaré, Nancy.

[Guilfoyle and Warner, 1994] Guilfoyle, C. and Warner, E. (1994). Intelligent agents : The new revolution in software. *Ovum report*.

[Harrouet, 2000] Harrouet, F. (2000). *oRis : s'immerger par le langage pour le prototypage d'univers virtuels à base d'entités autonomes, thèse de doctorat*. PhD thesis, Université de Bretagne Occidentale.

[Harrouet et al., 2002] Harrouet, F., Tisseau, J., Reignier, P., and Chevaillier, P. (2002). oRis : un environnement de simulation interactive multi-agents. *Revue des sciences et technologie de l'information, série Technique et science informatiques (RSTI–TSI)*, 21(4) :499–524.

[Hollis, 1991] Hollis, M. (1991). Penny pinching and backward induction. *Journal of Philosophy*, 88 :473–488.

[Horst, 1996] Horst, S. (1996). *Symbols, Computation and Intentionality : A Critique of the Computational Theory of Mind*. University of California Press.

[Husserl, 1913] Husserl, E. (1913). *Ideen zu einer reinen Phänomenologie und phänomenologischen Philosophie, erstes buch, 1913 (1950, éd. par W. Biemel) (trad. par Paul Ricoeur sous le titre* Idées directrices pour une phénoménologie*)*. Gallimard Paris, coll. Tel.

[Iglesias et al., 1999] Iglesias, C., Garrijo, M., and Gonzalez, J. (1999). A survey of agent-oriented methodologies. In Müller, J., Singh, M. P., and Rao, A. S., editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 317–330. Springer-Verlag : Heidelberg, Germany.

[Iglesias et al., 1997] Iglesias, C. A., Garijo, M., Centeno-Gonzalez, J., and Velasco, J. R. (1997). Analysis and design of multiagent systems using MAS-common KADS. In *Agent Theories, Architectures, and Languages*, pages 313–327.

[Jennings, 2000] Jennings, N. R. (2000). On agent-based software engineering. *Artificial Intelligence*, 117 :277–296.

[Jennings et al., 2000] Jennings, N. R., Faratin, P., Norman, T. J., O'Brien, P., and Odgers, B. (2000). Autonomous agents for business process management. *Int. Journal of Applied Artificial Intelligence*, 14(2) :145–189.

[Konolige and Pollack, 1993] Konolige, K. and Pollack, M. E. (1993). A representationalist theory of intention. In Bajcsy, R., editor, *Proceedings of the Thirteenth International Joint Conference*

on Artificial Intelligence (IJCAI-93), pages 390–395, Chambéry, France. Morgan Kaufmann publishers Inc. : San Mateo, CA, USA.

[Labrou et al., 1999] Labrou, Y., Finin, T., and Peng, Y. (1999). Agent communication languages : The current landscape. *IEEE Intelligent Systems*, 14(2) :45–52.

[Lamport, 1994] Lamport, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3) :872–923.

[Le Bodic and Favier, 2004] Le Bodic, L. and Favier, P.-A. (2004). A model for naturalistic virtual user simulation. *in proc. of Cognition and Exploratory Learning in Digital Age (CELDA 2004), to appear*.

[Loeck, 1990] Loeck, G. (1990). intentionality and artificial intelligence, review of husserl, intentionality and cognitive science. In *New ideas in psychology*, volume 8, pages 103–111. MIT Press.

[Luck et al., 1997] Luck, M., Griffiths, N., and d'Inverno, M. (1997). From agent theory to agent construction : A case study. In Müller, J. P., Wooldridge, M. J., and Jennings, N. R., editors, *Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages : Intelligent Agents III*, volume 1193, pages 49–64. Springer-Verlag : Heidelberg, Germany.

[Maes, 1989] Maes, P. (1989). The dynamics of action selection. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'89*, pages 991–997.

[Malle et al., 2003] Malle, B. F., Moses, L. J., and Baldwin, D. A., editors (2003). *Intentions and Intentionality : Foundations of Social Cognition*. Bradford Books.

[Manzotti et al., 2003] Manzotti, R., Nadel, J., and Metta, G. (2003). Adapt, deliverable item 2.1 : A theory of intentionality. Technical report, Information Society Technologies, for European Community.

[Merleau-Ponty, 1945] Merleau-Ponty, M. (1945). *Phénoménologie de la perception*. Edition Gaillimard.

[Merleau-Ponty, 1965] Merleau-Ponty, M. (1965). *Les sciences de l'homme et la phénoménologie*. Centre de documentation universitaire, Paris.

[Myers and Morley, 2001] Myers, K. L. and Morley, D. N. (2001). Human directability of agents. In *Proceedings of the international conference on Knowledge capture*, pages 108–115. ACM Press.

[Müller and Pischel, 1993] Müller, J. and Pischel, M. (1993). The agent architecture InteRRaP : Concept and application. Technical Report RR-93-26, DFKI Saarbrucken.

[Müller, 1987] Müller, J.-P. (1987). *Contribution à l'étude d'un agent rationnel : spécification en logique intensionnelle et implantation*. PhD thesis, Institut National Polytechnique de Grenoble.

[Müller, 2002] Müller, J.-P. (2002). Des systèmes autonomes aux systèmes multi-agents : interaction, émergence et systèmes complexes. *Mémoire d'habilitation à diriger des recherches, Université de Montpellier II*.

[Müller et al., 1995] Müller, J. P., Pischel, M., and Thiel, M. (1995). Modelling reactive behaviour in vertically layered agent architecture. In *M. J. Wooldridge and N. R. Jennings : Interacting Agents — Theories, Architectures, and Languages – Lecture Notes in Computer Science, No. 890*, pages 261–276. Springer Verlag.

[Newell, 1990] Newell, A. (1990). *Unified theories of cognition*. Harvard University Press.

[Nwana, 1995] Nwana, H. S. (1995). Software agents : An overview. *Knowledge Engineering Review*, 11(3) :205–244.

[Nwana et al., 1998] Nwana, H. S., Ndumu, D., and Lee, L. (1998). Zeus : A collaborative agents toolkit. In *Proceedings of the Third International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 377–392.

[Odell et al., 2000] Odell, J., Parunak, H. V. D., and Bauer, B. (2000). Representing agent interaction protocols in UML. In *AOSE*, pages 121–140.

[Pachoud, 1997] Pachoud, B. (1997). Trouble de la commande et du contrôle de l'action dans la schyzophrénie. *Le mouvement : des boucles sensori-motrices aux représentations cognitives et langagières, Actes de la Sixième École d'été de l'Association pour la recherche cognitive*.

[Parker, 1993] Parker, L. (1993). Designing control laws for cooperative agent teams. *Proceedings of the IEEE Robotics and Automation Conference*, pages 582–587.

[Petta and Trappl, 1997] Petta, P. and Trappl, R. (1997). Why to create personalities for synthetic actors. In *Creating Personalities for Synthetic Actors*, pages 1–8.

[Pirjanian, 1999] Pirjanian, P. (1999). Behavior coordination mechanisms – state-of-the-art. Technical Report IRIS-99-375, Institute of Robotics and Intelligent Systems.

[Popper, 1959] Popper, K. R. (1959). *The logic of scientific inquiry*. Basic Books, New York.

[Pratt, 1990] Pratt, V. (1990). Action logic and pure induction. In van Eijck, J., editor, *Proc. Logics in AI : European Workshop JELIA '90*, volume 478, pages 97–120. Springer-Verlag Lecture Notes in Computer Science.

[Rao and Georgeff, 1991a] Rao, A. S. and Georgeff, M. P. (1991a). Asymmetry thesis and side-effect problems in linear-time and branching-time intention logics. In Myopoulos, J. and Reiter, R., editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 498–505, Sydney, Australia. Morgan Kaufmann publishers Inc. : San Mateo, CA, USA.

[Rao and Georgeff, 1991b] Rao, A. S. and Georgeff, M. P. (1991b). Modeling rational agents within a BDI-architecture. In Allen, J., Fikes, R., and Sandewall, E., editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484. Morgan Kaufmann publishers Inc. : San Mateo, CA, USA.

[Rao and Georgeff, 1995] Rao, A. S. and Georgeff, M. P. (1995). BDI-agents : from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems (ICMAS'95)*, pages 312–319, San Francisco.

[Rational, 1997] Rational (1997). Unified modelling language (UML) version 1.0. Specification, Rational Software Corporation.

[Reticular Systems, 1999] Reticular Systems (1999). AgentBuilder. an integrated toolkit for constructing intelligent software agents. Technical report, Reticular Systems Inc.

[Ricordel and Demazeau, 2001] Ricordel, P.-M. and Demazeau, Y. (2001). From analysis to deployment : A multi-agent platform survey. *Lecture Notes in Computer Science*, 1972 :93–105.

[Rodin et al., 1998] Rodin, V., Harrouet, F., Ballet, P., and Tisseau, J. (1998). oRis : multiagent approach for image processing. *Proceedings SPIE'98*, pages 57–68.

[Rodin et al., 2000] Rodin, V., Raulet, V., and Nédélec, A. (2000). Using a multiagent language for collaborative interactive prototyping (poster session). In *Proceedings of the third international conference on Collaborative virtual environments*, pages 205–206. ACM Press.

[Sardina and Shapiro, 2003] Sardina, S. and Shapiro, S. (2003). Rational action in agent programs with prioritized goals. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 417–424. ACM Press.

[Sartre, 1943] Sartre, J. (1943). *L'être et le néant, essai d'ontologie phénoménologique*. Collection Tel, Editions Gallimard.

[Scaglione, 1997] Scaglione, M. (1997). *L'intentionnalité et les modèles artificiels, émergence et réalisation : deux côtés de la même pièce*. PhD thesis, Université de Neuchâtel.

[Scerri et al., 2001] Scerri, P., Pynadath, D., and Tambe, M. (2001). Adjustable autonomy in real-world multi-agent environments. In *Proceedings of the fifth international conference on Autonomous agents*, pages 300–307. ACM Press.

[Searle, 1983] Searle, J. (1983). *Intentionality, an essay in the philosophy of mind*. Cambridge University Press.

[Shelley, 1818] Shelley, M. (1818). *Frankenstein or the modern Prometheus*. Lackington, Hughes, Harding, Mavor and Jones edition.

[Shen et al., 2001] Shen, W., Norrie, D. H., Barthes, J.-P. A., and Norrie, D. H. (2001). *Multi-Agent Systems for Concurrent Intelligent Design and Manufacturing*. CRC Press.

[Shoam, 1990] Shoam, Y. (1990). Agent-oriented programming. Technical Report STAN-CS-1335-90, Computer Science Department of Stanford University.

[Sournia, 1991] Sournia, J. C. (1991). *Histoire de la médecine et des médecins*. Ed. Larousse, Paris.

[Thalman et al., 1997] Thalman, D., Noser, H., and Huang, Z. (1997). Autonomous virtual actors based on virtual sensors. In Trappl, R. and editors, P. P., editors, *Creating Personalities for synthetic actors*, pages 25–42. Springer Verlag.

[Tisseau, 2001] Tisseau, J. (2001). *Réalité Virtuelle - autonomie in virtuo*. Habilitation à diriger des recherche, Université de Rennes I.

[Torre, 2003] Torre, L. V. D. (2003). Contextual deontic logic : Normative agents, violations and independence. *Annals of mathematics and artificial intelligence*, 1–2 :33–63.

[Turing, 1950] Turing, A. (1950). Computing machinery and intelligence. *Mind*, 49 :433–460.

[Warren, 1983] Warren, D. H. (1983). An abstract prolog instruction set. Technical note 309, SRI International.

[Wegner, 1997] Wegner, P. (1997). Why interaction is more powerful than algorithms. *Commun. ACM*, 40(5) :80–91.

[West and Hubbold, 1998] West, A. and Hubbold, R. (1998). Research challenges for systems supporting collaborative virtual environments. *Proceedings of Collaborative virtual anvironments 98*, pages 11–23.

[Wielemaker, 2003] Wielemaker, J. (2003). Native preemptive threads in SWI-Prolog. In Palamidessi, C., editor, *Practical Aspects of Declarative Languages*, pages 331–345, Berlin, Germany. Springer Verlag. LNCS 2916.

[Wooldridge and Ciancarini, 2000] Wooldridge, M. and Ciancarini, P. (2000). Agent-Oriented Software Engineering : The State of the Art. In Ciancarini, P. and Wooldridge, M., editors, *First Int. Workshop on Agent-Oriented Software Engineering*, volume 1957, pages 1–28. Springer-Verlag, Berlin.

[Wooldridge et al., 2000] Wooldridge, M., Jennings, N. R., and Kinny, D. (2000). The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3) :285–312.

[Wooldridge and Jennings, 1995a] Wooldridge, M. J. and Jennings, N. R. (1995a). Intelligent Agents : Theory and Practice. *Knowledge Engineering Review*, 10(2) :115–152.

[Wooldridge and Jennings, 1995b] Wooldridge, M. J. and Jennings, N. R. (1995b). *Interacting Agents — Theories, Architectures, and Languages – Lecture Notes in Computer Science, No. 890*. Springer Verlag.

Annexe A

La notion d'intentionnalité

> Et feignant qu'il y ait une machine, dont la structure fasse penser, sentir, avoir perception ; on pourra la concevoir agrandie en conservant les mêmes proportions, en sorte qu'on y puisse entrer, comme dans un moulin. Et, cela posé, on ne trouvera, en la visitant au-dedans, que des pièces qui poussent les unes les autres, et jamais de quoi expliquer une perception. Ainsi c'est dans la substance simple, et non dans le composé, ou dans la machine qu'il faut chercher.
>
> G.W. LEIBNIZ *(La monadologie)*

Ce chapitre propose un survol des aspects philosophiques, psychanalytiques et neurophysiologiques du concept d'intentionnalité afin de considérer sa prise en compte dans l'élaboration d'un modèle comportemental d'agent autonome.
Nous n'avons pas la prétention, ni la légitimité, de mener des études approfondies dans ces domaines très spécifiques mais simplement d'identifier les caractérisitques principales de cette notion sous ces trois éclairages.
Nous tenterons d'une part de donner une définition de l'intentionnalité selon ces différents aspects, et nous tâcherons d'autre part de rapporter ce concept à la définition d'un modèle de comportement en discernant son importance dans les trois phases – perception, décision et action – qui le composent et surtout dans les liens qu'elles entretiennent.

A.1 Intentionalité ou intentionnalité ?

Le sens commun du mot *intentionalité* (avec un seul *n*) s'entend selon un abord purement téléonomique : c'est le but qui sous-tend un acte, acte physique ou acte de pensée. Ce n'est pas le but *à atteindre* (*intentionalité* n'est pas *finalité*) mais celui qui *motive* l'acte. La notion d'*intentionalité* (avec un *n*) et d'acte volontaire se rejoignent. C'est dans ce sens que des expressions comme *erreur intentionelle* ou *coup intentionel* doivent être interprétées.

Pour les philosophes, le terme *intentionnalité* (avec deux *n*) désigne le rapport qu'entretient toute attitude mentale à un contenu, à la chose pensée. La pensée est alors dite *intentionnelle* (avec deux *n*) et le terme *intentionnalité* (avec deux *n*) a une valeur déictique d'identification de cette *représentation* : l'intentionnalité est le lien de correspondance entre l'objet psychique (*objet* de la pensée) et l'objet réel (*visée* de la pensée). Ceci revient à dire que toute pensée contient la représentation de l'objet qu'elle vise, cette relation d'appartenance est l'*intentionnalité* de cette pensée. La signification de cette notion a évoluée dans ses nuances avec la succession des travaux la concernant, la facticité de l'objet réel (*visée* de la pensée) n'étant plus un postulat dans les travaux de HUSSERL. L'aspect philosophique de cette notion est présenté dans la section suivante.

Entre ces deux orthographes, nous avons décidé d'utiliser systématiquement *intentionnalité* (avec deux *n*) afin de garder à l'esprit que cette notion peut désigner bien plus qu'une relation causale. Nous préférons généraliser ce choix et conserver la définition la moins restrictive, même dans les contextes téléonomiques, plutôt que de laisser cette seule différence orthographique distinguer deux notions différentes (bien que très proches).

A.2 Philosophie et intentionnalité : la phénoménologie

Le terme *intentionnalité* dérive du terme latin *intendo* qui signifie *pointer vers, se diriger vers*. C'est dans ce sens dû aux scolastiques[1] de la fin du moyen âge que BRENTANO[2] l'a employé pour désigner ce que nous pourrions nommer *directionnalité* de la conscience : la conscience est toujours conscience *de* quelque chose. Une première définition naïve, mais suffisante dans un premier temps, consisterait à dire que c'est dans ce *de* que réside l'intentionnalité. Mais c'est sans doute EDMUND HUSSERL (1859 - 1938) qui donna à ce terme sa pleine signification comme fondement de la phénoménologie.

Ce courant philosophique qui prit racine à la fin du 19ème siècle peut être comparé au cartésianisme dont il hérite en partie : le but de HUSSERL était d'asseoir les sciences en justifiant les fondements logiques du raisonnement. Il a prit naissance en pleine crise du subjectivisme et de l'irrationalisme et s'inscrivait alors dans une opposition au psychologisme et au pragmatisme. HUSSERL luttait contre ce qu'il considérait être une crise des sciences : si, comme le propose l'empirisme, tout savoir est issu de l'expérience, ce principe ne peut être lui-même confronté à l'expérience. Nous ne pouvons pas avoir d'expérience de l'expérimentation, nous l'acceptons comme principe de découverte de la vérité. Cette forme de scepticisme peut donc mener à la remise en cause de toute connaissance acquise par l'expérimentation, à moins de pouvoir justifier définitivement la logique employée pour cette découverte de la vérité. C'est dans cette direction que s'inscrivaient les travaux de HUSSERL qui voulait effacer toute incertitude de la découverte par la mise en évidence d'un langage de la pensée (le *logos*) afin d'éviter une crise philosophique des sciences.

Cette volonté constitue la base de ce qui est nommé science éidétique (*ontologie de la nature*) qui aspire à la découverte d'une ontologie formelle ou logique pure (*Mathesis Universalis*), expression

[1] Pour les scolastiques, l'intentionnalité est la propriété de la pensée d'être toujours pensée d'un objet différent d'elle-même.

[2] Franz BRENTANO : philosophe et psychologue allemand (Marienberg, Saxe, 1838 - Zurich, 1917). Il souligna la relation d'*intentionnalité* (la conscience est toujours conscience *de*) dans les phénomènes psychiques qu'il classifia, et d'où il fit dériver l'esthétique, la logique et l'éthique. Il influença, entre autres, EDMUND HUSSERL et MAX SCHELER (données encyclopédiques, éditions Hachette).

de la logique de pensée avec laquelle nous prenons conscience de la réalité. Pour cela, il proposait de s'intéresser à la chose qui est objet de l'expérience, à la façon dont elle se présente à nous sans aucune hypothèse contrairement à l'expérimentation scientifique. HUSSERL pronait donc le retour à l'*essence* de la *chose même* [Husserl, 1913] :

- La véritable connaissance réside dans une intuition des essences.

La découverte de ces essences se fait, d'après HUSSERL, par la multiplication des points de vue : c'est en observant le phénomène sous différents angles (spatiaux, historiques,...) qu'il nous révèle pleinement son sens. Ces multiples variations d'un même phénomène ne laissent paraître que les traits invariables, intrinsèques à l'objet de l'expérience : les essences. HUSSERL nomme cette méthode, qui peut être apparentée au doute cartésien, la réduction phénomènologique. La chose, objet de l'étude, est réduite à ses essences, seuls traits constants dans la variation du point de vue, comme la mise en doute systématique dans l'approche cartésienne n'épargne que l'essence de la chose.

Ce travail d'analyse peut être effectué sur l'objet de la pensée, sans que celui-ci ne corresponde à une réalité physique. Ceci est nommé *inexistence intentionnelle*. Par exemple, une assemblée peut discourir de la longueur et de la couleur des défenses d'un éléphant rose, alors qu'il est évident qu'aucun spécimen n'est présent. Un raisonnement semblable peut être effectué sur un animal réel connu de tous mais qui est absent. La réduction phénoménologique n'implique donc pas la réalité physique de l'objet visé ; cet aspect de la phénoménologie est souvent désigné comme *mise en parenthèse de l'existence*. Ceci pourrait amener à remettre en cause l'existence de tout objet de nos expériences, notamment de nos expériences perceptuelles. Cependant, d'après HUSSERL, la vérité provient de l'intersubjectivité : la multiplication des points de vue permet de prendre conscience des essences de l'objet de l'analyse. Il évite par là même que la réduction phénoménologique ne mène au solipsisme[3], l'existence d'autrui étant préliminaire à l'intersubjectivité.

Ces explications donnent un bref aperçu des bases de la phénoménologie ; nous n'avons pas la prétention d'en détailler ici tous les aspects mais seulement d'expliciter le propos de base de ce mouvement déterminant dans l'évolution de la philosophie contemporaine, particulièrement aux travers des œuvres de JEAN-PAUL SARTRE, MAURICE MERLEAU-PONTY et MAX SCHELER qui furent directement influencés par EDMUND HUSSERL.

Nous retiendrons seulement pour notre étude que la phénoménologie présente l'intentionnalité comme *directionnalité* de la conscience. La conscience est conscience *de*, le désir est désir *de*, la crainte est crainte *de*... Tout ce qui constitue le psychisme est interaction avec l'environnement. La phénomènologie, à l'instar de l'objectivisme, abroge donc la limitation classique entre *intérieur* et *extérieur* : il ne saurait exister de *noèse* – acte de pensée – sans *noème* – objet de la pensée – le psychisme est donc entrelacement avec le monde. La phénoménologie définit une sorte d'appartenance entre le monde et la conscience, il s'agit d'une appartenance intentionnelle et non réelle. Le rapport de la conscience à son objet n'est pas celui de deux réalités indépendantes : nous ne pouvons être conscients d'objets qui n'existent pas (dont l'idée même n'appartient pas au monde), et nous pouvons à l'inverse avoir des intentions concernant tous les objets du monde, même imaginaires (je peux souhaiter capturer un éléphant rose). Le monde est bien potentiellement inclus, en terme d'intention, dans la conscience bien que nous ne puissions pas être conscients du monde dans sa totalité, puisque

[3] le solipsisme est la doctrine philosophique selon laquelle il n'y aurait pour le sujet pensant d'autre réalité que lui-même.

nous pouvons avoir des intentions sur tout objet du monde. Cette approche fonde le transcendant dans l'immanent : les intentions, intrinsèquement transcendantes à la conscience, lui sont immanentes.

Cette définition du psychisme comme entrelacement avec le monde remet en cause les approches psychologiques des époques antérieures au développement de la phénoménologie. Le moi psychologique n'est définissable que par les rapports de l'individu au monde ; il est vain de tenter l'analyse de mon comportement indépendamment de la situation dans laquelle il s'inscrit. MAURICE MERLEAU-PONTY a écrit [Merleau-Ponty, 1945] :

> Le monde n'est pas un objet dont je possède par-devers moi la loi de constitution, il est le milieu naturel et le champ de toutes mes pensées et de toutes mes perceptions explicites. La vérité n'*habite* pas seulement l'*homme intérieur*, ou plutôt il n'y a pas d'homme intérieur, l'homme est au monde, c'est dans le monde qu'il se connaît.

Pour nous ramener à la modélisation informatique de comportement, la prise en compte de la notion d'intention fait entrevoir la difficulté classique de représentation symbolique de l'environnement. En effet, si l'agent peut avoir des intentions sur tout objet de son environnement, et même sur des objets non présents dans l'environnement, il doit pouvoir représenter symboliquement tout l'environnement. Quel est alors l'ensemble minimum, l'alphabet — au sens de la collection de symboles – qui permet une telle représentation au sein de l'agent ?

Il faut restreindre les objets potentiels des intentions de l'agent pour ne pas se heurter au "*frame problem*".
La phénoménologie a inspiré, et inspire encore, de nombreux philosophes dans leur étude des phénomènes mentaux. Ces idées sont à l'origine des théories de BRATMAN, qui fondent les modèles d'agents *BDI*, ou des théories sur les actes de langages de SEARLE et AUSTIN, entre autres. Ces dernières donnent lieu à de nombreux travaux de formalisation de la communication entre agents (voire [Chicoisne, 2002] pour un développement détaillé de cet aspect).

A.3 Psychanalyse et intentionnalité : l'agentivité

Les phénomènologues ont enrichi les travaux en psychanalyse de leurs contemporains, notamment en proposant une révision du concept d'inconscient tel que l'avait établi FREUD.
Dans [Sartre, 1943] l'auteur définit la *psychanalyse existentielle* (pp 655-663) où il reproche à la psychanalyse freudienne son objectivisme et son causalisme : le phénoménologue accepte la thèse selon laquelle l'environnement exerce une influence sur le sujet, et que les rêves de ce-dernier puissent être porteurs de sens, mais il refuse l'idée d'une interprétation par l'analyste indépendante du sujet, seulement étayée par une symbolique *générale*. La signification symbolique de l'inconscient est réfutée puisque, pour le phénomènologue, seule la conscience est siège du sens. Pour SARTRE l'inconscient n'existe pas, mais le sujet a conscience de ses tendances profondes, "mieux ces tendances ne se distinguent pas de la conscience" [Sartre, 1943]. Cette conception n'est pas incompatible avec la notion de névrose, le sujet peut avoir une tendance profonde à satisfaire une pulsion et une seconde tendance qui refoule ce comportement. Par contre, elle interdit l'idée de prêter une nature au *ça* : le postulat causaliste de FREUD de la *libido*, base de l'événement traumatique, est rejeté.

MERLEAU-PONTY est moins virulent à l'égard du concept d'inconscient, il écrit d'ailleurs[4] :

[4] [Merleau-Ponty, 1945]

> L'idée d'une conscience qui serait transparente pour elle-même et dont l'existence se ramènerait à la conscience qu'elle a d'exister n'est pas si différente de la notion d'inconscient [...], on introduit en moi à titre d'objet explicite tout ce que je pourrai dans la suite apprendre de moi-même.

Il s'oppose par contre plus fermement à l'argument de FREUD de causation du comportement par le sexuel : d'après lui "si l'histoire sexuelle d'un homme donne la clef de sa vie, c'est parce que dans la sexualité de l'homme se projette sa manière d'être à l'égard du monde, c'est-à-dire à l'égard du temps et à l'égard des autres hommes" ([Merleau-Ponty, 1945], p 185).
Ces différentes critiques de la psychologie freudienne par les phénomènologues ont contribuées à une remise en question de son bien-fondé, incitant les psychanalistes de l'époque à examiner plus avant la légitimité du concept d'inconscient et la psychologie de l'inconscient (notamment avec les travaux de LACAN).

Nous retiendrons de ceci que la signification des représentations symboliques est propre au sujet et que les notions de *tendance* et d'*intention* participent du lien entre l'état mental d'un sujet et ses actes. De même, la concurrence et la contradiction entre ces tendances nécessitent la mise en place de mécanismes d'inhibitions.

Si la phénoménologie, et plus particulièrement la phénoménologie de la perception, a contribué avec le concept d'intentionnalité à l'établissement des bases de la psychologie moderne, c'est dans un sens beaucoup plus courant que le terme intentionnalité est employé dans ce domaine. En effet, pour les psychologues l'intention est à considérer du point de vue téléonomique : elle est le souhait de poursuivre un but précis.

Dans [Searle, 1983], l'auteur fait la distinction entre deux types d'intention : l'intention *préalable* à l'action et l'intention *dans* l'action. Cette taxonomie se retrouve en psychanalyse, dans l'analyse des symptômes de la schizophrénie par exemple. Ainsi, dans [Pachoud, 1997] l'auteur propose trois hypothèses d'explication des différentes formes de symptomatologie de la schizophrénie comme troubles de la production et du contrôle des intentions par le patient :

- la symptomatologie négative – l'initiative motrice du patient est déficitaire, son discours s'appauvrit (comportement *apathique*) – pourrait s'expliquer par "un déficit de l'initiation d'action volontaire, c'est-à dire un blocage de l'aptitude à mettre en acte une intention". Il s'agirait donc d'un trouble de l'intention préalable.
  Nous retiendrons de ceci que la capacité de produire une intention est nécessaire au déclenchement d'une action volontaire.

- la désorganisation, la diffluence du comportement ainsi que la distractibilité du shizophrène pourrait, toujours selon l'auteur, s'expliquer par une défaillance du mécanisme d'inhibition des actions réflexes par les actions *intentionnelles* : le patient est incapable d'ignorer certains stimuli et répond par des actes réflexes, bien que ceux-ci ne soient pas du tout adaptés à la satisfaction de l'intention actuelle du patient. Nous entendons ici acte au sens large, comprenant aussi les perceptions. Par exemple, un patient ne peut réprimer l'envie de tourner la tête pour identifier l'origine d'un mouvement en limite de son champ de vision, alors qu'il est occupé à une tâche nécessitant toute son attention visuelle.
  Nous retiendrons de ceci un nouvel indice de la nécessité d'établir un mécanisme d'inhibition entre les actions volontaires (initiées par une intention préalable, donc par le raisonnement) et

les actions réflexes (comportements réactifs, stimuli/réponse).

– la symptomatologie positive s'expliquerait par une défaillance de ce que les physiologues nomment *décharge corollaire* : la commande motrice envoyée par la partie centrale aux effecteurs est partiellement renvoyée à celle-ci par les voies réafférentes. Ce retour permet un contrôle de l'action en cours, et selon les psychanalistes il serait le principal instrument de l'agentivité, c'est-à dire de la prise de conscience de l'action en cours d'effectuation, et du fait que le sujet en est l'initiateur. La subtilité réside dans le fait que cette réafférence ne concerne pas obligatoirement une action volontaire. Par exemple, si un projectile se dirige vers vous à grande vitesse, vous levez le bras pour le stopper. Vous n'aviez pas l'intention *préalable* de lever le bras, il ne s'agit pas d'une action volontaire. Pourtant, au cours de ce mouvement, la réafférence de la commande motrice vous permet de contrôler au mieux la trajectoire de votre bras pour intercepter le projectile, et c'est *via* ce "*rebouclage*" de la commande que vous prenez en partie conscience de votre action. Si vous êtes interrogé sur ce que vous venez de faire, vous affirmerez avoir levé le bras sans intentions préalable, mais par réflexe. L'hypothèse de BERNARD PACHOUD est que le trouble de ce mécanisme de réafférence peut expliquer le syndrome de persécution chez le schizophrène : en l'absence d'un tel "*retour*" de la commande, ou du moins s'il est altéré, le patient prend conscience de ses actions par les voies perceptives habituelles (il se "*voit*" agir plus qu'il ne se "*sait*" agissant) , il perd alors l'agentivité. Il survient chez lui la sensation d'être spectateur de ses propres actions, comme si une autre personne agissait par l'intermédiaire de son corps. Selon l'auteur, cette hypothèse peut aussi expliquer les hallucinations auditives comme le résultat d'un discours intérieur dont le patient "*oublie*" qu'il est son propre interlocuteur. Ceci peut constituer une explication du syndrome de persécution, le patient n'étant plus l'agent de ses propres actes, il explique son comportement comme le fruit de "*forces*" qui l'obligent. C'est cet aspect de la schyzophrénie qui fonde la croyance populaire que cette maladie équivaut à un dédoublement de personnalité.
  Nous retiendrons de ceci que l'intention n'est pas uniquement l'élément déclencheur de l'action, elle permet de l'ajuster et de prendre pleinement conscience de l'acte en cours. Il faut donc bien distinguer les intentions *préalables* à l'action des intentions *dans* l'action, qui existent aussi dans l'accomplissement d'actions réflexes.

Ces hypothèses sont en partie confirmées par des expériences cliniques et rejoignent certains modèles établis par les neurophysiologues.

A.4 Neurophysiologie et intentionnalité : intentionnalité de la perception

Bien que la phénoménologie ne définisse le psychisme que comme le réseau de nos intentions *tissées* avec le monde, elle ne se heurte pas pour autant au problème classique du rapport entre le corps et l'esprit (le rapport entre les phénomènes mentaux et le cerveau : peut-on parler de psychisme sans corps qui en soit le siège ?). Dans [Searle, 1983] l'auteur en propose même une dissolution :

> Les états mentaux sont à la fois *causés par* les opérations du cerveau et *réalisés dans* la structure du cerveau (et dans le reste du système nerveux central).

Si cette affirmation peut nous sembler être une évidence biologique, elle n'est pas aussi claire-

ment exprimée dans les différents courants philosophiques qui ont abordés le problème "esprit-corps" (béhaviourisme, fonctionnalisme, physicalisme, dualisme,...).

Les philosophes comme les psychanalistes attendent donc désormais des réponses de l'étude physiologique du système nerveux en terme d'intentionnalité.

L'approche neurophysiologique des liens entre l'intention et l'action confirme la complexité des mécanismes mis en œuvre.
En effet, si l'intention permet, comme évoqué dans la section précédente, de "*paramétrer*", d'ajuster nos actions en permanence pour qu'elles soient en adéquation avec le but poursuivi, elle influence aussi nos perceptions.

En ce sens, nous parlerons d'actions perceptives, c'est à dire que les actes perceptifs, comme tout autre, sont exécutés différemment selon l'intention qui les sous-tend. Par exemple, avant de traverser une route vous êtes plus réceptifs à la vision des objets en mouvement et à l'audition des basses fréquences : votre intention préconditionne vos perceptions pour en atténuer certaines et en amplifier d'autres. Pour une description de ces phénomènes, nous renvoyons le lecteur à des ouvrages tels que [Berthoz, 1997].

La découverte des relations exactes entre les intentions et les actions – dont les actions perceptives – est délicate : la complexité du système nerveux et le manque d'outil appropriés à l'étude accroissent l'ardeur de la tâche. Il est impossible d'effectuer des mesures intrusives sur le cerveau humain (pour le bien-être du "*cobaye*") et les dispositifs d'imagerie limitent énormément les expériences possibles (limitation des mouvement, rayon d'action, inertie de mesure...). L'expérimentation animale ne permet de faire que des analogies approximatives : le manque d'explication par le patient, primordiale pour de telles expériences, et la faible complexité des expériences réalisables ne laissent que peu de perspectives. Dans de telles circonstances, aucun neurophysiologue ne se risque évidemment à proposer un modèle complet de comportement humain ! Cependant, faute de pouvoir trouver dans ces travaux *comment le cerveau fonctionne* nous pouvons trouver *pourquoi il ne fonctionne pas* : l'étude des pathologies cérébrales apporte énormément d'informations sur le fonctionnement supposé du cerveau. Si les études neurophysiologiques ne permettent pas de construire un modèle complet, elles permettent donc de valider, ou d'invalider, des hypothèses de fonctionnement.

Parmi ces hypothèses, les travaux récents tendent à confirmer le double rôle de l'intention :
- l'intention participe au déclenchement de l'action ;
- l'intention participe au contrôle de l'action.

Nous retrouvons alors les deux catégories – intention préalable à l'action et intention dans l'action – évoquées précédemment.

Sans entrer dans les détails , nous devons préciser afin d'être plus clairs par la suite que les études menées en clinique s'attachent particulièrement à l'observation du cortex orbitofrontal et basotemporal d'une part, et au cortex parietal postérieur d'autre part. Le premier semble fortement impliqué dans la génération de comportements intentionnels alors que le second est lui concerné par la génération de comportements automatiques. L'étude de cas cliniques de patients souffrant d'une lésion du lobe frontal permet donc de mettre en évidence des troubles de l'intention. Une fois encore, il n'est pas possible d'observer une "*production normale*" d'intention, si ce n'est par comparaison avec une "*production anormale*". En l'occurence, la "*production anormale*" correspond à un déficit, un

retard ou une inadaptation de l'intention. Cette obligation d'étudier "*des cas d'erreur*" plutôt qu'un fonctionnement normal ne fait qu'accentuer la difficulté d'analyse. A ce propos, YVES ROSSETTI et LAURE PISELLA écrivent dans [Bizzi and al., 2003] :

> Because pathologies of intention are well expressed by negative symptoms (i.e., a deficit in intentional control) rather than by positive symptoms (what would hyper-intention stand for ?), it is difficult, if not impossible, to provide a positive definition of intention. It is therefore useful to examine the functional deficits associated with intention disorders.

En ce qui concerne le déclenchement de l'action, le cas du syndrome de dépendance environnementale est riche d'enseignements. A force de répétition, certaines actions ne nécessitent plus de réflexion et deviennent des schèmes sensorimoteurs acquis. Ces schèmes peuvent être déclenchés indifféremment par réflexe ou volontairement. Les malades souffrant de dépendance environnementale ne sont plus capables d'inhibition volontaire du déclenchement réflexe de certains schèmes. Par exemple, un tel malade ne peut s'empêcher de saisir un peigne qui lui est tendu et de se coiffer alors qu'il lui est demandé d'ignorer cet objet[5]. Ce qui est observé est alors un déficit de l'intention : ce n'est pas le déclenchement de l'acte qui relève de l'intention mais son inhibition ou son arrêt : une fois un schème sensorimoteur en cours d'effectuation, seul un arrêt volontaire peut l'interrompre (sauf limitation physique évidemment). L'incapacité de *produire* l'intention préalable à l'action s'observe par exemple chez les patients ataxiques. Entre les deux, nous pourrions placer les personnes souffrant de la maladie de Parkinson : les gens atteints de cette maladie produisent normalement des intentions, ils sont parfaitement capables d'exécuter les actes correspondant (pas d'apraxie) mais un déficit en dopamine bloque certains neurotransmetteurs, inhibant alors le déclenchement de l'acte.

Au niveau du déclenchement de l'action, l'intention a donc le triple rôle d'initier les actes volontaires, d'inhiber les actes réflexes qui ne sont pas eupraxiques et de produire les signaux d'arrêt des actions entreprises.

Pour l'aspect contrôle de l'action, des expériences mettent en évidence des retards ou des ajustement de nos gestes en fonction du but poursuivi. Par exemple, nous ne saisirons pas un stylo de la même manière selon que notre but final est d'écrire avec ou de le poser sur une plateforme. De plus, dans ce deuxième cas le geste se fera d'autant plus lentement que la plateforme visée est petite.
Des expériences de suivi visuel ou de saisie de cibles en mouvement avec des sujets apraxiques et aphasiques mettent en évidence deux "*bandes de temps*" dans le contrôle de l'action. Il apparaît que pour des déplacements brefs (typiquement d'une durée inférieure à deux secondes) l'anticipation (d'action ou de perception) se fait selon un modèle sensorimoteur construit à la volée. Pour des phénomènes plus longs, l'anticipation se fait selon un modèle "*intentionnel*". Ainsi, un patient atteint de lésions du lobe frontal et donc aphasique n'aura pas de problème à ajuster son suivi (gestuel ou visuel) d'une cible dont la trajectoire admet des discontinuités brèves (moins de deux secondes). Par contre, si le but de l'expérience est de désigner un objet lors de sa deuxième présentation, après une dissimulation de plusieurs secondes (une trentaine), le même patient aura des difficultés et identifiera l'objet avec retard. De même, il peinera à réagir en différé à un stimulus ("*attrapez l'objet trente secondes environ après son apparition*") alors qu'il réussira dans les réactions à court terme ("*attrapez l'objet deux secondes environ après son apparition*"). Un autre patient qui a des troubles d'effectuation de l'action (apraxique) mais aucune atteinte du lobe frontal réussira et échouera dans les cas contraires.

[5] Nous pourrions dire qu'un tel geste est intentio*nn*el (au sens de la phénoménologie) sans être intentio*n*el (au sens de la téléonomie) : l'attitude mentale qui correspond au geste de coiffure contient une représentation de sa visée, bien que ce geste ne soit pas volontaire.

Ces expériences tendent à confirmer l'hypothèse selon laquelle nous construisons de nos actions deux modèles : un modèle à court terme et un modèle à long terme.
Le modèle à court terme est un schéma sensorimoteur qui nous permet des réactions rapides (de l'ordre de cent millisecondes) dans nos actions : rectification du mouvement de l'oeil lors d'un suivi visuel, modification de la position de la main lors d'une saisie d'un objet en mouvement... Ces réactions sont rapides parce qu'elles n'impliquent aucune décision : elles correspondent à l'exécution d'actions réflexes, acquises à forces de miliers de répétitions. Ces activités ont pour siège le lobe pariétal[6].
Le modèle à long terme est un modèle intentionnel construit dans le lobe frontal. Il permet d'anticiper l'évolution de l'objet de l'action (respectivement l'objet perçu) de manière consciente, et donc de prendre une décision de déclenchement volontaire, d'inhibition ou d'arrêt d'une action. De telles décisions sont toutefois plus lentes que les réactions automatiques (les expériences mettent en évidence une latence typique de l'ordre de trois cents millisecondes au minimum).
Pour expliquer les relations entre le modèle à court terme et le modèle à long terme, nous pouvons faire une analogie avec le *dead reckoning*. Ce terme désigne une technique de distribution de rendu visuel d'entités informatiques en mouvement. Afin de ne pas saturer le réseau, il faut éviter de transmettre en permanence la position de toutes les entités d'une simulation. La technique du *dead reckoning* consiste alors pour le serveur à transmettre aux clients un modèle cinématique de chaque entité. Dès que l'entité concernée "*sort*" de la trajectoire prévisible par ce modèle, un nouveau modèle de son mouvement est construit à la volée et envoyé aux clients. Les relations entre les deux modèles décrits précédemment sont proches de ce mode de fonctionnement : le modèle à court terme permet d'anticiper sur les variations immédiates de l'évolution de l'objet considéré, alors que le modèle à long terme permettra d'anticiper son évolution si il est "*perdu de vue*" par exemple. Dès la nouvelle acquisition de cet objet, un nouveau modèle sensorimoteur est construit à la volée dans le cortex pariétal, il est le support des réactions à court terme.

Au niveau du contrôle de l'action, l'intention a donc un double rôle : ajuster l'effectuation de l'acte en cours en fonction du but poursuivi et permettre la construction d'un modèle d'anticipation à long terme.

Les expériences cliniques décrites ici et leurs interprétations sont principalement extraites du troisième chapitre de [Bizzi and al., 2003]. La lecture de cet ouvrage dans son ensemble, et particulièrement des chapitres 3 et 4 apportera au lecteur curieux de ces aspects beaucoup de détails et de très nombreux points de départ pour une recherche plus approfondie.

A.5 L'intentionnalité dans le rapport à autrui

Les sections précédentes ont exploré la notion d'intentionnalité selon les approches philosophiques, psychologiques et physiologiques. Nous avons jusqu'à présent particulièrement insisté sur le rôle des intentions dans le déclenchement, le contrôle et l'arrêt de l'action. Il s'agit de l'aspect *interne* du comportement de l'agent. Le lien entre l'intentionnalité et la perception nous incite cependant à explorer plus avant l'aspect *externe* du comportement avec un œil "*intentionnel*".

Un point qui apparaît fondamental dans notre relation à autrui est notre capacité à reconnaître

[6](plus précisément : le lobe pariétal inférieur pour le déclenchement automatique de l'action et le lobe pariétal supérieur pour le contrôle dynamique d'effectuation)

les intentions de l'autre. L'identification des intentions qui sous-tendent les actions de l'autre nous permet d'interpréter ses actes, notamment ses actes de langage. La compréhension d'autrui, comme interprétation de ses actes ou de ses paroles, n'est possible que conjointement à la compréhension du "*pourquoi*" il agit. Or, cette identification de l'intention présuppose la reconnaissance d'autrui comme *ego*, c'est-à-dire comme sujet apte à éprouver un vécu pour lui-même. Nous n'avons pas de moyen de perception directe de l'intention, nous ne pouvons qu'interpréter les intentions d'autrui : nous supposons que sa conduite en fonction de son vécu n'est identique à la conduite que nous aurions en réponse au même vécu que si l'intention est la même. Pour l'énoncer plus simplement : sachant que dans une situation *S* l'intention *I* me mènerait à adopter la conduite *C*, si je perçois qu'autrui est dans la situation *S* et qu'il adopte la conduite *C* alors j'en conclue qu'il est animé par l'intention *I*.

Dans ce cas les intentions sont *prétées* à autrui par empathie : nous interprétons la conduite de l'autre en fonction de la connaissance de nos conduites. Le prérequis fondamental à de telles attitudes est donc la connaissances de ses propres conduites : pour raisonner par analogie, par mimétisme, sur les conduites d'autrui, il faut être capable de se projeter à sa place afin d'identifier ce que seraient nos conduites face au même vécu. Ceci implique pour le sujet qu'il se construise une représentation "*extérieure*" de lui-même : si le lien entre les intentions et les actes demeurait interne au sujet, il ne serait pas possible d'identifier l'intention d'autrui. Nous nous construisons donc, au moyens de nos perceptions, une représentation externe de nos conduites en tant que spectateur de nos propres actes, tout en éprouvant l'agentivité suffisante pour assimiler le lien entre nos intentions et nos conduites observées. A ce sujet, MAURICE MERLEAU-PONTY a écrit dans [Merleau-Ponty, 1965] :

> La connaissance de soi par soi est indirecte, elle est une construction, il me faut déchiffrer ma conduite comme je déchiffre celle de l'autre.

Un des enjeux de la psychologie moderne est de déterminer la part d'inné et d'acquis dans la construction de cette représentation de ses propres actes que le sujet construit. L'étude des comportements infantiles par exemple permet d'aborder l'analyse de ces mécanismes.
Le point clef de la perception des intentions d'autrui est qu'elle nécessite l'emploi d'un modèle du comportement de l'autre. Si dans les rapports humains nous analysons souvent autrui en fonction des connaissances que nous avons de nos propres conduites, nous sommes aussi capables d'adapter ce modèle. Par exemple, un adulte qui joue aux échecs avec un enfant ne lui prétera pas d'intentions très complexes de jeu, faisant l'hypothèse d'une relative faiblesse de ses capacités de raisonnement. Pourtant, les intentions profondes et la détermination des deux adversaires sont sans doute comparables, ce sont les moyens de mise en œuvre de ces intentions qui peuvent éventuellement faire défaut au joueur inexpérimenté. Cet exemple laisse entrevoir un autre aspect sensible du traîtement de l'intention d'autrui : la discrimination entre le conditionnement et le raisonnement. Pour des conduites simples, il n'est pas aisés de déterminer si elles constituent une réponse délibérée à une intention ou si leur déclenchement est de l'ordre du réflexe. Le fait pour le sujet de raisonner sur un modèle d'autrui qu'il se construit peut facilement le mener à une interprétation erronée des intentions "*perçues*". Prenons un exemple du quotidien. Un homme dont le chien fugue le jour d'un départ en voyage interprétera facilement cette conduite de son compagnon comme la volonté de le retarder. A son retour, l'animal qui reçoit une forte réprimende n'assimilera pas qu'il faut être présent les jour de départ, malgrès les explications de son maître… Le chien retiendra par contre certainement que quand il rentre au domicile après une promenade agréable, il est très mal accueilli. Cette expérience le conditionnera et ses escapades verront certainement leurs durées s'allonger par la suite. Cette erreur classique en terme d'éducation canine est dûe à l'interprétation anthropomorhique des conduites du chien par son maître. Au lieu de miser sur des capacités de raisonnement pour comprendre sa faute inaccessible

à l'animal, le bon maître dissimulera sa colère et flattera le chien à son retour. Celui-ci sera ainsi conditionné à ne pas quitter un domicile où il est choyé. Cet exemple peut paraître très simpliste et d'un intérêt très limité. Pourtant, le lecteur dubitatif peut s'employer à réfléchir à un modèle comportemental qui permette à un agent artificiel de se construire des représentations de ses conduites afin de les exploiter dans sa compréhension d'autrui. En confrontant le modèle imaginé à l'exemple simple du maître maladroit et du chien fugueur, la tâche de construction de ce modèle apparaît être d'une ampleur non négligeable.

Dans l'interprétation des intentions d'autrui, l'hypothèse des limites des capacités de raisonnement de l'autre est donc cruciale. La rationnalité infaillible des machines est alors un obstacle à la modélisation comportementale : il faut expliciter les limites du raisonnement (temporelles ou de complexité algorithmique) au sein du modèle comportemental. Par défaut, la machine tient un raisonnement trop "*parfait*" pour être crédible. Dans [Hollis, 1991], MARTIN HOLLIS prend l'exemple d'un jeu simple entre deux joueur pour illustrer à quel point l'application de la théorie des jeux par une machine est inadaptée à la modélisation crédible de joueur humain. Ce jeu consiste à partager dix pièces de monnaie entre deux joueur. Les pièces sont initialement posées sur une table et chaque participant joue à tour de rôle. Soit il prend une pièce et son tour de jeu se termine. Soit il prend deux pièces et le jeu cesse immédiatement. Le raisonnement le plus naturel nous amène à croire que les joueurs, si ils sont rationnels, vont terminer la partie avec respectivement six et quatre pièces : après avoir pris une pièce à tour de rôle, il est fort probable que le joueur qui a débuté la partie prenne deux pièces à son cinquième tour pour terminer avec le gain maximum. Pourtant, si les deux joueurs raisonnent de manière empathique afin d'anticiper sur les choix de l'adversaire, la théorie des jeux prouve que la partie s'achève dès le premier tour, le premier joueur saisissant deux pièces. En effet, si le pemier joueur peut anticiper sur la situation (6-4) de fin de partie, il peut préter à son adversaire les capacités de raisonnement suffisantes pour faire la même prévision. Il est alors fortement probable que le second joueur songera à prendre deux pièces à son troisième tour, cherchant ainsi à conclure sur la situation (4-5) plus avantageuse pour lui. Donc, le premier joueur a intérêt à chercher le (4-2), ce que peux anticiper son adversaire qui visera alors le (1-3) et ainsi de suite... La volonté de maximiser l'espérance de gain amènera alors le premier joueur à saisir deux pièces dès le début, ce qui dessert finalement les deux adversaires. Le lecteur intéressé trouvera dans [Dastani et al., 2003] une étude détaillée des liens entre les théories de la décision classique et qualitative, les systèmes de bases de connaissances tels que les modèles *BDI* et la théorie des jeux. Les différences et les intérêts mutuels sont explorés dans l'optique de la conception de modèles comportementaux. L'étude de l'application de la théorie des jeux dans le cadre de l'interaction entres agents (collaboration ou compétition) a donné lieu à de nombreux travaux, tels que [Delahaye and Mathieu, 1996] qui étudie l'etablissement d'une stratégie de coopération avec l'adversaire au travers de l'exemple du dilemme itéré du prisonnier.

Si l'interprétation des intentions d'autrui est nécessaire à l'interprétation de son comportement et à la communication avec lui, nous avons vu qu'elle est aussi la base de la plupart des comportements sociaux, tels que la collaboration par exemple. Cette dernière ne peut s'instaurer entre deux individus que si chacun identifie les intentions de l'autre. Parmi les comportements sociaux adoptés par des agents en interactions, l'apprentissage constitue un mécanisme d'importance majeure. L'acquisition d'un nouveau savoir au contact d'autrui nécessite d'identifier l'intention satisfaite par la conduite observée. Cette identification peut être explicitée par le tuteur : "*je vais t'apprendre à...*". Cette démarche est nommée apprentissage par démonstration : le tuteur désigne à l'apprenant la succession de conduites qui permettent de réaliser une intention. La principale difficulté est alors la contextua-

lisation des conduites et particulièrement l'assimilation du point de départ et de la terminaison de ce qu'il faut apprendre. Dans le cas où l'intention n'est pas explicitée par le tuteur, nous parlerons d'apprentissage par imitation : l'apprenant doit seul mettre en relation les conduites observées chez le tuteur (éventuellement à son insu) et l'intention ainsi réalisée. Ce comportement nous ramène au problème d'identification de l'intention, et notamment de sa granularité. Pour une conduite simple, où la chaîne causale est directe, l'identification d'une intention locale suffit. Par contre, pour des conduites plus complexes, des capacités de mémorisation et d'abstraction sont indispensables à l'apprentissage. Prenons deux exemples simples d'apprentissage par un enfant. La manipulation des interrupteurs pour allumer ou éteindre un éclairage ne nécessite aucune explication par un tuteur : l'observation seule de l'enfant lui suffit à assimiler la relation causale entre les positions du boutons et l'éclairage. L'apprentissage peut se faire par pure immitation : les moments de départ et de fin de la manœuvre, ainsi que l'intention satisfaite sont identifiées directement par l'enfant. Par contre, un autre exemple du quotidien représente une réelle difficulté d'apprentissage pour un enfant : le nouage des lacets de chaussure. Là aussi, l'enfant identifie sans peine l'intention qui motive la conduite, son début et sa fin. Pourtant il n'assimilera pas le geste seul par pure immitation.Dans ce cas, l'apprentissage par démonstration est-il la solution ? Certainement pas : même en décomposant cette conduite en une succession de conduites plus simples dont les intentions sont clairement identifiées, l'apprentissage reste délicat. Une telle décomposition pourrait donner lieu à l'explication suivante :

- faire une demi-clef avec les deux brins afin de limiter les tensions par frottement ;
- réaliser une ganse avec le brin tenu dans la main gauche qui sera maintenu immobile par la suite et donc nommé le *dormant*. Le but de cette ganse étant de faciliter la suppression ultérieure du nœud ;
- avec le *courant* (le brin tenu dans la main droitee), faire un tour mort autour de la ganse du *dormant* afin de limiter les tensions dans le *courant* et de former une boucle entre le *courant* et la demi-clef initiale ;
- former une ganse avec le *courant* afin de faciliter la suppression ultérieure du nœud et la passer dans la boucle précédemment formée ;
- souquer les ganses formées par le *dormant* et le *courant* afin de terminer le nœud.

Il est évident qu'une telle explication n'aidera pas un enfant (ni un adulte !) à faire ses lacets. Pourtant, l'intention de chaque opération intermédiaire a été explicitée. La granularité de l'intention qui est associée à une conduite lors d'un apprentissage n'est donc pas évidente. Dans cet exemple, il est fort probable que l'apprenant se construit des intentions associées à chaque opération qui ne correspondent pas nécessairement aux intentions "*réelles*" qui les justifient : un enfant n'étudie heureusement pas la théorie des nœuds avant de savoir faire ses lacets. Certaines de ces conduites sont donc surement apprises par pur conditionnement : elles sont effectuées sans autre *raison* que de participer à l'opération *globale* de nouage de lacets. Si le lien entre l'identification de l'intention et l'apprentissage semble indéniable, sa caractérisation exacte est encore l'objet de nombreux travaux, particulièrement en psychologie.

Cette section a mis en lumière une partie bien mince des formes de la notion d'intentionnalité dans le rapport à autrui. Nous pourrions par exemple prolonger cette revue par l'examen des implications morales et juridiques de ce concept. L'idée d'intention est en effet indissociable des notions de responsabilité et de culpabilité. De très nombreuses questions sont examinées par les experts de ces domaines. Par exemple, imaginons qu'un homme souhaite en blesser un autre. En approchant l'arme à la main, il affole sa victime qui prend la fuite et fait une chute mortelle. Comment doit-on considérer la responsabilité et la culpabilité de l'agresseur dans cette fin tragique ? Le lecteur intéressé par les

enjeux de la notion d'intentionnalité dans des domaines aussi variés que ceux qui ont été présentés ici (apprentissage, comportement social, aspect moral et juridique) pourra avec bénéfice se tourner vers la lecture de [Malle et al., 2003].

Annexe B

Détails techniques

B.1 Le fichier `intentionalAgent_pl.pl`

Voici l'intégralité du fichier `Prolog` qui est précompilé et lié avec le moteur d'inférence de `SWI-Prolog`.
Il implémente les prédicats assurant la sélection d'action et la communication avec `ARéVi`.

```
% Prolog file
% An intentional agent knows its class and instance id
% (see intentionalAgent.cpp)
:-dynamic(this_class/1).
:-dynamic(this_id/1).
% he has to declare behavioural rules
% e.g. : behaviouralRule([increase-energy-strong]) :-
%                         getProperty(energy, E),
%                         E < 10.
:-thread_local(behaviouralRule/1).
:-module_transparent behaviouralRule/1.
% he has to declare actions and matching trends
% e.g. : action(eat, [energy-increase]).
:-thread_local(action/2).

% symbolicStrength(+StrengthName, +StrengthValue)
%-------------------------------------------------
%
% strength value is a number or
% a symbol among weak, normal, medium and strong
% new strength can be dynamically declared
:-dynamic(symbolicStrength/2).
symbolicStrength(weak  ,1).
symbolicStrength(normal,100).
symbolicStrength(medium,10000).
symbolicStrength(strong,1000000).
symbolicStrength(Num,Num):-number(Num).
```

```
% getProperty(+PropertyName, ?PropertyValue)
%-----------------------------------------
%
% retrieve the Value of a Property declared in AReVi
%
:-module_transparent  getProperty/2.
getProperty(Property,Value):-
    context_module(M),
    M:this_class(Class),
    M:this_id(Id),
    M:getProperty(Class,Id,Property,Value).
% getProperty(+AgentClass, +AgentId, +PropertyName, ?PropertyValue)
%-------------------------------------------------------------------
%
% retrieve the Value of a Property declared in AReVi
% from the class Id and the instance Id of an agent
% througth a call to getValue which is foreign (see intentionalAgent.Ci)
:-module_transparent  getProperty/4.
getProperty(Class,Id,Property,Value) :-
     getValue(Class,Id,Property,Value).
% sendMessage(+ThreadId)
%----------------------
%
% send the message 'doIt' to the thread identfied by its Id
% this will trigger manageBehaviour
sendMessage(ThreadId):-
    \+(thread_peek_message(ThreadId,doIt)),
    thread_send_message(ThreadId,doIt).
% getDecision(-Decision)
%----------------------
%
% give actual decision back to AReVi
:-thread_local decision/1.
:-module_transparent getDecision/1.
getDecision(Decision):-
    context_module(M),
  M:decision(Decision),
    !.
getDecision([]).
% manageBehaviour
%---------------
%
% main loop : get and treat messages
:-module_transparent manageBehaviour/0.
manageBehaviour:-
    context_module(M),
    M:treatMessage,!,
    manageBehaviour.
% treatMessage.
%--------------
%
% treat thread messages :
%  if stop, thread exits
% else, call every behavioural rules
% choose the best action from the trend list they answer
:-module_transparent treatMessage/0.
treatMessage:-
    thread_peek_message(stop),
write('stop signal received'),nl,flush_output,
    thread_exit(stop).
treatMessage:-
    thread_get_message(_),
    context_module(M),
    M:findall( Trend,
     M:catch(M:behaviouralRule(Trend) ,E,warn(E)),
     TrendListList ),
    flatten(TrendListList,TrendList),
    chooseTrend(TrendList, ChoiceList),
    selectAction(ChoiceList,ActionList),
    retractall(decision(_)),
    asserta(decision(ActionList)).
```

```
% warn(+Message)
%--------------
%
% display a message if an error occurs within a behavioural rule
warn(E):-
    write('Failure during behavioural rule execution, message was :'),
    nl,
    write(E),
    nl.

% chooseTrend( +List, -ChoiceList)
%---------------------------------
%
% built the list of best choices
% from the whole list of trend
% ?- chooseTrend([speed-increase-100, pressure-decrease-101,
%         pressure-increase-100,speed-increase-1,speed-decrease-101],X).
% X = [speed-independent, pressure-decrease]
chooseTrend(List, ChoiceList):-
    builtPropertyTrendList(List, PropertyTrendList),
    chooseAll(PropertyTrendList, ChoiceList).

% builtPropertyTrendList( +List, -PropertyTrendList)
%---------------------------------------
% built a list of Property-TrendList pair
% from the whole list of trend
% ?- builtPropertyTrendList([speed-increase-100, pressure-decrease-101,
%         pressure-increase-100,speed-increase-1,speed-decrease-101],X).
% X = [speed-[100-increase, 1-increase, 101-decrease],
%      pressure-[101-decrease, 100-increase]]
builtPropertyTrendList(List,TrendListList):-
    findall(Prop, member(Prop-_-_, List) , PropertiesSet1),
    list_to_set(PropertiesSet1,PropertiesSet),
    findall( TrendList,
      sortByProperty(PropertiesSet, List, TrendList), TrendListList).

% chooseAll( +TrendListList, -ChoiceList)
%----------------------------------------
% built a list of Property-BestChoice pair
% from the list of Property-TrendList pair
% ?- chooseAll([speed-[100-increase, 1-increase, 101-decrease],
%              pressure-[101-decrease, 100-increase]], X).
% X = [speed-independent, pressure-decrease]
chooseAll([], []).
chooseAll([First|Others], [FirstChoice|OtherChoice]):-
    chooseOne(First, FirstChoice),
    chooseAll(Others, OtherChoice).

% chooseOne( +Property-TrendList, -Property-BestChoice)
%------------------------------------------------------
%
% choose a trend among : increase, decrease, keep and independent
% from a trend list
% ?- chooseOne(speed-[100-increase, 20-decrease], X).
%  X = speed-increase
chooseOne(Property-TrendList, Property-BestChoice):-
    findall(Strength, member(Strength-increase, TrendList),
                                          IncreaseStrengthList),
    sumlist(IncreaseStrengthList, IncreaseSum),
    findall(Strength, member(Strength-decrease, TrendList),
                                          DecreaseStrengthList),
    sumlist(DecreaseStrengthList, DecreaseSum),
    findall(Strength, member(Strength-keep, TrendList),
                                          KeepStrengthList),
    sumlist(KeepStrengthList, KeepSum),
    compareStrength(IncreaseSum, DecreaseSum, KeepSum, BestChoice).

% sortByProperty( +PropertiesSet, +List, -Property-TrendList)
%------------------------------------------------------------
%
% give one Property-TrendList pair
% from the set of valid properties and the whole trend list
% ?- sortByProperty([speed, pressure], [speed-increase-100,
%       pressure-decrease-101,pressure-increase-100,
%        speed-increase-1,speed-decrease-101],X).
% X = speed-[100-increase, 1-increase, 101-decrease] ;
% X = pressure-[101-decrease, 100-increase]
sortByProperty(PropertiesSet, List, Property-TrendList):-
    member(Property, PropertiesSet),
    findall(  StrengthValue-Trend,
      (member(Property-Trend-Strength, List),
                        symbolicStrength(Strength, StrengthValue)),
      TrendList) .

% compareStrength(+IncreaseScore, +DecreaseScore, +KeepScore, -BestChoice)
%-------------------------------------------------------------------------
%
% give the chosen trend among increase, decrease, keep and independent
% from the different score of these trends
compareStrength(Inc, Dec, Keep, Keep-keep) :- Keep > Inc,
                                        Keep > Dec, !.
compareStrength(Inc, Dec, Keep, Inc-increase) :- Inc > Dec ,
                                        Inc > Keep,!.
compareStrength(Inc, Dec, Keep, Dec-decrease) :- Dec > Inc,
                                        Dec > Keep,!.
compareStrength(_Inc, _Dec, _Keep, 0-independent).
```

```
% selectAction(+ChoiceList, -ActionList)
%-------------------------------------
%
% built best action list from ChoiceList
% using list of Action-TrendList pair
:-module_transparent selectAction/2.

selectAction(ChoiceList, ActionList):-
    findall( Action-TrendList, action(Action, TrendList), CandidateList),
    selectAction2(ChoiceList, CandidateList, ActionList).

selectAction2( _,[],[]):-!.

selectAction2(ChoiceList,CandidateList,ActionList):-
    maplist( evaluateCandidate(ChoiceList), CandidateList, UnsortedCandidateList),
    sublist(hasNonNullEffect, UnsortedCandidateList, WithoutNullAction),
    chooseBestAction(WithoutNullAction, ActionList).

hasNonNullEffect(Score-_):- Score =\= 0.

% chooseBestAction(+ActionList, -ChosenAction)
%---------------------------------------------
%
% give list of action to trigger from list
% of scored action

chooseBestAction([],[]):-!.
chooseBestAction(ActionList, [Action]):-
    keysort(ActionList, SortedCandidateList),
    last(SortedCandidateList, _Score-Action).

% evaluateCandidate(+ChoiceList, +Action-TrendList, -Score-Action)
%-----------------------------------------------------------
%
% give score of action from its trend list and
% desired trends (ChoiceList)

evaluateCandidate(ChoiceList, Action-TrendList, Score-Action):-
    evaluateTrendList(ChoiceList, TrendList, 0, Score).

% evaluateTrendList( +ChoiceList, +ActionTrendList, +InstantScore, -FinalScore)
%-------------------------------------------------------------------------------
%
% compute new score of action regarding next desired trend in
% ChoiceList

evaluateTrendList([] , _TrendList, Score, Score):-!.

evaluateTrendList([Property-(Strength-Trend)|Others],TrendList, Score, FinalScore):-
    select(Property-ActionTrend , TrendList, Rest),
    !,
    compareTrend(ActionTrend, Trend, Value),
    ValueAdd is Value * Strength,
    NewScore is Score + ValueAdd,
    evaluateTrendList(Others, Rest,NewScore, FinalScore).

evaluateTrendList([_|Others], TrendList, Score, FinalScore):-
    evaluateTrendList(Others, TrendList, Score, FinalScore).

% compareTrend(+ActionTrend, +DesiredTrend, -Factor)
%---------------------------------------------------
%
% Strength of the desired trend will be multiplied
% by Factor : it gives bonus of action
% regarding the considered property

compareTrend(Trend, Trend, 1).
compareTrend(_, independent, 0).
compareTrend(independent, _, 0).

compareTrend(increase, decrease, -1).
compareTrend(increase, keep, -1).

compareTrend(decrease, increase, -1).
compareTrend(decrease, keep, -1).

compareTrend(keep, increase, -1).
compareTrend(keep, decrease, -1).
```

B.2 Le fichier `intentionalObject.h`

```
#include "agentMethod.h"
#include "knowledge.h"
#include "prolog.h"
using namespace Prolog ;
using namespace AReVi ;
namespace intentionalAgent
{
class IntentionalClass ;
class IntentionalAgent ;
//-------------------------------------------------------------------------
// Class IntentionalObject
//-------------------------------------------------------------------------
class IntentionalObject : public ArObject
{
    //-------- Type information ------------------------------------------------
 public:
  AR_CLASS(IntentionalObject)
  // Type information
  typedef  map<long int,  ArRef<DynamicComponent> > KnowledgeMap      ;
  typedef  map<StlString, ArRef<AgentMethod>      > MethodMap         ;
  typedef  map<StlString, ArRef<DoubleProperty>   > DoublePropertyMap ;
  typedef  map<StlString, ArRef<IntProperty>      > IntPropertyMap    ;
  typedef  map<StlString, ArRef<StringProperty>   > StringPropertyMap ;

    //-------- Public operations -----------------------------------------------
    // Inspectors
    bool
    isUpToDate(void) ;

  // Knowledges
  int
    verify(const StlString & query) ;
  bool
    eval(const StlString & code) ;
  void
    declareKnowledge(const StlString & code, bool builtIn = false) ;
  void
    declareKnowledge(long int knowledgeIndex, const StlString & code,
                     bool builtIn = false) ;
  bool
    declareKnowledge(ArRef<XmlNode> node) ;
  void
    removeKnowledge(long int key) ;
  ArRef<DynamicComponent>
    accessKnowledge(long int key) ;
  void
    declareActionDescription(const StlString & knowledgeCode,
                             ActionDescription::TrendList trendList, bool builtIn) ;
  void
    declareActionDescription(long int knowledgeIndex,
                             const StlString & code,
                             ActionDescription::TrendList trendList, bool builtIn) ;
bool
  declareActionDescription(ArRef<XmlNode> node) ;
 void
   declareBehaviouralRule(const StlString & knowledgeCode,
                          BehaviouralRule::BalancedTrendList trendList,
                          bool builtIn) ;
 void
   declareBehaviouralRule(long int knowledgeIndex,
                          const  StlString & code,
                          BehaviouralRule::BalancedTrendList trendList,
                          bool builtIn) ;
 bool
   declareBehaviouralRule(ArRef<XmlNode> node) ;
const KnowledgeMap *
  getKnowledgeTable(void) const ;
 //----
 void
   declarePerception(const StlString & perceptionMethod) ;
 void
   removePerception(const StlString & perceptionMethod) ;
const set<StlString> *
  getPerceptionList(void) const ;

 //----
 // Properties
 void
   declareDoubleProperty(const StlString & propertyName,
                         const  StlString & code, bool builtIn = false) ;
 bool
   getDoublePropertyValue(const StlString & propertyName, double & value) ;
 void
   removeDoubleProperty(const StlString & propertyName) ;
 ArRef<DoubleProperty>
   accessDoubleProperty(const StlString & propertyName) ;
const DoublePropertyMap *
  getDoublePropertyTable(void) const ;
```

```
  //----
  void
    declareIntProperty(const StlString & propertyName,
                       const  StlString & code, bool builtIn = false) ;
  bool
    getIntPropertyValue(const StlString & propertyName, long int & value) ;
  void
    removeIntProperty(const StlString & propertyName) ;
  ArRef<IntProperty>
     accessIntProperty(const StlString & propertyName) ;
const IntPropertyMap *
  getIntPropertyTable(void) const ;
  //----
   void
     declareStringProperty(const StlString & propertyName,
                           const  StlString & code, bool builtIn = false) ;
   bool
     getStringPropertyValue(const StlString & propertyName, StlString & value) ;
   void
     removeStringProperty(const StlString & propertyName) ;
   ArRef<StringProperty>
     accessStringProperty(const StlString & propertyName) ;
const StringPropertyMap *
  getStringPropertyTable(void) const ;
  // Methods
void
     declareMethod(const StlString &  methodName,
                   const  StlString & code, bool builtIn ) ;
  void
    removeMethod(const StlString & methodName) ;
  ArRef<AgentMethod>
    accessMethod(const StlString & methodName) ;
const MethodMap *
  getMethodTable(void) const ;
  //----
  // Source management
 static
   StlString
    getSourceDirectory(void) ;
 static
   StlString
   getObjectDirectory(void) ;
 static
   StlString
   getXmlDirectory(void) ;
 static
   void
   setSourceDirectory(const StlString & directory) ;
 static
   void
   setObjectDirectory(const StlString & directory) ;
 static
   void
   setXmlDirectory(const StlString & directory) ;
static
bool
  xmlLoad(const StlString & fileName) ;
virtual
  bool
    xmlSave(void) {return false;};
StlString
  getName(void) ;
 StlString
   getParentName(void) ;
 void
   setParent(ArRef<IntentionalClass>) ;
virtual
StlString
  getClassName(void) ;
virtual
  StlString
    getSpecificSourceCode(void) ;
bool
  update(bool updatingClass = false) ;
virtual
inline
void
  copyAsNewClass(StlString /*className*/) {};
```

```
    //-------- Forbidden operations ----------------------------------------------
protected:
    //-------- Construction / Destruction ----------------------------------------
// AR_CONSTRUCTOR(IntentionalObject) ;
 IntentionalObject(ArCW & arCW) ;
 // virtual ~IntentionalObject(void) ;

  typedef ArRef<IntentionalAgent> (*IntentionalAgentConstructor)(void) ;
  //-------------
  // Private methods
  // Knowledges
static
    int
    _unsafeVerify(const StlString & query, Module m ) ;
 ArRef<DynamicComponent>
   _knowledgeLookUp(long int knowledgeIndex) ;
 // Properties
 ArRef<DoubleProperty>
  _doublePropertyLookUp(const StlString &  propertyName) ;
 ArRef<IntProperty>
   _intPropertyLookUp(const StlString & propertyName) ;
 ArRef<StringProperty>
   _stringPropertyLookUp(const StlString & propertyName) ;
 // Methods
  ArRef<AgentMethod>
    _methodLookUp(const StlString & methodName) ;
  // Source management
  bool
    _updateFile(bool updatingClass) ;
  bool
    _compileSpecificCode(StlString & compilationDate,
 StlString wishedDate,
 bool updatingClass) ;
void
  _copyAsNewClassCore(const StlString & className) ;
bool
  _xmlDump(ArRef<XmlNode> rootNode) ;
bool
  _xmlParse(ArRef<XmlNode> rootNode) ;
  void
    _parseDynamicComponentListNode(ArRef<XmlNode> node,
void(IntentionalObject::*declarationMethod)
                                   (const StlString &,const StlString &,bool)) ;
// Perception - Decision - Action Loop
  void
    _performPerception(ArRef<IntentionalAgent> caller) ;
virtual
inline
StlString
  _createEvalCode(const StlString & ) { exit(0);} ;
virtual
inline
bool
  _eval(void * ) {exit(0);} ;

  //-----------
  // Attributes
 ArRef<IntentionalClass> _parent ;
  StlString _name ;
  KnowledgeMap _knowledgeTable ;
  long int _knowledgeIndex ;
  set<StlString> _perceptionList ;
  DoublePropertyMap _doublePropertyTable ;
  IntPropertyMap    _intPropertyTable ;
  StringPropertyMap _stringPropertyTable ;
  MethodMap _methodTable ;
  bool _isUpToDate ;
 void *  _dynamicLibrary ;
 StlString _dynamicLibraryDate ;
  // Static
 static StlString _sourceDirectory ;
 static StlString _objectDirectory ;
 static StlString _xmlDirectory ;
};
} // END_OF_NAMESPACE
// inline code
//#include "intentionalObject.Ci"
#endif // INTENTIONALOBJECT_H
//---------------------------------------------------------------------------
```

B.3 Le fichier `intentionalClass.h`

```
// This software package is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; either version 2.1 of the
// License, or (at your option) any later version.
// This software package is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
// General Public License (file LICENSE-LGPL) for more details.
//------------------------------------------------------------------------------

#ifndef INTENTIONALCLASS_H
#define INTENTIONALCLASS_H 1

#include "intentionalObject.h"

using namespace AReVi ;
namespace intentionalAgent
{
//------------------------------------------------------------------------------
// Class IntentionalClass
//------------------------------------------------------------------------------
class IntentionalClass : public IntentionalObject
{
    //-------- Type information ------------------------------------------------
public:
    AR_CLASS(IntentionalClass);

    //-------- Construction / Destruction --------------------------------------

    //-------- Public operations -----------------------------------------------
// Class method

static
ArRef<IntentionalClass>
  getClass(const StlString & className) ;

static
  const map<StlString, ArRef<IntentionalClass> > *
  getClassTable(void) ;

StlString
  getClassName(void) ;

bool
  hasToBeRebuilt(void) ;

  // Properties

    void
      declareDoubleProperty(const StlString & propertyName,
                            double(IntentionalObject::*  methodPointer)(),
    const StlString & code) ;
    void
      declareDoubleProperty(const StlString & propertyName,
                            const StlString & code, bool builtIn = false) ;

    ArRef<DoubleProperty>
      accessDoubleProperty(const StlString & propertyName) ;

    ArRef<IntProperty>
      accessIntProperty(const StlString & propertyName) ;

    void
      declareIntProperty(const StlString & propertyName,
                         long int(IntentionalObject::*  methodPointer)(),
    const StlString & code) ;
    void
      declareIntProperty(const StlString & propertyName,
                         const StlString & code, bool builtIn = false) ;

    void
      declareStringProperty(const StlString & propertyName,
                            const StlString &(IntentionalObject::*  methodPointer)(),
    const StlString & code) ;
    void
      declareStringProperty(const StlString & propertyName,
                            const StlString & code, bool builtIn = false) ;

    ArRef<StringProperty>
      accessStringProperty(const StlString & propertyName) ;

// Methods

    typedef void(IntentionalObject::*  IntentionalObjectMethodPointer)(void) ;

   void
   declareMethod(const StlString & methodName,
                 IntentionalObjectMethodPointer, const StlString & code) ;

   void
     declareMethod(const StlString &  methodName,
                   const  StlString & code, bool builtIn ) ;

   ArRef<AgentMethod>
     accessMethod(const StlString & methodName) ;

 // Source management

    void
    setHeaderPreamble(const StlString & preamble) ;

    void
    setAttributes(const StlString & attributes) ;

    void
    setImplementationPreamble(const StlString & preamble) ;

    void
    setImplementationConstructor(const StlString & constructorCode) ;

    void
    setImplementationDestructor(const StlString & destructorCode) ;
```

```
    StlString
    getHeaderPreamble(void) ;

    StlString
    getAttributes(void) ;


    StlString
    getImplementationPreamble(void) ;

    StlString
      getImplementationConstructor(void) ;

    StlString
      getImplementationDestructor(void) ;

  StlString
  getSpecificSourceCode(void) ;
    bool
      generateClassFile(void) ;

      StlString
      getHeaderCode(void) ;

    StlString
      getImplementationCode(void) ;

    bool
      build(void) ;

bool
  update(bool useless = true) ;
    void
      copyAsNewClass(const StlString & className) ;
bool
  xmlSave(void) ;

bool
  xmlLoad(ArRef<XmlNode> root) ;
    // Instance creation
    ArRef<IntentionalAgent>
      getInstance(void) ;


//-------- Forbidden operations ----------------------------------------------
protected:
    AR_CONSTRUCTOR_1(IntentionalClass,
       const StlString &, name);
// Private methods
StlString
  _createEvalCode(const StlString & ) ;
inline
bool
  _eval(void * ) ;

// Attributes
// Class attributes
static
  map<StlString , ArRef<IntentionalClass> > _classTable ;
// Instance attributes

StlString _headerPreamble ;
StlString _headerAttributes ;
StlString _implementationPreamble ;
StlString _constructor ;
StlString _destructor ;

IntentionalAgentConstructor  _constructorMethod ;
void *  _library ;
 StlString _classLibraryDate ;
 bool _hasToBeRebuilt ;
};
} // end of namespace
// inline code
//#include "intentionalClass.Ci"
#endif // INTENTIONALCLASS_H
//--------------------------------------------------------------------------
```

B.4 Le fichier intentionalAgent.h

```
// LI2/ENIB - AReVi -
//-------------------------------------------------------------------------
// file        : IntentionalAgent.h
// description :
// creation date : 2003-11-24
// author(s)   : Pierre-Alexandre FAVIER (favier@enib.fr)
//-------------------------------------------------------------------------
// Copyright (C) 2003,2004  LI2/ENIB
//   Laboratoire d'Ingenierie Informatique (LI2)
//   Ecole Nationale d'Ingenieurs de Brest (ENIB)
//   CP 30815 - 29608 Brest Cedex - FRANCE
//   Tel: (33) 02.98.05.66.31 , Fax: (33) 02.98.05.66.29 , e-mail: li2@enib.fr
//
// This software package is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; either version 2.1 of the
// License, or (at your option) any later version.
// This software package is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
// General Public License (file LICENSE-LGPL) for more details.
//-------------------------------------------------------------------------
#ifndef INTENTIONALAGENT_H
#define INTENTIONALAGENT_H 1

// Version number definition : KEEP THIS EXACT SPACING !!!
#define PACKAGE_VERSION 0.06
// Log level definition:
// 0 : Important log for application, should be displayed
// 3 : Important log concerning a particular instance
// 5 : Information, can be safely ignored
// 9 : debug purpose
// Error level definition :
// 0 : Non recoverable error, application should terminate
// 3 : Important error, agent should be destroyed (if it still exists)
// 5 : "nice" error (method missing,...)

#include "intentionalObject.h"

using namespace AReVi ;


namespace intentionalAgent
{
//-------------------------------------------------------------------------
// Class IntentionalAgent
//-------------------------------------------------------------------------
class IntentionalAgent : public IntentionalObject
{
  //-------- Type information ------------------------------------------------
 public:
  AR_CLASS(IntentionalAgent)

  //-------- Public operations -----------------------------------------------
  //------------------------------
  //  INSTANCE
  //------------------------------
  // Methods
  bool
    perform(const StlString & methodName) ;

  // Perception - Decision - Action loop control
  double
    getPerceptionInterval(void) ;
  double
    getDecisionInterval(void) ;
  double
    getActionInterval(void) ;
  void
    setPerceptionInterval(double t) ;
  void
    setDecisionInterval(double t) ;
  void
    setActionInterval(double t) ;
  // Source management
  void
    copyAsNewClass(const StlString & className) ;
  StlString
    getSpecificSourceCode(void) ;
  bool
    xmlSave(void) ;
bool
  xmlLoad(ArRef<XmlNode> root) ;
```

```
  //-------- Forbidden operations ------------------------------------------
 protected:
  //-------- Construction / Destruction ------------------------------------
  AR_CONSTRUCTOR(IntentionalAgent) ;
  // Perception - Decision - Action loop control
  bool
    _prologActivation(ArRef<Activity> activity,double interval) ;
  bool
    _actionExecution(ArRef<Activity> activity, double interval) ;
  bool
    _perceptionExecution(ArRef<Activity> activity, double interval) ;
StlString
  _createEvalCode(const StlString & code) ;
bool
  _eval(void * library) ;
  //-----------------------------------------------------------------------
  // Attributes
  //-----------------------------------------------------------------------
  IntentionalAgent * self ;
  int _threadId ;
  // activities
  ArRef<Activity> _prologActivity ;
  ArRef<Activity> _actionActivity ;
  ArRef<Activity> _perceptionActivity ;
};

bool
plForeignPredGetValue(Term className,Term instanceId, Term property, Term value) ;
} // end of namespace ;


//---------------------------------------------------------------
// Macro definition to help dynamic building of IntentionalAgent
//---------------------------------------------------------------
#define IA_HEADER_PREAMBLE(_CLASSNAME, _PARENTNAME)                          \
                                                                             \
                                                                             \
using namespace AReVi;                                                       \
using namespace intentionalAgent ;                                           \
                                                                             \
extern "C" void declareDynamicPart(ArRef<IntentionalClass>) ;                \
                                                                             \
class _CLASSNAME : public _PARENTNAME                                        \
{                                                                            \
public:                                                                      \
friend void declareDynamicPart(void) ;                                       \
    AR_CLASS(_CLASSNAME) ;                                                   \
    AR_CONSTRUCTOR(_CLASSNAME) ;                                             \
                                                                             \
                                                                             \


#define IA_IMPLEMENTATION_PREAMBLE(_CLASSNAME, _PARENTNAME)                  \
using namespace AReVi;                                                       \
using namespace intentionalAgent ;                                           \
AR_CLASS_DEF(_CLASSNAME,_PARENTNAME);                                        \
                                                                             \
                                                                             \
extern "C"                                                                   \
ArRef<IntentionalAgent>                                                      \
getInstance(void)                                                            \
{                                                                            \
return ( (ArRef<IntentionalAgent>)NEW_##_CLASSNAME() );                      \
}                                                                            \
                                                                             \
                                                                             \
_CLASSNAME::_CLASSNAME(ArCW & arCW)                                          \
  : _PARENTNAME(arCW),                                                       \
    self(this)                                                               \
                                                                             \
{                                                                            \


#endif // INTENTIONALAGENT_H
//---------------------------------------------------------------------------
```

B.5 Le fichier `class_Predator.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<IntentionalClass classLibraryDate="Sun Oct 31 00:54:16 CEST 2004" name="Predator"
parent="Creature" dynamicLibraryDate="Sun Oct 31 00:54:19 CEST 2004">
  <KnowledgeList>
    <ActionDescription builtIn="FALSE">randomWalk
      <TrendList>
        <Trend Property="happiness" Trend="2"/>
      </TrendList>
    </ActionDescription>
    <ActionDescription builtIn="FALSE">purchase
      <TrendList>
        <Trend Property="energy" Trend="2"/>
        <Trend Property="nearestPreyDistance" Trend="0"/>
      </TrendList>
    </ActionDescription>
    <BehaviouralRule builtIn="TRUE">
      getProperty('energy',E),
      E &gt; 5000
      <BalancedTrendList>
        <Trend Property="happiness" Trend="2" Strength="weak"/>
      </BalancedTrendList>
    </BehaviouralRule>
    <BehaviouralRule builtIn="TRUE">
      getProperty('energy',E),
      (E &lt; 7000 -&gt;
               write('je suis fatigue'),nl
               ;
               write('je suis en forme, inutile de chasser'),nl, fail),
      getProperty('nearestPreyDistance', D),
      (D == 0 -&gt;
                write('mais aucune proie a l horizon ' ),nl, fail
                ;
                write('et je vois une proie a '), write(D), write(' metres'),nl),
      (D &lt; 20 -&gt;
                write('c''est assez pres pour vouloir chasser'),nl
                ;
                write('c est trop loin...'),nl, fail)
      <BalancedTrendList>
        <Trend Property="energy" Trend="2" Strength="normal"/>
      </BalancedTrendList>
    </BehaviouralRule>
  </KnowledgeList>
  <PerceptionList>
    <Perception>findNearestPrey</Perception>
  </PerceptionList>
  <DoublePropertyList>
    <DynamicComponent name="nearestPreyAngle" builtIn="TRUE">
      nearestPreyAngle = cart2DToAngle( self-&gt;_nearestPreyY,
                                        self-&gt;_nearestPreyZ)  - M_PI_2 ;
    </DynamicComponent>
    <DynamicComponent name="nearestPreyDistance" builtIn="TRUE">
      nearestPreyDistance = cart2DToDistance( self-&gt;_nearestPreyY ,
                                              self-&gt;_nearestPreyZ)  ;
    </DynamicComponent>
  </DoublePropertyList>
  <IntPropertyList>
    <DynamicComponent name="energy" builtIn="FALSE">
      energy = self-&gt;_energy ;
    </DynamicComponent>
  </IntPropertyList>
  <StringPropertyList/>
  <MethodList>
    <DynamicComponent name="findNearestPrey" builtIn="TRUE">
      double nearestPreyY,  nearestPreyZ , bestDistance ;
      nearestPreyY = 0 ;
      nearestPreyZ = 0 ;
      bestDistance = 10000 ;
      double x,y,z ;
      ArRef&lt;ArClass&gt; ia = ArClass::find('IntentionalAgent') ;
      if (ia)
      {
        StlVector&lt;ArRef&lt;IntentionalAgent&gt; &gt; instances ;
        ia-&gt;getInstances(instances, true) ;
        StlVector&lt;ArRef&lt;IntentionalAgent&gt; &gt;::iterator iter =
          instances.begin() ;
        while (iter != instances.end() )
        {
          if ((*iter)-&gt;getParentName() == 'Prey')
          {
           (ar_down_cast&lt;Creature&gt;(*iter))-&gt;_object3D-&gt;
              getPosition(x,y,z) ;
           self-&gt;_object3D-&gt;globalToLocalPosition(x,y,z) ;
          double distance, angle ;
          cart2DToPolar(distance, angle, y, z) ;

          if ((angle &gt; M_PI_2/2) * (angle &lt; M_PI_2 + M_PI_2/2))
          {
             if (distance &lt; bestDistance)
            {
               nearestPreyY = y ;
               nearestPreyZ = z ;
               bestDistance = distance ;
            }
          }
         }
      iter++ ;
        }
      }
```

```
      _nearestPreyY = nearestPreyY ;
      _nearestPreyZ = nearestPreyZ ;
    </DynamicComponent>
    <DynamicComponent name="purchase" builtIn="FALSE">
      double angle ;
      self-&gt;getDoublePropertyValue('nearestPreyAngle', angle) ;
      if (angle &lt; - self-&gt;_angleStep)
        self-&gt;perform('turnLeft' );
      if (angle &gt; self-&gt;_angleStep)
        self-&gt;perform('turnRight') ;

      self-&gt;perform('forward') ;
    </DynamicComponent>
  </MethodList>
  <HeaderPreamble>
    #include &quot;AReVi/Shapes/cone3D.h&quot;
  </HeaderPreamble>
  <HeaderAttributes>
    double _nearestPreyY ;
    double _nearestPreyZ ;
    //----- 3D
    ArRef&lt;Cone3D&gt; _cone ;</HeaderAttributes>
  <ImplementationPreamble>
    #include &quot;AReVi/Lib3D/object3D.h&quot;
    #include &quot;AReVi/Lib3D/material3D.h&quot;
    #include &quot;AReVi/Shapes/shape3D.h&quot;
    #include &quot;AReVi/Utils/maths.h&quot;
  </ImplementationPreamble>
  <Constructor>
    ArRef&lt;Material3D&gt; mat=new_Material3D();
    mat-&gt;setDiffuseColor(0.6,0.2,0.2);
    _cone = new_Cone3D();
    _cone-&gt;setRadius(0.5);
    _cone-&gt;setHeight(1);
    _cone-&gt;setBottom(true);
    _cone-&gt;writeMaterial(mat);
    _cone-&gt;setDoubleSided(false);
    ArRef&lt;Shape3D&gt; shape=new_Shape3D();
    shape-&gt;addRootPart(_cone);
    _object3D-&gt;setShape(shape);
  </Constructor>
  <Destructor>
  </Destructor>
</IntentionalClass>
```

Résumé :

Cette thèse s'inscrit dans le cadre de la spécification de comportements d'agents autonomes situés dans un univers virtuel.
L'objectif est la proposition d'un modèle comportemental qui exploite le concept d'intention comme lien sémantique entre les différentes étapes du cycle de développement logiciel.
A cette fin, la notion d'intention, et plus largement d'intentionnalité, est étudiée sous trois éclairages différents afin de considérer l'opportunité de son emploi dans l'exercice d'une telle spécification. Les aspects philosophiques (à travers la phénoménologie et la philosophie analytique), les aspects psychologiques et neurophysiologiques de cette notion sont ainsi abordés.
Il ressort de cette analyse :

- le rôle fondamental de l'intention dans le déclenchement, le contrôle en cours d'effectuation et l'arrêt de l'action ;
- le rôle de l'intentionnalité dans le dialogue entre les différents intervenants impliqués dans le processus de spécification, l'agent étant alors l'objet de leur discours.

Cette étude introduit une critique des approches de spécification comportementale existantes, particulièrement des architectures BDI (Belief Desire Intention). Il apparaît que celles qui exploitent la notion d'intention l'utilisent comme guide de la planification ou comme concept de modélisation. Il s'agit presque systèmatiquement d'approches symboliques.
L'ensemble de ces réflexions pose les bases du modèle proposé : c'est un modèle réactif qui exploite la notion d'intention pour effectuer une sélection d'action. L'agent emploie des règles comportementales décrites dans un langage déclaratif afin de générer des intentions à satisfaire en fonction de sa situation. L'exploitation de connaissances des effets présumés de ses actions sur lui-même lui permet de mener un raisonnement qualitatif pour sélectionner l'action la plus adaptée. La notion d'intention permet donc d'établir un lien entre un ensemble de règles comportementales symboliques et un corpus de méthodes impératives (les schèmes sensori-moteurs de l'agent) via l'explicitation du "sens de l'action".
D'autre part, l'instauration de ce lien sémantique au sein du modèle favorise la communication entre les différents intervenants impliqués dans la spécification du comportement. La proposition intentionnelle offre les concepts nécessaires à la fédération des points de vue de ces différents spécialistes. La traduction du comportement sous forme de règles comportementales se fait alors selon une sémantique proche de l'ontologie du domaine abordé par la simulation, le concept d'intention étant commun aux différentes phases de conception, depuis l'analyse jusqu'à l'implémentation.
Après avoir détaillé le modèle issu de cette proposition, les prérequis techniques à son exécution et l'application réalisée, la thèse conclut sur les bénéfices de cet apport sémantique au sein d'un modèle réactif pour la spécification dans son ensemble (depuis l'analyse du problème jusqu'à l'interprétation des résultats) et dessine les perspectives en matière de conception d'architectures hybrides. Les voies ainsi ouvertes sont essentiellement :

- l'apprentissage par l'agent de l'effet présumé de ses actions ;
- l'échange de savoir-faire entre agents ;
- la génération automatique d'explication, avec une application possible dans les environnements virtuels de formations ;
- l'élaboration d'une démarche de conception basée sur le modèle comportemental proposé.

Mots clés :

intentionnalité — systèmes multi-agents — réalité virtuelle — intelligence artificielle distribuée