Titre: Title:	IDEAL : a hybrid architecture for multi-agent systems
Auteur: Author:	Ming Bai
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Bai, M. (2003). IDEAL : a hybrid architecture for multi-agent systems [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7447/

UNIVERSITÉ DE MONTRÉAL

IDEAL : A HYBRID ARCHITECTURE FOR MULTI-AGENT SYSTEMS

MING BAI

DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)

SEPTEMBRE 2003

Your file Votre référence
ISBN: 0-612-97922-9
Our file Notre référence
ISBN: 0-612-97922-9

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

IDEAL : A HYBRID ARCHITECTURE FOR MULTI-AGENT SYSTEMS

présenté par : BAI, Ming
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. ROY, Robert, Ph.D., président
M. BERNARD, Jean-Charles, Ph.D., membre et directeur de recherche
M.SAMUEL, Pierre, Ph.D., membre

Dedication

TO YI-SHAN AND DAVID

TO MY PARENTS

Acknowledgements

First and foremost, I would like to express my deepest appreciation to Professor Jean-Charles Bernard, my supervisor, for his invaluable advices, patient guidance, helpful discussions and enthusiastic encouragement throughout the present investigation. Without these, this thesis would have been impossible.

Special thanks are due to Mrs. Madeleine Guillemette for her kindness and patience in proof reading this thesis.

My thanks go to my fellow colleague in my research group, Esmahi Larbi, for the useful discussions on the project and for the friendship with him.

The financial support provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) is gratefully acknowledged.

Finally, I am deeply grateful to my wife, Yishan Qiu, for her patience, understanding, encouragement and support. In addition, I am also deeply grateful to my parents for their constant encouragement and support throughout my education.

Abstract

Researches on Intelligent agents and multi-agent systems have been an active field in artificial intelligence society even in mainstream computer science in recent years. Some famous agent models and systems have been proposed. Due to the variety of complex application problems, the models do not always provide satisfying solutions.Such is the case for e-business applications, cruise missile control, air-battle management, network management, etc. To provide better supports to the applications, in this thesis, an intelligent agent model, IDEAL (**I**ntegrated **D**eliberative and r**E**active **A**rchitectures with **L**earning abilities) is proposed. An IDEAL agent is horizontal hierarchical hybrid architecture. It is partly based on BDI paradigm, thus making use of the three agent's primary mental attitudes (beliefs, desires, and intentions) as data structures, but also uses reactive patterns to deal with unpredictable events or events requiring fast response times. Two effective learning mechanisms are integrated into the architecture. Not only do they bridge the gaps between reactive and deliberative paradigms thus enhancing reaction speed for processing complex problems, but they also improve the performance of the system through optimization of the agent's behaviors. A specially designed dynamic planer with pseudo-parallel processing abilities is used to improve real time planning abilities. Qualitative comparisons of an IDEAL agent and some other agent models are presented.

On the basis of the IDEAL agent model, a multi-agents frame with a hierarchical broadcast frame is proposed. We show that it can be used to simplify and improve the design and development of intelligent agent applications. A detailled application example is given.

Résumé

Durant les dernières années, les systèmes multi-agents et agents intelligents ont fait l'objet de recherches intensives dans le domaine de l'informatique et de l'intelligence artificielle. Plusieurs systèmes et modèles d'agent ont été proposés. Néanmoins, ces modèles ne donnent pas de solutions satisfaisantes pour certaines applications complexes telles que le commerce électronique, la commande des missiles de croisière, la gestion de combat aérien et la gestion des réseaux etc....

Dans le cadre de ce mémoire, nous proposons un modèle d'agent intelligent, dit IDEAL (an agent **I**ntegrated **D**eliberative and r**E**active **A**rchitectures with **L**earning abilities) (agent intégrant des architectures délibérantes et réactives avec des aptitudes d'apprentissage), assurant de meilleurs supports aux applications. Un agent IDEAL est une architecture hybride hiérarchique horizontale. Il est partiellement basé sur le paradigme BDI et par conséquent emploie les trois caractéristiques mentales primaires d'un agent : les croyances, les désirs, et les intentions) comme structures de données. L'agent IDEAL utilise aussi les aspects usuels de réactivité pour traiter les événements imprévisibles ou pour réagir de façon rapide (fournir une réponse) aux évènements. Deux mécanismes d'apprentissage efficaces sont intégrés dans l'architecture que nous proposons. Ils établissent non seulement des liens entre les paradigmes réactifs et délibératifs pour améliorer la vitesse de réaction au cours du traitement des problèmes complexes, mais améliorent également la performance du système par l'optimisation des comportements de l'agent. Un planificateur dynamique, conçu spécialement avec des capacités de traitement pseudo parallèle, est utilisée pour améliorer les capacités de planification en temps réel de l'agent IDEAL. En se basant sur le modèle d'agent IDEAL, nous proposons un système multi-agents à structure hiérarchique avec communication par diffusion. Elle peut être employée pour la simplification, l'amélioration de la conception et du développement des applications d'agents intelligents. Des comparaisons entre le

modèle d'agent IDEAL et d'autres modèles sont présentées. En conclusion, un exemple d'application d'agents IDEAL est donné.

Condensé en français

Introduction

Depuis plusieurs années, les agents et multi-agents intelligents sont de plus en plus utilisés dans des domaines aussi variés que la gestion d'horaire [Sycara 97], la recherche d'information [Sheth 94], la gestion de réseau [Song 96] et le *e-business.* Le succès de ces applications a attiré l'attention des chercheurs dans tous les domaines et le concept d'agent a pris une place prédominante tant en intelligence artificielle que dans les autres champs de l'informatique.

À partir des travaux de [Nwana 96] et de [Wooldridge 95b], on reconnaît aujourd'hui que les agents devraient tous minimalement posséder des caractéristiques d'autonomie [Barber 99], de coopération de même que des capacités d'apprentissage.

Problématique et objectifs

Dans le présent mémoire, nous développons une nouvelle architecture d'agent que nous avons nommée IDEAL. Pour justifier la création, de toutes pièces, d'une nouvelle architecture plutôt que d'utiliser certaines qui existent déjà, il faut d'abord bien comprendre le domaine d'application que nous visons, c'est-à-dire le marché des options sur les commodités et encore plus spécifiquement celui des monnaies. Dans ce genre de marché, les décisions doivent parfois être prises très rapidement alors qu'à d'autres moments, il faut vraiment prendre le temps d'analyser la situation avant de décider. Ces deux aspects mettent en évidence la nécessité d'avoir un agent pouvant réagir rapidement mais devant aussi posséder des capacités de délibération. Conséquemment, notre architecture sera une architecture hybride combinant ces deux aspects et contiendra les mécanismes de contrôle nécessaires pour passer facilement d'un mode à l'autre.

Afin de permettre à notre agent de réaliser des tâches complexes, nous devons aussi lui le rendre capable de construire des plans à partir d'actions

élémentaires. Pour cela nous comptons intégrer une approche « *means-ends analysis*» afin de laisser à l'agent la capacité de choisir la meilleure méthode -ou combinaison de méthodes- pour accomplir chacune des tâches auxquelles il sera confronté.

Par ailleurs, comme il arrive souvent que les conditions de déclenchement d'un plan se répètent dans un environnement comme celui que nous visons, nous voulons doter notre agent de capacités telles que si des condition identiques réapparaissent, il n'aura pas à recommencer toute l'analyse de l'environnement pour établir un nouveau plan, mais pourra faire appel au plan déjà construit lors de la première occurrence de ces conditions. En fait, nous voulons intégrer à notre agent des capacités d'apprentissage.

Finalement, l'agent que nous désirons devra posséder des capacités de collaboration avec d'autres agents. Par exemple, considérons la situation suivante impliquant deux agents A et B. L'agent A, spécialisé dans les opérations concernant le yen japonais, possède n unités de ressources qu'il peut utiliser selon que les opportunités se présentent. L'agent B, spécialiste des opérations sur la livre sterling, possède aussi n unités de ressources. Supposons qu'étant données les conditions du marché, les opportunités pour l'agent A, d'agir avec profit soient très fréquentes; il peut arriver que l'agent A ne possède plus suffisamment de ressources pour continuer ses opérations. D'autre part, supposons que pour l'agent B, les occasions ne soient pas aussi avantageuses. Il lui reste un excédent de m unités de ressources. Sachant cela, l'agent A pourrait demander à l'agent B de collaborer avec lui pour investir dans des transactions sur le yen. L'agent B aurait le choix d'accepter ou de refuser de prêter ses surplus de ressources à l'agent A.

Derrière cet exemple simple, on voit la nécessité, pour notre architecture de posséder tous les mécanismes usuels de négociation et de collaboration.

Une architecture hybride

L'architecture que nous proposons s'appuie fortement sur le modèle d'agent Interrap de Mullër [Muller 95] [Muller 97] et sur l'approche BDI de Rao [Rao 95]. Il en résulte une architecture hybride constituée de plusieurs couches fonctionnelles horizontales superposées les unes aux autres[1]. À chaque niveau correspond essentiellement une des caractéristiques spécifiques que nous croyons qu'un agent intelligent devrait posséder. Les bases de connaissances de l'agent sont aussi partionnées en fonction de cette superposition. Les agents construits selon cette architecture se verront dotés de capacités réactives et délibératives, intégrées en un tout, et leur permettant d'exprimer des comportements globaux cohérents.

Par l'utilisation d'une architecture en couches, le développement modulaire d'un agent sera grandement facilité et l'ensemble de ses composantes deviendra vraisemblablement pus compact. D'autres parts, nous croyons qu'une telle architecture aura un impact important sur la robustesse des agents l'utilisant et en facilitera de beaucoup le déboguage dans le contexte d'applications précises. De toute évidence, une telle architecture en couches permet d'entrevoir, en considérant chaque couche indépendante, la possible utilisation de mécanismes parallèles améliorant ainsi les capacités computationnelles des agents. Finalement, comme les bases de connaissance seront elles aussi structurées en couches associées aux différents niveaux fonctionnels, la quantité des connaissances dont un agent aura besoin à un instant donné se limitera aux connaissances accessibles pour ce niveau. Les performances de l'agent en seront ainsi encore améliorées.

---

[1] Voir figure à la fin du paragraphe **L'aspect conceptuel**

L'aspect conceptuel

L'idée de base derrière l'agent IDEAL est la tentative d'émuler le plus près possible -et éventuellement implanter- la façon dont l'être humain pense et réagit lorsque confronté à un problème. Globalement, deux situations sont possibles.

Premièrement, il connaît la solution immédiatement sans qu'il lui soit nécessaire de considérer les détails du problème à résoudre. Cette solution peut lui venir de façon intuitive[2], c'est-à-dire, qu'il sait ce qu'il faut faire mais ne saurait expliquer pourquoi, ou, il est possible que l'humain ait déjà, dans le passé, été confronté au même problème, ou à un problème similaire, et qu'il se souvienne de la solution alors utilisée. Le souvenir d'expériences passées est une quasi garantie de la validité de la solution au problème actuel car cette approche a probablement alors été validée, d'une certaine façon, au moment même de sa première son application.

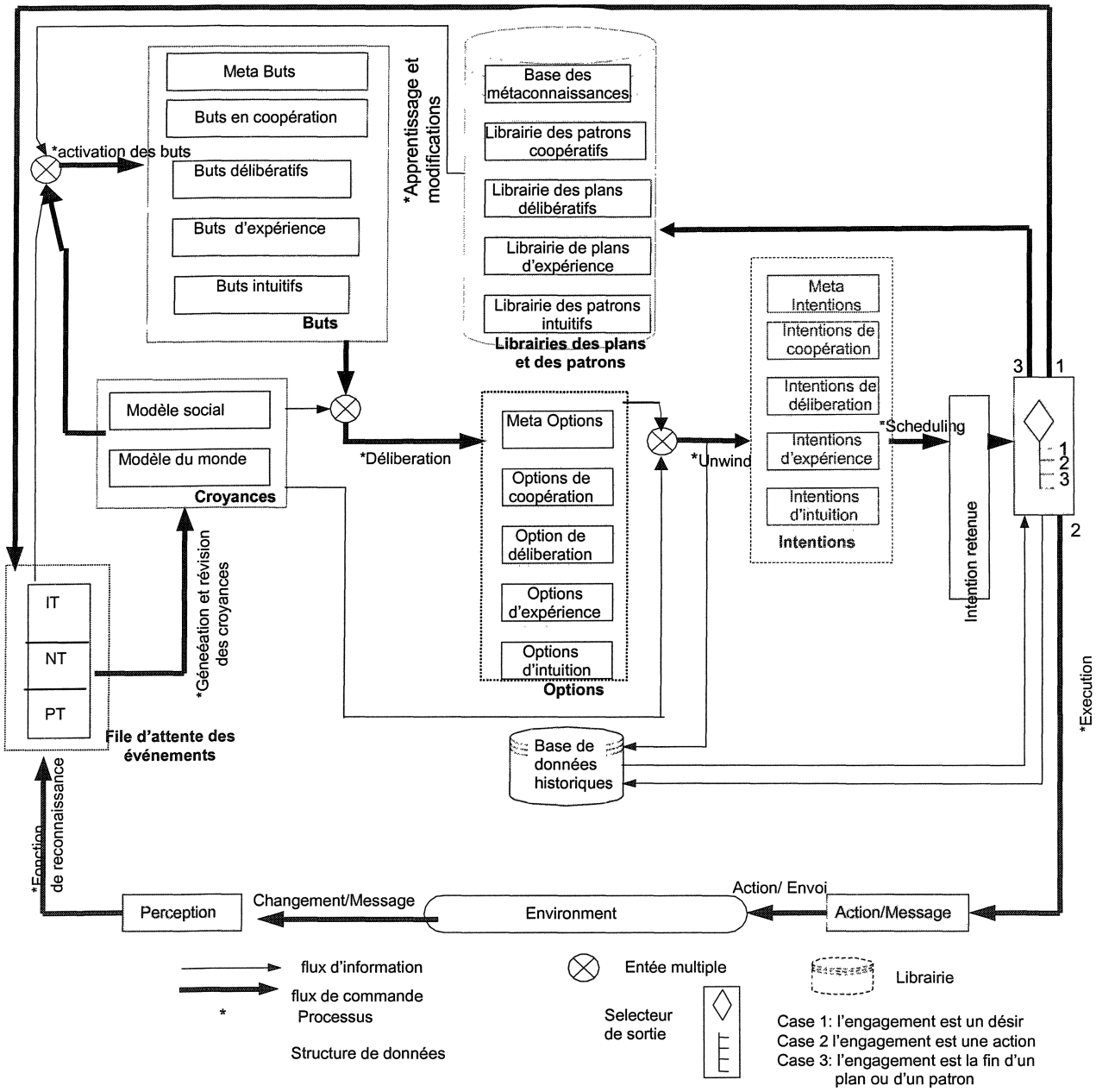
Dans la deuxième situation, il n'existe pas de solution immédiatement apparente. L'humain doit alors considérer et attentivement analyser l'ensemble des paramètres décrivant le problème. Éventuellement, certaines solutions pourront être inférées à partir de sa connaissance de la situation actuelle ou à partir de situations similaires déjà rencontrées. Dans certain cas, il devra même aller jusqu'à demander à d'autres de l'aide pour réussir à résoudre son problème. De cette discussion, quatre mécanismes différents sont retenus comme fondement à l'architecture de l'agent IDEAL: l'intuition (réflexe), l'expérience positive, la délibération et la coopération. La figure de la page suivante met en évidence l'architecture à niveaux superposés correspondante au travers les différents

---

[2] Les mécanismes de traitement des intuitions ne font pas partie de notre étude. Notons cependant qu'ils peuvent expliquer une réaction immédiate de l'humain.

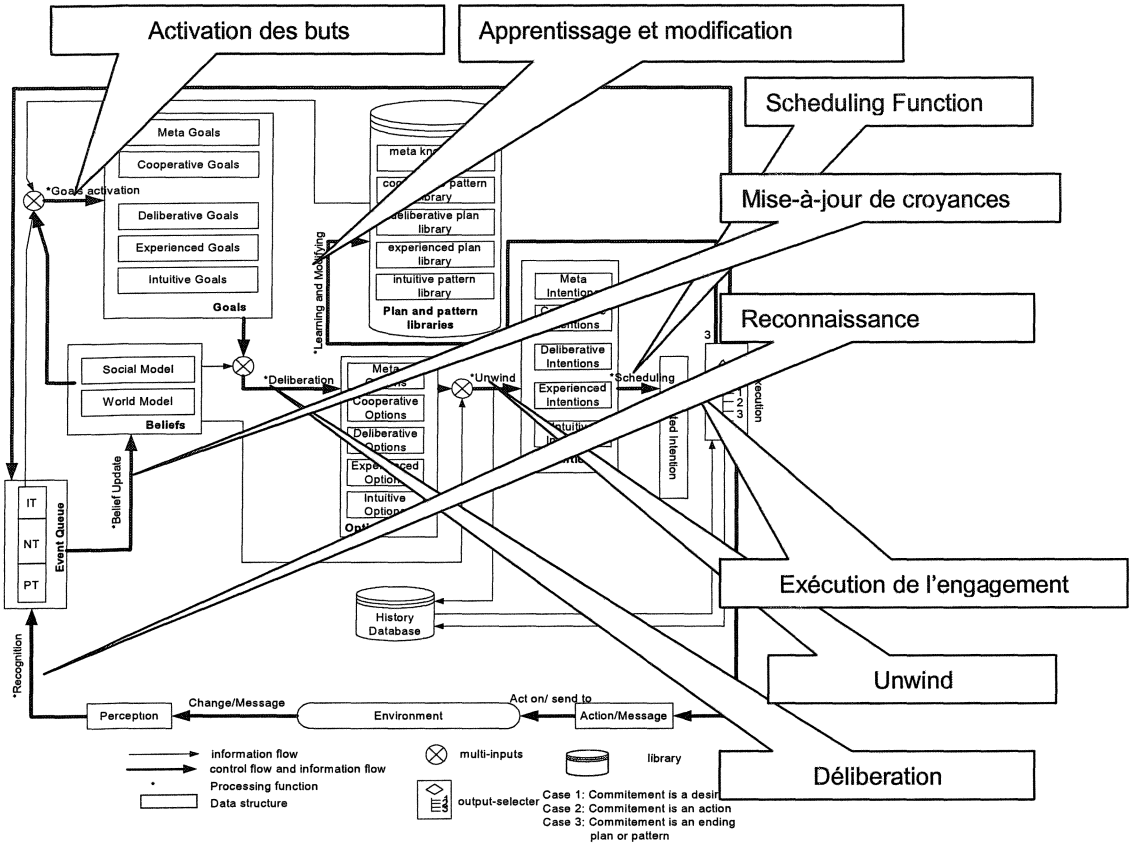
modules fonctionnels de l'agent. Ces fonctionnalités sont décrites au paragraphe suivant.

Architecture conceptuelle de l'agent IDEAL



L'aspect fonctionnel

La figure précédente met en évidence les relations qui existent entre l'aspect conceptuel et l'aspect fonctionnel de l'architecture proposée. En effet, les données: **Croyances, Buts, Options, Intentions** et **Librairies** sont structurées en couches superposées selon les concepts introduits plus haut. D'autre part, les flèches en gras représentent les flux de commande et les normales les flux d'information. Sur les flux de commandes, sont identifiées les principaux processus fonctionnels de notre modèle. Le schéma suivant indique leurs positions respectives et leur fonctionnalité sont définies comme :



- **Reconnaissance** transforme les perceptions en événements déclencheurs standards
- **Mise à jour des croyances** comme son nom l'indique met à jour les connaissances courantes et s'occupe aussi de la révision de l'ensemble des croyances afin d'en maintenir la consistance
- **Activation des buts** génère, selon événement qui l'active, un ou des plans (ou encore patrons) qui sont applicables par l'agent.

- **Délibération** extrait de l'ensemble des buts candidats la ou les solutions optimales réponses à l'événement qui est traité. Ce sont les options d'exécution possibles.
- **Unwind** est la fonction qui exprime les options trouvées en terme d'intentions.
- **Schedule** choisit l'intention ayant la plus grande priorité et la transmet à Exécution.
- **Exécution** commande l'exécution de l'intention et l'agent devient engagé à l'exécution du plan correspondant.
- **Apprentissage et modification** joue deux roles différents. D'une part, il génère les plans basés sur l'expérience en utilisant le mécanisme d'apprentissage par l'exemple. Il procure ainsi des patrons ayant des temps de réponse similaires à ceux des patrons intuitifs introduisant alors des effets identiques à ceux que produirait une approche de planification dynamique. D'autre part, ce processus modifie la pondération des plans et patrons associés, de telle façon que leur utilisation future devrait améliorer les performances de l'agent. Pour ce faire, il utilise un mécanisme d'apprentissage par *feedback*.

Les états internes de l'agent IDEAL

Au niveau des entrées/sorties on retrouve les éléments perception, information de négociation ainsi que les notions d'évènement, file d'attente des événements et action. En s'appuyant sur le modèle classique BDI, les notions de croyance, de désir, d'intention, d'objectifs, d'option et d'intention sont utilisées pour représenter les états mentaux de l'agent. Nous introduisons la notion de structure d'intention qui joue un rôle primordial lors de l'exécution des tâches. Il s'agit en fait d'une pile dans laquelle serons placés les séquences d'actions

diverses que l'agent devra accomplir réaliser pour accomplir sa tâche. La structure d'intention est fondamentale lorsque l'environnement de l'agent est dynamique. De façon simple, si un nouvel événement se produit quand l'agent s'est déjà engagé dans un autre traitement et que le nouvel événement est prioritaire à celui en traitement, le traitement s'arrête, l'état du système est empilé avec ce qui reste à compléter du traitement actuellement en cours et le nouvel événement est traité. Au retour de la complétion du nouvel événement, l'ancien état est dépilé, est analysé en termes de réalisabilité et selon le cas annulé ou remis en marche.

Les notions de plan et librairie de plans sont fondamentales à l'agent pour l'exécution correcte de ses tâches. Nous distinguons les plans fixés d'avance que nous appelons **patrons**, de ceux construits par l'agent lui-même que nous dénommons **plans**. Il y a les patrons pour les aspects réflexes et intuitifs et les patrons associés à la coopération. On distingue aussi le méta patrons qui sont utilisés lors de la recherche de plans. Ce sont en fait la méthode utilisée pour établir les plans.

Quant aux plans, ils sont strictement associés aux aspects délibératifs de l'agent. Les plans ayant déjà été construits et complétés avec succès lors de l'exécution de tâches antérieures sont appelés **plans d'expérience**. Lorsque des conditions demandant délibération ont déjà été rencontrées et que le plan alors établi s'est exécuté avec succès, ce plan déjà connu est d'abord utilisé. En cas d'échec, un nouveau plan est alors bâti.

Les patrons et les plans sont représentés par des graphes connectés enracinés orientés de type ET/OU, où les nœuds représentent l'état du système et les arêtes les actions permettant de passer d'un état à un autre. On nomme « actions » les opérations de base, non divisibles, que l'agent peut utiliser pour répondre aux modifications perçues dans l'environnement ou pour modifier son état mental. Elles sont disponibles via la librairie des actions de bases. L'utilisation de la représentation par graphe permet un très bon suivi de

l'exécution des plans. Pour réaliser les plans, deux algorithmes différents sont proposés. Le premier permet la création temporellement dynamique de plans et le contrôle de leur exécution. Le second permet d'exécuter, de façon non interruptible, des patrons. Le comportement de l'agent est optimisé par l'utilisation d'une méthode de pondération dynamique de la valeur des plans et patrons. Un module d'enregistrement de l'utilisation des plans et patrons ainsi que l'enregistrement des modifications de l'agent en découlant est défini et constitue la **base de données historiques**, une sorte de journal de ce qui s'est passé.

Le cycle d'exécution

Deux sortes de modification de l'environnement de l'agent existent. Les premières, dites ***info_nego***, correspondent à des échanges entre des agents coopérant à l'exécution d'une tâche. Les secondes représentent les modifications de l'environnement requérant une réponse de l'agent. Essentiellement ce sont les problèmes que l'agent doit essayer de résoudre. Dès l'occurrence d'une modification, Celle-ci est analysée par la fonction de reconnaissance et un événement, de type approprié, est inséré dans la file d'attente. D'une part, les croyances de l'agent sont mises à jour par la fonction de révision, modifiant ainsi le modèle du monde auquel l'agent se réfère et d'autre part l'information est simultanément transmise à tous les niveaux de la hiérarchie lesquels retiennent l'information seulement si elle leur est pertinente et la rejetttent autrement. Normalement, un seul des niveaux retient l'information.

Une fois la mise à jour des croyances terminée, la commande de l'agent est passée à la fonction activation des buts.

Les désirs sont des séquences d'actions de bases décrivant soit un comportement que l'agent peut vouloir exprimer, soit un chemin à suivre pour

atteindre un état spécifique. Tout désir réalisable pour un contexte donné est appelé un but. Notez qu'à tout but correspond un plan ou un patron.

Les informations pertinentes contenues dans la librairie des plans et patrons correspondant au niveau activé sont utilisées afin de déterminer l'ensemble des solutions réalisables, c'est-à-dire, l'ensemble des plans et patrons utilisables pour réaliser la tâche que l'agent doit accomplir suite à l'occurrence de l'événement. Cet ensemble contient donc tous les buts de l'agent étant donné son état mental présent, ses croyances et les plans et patrons à sa disposition.

Chaque plan ou patron possède un ***potentiel d'impact*** représentant la mesure que parmi tous les plans ce plan soit celui qu'il faille exécuter. Bien sûr, on désire trouver le ou les plans ayant le potentiel d'impact maximal. C'est le rôle de la fonction délibération qui retient parmi tous les buts identifiés ceux qui sont d'impact maximum. Les plans ainsi retenus s'appellent des options. Les options sont transmises à la fonction ***Unwind*** qui les exprime en terme d'intention. La commande est passée à la fonction ***Scheduling*** dont le rôle est déterminer quelle intention sera exécutée et l'insère dans la pile d'intention. La commande est passé à la fonction ***exécution.*** Selon l'engagement de l'agent, la commande est redirigée :

- vers la file d'attente s'il s'agit d'un désir,
- vers l'environnement s'il s'agit d'une action de base,
- vers la librairie des plans et patrons s'il s'agit de la fin d'un plan ou d'un patron et la fonction d'apprentissage est activée.

Simultanément, l'information est envoyée à la base de données historiques.

Un cadre multiagent.

La plupart des applicatios réelles sont complexes et souvent, les intervenant sont organisés en une structure de dépendance hiérarchique. En se basant sur cete

situation nous présentons un système multiagent hiérarchisé et caractérisé par une transmission de l'information de type diffusion (broadcast).

Nous caractérisons les propriétés propres à cette architecture et présentons une application reliée au domaine des transactions sur le marché des commodités. Plus spécifiquement, nous montrons que l'utilisation de ce modèle améliore le rendement des stratégies d'investissement lorsque comparé à celui obtenu par l'utilisation d'autres approches.

TABLE OF CONTENT

TABLE OF FIGURES

List of Appendices

Chapter 1: Introduction

1.1 The concepts of agent, intelligent agent and multi-agents

Intelligent agents and multi-agent systems have been applied to perform more and more tasks in recent years. These include calendar management and scheduling [Sycara 97], information filtering [Sheth 94], network management [Song 96] and business decision-making. The success of these applications attracts attention from researchers of computer science. The concept of an agent has become important in both artificial intelligence and mainstream computer science. Unfortunately however, there is no widely accepted definition about what an agent is, as, for example, there is one about "intelligence" in AI. Some researchers [Nwana 96] [Wooldridge 95b] have suggested using a set of characteristics to describe agents. H. S. Nwana points out that minimally, intelligent agents should exhibit **autonomy, learning and cooperation** abilities. Autonomy is the ability of a system to make its own independent decisions, without guidance from people or from other agents while learning is the ability to improve the system's performance through study of their own experience and cooperation is the ability to interact with other agents, possibly human, via some communication language.

If, as [Gilbert 96] says: "intelligence" can be explained as "the degree of reasoning and learning", then we define **an intelligent (software) agent** as:

> A software entity that carries out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employs some knowledge or representation of the user's goal or desires, and employs some degree of learning mechanism to improve the quality of its behaviors over time.

A **multi-agent system** (MAS) is a complex system constituted of, in certain manners, many individual (intelligent) agents. Multi-agent systems are well suited to work in complex environments, especially, those with unpredictable changes. MAS adopt distributed fashions to implement complex problem-solving abilities, with a reasonable performance/price, although each individual agent only has incomplete knowledge and limited resources. The characteristics and structures of MASs are application-oriented. Therefore, for different applications, different structures will be used.

The kernel of any intelligent agent is its architecture, i.e., the description of its modules and of how they work together. Thus, the design of agent architectures is how to construct intelligent agents so they satisfy the minimal characteristics set defined above. In fact, designing agent architectures has become a key research field in intelligent agents and multi agents systems.

1.2 Design motivations, tasks and requirements

One motivation of our research is to propose an intelligent agent model which would satisfy the requirements for e-business applications such as options trading decision making systems, electronic brokers, stocks markets management systems etc. The model could also be used for applications in other fields such as network management, air-battle management, intelligent tutoring systems, etc. Since agent architectures are application-oriented, specifying the design requirements by analyzing related applications becomes important. For example, e-business applications are very different from real-time control ones. Usually, they are not sensitive-time, but correctness and rationality of the systems' actions are truly crucial, particularly those executed on behalf of a human. For instance, we never require a currencies options trading assistant agent to make its decisions in a few microseconds because this is not necessary since currency futures' prices change one time a day. We do however expect its decisions to bring us lots of money. Most of the time, such applications are

characterized by large volume of data, dynamic and complex working environments and a necessity for optimal or near-optimal behaviors (solutions). Thus, from this analysis, we believe our design requirements could be described as follow:

- *robustness*.

  E-commerce applications do not allow occurrences of operations failure or system blocking. Regardless of any unpredicted situations, the application systems must provide a suitable solution with minimal risks based on its economic model even though it is probably not optimal.

- *learning abilities*

  E-commerce applications require good performance. Some learning mechanism will improve and optimize the systems' behaviors, step by step, through its experiences. We thus see learning as an essential component.

- *rational reactive speed*

  E-commerce applications are real-time applications. But unlike real-time control systems that require second or microsecond level of reactive speed, e-commerce environment changes occur within minutes, hours or even days. So providing rational fast solutions is mainly to improve the quality of the user interface and attract its interests. To realize this objective however, two things must be closely considered:

  - replanning should be avoided as much as possible
  - multi-threads and multi-tasks technologies should be used as much as possible.

- *cooperative abilities*

  E-commerce problems are often large and complex; they need multiple computing units to coordinately work together. Though each unit has only limited knowledge and resources, the performance obtained by having many units to work together in a cooperative way, is highly increased. We believe that multi-agent systems do provide better solutions for such applications.

It seems evident that any multi-agent system derived from an agent model satisfying the above specified requirements would certainly provide good support in solving e-commerce applications problems. Consequently, considering our design motivations and requirements, realizing our system will necessitate the following two steps:

- the establishment of an intelligent agent model that satisfies our design requirements, i.e., robust abilities in dynamic decision making and process optimization, and fast response time, all this with cooperative and learning abilities;
- the integration of the model into a multi-agents architecture frame.

The objectives of the present work are to analyze different agent architectures, confront their functionalities to our design requirements and establish a new agent architecture. The results presented here are part of a larger research problematic oriented towards the characterization of what is computational intelligence and what means do we have to integrate it, in a controlled way, into computer systems.

Let us first present a classification of agents as based on the literature and to which we have added our own point of view. This is the object of the next chapter.

Chapter 2: Agents classifying

In our view, agents can be roughly divided into three kinds of paradigms (or three generations): simple autonomous agents (A_A), cooperative hybrid agents (A_C) and intelligent agents (with learning abilities) (A_L). They are prototypes of different stages of agents' evolution, respectively. The different types of agent cover different parts of the "agents' minimal characteristics set" in such a way that we have $A_L \subseteq A_C \subseteq A_A$.

2.1 Simple autonomous agents

Most simple autonomous agents work in single agent mode. They only accomplish simple tasks because their limited abilities. According to the differences in the knowledge and controlling structures, simple autonomous agents are known as deliberative or reactive agents.

2.1.1 Deliberative agents (BDI architecture)

We define a deliberative agent as an agent architecture that contains its own internal symbolic model of the world and a related logical inference mechanism which uses the agent's knowledge to modify its mental states -its internal states- and to make decisions. Usually, a deliberative agent's mental states consist of beliefs (B), desires (D) and intentions (I) which, respectively, represent the agent's information, motivational and deliberative knowledge for that particular state. The mental states, also called mental attitudes, directly determine the system's behaviors and are critical for achieving adequate performance. Deliberative agents are often called BDI agents.

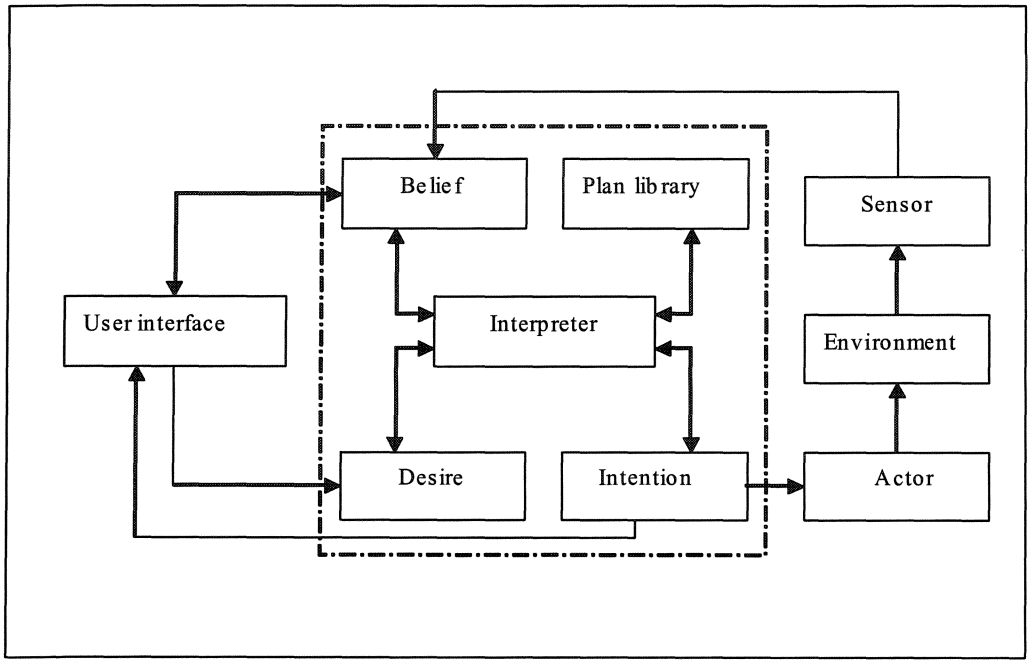


Figure 2.1 BDI agent architecture

Figure 2.1 shows the essential structure of a typical BDI agent [Rao 91] [Rao 95]. The agent consists of a set of facts about the world (Belief), a set of desires to be realized, a set of plans that describe how certain sequences of actions may be performed to achieve given goals or to react to particular situations, and an intention structure containing those plans that have been chosen for execution. An interpreter controls these components. At any particular point of time, goals are pursued with the use of active plans (intentions) and some events occur. These occurring events may or may not alter the beliefs, but if they do, the corresponding changes will trigger various new plans from the plan library. One or more of these triggered plans will then be chosen and placed in the intention

structure[3]. Finally, the interpreter selects an executable intention from the intention structure and executes one step of that intention. This will result in either the execution of an action and the establishment of a new sub-goal, or the conclusion to some new beliefs. At this point, the interpreter cycle repeats again. The new sub-goals and/or new beliefs will trigger new plans, one or more of them will be selected and placed on the intention structure, and again an intention will be selected from that structure and partially executed.

2.1.2 Non-deliberative agents (Reactive agents)

We define a reactive agent as an agent architecture that has no or only a very simple internal symbolic model of the world, only consists of several task-specific modules that checks continuously its environment and uses simple situation-action rules to make their decisions at run-time. Usually, reactive agents do not possess complex logical reasoning capacities. Their intelligence is not from the internal symbolic model but from the interaction with its environment, is implicitly part of the environment.

A famous example is Brooks' subsumption architecture [Brooks 86] as shown in Figure 2.2.

---

[3] The plans in the intention structure are then called intentions

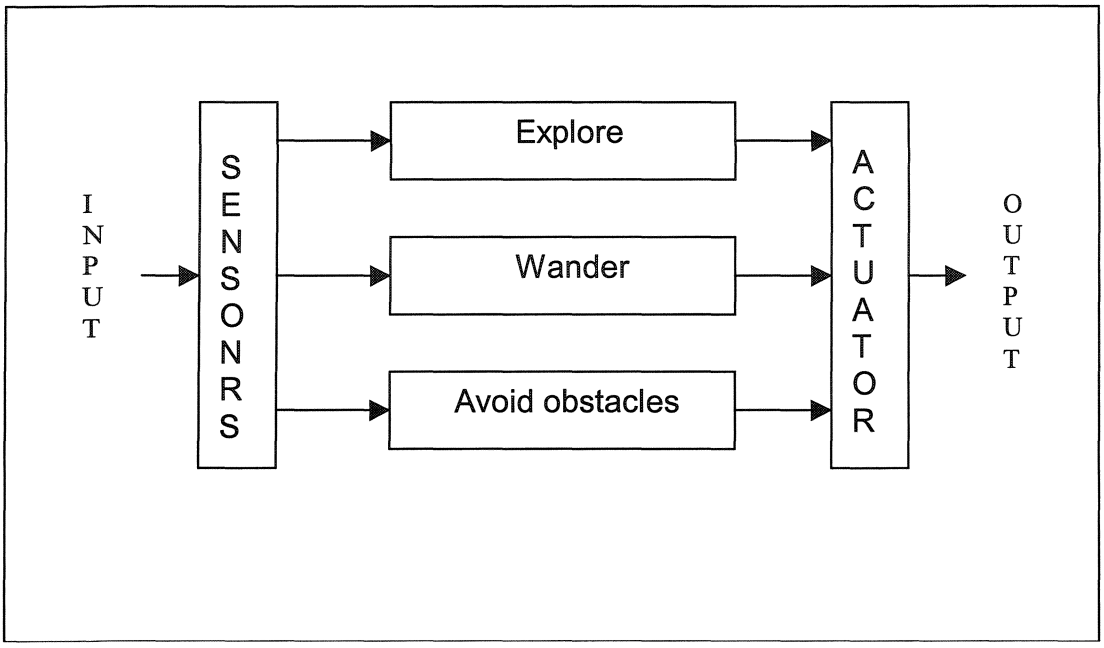


Figure 2.2 Subsumption architecture

Obviously, Brook's architecture is a hierarchy of task-oriented modules, each layer working in parallel and responsible for a very specific task. Between layers, no representational information is passed. The lower layers of the architecture are used to implement basic behaviors such as avoiding obstacles or wandering around in an area. Higher layers are used to incorporate facilities such as the ability to pursue goals (explore). Control is based on inhibition and suppression in the sense that higher-level layers can subsume the roles of the lower level layers when they wish to take control. During finite pre-programmed time intervals, upper layers are able to substitute or suppress the inputs to the lower layers and remove or inhibit the outputs from the same.

2.1.3 Comparisons between deliberative and reactive architectures

It is the differences of their knowledge base and control structure that give both deliberative and reactive agents their respective advantages and limits:

Possessing an internal knowledge base and central components such as reasoner and planer, makes deliberative agents express comprehensive capacities. Their central modules are not designed only with regard to the handling of some specific tasks, but can be used for a range of general problems. Unlike reactive agents, deliberative agents can decompose complex tasks into series of sub-goals and then achieve these by means of their internal knowledge base and reasoning capacities. They are thus suitable to solve complex problems. In addition, deliberative agents always provide optimal behaviors for achieving their goals since they plan for it. Note however that the time costs for the decompositions into sub-goals are very prohibitive.
Furthermore, since the deliberative agents' knowledge bases are pre formatted, it becomes difficult to update them during execution. This characterization of the knowledge bases is not suitable for dynamic environments where the changes of situations are often seen to happen. In deliberative agents, modeling the environment always leads to a high complexity of development, so that their application development is expensive and costly in time when compared with reactive agents. Finally, their most important disadvantage is their lack of robustness and error-tolerance: failure of central modules in deliberative agents always leads to failure of the whole system.

In contrast, the reactive paradigm uses no explicit knowledge base and no complex symbolic reasoning. The structure is decentralized and the amount of computation needed is small. All functionalities required to accomplish its tasks are contained in the hierarchy of modules themselves. Each module only has a precisely specified task and strictly possesses the pertinent related methods useful to fulfill the task. If a module fails, the agent probably can continue fulfilling the rest of its tasks. We can thus say that reactive agents provide fast, flexible

and robust behaviors in dynamic, continually changing, information-incomplete and unpredictable environments. The simple structure results in inexpensive application developments when compared with deliberative agents. However, reactive agents do not support complex solving-problem and do not provide agents with optimal behaviors. It is also difficult to implement learning and planning mechanisms in such simple structures.

2.2 Cooperative hybrid agents

Cooperative hybrid agents integrate reactive and deliberative capacities in a uniform architecture. Sometimes, social abilities are also integrated. This gives them the advantages of both reactive and deliberative agents but also some of the drawbacks. Such agents are currently employed in multi-agent systems. Each cooperative hybrid agent is normally designed as a hierarchical architecture. Reactive components are used to handle emergency situations or simply events such as interaction with the environment, exception case processing, etc and for which fast responses are required. Deliberative components are used for complex events decision making in order to achieve optimal behaviors. Muller's Interrap agent [Muller 97] is a typical example of hybrid architecture. Figure 2.3 shows the Interrap agent architecture.

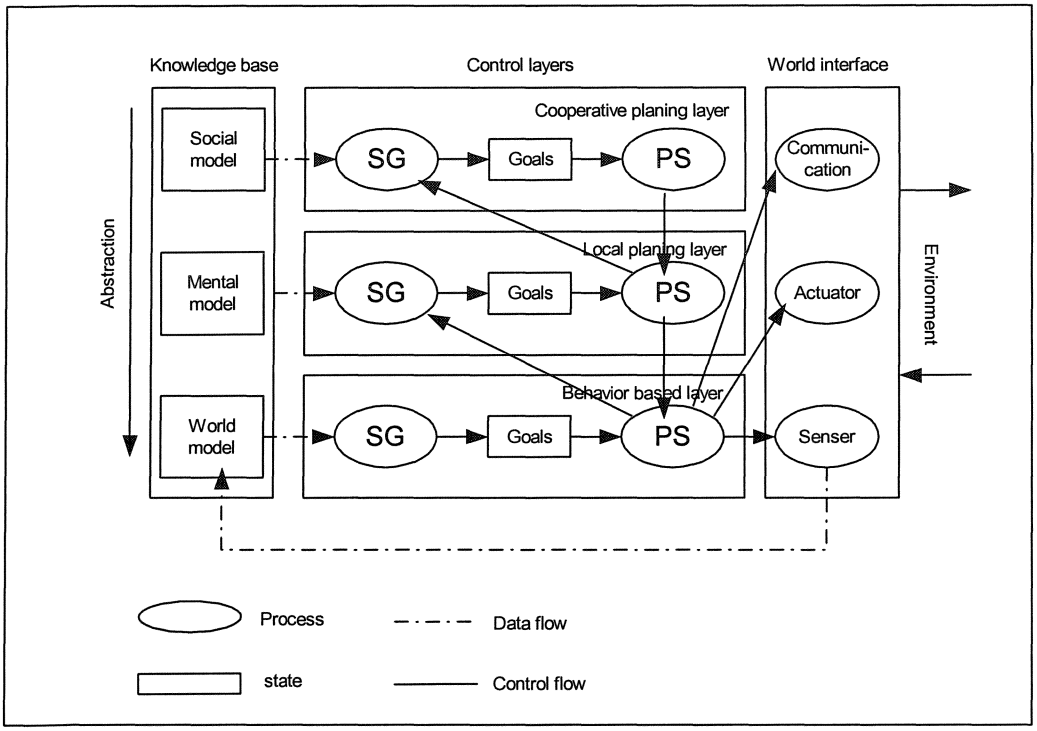


Figure 2.3 Interrap agent architecture

Muller's Interrap agent is "influenced" by the "abstract agent architecture" proposed by Bratman in [Bratman 87]. It consists of three parts: a knowledge base, some control layers and a world interface. The world interface is used to interact with the environment. The knowledge base is layered. It includes a world model that contains the agent's beliefs concerning the environment, a mental model that contains all beliefs the agent has about itself and social model that contains the agent's beliefs about other agents in the multi-agents system. From the bottom to up, the abstraction level increases layer by layer.

The control part, also layered, consists of a behavior-based layer (BBL), a local planning layer (LPL) and a cooperative planning layer (CPL). The BBL corresponds to the reactive component of the agent. It is the only component of the architecture that can directly interact with the world interface. LPL represents

the deliberative component of the agent. It processes long-term goals and uses BBL as its operation primitives. The CPL describes the cooperative capacities of the agent in the multi-agent system. It is in charge of communications and negotiations with other agents. The CPL's purpose is to generate cooperative plans.

Every control layer consists of two processes: the situation recognition and goal activation process (SG), and the planning, scheduling and execution process (PS). The SG process implements the acquisition and revision of beliefs and the generation and activation of goals. The PS process generates options from goals and derives intentions in accord with the given options by using a reasoning process. These intentions are scheduled into a plan which is then executed. Muller's contribution lies in the fact that he uses reactions to represent very short-term goals. These goals are achieved by executing patterns which are event-driven, i.e., started in reaction to the occurrence of an event. This approach bridges reactive systems with the concept of goal that usually only exists in deliberative systems.

Interrap is a vertical hierarchical architecture. The control is activated from the bottom layer to the higher layer. When a change in the agent's beliefs happens, the SG process of the lowest layer, BBL, recognizes it. If BBL is able to react to the change, then it is done via its PS process. Otherwise, it means that the change is beyond the capacities of treatment and competences of BBL. The control is shifted up to the next higher layer, LPL, where the treatment is done or eventually passed to CPL, the highest level.

The execution process of a decision is run from top to bottom. When some layer attains a decision from its PS process, the result –an intention- is passed downward to the next lower layer where operative primitives, defined at this layer level, are used to implement the decision. Only BBL can actually access the actuators of the world interface. Hence, all decisions of the agent are in fact executed at the BBL level. In order to reduce the time costs and also to maintain a fair speed of response, each control layer is allowed access only to its own

level knowledge base and those of any lower layers. For example, BBL strictly has access to the world model, but LPL can access both mental and world models.

When compared to simple autonomous agent architectures, the Interrap architecture represents a big progress in the evolution of agents. It integrates both deliberative and reactive aspects into hybrid architecture so as to benefit from both of them. It provides flexible, optimal and fast solutions. Obviously, it partly removes some shortcomings of deliberative agents such as slow response times and rigidity of structures. Interrap agents thus become quite suitable for dynamic environments. Given that it can use plans and that it has some symbolic treatment capacity, it can be used for general problem solving in complex environment. In addition, it integrates very flexible communication and negotiation mechanisms which enable agents to collaborate in a coordinated way when the system needs to solve problems in a distributed approach. By designing the cooperative module as the highest layer it becomes easy to control the lower layers while in the cooperative mode. Note that the suppression functionality of Brooks' subsumption architecture and the role of the CPL layer of the Interrap architecture are very similar. Their difference strictly lies on the use of different implementing approaches. In the Interrap agent, reactors[4] are introduced in order to monitor the reactive module. These reactors can effectively achieve the tasks of failure and/or exception handing happening in the reactive module.

As can be seen, in the Interrap's vertical hierarchical architecture, both the control module and the knowledge database are also divided into three corresponding layers. As for the control flow and the data flow, both are unidirectional. The concept of each layer is very clear and the advantages of such an architecture can be summarized as:

---

[4] This is a functionality not often seen in this kind of systems.

- each layer has only access to specific related parts of the knowledge base; this limits the amount of information needed to be accessed to.
- at any given time, there is one and only one layer that is working. This is very important for resources-limited systems. It reduces the loads in single CPU systems; it avoids system "hang-ups."
- it is easy to implement by means of object-oriented methods.

Obviously, there also exist limits in Interrap agents:

- as is true for other vertical architectures, there probably exist robustness problems for the Interrap agent. Mostly it comes from the fact that failure in any module of the system will result in failure of the entire system.
- the problem of transition-time costs is still present. In Interrap agents, the layers' tasks being disjunctive, if a triggering event can be processed in one of the lower layers then the highest layer is never aware of its occurrence. Thus, only after having been unsuccessfully tried at the lowest and the middle layers' capacities, is a triggering event sent to the highest layer. Once processed, it is sent back to the lowest layer again for execution. Because Interrap agents restrict the number of reactive patterns, the majority of the triggering events need to be dealt with by the highest layer or the middle layer. Therefore, Interrap agents become time-sensitive and can even be slow. Furthermore, if an agent meets failure while executing a plan, re-planning become necessary and the process becomes even slower.
- Interrap agents, or agents with vertical architecture, cut down the loads on the CPUs by sacrificing on reaction speed. It is a very effective method for resource-limited agents but given that the processing power of computers increases exponentially with each generation, and the more and more popular use of multi-processors computers, it is quite evident that the Interrap agents will fail to benefit from multi-threads or multi-task distributing processing.
- Interrap agents lack learning mechanisms to improve their performance.

2.3 Intelligent agents with learning abilities

As stated before, intelligent agents with learning abilities fit the highest level of our classification of agents and since no unique concept of "what is intelligence" is so far accepted, we shall tag as "intelligent agents", any cooperative hybrid agents having learning abilities. In fact, we believe that "intelligence" has to be equated with the ability to learn, i.e., the ability to improve one's behaviors from past experiences and activities. We also think that learning mechanisms have to be an integral part of any entity that exhibits intelligent behaviors. We beleive that intelligence without learning cannot be. Consequently, if and only if an agent possesses some learning abilities can it be called an intelligent agent. With this, we concur with Nwana [Nwana 96] when he writes that the minimal set of characteristics for an intelligent agent must contain autonomy, cooperation and learning.

Although there are many kinds of learning methods, artificial intelligence practice proves it difficult if not impossible to find one kind of learning methods that could efficiently satisfy all situations' requirements due to the complexity of the learning procedures. Similarly, it is difficult to find a general intelligent agent model. This explains why intelligent agents' design is mostly problem-oriented.

Chapter 3: IDEAL: a hybrid intelligent agent

Given our design requirements for an intelligent agent, both simple autonomous agent and cooperative hybrid agent architectures, as presented in the preceding chapter, do not provide sufficient support. We thus proposed a new model for an intelligent agent which we call IDEAL and which stands for "**I**ntegrated **D**eliberative and r**E**active **A**rchitectures with **L**earning abilities". This new architecture is based on Muller's Interrap model [Muller 95] [Muller 97] and Rao's BDI interpreter [Rao 95]. It is an hybrid architecture structured in such a way that the functionality and knowledge bases of the agent are layered into several hierarchical modules which define the different properties that an intelligent agent must possess from our point of view. In an IDEAL agent, reactive and deliberative behaviors are integrated into a new architecture in order to achieve some overall coherent behaviors.

There are many reasons that lead us to use a layered architecture. First, by using layered architectures, modular design of agents becomes easy. Next, use of layered architectures makes our agents more compact in architecture. Next, use of layered architectures will increase robustness of the agents and facilitate debugging of agent applications. Obviously, as different layers may run in parallel, by means of using layered architectures the agents computational capacities can be expected to go up. Finally, with a layered architecture, the amount of knowledge to which an agent need to access, at a given time, will be rather small since the needed knowledge will mostly be restricted to the one accessible by the corresponding active layer only.

In the following sections, we described the detail of the IDEAL agent.

3.1 The conceptual architecture of the IDEAL agent

In our point of view, Interrap agents have actually tried to use a methodology similar to that used by people to solve problems. Unfortunately, the level up to which this imitation is conducted stays quite low thus missing important features found in human decision making procedures. The inspiration for the IDEAL agent model comes from our way of life. That is, beginning from a simple attempt, an action's decision making procedure probably goes progressively to deliberative, even cooperative levels, as the complexity of the problems keeps growing.

The IDEAL agent basic idea is to attempt a more closely emulation of the human way of thinking and processing information and, eventually implement it. To understand this, let's look at how people deal with problems they are confronted to.

When a person meets a problem that has to be solved, he possibly can have an "intuitive" feeling of what must be done without even having to consider the details of the problem at hand. There are two possible reasons for this. First, he may "intuitively" feel what he must do without having to refer to anything. He cannot explain why, he just knows. Such mechanisms that entail this kind of feelings are not part of our research although we think that they are related to his field of knowledge. Second, the person may have previously met the same problem or problems with similar natures, which were then solved and their solutions recorded. If we have knowledge from past experiences with a type of problem, the correctness of the solutions is almost guaranteed since the previous solutions have probably been verified in actual applications. For problems without "intuitive" solutions, careful considerations must be given to the parameters describing the problem and, eventually, some solutions may be inferred strictly from his knowledge of the actual situation or from past experiences with similar problems. For more complex problems, he may even need to ask other people for some cooperation in order to solve his problem.

We thus can conclude that a person gets a solution to a problem, if the problem has some solution, in one of the four following different ways: by intuition, from

experience, after deliberation and/or by cooperation. These four approaches will constitute the bases of the functional modules of the IDEAL agent model.

In the following sections, we will first introduce the IDEAL agent's conceptual architecture and focus on the internal states and internal basic functions. Second, we shall present the agent's control architecture. Finally, we will discuss the agent's dynamic characteristics.

3.1.1 The IDEAL conceptual frame

The IDEAL conceptual architecture outlines what an IDEAL agent looks like. It defines the input and output of the IDEAL agent, describes the agent's internal states (mental states), the basic operations that are used to deal with these states, the relations between these operations as well as the relations between the states. The IDEAL agent conceptual architecture (model) is shown in figure 3.1.

As can be seen, the architecture of IDEAL agent is quite complex. All components of the architecture can be divided into two groups: mental states (including operational primitives and libraries) and operational functions. The characterization of the mental states makes use of the modules: **Perception, Beliefs, Goals, Options, Intentions, Event queue, Actions, Plan and pattern libraries** and **History database,** while the control functionality makes use of the processing functions: **recognition, belief generation and revision, goal activation, deliberation, unwind, scheduling, execution** and **learning and modifying.**

Figure 3.1 shows that an IDEAL agent uses different types of internal states: "intuitive", "experience-based", "cooperative" and "deliberative" which correspond to the approaches used in human thinking processes as explained before. This means that there are certain coherent correspondence relations between the

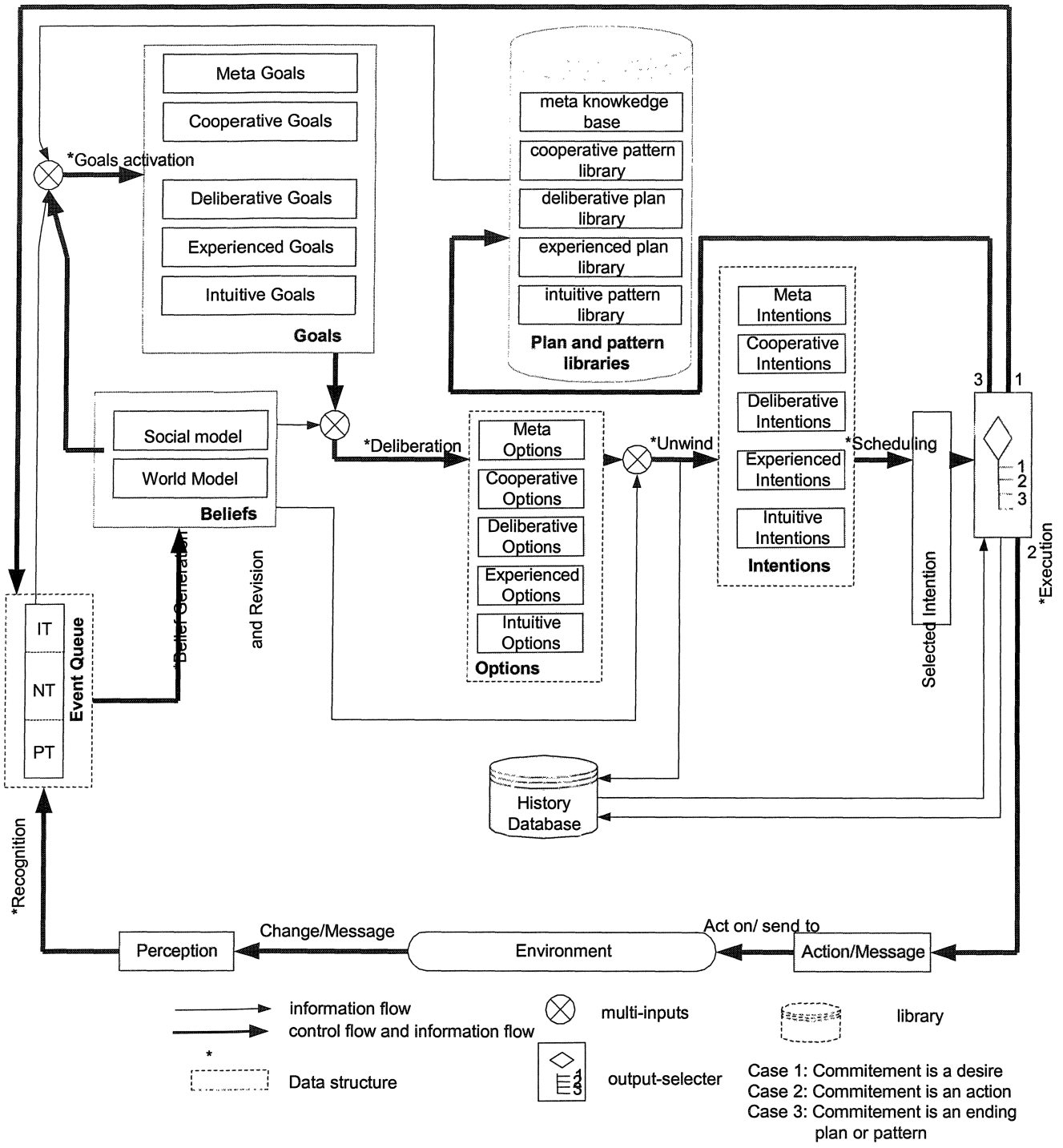


Figure 3.1 IDEAL agent's conceptual architecture

decision-making mechanisms of the IDEAL agent and that of the human thinking processes.

In other words, the conceptual architecture of the IDEAL agent is partly designed from simulating the human way of thinking. To know the meaning of each state, what role the states play as well as how the agent uses basic operational functions to deal with the states, we need to go inside the architecture of the IDEAL agent closer to the components. Once the role and functionality of each state and function will be detailed, it will become apparent that that the design of the agent, from internal states to architecture frame, from operational functions to control modes follows closely the way human decisions are made.

3.1.2 Description of the internal states of IDEAL agent

In the following section, we introduce a list of terms and definitions related to internal states and explain how they work. First we discuss and give details for the states associated with the input and output of the agent; it includes perceptions, negotiation information, events and event queue as well as actions. Then, several important mental attributes: beliefs; desires; goals; options and intentions are presented. We specially establish the differences between the concepts of desire and goal. We also introduce the concept of substitution to define option and we explain the functionality of the intention structure.

Following these, we depict one of the most important parts of the agent's static execution model: the plans and plan libraries. We clearly differentiate between intuitive patterns, cooperative patterns, meta-operation patterns, deliberative plans and experience-based plans. We describe the structures of these plans and patterns and illustrate how the use of connected-rooted-directed-and/or graphs to implement the bodies of the plans and patterns closely emulates executable programs. We also introduce two different algorithms and explain how their use permits completion of real time dynamic planning and execution of reactive non-interruptible patterns. A weight parameter method is used to

optimize the agent's behaviors. The algorithm to evaluate its value is also presented. Finally, the concept of history database is introduced and defined.

3.1.2.1 The Perception and negotiation information

A perception expresses a change in the external world into which the agent evolves. Perceptions can be changes in the environment states, instructions from users and/or messages from other agents. In IDEAL agents, perceptions are implemented as instances of an object class and all sensed changes are in fact processed by methods defined in the class.

As a special case of perception, called ***negotiation_info***, is used to denote negotiation information flowing between agents when a cooperative plan is executed. The negotiation information is only dealt with at the cooperative layer of the agent. In fact, "negotiation_info" is a special type of event introduced in the IDEAL agent model that gives the agents running a cooperative plan a vehicle to negotiate a solution if conflicts should happen. The agents involved in a conflict activate their negotiation mechanisms and make use of negotiation_info events to exchange information in order to resolve the conflict. In a more formal way, we have the following definitions

Definition 3.1 The perception

Let ***P*** be the set of all possible perceptions an IDEAL agent can receive and let p be an element of ***P***. Then ***p*** is a triple ***p*** = (sender, receiver, content) where, ***sender*** and ***receiver*** are natural integers, used to identify the sending and receiving agents. The ***content*** part is also a tuple (message, i_{true}), where the message is either empty or an expression of terms, and i_{true} is the true intention[5]. The constraint that ***sender*** $\neq$ ***receiver*** is also imposed

---

[5] The notion of intention and true intention are introduced later

More generally, P = S × R × Content, with ***S*** and ***R***, respectively the sets of all senders and receivers in the system, and **Content** is the set of all possible **content.**

Similarly, for the negotiation information, we have the following definition.

Definition 3.2 The negotiation information

Let ***Negotiation_Info*** be the set of all possible negotiation information and ***negotiation_info*** be an element of ***Negotiation_Info***. Letting ***exp(φ)*** be an expression whose arguments φ are terms, ***cop*** be a predicate name, ***neg*** a keyword, ***sender*** and ***receiver*** natural numbers, then the formula

neg(sender, receiver, **cop**(exp(φ)))

is a ***negotiation_info***.

3.1.2.2 The Actions

In any specific environment, the IDEAL agent's basic tasks are to achieve some given goals by executing a series of **action**s. In fact, any behavior an agent is able to exhibit corresponds to some action.

More formally, we can write the following.

Definition 3.3 The Actions

Let ***Actions*** be a set of actions and ***action*** be an element of ***Actions***. Let also ρ and φ represent expressions parametred in terms of belief, desire and negotiation_info as arguments, respectively corresponding to optional precondition and the post-condition. Let $x_1, x_2, x_3, \ldots, x_n$ be terms, ***act*** an action symbol and ***cre*** an optional constant. Using ***act_on*** as a keyword, "**;**" as a separating symbol and ***[...]*** as delimitation signs for optional items, then the expression

act_on$([\rho; \varphi;$ *cre;*$]$ ***act***$(x_1, x_2, x_3, \ldots, x_n))$

is called an ***action*** and is denoted ***act_on(act(x))***.

In our definition, the word "action" stands for an implemented function representing a single action that can be called upon by the agent to change the environment or bring itself from one state to another. All action functions are stored in the ***action function library***. The ***actions*** are used in plans and/or patterns. They correspond to the edges of the **connected-rooted-directed and/or** graphs used by the IDEAL agent to represent the bodies of plans or patterns. The optional item **cre** is only used with experience-based plans' bodies. It denotes a measure of the success rate of the action function.

3.1.2.3 The triggering events, events and the event queue

The concepts of events and triggering events are very important in IDEAL agents. An event queue buffers all information interacting with the agent, including information from the environment, from the user, from other agents as well as from intermediate reasoning. The concept of "event" itself greatly simplifies the design of reasoning mechanisms; specialization of some events as "triggering event" makes the agent's reacting mechanism extremely flexible. The following definitions and explanations clarify this.

Definition 3.4 The triggering events, events and the event queue

Let ***add*** and ***dele*** be unary operators and ψ be either a belief, a desire, an intention or a negotiation_info. Then ***add***(ψ) or ***dele***(ψ) is called **a basic triggering event**, denoted **bt**. **BT** represents the set of all possible **bt**s. Let ϕ be a desire or an action, ***i*** an intention and ξ a basic triggering event, then ***succ_desire***(ϕ), ***fail_desire***(ϕ), ***succ_plan***(ξ), ***fail_plan***(ξ), ***add_int***(***i***) *and* ***delete_int***(***i***) are called **quasi-triggering events**. The set of all possible *quasi-triggering events* is denoted **NT**. Let **TE** be the union of **BT** and **NT**, then any member of **TE** is called a **triggering event**. Let ***eq*** belong to **TE** and **i** be an intention, then the tuple (**eq, i**) is called an **event**.

In an IDEAL agent, all events active at a given time are placed into a data structure called the **event queue** and denoted **E**. For a given event **(eq, i)**, where **eq** is an element of BT and **i** is the **true intention** (denoted as $\mathbf{i}_{true}$), then **eq** is called a **premier triggering event**. Sometimes premier triggering events are called **external events**. We use the notation **pt** to denote premier triggering events. For events **(eq, i)** not having **i** = $\mathbf{i}_{true}$, **eq** is called an **intermediate triggering event** and denoted as **it**. These are also referred to as **internal events**. The set of all **pt**s and the set of all **it**s are respectively written as **PT** and **IT**. In fact, we have the following relations:

$$\mathbf{BT} = \mathbf{PT} \cup \mathbf{IT},$$

$$\mathbf{TE} = \mathbf{BT} \cup \mathbf{NT} = \mathbf{PT} \cup \mathbf{IT} \cup \mathbf{NT}.$$

Premier triggering events correspond to perceptions from external sources while intermediate events are the ones generated by the means-ends reasoning processes used by the agent for the treatment of some goal. As for quasi-triggering events, intermediate events do not activate any "real" goals. These "quasi goals", could we say, correspond to updates of the agent's internal states.

By using an event queue containing different types of information rather than simple invocations of goals as is done in logical programming, the IDEAL agent's triggering mechanism supports data-driven modes such as ***add****(belief) or* ***dele****(belief)*, goal-driven modes such as ***add****(desire) or* ***dele****(desire)* and means-end reasoning modes such as *(**dele**(desire), $i \neq i_{true}$)*. This gives the IDEAL agents great flexibility.

Furthermore, from our definition of an event we know that all events contain an associated intention. This expressive mode easily saves control point information in both a reasoning procedure and intention execution procedure. Therefore, it reduces the difficulties of designing intelligent agent with real-time reaction abilities and thus allows IDEAL agents to challenge other famous agents in terms of performance.

3.1.2.4 The Beliefs

As can be seen in Fig. 3.1, IDEAL agents' beliefs are divided into two complementary sets: B_w, containing the beliefs related to the world model and, B_s containing the beliefs related to the social model. In fact we can write:

$$\text{Beliefs} = B_w \cup B_s.$$

The world model includes the static properties of the environment, those of the agent itself as well as any inferred results that have been obtained by the execution of intentions. As for the social model, it includes information about other agents such as their configurations, transitions protocols and so on. The beliefs databases are structured in a layered way, the lower layer consisting of the world beliefs, the higher layer of the social beliefs. The basic objective of this layering is to reduce the costs of access. Obviously, reducing the amount of beliefs needed to be accessed will speed up the agent's reasoning process.
In fact, all the modules of an IDEAL agent are layered and each layer can access only the part of the beliefs databases it is related to. The levels of abstraction are defined as increasing from bottom to top. Since abstraction goes up from world model to social model, permission to access at some higher level also gives access permission to all other lower levels. For instance, any layered component having access to the social level has also access to the world model. On the other hand, if access is at the world model, the agent cannot access the social model. More formally, beliefs are defined as below.

Definition 3.5 The beliefs

Let ***Belief*** be the set of all available beliefs, ***belief*** be an element of ***Belief*** and ***Bel*** be a subset of ***Belief***, which represents the current beliefs of the agent at time ***t***. Given that $x_1, x_2, x_3, \ldots, x_n$ are terms, **bel** is a predicate name and **fact** is a keyword, then the formula **fact**(**bel**($x_1, x_2, x_3, \ldots, x_n$)) is a belief. More compactly, it is **fact**((**bel**(**x**)). Note that **fact**(**bel**(**x**)) is called a belief atom. The

negation of a belief, ¬ **fact**((**bel**(**x**)), and the conjunction of different beliefs, **fact**((**bel**(**x**)) ∧ **fact**((**bel**(**y**)) are also beliefs. Belief instances or facts are often called ***ground beliefs***.

3.1.2.5 The desires and goals

Desires describe behaviors an agent wants to exert or some states it wants to attain. Conceptually, if a desire is a realizable in a given context, it is called a goal. Therefore, sets of goals are always subsets of the desires. If a desire can activate a plan or a pattern which enables the agent to implement the desire, then we say it is an "implementable" desire. In other words, it becomes a goal. On the other hand, some desires may never be attainable for a specific system; when such is the case, the system would drop them. Thus, in IDEAL agents, desires only appear in the perception and reasoning processes (from the event queue). The module **Goals** is only used to represent track of goals and desires. In fact, goals are temporal variables affecting the agent's decision-making procedures. Once selected, they become part of an intention. In the agent's design, the goals are thus treated as temporal structures.

Goals are classified[6] as deliberative (G_d), experience-based (G_{ep}), intuitive (G_i), cooperative (G_c) and meta goals (G_m). The set of all possible goals is defined as

$$Goals = G_d \cup G_{ep} \cup G_i \cup G_c \cup G_m$$

This goal classification comes from the fact that the IDEAL agent possesses five different control layers, as will be shown below, and goals are always linked to the control layer from which they have been activated. The functionalities of the layers are different and exclusive, so the goals are also exclusive. For any specific triggering event, one and only one specific type of goals is activated and they are transferred in the corresponding control layer. There is however an

---

[6] The subscripts are d as deliberative, ep: experience-based, i: intuitive, c: cooperative and m: meta-reasoning. They are used to distinguish the different types of entities in relation to the layer they are connected to. Here they distinguish the goals.

exception to this rule and it happens when a triggering event will simultaneously generate goals linked to the deliberative layer and goals linked to the experience-based layer. This can happen when some experience-based intention, i_{ep}, can be used as an alternative to a deliberative intention i_d. However, when it happens, the G_d set of goals is filtered out at the scheduling stage. Formally, we give the definitions of desire and goals as below.

Definition 3.6 The desire

Let ***Desire*** be the set of all possible desires and ***desire*** element of ***Desire***. Let the prefix operators ∇ represent **!**, **?** or **~** respectively meaning *achieve*, *test if true* and *wait until true*. Let $x_1, x_2, x_3, \ldots, x_n$ be terms, ***gol*** a predicate name and ***des*** a keyword, then the formulas

des(!gol(x_1, x_2, x_3, …, x_n)),

des(?gol(x_1, x_2, x_3, …, x_n)),

des(~gol(x_1, x_2, x_3, …, x_n))

are ***desires***.

We use the notation ***des***(∇***gol***(***x***)) to represent a desire in its general form. Usually the desire ***des(~gol(x))*** has a time parameter defining the duration of the waiting period.

Since goals are "realizable" desires, we need a different definition for goals. They are defined in the following way.

Definition 3.7 The goals

Let ***de*** be a desire and ***pp*** be a partially instantiated plan or pattern –the distinction between the two comes further below– triggered by the desire in the current context. The pair ***(de, pp)*** is called a narrowed goal. The use of the prefix operators applied to the desires subdivides the goals ***achieve goal***, ***test goal*** and ***wait ... until goal*** respectively corresponding to ***!goal***, ***?goal*** and ***~goal***. More generally, if we let ***e*** = ***(eq, i)*** be an event and ***pp*** be a partially instantiated

plan or a pattern triggered by event ***e*** in the current context, then the pair ***(e, pp)*** is called a general goal, or simply, a goal. We know that event ***(eq, i)*** is made of a triggering event ***eq*** and an intention ***i***.

We have a particularity situation when ***eq*** $\in$ ***NT***. In these cases, there is no ***pp*** connected with the event. However, for the sake of convenience, we use the predefined "pseudo" plan or pattern written as ***False***; the resulting goal ***(e, False)*** a called a ***pseudo-goal***. A NT event never activates any plan or pattern but it still brings some feeds-back to the agent's behavior and activates an internal reasoning procedure. For such events, the agent only intends to commit the intention ***i*** specified by the event ***e***.

Use of different types of goals is one highlight feature of the IDEAL agent architecture. For example, since ***test goal*** and ***wait ... until goal*** are connected with plans instead of atomic actions, they can be used to implement complex test or recognition actions. In [Rao 94], Rao proposed the concept of "recognition plan" where all plans are divided into two types: execution plans and recognition plans. For different type of plans, the agent uses different interpretation and control methods. When an agent is executing a recognition plan, the system is in a suspended state which renders it unable to react in a real-time fashion to changes in the environment that are out of the range of the recognition plan. When compared with Rao's approach, our method is much simpler and more efficient and the disadvantages of Rao's model are truly overcome.

3.1.2.6 The options

Options represent the set of goals the agent will most possibly commit to. Conceptually, options are the most potential context-dependent goals. It means that an option is a goal the agent wants to achieve and the agent believes that goal to be implementable[7] from the given current system state and that among all goals activated by a triggering event, that goal is the most worth. The concept of

---

[7] By implementable we mean that a plan to achieve the goal exists and can be activated.

"the most worth" can be explained as follows. From definition 3.7, we know that a goal always includes a plan or pattern that specifies how the goal or pattern is to be achieved. In IDEAL agents, weights have been introduced as a measure of the correctness, success rate and costs of realizing plans and/or patterns. The plans and/or patterns having the highest value of weight are said to be the most potential ones. Saying that a given goal is an option implies that the plan included in the goal is the most potential.

Just as the goals, options are classified in correspondence with the layer from which they were generated. Hence, we write the total set of options as

$$\textbf{Options} = \mathbf{O_d} \cup \mathbf{O_{ep}} \cup \mathbf{O_i} \cup \mathbf{O_c} \cup \mathbf{O_m}$$

A more formal definition of options is given below.

Definition 3.8 The options

In order to be more formal in our definition of an option, we must first recall the notion of substitution. A substitution is defined as follows.

Let $I = \{1,2,3, \ldots, n\}$ be a finite set of natural integer and, $\forall i \in I$, let v_i and t_i respectively represent variables and terms, such that for all j, $j \in I$ and $i \neq j$ we have $v_i \neq v_j$ and $t_i \neq t_j$. If $\forall i \in I$, $v_i \neq t_i$, then the finite set

$$\alpha = \{v_1/t_1, v_2/t_2, \ldots, v_i/t_i, \ldots, v_n/t_n\}$$

is called a substitution. If $\alpha = \{v_1/t_1, v_2/t_2, \ldots, v_i/t_i, \ldots, v_n/t_n\}$ is a substitution and $m = f(\mathbf{v})$ is a formula where $\mathbf{v} = (v_1, v_2, \ldots, v_i, \ldots, v_n)$, then applying substitution α to $f(\mathbf{v})$ means simultaneously replacing every v_i in $f(\mathbf{v})$ by its corresponding t_i as given by α. We usually write $\alpha \cdot f(\mathbf{v}) \rightarrow f(\mathbf{t})$, where $\mathbf{t} = (t_1, t_2, \ldots, t_i, \ldots, t_n)$. The notion of substitution also implies that $\forall i \in I$, we have $\alpha \cdot v_i \rightarrow t_i$.

Now, let ***Bel_ins*** be a set of the agent's current beliefs instances and $\boldsymbol{p_1}$, $\boldsymbol{p_2}$, ..., $\boldsymbol{p_n}$ be plans and/or patterns. Let ***e*** be an event, $\boldsymbol{p_{max}}$ be the partially instantiated

potential plan or pattern associated with the event. The pair **(e, p_{max})** is then called an option.

Conceptually, a most potential plan is one of the weightiest applicable plans for a triggering event ***e***. Let there be substitutions η and φ such that for all substitutions $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ we have $\alpha_1 = \eta_1\varphi_1$, $\alpha_2 = \eta_2\varphi_2$, ..., $\alpha_n = \eta_n\varphi_n$, and such that for the set **plan** = $\{p_1, p_2, \ldots, p_n\}$ we have

$\eta_1 \cdot inv_1 = \eta_2 \cdot inv_2 = \ldots, = \eta_n \cdot inv_n = eq$ and,

$\alpha_1 \cdot pre_1 \in$ Bel_ins, $\alpha_2 \cdot pre_2 \in$ Bel_ins, ..., $\alpha_n \cdot pre_n \in$ Bel_ins.

Then **plan** is called a set of applicable plans for event ***e*** = (***eq, i***). In the above relations, for $j \in \{1,2,3,,,,n\}$, pre_j and inv_j are respectfully the precondition and invocation condition of plan p_j. Let maximum() be a function which selects from the set **plan** all plans having the highest weight value and let random() be a function which randomly selects a plan from the result of function maximum(). Then, p_{max} = random{maximum(**plan**)} is called a most potential plan. The most potential pattern is defined in a similar manner.

Just like goals, options are temporal variable used in an agent's decision-making procedure. Therefore, options are set as temporal structures in IDEAL agents.

3.1.2.7 The intention and the intention structure

Intentions are a subset of the goals in which the plans or the patterns have already been partially instantiated by the agent and the action to be executed has already been selected from the body of the plan according to the agent's current mental states. In fact, it explains how the agent does to achieve its goal. More clearly, we have

Definition 3.9 The intention and intention structure

Let p be a deliberative plan (see definition 3.11) and $p_1, p_2, \ldots, p_p$ be partially instantiated deliberative plans that are sub-graphs of p. Let **stack(v**[p_1| p_2| ...| p_p]) be a stack where $p_1, p_2, \ldots, p_p$ are the stacked elements, separated by "|", with p_1 at the bottom of the stack and p_p on the top, then this stack is called an

intention, which we shall denote ***i***. Here, **stack** is a keyword and ***v,*** a character string, is the name of the intention. In other words, an intention is an ordered set of partially instantiated plans or sub-graphs. Let ***I*** be a data structure containing the set of all current intentions $\{i_j\}$ at given time, then ***I*** is called the ***intention structure***. For a given event that has generated an intention –the plan selected to react to the event– the information about variables and substitutions, and the control points, all together form the **intention frame**.

Since experience-based plans and intuitive, cooperative and meta reasoning patterns –defined further below- can be viewed because of their structures as special cases of deliberative plans, the definition of intention is also available for use with these plans and patterns.

We give particular names to the following intentions:

stack(**v**[true: [true]:→True]) is called the ***true intention*** and is denoted $\mathbf{i_{true}}$,

stack(**v**[true:[true]:→ False]) is called the ***false intention*** and is denoted $\mathbf{i_{false}}$,

stack(**v**[i_{true} | inv_1: pre_1: → true]) is called the ***critical true intention*** and is denoted $\mathbf{i_{c\text{-}true}}$,

stack(v[i_{true} | inv_1: pre_1: → false]) is called the ***critical false intention*** and is denoted $\mathbf{i_{c\text{-}false}}$.

In these intentions, inv_1 and pre_1 are respectively the invocation condition and precondition of the first level plan, i.e. the top-level plan, of the intention stack.

In IDEAL agents, the concept of intention is very important. These stacks are used to track, in real time, the options generation, the means-end reasoning procedure, the changes in variables bindings and the moving of control points. Each intention stack ***i,*** in the intention structure ***I,*** denotes a separate process or task. This property enables the agent to deal with several events simultaneously present in the event queue as opposed to only one as done in [Brahman 87] [Weiss 99]. Obviously, our agents perform at a higher efficiency.

The intention structure ***I*** can store intentions of different types such as deliberative, experience-based, intuitive, cooperative and/or meta-reasoning since the plans and patterns libraries are non-homogeneous.

As for the relations between intentions and events in the event queue, we have the following explanation. An external event that belongs to the set of feasible events will necessarily generate one applicable top-level plan. Hence a new intention is created with i_{true} as the bottom of the stack and the plan is pushed on top of i_{true}. Finally, the intention is added to the intention structure.

An internal event that belongs to the set of feasible events will only generate one related sub-plan, namely, create an intention frame. The frame is then pushed on top of the intention which generated the internal event. When the body of the element that becomes the top of the intention stack is true or false, the associated intention frame is popped away. It means the end of a sub-plan, i.e., the processing of an internal event has ended. When an intention becomes a critical intention, it corresponds to the end of the processing of an external event. The top-level plan is popped away. If an intention is a true or false intention then the intention stack is deleted and post-processing begins.

3.1.2.8 The intuitive, cooperative and meta-operation patterns

Plans and patterns libraries play are critical in IDEAL agents. They define the space of all available solutions the agent is provided with. They contain the sets of all behaviors the agent can possibly use to react to its perceptions. We may say that the libraries reflect the agent's working capacities.

The different structures of plans not only result in different types of goals, options and intentions, but also justify the necessity of having different structures of control layers. The plans libraries of an IDEAL agent involve two types of structures of plans:

- goal-driven plans[8], used for long-term and complex tasks and
- data-driven reactive patterns, used for short-term needs and fast response tasks.

---

[8] Later, we will see that they are also data-driven in IDEAL agents

The non-homogeneous nature of the libraries brings flexibility to IDEAL agents and offers them sufficient support to operate in dynamic and unforeseeable environments.

In this section, we discuss reactive patterns which can be of three kinds:

- intuitive patterns (***ip***), used to "intuitively" deal with simple and unforeseen events,
- cooperative patterns (***cp***), used when "negotiate" events happen, and
- meta-operation patterns (***mp***), used to provide knowledge about how to select operational primitives to deal with the internal states of the agent.

Patterns are sequences of operational primitives that provide the IDEAL agent with the ability to choose and execute appropriate actions in a context-free manner and thus exhibit reactive behaviors. More precisely, we define patterns as below.

Definition 3.10 The intuitive, cooperative and meta-operation patterns

Let ***pt*** = (pt_1, pt_2, …, pt_p) be a set of premier triggering events, ***fc*** = (fc_1, fc_2, …, fc_m) and ***sc*** = (sc_1, sc_2, …, sc_j) be sets of beliefs or desires where p, m and j are natural integers. Let also ***ct*** be a constant, ***v*** be a character string, ***ty*** be a positive integer and the triplet ***(action, list[node]$_n$, list[edge]$_n$)*** represent a ***connected-rooted-directed and/or graph***, (abbreviated as CRDAO-graph) for ***$n \in N$***.Then

$$v(exp(pt):([exp(fc)];[exp(sc)];ct;ty;pattern\_type) \rightarrow (action, list[node]_n, list[edge]_n))$$

is called a pattern.

Here, ***v*** is the name of the pattern and the expression ***exp(pt)*** is called the **invocation condition** of the pattern. The two expressions ***[exp(fc)]*** and ***[exp(sc)]*** are optional parameters and are respectively called the **maintaining**

condition and the **success condition**. Also, ***ct*** is called the weight of the pattern and is given by ***ct = (cre-tc)***, where ***cre*** represents the credit of the pattern and ***tc*** is the time cost associated to the execution of the pattern. The positive integer ***ty*** is used to differentiate the types of patterns. Value of **4, 5** or **6** correspond to a cooperative patterns, values equal to **7, 8** or **9** represent a meta-operation patterns and values of **10,11,** or **12** are associated with intuitive patterns. The ***pattern_type*** parameter defines the mode of execution of the pattern. If it equals 0, then the acting executor will not check the maintaining condition or the preconditions of actions in the pattern. On the opposite, for a value of 1, the acting executor will always check these conditions.

The triplet ***(action, list[node]$_n$, list[edge]$_n$)*** is called the **body of the pattern**. In this triplet, ***action*** corresponds to what the agent will commit to execute while ***list[node]$_n$*** and ***list[edge]$_n$*** form the adjacency-list representation of a CRDAO-graph. In the graph, each edge is an ***action***. Under the constraint of the invocation condition of the pattern, each path, starting at the root node and ending at a leaf node, expresses an available sequence of actions that would execute the pattern. The invocation conditions are also known as the head of the pattern, the rest of the parameters: v, exp(fc_1,fc_2,..., fc_m), exp(sc_1, sc_2, ..., sc_j) and ct, all together, are referred to as the description of the pattern. For the sake of simplicity and clarity, we will sometimes omit the detailed description and refer to a pattern using the more compact form **[head → body]** and will write it as **[invocation → (action, list[node]$_n$, list[edge]$_n$)]**.

Let **IP** be a set of intuitive patterns and **ip** be an element of **IP**, then **IP = {ip}** represents the intuitive patterns library **IPL**. Similarly, **CP = {cp}** and **MP = {mp}** are respectively the cooperative patterns library **CPL** and the meta-operation patterns library **MPL**, also, called the meta-reasoning patterns library. **IPL**, **CPL** and **MPL** are in fact storage units. The three kinds of patterns work in a data-driven mode.

Once an event in the event queue is matched with the ***invocation condition*** of a pattern, then that pattern is activated. The pattern's name is used as its identifier. The ***success condition*** represents the state the agent will have achieved or attained after execution of the pattern. Except for the non-interruptible intuitive patterns used to process failures (pattern_type = 0), patterns are executed in a stepwise manner, one action at the time. This permit taking into consideration any changes that may have happened while one action was executing. Before the execution of the next action function, the execution controllers will check whether the ***maintaining condition*** is matched; if and only if the ***maintaining condition*** of the pattern is kept satisfied, will the execution of the pattern continue. The type of pattern, given by ***ty***, is used by the scheduling procedure to establish the priorities of patterns and plans present in the intention structure. The credit ***cre*** is a measure of a pattern's success rate. As for the weight ***ct***, it becomes important in the deliberative processes where the agent uses it to select the appropriate means to achieve a given goal. As previously presented, we know that ***ct = (cre-tc)***, where ***tc*** is the time cost associated to the execution of the pattern. From the definition of the weight ***ct***, we see that:

- for a given time of execution, the higher is the weight of a pattern, the higher is its success rate;
- for a given value of credit, the higher the weight of a pattern is, the faster is the pattern's execution time.

Consequently, among all activated candidate patterns for a given event, the appropriate pattern corresponds to one of those having the highest value of weight.

The value of ***tc*** is not known in advance and thus must be estimated. Evaluating ***tc*** is quite a complex problem given that the IDEAL agents' behaviors are context-sensitive. In general, the body of a pattern is a CRDAO-graph. Completion of a pattern generates a unique path from the root node to one of the leaf nodes as shown below in figure 3.2.

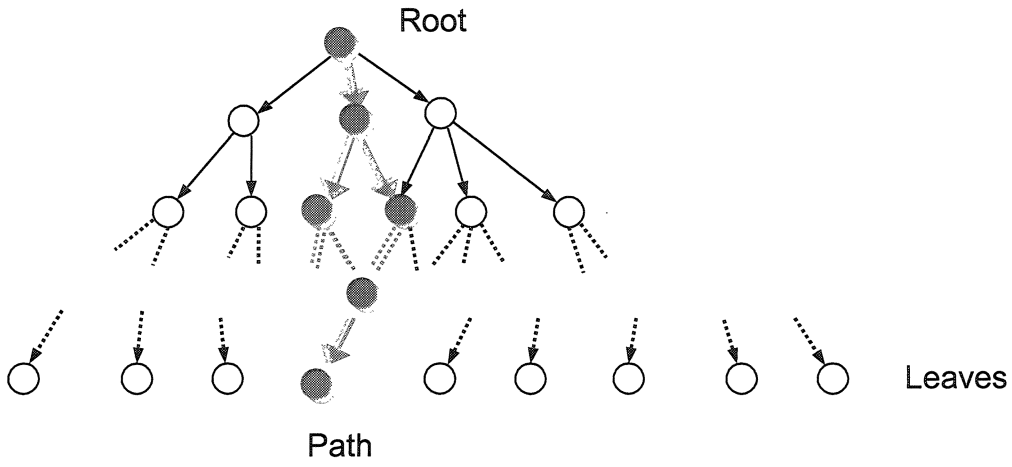


Figure 3.2 A path in a pattern's body

Starting at the root node, with the invocation conditions, the next node of the path is found by a depth-first algorithm using mental states. Hence, for different intermediate mental states, the path may possibly be different and the execution time varies accordingly. Therefore, the value of ***tc*** is approximated by the method defined below.

A method for approximating the value of ***tc***

Let **tc(ga)** be the value of the time costs for the execution of action **ga**[9] and let us consider the path characterizing the completion of a pattern. We have the following possible situations:

- for a node that is the starting point of a sequence of k nodes where each node has only one exiting edge, then running the node will cost

  $\mathbf{tc_s(node)} = \Sigma_{j=1 \text{ to } k}\ \mathbf{tc(ga_j)}$,

  where $\mathbf{ga_j}$ is the action associated with edge **j**;
- for an ***and***-node with k actions, all actions must be run, hence the cost of the node is:

---

[9] When the action **ga** is a **~ wait...** action, we define tc(ga) as its control time.

$\mathbf{tc_{and}(node) = \Sigma_{j=1\ to\ k}\ tc(ga_j)}$,

where $\mathbf{ga_j}$ corresponds to action **j** starting from the ***and***-node;

- for an ***or***-node with k possible actions, the node is successfully passed when one of the k actions is successful. In the worst case, all actions have to be tried before finding the successful one and the cost would be given by $\mathbf{tc_{or}(node) = \Sigma_{j=1\ to\ k}\ tc(ga_j)}$. In the best case, it would be $\mathbf{tc_{or}(node) = tc(ga_j)}$, namely the cost of executing only once the action $\mathbf{ga_j}$. We thus make a tradeoff by using $\mathbf{\frac{1}{2}(\Sigma_j\ tc(ga_j)\ for\ j=1,\ k)}$ the cost of the average case. Furthermore, when there is a wide gap between the $\mathbf{tc(ga_j)}$ values, we can have $\mathbf{tc_{or}(node) < tc(ga_{max})}$, where $\mathbf{ga_{max}}$ is the action whose cost is maximum. When such is the case, we directly pose the cost of the ***or***-node to be $\mathbf{tc(ga_{max})}$. Hence running the node will cost

  $\mathbf{tc_{or}(node) = Max\{\ \frac{1}{2}(\Sigma_j\ tc(ga_j)\ for\ j=1,k),\ tc(ga_{max})\}}$.

Recursively, we have the following situations:

- for an **and**-node that is the root of k sub-trees, running the node will cost

  $\mathbf{tc_{and}(node)_{st} = \Sigma_{j=1\ to\ k}\ tc(st_j)}$

  where $\mathbf{tc(st_j) = tc(ga_j) + tc(node_{st})}$.

  ga_j is an edge whom starting point is the root of the subtree, i.e., current and-node; $node_{st}$ is the end of ga_j; st_j is one of the sub-trees whom roots are the current and-node;

- for an ***or***-node root of k sub-trees, then running the node will cost

  $\mathbf{tc_{or}(node)_{st} = Max\{\ \frac{1}{2}(\Sigma_j\ tc(st_j)\ for\ j=1,k),\ tc(st_{max})\}}$

  where $\mathbf{st_{max}}$ is the sub-tree whose cost is maximum;

  normally, $tc_{or}(node)_{st} = 1/2 \bullet \Sigma_{j=1\ to\ k}\ tc(st_j)$; otherwise, if $tc_{or}(node)_{st} < Max(tc(st_j))$, then $tc_{or}(node)_{st} = Max(tc(st_j))$. Here, calculation of $tc(st_j)$ is the same with above mentioned.

We can finally write that:

- a path with an ***or***-node as root has an associated cost of **tc(p) = $tc_{or}(node)_{st}$** and
- a path with an ***and***-node as a root will cost **tc(p) = $tc_{and}(node)_{st}$**

The last element that needs to be explained, in the definition of patterns, is the ***Body of a pattern***. The body of a pattern specifies the means used to achieve the goals of the pattern by a combination of actions sorted in temporal order. Figure 3.3 illustrates the body of a pattern and its expression by adjacency-lists. The representation of the body of a pattern is a **connected-rooted-directed and/or graph**. The edges of the graph are labeled with atomic actions[10], such as **test (?), achieve (!), wait_until (~)**, ... The starting and ending nodes of an edge represent the begining and ending of an atomic action. The graph is connected, meaning it cannot be separated into independent sub-graphs. It is also rooted which means that it has a unique distinguishable starting node. Finally, directed graph means that the graph unwinds from the root node downward to a leaf node by following each edge. The ***and-or*** means that the edges exiting a node can either be ***and*** relations or ***or*** relations. We use adjacency-lists of edges that flow away from nodes to represent a CRDAO-graph.

When the execution of the body is successfully completed, a path is generated starting at the root node and ending at one or more of the leaf nodes. More precisely, if the last action comes from an **or**-node then the path ends with a single leaf node; if, however, the final actions are executed from an **and**-node then the path ends with many leaf nodes

[10] An atomic action is an action that cannot be divided into sub-actions.

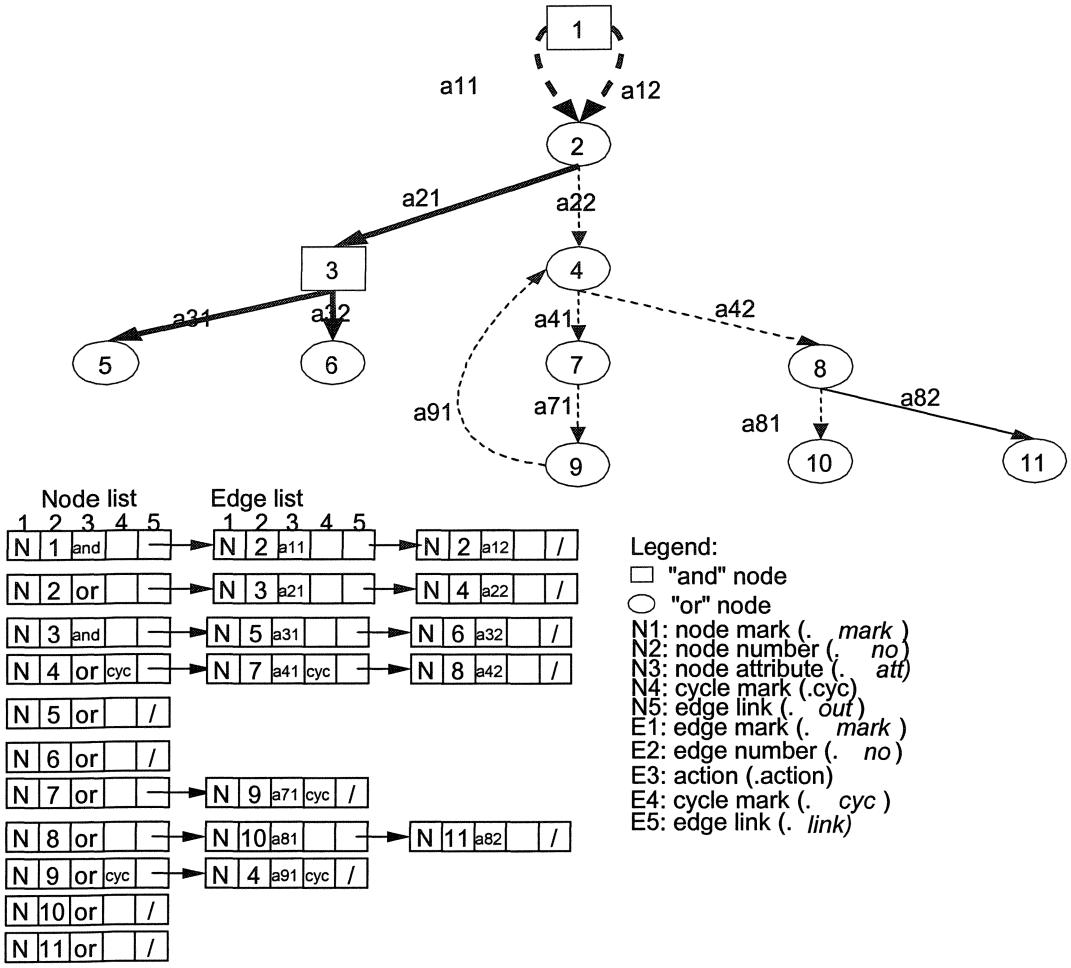


Figure 3.3 Body of a pattern and its representation of adjacency-list

In figure 3.3, the bold lines and the dashed ones outline two different paths. Note that if an or-node is in the path, then only one edge starting from this node is in the path contrary to **and**-nodes where all edges starting from the node are in the path. The generation of a path is closely related to the control algorithm used to run the body of a pattern. In IDEAL agents, the control algorithm is depth-first algorithm and is called **march-step** function. Details and comments of this function are given in the appendix A.

Below is given an example of execution for the path corresponding to the dashed lines in the figure 3.3. Let us suppose that

- the time to execute the "***execute***" method corresponds to **t**;
- the pattern_type is 1, so the verification of the maintaining condition is omitted;
- conditions a_{11}.con, a_{12}.con, a_{22}.con, a_{71}.con, a_{91}.con, a_{42}.con, a_{81}.con are always true,
- condition a_{21}.con is false for the time interval $[t_2, t_4]$,
- condition a_{41}.con is true for $[t_1, t_5]$, but becomes false at time t_6.

At time t_1, action a_{11} is started and since condition a_{11}.con = T, the action a_{11} is successfully executed and marked as done. The control goes to right since node 1 is an ***and***-node and thus is not a restriction-free node. At time t_2, a_{12}.con = T and action a_{12} is executed successfully and is also marked as done. Since all actions starting at node 1 are marked, node 1 becomes restriction-free and the control goes down to node 2. Node 2 is restriction-free, as it is an ***or***-node. At time t_3, the condition a_{21}.con = F so the execution of action a_{21} fails. The control goes to the right, to try executing action a_{22} defined as an alternative to action a_{21}. Since a_{22}.con = T, action a_{22} is executed successfully at time t_3 and is marked as done. The control goes to node 4 and since at time t_4, a_{41}.con = T, the action a_{41} is executed successfully. However, action a_{41} is not marked as done because it is part of the cycle 4 → 7 → 9 → 4. The control goes down to the next level, at node 7, according to depth-first algorithm. At time t_5, a_{71}.con = T, so the action a_{71} is executed successfully but it is not marked for the same reason as action a_{41}. The control goes down to the next level at node 9. At time t_6, a_{91}.con = T and the action a_{91} is executed successfully and is also not marked. The multi-threads, multi-tasks technologies should be used as much as possible.control goes up to node 4, which corresponds to the end-node of the action a_{91}. Note

that a cycle[11] has been formed. At time t_7, conditon a_{41}.con = F and the action a_{41} execution fails. The cycle is ended and the control goes right to action a_{42}, the alternative to action a_{41}. The action a_{42} is executed successfully, since a_{42}.con = T and it is marked as done. The control goes down to node 8, the next level according to depth-first algorithm. At time t_8, since a_{81}.con = T, the action a_{81} is executed successfully and is also marked. Next, the control goes to node 10, which happens to be a leaf-node. Since node 8 is an or-node, the pattern has thus been successfully completed. The Boolean value T is returned to the execution controller.

It must be emphasized that the relative positions of the edges of the ***and/or*** graph are important. Since the march-step function is based on a depth-first algorithm, searching goes from top to bottom starting at the left and for a given level, if an horizontal shift is necessary, is goes from left to right. Thus, when introducing alternative actions which we want to be ordered by priority, the leftmost one has the highest priority and the left to right order should correspond to the decreasing order of priority.

It must also be noted that there are some special cases for which the "***execute***" method does not have to check the maintaining conditions and preconditions. In such cases, the path of execution would correspond to all leftmost edges of the graph, starting with the root-node down to the leftmost leaf-node. The path formed of the bold lines in the figure 3.3 illustrates such a situation.

The action's preconditions are an essential factor in determining the direction of a path. A path always goes forward, in a depth-first manner, along the edges for which the actions' preconditions are true. Obviously, when using the march-step function as defined above, *a pattern's body implemented by a connected-rooted-directed* ***and/or*** *graph closely simulates basic program structures and its implementation is very straightforward.* We need only precise that a cycle always

---

[11] This cycle was introduced by the application developer. We suppose some condition will eventually change so that the cycle will break; otherwise the developer has not done his job properly. In our example, condition a_{41} changes.

begins with an ***or***-node that has at least two exiting edges. The first edge of a cycle usually controls the execution of the cycle which stops when its condition becomes false. Immediately to the right of the starting edge of a cycle, there must be another edge to provide an exit to the cycle.

Finally, we emphasize the fact that the number of patterns needs to be limited if the agent is to exhibit good reactive speed.

3.1.2.9 The deliberative plans

The most important component of the IDEAL agent are the **plans** which can either be deliberative (***dp***), enabling the agent to deal with complex problems, or experience-based (***ep***) that replace ***dp*** plans for events the agent deals with and that often happen. Experience-based plans will be introduced in the next section. Let us first see how deliberative plans operate.

A deliberative plan is a special combination of operational elements of the agent. It can be viewed as an abstract structure, or recipe, helping the agent short-circuit time costs related to searching through large solutions space. It gives the agent a context-sensitive perspective enabling its decision making process to perform in such a manner that it speeds-up its computational reasoning capacities. More precise and formal details are given below.

Definition 3.11 The deliberative plans

Let $te_1, te_2, \ldots, te_p \in$ **TE** be triggering events. Let also $pre_1, pre_2, \ldots pre_m$ and $po_1, po_2, \ldots po_j$ be beliefs or desires, ***p***, ***j*** and ***m*** be natural integers. The parameters ***v***, ***ct***, ***cre***, ***tc***, ***ty*** have the same meanings as those used for the definition of ***patterns***. Let ***plan_type*** be a variable that takes either the value ***variable*** or the value ***invariable***. The triple $(da, list[node]_n, list[edge]_n)$ represents a connected-rooted-directed and/or graph with edges, labeled da_k for k = 1, 2, 3 …, representing either desires or atomic actions. Then a deliberative plan ***v*** is written as:

v ((exp(te_1, te_2, ..., te_p):exp(pre_1, pre_2, ...pre_m)):

([exp(po_1, po_2, ...po_j)];ct;ty;plan_type)

→ (da, list[node]$_n$, list[edge]$_n$)).

The expression ***exp(te_1, te_2, ..., te_p)*** is called the ***invocation condition*** while ***exp(pre_1, pre_2, ..., pre_m)*** is the precondition[12]. The expression ***exp(po_1, po_2, ..., po_j)*** represents the post-condition and is optional. The ***plan_type*** parameter indicates the mode of execution. The ***ty*** parameter is used to differentiate the execution priorities of plans. The integer values of 1 to 3 identify plans that may be deliberative or experience-based. As presented before, values 4 to 12 are reserved for patterns. The triplet ***(da, list[node]$_n$, list[edge]$_n$)*** is called the **body** of a plan.

The concept of "path" can also be used in the body of a plan in the same way as it is used for pattern's body. The only difference is that the parameter ***da*** may be a desire or an action, instead of a strictly atomic action as defined for patterns. We often casually call the invocation condition and the precondition part, the **head** of the plan. The plan name ***v***, the post-condition, weight, ty and plan_type together are referred to as the **description** of the plan. Just as we did for patterns, in some situations we shall omit the description of the plan and only use the head and body parts to express a plan. For example,

[invocation : precondition: → (da, list[node]$_n$, list[edge]$_n$)]

represents a plan.

In the definition of a plan, ***invocation condition*** describes the situations where the plan can be triggered. Notice that the invocation condition is not restricted to premier triggering events as it was for patterns; it can be any triggering events. Hence, deliberative plans can be used to deal with any type of input. This makes the deliberative layer is a kind of "omnipotent" layer.

We must emphasize that only deliberative plans can deal with intermediate events which, in fact, can correspond to invocations of some reasoning

---

[12] Most of the times people call it the **context**.

procedures. Consequently, the executions of deliberative plans can be seen as reasoning processes. It is obvious that deliberative plans need and have stronger and better information processing abilities than do patterns. Of course, they take longer to execute.

The ***precondition*** describes which mental states must hold for the plan to be executed. The use of ***precondition*** makes the plans context-sensitive and thus limits the number of usable candidates among all candidates activated by the invocation conditions alone. It certainly speeds up the overall deliberative procedure. As for ***post-condition***, it describes the effects we want the plan to have. At the end of the execution of a deliberative plan, the post-conditions are verified. If they are satisfied the plan has succeeded, otherwise, it has failed.

The relation ***ct = credit – tc*** and the parameter ***ty*** have the same meanings, types and domains as those defined for patterns. However, we need to indicate that the value of ***tc***, which is related to the longest path among all possible paths for a plan may involve the executions of several different sub-plans. The parameter ***plan_type*** is used in cooperative situations. In such cases, if ***plan_type*** = variable, it means that the executing plan which can possibly receive from a negotiation procedure a request to "end early the current execution" will decide to reject or accept the request. Otherwise, when ***plan_type*** = invariable, such a request will be refused.

Since the edges of the graph are allowed to be desires, the ***(list[node]$_n$, list[edge]$_n$)*** can induce a **nested**-connected-rooted-directed-and/or graph. A desire corresponds to an "end" that the agent intends to attain, it is not an atomic action. Consequently, a desire cannot directly call the execution of an action function; it needs to be further interpreted to find the appropriate "means" to attain it[13]. The appropriate means will then be embedded into the body of the plan which, in this way, becomes a **nested** graph. Note that the nesting process

---

[13] This process is, in fact, the so called means-ends reasoning process.

will continue until all edges of the resulting solution path are executable atomic actions. Consequently, in IDEAL agents, a deliberative plan can be viewed as an event-invoked and context-sensitive recipe that allows hierarchical decomposition and execution of goals. The use of deliberative plans is a kind of "divide-and-conquer" procedure enabling the resolution of different types of complex problems.

All desires, whether coming from the agent itself or from the external environment, are considered equally important by the agent. Therefore, even though confronted with an internal desire, the agent has to check for other desires that could be present in the event queue so as to decide which must first be achieved. This gives the agent a dynamic planning ability, not present in classical planing processes, that enables it to react to environment changes in a real time fashion. This dynamicity automatically makes IDEAL agents commit their decisions as late as possible which, once committed, stay committed as long as possible. This reduces the necessity of re-planing and also decreases the occurrences of plan failures usually found in classical planing.

The function used to interpret and control the execution of plans is quite similar to the one used for the patterns. There are, however, some differences. First, contrary to a pattern execution where a path always proceeds along a single direction and involves only one graph, the execution of a plan involves a set of graphs and some procedures for forward searching and for backtracking. Second, a plan is not guaranteed to be continuously executed. It can be momentarily hanged when the scheduling module makes such a decision according to the contents of the event queue and of the intention structure, for example, when a new intention with higher priority appears in the intention structure. When the plan's priority becomes the highest again, it is restarted to be completed Therefore, the algorithm used to unwind the body of a plan must be able to save break point information after committing a given desire or action (***da***).

The use of different methods of search, such as breadth-first and depth-first algorithm, to expand a deliberative plan, will necessitate the use of different data structures [Rao 94]. For breadth-first methods, we could use partially instantiated graphs to represent a top-level plan and all its hierarchically related sub-plans [Weiss 99] and for depth-first methods, we could use a stack [Bratman 87] [Rao 97a]. Now, in situations where the number of ***and***-nodes is much greater than the number of ***or***-nodes, the breadth-first approach is very efficient, while when the opposite is true, the depth-first approach is often more effective. Since we believe that in the graph representation of the bodies of deliberative plans the number of ***or***-nodes is much greater than the number of ***and***-nodes, to unwind a plan we will use a modified depth-first algorithm that we call the ***next-step*** algorithm. It is based on the ***march-step*** algorithm and is used in the unwinding module of the deliberative layer. Given as input the body of a plan and some control information, the algorithm will determine the appropriate ***da*** (desire or atomic action) which will be committed next and, if needed, save the break point control information for later use. Due to the reason of space, details and comments of this algorithm are given in in the appendix.

When we compare deliberative plans with patterns, at first they may seem very similar but there are significant differences between them. These differences are the reason why they have different operational semantics. In deliberative plans the limits of the invocation conditions are extended by allowing not only premier triggering events as components of the invocation conditions, but also intermediate triggering events as well as quasi-triggering events to also be part of the invocation conditions. In addition, there are more types of commitments than there are for patterns. This gives the agent a capacity that can be used to generate some reasoning procedures to help solve complex problems. Plans can thus be used to deal with more complex problems than patterns can but, inevitably, it means an increase in execution time.

The use of preconditions in plans reduces the number of candidates that can be activated for some given invocation conditions, which in itself speeds up the

deliberative procedures and enables without limits the number of plans the agent can use.

It is easy to see that in order to guarantee good reactive speed, the number of available patterns must be limited. As a consequence, the number of events that can possibly trigger patterns is also limited with the implication that the range of problems that can be solve by control layers that use patterns is also limited. Such is not the case for the deliberative layer that uses plans.

Using a dynamic planer such as the one described above enriches the execution modes of intentions, improves real-time reaction ability and reduces the possibilities of re-planning and of plan failures. All this increases the robustness of the agent.

3.1.2.10 The experience-based plans

The experience-based plans is the way by which simulating the human-thinking procedure is integrated in the design. From the name, we already have some intuitive sense of what they are. Let us first say that experience-based plans are made of primitive operations and they are widely available to triggering events because they are plans. They are related to the agent's past behaviors and are called upon to solve problems similar to problems previously met by the agent and successfully solved. Such plans will be very efficient since a large part of it has previously been validated. In the IDEAL agent's architecture, experienced-based plans are used as alternatives to deliberative plans in order to provide context-sentive and reasoning-free fast optimal solutions to applications. In a more detailed way we define the experience-based plan as follow.

Definition 3.12 The experience-based plans

Let $exp(pt_1, pt_2, ..., pt_p)$ be an invocation condition represented as an expression of premier triggering events, $pt_1, pt_2, ..., pt_p$. Let $pre_1, pre_2, ...pre_m, po_1, po_2, ...po_j$, ***v, ct, cre, tc, ty***, and ***plan_type*** be parameters defined as in the definition 3.11. Also, let $n_1, n_2, ..., n_k$ be a sequence of nodes denoted as $n(k)$ and let ***p,***

m, j and ***k*** be natural integer. Each node of ***n(k)*** contains a pair (a_k, [{aa_k}]). Here a_k is an action[14] and {aa_k} is a list of actions. The number[15] of elements in the list can vary between 0 and 4. Then the expression

v((exp(pt_1, pt_2, ..., pt_p):exp(pre_1, pre_2, ...pre_m)):

([exp(po_1, po_2, ...po_j)]; ct; ty; plan_type) → n(k))

is called an ***experience-based plan***. The sequence of nodes **n(k)** is the body of the plan. Each node in ***n(k)*** is an ***or-node*** of the connected-rooted-directed and/or graph representation of the plan. Note that this corresponds to a special type example of a CRDAO graph. In the adjacency-lists representation mode, the body n(**k**) can be written as

(action, list[node]$_i$, list[edge]$_i$), for i = 1, 2, ..., k

with edges labeled action(**k**) representing atomic actions. As for **action**, it represents the edge the graph-control-pointer is pointing to.

Therefore, an experience-based plan can be written as

v((exp(pt_1, pt_2, ..., pt_p):exp(pre_1, pre_2, ...pre_m)): ([exp(po_1, po_2, ...po_j)];

ct; ty; plan_type) → (action, list[node]$_k$, list[edge]$_k$)).

In situations where the detailed description of an experience-based plan can be omitted, the plan is simply denoted as

[head → body] or,

[inv: pre: → (action, list[node]$_k$, list[edge]$_k$)],where **inv** stands for invocation condition, **pre** for the precondition. The figure 3.4 shows the body of an experienced-base plan, the bold lines outline the main path of the body.

Let ***ep*** be an experience-based plan and **EPL** be the set of all available experience-based plans, then **EPL** is called the experience-based plan library.

---

[14] See definition 3.3.

[15] Limited to a value less than 5 for performance reasons.

Since in the definition, we have set the ***invocation condition*** to be an expression of premier events, it means that the experience-based plan layer can deal strictly with PT events. Like patterns, experience-based plans are sensitive to external events. No reasoning procedures appear in the experience-based plan layer. Hence, the use of experience-based plans will provide the agent with faster speed of response and shorter execution time than deliberative plans. The

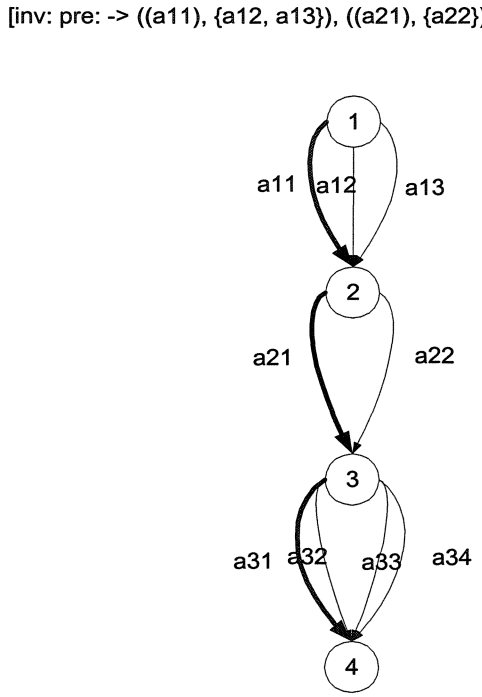


Figure 3.4 The Body Of An Experience-Based Plan

use of the ***precondition*** in the definition brings more extensive applicable range and faster deliberation procedure to the experience-based plans as compared to patterns. Like deliberative plans, the execution procedure of an experience-based plan may be changed in accord with the value of ***plan_type***. In other words, experience-based plans may be adapted to complete some cooperative tasks.

We use the term "experience-based plan" instead of "experience-based pattern", since we require that experience-based plans be run in the "plan execution" mode. Note that this implies that the agent can react in a real-time manner to changes in the environment since they must obey the principle of "committing their decisions as late as possible". Clearly, this type of commitment is satisfied by experience-based plans. Because of the body of an experience-based plan has the same format as that of a deliberative one, the ***next-step*** function can be used for experience-based plans.

The experience-based plans are results of the agent learning procedures. In the presented architecture, the creator, and manager, of the experience-based plans library (EPL) is the agent's learning module. Once a top-level deliberative plan triggered by an external event has been successfully completed, the learning unit analyzes the history database (**His**) in an attempt to identify a path from the starting-node to an end-node. If such a path exists, the learning unit will create an associated experience-based plan and add it to the experience-based plan library **EPL**.

Obviously, among all available plans for the same external event, the experience-based plan has the maximum success probability. Therefore, for routine events that the agent deals with, experience-based plans will replace deliberative plans to provide satisfying reasoning-free solutions. The agent's performance is enhanced. Note that when an IDEAL agent is started for the first time, its EPL is empty. However, this is not a problem, since some experience-based plans will be appended to the **EPL** as soon as a deliberative plan have succeeded.

Let us now explain how, without the support of reasoning mechanisms, the experience-based plan layer avoids plan failure or the necessity of re-planning. We know that each edge of the body of an experience-based plan represents an atomic action and that each node is always an or-node. This really means that, in

the ***next-step*** algorithm, each action in the main path of the plan has several possible alternatives solutions coming from the agent's past experiences of similar situations.

The precondition of an action is tested in the ***next-step*** algorithm. If an action fails to pass the test, the algorithm tests, in order of adjacency, each alternative until it eventually gets to an appropriate action. With this mechanism, re-planning is avoided and furthermore, plan failure is reduced. Here appears the reason why having our dynamic planning process always making its decision as late as possible[16], is a strong advantage. In the worst case, all alternatives fail the preconditions tests and the acting executor generates a desire of the type ***fail_e_processing(action, i_e)*** to initiate an exception handling pattern. The pattern seeks **DPL** for an alternative to the failed action. If one is found, a desire of the type ***(succ_desire$_{ep}$(action), i_e)*** is returned and the action that failed in the experience-based plan is replaced with the one returned by the ***(succ_desire$_{ep}$(action), i_e)*** desire. The controller saves this new action in a buffer until the plan is successfully completed or is aborted. If no alternative action is found, it returns the special event ***(fail_plan$_{ep}$(e, i_e))*** to start the exception handling pattern that deals with unforeseen events. This method guarantees robustness to the system.

From the above description, we see that as a mixture of deliberative plans and patterns, an experience-based plan inherits advantages from both. This provides IDEAL agents with fast solutions for routine problems.

3.1.2.11 The history database

The history database contains the information of all actions done by the agent. The history data is plan-related and will be used by the learning and modifying

---

[16] In many planing modes, a plan is a fixed sequence of action. It can not be changed as environment changing.

module to evaluate the plans and patterns. First, some important definitions are given.

Definition 3.13 The history data records and History database

We call path atoms the triplets $p\_a_j = (action_j, [mark_j], [alternative_j])$, for j = 1, 2, …. , and defined as follows. The first parameter identifies an action. The optional second parameter is either S or F meaning that the action was a success or a failure. When the action failed, the third parameter identifies the action that was successfully used as an alternative to the failed one.

Let ***v***, ***pp*** and ***ty*** be character strings, ***e*** be an event and ***e_mark*** be a Boolean value, then

$$v(e; pp; ty; [p\_a_1; p\_a_2; \ldots p\_a_j; \ldots p\_a_n;] e\_mark)$$

where symbol "***;***" is used as a separator,

is called a ***history data record*** in regards to event ***e*** and is denoted ***his_d(e)***. The name of the record is ***v*** and ***pp*** is the name of a plan or a pattern and the optional items p_a_i are path-atoms. When there is no ambiguity we use ***pp*** to represent the record itself. The record contains the information about all the actions executed by the agent to complete the plan or pattern ***pp***. The boolean variable ***e_mark*** indicates if ***pp*** has ended successfully.

We call the ***history database*** the set $His = \{his\_d(e_m)\}$, for m =1, 2, 3, …, where $his\_d(e_m)$ denotes the history data record associated with the m^{th} event that has occurred. Note that the history data records $his\_d(e_m)$ are not mental components; they can be viewed as *off-line processing* data components. Their contents never influence the operating semantics of an agent. In fact, history data records feedback the effects of the agent's behaviors to the learning module to update the weights of the plans or patterns and to eventually generate new experience-based plans.

Since history data records are events-related and events are plans-related, the use of the same name to represent both the history data record and the related

plan (pattern) will help to simplify storing operations of the history data. Event ***e*** is used as a parameter to cover three aspects. First, it identifies the event that triggered the plan or pattern. Second, it contains information to the parent record of a history data record; later, we will see that this related information is very important. Finally, it informs the agent on the type of storage records it must use. If event ***e*** is an external event, the agent uses permanent storage records. On the other hand, if the event is an internal event, the agent uses temporary storage records. These will be appended to their parent record and then committed to the learning module. The execution of a deliberative plan ***pp*** may involve several his_d records for the sake of including reasoning procedures. For an experience-based plan only one his_d record is involved.

When the unwinding function creates an intention, the triggering event is a premier triggering event and a permanent his_d record is created in order to describe the execution of a top-level plan. If unwinding function adds an intention frame it corresponds to an intermediate triggering event IT. In this case, only a temporary his_d record is created, describing the execution of a sub-plan. When a NT event of the type "action success" (s_f_sign = 1) is generated, then the acting executor will add the action into a his_d record. The record's has the same name as that of the intention frame on the top of intention stack *i* included in the NT event. If a NT event of the type "desire success" (s_f_sign = 3) is generated, then the acting executor will copy all actions in the associated temporary his_d record to its parents record and then send the temporary record to the learning module. The temporary record's name is the same as that of the intention frame on the top of intention stack *i* included in the NT event. In the case of a "desire fail" event, the related temporary his_d record is only sent to the learning module while for a NT event: "plan success" , the permanent his_d record is committed to the learning module. In the historical record, the name *v* is the name of the plan triggered by event *e*, each path-atoms is an action that has successfully been completed during the execution of the plan, and all the path-atoms form a path in the plan's body. Using *v*, the learning module can get the head and description

parts of the plan *pp* from DPL. This information is used as the head and description of an experience-based plan, all path-atoms forming body of the experience-based plan.

The parameter *e_mark* has two uses:

1. it is used as the ending sign of a his_d record,
2. it denotes the "implementation status": success or failure, of plan or pattern *pp*.

Therefore, it is used as the basis of updating the credit of the plan or pattern *pp*. The optional item p_a_j will not appear in all his_d records related to patterns. The reason is that a pattern's body is viewed as an entirety by the learning module. The learning module does not evaluate each action in the pattern's body. Consequently, if *pp* is a pattern name, the associated his_d record only includes five items *v(e; pp; ty; e_mark)*. If it is a deliberative plan name, each path-atom only includes one action atom and the action must have been successfull.

Optional items *mark* and *alternative* are designed for experience-based plans. When an action fails in the execution of an experience-based plan, *mark* is assigned value F in the related path-atom triplet of the his_d record. Otherwise, it gets a value S. The success alternative to a failed action is assigned to *alternative*. The learning module will evaluate each action in the his_d record and the priorities of those marked with *F* will be changed in the related plan. It means that the position of the action atom in the node will be changed.

As the end of this section, we need to point that goals, options and intentions describe the dynamic characteristics of the agent while beliefs and plans consist of rule of execution.

3.1.3 Functionalities of the operational functions of IDEAL agent

The basic operation functions part of the IDEAL agent architecture are shown in figure 3.5. Below we will describe the functionalities of these operational

functions. More details of these functions, such as the definitions and basic operating semantics, are given in Appendix C.

In addition, when these basic operation functions are mapped into the control architecuture of an IDEAL agent, as will be shown in the section 3.2, the operational semantics of the projected components may be enriched in different degrees. Mostly this happens because the implemented data structures of the mental states are different for the different layers. However, given the the limits in length of this thesis, the description of most of these enrichments will be omitted.

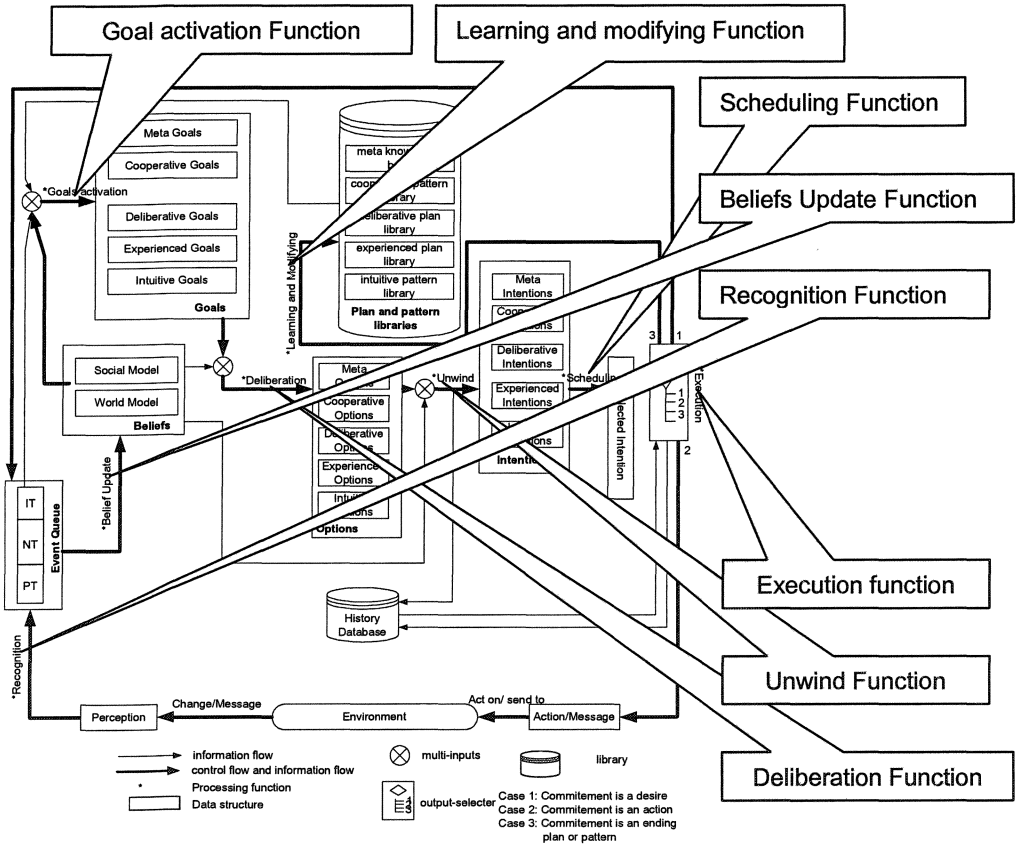


Figure 3.5 Operational functions

- Recognition function() describes how perceptions are transformed into standard triggering events.
- Belief update function() updates current belief database and also revises all beliefs so as to keep their consistency.
- Goal activation()describes how to generate applicable plans and patterns for given events.
- Deliberation function() describes how optimal solutions for a given event, options, are extracted from the candidate goals.
- Unwind function() maps the agent's options into the intentions.
- Schedule function() selects the intention with the highest priority to commit.
- Execution function() commits the selected intention to execution.
- Learning and modifying() function involves two aspects. One side, it generates experience based plans by using the rote learning mechanism. So that it bridges the gap between modern dynamic planning methods and classic planning methods. It provides intuitive patterns' respond speed, but also effects generated from dynamic planning. On another side, it modifies the credits of the related plans in a way that should improve the future performance of the agent by using feedback-based learning mechanism.

3.1.4 The IDEAL agent

After defining all mental states and describing all basic operation functions, we finally can define that IDEAL agent itself.

Definition 3.14 IDEAL agent

Let P, B, G, O, I, E respectively be perceptions, beliefs, goals, options, intentions, and events and let PPL, AFL, HD respectively be the plan and pattern libraries, the acting function library and the history database. Let also

SENSING = (belief_update(e_filter(reco()))),

LAYER = (unwind(deli(goal_activation()))) and,

COMMITTING = (exec(sch(I_filter())))

be compositive functions and let LEARN be a set of learning and modifying functions, **v** be a character string, then, at any time, the tuple

v(P, B, G, O, I, E, PPL, AFL, HD, SENSING, LAYER, COMMITTING, LEARN)

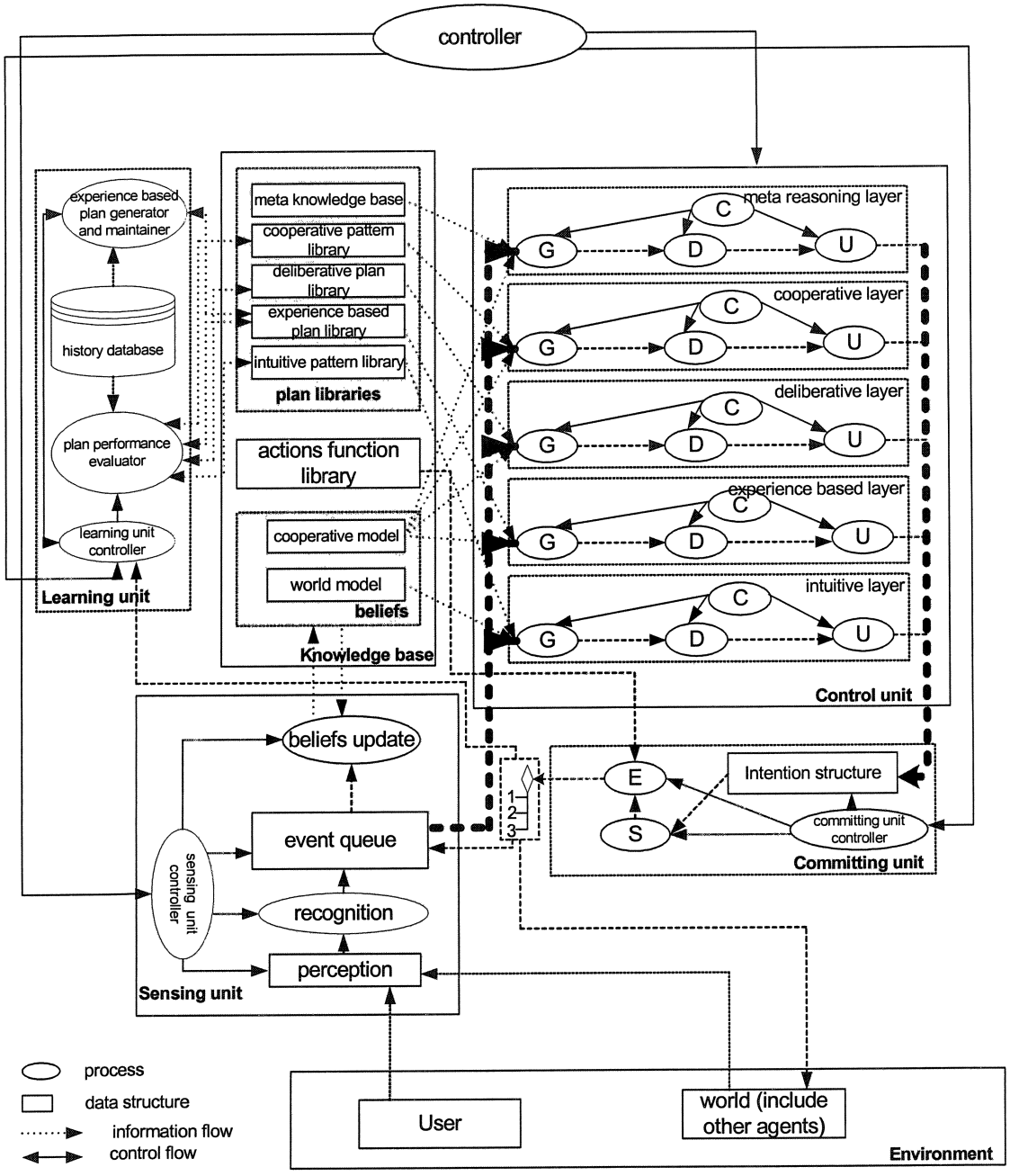
is called an **IDEAL agent**.

v is name of the agent and is used as its identifier. SENSING updates the event queue and the beliefs, LAYER generates one single intention for each triggering event and COMMITTING executes, in correct temporal order, an action or a desire from the intention with highest priority presnt in the intention structure. LEARN updates PPL according to HD. The identifiers belief_update() and reco() represent the belief generation and revison function and the recognition function. The terms deli(), exec() and sch() respectively denote the deliberation function, the execution() function and the schedule() function.

3.2 The control architecture of an IDEAL agent

By transferring the IDEAL agent's conceptual model into an implementable system we map the conceptual model into a practical layered framework which really is the IDEAL agent's control architecture. The control architecture outlines the functionality and the decision making process of the agent. It also describes every components of all units of the agent, particularly their functionality and structure as well as the temporal relations between them. In addition, it depicts the roles and structural aspects of each individual control layers. Finally, the data flows and the control flows of each layer are extensively discussed.

The control architecture is show in figure 3.6.



C: layer controller
D: deliberation module
E: execution module
G: goal activation module
S: scheduling module
U: unfold module

Figure 3.6 IDEAL agent control architecture

The IDEAL agent's control architecture is implemented as a horizontal layered architecture constituted of six different units:

- a ***control unit*** containing five layers organized in a vertical hierarchy;
- a ***hierarchical knowledge base*** shared by the control and the committing units and updated by the learning unit and the belief update module;
- a ***learning unit*** consisting of two processing components and a history database. One component used as knowledge update, another one used as knowledge maintaining;
- a ***sensing unit*** consisting of two processing modules and two data (structure) modules. The former used as input processing and the lattter used as input buffer.
- a ***committing unit*** generating and outputing the agent's behaviours

a ***agent controller*** supervising the operations in all components, and coordinating and synchronizing the relations between all units.

3.2.1 The control unit

In the ***control unit*** we find the five control layers respectively called the deliberative (DL), experience-based (EL), intuitive (IL), cooperative (CL) and meta reasoning (ML) layer. They are implemented as five concurrent independent processes[17]. This means that every time an event will occur, it will be concurrently sent to each control layer where, if the invocation conditions are satisfied by any plan or pattern the layer has access to, the event will be accepted and the treatment started. For layers where no plan or pattern can be found, the event will be dropped by the layer. There is no direct information transmission between the layers; the only way to exchange information is via the use of "commit" messages sent to the event queue. Each control layer possess

---

[17] These can be programmed as concurrent procedures or threads.

its own set of basic functions and is designed to deal specifically with only one type of problems.

There are no common parts between each type of problems with the exception of the deliberative and the experience-based types of problems, as explained in the preceding section. The experience-based layer provides a fast and flexible alternative solution to problems of the deliberative type that are often met by the agent. A pattern-like mechanism replaces the means-end reasoning mechanism normally used by the deliberative layer. The deliberative layer complex reasoning processes are transformed into a sequence of simple matching or tables lookup processes. As a result, this mechanism reduces the costs related to the reasoning processes and improves the reaction time.

The intuitive layer control uses reactive patterns to deal with unforeseen events or events that require fast response times. It must be mentioned that the intuitive layer control is an important guarantee to the robustness of the IDEAL agent. The so-called "unforeseen events" are events that the agent has not been designed to solve. When such events occur, all layers except the intuitive layer fail to deal with them. Whatever the current beliefs are, to process the event the agent starts, from the intuitive layer, a "failure_p_process" program defined as a situation-oriented non-interruptible reactive pattern. In this way, the robustness of the system is guaranteed. This characteristic is often called "*weak completeness*" [Wooldridge 92]. As for the cooperative layer control, it deals with "negotiate" events resulting from the execution of a multiagent cooperative plan. It works in a reactive pattern manner so as to guarantee that the system remains conflict-free. Finally, the meta-reasoning control layer provides the needed knowledge for deliberative operations required in the deliberative layer. It helps the deliberative layer to provide optimal solutions to given problems.

From figure 3.6, we notice that each control layer has the same structure though we know that the tasks to be realised are quite different. In fact, each structure is composed of similar modules, namely, a goal activation(G), a deliberation(D) and an unwind(U) module. The fundamental reason behind this "pseudo generic

structure" is that the functional modules G, D and U are projections of the corresponding operational functions of the conceptual model into each layer's task domain. The use of such a structure facilitates the description of problems since each functional layer can be viewed as an instance of the control structure to be applied to specific instances of problems.

For each layer there is also a ***process controller*** (C) that generates and manages the local control flow. The local control flow goes from goal activation G to options selection D, to plan interpreting in U. Control is then passed to the *commiting unit* where, from the intention structure, it goes through intentions scheduling S and intention execution E, then is sent to the event queue and finally back to goal activation G. This path of control corresponds to the operational process described in the conceptual model. Note that each control layer strictly has access to its corresponding portion of the ***knowledge base***.

3.2.2 The knowledge base

The knowledge base has three different components. One part contains information defining the beliefs the agent has about the world and its social properties. The second part is the layered plans and patterns libraries containing the knowledge related to the different mode of treatment the agent can use to play its role. As for the third part, it corresponds to the ***actions function library***. As mentioned in section 3.1, the first objective of partitioning the knowledge base is to reduce the amount of information each layer needs to search through thus reducing the system reacting time. The second objective is to facilitate the storage of the different types of knowledge and allow the building of the knowledge base with non-homogeneous characteristics so as to make the IDEAL agent's behaviors more flexible.

The ***Beliefs*** database is divided in two different parts: the world model and the cooperative model. The access right to each part of the ***Beliefs*** database goes from top to bottom. By this we mean that if a layer is given access to the cooperative model, it also has access to the world model. However, a layer that

is given access to the world model cannot access the cooperative model. For example, the intuitive layer only has the right to access the world model; it thus cannot access the cooperative model. On the other hand, the deliberative layer is given access to the cooperative model. It can thus also access the world model.

Currently, the ***plans and patterns libraries*** of an IDEAL agent are organized as a set of object databases corresponding to the different layers. Plans and patterns are stored into their corresponding databases. Every plan or pattern is partitioned into three parts: the head, the description[18] and the body. The head contains the invocation conditions plus, for plans, the preconditions. In each database type the heads of plans are organized into an index. The bodies are structured as connected-rooted-directed ***and/or*** graphs where nodes represent states the agent can attain and edges represent the operations (either desires or basic actions) for going from one state to another. Each graph is defined by a node list and an edge list [Thomas 90].

Accessing plans and patterns libraries corresponds to search requests on databases and can thus be implemented by member functions defined on the object databases. The ***actions functions library*** can be defined as a super class of all processing units where each function uses a plan name and an action name as parameters. In this way, when the action executor (the actor) receives an action name committed by the execution module, it can directly call from the ***actions functions library*** the related function to fulfill the action.

3.2.3 The learning unit

The ***learning unit*** is totally independent from the control layers processes. It is constituted of the four following component: the ***experience-based plan generator and maintainer***, the ***plans performance evaluator***, the ***process controller*** and the ***history database***.

---

[18] The next chapter gives detailed definitions of patterns and plans

The *process controller* supervises the learning cycle. When it receives a message, it starts the *plan performance evaluator* in order to update the plan or pattern performance value. According to the information stored in the *history database*, the evaluator either increases or reduces the weight of the plan and calls a database update operation to update the corresponding plan or pattern library. If the updated plan is a deliberative plan, then the *experience-based plan generator and maintainer* component is started. It first checks the information stored in the history database to know whether the plan has been successfully fulfilled. If so, the module connects together all the actions taken by the successful plan. The resulting sequence becomes an experience-based plan. In fact, this plan is a path from the root node to one of the leaf nodes of the embedded connected-rooted-directed ***and/or*** graph (intention) of the successful deliberative plan. By using database operations functions, the experience-based plan library is then updated consequently. The learning process ends and the process controller, using database operation functions, removes the related information from the history database and continues with the next learning cycle.

3.2.4 The sensing unit

Although terms such as “sensor”, “executor” or “world interface” are not explicitly defined in the IDEAL control architecture, the roles implied by these terms are supported. Unlike Interrap [Muller 97] or PRS [Georgeff 89] agent models, the IDEAL agent works in a pure “soft” environment, meaning that its inputs and outputs are not physical signals that need to be interpreted. Therefore, there is no need for “sensors” and “executors” in our model. The IDEAL agent is designed with the hypothesis that its inputs will be messages from other agents or users and that these messages will satisfy the format our agent uses.

According to the speech act theory [Sea 69], receiving messages can be viewed as a sensing process modeling the agent’s perception data structure (buffer) as a sensor. Similarly, sending messages can be viewed as specific type of actions. Safe synchronous and asynchronous communication mechanisms for sending

and receiving messages can be directly implemented by operation functions defined in the ***perception*** and the ***action function library***.

The *message honesty* rule can easily be satisfied by these operation functions since it essentially means: *for a given message, the identifiers of the sender and receiver cannot be equal*. In our agent, the sensing unit controller uses the recognition function and belief update function to simulate the role of "sensoric subsystem" as can be found in Müller's Interrap agent [Muller 97]. The sensing unit controller monitors the perception buffer. When a change is noticed, it uses the recognition function to generate an event of the appropriate type. Then it uses the belief updating function to modify the agent's current beliefs.

3.2.5 The committing unit

In IDEAL agents, the commitment to be executed includes sending messages and performing specific actions. Both sending messages and performing actions are realized by the action executor which call the appropriate actions functions from the actions functions library. We distinguish three different modes for action execution: non-interruptible execution, interruptible execution and plan execution.

The non-interruptible execution mode is used with intuitive patterns activated only when some unforeseen situation or failure of some going on process that needs to be dealt with. It can also be used with meta reasoning patterns. In this mode, the preconditions of the actions defined in the body of the pattern are never checked. The pattern is just executed without interruption. Note that the same pattern could be used in an interruptible mode. Only when in the non-interruptible mode is there no control. The body of a pattern is a list of action names and the action executor uses these names to call the corresponding actions functions from the actions functions library.

The interruptible execution mode is the normal mode used by all patterns. The body of some of these patterns may represent complex operations. While executing the pattern, at a given step, the environment can change. After executing a step a check must be made on the environment state to decide

whether the execution continues or stops because it may happen that the precondition of the next action of the pattern is not satisfied anymore. Hence, the acting executor uses the march-step algorithm and chooses the appropriate action for the next step. If no possible action can be found for the next step a failure event is generated and sent to the event queue and the execution of the pattern ends. When a pattern is running in the interruptible mode, the layer control does not deal with changes in the event queue. However, the agent's beliefs are always updated since the sensing unit is running in parallel.

The plan execution mode works in a way similar to the interruptible execution mode. The difference is that the plan execution mode deals with new occurrences of event while the interruptible execution mode never does.

We want the IDEAL agents to have a certain dynamic capacity of planning. In fact, we would like the agents to make their decisions as late as possible to take into account the latest changes in the environment. Obviously, agents working in complex dynamic environments should be provided with such ability. Particularly, the agents should have the ability to deal with events that have occurred during the execution process of an intention and which have been given higher priorities than that of the event presently being processed. The plan executing processes defined in our agent satisfy this requirement.

Before the selection of a plan by the scheduling module, the unwind module uses the next-step function, very similar to the march-step function, to decide which action is to be executed next. If the commitment sent to the execution module is a desire, then the execution module sends it to the event queue as a new triggering event. If the commitment is only an action then the module, through the use of the acting executor, calls the related function from the actions function library and sends an NT event to the event queue. The agent then deals with the event queue. If there are new triggering events with higher priorities, then the agent must first process these before completing the currently executing plan. The current plan is thus suspended and its information stored in memory; it will be finished at a later time. Otherwise, the schedule module enables the

execution of the current plan to continue. It is the schedule module that commands the above process.

3.2.6 The control flows in the control architecture

The IDEAL agent is built as an horizontal layered architecture. The activation of the agent is triggered from the perception buffer and the control is then shifted to the control unit with its five functional concurrent layers. As previously said, each layer is built with a given specific competence and all layers simultaneously receive the same information. Since each layer is totally unaware of any of the other, all it can do is decide whether it has the competence or not to deal with the specific problems it received. Consequently, five competence-based clockwise control flows are formed and, normally, at least one of the control flows is contributive by tempting to find the solution for the given problem. However, we have to provide for situations where events cannot be dealt with by any of the layers. Two such situations are distinguished: “non-implementable desire” and "unforeseen events".

The case of a non-implementable desire corresponds to situations foreseen by the agent designer but to which the agent is not to respond. These events are simply dropped from the agent since no layer has the means to provide some solution. The case of “unforeseen” situations is self explicit. It corresponds to situations which should never happen. In such cases, the agent has to proceed with a *fail_processing pattern* otherwise the system would crash with unpredicted consequences such as, for example, lost of precious information.

From this analysis, we see that, normally, an IDEAL agent has five control paths and five corresponding control flows to deal with very different types of tasks. Each path passes through four stages: ***perception, decision-making stage, action execution*** and ***learning***.

Five additional control paths result from the interplay among the functional layers. Three of those describe the additional control flows dealing with “unforeseen events”. Starting either from the deliberative, cooperative or experience-based

layer, for each path, the flow goes through the ***perception, decision-making*** and ***action execution*** stages then goes on to the intuitive layer to process the *fail_processing* patterns. At this layer the flow goes through the ***decision-making,*** the ***action execution*** and the ***learning*** stages. These three paths express the indirect interplay existing between the deliberative, experience-based and cooperative layers, and the intuitive layer.

The fourth additional control path represents the control flow for situation such that for a given node[19] of an executing experience-based plan, the search for an action fails and a "seek alternative" event is generated. Starting in the experience-based plan layer, the control flow goes trough the ***perception***, ***decision-making*** and ***action execution*** stages, passes to the intuitive layer where it goes trough the ***decision-making*** and ***action execution*** stages then returns to the experience-based plan layer where it goes through the ***decision-making, action execution*** and ***learning*** stages.

The last additional control path describes the indirect interplay that takes place between the deliberative and the meta reasoning layers when the agent is dealing with “complex decision making” problems. Starting in the deliberative layer, it goes trough the ***perception***, ***decision-making*** and ***action execution*** stages, from there accesses the meta reasoning layer where it goes through the ***decision-making*** and ***action execution*** stages, then returns to the deliberative layer where it goes through the ***decision-making, action execution*** and the ***learning*** stages.

The manager of the control flows, the ***agent controller***, monitors, coordinates, synchronizes and controls all eight independent processes of the agent:

- five defined by the independent layers of the control unit,
- one defined by the learning unit,
- one defined by the sensing unit, and

---

[19] See Figure 3.5

- one defined by the committing unit.

It also schedules the temporal relations among all eight processes. It is the agent controller working procedures that decides of the agent's behaviors. At time zero, the agent controller starts all processes. When the sensing unit controller observes a change in its buffer, it makes a copy of the buffer and then the buffer is emptied. Next, the process controller uses the recognition module to eliminate redundant information and identify the perception. Then it checks if there is a *beliefs access lock* established by the agent's other components[20]. If not, it calls the beliefs update module. Otherwise it waits until the access is unlocked and then updates the beliefs. The sensing unit controller then updates the event queue. If the event queue is empty, the agent controller will check if the intention structure is also empty. If so, the sensing unit keeps cycling until something new is sensed or until the agent is turned off. If the intention structure is not empty the agent controller executes the next scheduled intention of the structure.

When the agent controller finds that the event queue is not empty, it sends a copy of it to each of the five layer controllers and then empties it. Then the flow of control is given to the control unit. Each functional layer independently deals with the events it received, generates the appropriate intentions and puts them into the intention structure. When all layers have finished their processing tasks they enter an action waiting mode and the control is shifted to the committing unit controller. The controller uses the scheduling module to sort all intentions in the intention structure, selects the intention with the highest priority and sends it to the execution module. The execution module executes the action or desire on top of the intention (stack) and simultaneously sends a signal to the other layer controllers to change from action waiting mode to event waiting mode. From the execution of the action, a new event is generated and sent to the event queue. The still active layer controller then goes into the event waiting mode. Once the

---

[20] Any component accessing the beliefs prohibits the sensing unit updating the beliefs.

agent controller receives an ***action-finished*** signal, it will shift the control to deal again with the event queue and a new cycle is started.

Chapter 4: The multi-agent system with hierarchical broadcast architecture constructed by IDEAL agents

Integration of the IDEAL architecture into multi-agent systems

In Chapter 1, we mentioned that real world application problems are often very complex. The use of single-agent system is often insufficient to provide satisfying solutions; multi-agent systems are frequently used in such cases. A multi-agent system (MAS) is a system having several individual agents, called the basic agents, normally working, in some collaborative manner, under the control of an overall supervising agent called the managing agent. The choice of the basic agents' architectures and the establishment of the coordination and supervision characteristics of the managing agent are key elements in the performance of the resulting system. These choices are usually based on the nature and intrinsic complexity of the problems to solve.

Many MAS applications are modeled with the basic agents having the same architecture so as to attain better performance than the same applications solved with basic agents of different architectures. A few examples of such applications are network management, stock market trading, traffic management, intelligent tutoring and robot teams. For most of these, the managing agent has an architecture that differs from the basic agents' one. However, the IDEAL agent architecture can be used both for the managing and basic agents. The only difference between agents of different levels will reside in the type of their knowledge bases. Given this, we believe that the overall coordination of the resulting MAS would be best achieved by the use of a hierarchical broadcast architecture.

4.1 A multi-agent system with a hierarchical broadcast frame (MASHBF)

Figure 4.1 shows the architecture of a multi-agents system with a hierarchical broadcast frame formed by a *Manager agent*, a number of *Sub-manager agents* and *Basic agents*. This type of architecture is called a *star connection*.

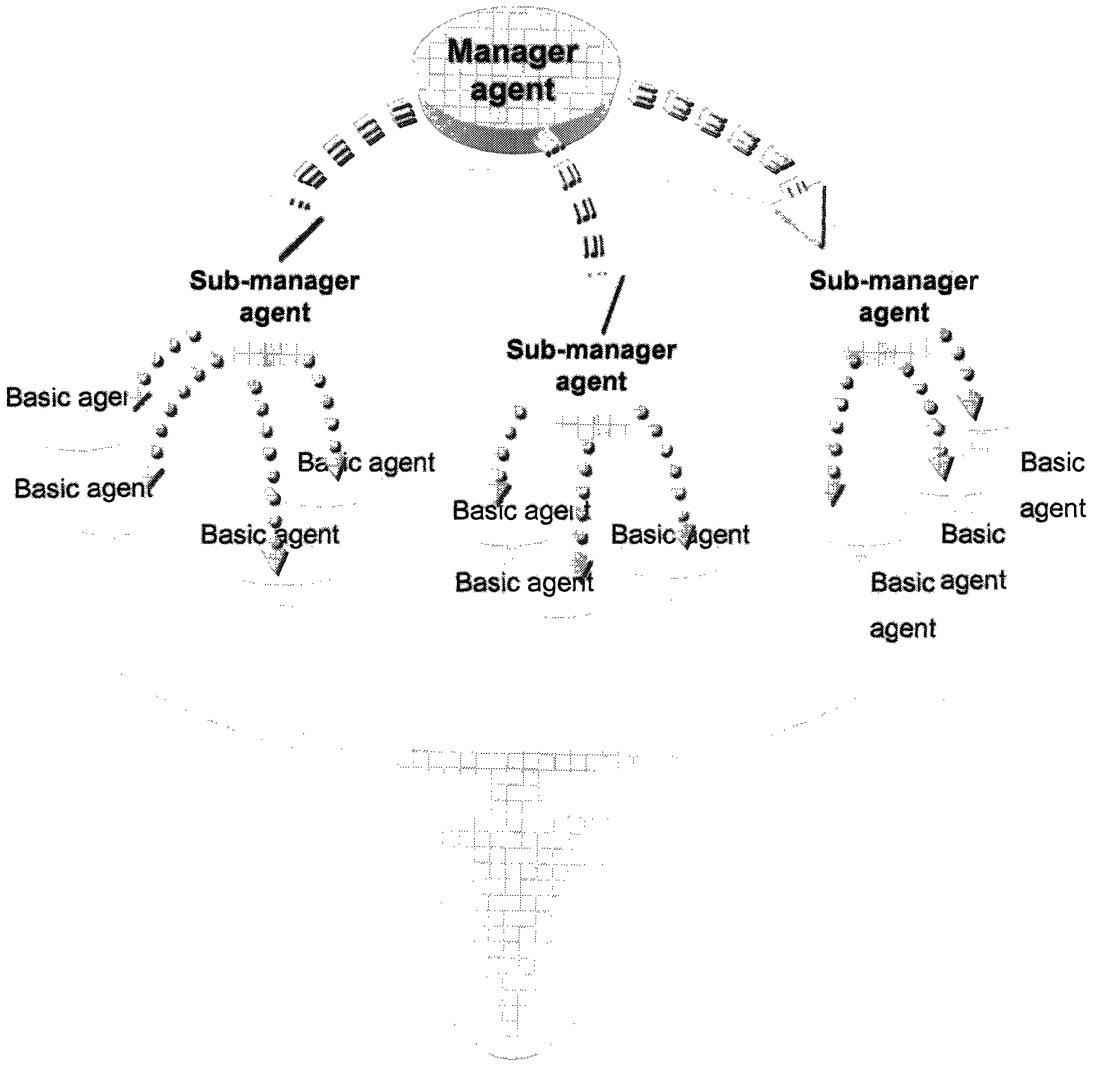


Figure 4.1 A multi-agent system with a hierarchical broadcast frame

The reason behind calling it a hierarchical broadcast frame comes from the way it works. In the general sense, a broadcast is associated with the fact that any request for information is sent to all the agents. The request is answered to by any agent which has the appropriate information. In a hierarchical broadcast the request is only sent to the next lower level of agents and only agents from this level can answer the request. It means that agents separated by at least one intermediate level of agents cannot directly communicate.

In a multi-agent system, a large and complex task is decomposed into different sub-tasks that are sent to the next lower layer. The process of decomposing and sending to the lower layer is repeated until a lowest layer is reached where the complexity and size of the subtasks are such that they can be performed by *basic* agents. Together, the basic agents with resource-limited capacities will cooperatively solve the complex task under the *Manager* agent coordination and management. The number of layers of the system is decided according to the complexity of the problem, the resources of individual basic agents and the lower bound of the needed response speed. The manager agent capacities include all aspects related to collaboration[Bernard 2000] and competition between agents [Esmahi 2000] of lower levels.

The manager and sub-manager agents have to collect and exchange information with agents of their next lower level, make and schedule global cooperative plans, and map these into sub-tasks that will become local plans to the individual next lower level agents. Local conflict situations as well as redundancy and time ordering problems are dealt with by concertation between agents of the appropriate levels and their immediate supervisor. Global conflict situations are always dealt with by the *Manager* agent. Given the cooperative layers present in each agent, all the necessary exchanges of information regarding the mental states is done through communication using simple “propose" and "report” modes. The responses of negotiation requests sent by the manager agent are defined, at the basic agents levels, as cooperative patterns. Furthermore, since

the IDEAL agent architecture is built to react in a dynamic way to changes in the environment, plans at all levels are automatically modified in reaction to these changes. For conflictual situations that could appear between agents at the basic level, we need only include simple negotiation protocols[21] primitives in their cooperative patterns library to give them the ability to reach an agreement. At the manager level, the executing plan is modified in accord with the reached agreement and the modified global plan can continue to be executed.

Let us explain, in a more detailed way, the working principle underneath the present system[22]. There are two further unconditional considerations in our MAS structure:

- only the manager agent can communicate with the user,
- only the basic agents can sense the real world.

When the MAS is first activated, the mental state of every agents of the MAS needs to be established. In order to do so, the manager sends to all its sub-manager agents a request for information on their mental states. Since their own mental states must also be established, in reaction to this initial request, all sub-managers send to all of their basic agents a similar request. As for the basic agents, they build their own initial mental states by sensing the real world. The corresponding information is reported back to the agents of the next higher layer in answer to their request. Their mental states thus established, the sub-managers send to the manager the requested information from which the initial mental state of the manager is finally established. As a result of this initial process, all agents of the system have updated their respective knowledge bases in accord with the present state of the world. All the agents are in their waiting modes.

---

[21] These include primitives such as “propose", "accept", "reject" and "report" primitives.

[22] In some applications, each sub-manager agent or basic agent have fixed roles, as is the case in real organizational structures.

When the manager agent receives from the user a request for a task, it is analyzed within the perspective of the information available from the manager's knowledge base and a cooperative global plan is generated to achieve the given task. The plan is then mapped into a set of redundant-free and conflict-free subtasks which are broadcast to the agents at the next lower level. Each sub-manager agent extracts from this set one of the subtasks corresponding to its abilities and sends to the manager a message identifying the accepted subtask. Note that at any level it may happen, for some agents, that no subtask corresponds to their abilities. These agents just stay in their waiting mode. With their accepted subtasks, the sub-manager agents proceed in a similar way and broadcast a set of sub-subtasks to the basic agents. From their accepted tasks, the basic agents establish their local goals and then, local plans are generated and executed. Since the broadcast tasks are conflict-free, there is no needs for negotiation at this time and the cost of communication is minimized. The basic agents, working in a cooperative and coordinated way under the supervision of their sub-manager agent, complete their given tasks and return the results to their supervisor. The sub-manager agents proceed in the same way at their own level. The overall results are synthesized in the manager agent. The task requested by the user is completed.

For application fields where changes in the environment can't be predicted, occurrences of uncontrolled events are possible. For such cases, it becomes necessary to modify some of sub-manager agents' plans and even the manager's global plan. Trying to modify already executing plans may provoke conflicts at the levels of agents pursuing specific goals. Consequently, the sub-manager or manager agents must start some negotiation processes in an attempt to find a solution to the generated conflicts. The first negotiation processes always take place between sub-manager agents and basic agents since it is at the basic agents level that changes in the environment are first noticed and conflictual situations identified. The sub-manager agents propose

modified requirements to the basic agents[23] in response to their report of conflict occurrences. The basic agents, acting as self-interest entities, decide to either accept or reject the new requirements from their sub-manager agent. Using the answers of its basic agents, a sub-manager agent either confirms that an agreement has been reached and the negotiation process is ended or judges if another negotiation cycle is needed. When and only when an agreement has been reached, do the sub-manager agent modify its local plans and broadcasts new sub-goals to its basic agents. If the negotiation process fails and no agreement is reached, then the conflicts are committed to the manager agent where a similar process is repeated.

In fact, the inspiration about our "hierarchical broadcast frame" comes from e-business applications where, for many situations, the real world itself is organized with hierarchical characteristics. At the lower levels, solutions to problems are usually clearly defined, the dependence relations between the hierarchical levels easy to understand and the top-down mechanisms of decomposition very well known. Therefore, multi-agent systems with hierarchical broadcast frames are especially well suited for -but not limited to- these applications. A typical example of such an application is the trading market. In the following section, we present in a more detailed manner the futures options trading operation process.

4.2 A futures options trading system

A futures options trading system is a trading systems where the objects traded are options on futures contracts. A future contract consists in buying, at today's price, a large[24] quantity of a given commodity such as sugar, coffee, ... However, since the quantities involved are so large, when the contract is made you need

---

[23] All agents use their cooperative layers to negotiate.

only pay a small percentage of its total value which, depending on the commodity, varies from 2% to 5%. The contract also specifies a closing date, from 3 to 6 or 9 months later, and at which time you have to pay the difference. Now, if the price of the commodity increases, the value of your contract increases and it is easy to find somebody that is willing to buy it from you. You are making money. However, if the price decreases you are left with your contract and you are loosing money. It is a very risky business even for the people who really need the commodity. Now, let us introduce the notion of option.

An option can be seen as a right on a specific item. When you buy an option, it means you buy a certain right on this specific item. In the trade market world, options are related to specific items such as stocks, index, futures.... Basically, there are two types of options: the ***call*** option which gives you the right to ***buy*** at a given fixed price and the ***put*** option which gives you the right to ***sell*** at a fixed given price. Hence, when you trade in futures options, it means that you buy and/or sell rights on futures contracts.

In a futures options trading group there usually is a trader, some trader assistants and some analyzers. Figure 4.2 gives a schematic view of its MAS representation. The analyzer periodically collects information on the price of options in future markets, analyzes the information and summits suggestions to the trader assistants. These suggestions specifiy which contracts should be bought and which contracts should be sold, the number of contracts to trade on, the types of options, the risk parameters, the profit estimation, ... The trader assistant synthesizes into reports the information from the different analyzers (each analyzer is using a different economic model) and summits an optimal plan about his category of future options. For example, the yen option, pound option, mark option and $Can option all belong to the category of currency options. Other categories of futures options could be live cattle, precious metals, ...

---

[24] Usually, the quantity is measured in terms of 100 000 units or even million of units. For example, it could be 1million lbs of coffee.

Based on the reports received from the trader assistants of each category of future options, the trader establishes a global plan to buy and sell futures options for many of the categories, according to the expected profits, the amount of available money, the time periods of the options, the associated risks,.... The plan is distributed to the trader assistants which call their specific analyzers to execute their part. Each analyzer complete a series of trading operations such as buying, selling, market monitoring, ..., and returning profits or losses to the trader.

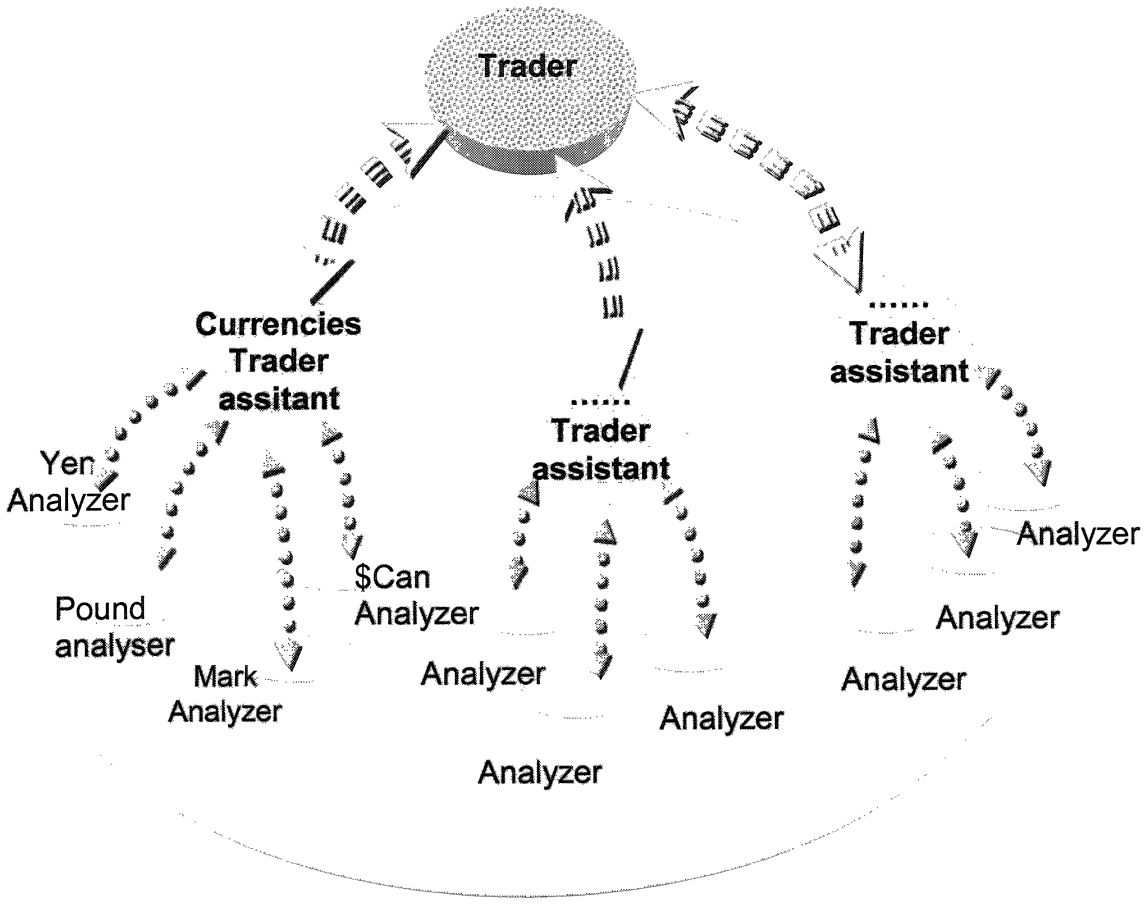


Figure 4.2 A futures options trading system

The trader has a limited amount of money to invest and his global goal is to optimize the return on its investments. For simplicity, let us consider options on currencies futures and only for two different currencies: the yen and the mark. Consider that for a given current knowledge, the system is started with an investment task assigned to the yen option analyzer agent (at the basic agent level) and with the mark option analyzer agent (also at the basic agent level) activated in a market-monitoring mode. At a later time, the system shows that the yen option trading plan is getting close to its expiration date and only a small profit can possibly come out of it. At the same time, the mark option trading plan recognizes a large profit opportunity but finds itself in a conflict situation since he has no money to buy options contracts.

The conflict is reported to the currencies trader assistant agent which, after consideration, decides to end early the executing yen option trading plan and invest the recuperated money into a mark option trading plan. The trader assistant agent thus negotiates with the yen option analyzer agent to end early its current plan. If the latter accepts, it will sell the contracts it holds, return the money and start a new market monitoring plan. On the another hand, the trader assistant agent accepts the proposal of the mark option analyzer agent which will use the now available money to buy contracts options on the mark and hopefully make more money.

4.3 On the use of the IDEAL architecture in MAS

Obviously, due to the natural correspondence existing between our “hierarchical broadcast frame” and actual problems, applying our multi-agent hierarchical broadcast approach to solve complex e-commerce problems will be much more simple than using less corresponding architectures. It will certainly facilitate the design of applications and surely diminish the costs. We do not believe, of course, that the same problems cannot be resolved by single-agents or other multi-agents system with different architectures, but we believe that their problem-solving processes would be a lot more complex to establish and the

associated costs would also be much higher. Let us compare our MAS

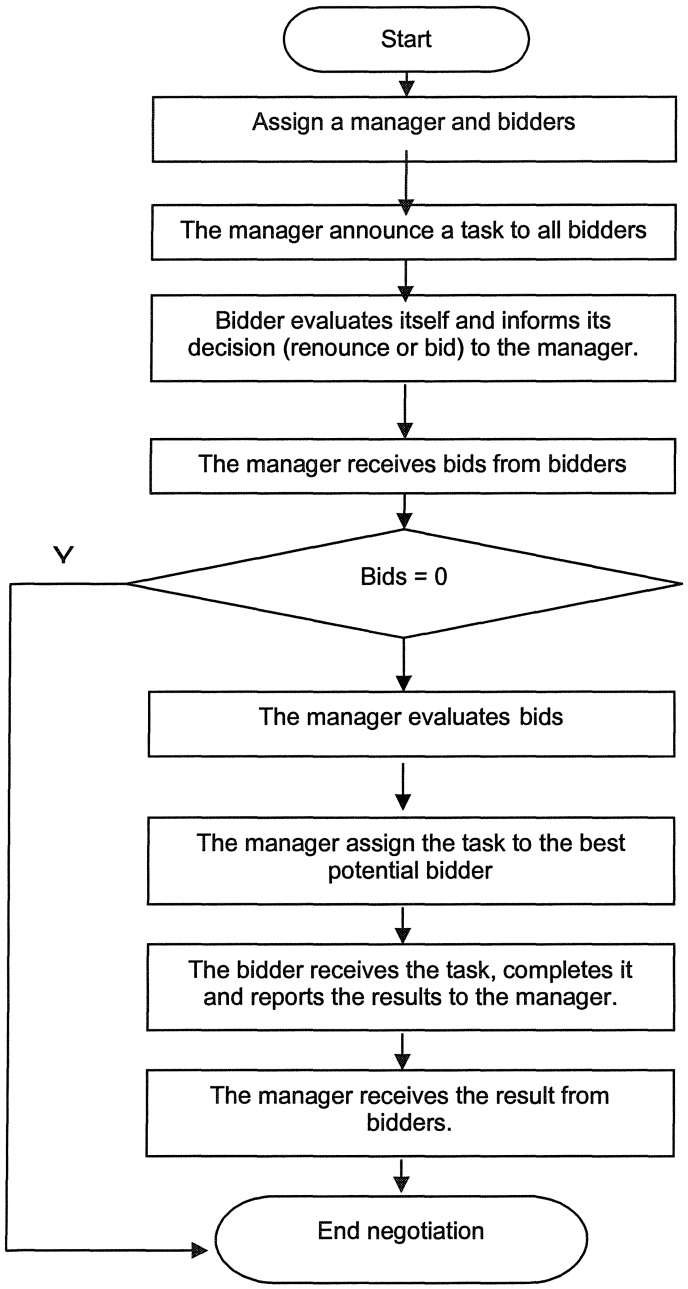


Figure 4.3 The working algorithm of the contract net

hierarchical broadcast architecture with two other well known MAS approaches: the ***contract net*** [Muller 97] and the ***joint plan*** [Muller 97] structures.

In the contract net, all basic agents, called bidder agents, initially do not know the individual roles they will play. They only have knowledge of their capacities about certain tasks and all they can do is to send that information to their manager. This is called the bidding. As for the manager, it has no knowledge about the bidder agents so it needs their bids to make a decision. The tasks assignment depends on the result of the manager evaluation of all bids. During the process of assigning the tasks a negotiation process takes places since more than one basic agent may have bid for the same tasks.

Once the negotiation process has succeeded, the global plan is executed. Note that the contract net approach is known not to be very appropriate to dynamically changing environment. Figure 4.3 shows the working algorithm of the contract net.

For the joint plan structure, a voting process is needed to assign the basic agents' roles and very often a negotiation process is necessary to establish a global plan. Given the nature of the mechanism used to specify the basic agents roles, when conflicts happen it becomes difficult to identify what the conflicts are about and which participants are part of the conflicts. Part of the answer to this is that the basic agents must posses strong individual joint plan libraries. Thus the joint plan approach is, as the contract net, not very appropriate to dynamically changing environment. Figure 4.4 shows the working algorithm used in the joint plan approach.

In our hierarchical broadcast approach, the roles of the basic agents are totally defined and those are known to the higher agents. Furthermore, each agent's view is strictly limited to its own role so conflicts with other agents of the same levels are impossible. Hence, there is no need for voting or bidding processes and the establishment of the global plans do not need any negotiation process as it is the case with both the contract net and joint plan approaches. The

assignment of the individual tasks are always appropriate to the size and capacities of the lower agents. As for conflict situations occurring at the lower

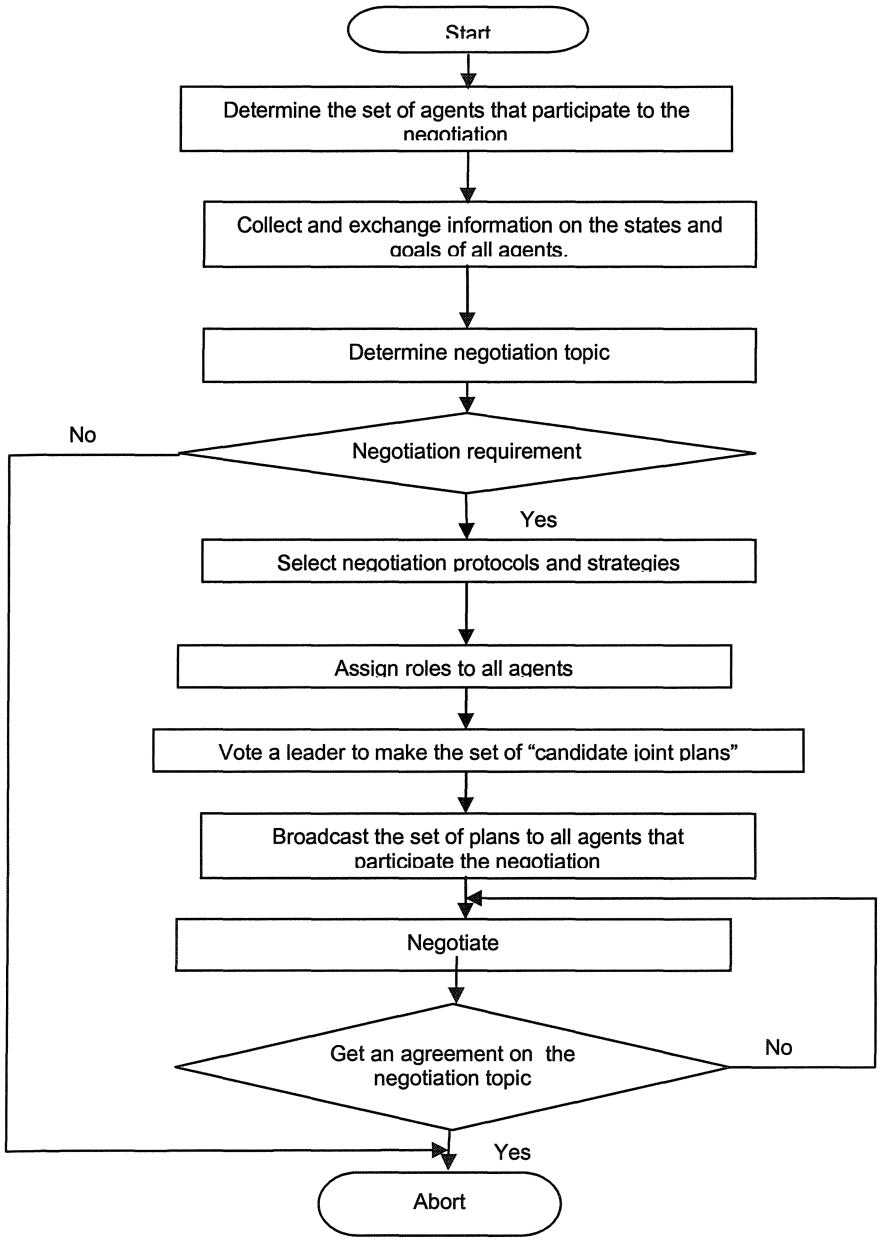


Figure 4.4 A negotiation process of a joint plan

levels, the higher agents are immediately aware of them and they can easily be resolved by the use of the simple reactive patterns.

As we mentioned in Chapter 1, multi-agents architectures are application oriented due to the complexity of real practical problems. Without any doubts, “contract net” and “joint plans” approaches are two good mechanisms to use in multi-agents systems applications. However, for many applications which need a multi-agents system approach, particularly e-business applications, the close correspondence between our “hierarchical broadcast structure” and the real application structure makes our approach much better than any other actually known. Such is particularly the case with e-business applications and network management problems to mention only a few.

4.4 Summary of advantages

To summarize the advantages of our approach, we point out some of its important characteristics. As mentioned in section 4.1, each agent in the system acts as a self-interested agent. If and only if the requirements from their higher agents do not enter in conflict with their actual executing goals, will the agents accept to accomplish the new requirements. If conflicts do happen during the execution of a cooperative plan, each basic agent need only use cooperative patterns to negotiate with their higher level agent to attain an agreement. Basic communication primitives such as “propose", "accept", "reject", "confirm" and "report” plus simple negotiation protocols and strategies are implemented in their cooperative patterns library. This permits the use of reactive patterns to achieve “cooperative goals” in IDEAL agent. The use of cooperative layers, containing reactive patterns, both at the manager and basic agents levels replaces the complex cooperative procedures used by the joint plan and contract net approaches. Our approach brings us the following benefits. It increases the capacities of establishing cooperative plans in a multi-agent system. It also reduces the costs of communication and negotiation. Our hierarchical approach

makes it easy to introduce learning mechanisms so as to give the agents the capacity of self improvement. Finally, the simplicity of the cooperative layer of the IDEAL architecture increases the robustness of the agents.

Chapter 5: An intelligent currency options trading assistant system (ICOTAS)

In this chapter, we present an example of application in the field of currency options trading. The resulting system is an intelligent currency options trading assistant system (ICOTAS). Our objective is to illustrate how IDEAL agents are used to build real application systems. First, we present the concepts of currency options and option trading, even though some of them have introduced in the section 4.2. Then, we propose the design motivation of the ICOTAS system. The strategies used in currencies options trading are discussed and the outlines of an ICOTAS system are given. After presenting examples of plans and patterns used by ICOTAS, we illustrate a scenario of a currency option trading procedure.

5.1 The concepts of the currency options

A currency option can be seen as a right on a currency future contract. When you buy an option, it means you buy a certain right on this currency future contract. There are two types of currency options: the ***call*** option which gives you the right to ***buy*** at a given fixed price currency future contracts while the ***put*** option which gives you the right to ***sell*** it at a fixed given price. This fixed price is called the strike price. Obviously, when the price of some currency increases, if you hold the related call option, you make money. Similarly, when the price of some currency decreases, if you hold the related put option, you make money too. With each option there is a closing date that specifies the day at which the option expires. For currency future options, it is the second Friday preceding the third Wednesday of the month. Note that the option closure date is different from that of the related currency future contract.

The “intrinsic value” of an option, if it has any, is always the actual cash difference between the strike price and the current price of the underlying futures

contract. For example, let us assume that the March Japanese yen futures contract is priced at 0,7000 cents par yen, then the March 69 yen call option has an intrinsic value of 0,0100 cents per yen while the March 69 yen put option has no intrinsic value. Similarly, if the price of the futures contract is at 0,6800 cents per yen the March 69 yen put option will have an intrinsic value of 0,0100 cents per yen and the March 69 yen call option has no intrinsic value. Options with no intrinsic value are called "out-of-the-money" and those with intrinsic value are called "in-the-money". In options trading, to guarantee that you can buy in the options you want, you must pay the nearest out-of-the-money price. Consequently, if you want to purchase a call option for a given strike price, you must pay for the next higher strike price than the actual one; for a put option, you pay for the next lower strike price.

Since the investment needed in options trading activities is far lower than the one needed in futures contracts trading, many investors, especially non-professional investors, are attracted. However, buying and selling which kind of options in order to make money is a complex decision making problem. It involves many factors such as risks estimating, profits settling, option price choosing, investing period planning, market change predicting... etc. Professional investing knowledge is absolutely needed. Therefore, the eager investors are in need for some intelligent systems which could be used to provide professional help in the trading activities.

Trading currencies options involves buying and selling options related to currencies futures contracts. Having access to expertise in the field of currency futures trading, this motivated us to develop an ICOTAS.

5.2 The design motivation of the intelligent currencies options trading assistant system

The motivation to design an intelligent currencies options trading assistant system is to provide personal investors with solutions to currencies options problems. Such a system can accept delegations from personal investors and in a way similar to human brokers, work out investing plans, monitor markets changes, make trading operations, and bank profits of trading.

5.3 The investing strategies used in currencies options trading

Changes in the prices of currencies involve political, economical and social factors and many other aspects. In theory, predicting changes in the market is impossible. “No one, not even the most astute currencies futures analysts, could say with absolute certainty, the direction of volatile currency’s next move” [Hume 90]. Seeking out strategies that provide the possibility of huge profits without having the risks of huge losses has always been the dream of all investors. There are many kinds of investing strategies for currencies options trading. In general, experienced personal investors often use some prediction-based investing strategy. When there are no important events happening in the political, economical and social aspects of the world, investors analyze the changes in the price of the currencies that happened during the recent past period of time, mine out patterns of changes and build some measure of tendency for future changes. On the base of this prediction, the experienced personal investors, -whom we shall call speculators- choose some investing options. At a later time, before the expiration time of the options, the speculators decide, on the base of their predicted tendency to sell or not the options they are holding. Obviously, such investing procedures have “unlimited risks” [Hume 90]. If the prediction is correct, they make money, otherwise, they loose money.

In ICOTAS, we use a better strategy, called the "yoke" approach. This strategy involves the purchase of both the out-of-the-money call option and the out-of-the-money put option with the closest striking prices. The meaning of "yoke" can be understood as the fact that the two options wrap around the futures price in the same way that two hands might wrap around a fast moving object. It is not based on prediction about futures changes tendency, but on more short terms changes in the price of the futures contracts. In this strategy the belief is that before the price of a currency attains a balance point, the price's change will evolve either as an oscillation around a given fixed price or as a sustained direction change process. It can easily be seen that the use of such a strategy does not intend to profit from both options, but instead, hopes to take a small loss on one direction and make a large gain on the other so that globally the result is a sizeable profit. In this way, once the price of the currency futures makes a substantial move whether shooting up or falling sharply, the yoke strategy is capable of producing very high profits. Obviously, the yoke strategy is only suited for highly volatile markets and happily currencies futures markets are such markets.

The charm of the yoke strategy is that it provides speculators with sized profits, without the risk of huge losses. Globally, the "yoke" investing strategy provides the potential for frequent returns on investment of 50% up to 150% for investment periods of two months or less. For some large move in the currency price the profit can even be as high as 350% to 500%. The investing risks can be reduced to a limited level defined by investors.

In principle, the longer the remaining time of an option, the smaller the trading risk of the option. However, the longer the time remaining of an option is, the more expensive the price of the option is. In actual trading, if the value of one option does not rise to a level sufficient to cover the cost of both options before expiration, then investors lose money. Seen this way, it may seem that the less you pay for both options, the less the value of either one will have to change before it starts producing a profit and thus the lower the trading risk of the options

is. However, in actual trading, the lower you pay for premiums, the shorter the time remaining of the options you bought is, the higher the trading risk is. Hence, strictly selecting a good investing strategy is not enough, determining the price you pay for the options and timing your entry to the trade are also a key factors. The operations used to evaluate both of these factors are decomposed into sets and sequences of basic operations. In our ICOTAS, they are organized and defined as operation rules.

For example, for a Japanese yen analyzer agent (JYAA) part of an ICOTAS, "the right market conditions for entering a trade" are situations such that the price of Japanese yen futures either

> moves sharply more than 200 points or more within 3~5 days **:** a ***burst situation***, or
>
> moves in same direction over 3~4 weeks and more **:** a ***sustained situation***, or
>
> remains volatile within the 200~300 points' range **:** a **fairly stable situation**, or
>
> moves sharply upward at least 100 points within 3~4 days and then reverses sharply at least 100 points and moves upward again, and then goes down … **:** a **choppy situation**.

These rules are called threshold rules. When the market conditions are right, it is not unreasonable to see changes in the price of Japanese yen futures of over 230 points in one direction [Hume 90]. This represents about $2300 gain in a Japanese yen futures contract trading procedure. Hence, in our yoke strategy we state as our *basic entry rule* the fact that only when the total premium for both call and put options will be $2300 or less, will our risk on a single 'yen yoke' combination be limited. The loss will never be more than $2300. Furthermore, according to our above discussion, when there are more than one set of options satisfying the basic entry rule, the best candidate is the set with the most remaining time. This we call "the longest remaining-time" rule.

Consequently, by following these rules a Japanese yen analyzer agent can run the yen yoke strategy. JYAA recognizes the right time to begin a trade by using the threshold rule. It purchases the best yen yoke combination of options using both the basic entry rule and the longest remaining-time rule. When the price of the Japanese yen futures moves over the 230 points mark, a JYAA starts making money for the investors.

Of course, the yoke strategy is only one possible investing strategy among many that can be used by ICOTAS. The currency trading analyzer agents in ICOTAS can select the investing strategies according to the user's investment requirements, but the yoke strategy remains the one most often used. All investing strategies and operation rules used by ICOTAS are implemented by plans and patterns. It is through the execution of plans and patterns that ICOTAS exercise currencies options trades for its users. Examples of plans and patterns will be shown later.

5.4 The intelligent currency options trading assistant system (ICOTAS)

The intelligent currencies options trading assistant system is a multi-agents system that simulates human traders' behavior. It is used to personally enter currencies options trades. It can accept delegations from personal investors and "smartly" manage an investment in order to gain a sizeable profit during a given period of time specified by the investors. It has, to some degree, abilities to learn from its experiences, i.e., to self-improve its behavior. Figure 5.1 illustrates how ICOTAS are developed.

There are six different units in the system. On the client side, the user interface agents are in charge of the interactions between the investors and the server side of the system.

The registration agent and the registration database implement the management of the users' registration. At this level, are also found the users' identities

recognition functions and users' investment accounts management procedures. Only validated users have the right to use ICOTAS to exercise currencies options trades. A validated user is a registered with a non-empty investment account.

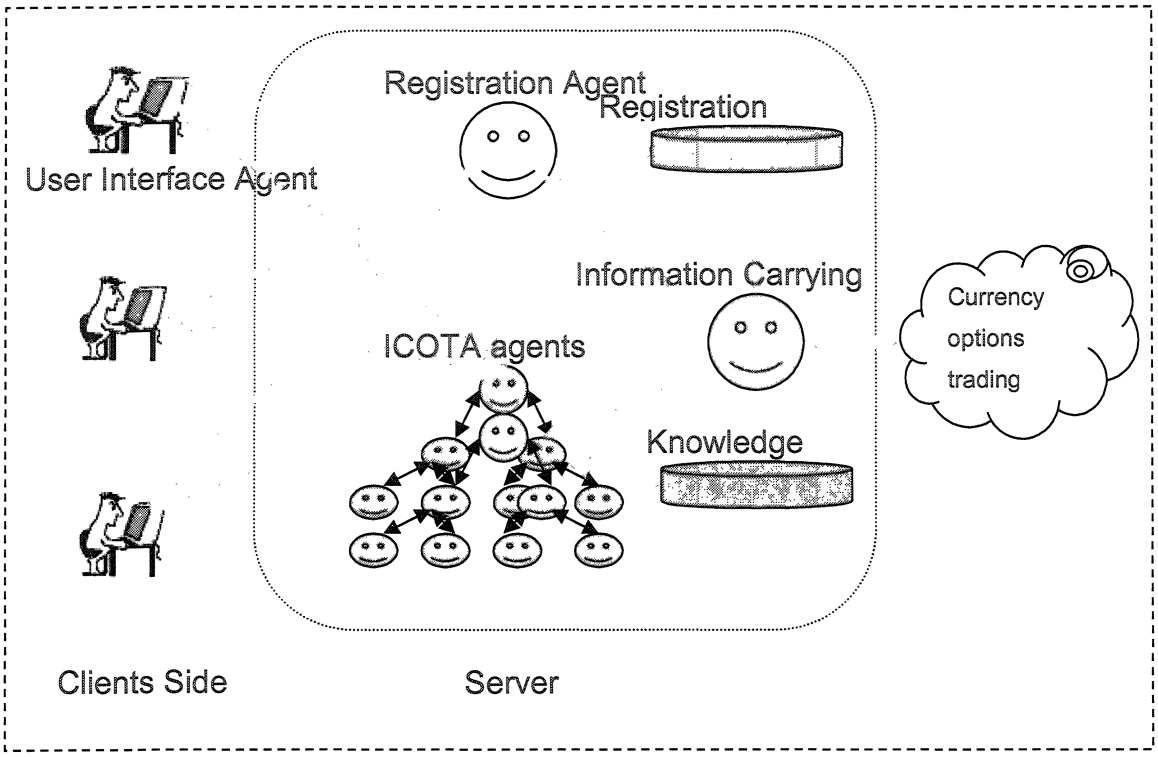


Figure 5.1 The architecture of intelligent currencies options trading assistant system

The information carrying agent is in charge of communications between ICOTAS and on-line currencies options trading services. Its principal tasks are data history retrieval, current markets information gathering and trading message interpretation.

Intelligent currencies options trading assistant (ICOTA) unit and knowledge base together implement the activities of investing strategies making and currencies options trading. There are two types of agents in the ICOTA unit: the trader agent and currencies analyzer agents. The trader agent works like a trader who

estimates the currencies futures markets situations, selects right investing strategies, works out global investing plans, manages the users' investments and coordinates the currencies analyzer agents' activities. A currency analyzer agent, such as a Japanese yen analyzer agent, works like a trader's assistant who is responsible for all trading operations in futures and related options for a given currency. According to the investors' requirements, the JYAA generates an appropriate local investing plan, monitors the changes in the price of the currency futures and selects the right time to start a trade. Note that in the above architecture, the ICOTAS may be a multi-agents system, but individually taken, all agents are implemented with the IDEAL agent architecture.

When a user wants to make investing operations, he (or she) first must pass some identity validation. Once the registration agent confirms the presence of a valid identity, communications between the user and the ICOTA unit are started. At this time, the user may require the ICOTA unit to show, as plots for example, the price evolution of some given currencies futures or inquire about which currencies futures offer the best investing perspectives. The request will invoke a plan called "markets-situations-analyzing" in the trader agent which will return expected results by the user. Next, the user may specify his investment parameters that include the chosen currencies futures, expected profit, risk and investing time, and delegates ICOTAs to make trading operations on his behalf. The trader agent will analyze the requirements and current markets conditions and verify the cash balance in the user's investment account. If any rules are broken or there are contradictions in the requirement, then the requirement is rejected. Otherwise, a global investing plan is triggered. The execution of the plan will result in several sub-tasks being broadcasted to currencies analyzer agents (CA agent). Each CA agent strictly accepts the sub-tasks that fall in its competence field. Sub-tasks can be seen local goals. For example, a JYAA strictly accepts Japanese yen futures options trading tasks. The agents that have accepted sub-tasks send to the trader agent an acceptation confirmation message. Local plans are generated in order to achieve the required tasks. The

activated CA agents complete the currencies-options-trading operations and return the associated profit or loss information to the trader agent. The trader agent accordingly updates the balance of the user' investing account and sends to the user a report via the user interface agent.

To show more details of CA agents making trade operations of currencies options, we will illustrate some samples of plans (patterns) used in CA agents below and give scenario of a currency option trade procedure in the next section. The figure 5.2 shows a pattern used by JYAA.

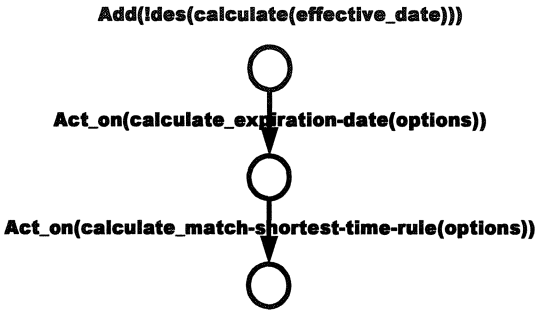


Figure 5.2 Sample 1 of patterns used by JYAA

The head of the pattern is *Add(!des(calculate(effective_date)))*. The pattern is used to filter out options candidates that satisfy the given investing time limit. Once the head is matched, then the agent first executes an action that selects all options candidates expiring before investing time is over. Then, it runs the second action that filters out candidates that satisfies "the minimum risk time rule"[25]. The result of executing this pattern is a set of options that expire before the investing time limit and that with minimum investing time risk.

---

[25] The rules states that effective dates of entry to a trade are such days that are prior to 4 weeks of the option contract expiration.

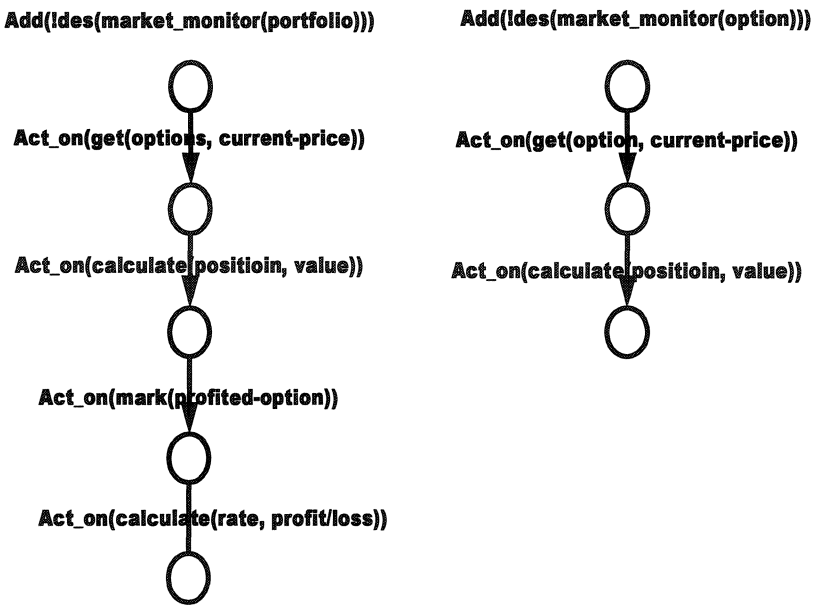


Figure 5.3 Sample 2 of patterns used by JYAA

Figure 5.3 shows another two patterns used by JYAA. The left pattern is used to monitor the state of a combination of options that have been bought. Once the invocation of the pattern is matched, then the agent executes action *Act_on(get(options, current- price))* to get the current price of each option in the set. And then, it runs actions that calculate the current value of the set of options, mark the option that has made profit, respectively. Finally, it calculates the rate of profit/loss of the set of options. The right pattern is used to monitor the state of a specific option.

In figure 5.4, we show a plan used by a JYAA. This plan is used to recognize the markets condition necessary to start a trade of options. The head of the plan is *Add(!des(get(market-tendency))) : at_least(market-data, 5)* with precondition[26] stating that "at least 5 weeks of market data are to be used". When the conditions in the head of the plan are matched, the agent executes an action to

---

[26] The precondition is the part following the " *:* " symbol.

get price information of the given currency futures for the 4 most recent weeks or more and if the plan gets the right to continue (see the chapter 3), the agent uses the default algorithm to recognize the type of market change.

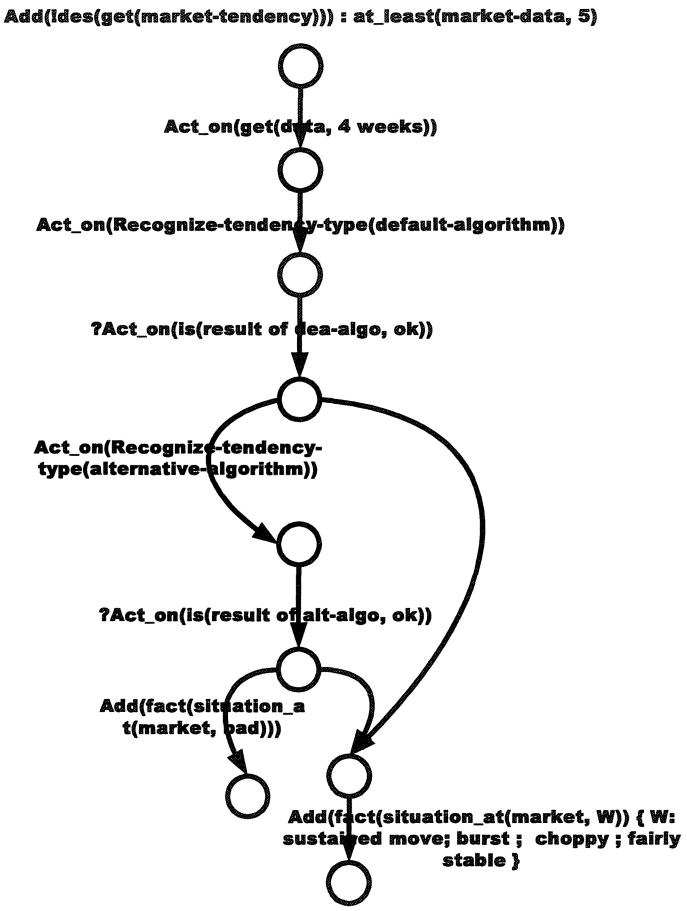


Figure 5.4 Sample 1 of plans used by JYAA

The result is tested in action *?Act_on(is(result of dea-algo, ok))* using recent data. If the result passes the test, then a new belief *fact(situation_at(market, W))* is generated. Otherwise, the agent will use another alternative algorithm. (*In the design, we use such a rule that the precondition of the leftmost flow-out edge of a node always is negative of holding condition of testing-type flow-in edge of the node, i.e., it simulates, under the case that there only are two flow-out edges,*

the control structure "IF '?condition' is not true, THEN run the leftmost edge, ELSE run the rightmost edge"). Suppose the result obtained by the new algorithm still cannot pass the verification test, then situation of the market is undeterminable by the agent. The agent announces that the market condition is bad and the new belief *fact(situation_at(market, bad))* is generated.

Figure 5.5 shows a ***top-level*** plan used by a JYAA. The head part of the plan is:

Add(!des(invest(op, X))) : fact(situation_at(risk, Y)) & fact (situation_at (profit, Z)) & fact(at_least(invest-time, V)) & (X = Jap) & (Y = small) & (Z <= 50%) & (V >= 4)

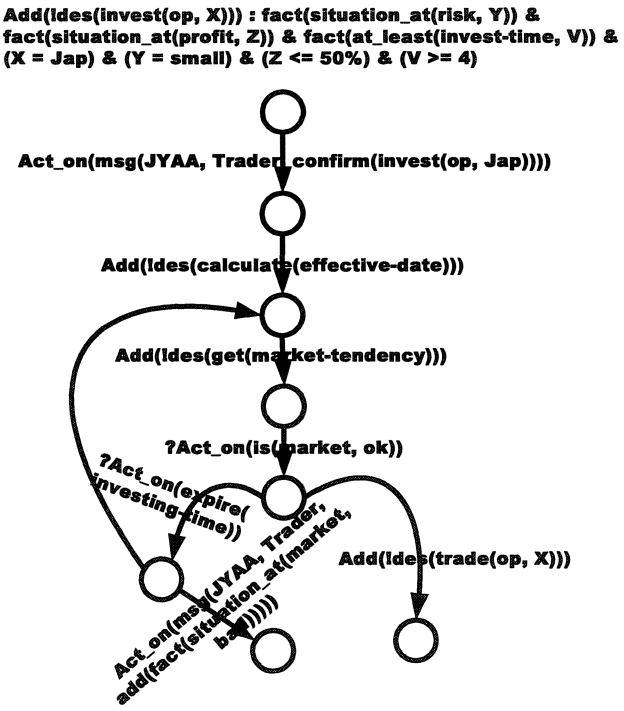


Figure 5.5 Sample 2 of plans used by JYAA

It means that this plan will be used to trade Japanese yen futures options with precondition of that the risk is small, the net-profit target is less than 50% and the

investing period of time is at least 4 weeks long. Once the conditions are matched, the body of the plan is executed.

First, a message is sent to the trader agent to confirm that a JYAA has accepted the sub-task "*invest(op, Jap)*". If the plan is assigned the right to continue execution, i.e., there are no other plans, in the intention structure of JYAA, with higher priority, a desire is achieved by invoking a sub-plan that selects options that expire before the end of the investing time and with minimum investing time risks. Next, the agent commits a desire that analyzes current currencies markets situations. If the analyzing result passes the test *?act_on(is(market, ok))*, then "the threshold rule" is satisfied. The agent will attempt to achieve the desire *!des(trade(op, X))* that triggers a real options trading procedure. Otherwise, it means that it not a good time to entry trade of the currency futures options now, that is, the agent has to wait. The agent checks if the investing period time is still acceptable. If so, then the agent continues to analyze the markets and seeks a right time to start the trade of options. Otherwise, it informs the trade agent that the given task -*invest(op, Jap)*- can not be completed because the market conditions are bad for the investing period of time given by the user.

The figure 5.6 illustrates the ***local*** plan used by JYAA when the market condition are acceptable for option trading on some Japanese yen futures contracts. It represents how to complete the trading operations for the yoke combination of Japanese yen futures options. Preconditions are set to *small risk* and *less than 50% net-profit target* and *sustained move market condition* and *at least 4 weeks investing time*. This plan accesses the set of options with minimum investing time risks generated by its top-level plans (Fig.4) among which, when all conditions are matched , the agent tries to select the set of options satisfying the basic entry rule. If such a set of options does not exist, then it means that ***now*** is not good time to start a the trade. The agent continues to seek a right time until expiration of the investing time period in which case, the agent will send the trader agent an information message stating the "abandon of the investing activities because of

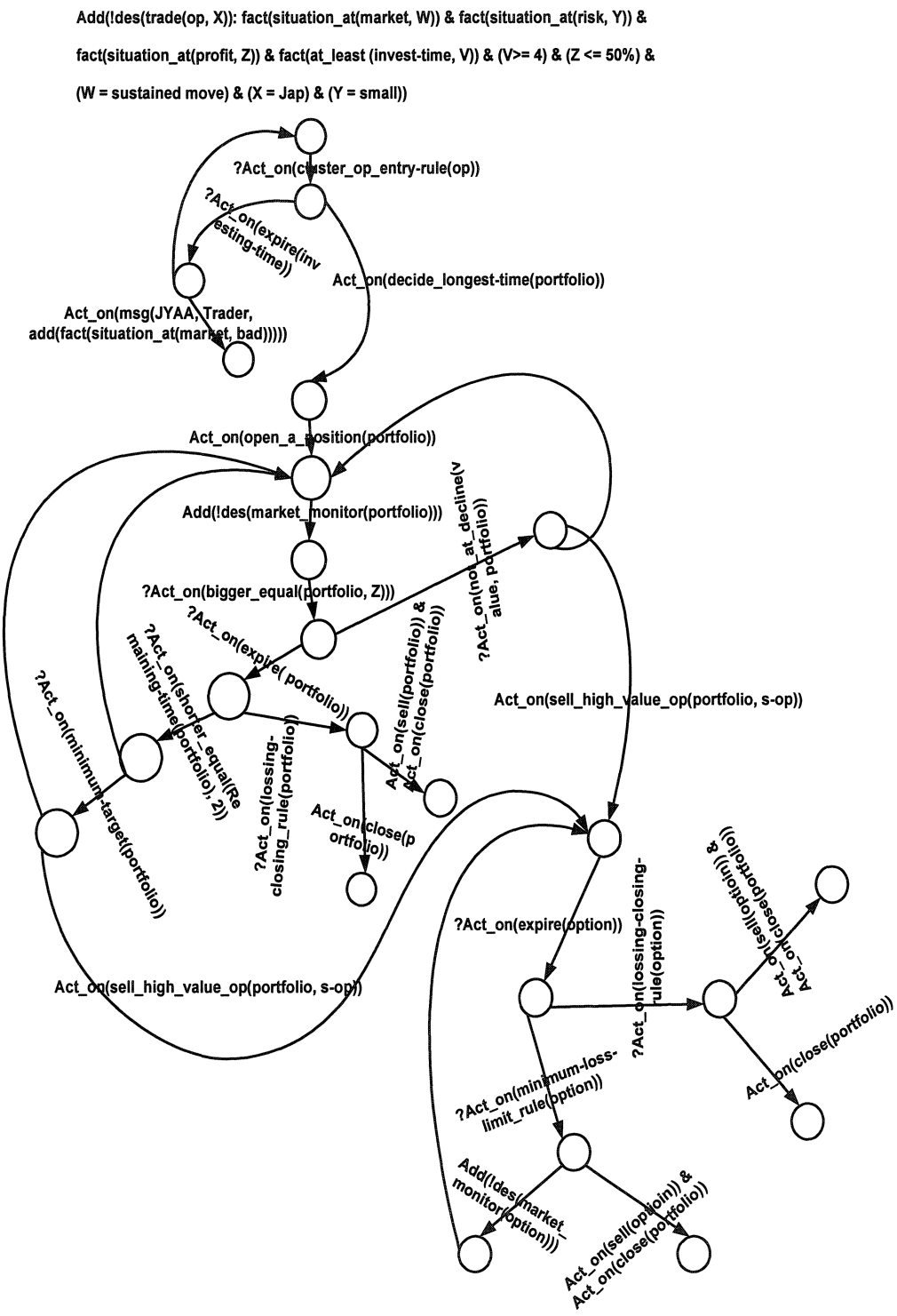


Figure 5.6 Sample 3 of plans used by JYAA

bad market condition." Whenever the agent finds a good time to start a trade, it uses the longest remaining-time rule to choose the best combination of options for the yen yoke investment strategy. The agent then runs the action *Act_on(open_a_position(portfolio))* to purchase the selected set of options. After this, the agent commits a desire *Add(!des (market_monitor(portfolio)))* that invokes a sub-plan to monitor the change of value of the "yen yoke set". Once the value is bigger than the targeted net-profit and the value has a declining tendency, the agent sells the option that generated the profit. For the remaining option, the agent checks, every day until it expires, if its value matches the condition of "the minimum loss limit rule" stating that "$1000 is the lower limit for the loss associated with this option. If yes, then the option will be sold and the whole trade is ended. Otherwise, when the option expires, the agent checks if its value matches the condition of "losing option closing rule" that says: "unless a losing option is worth at least $100, do not bother to sell it". If yes, the agent sells the option to reduce the loss. Otherwise, the agent closes the entire trade. In situation when value of the "yen yoke set" has not yet attain the net-profit target, the agent checks its value every day. When the remaining time becomes less than 2 weeks, the agent "smartly" changes its net-profit target, and checks if the value of the set of options matches the condition of "minimum net-profit target rule" stating that "$700 is recommended as a minimum net-profit on a timed option trade". If yes, then the agent sells the profit-making option. Otherwise the agent wait for good profit time.

5.5 Scenario of a currency option trading procedure

Based on the presented samples of plans and patterns used by the Japanese yen analyzer agent, we give a scenario for a Japanese yen futures options trading procedure to illustrate how Japanese yen analyzer agent completes trading operations of options on behalf of the personal investor.

Suppose that the pattern of Japanese yen futures price since 2002 February is shown in the figure 5.7.

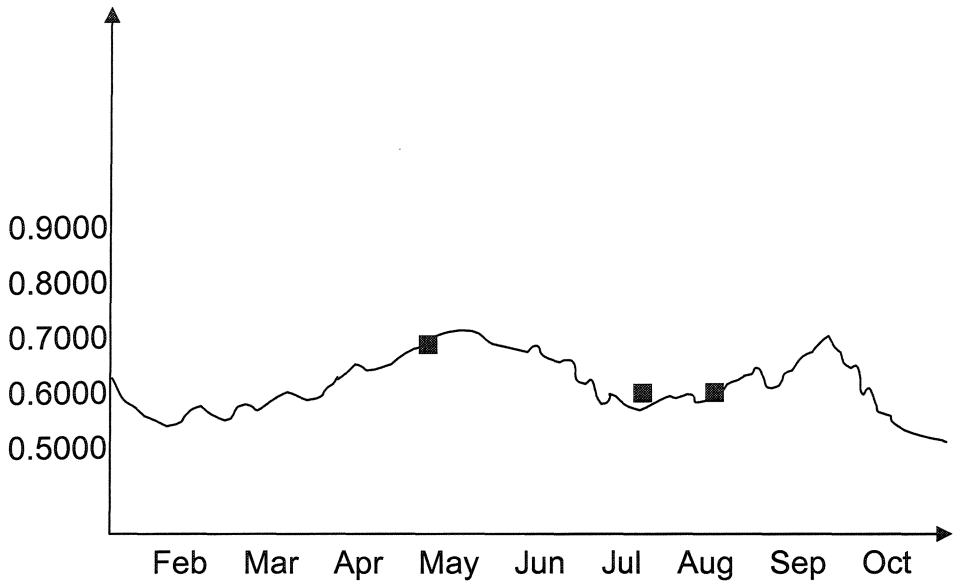


Figure 5.7 The pattern of Japanese yen futures price in 2002

In May 3, a validated personal investor used ICOTAS to do currencies futures options trading. The information provided by the trader agent indicated that Japanese yen futures price had a sustained move in recent a period. The market condition provided a potential chance to make money. Thus, the user ask for a trade of Japanese yen futures options with additional conditions "small risk, more than 30% profit, shorter than 17 weeks time", and delegated the ICOTAS to do

FUTURES PRICES				
May 3, 2002 JAPANESE YEN (IMM) 12.5 million yen; $ per yen (.00)				
	Open	High	Low	Settle
June	.6841	.6843	.6828	.6836
Sept	.6887	.6905	.6854	.6871
Dec	.6901	.6922	.6889	.6895

FUTURES PRICES						
May 3, 2002 JAPANESE YEN (IMM) 12.5 million yen; cents per 100 yen						
	Calls settle			Puts settle		
	July-c	August-c	Sept-c	July-p	August-p	Sept-p
68	1.76	1.93	2.15	.06	.25	.46
69	.91	1.17	1.28	.20	.53	.77
70	.33	.69	0.94	.62	.88	123
71	.11	.37	.58	1.39	1.64	1.85

Table 5.1 The prices of Japanese yen futures and options in May 3, 2002

trade operations on his behalf. Therefore, the trader agent broadcasted a sub-task *invest(op, Jap)* to the analyzer agents. JYAA accepted the task and activated a local goal to autonomously implement the task. JYAA purchased a proper set of options constituted of a 68 put option and a 69 call option for September Japanese yen futures contract since the actual price of the September futures contract is 0.6871 ¢/¥ through the use of the yen yoke investing strategy and operation rules defined by us. With the changes of price of Japanese yen futures, JYAA "smartly" sold both call options and put option, respectively, at reasonable time. Finally, JYAA returned the investor a sizeable profit. The trading operation procedure is illustrated in the figure 5.8.

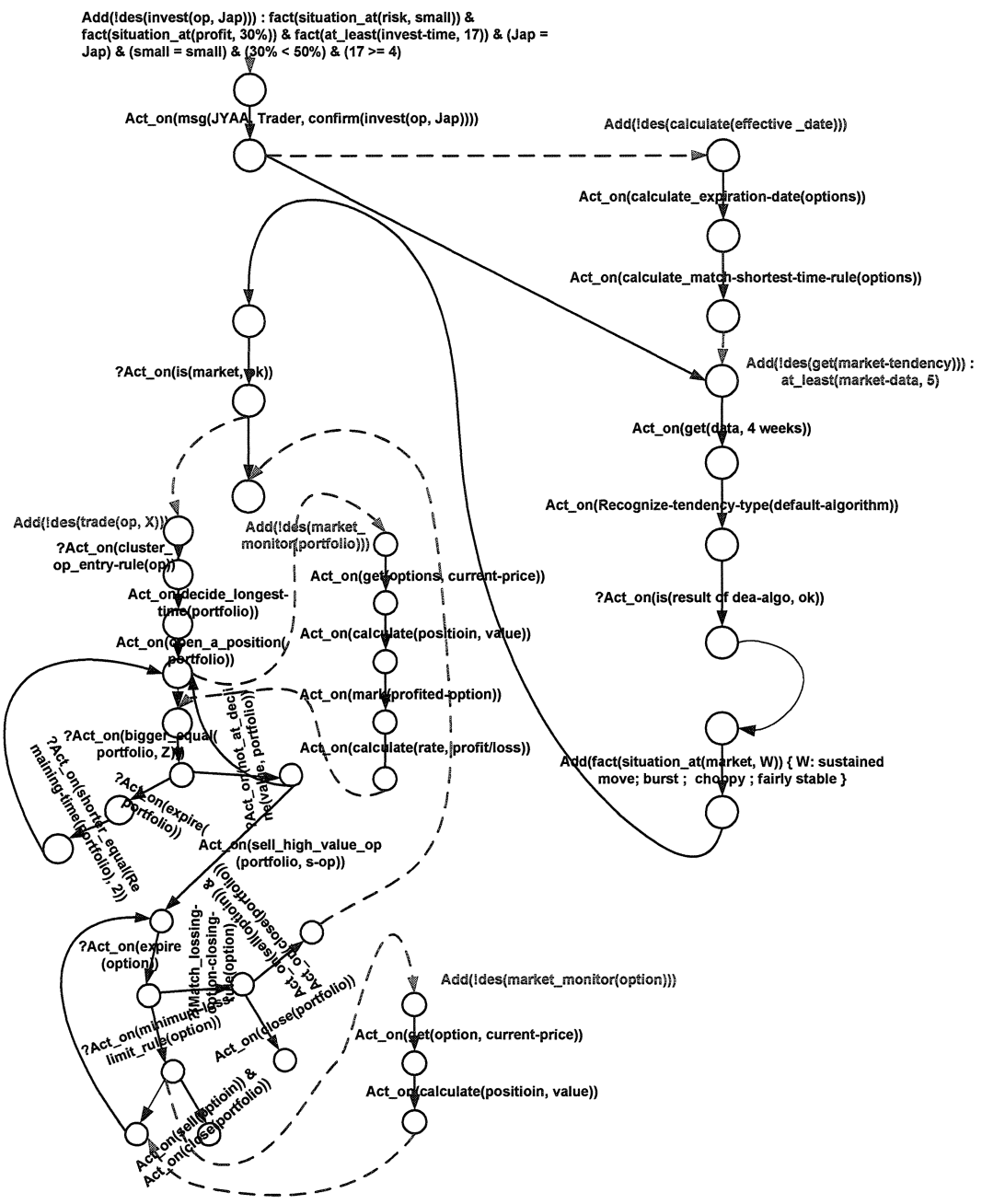


Figure 5.8 Scenario of Japanese yen futures option trading

where the red dotted lines represent connections between top-level plan and sub-plans.

When JYAA accepted the *invest(op, Jap)* task, the invocations and preconditions in the heads of each related plans were matched, the plans were partially instantiated and the preconditions became ground beliefs (see figure 5.8). After deliberation, a plan was selected. With execution of the top-level plan, first a confirmation of acceptance of the task was sent to the trader agent, and JYAA got the right to access the investor's account. After committing a desire, a sub-plan was activated. In the sub-plan, the agent selected sets of options respecting the investing time requirements. Only the options for the June and September futures contracts were validated (see the table 5.1). Then the agent refined the candidates using "the minimal risk time rule", thus, only the options for the September futures contracts were chosen. The sub-plan ended, and control returned to the top-level plan. A new desire was committed and another sub-plan was activated, which recognized the market condition situation as "sustained move". Thus, the top-level plan launched a reasoning procedure from which, the plan chosen to be activated was the plan shown in figure 5.6. In the plan, the agent used the method of "out-of-the-money" defined in "yen yoke" strategy, "the basic entry rule" and "the longest remaining time rule" to filter the candidate options. Consequently, the only possible combination of options was constituted of the September 69 call option and September 68 put option. Then a XML file containing the appropriate information was sent to some on-line currencies trading service, where the set of options was bought on behalf of the user. (In ICOTAS, we presented it as opening a position). The next step was to find a proper time to sell the profit-generating option. The plan did very well. After a month, the put option was marked as having generated its targeted profit. Starting on August 2nd, the value of the put option declined for two consecutive days. It was our signal to sell. Hence another XML file was sent to the same on-line currencies trading service and the 68 put option was sold. The agent continually monitored the market to sell the remaining 69 call option. On August 16th, the value of the option matched the losing-option-closing rule and the agent sold the option via a XML file and closed the position. The control returned to the

top-level plan, and the trade operations procedure ended. The investor's account was updated.

5.6 The results and discussion

The intelligent currencies options trading assistant system is still a complex system under development. To show its capacities and performance, we completed an off-line theoretical simulation of a JYAA using 1988's Japanese futures and futures options data as provided in [Hume 90]. The result of the simulation is quite heartening. In the simulation, we used three different investing strategies: a prediction-based gamble strategy, a "yoke" combination strategy and a smart learning "yoke" combination strategy. In 12 months, the "yen-yoke" agent totally found 15 different times when trades could be started. Over the same time period, we ran the "gamble" agent and the "smart" agent. The results are shown in the figure 5.9 and the figure 5.10. In fact, we use a modified prediction-based gamble strategy in the simulation. That is, although we use different algorithms for Japanese futures price's tendency recognition, we still use the operations rules that are normally specific to yen-yoke agents, such as "the longest remaining time rule", ... It reduces the possibility of speculation failure for the gamble agent. Furthermore, we find that in some time periods, the gamble agent's profits are higher than the one for the yen-yoke agent. The reason is that the gamble agent's premiums are always lower than those of the yen-yoke agent and the smart agent.

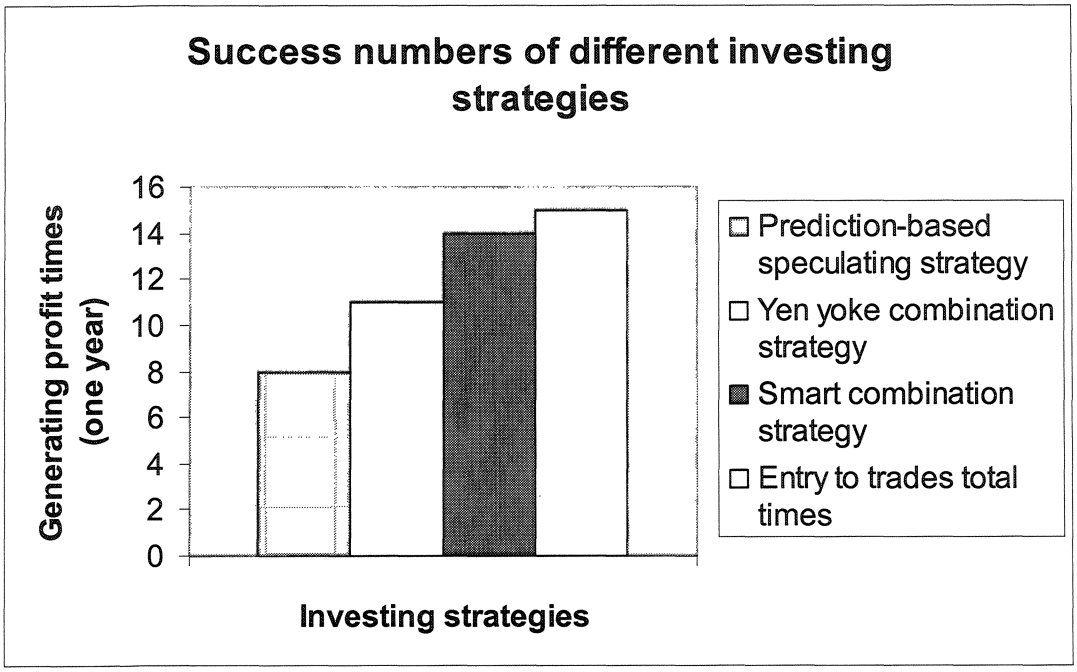


Figure 5.9 Comparisons of performance of three investing strategies (1)

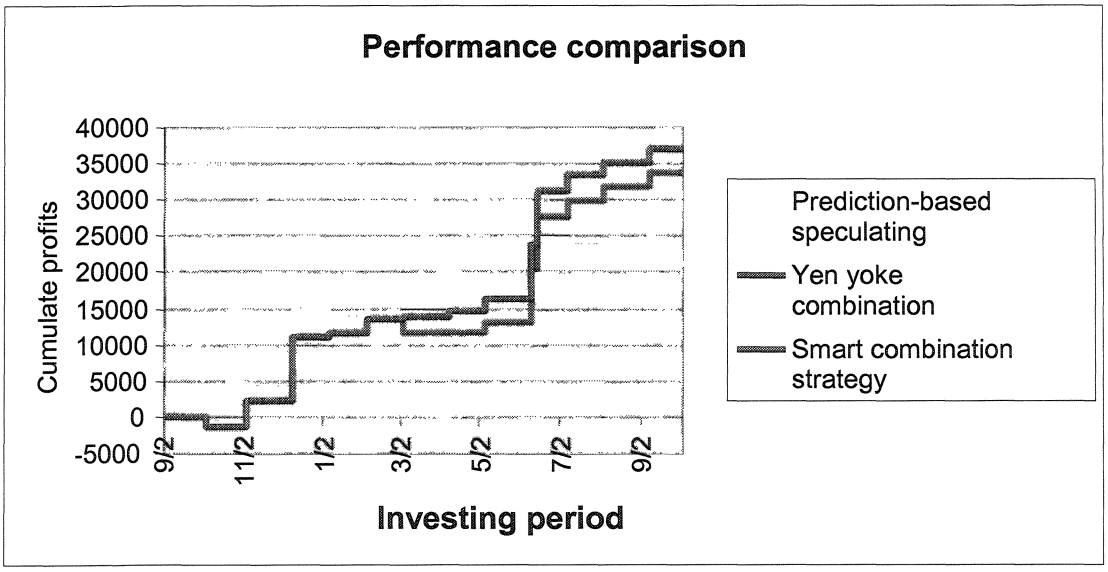


Figure 5.10 Comparisons of performance of three investing strategies (2)

But over a long period of time, the yoke combination strategy reduces the investing risks and increases the gain potentials.

Our results show either in terms of number of times profits are generated or in terms of cumulate profits, the yen-yoke agent's performance is far better than the gamble agent. From our results we also see that the smart agent always provides the best performance. Mostly, it comes from the fact that its control ability over the option selling time gradually improves through learning.

It is important to notice that the results show that ICOTAS can effectively satisfy the system's design requirements as stated in 5.2. What is also important to notice is that the example shows that IDEAL agents are very suited to serve intelligent e-commerce applications. The intelligent systems built with IDEAL agents can effectively provide satisfying solutions to e-commerce applications problems.

Chapter 6: Comparisons

As mentioned in the chapter 1, the aim of the research is an attempt to propose a new agent model, through the use and integration of new and already existing approaches and mechanisms, in order to satisfy the requirements for complex agent systems such as e-commerce, network management, traffic control, air-battle management and agent teaching. We belief that we have attain this objective. To reveal this, we give out a detailed qualitive comparasions, in this chapter, between our architecture with famous agent models and frames, the results should illustrate that IDEAL agents really provide better qualities than those famous ones in almost each aspect of an agent architecture.

6.1 Qualitative comparisons between IDEAL agent model and related agent models

From an architectural point of view, if we compare the IDEAL agent model with other famous models, we see that the IDEAL agent possesses almost all of the advantages brought forward by these models while expressing none of their disadvantages. Therefore, the IDEAL model is more creative, more efficient and more performing. Table 6.1[27] shows the qualitative comparison factors and their values for the different agent models the most often found in literature.

---

[27] In the table, the sample for reactive agent is Brooks' subsumption architecture. The sample for deliberative agent is Bratman's agent model or Rao's agent interpreter.

MODEL	REACTIVE AGENT	DELIBERATIVE AGENT	INTERRAP AGENT	PRS/ DMARS	IDEAL AGENT
ARCHITECTURE	SIMPLE	SIMPLE	HYBRID	SIMPLE	HYBRID
LAYERS	SINGLE LAYER	SINGLE LAYER	VERTICAL MULTIPLE LAYERS	SINGLE LAYER	HORIZONT. MULTIPLE LAYERS
REASONING ABILITIES	NO	YES	YES	YES	YES
META-REASONING ABILITIES	NO	NO/ YES	NO	YES	YES
REACTION SPEED	FAST	SLOW	FAST	SLOW	FAST
LEARNING ABILITIES	NO	NO	NO	NO	YES
PARALELL PROCESSING ABILITIES	NO	NO	NO	NO	YES
COOPERATIVE ABILITIES	NO	NO/ YES	YES	YES	YES
DYNAMIC PLANING ABILITIES	NO	NO	NO	NO/ YES	YES

Table 6.1 Qualitative comparisons between IDEAL agent and famous agents

6.1.1 Comparison with simple agent models

Reactive agents may provide fast solutions but are only able to deal with simple problems. Deliberative agents, while providing optimistic solutions to complex problems have to use more complex and time consuming reasoning procedures. With the IDEAL agent, we present an integration of both reactive and deliberative features. The use of an intuitive planning unit provides fast "intuitive" solutions to reactive situations while the experience-based planning unit provides for *fast* and *optimistic* solutions to complex problems. Furthermore, the IDEAL agent uses its learning unit to improve its performance over time.

Clearly, what the simple agents do not possess are the IDEAL agent abilities of dynamic real time planning and of meta-reasoning, its robust satisfaction of the weak completeness property, its cooperative ability, its learning ability and its

parallel processing capacity. All these features, which by themselves are very important ones, become even more so when we understand that they are absolute necessities for intelligent agents.

6.1.2 Comparison with the Interrap agent model

In our opinion, the Interrap agent model is among the best hybrid agent models available. However, as we shall show, we believe the IDEAL model to be better.

The Interrap model is built as a vertical layered architecture; the control flow is first bottom-up and then top-to-bottom. The IDEAL model is built as a horizontal layered architecture with parallel horizontal control flows which are always oriented in the same direction. For any triggering event, the IDEAL agent's response time is decided by the slowest of the five planning units. However, since after training the experienced-based planning unit will often replace the deliberative planning unit, the IDEAL agent can provide the same level of reactivity than that of the behavior-based layer of the Interrap agent. The IDEAL agent is then working in its pattern-like execution mode and is far faster than the Interapp local planning layer and cooperative layer.

The one flaw of the Interrap model is its lack of robustness. For a given problem, once the lower layers of the vertical architecture, particularly the behavior-based layer, stop working, the whole agent becomes paralyzed. By using a horizontal layered architecture, the IDEAL agent overcomes this problem.

As for the cooperative abilities of both models, due to its different architecture, the IDEAL agent provides simpler design structure, faster speed and stronger processing abilities by its use of a top level manager agent.

Finally, the Interrap model does not support such important features as dynamic real time planing, meta-reasoning nor learning abilities. This makes the IDEAL agent even more advantageous to use.

6.1.3 Comparison with the PRS/dMARS agent models

PRS/dMARS agent models can also be seen as excellent systems. They have well-designed dynamic planers and good meta-reasoning mechanisms that can be applied to planning and intentions scheduling. They also possess well-designed exception-handling processes which improve the system's robustness. But IDEAL agents are remarkably more efficient than the PRS/dMARS agent on many aspects.

First, the IDEAL agent model provides more features than the PRS/dMARS ones, such as parallel processing, cooperative abilities, fast reactive abilities, and learning mechanisms.

Second, there are several factors that limit the PRS/dMARS agents' performance when compared with IDEAL agents:

- a single level knowledge base which results in longer plan invoking time;
- a single level competent layer which results in more complex knowledge base and architecture design thus requiring longer reasoning time;
- a lack of reactive mechanisms (pattern working modes) which implies the use of deliberative modes to deal with all problems, particularly those that have been dealt with before need to be dealt with again in deliberative modes thus needing longer processing times;
- the use of a deliberative triggering-event selection and of an intention scheduling mechanism can result in input-output bottleneck problems thus increasing the response times.

Another point is that PRS/dMARS models do not provide independent cooperative components. This increases the application design's complexity and limits the model's cooperative abilities.

Also, PRS/dMARS models have no learning abilities, meaning that they can't self-improve over time.

Finally, the PRS/dMARS architectures do not support parallel processing and thus cannot make use of the strong hardware resources of modern computers.

All these disadvantages are overcome in IDEAL agents.

6.1.4 Comparison with other horizontal layered architectures

When compared with agents using horizontal layered architecture such as TuringMachine [Ferguson 92a] [Ferguson 92b], IDEAL agents are simpler and more efficient. In particularly, the use of a competence-based layered approach avoids the necessity of inhibition and suppression mechanisms thus simplifying the agent's architecture. As for mechanisms such as dynamic real-time planing, multi-inputs batch processing, output control based on priority queues, meta-reasoning and learning abilities, they cannot be found in any horizontal layered agent models but the IDEAL model.

6.2 Comparison of a multi-agents architecture based on IDEAL agents with other implemented multi-agents architectures

The IDEAL agent system architecture is based on the "divide-and-conquer" approach often applied to solve complex problems that may be described in terms of "*consist-of*" relations. Since "*consist-of*" relations widely exist in entities in the real world, we believe that using our multi-agents architecture to deal with problems that can be decomposed according to "consist-of" relations is more effective, more economic and easier to implement and shows more performance than using other famous multi-agent architectures such as "contract net" (figure 4.3) and "joint plan" (figure 4.4).

6.2.1 Comparison with the "contract net"

When comparing the IDEAL agent system with the "contract net", the following facts are made obvious.

In the IDEAL architecture, we notice that

- each agent is assigned a fixed role and there are no redundant agents;
- manager agents (sub-manager agents) have complete knowledge of sub-manager agents (basic agents);

- the problems are decomposed with respect of the competences of sub-manager agents (basic agents) which means that the sub-tasks' sizes and complexities are appropriate to sub-manager agents;
- each agent's view of its role remains unchanged; they are indifferent to other tasks;
- in the process of task decomposition and global plan making, no negotiation are required;

while for the "contract net" architecture, we observe that

- bidder agents do not know of their roles; their only ability is to inform the manager agent of their capacities about certain tasks and await eventual instructions from the manager;
- the manager has no knowledge about the bidders; it needs information from the bidders in order to make decisions;
- the tasks assignment depends on the results of the manager's evaluation of all bids;
- without some negotiation processes, the architecture cannot work.

It can easily be seen that the IDEAL architecture minimizes the costs of negotiation and communication and thus is more performing.

6.2.2 Comparison with the "joint plan" architecture

As for comparing the IDEAL agent system architecture with the "joint plan" architecture, we know that the "joint plan" architecture first needs a voting process to assign individual agent's role, then a second negotiation process to generate global plans. In the IDEAL agent this is not needed.

Furthermore, if conflicts happen between basic agents, with the IDEAL architecture, it can be detected by the sub-manager agents. The same is true for conflicts between sub-manager agents where it is the manager agent that can detect them. Consequently, when needed, the negotiation topics and the participants are easy to identify. In "joint plan" architectures however, it becomes very difficult to solve this problem.

Another important difference is that in "joint plan" architectures, agents must possess strong joint-plan libraries while in the IDEAL architecture all basic agents (sub-manager agents) only use simple reactive pattern libraries.

Finally, the IDEAL agent architecture can resolve conflicts that happen during the execution of a global plan thus making it extremely suitable for dynamic environments, while in "joint plan" architectures it is, on the contrary, very difficult to realize since the negotiation processes can be used only before the execution of the joint plans. Once again, it is clear that the IDEAL architecture costs for negotiation and communication are a lot less than those necessitated by "joint plan" architectures.

Chapter 7: Conclusions

In the last chapter, we will give out general evaluation of the whole work. The first, we summarize that our contributions in creating IDEAL agent model and building MASHBF (multi-agent system with hierarchical broadcast frame). By illustrating their characteristics, we can reveal an important fact that the work has accomplished its design tasks and requirements. And then, we give conclusions of the research. The last, we propose the direction of future research.

Now, we concentrate our attentions on summaries for two aspects of our research:

- creating an agent model able to support "intelligence" concept;
- building an appropriate multi-agents frame based on the created model.

7.1 Characteristics of IDEAL agents

7.1.1 The intelligent agent model

We propose an intelligent agent architecture that we call IDEAL (which stands for **I**ntegrated **D**eliberative and r**E**active **A**rchitectures with **L**earning abilities). From the structural point of view, an IDEAL agent is a horizontal-layered architecture able to accept and deal with input sets in a real-time mode and to produce output sets with priority modes. In its planning mechanisms, the IDEAL agent uses **dynamic hybrid planning units** working in pseudo-parallel mode and in conjunction with a learning unit. The resulting agent model, the IDEAL agent model, makes use of concepts, technologies and mechanisms such as learning, dynamic planning, parallel processing, batch processing, shared data buffers, hybrid layered architecture, etc. which, by the way, are shed light on as they are introduced.

The characteristics of IDEAL agents are

- the integration of both reactive and deliberative abilities,
- the use of dynamic planning,
- the use of pseudo-parallel processing,
- the integration of learning mechanisms, and
- the use of reasoning-free and context-free cooperative patterns.

The importance and impact of each are briefly presented hereafter.

7.1.1.1 The integration of both reactive and deliberative abilities

Reactive and deliberative architectures are integrated into a hybrid agent which possesses the advantages of both architectures. It provides the agent not only with fast and robust reactive capacities, but also provides strong reasoning abilities to help find optimistic solutions to complex problems.

7.1.1.2 Dynamic planning

Using dynamic planning units improves many facets of the agent performance. Such planning units utilize the strategy of making their decisions as late as possible; this induces a reduction of the re-planning necessity and sometimes even avoids it completely.

The algorithms "next-step" and "march-step" give an increases performance to usual systems' real-time reaction capacities. They guarantee that deliberative plans, experience-based plans and interruptible patterns have the ability to react, in real time, to changes in the agent's internal and external states.

The use of alternative components in patterns, in conjunction with the "march-step" algorithm, avoids execution failure of patterns. This improves the robustness and response-time of the system.

Representing the bodies of plans and patterns by connected rooted-directed-and/or graphs simplifies and solves many problems usually encountered in implantations of intelligent agents. For example:

- since the bodies of both plans and patterns have the same structure, it simplifies the design of the knowledge bases, the design of plans and

patterns themselves as well as the design of an interpreter for the plans and the execution mechanism for the patterns;

- it reduces the requirement of storage space for both plans and patterns; (This is of considerable worth for mobile agents since it reduces the load of information needed to be transformed.)
- constraining the allowed components in the graphs to sequences, branches and cycles, entails the fact that ***the agent programs will be provable***;
- it eases and simplifies the development of formal mechanisms needed to define, describe, design, program and prove the intelligent agent's behaviors.

The meta-reasoning approach used to solve the usual problems of operating knowledge gives the system an increased flexibility. This is particularly helpful when treating complex problems. It enables the agent to generate optimistic solutions.

Finally, the use of an "exception-processing" mechanism makes the system satisfy the "*weak completeness requirement*" and guarantees its robustness.

7.1.1.3 Pseudo-parallel processing

Defining the planning units as parallel processes to reduce the search space in each unit results in shortening the system response time. To simplify the design of the parallel structures and further enhance the system response speed, the task assignment procedure is based on competence and uses shared data buffers. It is worth mentioning that this architecture is very well suited to modern multi-processors environments, and even more, to distributed environments. In such environments, our architecture will achieve its best performance.

In addition, by using multi-inputs batch processing, we improve the agent's real time processing abilities and increase the overall system's sensitivity. Since the execution and control units operate on a priority base, have their own buffers and suspension capacities, they have the ability to "smartly" generate output sets.

This further improves the system's real time response abilities and increases the system's sensitivity features.

7.1.1.4 Integration of learning mechanisms

The integrated learning mechanisms very much improve the system's performance. Undoubtedly, it is the system's learning capacities that give an IDEAL agent its advantage over other famous agents. None of those possess learning abilities corresponding to the true sense of learning.

Rote learning, as used in the IDEAL agent, is based on the agent's experiences. It improves the system response abilities because it avoids complex reasoning procedures and reduces the search spaces. The IDEAL agent can thus provide more optimistic solutions with reactive agents' speeds. The other learning mechanism integrated in the IDEAL agent is based on feedback and is inspired from the foraging behavior of ants. It improves the behaviors' correctness and rationality of the agent thus giving it the ability to provide optimistic solutions to complex problems.

By means of both these learning mechanisms, after a training period the IDEAL agents can adapt to their environments therefore making them exhibit fast, effective, rational and optimistic reactions to changes.

7.1.1.5 The use of reasoning-free and context-free cooperative patterns

Introducing the use of reasoning-free and context-free cooperative patterns instead of complex cooperative plans to deal with conflicts that can happen during the execution of cooperative plans reduces the time needed for negotiation procedures since the amount of information needed to be processed is rather quite smaller. In this way, the agent's response time is also reduced.

IDEAL agents can still cooperatively work together in a multi-agents system to accomplish complex tasks. The use of cooperative patterns at the basic agents level and supervision by a manager agent at the highest level –such as described in the hierarchical broadcast architecture that we presented- increases

the system cooperative abilities, simplifies the negotiation procedures and reduces the complexity of the system's design.

7.1.2 The multi-agents frame

As a complex application of the IDEAL agent model, we proposed a multi-agents architecture with hierarchical broadcast frame. In this architecture, a global goal, activated by a complex problem, can be decomposed into simpler and smaller sub-goals that are then transferred by broadcasting to lower layers. The decomposition and broadcast are hierarchically executed from top to bottom until, at the lowest layer, the basic agents are activated when receiving information pertaining to their abilities.

Each agent in the architecture deals only with the requests that it believes it is competent to deal with and returns the results to its immediate superior. When changes happen in the environment, some plans, in execution at that moment by agents, are possibly modified. Thus, some conflicts may appear among related agents.

When this happens, the agents of the same level negotiate with each other under the mediation of the immediate superior in order to achieve some agreement. In this way, conflicts are resolved by the whole system working in a cooperative mode. Global goals are attained under the management of the coordinating topmost agent.

7.2 Conclusions

Here, we can conclude previous chapters. Summarily, in our research we have proposed a definition of an intelligent agent and a classification method of software agents based on a minimum set of agent's characteristics commonly accepted which clearly distinguishes between simple agents, hybrid agents and intelligent agents.

From there, we have built IDEAL, an intelligent agent model which overcomes some of the flaws or defects present in existing agent models.

We have applied the IDEAL agent to the design a multi-agent architecture which can simplify and improve the design and development of intelligent agent applications.

We have goven an example which shows how IDEAL agent and MASHBF work.

Finally, through comparisons with other current agent models, we illustrate that agents built from our specifications provide for better performance. We are confident that our work will stimulate further research efforts in the field of intelligent agent research.

7.3 Further work

In direct continuation of the actual IDEAL agent's research and development, efforts should be centered on creating a formal mechanism, improving the learning mechanisms and the deliberative and experience-based plans generation mechanisms. Further efforts should also be oriented toward the use of fuzzy representation and fuzzy reasoning mechanisms. Another interesting avenue of research would be to investigate the possible use of IDEAL agents as nodes in neuron networks.

7.3.1 Building a formal system of the intelligent agent

To build a formal mechanism for intelligent agents can be viewed as establishing an abstraction for agent model. In order to do so we will need to build some system for the syntax and semantic aspects, establish some proof theory and develop an abstract interpreter. In fact, it will be important to find the correspondences between all of those. The formal mechanism could then be used to unify the theoretical specification and practical design of an intelligent agent. Consequently, the mechanism could be used to facilitate all the aspects of

the specification, design, programming, verification and application development of intelligent agents.

The establishment of such formal models is, in fact, a problem many researchers are trying to solve. Our approach to the problem would not be trying to solve the general problem by itself but instead, and in a more modest way, try to integrate formalization into our agent model.

7.3.2 Improvement of the learning mechanisms

The learning mechanisms improvement efforts are justified by the fact that in the present version of the IDEAL agent, feedback learning is based on the assumption that *once an action has successfully been completed the expected desire to be attained by the executing the action is, in fact, attained*. In reality, this can possibly be far from the truth.

The strict existence of the learning mechanism is still not enough to attain real good performance. There is a need to measure the real effects of executing an action and to estimate the difference between the expected *designed* desires and the desires that are really attained. Making this difference as small as possible would be very helpful to optimize the agents' behaviors. The agents' actions could be more exactly defined and their behaviors would thus become more rational. Of course, how to measure or estimate the effect of an action and what standards should be used to do so are still complex open issues.

Another interesting idea would be to build an off-line simulator that could be use as a learning mechanism. When, for a given situation, the executions' effects of all available plans all present unreasonable deviations from what the design was expected to do, this simulator would be started to modify the executed plans or even create new plans in order to get the expected results. We think that cooperative learning ability is important for better intelligent agent. So far, IDEAL agents do not have this ability but we should attempt to integrate it into our architecture. This mechanism would not only permit an IDEAL agent to cooperatively accomplish a complex learning task with the help of other agents,

but would also permit to acquire from other agents needed abstract knowledge and even transform it using a model recognition layer as in [Ferguson 92a, b[28]] and as in meta reasoning.

7.3.3 Improving the generation mechanisms of plans

In the current version of the IDEAL agent, any event invoking an experience-based plan also simultaneously invokes at least one deliberative plan. This is very effective for the training of the IDEAL agent, since at the beginning, the number of experience-based plans the agent possesses is quite small given that it has had very few experiences. Hence, when started for the first time, IDEAL agent encounters events for which either there is no related experience-based plan to invoke or, if such an experience-based plan exists, it easily enters a failure-mode. In such cases, the executed plan is the deliberative plan that was simultaneously invoked with the experience-based plan. From its use, a new experience-based plan is added to the knowledge base of the agent. This training-mode avoids the time delays introduced by competence layers such as in the Interrap model. However, after the agent has enough experiences, the deliberative plans need not to be simultaneously invoked. In further version of IDEAL agents, we shall try using a mechanism such that as the agent builds its experience-based plan library, it simultaneously builds a related deliberative plans index. Should an experience-based plan be invoked, only the related index would be called into the buffer as a reference. At the same time, the deliberative planing unit would either be suspended or permitted to end the planning action only when the experience-based plan unit ends its own planning (at least generate a candidate). The deliberative plan candidates would be put into the buffer instead of the intention structure and in this way, there would be no need to filter them again. If the executing experience-based plan fails then the

---

[28] Ferguson's Turing Machine does not support learning functionality.

references given by the index and the associated candidates present in the buffer are immediately used to complete the expected action.

7.3.4 Using fuzzy representation and fuzzy reasoning

We believe that the use of fuzzy knowledge representation and fuzzy reasoning would enhance the overall capacities of the IDEAL agent to deal with real situations which could not otherwise be dealt with. Introducing fuzzy mechanisms into IDEAL agents would certainly change and increase the complexity of the architecture of all layered levels. However, the diversification of the fields of application the IDEAL agents could cope with would, to say the least, easily equilibrate the possible draw-backs resulting from the increased complexity.

Furthermore, with the use of an integrated fuzzy approach, we can imagine using the IDEAL agents as nodes of a neural network which would possess intelligent flow control abilities paired with intelligent local solving power. What would be the efficiency of such a network and what type of problem could it address are certainly interesting subjects for future research efforts.

References

[Barber 99] K. S. Barber, C. E. Martin, "Agent Autonomy: Specification, Measurement, and Dynamic Adjustment." In Proceedings of the Autonomy Control Software Workshop at Autonomous Agents 1999 (Agents'99), 8-15. May 1, 1999. Seattle,
WA.

[Bernard 2000] J. C. Bernard, L. Esmahi, "Using dependency relations in the negotiation process of a Multiagent System", First International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR-2000), Jul-31-Aug06, L'Aquila, Italy.

[Bratman 87] M. E. Bratman, D. J Isael, and M. E. Pollack, "Toward an architecture for resource-bounded agents", Technical report CSLI-87-104, Center for the study of language and information, SRI and Standford University, August 1987.

[Brooks 86] R. A. Brooks, "A robust layered control system for a mobile robot", IEEE Journal of Robotics and Automation, Vol RA-32(1), 1986.

[Cohen 90] P. R. Cohen, H. J. Levesque, "Intention is choice with commitment", Artificial Intelligence, 42(3), 1990.

[Ferguson 92a] I. A. Ferguson, "TouringMachines: Autonomous Agents with Attitudes", IEEE Computer, 1992, 25(5), May 1992.

[Ferguson 92b] I. A. Ferguson, "TouringMachines: An Architecture for Dynamic, Rational, Mobil Agents", Ph.D. thesis, The university of Cambridge, October 1992.

[Gilbert 96] D. Gilbert, M. Aparicio, B. Atkinson, S. Brady, J. Ciccarino, B. Grosof, P. O'Connor, D. Osisek, S. Pritko, R. Spagna, and L. Wilson, "Intelligent Agent strategy White paper", IBM Corporation, Research Triangle Park, 1996.

[Georgeff 89] M. P. Georgeff and F. F. Ingrand, "Monitoring and control of spacecraft system
using procedural reasoning", proceeding of the space operations automation and robotics workshop, Houston, July 1989.

[Hume 90] Hume & Associates, "The Yen Yoke", 1990.

[Hoek 94] W. V. D. Hoek, B. V. Linder, and J. J. ch. Meyer, "a logic of capabilities", Proceeding of the thrid international symposium on the logical foundations of computer science, Lecture Notes in computer science LNCS 813, Springer Verlag, Heidelberg, Germany, 1994.

[Jennings 92] N. R. Jennings, "On being responsible", Decentralized A. I. 3 edited by Y. Demazeau and E. Werner, North Holland, Amsterdam, The Netherlands, 1992.

[Kinny 94] D. Kinny, M. Ljungberg, A. S. Rao, E. A. Sonenberg, G. Tidhar, and E. Werner, "Planed team activity", Artificial social systems, Lecture notes in artificial intelligence, Amsterdam, Netherlands, Springer Verlag, 1994.

[Esmahi 2000] L. Esmahi, "Protocoles de cooperation dans les systèmes multiagents : Une approche basée sur les relations de dépendances". Thèse de doctorat, École Polytechnique de Montréal, Octobre 2000.

[Lesperance 95] Y. Lesperance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R B. Scherl, "Foundations of a logical approach to agent programming",

Working notes of the IJCAI-95 workshop on Agent Theories, Architecture, and Lanuages", Montreal, Canada, 1995.

[Linder 95] B. van Linder, W. van der Hoek, and J. J. C. Meyer, "How to motivate your agents?", in working notes of the IJCAI95 Workshop on agent Theories, Architecture, and languages, Montreal, Canada, 1995.

[Maarten 2000] Maarten Sierhuis, W. J. Clancey, and R. V. Hoof, "BRAHMS, A multi-agent programming language for simulating work practice", Tech report, Research Institute for Advanced Computer Science, 2000.

[Mitchell 97] Tom M. Mitchell, "Machine Learning", McGraw-Hill Companies, Inc., 1997.

[Muller 95] Jorg P. Mulle, M. Pischel, and M. Thiel, "Modeling reactive behaviors in verticall layered agent architectures", Artificial intelligence: Theories, architectures, and languages, Lecture Notes in artificial intelligence LNAI 890, Heidelberg, Germany, 1995, Springer Verlag.

[Muller 97] Jorg P. Muller, "The design of Intelligent Agents: A Layered Approach", Lecture Notes in Artificial Intelligence 1177, Springer, 1997.

[Nwana 96] H. S. Nwana, "Software Agents: An Overview", Knowledge Engineering Review, Vol. 11, No 3, pp.1-40, Sept. 1996.

[Russell 95] Stuart J. Russell and Peter Norvig, "Artificial Intelligence: a modern Approach", Prentice-Hall Inc., 1995.

[Rao 91] A. S. Rao and M. P. Georgeff, "Modeling Rational Agents Within a BDI Architecture", Tech. Note 14, Australian Artificial Intelligent Institute, February 1991.

[Rao 92] Anand S. Rao and Michael P. Georgeff, "An abstract architecture for rational agents". In Proceedings of the third international conference on principles of knowledge representation and reasoning, KR'92, pages 439-449, Boston, MA, 1992.

[Rao 94] Anand S. Rao, "Multi-agent mental-state recognition and its application to air-combat modelling", Technical Note 50, Australian Artificial Intelligent Institute, June, 1994.

[Rao 95] Anand S. Rao and Michael P. Georgeff, "BDI Agents: from Theory to Practice", Tech. Report 56, Australian Artificial Intelligent Institute, April 1995.

[Rao 97a] Anand S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language", "agents breaking away", 1997.

[Rao 97b] Anand S. Rao and Klaus, Fischer, "BDI Agents Making Decisions in business and Industrial Practice", Anand S. Rao and Klaus Fischer, Tutorial report, in Workshop of Autonomous agent 97', May, 1997.

[Scott97] Scott Moss, Helen Gaylard, Steve Wallis and Bruce Edmonds, "SDML: A Multi-Agent Language for Organizational Modelling",Tech report, Centre for Policy Modelling, Manchester Metropolitan University, 1997.

[Sea 69] J. R. Searle, "Speech acts", Cambridge university press, 1969.

[Sycara 97] Katia Sycara, K. Decker, A.S. Pannu, M. Williamson, and D. Zeng, " Distributed Intelligent Agents", Proceeding of workshop in autonomous agents 97's, May 1997.

[Sheth 94] B. D. Sheth, "A Learning Approach to Personalized Information Filtering", Master thesis, MIT, 1994.

[Shoham 93] Y.shoham, "Agent-oriented programming", Artificial Intelligence, 60(1), 1993.

[Song 96] H. Song, Stan Franklin, and Aregahegn, "SUMPY: A Fuzzy Software Agent", Tech. Report, Institute for Intelligent Systems, The Univ. Of Memphis, 1996.

[Singh and Asher 90] M. Singh and N. Asher, "Towards a formal theory of intentions", Logics in AI edited by J. van Eijck, Springer Verlag, Amsterdam, Netherlands, 1990.

[Thomas 90] A. J. Thomas, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, "introduction to algorithms", The MIT press, 1990.

[Thomas 95] S. R. Thomas, "The PLACA agent programming language", Intelligent Agents: Theories, Architectures, and Languages, Lecture Notes in Artificial Intelligence LNAI 890, Amsterdam, Netherlands, 1995, Springer Verlag.

[Virginia 2002] Virginia Dignum et al.,"Agent Societies: towards frameworks-based design", Achmea & University Nyenrode, The Netherlands, 2002.

[Weerasooiya 95] D. Weerasooiya, A. Rao, and K. Rammamohanarao, "Design of a concurrent agent-oriented language", Intelligent Agent: Theories,

Architecture, and Languages", Lecture Notes in Artificial Intelligence LNAI 890, Amsterdam, Netherlands, 1995.

[Weiss 99] Gerhard Weiss, "Multiagent systems: a modern approach to distributed artificial intelligence", The MIT press, 1999.

[Wieke 2002] Wieke de Vries, Frank S. de Boer, Koen V. Hindriks, Wiebe van der Hoek, and John-Jules C. Meyer, "A Programming Language for Coordinating Group Actions", B. Dunin-Keplicz and E. Nawarecki (Eds.): CEEMAS 2001, LNAI 2296, pp. 313–321, Springer-Verlag, Berlin, Heidelberg, 2002.

[William 2001] William J. Clancey, P. Sachs, M. Sierhuis, and R. van Hoof, "BRAHMS: SIMULATING PRACTICE FOR WORK SYSTEMS DESIGN", Tech report, NASA/Ames Research Center, 2001.

[Wooldridge 92] Michael J. Wooldridge, "The logical modeling of computational multi-agent system, Ph.D. thesis, department of computation, the faculty of technology, University of Manchester, August 1992.

[Wooldridge 94] Michael J. Wooldridge, and Michael Fisher, "A decision procedure for a temporal belief logic", Proceeding of the first international conference on temporal logic, Bonn, Germany, 1994

[Wooldridge 95a] Michael J. Wooldridge, and N. R. Jennings, editor, "Intelligent Agent: Agent Theories, Architectures, and Languages", Springer-Verlag, 1995.

[Wooldridge 95b] Michael Wooldridge, and Nicholas R. Jennings, "Intelligent Agents: Theory and Practice", knowledge Engineering review, Feb. 1995.

[Wooldridge 96] Michael J. Wooldridge, J. P. Muller, M. Tambe , editor, "Intelligent Agent (II): Agent Theories, Architectures, and Languages ", Springer-Verlag, 1996.

[Wooldridge 97] J. P. Mulle, M. J. Wooldridge, and N. R. Jennings, editor, "Intelligent Agent (III): Agent Theories, Architectures, and Languages ", Springer-Verlag, 1997

Appendix A

The ***march-step*** algorithm

The algorithm used to control the execution of patterns: **march-step ()**

. Pseudo-code for the ***march-step*** algorithm:

```
March-step(node list, edge_list, node_number, pattern_type)

 5:  for i = 1 to node_number ;
10:   next[i] = nodelist[i].out ;
          //Assign the address of the left-hand first edge
          //starting from node i.
12:  end for
15: i =1 ;
20: nodelist[i].mark = D;
      //Colored root
25: notfinished = T ;
30: while notfinished == T do
35:    if next[i] == nil then
40:      notfinished = F ;
45:      return F ;
      //"The pattern fails"
50:   end if
55:   p = next[i]^.no ;
      //An edge's end → p
60:   if  (next[i]^.mark = N) then
      // Check if the edge has not been passed
65:      pass = F ;
      //Initiate variable pass
70:      pass = execute next[i]^.action ;
      // Try to accomplish action and return a Boolean value
      // "execute" is a method. It checks the maintaining
```

```
      // conditions and then runs the function corresponding
      // to the action name when preconditions
      // of the action is true
      // Otherwise, it returns a fail value.
75:     if pass == T then
      // Action success

80:      case(nodelist[i].att == "or" & next[p] ≠ nil) or
         (nodelist[i].att == "and" & next[i].link == nil &
            next[p] ≠ nil) do
       //Edge's starting point is "or" node and final is
       //not a leave, or the starting point is "and" and
       //the edge is the last edge and end point is not a
       //leave, then control goes on in depth-first
       //direction.
85:    if (nodelist[i].cyc=="cyc" or next[i]^.cyc == "cyc") then
90:               ;
      // Do not mark an edge in a cycle.
95:            else
100:              next[i]^.mark = P ;
105:              next[i] = next[i]^.link ;
110:   end if
115:   nodelist[p].mark = D ;
120:            i = p ;
125:            break ;
130:   case (nodelist[i].att == "or" & nodelist[p].out == nil)
           or (nodelist[i].att == "and" & next[i].link == nil
                &(nodelist[p].out == nil) do
135:            notfinished = F ;
140:            return (T and post-condition of the pattern) ;
             // "The pattern gets success". The end of the edge
             //is a leave node.
145:   case(nodelist[i].att == "and" & next[i].link ≠ nil
146:   do
150:   if (nodelist[i].cyc=="cyc" or next[i]^.cyc == "cyc") then
155:              ;
```

```
160:           else
165:              next[i]^.mark = P ;
170:              next[i] = next[i]^.link ;
175:           end if
180:           nodelist[p].mark = D ;
185:           break ;
            //An edge is "successful passed", but search stays
            //in the same level: starting point is
            //an "and" node and end is not a leave node.
190:        end case
195:     else
         // Cases: pass == F, i.e.,
         //"an edge fails to be passed."
200:        case (nodelist[i].att == "or" & next[i].link ≠ nil)
202:        do
205:        if (nodelist[i].cyc == "cyc" or next[i]^.cyc ==
            "cyc")  then
210            ;
215:           else
220:              nodelist[p].mark = D ;
225:              next[i].^mark = P ;
230:           endif
235:           next[i] = next[i]^.link ;
240:           break ;
245:        case (nodelist[i].att == "or" & next[i] == nil) or
250:              (nodelist[i].att == "and") do
255:           next[i] = nil ;
260:           nodelist[i] = D ;
265:           next[i]^.mark = P ;
270:           notfinished = F ;
275:           return F ;
               //"The pattern fails"
280:        end case
285:     end if
290:  else
295:     next[i] = next[i]^.link ;
```

```
300:  end if
305:end do
```

The comments for the ***march-step*** algorithm:

The **march-step** function's input is the adjacency-lists that represent the body of the pattern that will be executed, and its output is a Boolean value that indicates whether or not the pattern has successfully been completed.

Appendix B

The ***next-step*** algorithm

The algorithm used to interpret and control the execution of plans: **next-step ()**

Assumption: *if the precondition of an action is satisfied, then the action will always successfully be executed.*

Deduction: *from the above assumption, we can see that an action will never be committed if its precondition is not verified.*

Pseudo-code for the ***next-step*** algorithm:

```
Next-step(input: node_list, edge_list, node_number,
                 control_point, next, s_f_sign ;
          output: da, next, control_point)
// s_f_sign := natual integer;
//    =0 : initiation
//    =1 : action success
//    =2 : action fail
//    =3 : desire success
//    =4 : desire fail.
// da := True| False| desire | action.
//
 5:  if s_f_sign = 0 then
10:    for i = 1 to node_number ;
15:         next[i] = nodelist[i].out ;
20:    end for
25:     i = 1 ;
30:  else
35:     i = control_point ;
40:  end if
45:  notfinished = T ;
50:  while (notfinished == T) do
```

```
55:     if next[i] == nil then
65:       CASE
66:    1: (s_f_sign = 0) do
70:          da = False ;
75:          notfinished = F ;
80:          return (da, _, _) ;
85:    2: (s_f_sign = 1) do
90:          da = True ;
95:          notfinished = F ;
100:         return (da, _, _) ;
105:    end CASE
110:   end if
115:   if (s_f_sign = 3) then
117:     nodelist[i].mark = D ;
118:     p = next[i]^.no ;
120:     CASE
121:    1: (nodelist[i].att == "or" & next[p] ≠ nil)or
           (nodelist[i].att == "and" & next[i].link == nil &
            next[p] ≠ nil) do
125:       if (nodelist[i].cyc=="cyc" or
                 next[i]^.cyc == "cyc") then
130:             ;
135:       else
140:          next[i]^.mark = P ;
145:          next[i] = next[i]^.link ;
150:       end if
155:       nodelist[p].mark = D ;
160:       i = p ;
165:       //end case 1 ;
170:    2: (nodelist[i].att == "or" &
              nodelist[p].out == nil) or
             (nodelist[i].att == "and" &
              next[i].link == nil &(nodelist[p].out == nil)
              do
175:         next[i] = nil ;
180:         da = True ;
```

```
185:        return(da, _, _) ;  //end of case 2
190:     3: (nodelist[i].att == "and" & next[i].link ≠ nil) do
195:        if (nodelist[i].cyc=="cyc" or
                    next[i]^.cyc == "cyc") then
200:              ;
205:        else
210:          next[i]^.mark = P ;
215:          next[i] = next[i]^.link ;
220:       end if
225:         nodelist[p].mark = D ;
230:        // end of case 3 ;
235:       end CASE
236:    endif // end of if (s_f_sign = 3)
240:    if (s_f_sign = 4) then
242:      nodelist[i].mark = D ;
243:      p = next[i]^.no ;
244:     CASE
245:       1:(nodelist[i].att == "or" & next[i].link ≠ nil) do
250:         if (nodelist[i].cyc == "cyc" or
                    next[i]^.cyc == "cyc") then
255:           ;
260:         else
265:           nodelist[p].mark = D ;
270:           next[i].^mark = P ;
280:         endif
285:         next[i] = next[i]^.link ;
290:        // end of case 1 ;
295:      2: (nodelist[i].att == "or" & next[i] == nil) or
               (nodelist[i].att == "and") do
305:           next[i] = nil ;
310:           nodelist[i] = D ;
315:           next[i]^.mark = P ;
320:           da = False
325:           return(da, _, _) ; // end of case 2
330:      end CASE
335:   endif //end of if (s_f_sign = 4)
```

```
340:      nodelist[i].mark = D;
350:      p = next[i]^.no ;
355:      if  (next[i]^.mark = N) then
360:        if attribut (next[i]^.da) = "desire" then
            // attribute is a function that test attribute of
            // next[i]^.da, and return value: "desire" or
            // "action".
365:          da = next[i]^.da ;
370:          return (da, next, i) ;
375:       else // attribut (next[i]^.da) = "action"
380:          pass = F ;
385:          pass = test next[i]^.da ;
            // "test" is a method.It checks the precondition
            // of the da, and returns a Boolean value.
            // If the condition is true, then it returns T,
            // otherwise, it returns F.
388:          da = next[i]^.da ;
390:    if pass == T then
395:      CASE
397:     1: (nodelist[i].att == "or" & next[p] ≠ nil)
             or(nodelist[i].att == "and" &
           next[i].link == nil & next[p] ≠ nil) do
400:       if (nodelist[i].cyc=="cyc") or
             (next[i]^.cyc == "cyc") then
405:         ;
410:       else
415:          next[i]^.mark = P ;
425:          next[i] = next[i]^.link ;
430:       endif
435:       nodelist[p].mark = D ;
440:       i = p ;
445:       return(da, next, i) ; //end of case 1:
450:     2:(nodelist[i].att == "or" & nodelist[p].out == nil)
            or (nodelist[i].att == "and" &
            next[i].link == nil &(nodelist[p].out == nil)
            do
```

```
455:            next[i] = nil ;
460:           return(da, next, i) ; //end of case 2:
465:     3:(nodelist[i].att == "and" & next[i].link ≠ nil) do
470:        if (nodelist[i].cyc=="cyc") or
                (next[i]^.cyc == "cyc") then
475:             ;
480:             else
485:              next[i]^.mark = P ;
490:              next[i] = next[i]^.link ;
495:           endif
500:           nodelist[p].mark = D ;
505:           return(da, next, i) ; //end of case 3:
510:     end CASE
515:    else // of if pass == T
  // Cases for pass == F, i.e., "an edge fails to be passed."
517:   CASE
520:      1:(nodelist[i].att == "or" & next[i].link ≠ nil) do
525:         if (nodelist[i].cyc == "cyc") or
             (next[i]^.cyc == "cyc") then
530            ;
535:          else
540:           nodelist[p].mark = D ;
545:           next[i].^mark = P ;
550:         endif
555:         next[i] = next[i]^.link ;
560:   // end of case 1
565:     2:(nodelist[i].att == "or" & next[i] == nil)
               or (nodelist[i].att == "and")
             do
575:           next[i] = nil ;
580:           nodelist[i] = D ;
585:           next[i]^.mark = P ;
590:           da = False ;
600:           return(da, _, _) ; //end of case 2
610:      end CASE
615:     endif //of pass == T
```

```
620:    endif // attribut (next[i]^.da) = "desire"
630:    else // of if (next[i]^.mark = N)
635:      next[i] = next[i]^.link ;
640:    endif
670: end // of while … do
```

The comments for the ***next-step*** algorithm:

From the input node_list and edge_list, we construct the adjacency-lists of the flow-away edges of a CRDAO graph. The graph is the body of the plan being executed. The variable control_point is a node number used as the starting point of the search. The list next is called the **critical edges list**. It is used to save **the critical edges**, namely, those among all flow-away edges of each node that will be processed. As the plan is executed, the content of variable next will be changed. The variable s_f_sign is used to express the success or failure of the last committed edge **da**. We proceed in this way because the function only commits the edge **da** to execution, the execution itself of **da** is external to the function. Therefore, the acting executor (actor) needs to return a success or failure information. This information is important to the march-step algorithm since it has to choose different paths according to the value of s_f_sign in order to satisfy the restriction relations of the edges.

The domain of s_f_sign is [0, 4]. Respectively, value 0 means the beginning of the execution of a new plan, value 1 denotes an action's success, value 2 denotes an action's failure, value 3 denotes a desire's success and value 4 denotes a desire's failure. Note that given the assumption we made to the fact that a committed atomic action will always be successful, s_f_sign = 2 will never happen.

Since a desire is an "end" the agent intends to achieve, then executing the desire implies that the agent tries to find a “means” for the “end”, namely a sub-plan, and tries to complete the sub-plan. We denote as True the fact that a sub-plan triggered by a desire has been successfully completed. Similarly, we denote as False the case where the triggered sub-plan has failed. Consequently, the

possible values of the output of the algorithm which normally should be a desire and or an atomic action can also be either True or False.

Appendix C

The definitions and basic semantics of the operation functions of an IDEAL agent

The definitions and basic semantics of the operation functions of an IDEAL agent are detailed below.

1. The recognition function

The **recognition** function describes how perceptions are transformed into standard triggering events. Let ***p*** = (sender, receiver, content) be a perception, then we have the recognition function defined as

Formula 1

*recognition(**p**) := (eq, i)* • (eq $\in$ PT, (eq, i) $\in$ content),

where, *i* is a true intention, and *eq* is a triggering event. The notation • (...) is used to represent some restricting conditions. We say that the *recognition* function with ***p*** as an argument identifies a specific event *(eq, i)* and puts it in the *event queue*.

No special world interface or data transformation functionality are introduced since we suppose that the information transmission in IDEAL agents obeys the definition 3.1 and the definition 3.4. Note that for environments with other specificity of messages treatment, we need only add a corresponding world interface and modify the recognition function so it can make the correct connections between perceptions and standard triggering events.

2. The belief generation and revision function

The *belief generation and revision* function accomplishes two tasks. It adds new beliefs into the current belief database and also revises all beliefs so as to keep their consistency.

If we let B_{new}, B_{old}, B_{update}, B_{com}, B_{non} respectively represent new beliefs, beliefs before updating, updated beliefs, compatible beliefs and non-compatible beliefs, then the function *belief generation and revision* can be seen as the mapping

$$B_{new} \times B_{old} \rightarrow B_{update}\ , \quad \text{where } B_{new} = B_{com} \cup B_{non}\ .$$

If we let *b* be a belief, the function is defined as

Formula 2.1

$$\textit{belief generation and revision } (B_{old}, B_{new})$$
$$:= B_{update} \bullet (\forall\ b \in B_{update}, \textit{ such that } b \in (B_{old} \cup B_{com}) \ \&\ (b \notin B_{non}))\ .$$

Note that the function *belief generation and revision* does not remove duplicate beliefs that may be present in the event queue. This specific task is implemented in the *event_filter* function defined hereafter.

Let E_{old}, E_{new}, represent the event set before updating, the event set after updating, respectively. We have

Formula 2.2

$$\textit{event\_filter}(E_{old}) := E_{new} = \{ e_i \} \bullet (\forall e_i \in E_{new}, \textit{ if } i \neq j, \textit{ then } e_i \neq e_j)$$

<u>*3. The goal activation function*</u>

The role of the *goal activation* function is to connect events with plans and patterns. It does so by matching triggering events in the *event queue* with invocation conditions of plans and patterns. Finding the appropriate plans or patterns permits the achievement of the the goals implied by the events.

The *goal activation* function is defined as

Formula 3

$$\textit{goal activation: } E \times Bel \times PPL \rightarrow Goals\ ,$$

where, *PPL* = dpl ∪ epl ∪ ipl ∪ cpl ∪ mpl, is the union of all the plans and patterns libraries. *Goals* is defined as the union of $G_d \cup G_{ep} \cup G_i \cup G_c \cup G_m$, and *E* is the set of events present in the *event queue*. *Bel* is the set of current beliefs.

As mentioned before, the five control layers of the IDEAL agent can only access their corresponding part of the plan library. Evidently, we want each different type of plans to generate corresponding different type of goals. We thus have, for each of the control layers, different *goal activation* functions. For any given event present in the event queue –remember that the events are simultaneously accessed by each layer-, if some layer's *goal activation function* invokes a plan or a pattern from its plans or patterns library, then the event is linked to the plan or the pattern. This means that the agent intends to take some actions in response to the event. In other words, a goal is activated.

The execution of a deliberative plan often includes means-ends reasoning procedures that decomposes a complex problem into several easier-to solve sub-problems. It follows that deliberative goals can cover complex contents and are thus suited for long-term complex tasks.

From the intuitive, cooperative and meta-reasoning layers activated goals correspond to reactive context-free goals. The context-free nature of the reactive goals makes them very efficient. However, since the actions related to reactive goals are almost hard-wired, they lack flexibility since they do not have access to detailed decision-making processes. The expressive range of reactive goals is thus quite limited which makes them suited only short-term simple tasks.

As for cooperative goals which are generally defined as sets of sub-goals shared by different agents, we could think that the inherent complexity of cooperative plans would necessitate corresponding complex cooperative goals. However, there are two reasons that makes this totally false in the IDEAL agent. First, each cooperating agent in the system only tries to achieve its part of the goal, i.e., its specific sub-goal. Second, although cooperative operations are

implemented through the cooperative layer of the agents, overall cooperation is implemented by means of hierarchical broadcast structures consisting of supervisor agents and worker agents. The supervisor decomposes the global cooperative goals into sub-goals that are given to the worker agents. It comes out from this approach, that the only effective goals the workers receive are strictly reactive patterns.

Although goals linked with the experience-based layer are context-sensitive, the procedure mapping the goals into intentions is hard-wired. It thus short-circuits the means-ends reasoning processes making the processing of experience-based goals very efficient. Having both advantages of reactive and deliberative styles, experience-based goals are used as alternatives to deliberative goals when dealing with complex problems that have been met before.

There three possible situations that are exceptions to the *goal activation* function. For those situations the function will either activate no goal or activate special goals. First, for the events identified as "external non-implementable desires" and "beliefs updating without invocation" no goal is activate in any of the control layers. The *goals activation* functions drops them away. Second, when quasi-triggering events, such as ***succ_plan***[29], ***fail_desire***, etc., are encountered, they are dropped away by IL[30], CL, EL and ML. Only the DL *goals activation* function can deal with them by generating goals of a special kind, called pseudo-goals, that are not related to any plan. Finally, for events qualified as "internal non-implementable desires" the function proceeds as in the second case.

4. The deliberation function

The *deliberation* function describes how options for a given event are extracted from the candidate goals. Therefore, some sort of evaluation functionality has to be integrated in this function. In fact, the function is a mapping defined by:

Formula 4

---

[29] These events will be discussed later.

deliberation: Goal $\times$ *Bel* $\rightarrow$ *Option*

where *Goal* is the set resulting from the union of the goals sets G_d, G_{ep}, G_i, G_c and G_m, *Option* is similarly defined for the options sets O_d, O_{ep}, O_i, O_c and O_m, and *Bel* is the set of current beliefs. The *deliberation* functions in the five control layers deal with goals in their individual goal data structure, in parallel mode, using the weights of the goals to establish the possible options.

Since an event can possibly generate more than one options, to choose the best option the agent has to use some method to make its choices. The methods are different for each layer. For example, since in the deliberative layer we expect an optimistic solution meta reasoning will be started. This is not implented in the deliberation function however. Instead, the *deliberation* function in DL does not limit the number of options that can be generated leaving the problem of chosing the best one to the *unwind* function.

In EL, IL, CL and ML, time efficiency is required, hence the choice is done randomly. What it really means is that when several plans or patterns have the same weight value, the agent randomly selects one as the option. Thus, in these four layers, the *deliberation* function always generates a single option for any given event.

5. The unwind function

The *unwind* function maps the agent's options into the intentions space[31]. It is defined as :

Formula 5

Unwind: Option $\times$ *Bel* $\rightarrow$ *I*

where *Option* and *Bel* are defined as above and *I* corresponds to the set of intentions defined by $I_d \cup I_{ep} \cup I_i \cup I_c \cup I_m$.

---

[30] DL, EL, IL, CL and ML represent deliberative layer, experienced-based layer, intuitive layer, cooperative layer and meta-operation layer, respectively.

[31] In some agent models, this function is called planing.

As mentioned before, one of the tasks of the *unwind* function is to deal with quasi-triggering events (NT events). Though NT events do not directly activate plans or patterns, they may cause changes of the agent's internal states. These NT events strictly activate special deliberative goals which are mapped into intentions on a case-by-case basis. Once these intentions executed, the related mental states of the agent are updated. In the deliberative layer, options are interpreted into new intention stacks or intention frames[32] according to the types of events that activated the goals. The control pointers of the bodies are set to point to the actions to be executed by these intentions.

In the other four layers, the role of the *unwind* function is quite simple. It first creates a new intention stack where it puts a "true intention". Then, it pushes on top of this "true intention", the triggering event that invoked the goal and the pattern containing the actions needed to achieve the goal. Finally, it sets the control pointer to the actions that will be executed in the pattern's body. Intentions that are just modified are inserted into the intention structure **I** to wait for sorting by the *schedule* function.

6. The schedule function

The *schedule* function acts as a multi-way selection switch. First, it sorts all intentions in the intention structure according to their priorities. It then selects the intention with the maximum priority as its output. An intention's priority is either 1, 4, 7 or 10 depending if its type is deliberative or experience-based, meta-reasoning, cooperative or intuitive. The intermediate values between two neighbor types are used to refine the priority of intentions of identical types. For instance, when the *execution* function commits to execution a desire that is not an atomic action, the priority of the intention is add 1 and is pushed in the event queue. In the next cycle, this intention will prior those of the same that are to be

---

[32] See the definition 3.9 and the definition 3.11.

executed. With this approach we hope to finish the internal means-ends reasoning process as soon as possible.

When there are several intentions with the same maximum priority, two solutions are possible: either use meta reasoning or choose randomly. It is our belief that the *schedule* function is the bottleneck of the agent's operations since the outputs of the five layers are synchronized here. Since efficiency is important, we randomly choose one among the results as output of *schedule* function. It is quite obvious that the *schedule* function corresponds to the mapping

schedule: $\mathbf{I} \rightarrow i$.

where **I** is the intention structure and i the selected intention.

In a more formal notation we have:

Formula 6

$$sch(I, Bel) := (i \,||\, i = random_{prio}\{max_{prio}(I')\} \,\&\, priority(i)>0,\ I' \subseteq I,\ \forall i' \in I',$$
$$priority(i'_i) \geq priority(i'_j) \text{ when } (i \neq j) \,\&\, (i>j>0)\ ) \bullet (number(I) \geq 1,$$
$$I' = sorting_{prio}(I)).$$

Note that $sorting_{prio}(\mathbf{I})$ is a function that sorts all elements of a set ***I*** in an ascending order of a given prioritizing property of the elements while $random_{prio}\{\mathbf{A}\}$ is a function which returns one single randomly selected intention from the set of intentions ***A*** used as argument for the function. The function $max_{prio}()$ returns the set of all intentions having the same highest priority for the intention structure used as argument. Function *priority(**Intend**)* returns the priority of the specific intention ***Intend*** and *number(**S**)* is the function that returns the number of elements of the argument set ***S***.

Before calling the *schedule* function, one last thing must be done. As mentioned above, an experience-based plan can be used as an alternative to a deliberative plan when a triggering event simultaneously activates plans of both types.

Consequently, when this situation appears, the intention linked to the deliberative plan is deleted and the intention with the experience-based plan kept. To do so we use the *I_filter* function as defined hereafter.

7. The function I_filter

The function is defined as

Formula 7.1

$$I\_filter:\ I_{redun} \times Bel \rightarrow I_{non\text{-}redun}.$$

where, I_{redun} represents the intention structure before using the function I_filter() while $I_{non\text{-}redun}$ represents it after using the function.

To better understand how it works, let us introduce the following notation. Let Γ be a binary relation such that $\Gamma \subseteq I_{dp} \times I_{ep}$, $\forall \gamma = (i_{dp}, i_{ep}) \in \Gamma$, $i_{dp} \in I_{dp}$, $i_{ep} \in I_{ep}$. The pair γ is constituted of the deliberative intention i_{dp} and the experience-based intention i_{ep} simultaneously generated by a single triggering event. Then we write that

Formula 7.2

$$I\_filter(I, Bel) := I' \bullet (\forall i_{dp} \in I, \text{ if } \neg\exists\, i_{ep} \in I, \text{ so that } (i_{dp}, i_{ep}) \in \Gamma, \text{ then } i_{dp} \in I'.$$
$$\text{Otherwise, } i_{dp} \notin I'.\ \forall i \notin I_{dp},\ i \in I \rightarrow i \in I')$$

I' is thus the intention structure without redundant i_{dp}. Note that we have represented the sets of i_{dp} and i_{ep} respectively by I_{dp} and I_{ep}.

8. The execution function

The *execution* function commits the selected intention to execution. In fact, it commits to execution the current element of the temporal ordered sequence of elements scheduled in the intention. We must recall that these elements can be actions, desires or procedures and that the algorithm used for the scheduling is a depth-first algorithm (***next-step()***).

If the intention committed is a stack consisting of deliberative plans related to current context, the action or desire that will be determined by the depth-first algorithm comes from the body of the plan at the top of the intention. This action

or desire is the one committed by the *execution* function. As the *schedule* function, the *execution* function is used for the five different layers. Given the non-homogeneity of the possible intention structures, the complexity of the *execution* function is quite large. For the different types of intentions, what is committed by the *execution* function is different.

For a deliberative intention, there are two different situations. When what committed is a desire (sub-goal), the function assembles the desire and the intention the desire belongs to, into a pair, and puts the pair into the agent's event queue. It results that an IT event is generated. When an action is commited - sending messages is a type of actions - then the action is committed to the acting executor (**actor**). When the action is executed, it is monitered and a result of either success or failure is returned. The result and the intention the action belongs to, together as a pair, are added into the agent's event queue. A NT event is generated and changes to the agent's internal states will result.

If the agent "executes" an experience-based intention, all is similar to what happened in the second situation in a deliberative intention, since an experience-based plan is the special case of a deliberative plan.

In the case of a reactive intention (intuitive, cooperative or meta-reasoning), two cases are also possible. When the intention committed is a non-interruptible pattern, the acting executor recognizes it as non-interruptible, executes it immediately and the resulting pair is sent to the event queue. In the case of an interruptible pattern, the acting executor also recognizes it as such and calls the function ***march-step***. An atomic action is executed under the control and monitoring of the function *march-step*. It can easily be seen that executing an interruptible pattern by means of the function *march-step* is very similar to what happens in the second situation of the execution of a deliberative intention.

The intentions for updating internal states and processing success and failure events are also committed by the *execution* function. These intentions are treated as deliberative intentions. The only difference is that their bodies

correspond to special events. Like desires, a pair is formed and put into the event queue. Then it results in the expected updating and processing being done.

Finally, we can write the *execution* function as

Formula 8

$$execution:\ i \rightarrow (eq, i) \bullet (eq \in NT)\ \ or$$

$$execution:\ i \rightarrow (eq, i) \bullet (eq \in IT)\ .$$

9. The learning and modifying function

The *learning and modifying* function involves two aspects. First, it generates the fast reactive operational procedures, the so-called experience based plans, enabling the agent to efficiently deal with complex situations that occur frequently. On the other hand, it modifies the credits of the related plans in a way that should improve the future performance of the agent. The *learning and modifying* function works in off-line mode, hence the execution of the function itself does not augment the time costs of reacting to perceptions.

To establish the expected functionality, two types of learning mechanisms are used in the module. The first one belongs to the rote learning type. The idea is to scan the history database seeking for paths taken by successfully completed deliberative plans. For each such identified path, the function will assemble it into a new experience-based plan and add it into experience-based plans library. The procedure itself has already been described in the "history database" part presented above.

It is important to note thet the concept of experience-based plans is one of the highlight spots of IDEAL agents. It bridges modern dynamic planing methods with classic planing methods. The experience-based plans provide intuitive patterns' respond speed but are generated by dynamic deliberative planing. As mentioned before, this learning method is inspired by examples in our own lives. When you meet some complex thing for the first time, you possibly need to plan and think about it. If the experience was successful it is naturally stored in your brain.

When you meet the same complex situation you do not re-plan nor analyse it again in all details. You repeat your reaction as if it has an intuitive flavor once you have made sure that the environment conditions are same!

The second learning mechanism is based on feedback. The *learning and modifying* function revises the credit values of plans stored in the history database according to the results of their executions. The idea is partly inspired from the foraging behavior of ants.

When a plan is formed and is stored, it is assigned a credit value of 2. This indicates that this plan is a potential path toward a goal. When the plan is executed again, the *learning and modifying* function evaluates it in the following manner. First, a value of 1 is subtracted from its credit value and then the plan is re-evaluated using information stored in history database. If the plan was successful, then its credit value is added by 2. Otherwise, the credit updating ends. Thus, through using credits in such a way to weigh plans, it gives the weightiest plan the biggest probability of achieving the goal successfully. In our opinion, credits also describe the performance of the plans. In a way a plan reflects the prediction made by the agent's designer about future changes of the environment for given contexts. The agent's designer believes that in given context, the execution of the body of the plan will rationally be a response to some change of the environment, correctly achieve the corresponding goal and provide the best performance. Hence, if the execution of a plan is a success, it states that 1) the designer's prediction about change of the environment was correct, 2) the intended goal has been achieved and 3) the best performance was obtained. Therefore, the weightiest plan not only has the biggest probability for achieving its associated goal, but also has the biggest probability of obtaining the best performance. Remembering that credit values are used in the *deliberation* function, it follows that by using this second learning method, IDEAL agent's behaviors are not only kept rational but also made optimal. The *learning and modifying* function is presented below on a case by case basis.

Denoting ***dp_update*** the credit updating function for deliberative plans, it corresponds to the mapping

$dp\_update: HD \times DP_{old} \rightarrow DP_{new}$

where, HD and DP respectively denote the history database and the deliberative plan library. The subscripts are self explaining.

More formally, we have

Formula 9.1

$dp\_update(his\_d(e), dp_{old}) := (dp_{new} \,||\, dp_{new} = (add\_credit(head_{old}, \delta(e\_mark)) \rightarrow body_{old})) \bullet (\delta() = 2 \text{ when } e\_mark = S, \delta() = 0 \text{ when } e\_mark = F. ty = deliberative);$

$\delta()$ is a two-values function that returns 2 when its argument is S, returns 0 when its argument is F. ***add_credit***(*head*, *Y*) is the function that adds the value *Y* to the credit part of the *head*.

Similarly, we have the credit updating functions for intuitive, cooperative and experience-based plans, we omit them.

Let us now consider the situation when a new experience-based plan is created. For such cases, let ***ep_ge***() be the experience-based plan generating function and ***ep_com_add***() be the experience-based plan appending function. We see that ***ep_ge***() is a mapping given corresponding to

$ep\_ge: HD \times DP \rightarrow EP_{new}$

where, HD and DP respectively denote the history database and the deliberative plan library. EP_{new} represents the set of newly created experience-based plans. Using ***head_plan***(*plan name, plan library name*), a function which returns the head of the plan identified by *plan name* and *plan library name*, then $head_d$ = ***head_plan***(*pp, dpl*) represents the head of the *deliberative* plan *pp*. Writing the history data record for event *e* = *(eq, i)* as

$his\_d(e) = v(e; pp; ty; p\_a_1, \ldots, p\_a_n; e\_mark)$

and letting ep_{eq} represent an experience-based plan candidate for the triggering event *eq*, then we have

Formula 9.2

$ep\_ge(his\_d(e), dpl) := (ep_{eq} \,||\, ep_{eq} = c(head_d :\rightarrow action_1, ..., action_n)) \bullet$
$(c = pp, head_d = head\_plan(pp, dpl), action_1 = p\_a_1, \ldots, action_n = p\_a_n,$
$e\_mark = S, ty = deliberative)$;

Proceeding similarly for the experience-based plan appending function, we see that ***ep_com_add***() is a mapping given by

$ep\_com\_add: EPL \times EP \rightarrow EPL_{new}$.

where, EPL and EPL_{new} denote the experience-based plan library before and after adding the set of new experience-based plans to EP_{new}.

Let ***differentiate_merge***(*Lib*, *pl*) be a multi-valued function that appends plan ***pl*** to library ***Lib*** when ***pl*** is not already present in ***Lib***, and otherwise merges different parts of ***pl*** to the related places of the plan ***pl*** already present in ***Lib***. Then we can write

Formula 9.3

$ep\_com\_add(EPL, ep) := (EPL_{new} \,||\, EPL_{new} = EPL \cup differente\_merge(EPL, ep))$.