SYSTÈMES MULTI-AGENTS

Programmation d'agents intelligents
Vers une refonte des fils de raisonnement

Benjamin Devèze - Encadrant : M. Patrick Taillibert
Université Pierre et Marie Curie, Master II IAD, rapport de stage

ABSTRACT **Le présent document synthétise les travaux qui ont été menés lors de mon stage de fin d'études de Master Recherche IAD, effectué au sein du groupe THALES. Afin de concevoir et développer des systèmes de mission plus intelligents, plus tolérants et moins complexes, le service qui m'a accueilli emploie les systèmes multi-agents et, notamment, un modèle d'agent basé sur les fils de raisonnement. Ce modèle facilite le traitement des messages, la gestion de plusieurs contextes simultanés ainsi que la description d'agents cognitifs. Nous avons effectué un travail bibliographique conséquent et un état de l'art complet sur la conception et l'implémentation d'agents intelligents. Cela nous a permis de positionner le modèle des fils de raisonnement dans le domaine et de le rapprocher des langages de coordination proposés au milieu des années 90 tels que COOL ou encore AgenTalk. Ce document retrace, en outre, notre étude des fils de raisonnement exposant les limitations que nous avons identifiées, les réflexions qu'elle nous ont inspirées ainsi que nos propositions d'extensions du modèle. Ces extensions visent notamment à augmenter le niveau d'abstraction du modèle et à introduire des buts explicites permettant de conférer plus de proactivité aux agents. Nous décrivons également quelques unes des applications du service dont le développement à été considérablement facilité par les fils de raisonnement. Finalement, nous exposons quelques directions pour les travaux futurs que nous espérons pouvoir mener dans le cadre d'une thèse CIFRE.**

Mots-clés : systèmes multi-agents ; agents cognitifs ; modèle d'agent ; fils de raisonnement

I. Introduction

Le présent document synthétise les travaux qui ont été menés lors de mon stage de fin d'études de Master Recherche IAD, qui a été effectué au sein du groupe THALES. Du fait du caractère orienté recherche de ce stage, nous avons pris le parti de présenter des idées et des réflexions qui, bien que non encore abouties ou formalisées rigoureusement, ont constitué une part significative de cette période. Il nous a donc paru judicieux d'en faire mention afin que ce document remplisse réellement son usage et puisse servir de base à des travaux futurs. Compte tenu des exigences relatives aux rapports de fin de stage, il nous semble donc préférable de prévenir le lecteur qu'une partie significative du contenu présenté ici est encore à un stade embryonnaire et que nous l'aurions éludée dans une perspective de soumission pour une conférence. Il faut également signaler que ce stage a été abordé comme un travail préliminaire à une thèse sur le même thème ce qui explique certaines orientations qui ont été prises, dont notamment l'important travail bibliographique qui a été mené ainsi que les différentes voies qui ont été explorées sans réelle focalisation. Bien que ces orientations se soient, pour l'instant, faites au détriment des réalisations pratiques mises en oeuvre, nous pensons qu'elles se justifient pleinement dans le cadre d'une perspective plus large. Nous avons commencé à implémenter certaines propositions dans le modèle actuel et comptons finaliser ce travail durant le dernier mois de stage.

L'ensemble s'articulera comme suit. Dans une section 2 nous décrirons brièvement le contexte du stage, nous rappelerons le sujet du stage ainsi que la problématique qui s'offrait à nous. Ce sera également l'occasion de présenter l'existant ayant servi de base à cette étude. Dans une section 3, il s'agira de replacer ce travail dans l'état de l'art foisonnant des modèles et langages de programmation pour les agents. Il conviendra alors de présenter dans une section 4 les réflexions et extensions que nous avons proposées ainsi que quelques considérations relatives à l'implémentation pratique de notre système. La section 5 décrira un exemple ainsi que quelques applications visant à illustrer l'utilisation pratique du modèle proposé. Nous porterons un regard critique sur le travail qui a été effectué durant ce stage dans une sec-

tion 6. La section 7 concluera le rapport et ouvrira des perspectives de développements ultérieurs. Suivra une bibliographie recensant la majorité des ouvrages et articles consultés au cours du travail de recherche comprenant tant les documents cités ici que ceux non explicitement référencés.

II. Contexte

A - Le service

Le stage a été effectué au sein de la division *Airborne Systems* du groupe THALES[1] et plus précisément au sein du service d'études en Intelligence Artificielle rattaché à la Direction Technique *Mission and Avionics Systems*.

Ce service principalement composé d'étudiants issus d'écoles d'ingénieurs ou d'universités est placé sous la responsabilité de Monsieur Patrick TAILLIBERT[2] et de Monsieur Bernard BOTELLA et s'intéresse tout particulièrement à l'application des techniques de l'Intelligence Artificielle (IA) aux systèmes aéroportés. Situé à la frontière entre recherche universitaire et pragmatisme industriel, le service tente de bénéficier au mieux de la collaboration entre université et industrie. Par le biais de ses employés, de ses stagiaires et de ses thésards il est résolument ouvert vers la communauté scientifique du domaine, avec laquelle il intéragit.

Les principaux domaines étudiés au sein de ce service sont : la représentation des connaissances, le diagnostique automatique, la planification, l'arithmétique des intervalles, la programmation logique et à contraintes, le raisonnement en temps contraint ainsi que les systèmes multi-agents, objets du présent stage. Signalons à cet égard les travaux de thèse en cours de Caroline Chopinaud [CFST05, CST05] sur le contrôle dynamique d'agents autonomes et de Cédric Dinont sur les agents conscients du temps [DDMT06b, DDMT06a].

B - Les systèmes multi-agents dans le service

Avant de rentrer dans le vif du sujet il est nécessaire de comprendre les problématiques de l'entreprise qui doivent nous guider dans nos recherches. Il est tout d'abord légitime de se demander pourquoi les systèmes multi-agents intéressent le service, nous allons y répondre très succintement et exposer la vision des agents qui s'est dégagée dans l'équipe. Les systèmes de mission qui intéressent le service font intervenir de nombreux composants indépendants tant logiciels, humains et matériels. Ces systèmes volumineux sont soumis à des contraintes temps réel molles. L'objectif est d'accroître l'autonomie et l'intelligence de ces composants ainsi que la tolérance aux fautes des systèmes sans accroître leur complexité structurelle. C'est ce qui a motivé les investigations dans le domaine des systèmes multi-agents.

Nous n'entrerons pas ici dans les débats relatifs à la définition d'un agent ou encore de l'autonomie mais renvoyons à [Fer92, Fer95, WJ95a, Wei99, Col01, Woo02, RN03] et [Cho05]. Contentons nous d'exposer succintement les caractéristiques les plus saillantes que nous souhaitons pour nos agents :

- **Autonomie :** un agent doit être capable d'agir sans intervention humaine ou autre. Il doit également avoir un certain contrôle sur son état interne et les actions qu'il entreprend, ce qui correspond à l'acception relativement classique de l'autonomie [Cho05]. Nous adoptons une vision opérationnelle du principe d'autonomie qui dispose qu'*un agent A est autonome par rapport à B si B ne peut pas prédire à coup sûr le comportement de A* . Une telle exigence a des conséquences extrêmement fortes sur la façon de concevoir un système, tant sur sa décomposition en agents que sur la définition du comportement individuel de chacun d'entre eux. Il permet une prise en compte dès la conception de la tolérance aux fautes et confère plus de robustesse aux systèmes. Le principe de **dépendance limitée** en est une conséquence : le seul mécanisme reliant les agents les uns aux autres est celui de la passation de messages, aucune autre ressource (en particulier la mémoire) ni aucun autre moyen de synchronisation (par exemple les sémaphores) ne sera à la disposition des agents, sauf à mettre en place un troisième agent chargé de la médiation.
- **Réactivité :** un agent doit être capable de répondre aux évènements extérieurs dans un temps raisonnable, dans le but de satisfaire ses objectifs.
- **Proactivité :** un agent doit être capable d'exhiber un comportement dirigé par des buts en prenant l'initiative, pour satisfaire ses objectifs.
- **Cognition :** les connaissances des agents doivent être décrites en terme d'états mentaux et leurs comportements sont intentionnels (buts et plans explicites).
- **Communication / capacités sociales :** un agent doit être capable de communiquer avec les autres agents (possiblement humains), pour

[1] THALES est un grand groupe mondial de l'électronique présent sur 3 marchés : défense, aéronautique et sécurité. Fort d'un chiffre d'affaires de 10,3 milliards d'euros en 2004, THALES emploie 60 000 personnes dans plus de 50 pays et a une position de leader dans le domaine des hautes technologies.

[2] encadrant du stage

satisfaire ses objectifs.
– **Introspection :** un agent doit pouvoir examiner et réfléchir sur ses propres croyances, buts, actions et plans.

C - Implémentation des agents

La mise en oeuvre effective des systèmes multi-agents nécessite un environnement de développement et de déploiement. C'est ainsi que lors d'un stage effectué l'an dernier au sein de la même équipe j'ai conçu et développé ALBA une bibliothèque générique pour le déploiement d'agents écrits en Prolog. Suite à ce développement, j'ai pris l'initiative d'écrire et de soumettre un article [DCT06] qui a été accepté à ProMAS et que j'ai présenté à Hakodate (Japon) durant ce stage. Le lecteur pourra donc prendre connaissance de cet article et trouver plus de détails sur ALBA en consultant ce document qui figure en annexes (Cf. Annexe 1). ALBA n'est pas une plateforme, il s'agit d'une bibliothèque décentralisée, embarquée par tous les agents qui fournit les services nécessaires au déploiement des agents : communication, création d'agents, migration.

Les agents du service sont programmés en Prolog dont les mécanismes de haut niveau (unification, retour arrière, puissance d'expression, métaprogrammation, manipulation d'arbres) sont bien adaptés à la programmation d'agents cognitifs. Le langage Prolog autorise une excellente productivité et s'avère donc un excellent langage pour le prototypage rapide, ce qui est très intéressant dans le cadre d'une activité de recherche conduisant à des allers et retours répétés entre conception et expérimentation.

D - Les Fils De Raisonnement

1. Présentation générale Si ALBA fournit la couche basse permettant le déploiement et l'exécution des agents, elle ne préjuge en rien de la méthodologie à appliquer pour développer en pratique le comportement des agents. Il a donc été nécessaire de développer un modèle d'agent qui exploite les fonctionnalités de la bibliothèque et soit bien adapté aux problématiques des systèmes de mission. Nous allons donc décrire le modèle utilisé dans le service pour la programmation des agents en Prolog, à savoir les Fils de Raisonnement (FDR). Il faut bien comprendre que les FDR utilisent ALBA pour les traitement de bas niveau et agissent comme une couche de niveau supérieur devant totalement masquer les fonctionnalités d'ALBA qui ne seront pas directement accessibles par le programmeur d'agent (Fig. 1). Les FDR ayant fait l'objet de ce stage nous allons expliquer en détails leur mode de fonctionnement ce qui s'avèrera nécessaire pour bien appréhender la suite du rapport.

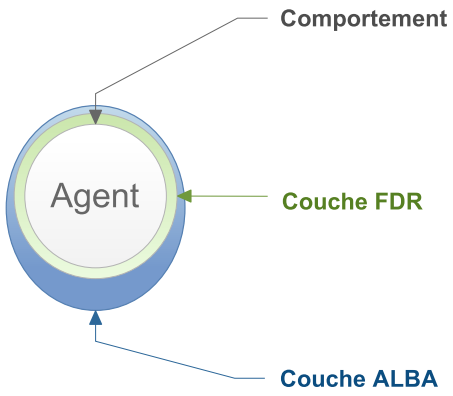


FIG. 1 – Organisation d'un agent

Nous avons expliqué dans l'introduction que l'une des caractéristiques importantes de l'approche multi-agents est celle d'autonomie, permettant de concevoir un système dans lequel les composants logiciels eux-mêmes (les agents) sont conçus de telle sorte qu'ils puissent disposer d'une capacité propre de décision qui n'est pas forcément connue des autres agents lors de leur conception (la retombée attendue étant une réduction de la complexité structurelle des systèmes ainsi qu'une prise en compte plus naturelle de la tolérance aux fautes). Dès lors, la question fut posée de comment implanter cette autonomie ? Les agents ne partageant pas de mémoire, la seule perception qu'ils aient du fonctionnement des autres agents se fait par le biais des messages qu'ils reçoivent. Il est donc rapidement apparu nécessaire de fournir au concepteur d'agent une architecture facilitant l'analyse des messages reçus. Les agents s'engageant dynamiquement dans de nombreuses conversations avec de nombreux agents différents il fut également nécessaire de donner la possibilité de gérer simultanément plusieurs contextes afin de s'accommoder du fait que les agents peuvent recevoir n'importe quel message à n'importe quel moment et venant de n'importe quel agent.

Afin de bien se familiariser avec les concepts des FDR considérons quelques métaphores de la société humaine qui, on le sait, s'avèrent souvent utiles dans le domaine des systèmes multi-agents. Lorsqu'un individu reçoit son courrier, il aura tendance à effectuer un tri et un raisonnement local lui permettant de classer son courrier et de rattacher les lettres à des contextes antérieurs (par exemple réception d'une réponse administrative suite à une demande de sa part) voire à créer de nouveaux contextes (par exemple la réception d'une lettre lui annonçant un héritage entrainera la création d'un contexte lui permettant de traiter les courriers ultérieurs relatifs à cette question). Il est également capable de relier les

lettres présentant un rapport entre elles, d'ignorer les publicités ou les messages de peu d'intérêt. Évoquons également les fils de discussion sur Internet qui permettent de trier les messages suivant leur objet et de gérer plusieurs contextes simultanés. Ce sont notamment ces visions qui ont conduit au développement des FDR.

Chaque FDR peut être vu comme un contexte. Un patron de FDR est décrit sous forme d'automate étendu temporisé représentant une connaissance procédurale de l'agent associée à un contexte. Lorsqu'un patron de FDR est instancié il se voit associer une mémoire locale. Le modèle comprend également une mémoire globale accessible depuis tous les FDR. Les messages sont lus au niveau d'un point névralgique, le *switch*, qui se charge de les aiguiller en fonctions de règles de grammaire (règles pouvant travailler sur le contenu du message et l'expéditeur) vers le ou les FDR appropriés (Fig. 2). Les FDR peuvent être instanciés et détruits dynamiquement en fonction de l'évolution du système et il est ainsi possible d'avoir plusieurs instances d'un même FDR à un instant donné. Il en serait, par exemple, ainsi pour un agent répondant à plusieurs appels d'offre en même temps, il instancierait alors plusieurs FDR de type appel d'offre pour gérer les différentes conversations. Si un message n'est pas filtré par le *switch*, il est expédié automatiquement vers le FDR par défaut ou FDR principal qui est instancié automatiquement au démarrage de l'agent. Il est possible d'ajouter et de supprimer des règles de grammaire dynamiquement ainsi que de modifier des patrons de FDR existants et d'en rajouter de nouveaux ce qui permet d'adapter dynamiquement le comportement des agents.

2. Langage de description des FDR Un patron de FDR est décrit par une liste d'états ainsi que par les transitions entre ces états. Les états *begin* et *end* sont respectivement l'état initial et l'état final de l'automate. Chaque transition de l'automate exprime une condition associée à une action, l'ensemble étant décrit par une clause Prolog présentant la forme suivante :

fdr(Type, State-in, States-out, Message, Sender, PARMFDR, Action).

fdr(Type, State-in, States-out, Message, Sender, PARMFDR, Action) :- Filter.

Les conditions d'applicabilité d'une règle portent sur l'état courant du FDR, sur le message qui a été reçu, sur l'identité de l'expéditeur du message, ainsi que sur un filtre facultatif ayant accès à la mémoire local du FDR, à la mémoire globale de l'agent ainsi qu'au message reçu et à l'expéditeur. PARMFDR est une variable qui contient toutes les informations relatives aux FDR qui sont accessibles au programmeur. La règle spécifie également le ou les états de sortie possibles ainsi qu'une action qui correspond à la procédure à exécuter lorsque la règle est déclenchée.

Une action présente la forme suivante :

action(Message, Sender, PARMFDR)

Les actions sont des prédicats pouvant contenir du code Prolog, des appels à d'autres prédicats mais elles peuvent également modifier la mémoire locale du FDR et la mémoire globale de l'agent, modifier les règles de grammaire du *switch*, modifier les *timeouts* (nous y reviendrons), définir l'état suivant, instancier de nouveaux FDR ou encore fournir une liste de messages à expédier. Ces messages seront envoyés automatiquement par le système à la fin de l'action lorsque le FDR instancié bascule sur un nouvel état.

Il est possible de définir une valeur de *timeout* (valeur en secondes ou *off*) associée à chaque état. Ainsi, si aucun message n'a été reçu pendant le temps spécifié, un *timeout* est automatiquement envoyé par le système ce qui permet de se prémunir contre des attentes infinies de réponses d'autres agents, de prendre les dispositions qui s'imposent si l'on ne reçoit pas les messages attendus dans les délais prévus et donc de tenir compte de l'autonomie des autres agents ou de leurs dysfonctionnements éventuels.

Nous renvoyons à l'exemple de la section 5 pour compléter cette description informelle des FDR.

3. Modèle d'exécution Nous allons décrire brièvement le modèle d'exécution qui régit l'évolution du système. Lorsque le *switch* est initialisé le FDR par défaut est créé ainsi que la mémoire globale de l'agent. Notons que la structure des mémoires est totalement libre. L'interpréteur maintient une structure de donnée contenant notamment l'ensemble des noms des FDR instanciés ainsi que leur état courant, leurs mémoires locales, les informations relatives aux *timeout*, etc. Il présente également une file de messages contenant l'ensemble des messages reçus. Voici à quoi ressemble une itération du cycle d'exécution :

1. calcul du *timeout* à appliquer (minimum des *timeouts* restants pour toutes les instances de FDR confondues)
2. lecture de tous les messages en attente pendant au maximum *timeout* secondes
3. choix du message à traiter par l'application du prédicat *filtering_strategy* qui peut être redéfini par l'utilisateur pour sélectionner le message à considérer dans un ordre différent d'une file classique (comportement par défaut)
4. application des règles de grammaires sur le message choisi pour déterminer l'ensemble des FDR destinataires, s'il n'y a aucun message à

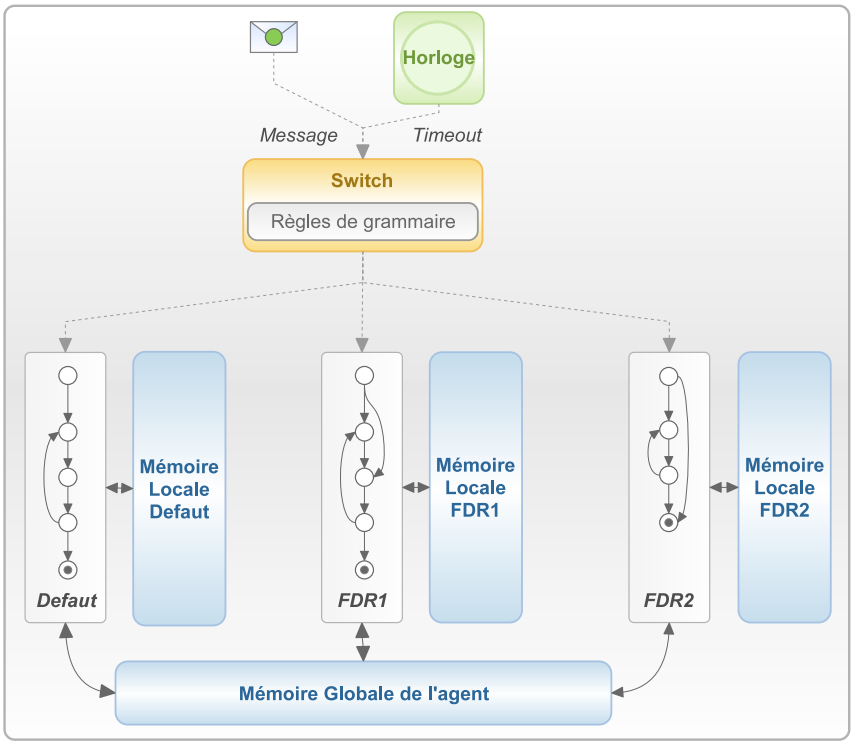


FIG. 2 – Vue générale du fonctionnement des FDR

traiter dans le temps imparti alors on réveille les FDR en *timeout* par un *timeout*

5. application de la partie action des règles dont la partie condition est vérifiée et qui étaient destinataires du message
6. élimination des FDR dans l'état *end* et retour en 1.

Pour le moment, le parti a été pris de n'utiliser qu'un seul processus par agent et de ne pas utiliser les *threads*. Les FDR, d'un certain point de vue, pallient l'absence de *threads* en nous permettant d'isoler différents contextes évoluant en parallèle mais exécutés séquentiellement. En parcourant la littérature, on constate que de nombreux systèmes utilisent plusieurs *threads* par agent et restent souvent relativement silencieux sur les problèmes que cela pose en pratique [Lee06] : risque d'interblocage, mécanismes nécessaires au maintien de la cohérence des données lors des accès concurrents, lourdeur des systèmes. Dans notre approche, les règles étant déclenchées les unes après les autres, il n'est pas nécessaire de mettre en place de systèmes d'exclusion mutuelle lors des accès à la mémoire globale de l'agent ce qui réduit la complexité du système tout en offrant la gestion de plusieurs contextes en parallèle.

E - SUJET DU STAGE ET PROBLÉMATIQUE

L'état d'avancement des recherches dans le domaine des systèmes multi-agents n'a pas encore permis de dégager un modèle d'agent susceptible de satisfaire à l'ensemble des applications mises en oeuvre au sein de la division. Il apparaît même assez clairement que les applications réalisées dans la division aéronautique font appel à des modèles encore peu étudiés par la communauté scientifique (agents temps réel, tolérance aux fautes).

Comme nous l'avons vu, une réflexion a donc été menée afin de définir le modèle d'agent le mieux adapté aux problèmes posés par les applications de l'entreprise. Elle a fait ressortir l'importance d'un modèle cognitif distinct des compétences proprement dites de l'agent. C'est à travers ce composant que le programmeur pourra coder la réflexion que l'agent porte sur sa propre activité ainsi que sur le monde qui l'entoure et c'est donc à travers ce composant que devra se réaliser l'autonomie et la proactivité de l'agent caractéristiques essentielles retenues. C'est dans ce contexte que fut développé le modèle des fils de raisonnement permettant de programmer des agents capables de gérer plusieurs contextes simultanément, les contextes étant déterminés par la syntaxe et la sémantique des messages reçus. Ce modèle a été utilisé dans plusieurs applications (cf. section 5) et s'est révélé extrêmement simple et pratique à utiliser notamment grâce à la séparation des contextes qui permet au programmeur de se concentrer sur un problème local attaché à un contexte précis.

Les objectifs du stage étaient dans un premier temps de partir des résultats obtenus et d'étudier plus

précisément le concept de fil de raisonnement afin d'en dégager les éléments essentiels et de pouvoir le comparer aux modèles d'agent existant par ailleurs. Un important travail bibliographique était donc prévu afin de positionner le modèle dans l'état de l'art. Il s'agissait ensuite de mettre en avant les limitations du modèle et d'en proposer des extensions afin de monter en abstraction et de se rapprocher du modèle attendu que nous avons exposé en introduction.

III. État de l'art

Cette partie a pour vocation de replacer le modèle des FDR dans l'existant ce qui a nécessité un travail bibliographique conséquent. Il est à cet égard intéressant de constater que les travaux les plus proches des FDR ont été trouvés en marge des ouvrages classiques sur les architectures d'agent ou sur les langages de programmation pour les agents mais dans le domaine des langages de coordination. Cela s'explique par le fait que la communauté scientifique du domaine se situe généralement à un niveau d'abstraction plus élevé en attribuant des attitudes mentales aux agents puis en étudiant les moyens de les formaliser et de les implémenter. Au contraire les FDR sont nés d'une approche très pragmatique visant à répondre aux problèmes que posent le traitement des messages. Si cette partie sera volontairement brève, nous invitons le lecteur à consulter l'état de l'art intitulé « Conception et implémentation d'agents intelligents » qui a été rédigé durant ce stage et qui se trouve en annexes du présent document (Cf. Annexe 2). Il a, en effet, semblé nécessaire de dresser un bilan complet du domaine car c'est forts de cette vue d'ensemble que nous pourrons tirer parti de l'expérience acquise ces dernières années et proposer des outils robustes et cohérents répondant aux problématiques qui ont été décrites dans la section 2.

A - Les langages de coordination

La description d'un système multi-agents par la spécification de protocoles de coordination régissant le comportement des agents constitue une approche pouvant permettre de décrire des agents aux comportements complexes ainsi que leurs intéractions.

1. AgenTalk AgenTalk [KIO95] est un langage permettant de décrire des protocoles de coordination dans les systèmes multi-agents. Un processus de haut niveau utilisé dans le cadre d'un processus de coordination est un protocole de coordination. AgenTalk est avant tout un langage de programmation permettant de décrire des agents qui se comportent selon des protocoles définis. AgenTalk introduit pour ce faire la notion de *script* qui permet de définir des protocoles sous forme d'automates étendus extrêmement similaires aux FDR. De la même façon, un agent peut exécuter plusieurs *scripts* simultanément lorsqu'ils sont impliqués dans plusieurs processus de coordination. L'ensemble étant extrêmement proche des FDR nous allons nous intéresser aux particularités d'AgenTalk. À défaut de mémoire locale chaque *script* possède un ensemble de variables modifiables durant l'exécution. Le langage permet aux agents de redéfinir les fonctions définies au niveau des *scripts* afin d'induire des comportements différents à partir d'un même *script*. AgenTalk propose une notion d'héritage entre *scripts* qui permet de définir des protocoles plus élaborés à partir de *scripts* existants. Le système propose également une notion d'héritage dynamique, lorsqu'un *script* en invoque un autre ce dernier devient le descendant du premier et a accès aux variables de son parent. Le modèle d'exécution se rapproche également de celui des FDR si ce n'est que lorsqu'il existe plusieurs règles applicables la règle du fils le plus récemment invoqué est sélectionnée.

AgenTalk a été implémenté en LISP, il est disponible ainsi que sa documentation à l'adresse suivante : `http://www.kecl.ntt.co.jp/msrg/topics/at/`

2. COOL COOL (COOrdination Language) [BF95, BF97] est un langage utilisant KQML qui permet de décrire des protocoles de coordination. Il est né de l'idée que les problèmes de coordination peuvent être traités par une représentation explicite des connaissances portant sur les processus d'intéraction entre les agents. Ces connaissances permettent de séparer les capacités de l'agent liées à ses intéractions sociales de celles liées à la résolution de problèmes individuels.

Une activité de coordination est modélisée par une conversation entre deux ou plusieurs agents, conversation spécifiée par un automate à la manière des *scripts* ou des FDR. Les états de l'automate représentent l'état dans lequel se trouve la conversation. Il y a un état initial et plusieurs états de terminaison. Les messages échangés sont des actes de langage (KQML étendu). Un ensemble de règles de conversation définit les messages que reçoit un agent dans un état donné, les actions locales à effectuer, les messages à envoyer et les changements d'état. Un ensemble de règles de recouvrement d'erreurs spécifie comment traiter une incompatibilité entre l'état de la conversation et les messages reçus. Un ensemble de règles de continuation décrit comment les agents acceptent les requêtes de nouvelle conversation ou choisissent de poursuivre dans une conversation existante. Les classes de conversation spécifient les états, les règles de conversation, les règles d'erreur spécifiques à un type de conversation. Les règles contiennent des variables locales ainsi qu'un environnement de conversation persistant qui permet de définir des variables accessibles d'une règle à l'autre. Chaque agent a plusieurs classes de conver-

sation qu'il peut utiliser pour communiquer avec les autres agents. Les conversations courantes sont des instances de ces classes de conversation que l'agent crée lorsqu'il s'engage dans une conversation.

Du fait de délais dans la transmission des messages, de messages perdus, ou de messages non attendus, il se peut qu'un message reçu dans une conversation ne puisse être traité par aucune règle. Les agents peuvent alors utiliser des structures de conversations plus élaborées ou effectuer des traitements spécifiés par les règles d'erreur : ignorer le message, initier une conversation avec l'expéditeur pour clarifier la situation, changer l'état de la conversation ou encore rapporter une erreur.

Les messages transmis contiennent un identifiant de conversation qui permet de les orienter dans les bonnes conversations courantes des agents ainsi que de créer de nouvelles conversations.

COOL inclut un mécanisme permettant de gérer des conversations multiples et de maintenir des dépendances entre les conversations (par exemple une conversation qui est en attente de la fin d'une autre conversation). Il est possible de construire des arbres de conversations et de faire circuler les messages d'un niveau à l'autre en y adjoignant éventuellement des annotations le long de leur parcours. Normalement les conversations s'exécutent en parallèle mais il est possible d'interrompre une conversation jusqu'à ce que d'autres conversations atteignent certains états.

COOL offre également un sous-système qui permet de faire intervenir un agent humain qui pourra intervenir dans certaines conditions, cette intervention sera ensuite enregistrée pour servir ultérieurement. Il propose une interface graphique permettant de visualiser et éditer les conversations.

3. Discussion Qu'on les appelle *scripts*, conversations ou FDR, on comprend aisément que ces systèmes sont extrêmement proches et partagent la même philosophie. Dans tous les cas, la part belle est faite aux messages qui régissent les intéractions entre les agents. Finalement *script*, FDR ou conversation peuvent être vus comme une connaissance procédurale de l'agent pour atteindre un but, ce qui rapproche ces systèmes de PRS [IGR92]. Cependant, AgenTalk et COOL s'intéressent aux protocoles d'intéraction entre les agents favorisant le point de vu social tout en négligeant de prendre en compte l'agent comme système intentionnel exhibant un comportement proactif. PRS décrit lui l'intéraction entre un agent et son environnement se focalisant sur l'aspect individuel et négligeant les intéractions sociales. Nous pensons que les concepts d'AgenTalk et COOL n'ont pas été poussés jusqu'au bout et espérons les étendre. Un autre objectif serait également de réconcilier les approches des langages de coordination et de PRS pour les combiner.

On peut noter une différence entre les FDR et ces deux langages de coordination : il s'agit de l'utilisation des règles de grammaire pour filtrer les messages. COOL utilise des identifiants de conversation intégrés dans les messages pour les aiguiller tandis que dans AgenTalk le programmeur gère explicitement des identifiants dans les messages ainsi que dans les conditions des transitions. Nous pensons que les règles de grammaire apportent plus de flexibilité au filtrage notamment du fait que l'agent puisse les modifier dynamiquement et qu'elles permettent aisément d'aiguiller un même message vers plusieurs FDR. Cela permet de clairement définir des contextes en fonction de la syntaxe et de la sémantique des messages reçus.

Il nous semble également que du fait de l'utilisation de Prolog, des facilités qu'offre ce langage en terme de métaprogrammation et de la simplicité du langage de description utilisé, les FDR vont permettre de pousser plus loin les principes qui ont été introduits par COOL et AgenTalk développant la dynamicité, la flexibilité et les capacités d'introspection de nos agents.

B - Architectures pour les agents

De nombreuses architectures d'agents ont été étudiées [Fer92, WJ95a, Col01, Woo02]. On distingue principalement :

- les approches délibératives, incarnées par des systèmes tels que IRMA, AUTODRIVE ou HOMER, pour lesquels la prise de décision est le fruit d'une déduction logique.
- les architectures réactives, incarnées par l'architecture *Subsumption*, Pengi ou encore les *Situated Automata* [RK96] pour lesquels la prise de décision est implémentée comme un lien direct entre situation et action sans appel à des raisonnements symboliques complexes
- les architectures BDI, telles que PRS [IGR92], pour lesquelles la prise de décision et le fait de manipulation de structures de données représentant les croyances, les désirs et les intentions de l'agent
- les architectures en couches, incarnées par TouringMachines [Fer92] et INTERRAP [MP93], pour lesquelles la prise de décision se fait au niveau de plusieurs couches logicielles raisonnant à des niveaux d'abstraction différents

Il faut noter que les approches hybrides visant à combiner approches délibératives et réactives et, notamment les architectures en couches, tentent de trouver un compromis entre réactions de type réflexe

et les réactions délibératives, mais ne sont pas adaptées lorsque les actions que l'agent réalise sont elles-mêmes des délibérations longues sur lesquels les agents doivent raisonner.

IV. Réflexions et Extensions

Nous allons ici nous attacher à décrire les principales limitations des FDR telles que nous les avons présentés et tenter d'exposer nos réflexions sur les moyens de les surmonter ainsi que des pistes de recherche pour les futurs développements du modèle. Nous procéderons itérativement en nous intéressant successivement à différentes thématiques puis nous tenterons d'unifier ces réflexions dans un bilan qui proposera les prémisses des nouveaux FDR.

A - De la proactivité des agents

Dans l'idéal, nous voudrions des agents ayant une représentation symbolique de leur environnement, ayant des buts à plus ou moins long terme et capables de délibérer rationnellement et de planifier afin d'agir dans le sens de la satisfaction de ces buts. Ces buts doivent pouvoir être donnés en amont par le développeur d'agent, et ensuite, bien sûr, être mis à jour dynamiquement durant la vie de l'agent en fonction de son état interne, de l'évolution de son environnement et de ses intéractions avec les autres agents. De ce point de vu les agents doivent être capables d'exhiber un comportement réellement proactif.

Nos agents doivent également être capables de rester réceptifs aux changements de leur environnement (qui se traduisent par les messages qu'ils reçoivent) et de réagir en conséquence. Il semble donc qu'il soit nécessaire de se tourner vers une approche hybride, ce qui pose de réelles difficultés.

On sait qu'il est relativement « simple » de mettre en place une architecture exhibant un comportement dirigé par des buts en introduisant des hypothèses très simplificatrices (l'environnement ne change pas durant la phase de planification, les buts restent valides durant la procédure, ...). Nous ne pouvons évidemment pas nous reposer sur de telles hypothèses car nos agents doivent évoluer dans des environnements complexes, dynamiques et multi-agents.

Il est également relativement simple de construire des systèmes purement réactifs. Il semble par contre beaucoup plus problématique de trouver le juste équilibre entre comportement dirigé par des buts et comportement réactif. Nos agents, qui ont des capacités de calcul limitées, une connaissance partielle de leur environnement et des autres agents doivent être capables de réagir dans un temps raisonnable aux changements de l'environnement (survenance d'une menace, satisfaction d'un objectif critique à courte échéance, ...) tout en exhibant un comportement rationnel à long terme. Ils doivent pouvoir, après ou pendant une planification d'actions visant à satisfaire les buts pour lesquels ils ont été conçus, tenir compte d'évènements extérieurs qui peuvent influencer considérablement la délibération courante. C'est ainsi que la survenance d'un évènement extérieur peut complètement invalider la délibération courante, la rendre inutile car le but à déjà été satisfait, voire la simplifier car un sous-but a été réalisé par un autre agent.

Le compromis entre comportement dirigé par des buts et comportement réactif est difficile à réaliser et d'ailleurs, les humains ont fréquemment du mal à le faire. Il est certain que se projeter et planifier à trop long terme en restant aveugle aux nouvelles informations pouvant survenir risque de mener droit à l'échec. De même, passer d'un but à un autre sans jamais se focaliser sur un objectif à long terme risque de conduire à la satisfaction d'aucun but. [Fer92, Wei99, RN03] exposent très bien cette problématique qui reste l'un des problèmes majeurs du domaine.

Ce que nous venons de décrire correspond à la vision idéale vers laquelle nous voulons tendre et nous ne prétendons évidemment pas avoir atteint ces objectifs. Il est toutefois utile de rappeler ces objectifs, aussi ambitieux soient-ils, afin de prendre la mesure du travail qu'il reste à accomplir.

Si on en juge par le modèle d'exécution décrit à la section 2, on comprend aisément que la proactivité des agents utilisant le modèle des FDR que nous avons décrit est extrêmement limitée. On remarque que l'agent passe une partie, potentiellement conséquente, de son temps à attendre des messages ou l'apparition de *timeouts*. Durant ces périodes, il reste donc inactif alors qu'il serait souhaitable qu'il puisse faire preuve d'initiative et agir afin de satisfaire les objectifs pour lesquels il a été conçu. En somme, les FDR activent selon les messages reçus des règles de type condition-action et seuls les *timeouts* permettent de déclencher de manière un peu artificielle des comportements proactifs. Bien qu'il soit possible d'attribuer à nos agents une certaine forme de buts dans le sens où ils ont pour buts de suivre le comportement décrit par les FDR instanciés, cela reste un peu alambiqué. Bien sûr cela ne signifie pas que nos agents soient purement réactifs. Les FDR nous ont permis de mettre en oeuvre des agents possédant une mémoire, une représentation symbolique du monde, la capacité de planifier leur trajectoire et de délibérer pour gérer leur emploi du temps, mais nous pensons que pour augmenter le niveau d'autonomie et d'intelligence de nos agents il convient de doter le modèle de buts explicites ce qui va nous amener à réviser un peu le sens

que l'on donne aux FDR.

La proactivité peut être assimilée à une prise d'initiative de l'agent non uniquement gouvernée par les messages qu'il reçoit. Cette prise d'initiative nécessite que l'agent soit à même de déterminer les actions à accomplir. La planification de ces actions nécessite que l'agent ait des buts explicites à atteindre. La question qui se pose est donc de trouver la manière d'intégrer des buts explicites et une forme de sélection de l'action permettant aux agents d'exhiber un comportement proactif tendant à la satisfaction de leurs buts.

Si jusqu'ici nous avons surtout présenté les FDR comme des protocoles de coordination, il est clair que leur structure est également propice à la définition de plans. Ainsi l'ensemble des FDR potentiellement instanciables peut être assimilé à une bibliothèque de plans qui décrivent des séquences particulières d'actions et de tests à accomplir pour atteindre des buts donnés ou pour réagir à certaines situations, à la manière des *Knowledge Areas* de PRS.

Plusieurs approches s'offrent à nous pour intégrer les buts dans les FDR. Il est tout abord possible d'adjoindre à la structure des FDR une condition d'invocation optionnelle. La condition d'invocation se décline en deux partie : la partie déclenchement est une expression logique décrivant les évènements qui doivent survenir pour que le FDR soit invocable, la partie contextuelle est une expression logique spécifiant les conditions devant être vraies pour que le FDR soit invocable. La partie déclenchement peut ainsi se brancher sur des modifications portant sur les mémoires de l'agent (Cf. la définition des évènements et la gestion de la mémoire ci-après) et notamment sur l'ajout de données en mémoire de l'agent que le programmeur assimile à des buts. Il convient ensuite d'ajouter dans le modèle d'exécution une étape consistant à vérifier les FDR déclenchables et à les instancier. Une fois instanciés, ces FDR sont intégrés comme les autres dans le modèle d'exécution.

On peut imaginer une approche à la STRIPS passant par l'ajout d'un champs précondition et d'un champs effet aux FDR pour pouvoir les composer pour atteindre des buts donnés comme un état du monde à atteindre en procédant par chaînage arrière.

Une approche intéressante serait de confier la gestion des buts à un FDR particulier écrit par le programmeur, nous l'appelerons FDR de contrôle. Lorsque l'agent a un but à atteindre, il l'ajoute dans sa mémoire globale, par exemple sous la forme $goal(Goal, Deadline)$. Le FDR de gestion des buts se branche sur les évènements mémoires liés à l'ajout et à la suppression des buts. N'importe quel FDR de l'agent peut donc faire des demandes d'ajout et de suppression des buts. Le FDR en charge de la gestion de ces buts se réveille sur ces évènements et se charge de sélectionner les buts sur lesquels se focaliser en fonction du contexte. Une fois qu'il a décidé les buts sur lesquels se focaliser, ces buts deviennnent des intentions. Le FDR recherche alors les FDR susceptibles de permettre la satisfaction des intentions et les instancie. Plus précisément une intention est un but qui a été sélectionné associé au FDR qui a été instancié pour atteindre ce but. Le FDR de contrôle se charge de gérer les buts en mémoire et de contrôler l'exécution des FDR relatifs aux intentions courantes.

Lorsque le temps imparti pour le but est dépassé ou bien que le FDR de contrôle a reçu une demande de suppression du but, il peut envoyer un signal pour mettre fin au FDR qui avait été invoqué pour satisfaire ce but. Il peut cependant être brutal d'interrompre un FDR qui a commencé a s'exécuter afin d'atteindre un but donné. En effet, en déroulant le plan l'agent a pu s'engager dans des protocoles de négociation avec d'autres agents ou encore effectuer des actions sur l'environnement qu'il conviendrait de défaire. C'est pourquoi au fur et à mesure de leur exécution les FDR associés à un but pourront gérer une pile d'actions à effectuer lors de la réception du signal de fin d'exécution émanant du FDR de contrôle.

Nous pensons qu'il est nécessaire d'introduire une hiérarchie dynamique à la AgenTalk des FDR instanciés. Lorsqu'un FDR en crée un autre, il en devient le parent. Ainsi, le FDR de contrôle invoque un premier FDR pour atteindre un but, ce dernier invoquera potentiellement d'autres FDR qui deviendront ses descendants. Lorsque le FDR de contrôle mettra fin au premier FDR l'ensemble des FDR descendants prendront également fin. On peut rapprocher cette structure des piles d'exécution lors d'appels imbriqués de fonctions.

Cette approche est très flexible dans le sens où il est très facile de modifier le FDR de contrôle et notamment la politique de gestion des intentions et ainsi de modifier la proactivité de l'agent.

Si le système est surchargé, la politique de filtrage des évènements (Cf. paragraphe Divers) peut momentanément ignorer tous les évènements relatifs aux ajouts de buts voir même mettre fin aux FDR associés aux intentions courantes et le système basculera dans un mode réactif. Dès que les évènements ont été traités et que le système se restabilise l'agent pourra à nouveau se focaliser sur ces buts et faire preuve de proactivité.

On peut envisager d'ajouter d'autres données à la structure des FDR comme par exemple l'utilité espérée, la probabilité de succès du plan, etc. qui pourront être utilisées pour construire des fonctions de

sélection de plans basées sur la théorie de la décision lorsque plusieurs plans peuvent être utilisés pour atteindre un même but.

On comprend donc qu'il est envisageable de mettre à profit l'ensemble des travaux menés dans le domaine de la planification (notamment la planification hiérarchique) pour adapter le comportement des agents en fonction du domaine d'application, des conditions de l'environnement, etc.

Nous nous sommes jusqu'ici placés dans un cadre où l'agent se sert uniquement de ses plans en bibliothèque pour atteindre ses buts. Si aucun FDR n'a été prévu pour atteindre un but particulier il serait envisageable de faire appel à un planificateur externe pour générer des plans qui pourront ensuite être stockés dans la bibliothèque de plans.

Les plans des agents peuvent être incomplets et la connaissance pour les corriger n'être accessible que durant l'exécution. Les agents doivent donc être capables d'étendre et de modifier leurs plans pendant l'exécution. Ainsi les FDR, déjà dans leur forme actuelle, peuvent présenter des transitions partiellement instanciées, l'état sur lequel se brancher étant une variable ce qui permet de construire dynamiquement des parties de plan et de les greffer au fur et à mesure de l'exécution. A titre d'exemple, lorsque l'on rentre en contact avec une administration pour une démarche particulière, on ne connaît pas la démarche précise à suivre. L'administration nous fournit alors le protocole à suivre en cours de conversation et nous en tenons compte pour élaborer notre plan dynamiquement.

Finalement, les agents ont leurs propres plans qui leur permettent d'atteindre leurs buts. Les plans des agents représentent de manière explicite les intéractions avec les autres agents, ces intéractions se font par échanges de messages. Les agents ne peuvent prédire le comportement exact des autres agents, mais ils peuvent déterminer des classes de comportements attendus. De ce fait, les plans des agents sont conditionnés par les actions et réactions possibles des autres agents.

B - De la mobilité

ALBA permet de faire migrer les agents. Comme cela est mentionné dans [DCT06], ALBA supporte une migration faible dans le sens où le processus est relancé sur la machine distante sans sauvegarde de la pile d'exécution, toutefois la bibliothèque fournit des facilités pour la sauvegarde et la restauration de l'état de l'agent avant et après la migration. Dans ces conditions, si le modèle d'agent est compatible avec la migration il est possible d'assurer une migration forte et de permettre une reprise transparente de l'exécution de l'agent sur la nouvelle machine d'accueil. Voyons comment cela se déroulera en pratique.

Nous proposons d'ajouter une encapsulation du prédicat ALBA de migration dans les prédicats offerts par les FDR dans la partie action des transitions :

migrate(Host)

Le système s'assurera que l'appel à la migration est la dernière opération effectuée pour la transition où elle est appelée. L'appel entraînera les étapes suivantes :

1. sauvegarde des états, des mémoires et des données pertinentes des différents FDR
2. appel à la migration d'ALBA
3. une fois sur la nouvelle machine d'accueil restauration des données
4. mise à jour des dates de réveil avec la nouvelle heure
5. reprise transparente de l'exécution

On comprend dès lors que ce système permet d'assurer le support d'une migration forte pour notre modèle celui-ci étant parfaitement défini par les données sauvegardées.

C - De la nécessité d'une communication entre FDR

Dans nos applications, le besoin s'est fait sentir de faire « communiquer » différents contextes. Envisageons un exemple illustrant cette constatation. Imaginons que nous ayons un FDR, *vacance*, dédié à l'organisation de nos vacances et un autre FDR, *travail*, consacré à notre travail professionnel. Soudain, *travail* reçoit un message posant une contrainte urgente qui va nous forcer à replanifier nos vacances. On voit que les contextes ne sont pas forcément indépendants et l'on sent bien qu'il semble nécessaire d'introduire des mécanismes permettant de les faire communiquer.

Il est possible, en l'état, de faire communiquer les FDR par le biais de la mémoire globale de l'agent mais cela implique de mettre en place un contrôle répété de la mémoire dans les FDR devant communiquer. Cela s'avère peu satisfaisant et inutilement consommateur en ressource processeur. Il est également possible de faire communiquer les FDR en s'envoyant des messages à soi-même et en modifiant les règles de grammaire de telle sorte que les messages émis soient bien aiguillés vers le ou les FDR destinataires. Cela reste malgré tout un peu artificiel et il nous semble judicieux d'introduire le concept de signal interne qui permet de distinguer proprement les messages qui sont échangés avec les autres agents sous forme d'actes de langage des signaux internes qui sont des

mécanismes propres à l'agent.

La solution technique envisagée serait donc de mettre en place un mécanisme de signaux internes à l'agent constituant des évènements susceptibles de réveiller les FDR au même titre que les messages. L'adjonction d'un tel mécanisme dans le modèle actuel ne pose pas de problème technique et peut se faire simplement. Il suffit d'ajouter une routine accessible dans la partie action des transitions de la forme :

send_signal(Signal, FDR_Destinataire)

Si *FDR_Destinataire* est une variable le signal est envoyé à tous les FDR instanciés, s'il s'agit d'un nom de type le signal est aiguillé vers tous les FDR instanciés du type donné sinon il est envoyé uniquement à l'instance de FDR donnée. La variable Signal peut être un terme Prolog quelconque. Lors de l'envoi d'un signal, le système se charge automatiquement d'ajouter le signal à transmettre dans la file des messages en attente de traitement et de le convoyer aux bons FDR.

Si le mécanisme de base est simple à mettre en place de nombreuses difficultés se posent quant à son utilisation pratique et à ses implications potentielles. Le premier problème qui se pose est celui qui consiste à déterminer à quelles instances de FDR il convient d'envoyer un signal (un broadcast systématique de tous les signaux serait trop brutal). Pour reprendre l'exemple précédent, si une tâche urgente apparaît dans *travail* il faut être capable de détecter un éventuel conflit, de déterminer les contextes qui sont affectés par ce nouvel évènement et éventuellement être capable de mettre en place une négociation entre ces différents contextes afin de déterminer par exemple que le contexte professionnel est prioritaire sur les autres FDR qui doivent replanifier. On peut proposer un mécanisme où les données mémoires seraient marquées automatiquement par les instances de FDR qui les ont posées ou modifiées. Ainsi, lors d'un conflit sur une donnée il serait plus facile de déterminer avec quelles instances communiquer. Toutefois, la donnée, source du conflit, peut très bien impacter un FDR qui n'a jamais agit sur celle-ci. Dès lors, on peut penser à un système dans lequel les FDR pourraient s'abonner pour recevoir des signaux lors de modifications ou accès à des zones déterminées.

L'introduction des signaux risque de complexifier significativement l'écriture des FDR. Ils sont en effet susceptibles de survenir dans tous les états des FDR et il risque d'être délicat de les gérer proprement. On peut toutefois penser que les mécanismes des FDR et notamment la possibilité de pouvoir définir des actions associées à la survenance d'un évènement précis quelque soit l'état dans lequel on se trouve [3] permettront de gérer convenablement cette difficulté.

Finalement le système de signaux internes proposé trouve à s'appliquer dans bons nombres de situations pratiques (un exemple d'utilisation est fourni ci-après dans la partie consacrée aux artifacts) ce qui semble justifier son intégration. Toutefois, si l'exemple des vacances peut être vu comme un partage de ressources entre FDR, il peut être également un problème (plus général) de gestion de la cohérence de la mémoire de l'agent. Il conviendra d'étudier plus en avant les problèmes de cohérence de mémoire, les questions relatives à la détection des conflits et aux mécanismes à mettre en place pour assurer une médiation efficace entre les contextes en conflit.

D - De la question du temps

Comme dans notre vie quotidienne, le temps joue un rôle crucial dans les systèmes de mission. Il convient donc d'y consacrer une attention toute particulière. Le lecteur attentif n'aura pas manqué de constater plusieurs problèmes relatifs à la gestion du temps dans le modèle. Cette partie leur sera consacrée.

L'un des points fondamentaux que nous avons passé sous silence concerne la durée des actions. Nous nous sommes tout juste contentés de mentionner que les exigences temps réel de notre modèle étaient relativement souples. Il faut bien avoir présent à l'esprit qu'en l'état chaque agent est constitué d'un unique processus. Il est donc clair que si les actions requièrent d'importants calculs et sont par conséquent coûteuses en temps (voire potentiellement infinies), non seulement l'agent déclenchera les *timeouts* avec du retard mais en plus il ne sera pas à même de traiter les messages qu'il a reçu durant cette période.

Ce problème a été identifié de longue date puisqu'on en décèle déjà implicitement la trace dans les travaux de Shoham sur Agent-0 [Sho93] pour lequel il impose que la durée de chaque cycle soit inférieure à une constante de temps posée nommée *time grain*. Une autre approche consiste à déclencher les actions dans des *threads* séparés et l'on obtient alors des agents constitués d'un nombre important de *threads* avec les risques d'interblocage que cela comporte et la lourdeur des mécanismes nécessaires au maintien de la cohérence des données.

Les travaux de Cédric Dinont s'intéressent à cette question et proposent d'adopter des entités de premier plan que les agents peuvent utiliser comme outils de calcul : les *artifacts computationnels*

[3] il suffit de mettre une variable pour l'état courant dans la définition de la transition

[DDMT06b, DDMT06a]. Le concept d'artifact a été introduit par une équipe italienne dans le cadre de ses travaux sur les mécanismes de coordination dans les systèmes multi-agents [ORV+04, MV05]. Si un agent est vu comme un système dirigé ou orienté par des buts, un artifact est une entité qu'un agent utilise pour atteindre ses buts. Un artifact est caractérisé par une interface d'usage (ensemble d'opérations de deux types : exécution d'une action ou perception de la terminaison d'une action), par un ensemble d'instructions opératoires (*Operating Instructions* correspondant à la description formelle de la manière dont les agents peuvent utiliser l'artifact), sa fonction (description du service rendu par l'artifact) et une spécification de son comportement. Les instructions opératoires sont exprimées à l'aide d'un formalisme à base d'algèbre de processus. Dès lors, sans formaliser plus rigoureusement ce principe pour l'instant, le programmeur est invité à reléguer les calculs « longs » à des entités externes. L'agent a le contrôle sur ces entités et il peut les interrompre et les relancer à loisir en fonction du contexte.

Un autre point contestable est celui de la gestion des *timeout*. Le *timeout* est actuellement géré comme un message particulier ce qui n'est conceptuellement pas très correct. D'autre part, en pratique, son usage a été souvent détourné de son sens premier. Étudions donc brièvement les différents cas d'utilisation du *timeout* dans les applications actuelles :

- exécuter une action alternative lorsqu'aucun évènement n'a été reçu dans le délai imparti. Typiquement on se met en attente de messages émanant d'autres agents et l'on veut pouvoir effectuer des actions si l'on ne reçoit rien dans un délai raisonnable, cela permet d'améliorer la tolérance aux fautes puisque l'agent va pouvoir tenter de s'adapter à un éventuel problème révélé par la non réception d'un message attendu. Cela correspond à l'utilisation classique du *timeout*,
- faire une action dans x secondes. Il s'agit là de planifier l'exécution d'une action donnée ou d'une transition d'état au bout d'un délai d'attente. On exhibe ainsi une forme limitée de proactivité, l'agent posant une action qui n'est pas motivée par un message émanant de l'extérieur,
- faire une action toutes les x secondes (bouclage sur le même état), c'est par exemple ainsi que sont fait les raffraichissements des interfaces graphiques pour l'instant ce qui semble perfectible.

Nous pensons qu'il conviendrait d'étudier la mise en place d'un service de réveil dans les FDR en complément du *timeout*. Les *timeouts* ne devant plus être utilisés que dans leur acception première, à savoir la définition d'un temps d'attente maximum pour la survenance d'un évènement avant d'adopter un comportement alternatif. Une solution possible qui a été utilisée dans certaines applications du service consiste à utiliser un processus externe (potentiellement un artifact) servant d'horloge. Les agents utilisent alors cette horloge par le biais de messages pour lui demander l'heure ou encore pour lui demander de lui envoyer des messages à une heure donnée. Cette approche apporte une certaine souplesse dans le sens où elle permet d'accélér ou de ralentir le temps en intervenant sur l'horloge et assure que l'ensemble des agents seront synchronisés. Elle pose toutefois le problème de la lourdeur des échanges de message et d'une centralisation excessive au niveau de cette entitée. On peut également envisager de dédier un thread dans chaque agent assurant le même service. On peut enfin penser ajouter des routines permettant de déclencher des évènements temporels similaires à ceux que nous venons de décrire dans le modèle des FDR. Le modèle d'exécution devrait alors tenir compte des appels de ces routines dans la gestion des *timeout*. En ce sens, la notion interne de *timeout* semble primitive et paraît permettre d'exprimer tout ce que l'on peut vouloir faire sur le temps dans nos applications. Il conviendra de s'en convaincre en étudiant plus précisément les travaux qui ont été menés en logique temporelle.

Plus précisément ces routines accessibles dans les parties actions des transitions devraient permettre de spécifier des réveils en temps absolu et relatif et notamment permettre d'exprimer ce genre de comportement :

- Déclenchement d'une action à 19h30 (timeout = 19h30 - now)
- Déclenchement d'une action après 19h30 (timeout $>$ 19h30 - now)
- Déclenchement d'une action avant 19h30 (timeout $<$ 19h30 - now)
- Déclenchement d'une action dans 2s (timeout = 2)
- Déclenchement d'une action toutes les 10s pendant une heure

Dans la pratique, le besoin s'est fait également sentir d'une autre forme de *timeout* qui ne soit pas réinitialisé lorsque l'agent reçoit un message. Ainsi, on peut vouloir lancer un appel d'offre assorti d'une *deadline*, par exemple de 10 secondes. On aurait alors un état avec un timeout de 10 secondes mais qui ne se réinitialiserait pas à chaque réception de messages. Bien évidemment on peut déjà le faire avec le *timeout* actuel moyennant une petite manipulation du programmeur qui doit se charger de remettre à jour le *timeout* sur la réception des messages.

Nous estimons enfin qu'il est nécessaire de dater tous les évènements reçus afin de pouvoir mettre en place des raisonnements plus évolués sur le temps.

Par exemple, décider que tous les évènements d'un type donné datant de plus de x secondes peuvent être ignorés.

E - De l'intégration des artifacts

Nous pensons que les artifacts joueront un rôle important dans le cadre de nos applications futures. Des travaux sont donc actuellement en cours dans le service pour faciliter l'intégration des artifacts dans nos systèmes. Ils font un usage intensif des FDR et il semblerait que le modèle puisse ouvrir de nouvelles perspectives pour l'utilisation pratique des artifacts par les agents. L'idée est sommairement la suivante. Lorsqu'un agent veut utiliser un artifact il accède aux *Operating Instructions (OI)* (Cf. paragraphe précédent). L'agent convertit automatiquement cet *OI* sous forme de FDR qui décrit l'ensemble des intéractions que l'agent peut potentiellement avoir avec l'artifact. Un autre FDR « pilote » ce dernier par l'envoi de signaux internes ce qui permet de faire avancer le FDR correspondant à l'*OI*, déroulant partiellement le mode d'emploi en fonction des choix faits par l'agent. Le programmeur implémente ces choix ainsi que les traitements à effectuer en fonction des réponses de l'artifact. De plus amples informations sur la question peuvent être trouvées dans le rapport de Paul-Edouard Marson [Mar06] (à paraître).

F - De la visualisation des FDR

Le travail que nous venons de mentionner concernant les artifacts repose en partie sur une génération automatique de FDR. Bien que le langage utilisé pour décrire les FDR soit relativement lisible il aurait été fastidieux de comparer les résultats attendus aux résultats générés en lisant le code clause par clause. Un outil de visualisation des FDR a donc été développé en Prolog pour permettre d'afficher directement certain ou tous les FDR (Fig. 3) en mémoire ou de les exporter en fichier png ou dot. L'ensemble fait appel au projet de visualisation de graphes *Graphviz*.

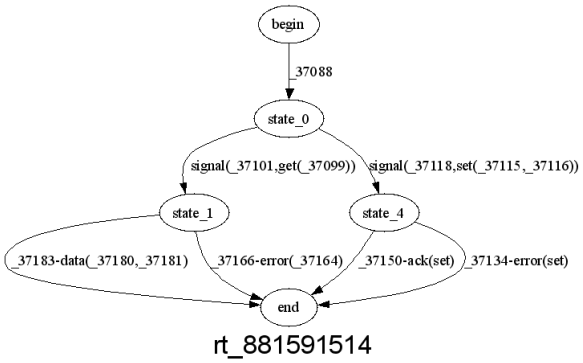


FIG. 3 – Exemple de visualisation

L'opération inverse consistant à dessiner graphiquement l'automate des différents FDR et à générer le code associé est également envisagé et pourrait contribuer à accélérer le développement du comportement des agents.

G - De la tolérance aux fautes

La prise en compte, au niveau du formalisme, de la tolérance aux fautes, pourrait être un aspect intéressant à mettre en oeuvre.

Dans cette optique, il serait souhaitable de pouvoir faciliter l'écriture de protocoles plus avancés et notamment de pouvoir rajouter une notion de nombre d'occurences. Par exemple, de pouvoir spécifier qu'on accepte de pouvoir boucler dans un état au plus n fois sans avoir à laisser au programmeur le soin de gérer lui-même des compteurs internes. Il serait alors moins fastidieux de définir des protocoles du type : l'agent fait une demande à un autre agent, attend une réponse pendant un temps défini et relance l'agent en question, si au bout de cinq relances l'agent ne répond pas alors on adopte une stratégie alternative.

Il convient de faciliter la gestion des erreurs. Nous proposons de développer le mécanisme de gestion d'exceptions suivant. Imaginons que l'on ait défini que dans tous les états d'un FDR on puisse recevoir le message d'erreur $error$ et que cette erreur soit de nature à empécher complètement le déroulement du FDR en question. Imaginons également que le traitement de cette erreur appelle un comportement précis défini sous forme d'automate. Il serait appréciable de pouvoir raccrocher l'automate de reprise d'erreurs au niveau de l'état où le message d'erreur a été reçu. Cet automate pourrait alors rajouter ses règles de grammaire propres. Il se lancerait dans son protocole de reprise d'erreurs et serait également chargé de sauvegarder tous les messages qui ne le concernent pas mais qui étaient destinés au FDR initial. Lorsque le traitement d'erreur s'achève, l'automate enlève ses règles spécifiques et rend la main à l'état qui a reçu l'erreur en lui fournissant les messages sauvegardés.

Le besoin s'est également fait ressentir de réinitialiser un FDR lors de la survenance d'un évènement. Il pourrait donc être intéressant de prévoir une fonctionnalité intégrée permettant de réinitialiser un FDR ce qui reviendrait en pratique à nettoyer la mémoire locale et à repartir de l'état $begin$.

H - Des actions

Les actions constituent les compétences à proprement parler de l'agent. Elles sont appelées lors des transitions d'états et prennent la forme de prédicats Prolog. Pour y voir plus clair, recensons les différentes procédures pouvant être appelées dans les actions :

– code Prolog
– appels encapsulés à ALBA
  envoi de messages aux agents
  invocation d'artifacts
  création d'agents
  migration
– appels relatifs aux FDR
  instanciation de nouveaux FDR
  modification, ajout, suppression de patrons de FDR
  modification dynamique de FDR instanciés
  émission de signaux internes
  modification de la mémoire locale du FDR et globale de l'agent
  opérations de test et de *matching* sur la mémoire locale et globale pour récupérer des valeurs

Nous pensons qu'il est souhaitable que les actions constituant un effet de bord (envois de messages, création d'agents, ...) soient appliquées à la fin juste avant le basculement dans le nouvel état tandis que la migration sera toujours effectuée en dernier après le basculement d'état pour assurer la reprise transparente de l'exécution du comportement de l'agent. La centralisation de ces traitements devrait permettre d'analyser plus facilement les conséquences des actions et de raisonner dessus. D'autre part lors d'un échec dans une action aucun retour arrière (*backtrack*) n'a été prévu à la base car les effets de bord tels que des envois de messages ne peuvent évidemment être défaits. Avec cette approche, si un échec survient dans les traitements avant les effets de bord il est possible de revenir en arrière et d'essayer d'autres règles sans craindre de conséquences irréversibles liées aux effets de bord. Quant au traitement des échecs liés aux actions sur l'environnement ou dans le cadre des intéractions de l'agent avec les autres agents elles doivent être prises en compte dans la définition même des FDR.

I - Divers

Nous regroupons ici diverses autres modifications dont l'importance ne justifiait pas à elle seule l'attribution d'une partie entière.

Comme cela a été mentionné, par défaut, le système traite les messages dans l'ordre d'arrivée suivant une structure de file, l'utilisateur peut modifier ce comportement en définissant des stratégies de filtrage. Cela peut s'avérer utile pour donner la précédence à certains agents, pour traiter certains messages en priorité ou encore pour ignorer les messages non pertinents lorsque l'agent est surchargé ce qui lui permet d'exhiber un comportement adaptatif. Dans la version précédente du prédicat *filtering_strategy*, seule la file contenant l'ensemble des couples *(Message, Sender)* en attente de traitement est disponible. Il convient de transmettre l'état actuel des FDR et des mémoires afin de permettre des stratégies de filtrage plus fines et mieux adaptées au contexte. Il est également nécessaire d'étendre le prédicat pour qu'il ne traite non plus uniquement les messages mais des évènements tels que décrit ci-après.

Il serait souhaitable de favoriser la modularité et la réutilisation des FDR et notamment de s'inspirer de certains principes de la programmation orientée objet. Il serait ainsi utile que les FDR deviennent des composants indépendants et réutilisables. En l'état, nous avons des agents qui définissent différents FDR sans réelle structure avec d'un côté la définition des transitions et de l'autre le code des actions qui sont appelées. Ces définitions sont souvent très similaires et sont copiées dans les différents agents. Par indépendant, nous entendons qu'il serait bon que la définition des transitions et des actions forme un ensemble cohérent. Cela permettrait de composer un agent à partir de ces différentes unités logicielles tout en permettant à l'agent de surcharger certaines actions. Cela autoriserait également les agents à se transmettre des compétences en s'échangeant par messages ces blocs comprenant à la fois la définition des actions et la définition des transitions.

Il n'est pas non plus exclu d'introduire une notion d'héritage dans les FDR afin de pouvoir les spécifier itérativement à la manière de ce que propose AgenTalk. Pour favoriser la réutilisation, nous envisageons également de pouvoir rendre ces unités logicielles paramétriques. Cela favoriserait la description de protocoles génériques, par exemple cela permettrait de définir un protocole du type « exécuter A fois B, attendre C secondes, faire D, si je n'ai reçu aucun message pendant E secondes, ... » et lorsqu'on instancie le FDR on peut lui fournir les paramètres A, B, C, D, E que l'on souhaite ou utiliser les valeurs par défaut.

J - Vers un nouveau modèle

1. Modification de la structure des transitions Nous avons vu que les transitions sont de la forme condition-action et que chaque transition définit un ensemble d'états accessibles depuis un état. Dès lors, c'est dans le code de l'action que le programmeur détermine le prochain état dans lequel basculer. Avec cette approche un FDR peut être représenté par un graphe bipartite. L'utilisation pratique de cette approche a révélé que dans la majorité des cas la décision se faisait dès le début de l'action et que le code associé à chaque branchement d'état était bien séparé. Nous proposons donc de simplifier cette structure en n'autorisant pour chaque transition qu'un seul état de sortie, les conditions gouvernant les choix étant intégrées dans le filtre ou dans la partie condition des règles. Les FDR peuvent alors

être représentés par des graphes simples donc les arcs symbolisent une condition associée à une action (Fig. 4). Cette décision permet également de faciliter l'analyse des transitions et donc augmente les possibilités d'introspection.

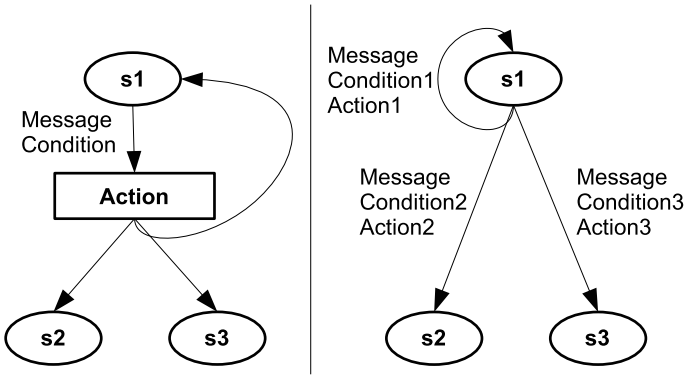


FIG. 4 – Structure des transitions

2. De l'organisation des mémoires Dans le modèle initial aucune contrainte n'était imposée quant à la représentation des différentes mémoires. La nouvelle version va passer à un système de type attribut-valeur où la valeur restera un terme Prolog quelconque. Cela restreint assez peu l'utilisateur, permettra d'accéder rapidement aux valeurs par le biais de tables de hachage et permettra de mettre en place le support des évènements mémoires décrits ci-après.

Nous ajoutons également les prédicats suivants permettant de manipuler les mémoires :

get_local(Key, Value) : permettant d'accéder à la valeur associée à la clé Key de la mémoire locale du FDR
set_local(Key, Value) : permettant de créer ou modifier la valeur associée à la clé de la mémoire locale du FDR

get_global(Key, Value) : équivalent pour la mémoire globale de l'agent
set_global(Key, Value) : équivalent pour la mémoire globale de l'agent

Cette spécification sera implémentée sous forme de variables *mutables*, i.e. à assignations multiples et non persistantes au *backtrack* de Prolog ce qui permet de tirer parti du mécanisme de retour arrière offert par le langage.

3. Introduction de la notion d'évènement Les FDR présentés dans la section 2 se déclenchaient sur l'arrivée de messages et sur les *timeout*. Forts des éléments qui ont été présentés ici, nous proposons d'étendre cela par l'introduction de la notion d'évènement.

Les règles de transition deviennent alors :

fdr(Type, State-in, State-out, Event, PARMFDR, Action) :- [Filter].

Voici la liste des différents évènements possibles :

Liste des évènements

msg(Message, Sender) : réception d'un message s'unifiant avec Message provenant de Sender. Cet évènement caractérise la communication entre les agents ou entre agent et artifacts. Les messages échangés entre agents prennent la forme d'actes de langage de type FIPA-ACL ou KQML ce qui les distingue des messages échangés avec les artifacts. A terme il sera peut-être judicieux de ne pas employer l'atome **msg** pour les communications avec les artifacts mais d'introduire un autre atome pour marquer la séparation entre les deux.

signal(Signal, FDR_ID) : réception d'un signal interne Signal émanant du FDR FDR_ID

timeout : le temps d'attente alloué est dépassé.

+LKL(K, V) : ajout d'une valeur V de clé K dans la mémoire locale, il est possible d'instancier V, de l'instancier partiellement ou de ne pas l'instancier du tout pour en récupérer la valeur.

-LKL(K, V) : suppression de la mémoire de la valeur V de clé K.

LKL(K, V1, V2) : modification du champs de la clé K de la valeur V1 en V2.

+GKL(K,V) : équivalent au niveau de la mémoire globale.

-GKL(K,V) : équivalent au niveau de la mémoire globale.

GKL(K,V1, V2) : équivalent au niveau de la mémoire globale.

L'introduction des évènements portant sur les modifications de la mémoire posent toutefois des problèmes réels d'implémentation et pourrait gravement compromettre la réactivité du système. Nous proposons de stocker dans une table de hachage les clés des évènements mémoires à surveiller associés aux identifiants des FDR intéressés. Lors des modifications mémoires, les évènements appropriés sont générés automatiquement en utilisant la table et dirigés vers les FDR qui conviennent.

L'introduction des évènements mémoires posent d'autres problèmes qui devraient conduire à une remise en question du modèle d'exécution. Nous avions vu qu'à chaque cycle l'ensemble des FDR susceptibles de recevoir un message donné étaient

déclenchées. Lorsqu'un évènement mémoire survient et que plusieurs FDR sont en attente de cet évènement, l'exécution de l'ensemble des FDR pourraient poser des difficultés. Par exemple, le premier FDR déclenché pourrait remodifier la mémoire rendant l'évènement caduque et l'exécution des autres FDR potentiellement inappropriée. Cela engendre une forme d'indéterminisme et il est nécessaire d'assurer que l'évènement est valable lorsqu'un FDR est activé. On comprend bien qu'avec des évènements uniquement liés aux messages la question ne se posait pas vraiment.

K - Implémentation pratique du modèle

1. La programmation logique structurée [Tai93] Pour l'implémentation pratique du modèle d'agent que nous avons décrit nous envisageons de recourir à la programmation logique structurée. Nous allons donc brièvement décrire cette approche avant de motiver ce choix par les bénéfices qu'elle pourrait apporter dans le contexte qui est le notre.

Classiquement, un programme Prolog peut être vu comme un ensemble de formules logiques (clauses de HORN) décrivant le monde sur lequel opère le programme, l'exécution consistant à démontrer un théorème. Les programmeurs lui ont souvent préféré une vision plus procédurale où chaque élément du langage est un arbre et l'opération de base est l'unification couplée à un mécanisme de recherche en profondeur d'abord. De part sa vision plus procédurale, elle a favorisé une pratique, désignée par effet de bord, consistant à développer des programmes opérant des ajouts et/ou retraits dynamiques de clauses ou utilisant des extensions résistantes au retour arrière. Les effets de bord sont généralement utilisés par besoin de persistance par rapport au mécanisme naturel de retour arrière de Prolog, par le besoin de mises à jour successives de valeur ou encore par le besoin rapide d'accès aux données. Cette pratique ne va toutefois pas sans quelques inconvénients dont notamment : la résistance au retour arrière qui, bien que recherchée, conduit à faire perdre à Prolog sa simplicité sémantique, la perte des liens de variables (nous y reviendrons), l'inefficacité due à la relative lenteur des opérations d'ajout/retrait de clauses. C'est pour pallier ces difficultés que fut proposée la programmation en Prolog sans effet de bord.

La programmation logique structurée repose sur quelques principes dont voici les principaux. Il convient de privilégier la représentation par arbre rationnel (un unique terme de la pile globale) plutôt que par clauses. Si des données doivent être liées entre elles, ne pas représenter les liens par des constantes mais par des variables. Si des données sont modifiables, ne pas les représenter par des variables Prolog mais par des instances du type variable *mutable*. Si un accès rapide à des données est nécessaire, ne pas recourir au mécanisme des clauses mais programmer un mécanisme efficace d'indexation des données reposant sur des tables de hachage. Cette approche permet de maintenir les programmes dans l'un des modèles théoriques de Prolog (pur Prolog plus *cut*) et facilite considérablement la représentation des données et leur manipulation.

2. Application de la programmation logique structurée au modèle Comme nous l'avons vu, chaque FDR décrit un patron de conversation sous la forme d'un ensemble de clauses, chaque clause décrivant une transition. L'interpréteur stocke les mémoires et l'état courant des FDR instanciés et se sert des clauses de définition pour faire avancer les automates. Dès lors, la modification dynamique de la description d'un FDR impacte l'ensemble des FDR instanciés et peut même s'avérer dangereuse. A titre simplement illustratif, imaginons qu'une instruction retire toutes les clauses relatives à un état λ d'un FDR de type donné (il s'agit un peu d'un exemple jouet mais suite à la découverte d'un dysfonctionnement dans l'état λ un mécanisme de contrôle de l'agent pourrait vouloir empêcher l'agent de tomber dans cet état par le biais de ce procédé) et que plusieurs FDR de ce type soient instanciés. Il se peut que certains des FDR instanciés soient dans l'état λ et y restent alors définitivement bloqués. L'utilisation de clauses distinctes ne nous permet en outre pas de faire des liens entre les variables utilisées. Cette approche présente donc plusieurs limitations pouvant être écartées par une utilisation judicieuse de la programmation logique structurée.

Nous proposons donc de décrire chaque FDR sous la forme d'un terme unique ce qui nous permettra de pouvoir exploiter les liens entre les variables. On peut légitimement se demander ce que pourrait concrètement apporter la possibilité d'avoir accès aux liens de variable. Pour s'en faire une meilleure idée étudions un exemple précis, supposons que nous voulions décrire un protocole de communication selon le schéma suivant : envoi d'un appel d'offre, puis poursuite de la conversation avec le premier agent qui répond. Avec les FDR sous forme de clauses, il est possible d'assurer ce mécanisme mais il est nécessaire de stocker en mémoire le nom du premier agent qui répond, puis de vérifier à chaque étape que l'expéditeur du message reçu est celui attendu, ou encore de modifier les règles de grammaire de filtrage des messages avec le nom de l'agent qui a répondu en premier. Cela reste fastidieux. En utilisant les liens de variables, il suffit d'utiliser la même variable pour les différentes transitions et dès l'instanciation de la variable en question l'ensemble des règles de transition se modifieront. Les liens permettent également

de décrire des comportements plus sophistiqués. Supposons que nous ayons deux transitions partiellement instanciées partageant le même état futur encore inconnu X. L'instanciation dynamique d'un bout de plan sur X sera automatiquement effective dans les deux transitions.

L'interpréteur manipulera également un seul terme Prolog ce qui va permettre d'augmenter la flexibilité générale et de gagner certaines fonctionnalités par le biais de l'unification sans que les performances en patissent grâce au système d'indexation. A chaque fois qu'un FDR sera instancié, l'interpréteur en copiera la structure dans ses données de travail, il ne se contentera donc plus de gérer la mémoire et l'état courant tout en lisant une définition commune et partagée de la structure d'un FDR. Chaque contexte aura une copie de la structure (avec des liens de variables locaux) et pourra travailler dessus de manière indépendante sans affecter les FDR du même type. Il sera également plus facile d'attacher des données modifiables aux différents FDR ce qui s'avèrera utile pour annoter les plans tels que nous l'avons proposé dans la section s'intéressant à la proactivité des agents.

Nous pensons développer un langage intermédiaire qui tirera profit de cette nouvelle approche et qui sera manipulé par l'intepréteur du modèle. L'interpréteur du modèle travaillera sur un terme de la pile globale. À terme, nous pensons développer un langage de plus haut niveau qui sera traduit dans ce langage intermédiaire lui-même exécuté par l'interpréteur.

V. Exemple & Applications

A - Un exemple

Un exemple très simple est présenté à la figure 5, le code associé permet d'avoir une idée plus précise de l'utilisation pratique des FDR. On remarque que, grâce à l'utilisation de variables pour les états, il est très simple de spécifier la fin de l'agent quelque soit son état ou encore de faire des règles appelant un comportement par défaut lors de la réception de messages inconnus, et ce, quelque soit l'état courant de l'agent.

B - Applications

Dans un contexte industriel, l'étude des thèmes de travail qui ont été présentés ne peut se concevoir que si elle est guidée par des besoins applicatifs concrets. Ceux-ci permettent non seulement d'identifier les points sur lesquels il y a lieu de faire porter l'effort de recherche mais également de fournir la base sur laquelle les résultats obtenus peuvent être validés, permettant ainsi d'aller bien au delà du classique « cas d'école ». Les FDR ont donc été employés dans plusieurs applications faisant intervenir des systèmes multi-agents. Ils se sont avérés d'une utilité précieuse pour le développement du comportement des agents. Nous allons décrire brièvement quelques unes d'entre elles afin de mieux cerner les domaines pour lesquels ce modèle a déjà pu être utilisé avec succès.

Interloc (Fig. 6) est une application illustrant la localisation passive de cibles marines utilisant les techniques de propagation de contraintes sur intervalles. Dans Interloc des avions cherchent à détecter des bateaux sans utiliser leur radar mais en exploitant les émissions des cibles pour en déduire leur position. Un nombre substanciel d'agents (un agent par bateau et par avion, un agent de gestion de l'interface graphique, un agent de mesure ainsi que des artifacts computationnels) est amené à intervenir et intéragir en même temps ce qui a été grandement facilité par les FDR. Cette application sert également pour expérimenter et démontrer les différentes techniques que nous étudions et en particulier les capacités de tolérance aux fautes apportées par l'approche multi-agent.

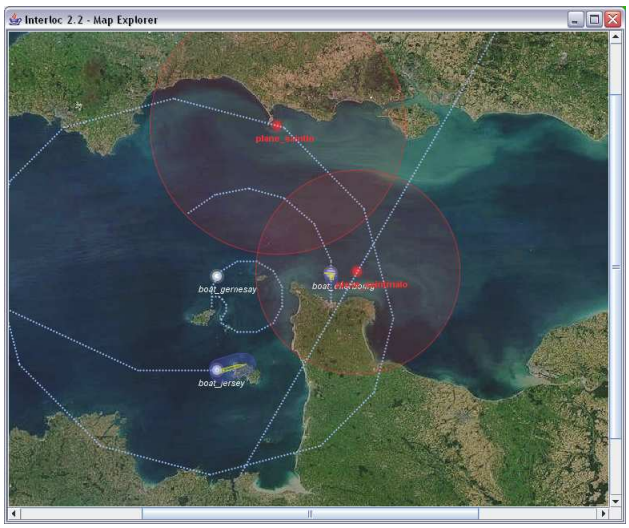
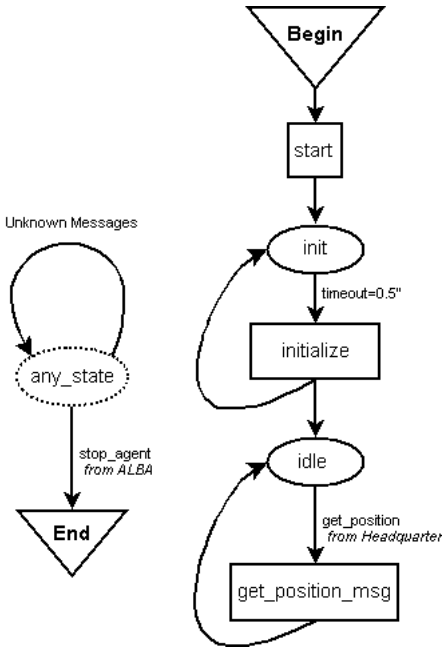


Fig. 6 – Interface d'Interloc

Synapse (Fig. 7) est une application illustrant la coordination de drones. Elle fait intervenir plusieurs drones ayant des caractéristiques différentes (vitesse, portée radar, capacité d'observation, ...). Ces drones vont recevoir un appel d'offre d'observation d'une zone et y répondre en fonction de leur agenda et de leurs caractéristiques. Le quartier général convient alors d'attribuer la mission à deux drones qui doivent se synchroniser l'un reste en zone protégé et utilise son radar pour transmettre les coordonnées précises de la cible à l'autre drone qui a coupé son radar pour pénétrer en zone ennemie. Les FDR et la forte produc-




```
fdr_timeout(main, init, 0.5).
fdr(main, begin , init          , _Message       , _Sender    , _ParamFDR, start).
fdr(main, init  , [init,idle]   , timeout        , _Sender    , _ParamFDR, initialize).
fdr(main, idle  , idle          , get_position(_), headquarter, _ParamFDR, get_position_msg).
fdr(main, _State, end           , stop_agent     , alba       , _ParamFDR, stop).
fdr(main, _State, _State        , _Message       , _Sender    , _ParamFDR, default).
```

FIG. 5 – Exemple

tivité de Prolog ont permis de développer un prototype de cette application en un temps très réduit. Les FDR se sont également montrés bien adaptés pour la mise en place du protocole Contract Net.

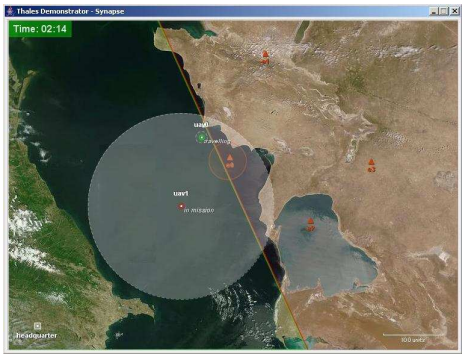


FIG. 7 – Interface de Synapse

Aerial (Fig. 8) est une application travaillant dans la continuité de Synapse mais qui fait intervenir Airplan (un planificateur de trajectoire) pour le calcul des trajectoires des drones. Cela permet notamment d'avoir des drones qui répondent intelligemment aux appels d'offre en planifiant leurs trajectoires tenant compte d'obstacles fixes et mobiles ainsi que du vent.

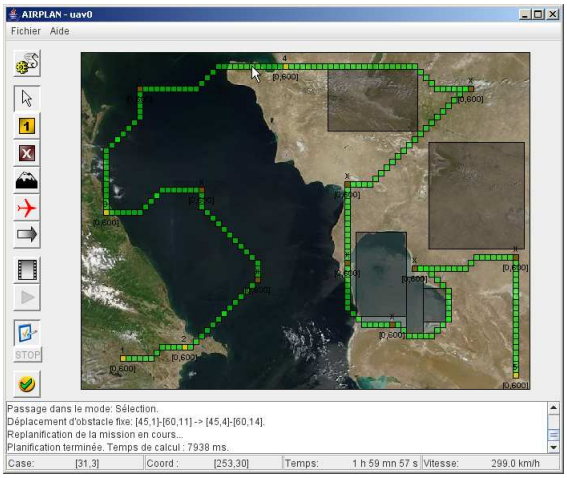


FIG. 8 – Interface d'Aerial

VI. ÉVALUATION CRITIQUE

À titre personnel ce stage a été riche d'enseignements. Il me semble avoir parfaitement rempli son office de formation à la recherche puisque j'ai eu l'occasion d'écrire un article pour ProMAS et de le présenter. Il m'a également permis de mieux appréhender les difficultés inhérentes à un travail de recherche avec le lot de frustations et de remises en question que cela comporte. Cela fut par exemple

le cas lors de la rédaction de l'état de l'art et de la découverte des travaux très similaires aux FDR, mais il me semble toutefois que cela fait également partie du travail de chercheur que d'être capable de se positionner dans l'existant et d'admettre avec humilité que ses idées ont déjà été proposées. Cela constitue une étape nécessaire permettant de mieux prendre du recul, d'identifier les limitations des travaux précédents et de s'efforcer de proposer des améliorations pour s'en affranchir.

Toutefois, bien que beaucoup d'idées aient été brassées, nous regrettons de ne pas avoir pu les creuser plus en avant ni les mettre en oeuvre dans une implémentation pratique, le temps nous ayant manqué, suite principalement au travail bibliographique, à la rédaction de l'état de l'art ainsi qu'à la préparation du voyage au japon et à la semaine passée sur place. Cela est d'autant plus regrettable qu'une implémentation pratique permet généralement d'identifier de nouvelles problématiques et alimente par là même le travail de recherche. La situation était toutefois un peu moins problématique dans le sens où l'implémentation actuelle des FDR fonctionne bien et permet déjà de faire beaucoup de choses. Il nous a semblé préférable plutôt que de mettre en oeuvre, cahin caha, quelques propositions dans le modèle actuel de tirer parti de l'expérience acquise pour poser les bases d'un nouveau modèle plus complet qui sera intégralement réimplémenté. Nous avons toutefois commencé à implémenter certaines propositions et envisageons de mettre à profit le dernier mois de stage pour compléter l'intégration de :

- la gestion des signaux
- la représentation des mémoires et les prédicats de manipulation des mémoires
- l'encapsulation de toutes les fonctionnalités ALBA qui n'ont pas été reprises dans les FDR (migration, création d'agents, etc.)
- une simplification d'écriture pour la gestion des messages dans les actions
- les facilités de réinitialisation des FDR
- la correction d'un bug dans la gestion actuelle de la migration qui empêche une reprise parfaite de l'exécution lorsque plusieurs FDR sont déclenchés

VII. Perspectives & Conclusions

À la lecture de la section 4 il est clair que les perspectives ne manquent pas. Il conviendra tout d'abord d'affiner les idées qui ont été proposées, de les formaliser plus rigoureusement et de les implémenter dans un système opérationnel. Nous espérons ainsi à terme pouvoir proposer un cadre conceptuel pour des agents cognitifs capables de prendre en compte simultanément les aspects cognitifs pour l'implémentation des fonctions intelligentes mais également les aspects réactifs - ou temps réel - pour l'intégration des capteurs et de manière plus générale de l'environnement.

Nous envisageons également de donner corps à ces idées par le développement d'un langage de haut niveau, dont la syntaxe et la sémantique opérationnelle sont à définir, permettant de décrire simplement le comportement des agents cognitifs. À cet égard, il n'est d'ailleurs pas exclu qu'un effort soit consenti dans la mise en place d'un environnement complet de développement permettant de faciliter considérablement l'implémentation des agents et intégrant notamment les aspects suivants : définition partielle du comportement des agents par le biais d'outils graphiques, génération automatique de code, éditeur adapté au langage, déploiement de systèmes multi-agents, outils de débuggage.

De manière plus générale, un travail important reste à accomplir pour se doter des outils nécessaires à l'expérimentation en vraie grandeur de systèmes de missions évolués. Il conviendra donc de s'appuyer sur l'expérience qui a été acquise pour concevoir et développer l'outillage (refonte d'ALBA pour la couche basse des agents, refonte des FDR, développement du langage de description des agents) nécessaire au déploiement des agents et de valider ces développement par l'expérimentation dans un environnement de simulation technico-opérationnelle voire même à terme d'effectuer une expérimentation en vol appliquée aux avions de surveillance. Tous ces développements devront se faire en gardant à l'esprit notre objectif central qui est de faciliter la conception et le développement de systèmes plus intelligents, moins complexes et plus tolérants.

En somme, il y a largement de quoi s'occuper quelques années et c'est ainsi que des démarches dans le sens d'une thèse CIFRE ont été initées au sein du groupe THALES. Nous espérons donc pouvoir devenir l'un des artisans de la concrétisation de ces ambitions et pouvoir apporter notre modeste contribution au succès des systèmes multi-agents dans le monde industriel.

VIII. Remerciements

Je tiens à remercier chaleureusement Monsieur Patrick TAILLIBERT pour la confiance qu'il m'a accordé et pour m'avoir accompagné dans mon périple japonais ainsi que l'ensemble de l'équipe pour l'excellente ambiance de travail qui a régné durant ce stage.

IX. Réféfences

*

[ABLP02] J. Alferes, A. Brogi, J. Leite, and L. Pereira. Evolving logic programs, 2002.

[BBCP05] Fabio Bellifemine, Federico Bergenti, Giovanni Caire, and Agostino Poggi. *JADE - A Java Agent Development Framework*, chapter 5. Bordini et al. [BDDFS05], 2005.

[BBD+06] Rafael Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge Gomez-Sanz, Joao Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. In *Informatica 30*, pages 33–44, 2006.

[BBdOJ+02] Rafael H. Bordini, Ana L.C. Bazzan, Rafael de O. Jannone, Daniel M. Basso, Rosa M. Vicari, and Victor R. Lesser. Agentspeak(xl) : Efficient intention selection in bdi agents via decision-theoretic task scheduling, 2002.

[BD94] S. Bussmann and Y. Demazeau. An agent model combining reactive and cognitive capabilities. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS-94)*, Munich, Germany, September 1994.

[BDDFS05] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni. *Multi-Agent Programming : Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer, 2005.

[BDKTV97] Frances M. T. Brazier, Barbara Dunin-Keplicz, Jan Treur, and Rineke Verbrugge. Modelling internal dynamic behaviour of BDI agents. In *ModelAge Workshop*, pages 36–56, 1997.

[BF95] M. Barbuceanu and M. S. Fox. Cool : A language for describing coordination in multiagent systems. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference oil Multi-Agent Systems (ICMAS-95)*, pages 17–24, San Francisco, CA, USA, 1995. AAAI Press.

[BF97] Mihai Barbuceanu and Mark S. Fox. Integrating communicative action, conversations and decision theory to coordinate agents. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 49–58, Marina del Rey, CA, USA, 1997. ACM Press.

[BFVW03] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifiable multi-agent programs. In *PROMAS*, pages 72–89, 2003.

[BHV05] R. H. Bordini, J.F. Hübner, and R. Vieira. *Jason and the golden fleece of agent-oriented programming*, chapter 1. Bordini et al. [BDDFS05], 2005.

[BIP91] Michael E. Bratman, David Israel, and Martha Pollack. Plans and resource-bounded practical reasoning. In Robert Cummins and John L. Pollock, editors, *Philosophy and AI : Essays at the Interface*, pages 1–22. The MIT Press, Cambridge, Massachusetts, 1991.

[BPLM04] Lars Braubach, Alexander Pokahr, Winfried Lamersdorf, and Daniel Moldt. Goal representation for bdi agent systems. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Second International Workshop on Programming Multiagent Systems : Languages and Tools*, pages 9–20, 7 2004.

[BR93] J. Blythe and W. Scott Reilly. Integrating reactive and deliberative planning for agents. Technical Report CMU-CS-93-155, Carnegie Mellon University, Pittsburgh, PA, USA, 1993.

[Bro86] R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1) :14–23, March 1986.

[CFST05] Caroline Chopinaud, Amal El Fallah-Seghrouchni, and Patrick Taillibert. Dynamic self-control of autonomous agents. In *PROMAS*, pages 41–56, 2005.

[Cho05] Caroline Chopinaud. Du problème de définir l'autonomie. Technical report, Université Pierre et Marie Curie, 2005.

[Col01] R.W. Collier. *Agent Factory : A Framework for the Engineering of Agent-Oriented Applications*. PhD thesis, University College Dublin, 2001.

[Cos02] S. Costantini. Meta-reasoning : a survey. In A. Kakas and F. Sadri, edi-

tors, *Computational Logic : From Logic Programming into the Future : Special volume in honour of Bob Kowalski*. Springer-Verlag, Berlin, 2002.

[CST05] Caroline Chopinaud, Amal El Fallah Seghrouchni, and Patrick Taillibert. Automatic generation of self-controlled autonomous agents. *iat*, 0 :755–758, 2005.

[CT02] S. Costantini and A. Tocchio. A logic programming language for multi-agent systems, 2002.

[CT04] Stefania Costantini and Arianna Tocchio. The dali logic programming agent-oriented language. In *JELIA*, pages 685–688, 2004.

[DCT06] Benjamin Devèze, Caroline Chopinaud, and Patrick Taillibert. Alba : a generic library for programming mobile agents with prolog. In R.H. Bordini, M. Dastani, J. Dix, and A.F. Segrouchni, editors, *Fourth international Workshop on Programming Multi-Agent Systems (PROMAS-2006)*, May 2006.

[DdBDM03] M. Dastani, F. de Boer, F. Dignum, and J. Meyer. Programming agent deliberation : An approach illustrated using the 3apl language, 2003.

[DDMT06a] C. Dinont, E. Druon, P. Mathieu, and P. Taillibert. Artifacts for time-aware agents. *AAMAS'06*, 2006.

[DDMT06b] C. Dinont, E. Druon, P. Mathieu, and P. Taillibert. Les artifacts de calcul - une solution aux délibérations longues. *JFSMA'06*, 2006.

[DE94] W. H. E. Davies and P. Edwards. Agent-K : An Integration of AOP and KQML. In T. Finin and Y. Labrou, editors, *Proceedings of the CIKM'94 Workshop on Intelligent Agents*, Gaithersburg, MD, USA, 1994.

[Din03] Cédric Dinont. Un modèle et un langage d'agent. Master's thesis, LIFL, 2003.

[dKLW97] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dMARS. In *Agent Theories, Architectures, and Languages*, pages 155–176, 1997.

[DKS02] Jürgen Dix, Sarit Kraus, and VS Subrahmanian. Agents dealing with time and uncertainty. In *AAMAS '02 : Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 912–919, New York, NY, USA, 2002. ACM Press.

[Don01] Stéphane Donikian. Hpts : a behaviour modelling language for autonomous agents. In *AGENTS '01 : Proceedings of the fifth international conference on Autonomous agents*, pages 401–408, New York, NY, USA, 2001. ACM Press.

[dS06] Etienne de Sevin. *An action selection architecture for autonomous virtual humans in persistent worlds*. PhD thesis, EPL, 2006.

[DvdHD03a] M. Dastani, J. van der Ham, and F. Dignum. Communication for goal directed agents, 2003.

[DvdHD03b] Mehdi Dastani, Jeroen van der Ham, and Frank Dignum. Communication for goal directed agents. In *Communication in Multiagent Systems*, pages 239–252, 2003.

[DvRDM03] M. Dastani, B. van Riemsdijk, F. Dignum, and J. Meyer. A programming language for cognitive agents : Goal-directed 3apl, 2003.

[DvRM05] Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Meyer. *Programming Multi-Agent Systems in 3APL*, chapter 2. Bordini et al. [BDDFS05], 2005.

[DZ05] Jürgen Dix and Yingqian Zhang. Impact : A multi-agent framework with declarative semantics. In R.H. Bordini, M. Dastani, J. Dix, and A.E.F Seghrouchni, editors, *Multi-Agent Programming*, LNAI 3346. Springer, 2005.

[ESP99] T. Eiter, V. Subrahmanian, and G. Pick. Heterogeneous active agents i : Semantics, 1999.

[Fer92] I. A. Ferguson. *TouringMachines : An Architecture for Dynamic, Rational, Mobile Agents*. PhD thesis, Cambridge, UK, 1992.

[Fer95] Jacques Ferber. *Les systèmes multi-agents, vers une intelligence collective*. InterEditions, Paris (France), 1995.

[FGI+02] Alessandro Farinelli, Giorgio Grisetti, Luca Iocchi, Daniele Nardi, and Riccardo Rosati. Generation and execution of partially correct plans in dynamic environments. In *Proceedings of the Third International Workshop on Cognitive Robotics*, 2002.

[Fis93] Michael Fisher. Concurrent METATEM - a language for modelling reactive systems. In *Parallel Architectures*

and Languages Europe, pages 185–196, 1993.

[Fis94] M. Fisher. A survey of concurrent METATEM – the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic - Proceedings of the First Intemational Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag : Heidelberg, Germany, 1994.

[FSS03] Amal El Fallah-Seghrouchni and Alexandru Suna. Claim un langage de programmation pour des agents autonomes, intelligents et mobiles. *Technique et Science Informatiques*, 22(4) :91–105, 2003.

[GPP+99] Mike Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Mike Wooldridge. The belief-desire-intention model of agency. In Jörg Müller, Munindar P. Singh, and Anand S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 1–10. Springer-Verlag : Heidelberg, Germany, 1999.

[HBdHM99] Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4) :357–401, 1999.

[HdBvdHM01] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming with declarative goals. *Lecture Notes in Computer Science*, 1986 :228–??, 2001.

[IGR92] Francois F. Ingrand, Michael P. Georgeff, and Anand S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert : Intelligent Systems and Their Applications*, 7(6) :34–44, 1992.

[KIO95] Kazuhiro Kuwabara, Toru Ishida, and Nobuyasu Osato. Agentalk : Coordination protocol description for multiagent systems. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*, page 455, San Francisco, CA, 1995. MIT Press.

[Kuw96] K. Kuwabara. Meta-level control of coordination protocols. In *Proc. Second International Conference on Multi-Agent Systems (ICMAS '96)*, pages 165–172, 1996.

[LAP01] J. Leite, J. Alferes, and L. Pereira. Minerva - a dynamic logic programming agent architecture, 2001.

[Lee06] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, January 10 2006.

[Mar06] Paul-Edouard Marson. Artifacts, abstractions pour la modélisation et la mise en oeuvre des environnements multi-agents. Master's thesis, Université Paris IX Dauphine, 2006.

[MB02] A. Moreira and R. Bordini. An operational semantics for a bdi agent-oriented programming language, 2002.

[MC95] Frank G. McCabe and Keith L. Clark. April – agent process interaction language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents : Theories, Architectures, and Languages (LNAI volume 890)*, pages 324–340. Springer-Verlag : Heidelberg, Germany, 1995.

[MP93] J. Muller and M. Pischel. The agent architecture interrap : Concept and application, 1993.

[MV05] Andrea Omicini e Alessandro Ricci Mirko Viroli. Engineering mas environment with artifacts. *E4MAS'05*, 2005.

[ND06] Peter Novák and Jürgen Dix. Modular bdi architecture. In *5th International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2006),Hakodate, Japan*, May 2006.

[OD01] A. Omicini and E. Denti. From tuple spaces to tuple centres, 2001.

[ORV+04] A. Omicini, A. Ricci, M. Viroli, C. Castelfranchi, and L. Tummolini. Coordination artifacts : Environment-based coordination for intelligent agents. In *AAMAS'04*, 2004.

[Osh01] Alexander Osherenko. Plan representation and plan execution in multi-agent systems for robot control. In *PuK*, 2001.

[PBL05] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. *JADEX - A BDI Reasoning Engine*, chapter 6. Bordini et al. [BDDFS05], 2005.

[PL00] Lin Padgham and Patrick Lambrix. Agent capabilities : Extending BDI theory. In *AAAI/IAAI*, pages 68–73, 2000.

[Rao96] Anand S. Rao. AgentSpeak(L) : BDI agents speak out in a logical computable language. In Rudy van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.

[RCO04] R. Ross, R. Collier, and G. O'Hare. Af-apl - bridging principles & practice in agent oriented languages, 2004.

[RG91] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc. : San Mateo, CA, USA, 1991.

[RG95] A. S. Rao and M. P. Georgeff. BDI-agents : from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[Ric01] Nadine Richard. *Description de comportements d'agents autonomes évoluant dans des mondes virtuels*. PhD thesis, ENST, 2001.

[RK96] S. J. Rosenschein and L. P. Kaelbling. A situated view of representation and control. In P. E. Agre and S. J. Rosenschein, editors, *Computational Theories of Interaction and Agency*, pages 515–540. The MIT Press : Cambridge, MA, USA, 1996.

[RMP96] Michael Rosinus, Jörg P. Müller, and Markus Pischel. An agent specification language, 1996.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence : A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[Ros96] Michael Rosinus. Aladin - a language for designing interrap agents, 1996.

[RV05] Alessandro Ricci and Mirko Viroli. A timed extension of respect. In *SAC '05 : Proceedings of the 2005 ACM symposium on Applied computing*, pages 420–427, New York, NY, USA, 2005. ACM Press.

[Sad05] David Sadek. *Artimis Rational Dialogue Agent Technology : an Overview*, chapter 9. Bordini et al. [BDDFS05], 2005.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1) :51–92, 1993.

[Sin97] Munindar P. Singh. A customizable coordination service for autonomous agents. In *Agent Theories, Architectures, and Languages*, pages 93–106, 1997.

[SMWC97] M. Schroeder, R. Marques, G. Wagner, and J. Cunha. Cap - concurrent action and planning : Using pvm-prolog to implement vivid agents, 1997.

[SS05] Amal El Fallah Seghrouchni and Alexandru Suna. *CLAIM and SyMPA : a programming environment for intelligent and mobile agents*, chapter 4. Bordini et al. [BDDFS05], 2005.

[Sun05] Alexandru Suna. *CLAIM et SyMPA : Un environnement pour la programmation d'agents intelligents et mobiles*. PhD thesis, Université Pierre et Marie Curie, 2005.

[Tai93] Patrick Taillibert. Programmer en prolog sans effet de bord. Technical report, Dassault Electronique, 1993.

[Thi05] Michael Thielscher. Flux : A logic programming method for reasoning agents. *TPLP*, 5(4-5) :533–565, 2005.

[Tho95] S. Rebecca Thomas. The placa agent programming language. In *ECAI-94 : Proceedings of the workshop on agent theories, architectures, and languages on Intelligent agents*, pages 355–370, New York, NY, USA, 1995. Springer-Verlag New York, Inc.

[TV02] Jean-Christophe Trigaux and François Vermaut. Méthodes et outils de conception orienté-agent. Master's thesis, FUNDP. Institut d'informatique, 2002.

[vdHW03] W. van der Hoek and W. Wooldrige. Towards a logic of rational agency, 2003.

[vRvdHM03] Birna van Riemsdijk, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming in dribble : from beliefs to goals using plans. In *AAMAS '03 : Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 393–400, New York, NY, USA, 2003. ACM Press.

[Wag96] G. Wagner. Viva knowledge-based agent programming, 1996.

[WBPL06] Andrzej Walczak, Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Augmenting bdi agents with deliberative planning techniques. In *The 5th International Workshop on Programming Multiagent Systems (PROMAS-2006)*, 2006.

[Wei99] Gerhard Weiss, editor. *Multiagent systems : a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, MA, USA, 1999.

[Win05] Michael Winikoff. *JACK Intelligent Agents : an Industrial Strength Platform*, chapter 7. Bordini et al. [BDDFS05], 2005.

[WJ95a] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents : Theory and practice, 1995.

[WJ95b] Michael J. Wooldridge and Nicholas R. Jennings. Agent Theories, Architectures, and Languages : A Survey. In Michael J. Wooldridge and Nicholas R. Jennings, editors, *Workshop on Agent Theories, Architectures & Languages (ECAI'94)*, volume 890 of *Lecture Notes in Artificial Intelligence*, pages 1–22, Amsterdam, The Netherlands, January 1995. Springer-Verlag.

[Woo96] Michael Wooldridge. Practical reasoning with procedural knowledge. In *Formal and Applied Practical Reasoning*, pages 663–678, 1996.

[Woo02] Michael Wooldridge. *Introduction to MultiAgent Systems*. John Wiley & Sons, June 2002.

[WRR95] D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a concurrent agent-oriented language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents : Theories, Architectures, and Languages (LNAI Volume 890)*, pages 386–402. Springer-Verlag : Heidelberg, Germany, 1995.