Conception et réalisation d'un environnement de programmation d'applications en Intelligence Artificielle Distribuée

Jacques Gamon

CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS
CENTRE REGIONAL ASSOCIE DE
GRENOBLE (CUEFA)

MÉMOIRE

présenté en vue d'obtenir

le DIPLOME D'INGÉNIEUR C.N.A.M.

en

INFORMATIQUE

par

Jacques GAMON

Conception et réalisation d'un environnement de programmation d'applications en Intelligence Artificielle Distribuée

Les travaux relatifs au présent mémoire ont été effectués dans l'équipe SIC du laboratoire TIMC/IMAG
sous la direction de Madame Sylvie PESTY

Remerciements

Je tiens à remercier,

Monsieur Claude KAISER, professeur au Conservatoire National des Arts et Métiers, de l'honneur qu'il me fait en présidant le jury de ce mémoire.

Madame Sylvie PESTY, maître de conférences à l'Université Pierre Mendès France de Grenoble, responsable de ce stage, pour ses conseils judicieux, et surtout pour son soutien moral.

Monsieur Jacques COURTIN, professeur à l'Université Pierre Mendès France de Grenoble, pour avoir jugé ce mémoire et comme membre du jury.

Monsieur Alain CAZES, maître de conférences au Conservatoire National des Arts et Métiers, en tant que membre du jury.

Monsieur Didier ADELH, directeur de projets à la société Alcatel TITN, pour ses conseils et comme membre du jury.

Madame Catherine GARBAY, directeur de recherche CNRS, pour m'avoir accepté dans son équipe.

Je tiens également à remercier mes collègues de l'équipe SIC, pour leur sympathie, Olivier pour ses compétences techniques et Emmanuelle, Adel, René-Pierre, Emmanuel, Marc, Arturo.

Merci également à tous les membres du laboratoire TIMC que j'ai côtoyés durant cette année.

Merci à mes camarades de travail qui m'ont permis de garder le contact avec l'entreprise, et plus particulièrement à Ilda pour ses conseils avisés sur Word.

Enfin merci à ma famille, Flora et Chantal, pour avoir supporté mes sautes d'humeurs.

A Flora et Chantal

Résumé

Ce mémoire traite de la conception et de la réalisation d'un environnement de programmation en Intelligence Artificielle Distribuée. Le but est de concevoir un système permettant la réalisation de logiciels pour la résolution de problèmes par coopération d'agents. L'approche envisagée est étudiée conjointement avec les avantages procurés par le Génie Logiciel, dans une étude incluant l'évaluation de systèmes voisins de celui proposé.

La réalisation effectuée de façon modulaire sur des bases de qualité logiciel, propose un ensemble d'outils essentiels tels qu'une Librairie de classes génériques C++, un Langage d'agent, un Précompilateur et une Interface graphique. Ces outils font partie d'un environnement et sont utilisés dans une méthodologie de conception d'une application Multi-Agents.

Ce travail sert de base pour envisager des évolutions possibles du concept Multi-Agents utilisé, les perspectives étant évoquées tant sur le plan conceptuel que technique.

Mots-Clés : Intelligence Artificielle Distribuée, Systèmes Multi-Agents, Langage d'Agent, Résolution de problème, Génie Logiciel, Environnement de programmation, Méthodologie.

Keywords : Distributed Artificial Intelligence, Multi-Agent System, Agent Language, Problem Solving, Software Engineering, Programming Environment, Methodology.

Table des matières

Liste des figures

Introduction

Posons d'emblée quelques mots-clés pour jalonner le cadre de ce mémoire, ce sont "Intelligence Artificielle Distribuée" et "Systèmes Multi-Agents", "Conception de logiciel" et "Résolution de problème", "Méthodologie" et "Génie Logiciel".

Le travail exposé dans ce mémoire s'est déroulé au sein de l'équipe SIC, du laboratoire TIMC/IMAG. Le laboratoire TIMC (Techniques de l'Imagerie, de la Modélisation et de la Cognition) fait partie de l'institut IMAG (Informatique et Mathématique Appliquée de Grenoble). Il est associé au CNRS (Centre National de la Recherche Scientifique) et dépend également de l'INPG (Institut National Polytechnique de Grenoble) et de l'Université Joseph Fourier. L'insertion prochaine du laboratoire TIMC dans l'Institut Albert Bonniot, nouveau pôle fédérateur Médecine-Informatique-Biologie, renforce l'idée de l'informatique collaborant à la recherche biologique pour l'application à la médecine.

L'équipe SIC (Systèmes Intégrés Cognitifs) comprend une dizaine de chercheurs et est sous la responsabilité de Catherine GARBAY. Les recherches concernent l'Intelligence Artificielle de façon générale et plus précisément l'Intelligence Artificielle Distribuée, qui à travers l'approche technologique dite "Multi-Agents" articulent les travaux de l'équipe SIC autour d'un objectif technologique bien précis : la conception de "Systèmes Intégrés Cognitifs". Ce sont des systèmes capables d'intégrer diverses formes de connaissances et de traitements pour la modélisation, la simulation et la résolution de problème. Le domaine d'application privilégié de l'équipe est celui de la cytologie (étude des cellules en biologie) et a donné naissance à plusieurs projets : le diagnostic biomédical, la vision par ordinateur, l'acquisition des connaissances et la modélisation et la simulation de la vie cellulaire. L'approche Intelligence Artificielle Distribuée est à la base de tous ces travaux et particulièrement le système Multi-Agents MAPS développé au sein de cette équipe, et qui est un système de résolution de problème par coopération d'agents.

Si l'Intelligence Artificielle s'efforce de modéliser ou de simuler le comportement intelligent d'un individu, l'Intelligence Artificielle Distribuée s'intéresse à des comportements intelligents collectifs qui sont le produit de l'activité coopérative de plusieurs individus. Parmi ce courant l'approche technologique qui nous intéresse ici, appelée Multi-Agents, a pour principe de définir un ensemble d'agents, entités informatiques actives et relativement autonomes, communicant par envoi de messages. Mais cette approche peut être vue également sous l'angle de la méthodologie de conception de logiciels. La finalité de cette approche étant la

résolution de problème par coopération d'agents, la question est alors : comment concevoir un logiciel Multi-Agents tout en gardant une vision suffisamment abstraite du problème à résoudre? En effet envisager un problème nécessite de pouvoir rester à un niveau d'abstraction élevé pour ne pas mélanger les concepts de résolution avec les difficultés d'une programmation.

Le logiciel MAPS, base de cette étude, a été conçu en vue des objectifs suivants :

- offrir une modélisation haut niveau des problèmes à résoudre,
- favoriser l'exploitation de ressources hétérogènes, la coopération de tâches et d'activités différentes au sein d'un même système,
- développer des stratégies complexes de résolution de problème.

Ce logiciel est un environnement générique permettant la conception de système Multi-Agents pour la résolution "experte" de problèmes. Il est une plate-forme d'expérimentation de recherche et de réflexion pour l'équipe SIC mais aussi un concept fédérateur de l'ensemble des travaux de cette équipe.

Néanmoins le besoin d'un "schéma de conduite" dans la génération d'une application - nous parlerons de *méthodologie* - nécessite que les "outils" à la disposition de l'utilisateur, et parmi ceux-ci un *langage* adapté à ce mode de pensée, soient disponibles dans un *environnement de programmation.* Le travail a donc consisté à "repenser" et reconcevoir le logiciel MAPS dans une optique Génie Logiciel, la seule capable de fournir rigueur et qualité.

En résumé le travail est de réaliser un environnement de programmation d'aide au développement d'applications Multi-Agents, en effectuant un lien étroit entre Intelligence Artificielle Distribuée et Génie Logiciel.

Le chapitre 1 décrit le cadre et l'environnement de ce mémoire, la description entreprise est descendante. Depuis le développement de l'informatique et de l'Intelligence Artificielle, les techniques se sont affinées, nous verrons quelles sont les directions qui ont été envisagées avec leurs problématiques. Parmi ces directions les Systèmes Multi-Agents sont abordés plus en profondeur car ils constituent l'enveloppe de notre projet.

Le chapitre 2 est un prolongement du premier chapitre en ce sens qu'il décrit "un" système Multi-Agents. Ce chapitre développe, sur le système MAPS, les principes énoncés au chapitre précédent. Nous finirons par une rapide analyse critique pour soulever les difficultés et les points susceptibles d'évolutions.

Le chapitre 3 concerne l'étude bibliographique et est un tour d'horizon des sujets qui peuvent nous apporter une aide, un moyen de compréhension et de développement dans notre entreprise. Les sujets abordés seront successivement "le Génie Logiciel", "les Langages à objets" et quelques "systèmes voisins". Le premier sujet est important à évaluer car c'est lui qui nous apportera la rigueur nécessaire. Le deuxième sujet permettra d'envisager l'approche orientée objet comme point de départ pour parvenir à l'approche orientée agent. Le troisième

sujet donnera une "idée" des différentes façons d'envisager la réécriture du système MAPS au travers de la description de quelques systèmes voisins.

Le chapitre 4 est la réalisation : le système COALA. Nous verrons le pourquoi de ce nom. Il a été envisagé en trois axes, méthodologie, environnement et langage. Ce sont ces trois axes qui seront développés.

En Conclusion nous évaluerons l'apport de l'approche objet par rapport à l'approche agent et nous donnerons les perspectives de ce projet, tant sur le point évolution des concepts que sur le plan technique.

La figure 1 résume le plan de ce mémoire.

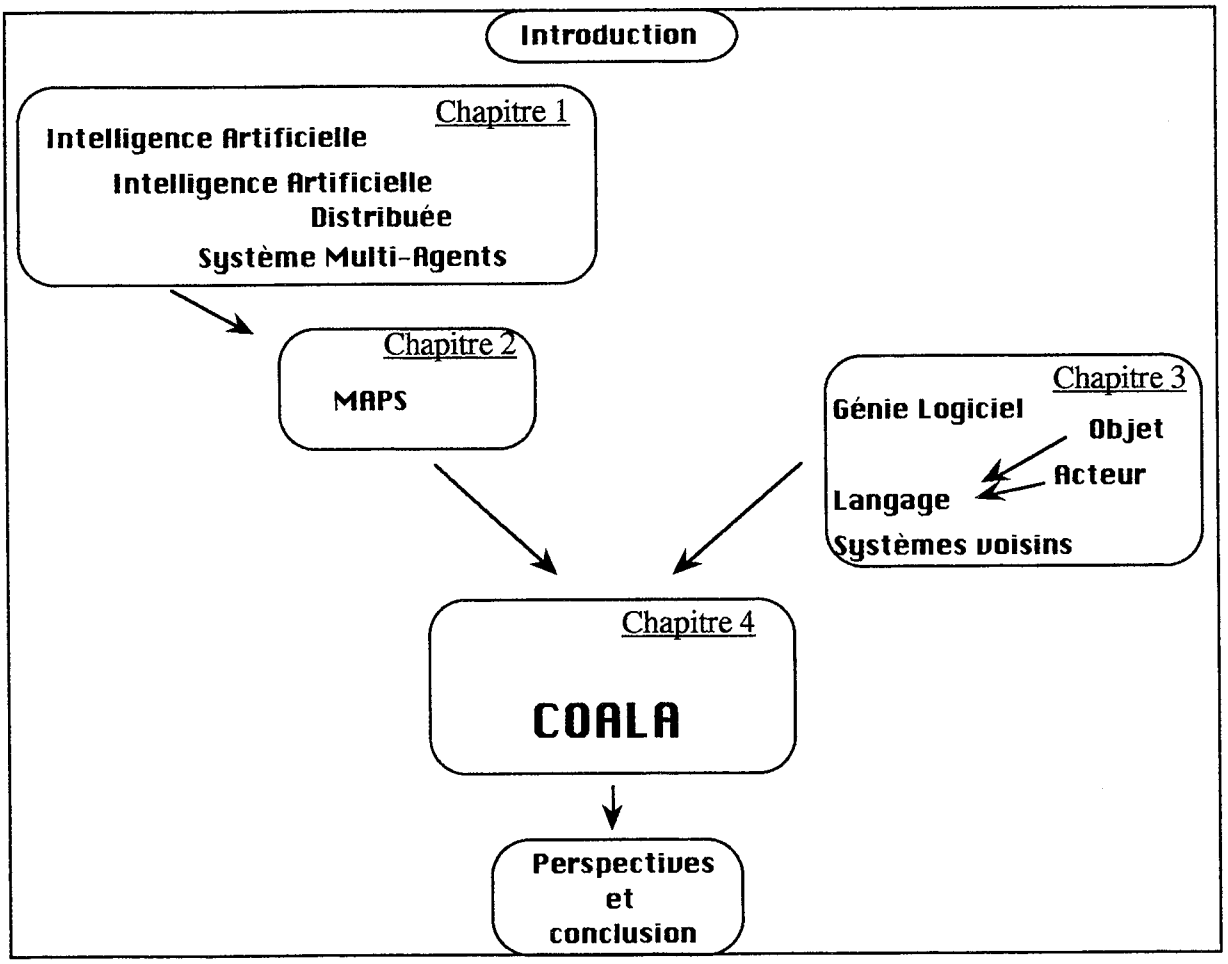

Figure 1 : Plan du mémoire.

Diverses annexes donnent des détails sur le langage COALA et sur la grammaire du langage COALA, un exemple de programmation et de correspondance entre langage et librairie de classes C++, des précisions sur les critères de qualité, des informations techniques matérielles et logicielles, ainsi que quelques définitions générales de termes de la langue française réemployés en Intelligence Artificielle.

Chapitre 1 - Cadre et Environnement

I - Introduction

Reproduire l'intelligence a été de tout temps un but, sinon un rêve, celui de recopier dans une "machine" les activités de cognition d'un être humain, permettre à une machine de penser, de réaliser des activités depuis toujours réservées à des êtres pensants.

De Frankenstein et autre Golem au joueur d'échecs de Kemplen la volonté de copier tant sur le plan biologique que sur le plan mécanique a toujours été virulente.

Avant que l'électricité puis l'électronique ne stimulent les esprits, la mécanique a beaucoup suggéré, témoin les automates de Vaucanson.

Même si les ambitions des ancêtres de ces précurseurs ne restent que dans la science-fiction, comment oublier ces magiciens des temps modernes qui malgré des prétentions sans cesse revues à la baisse on fait évoluer la pensée pour nous permettre d'être à l'aube du vingt et unième siècle de plain-pied dans la réalité informatique ?

Les progrès de l'électronique ont beaucoup fait pour faire évoluer les possibilités de réflexions. Des capacités de traitement, de stockage toujours plus grandes ont permis l'évolution des concepts qui étaient parfois déjà présents. En effet selon Hubert Reeves, par exemple les notions mathématiques ont été créées avant que l'on se rende compte de leur impact sur la réalité [REEVES 90]. Plus techniques, les calculs de Charles Babbage pour réaliser un calculateur se sont révélés exacts avant que l'on puisse réaliser ses ambitions à cause de capacités technologiques insuffisantes en mécanique [SWADE 93].

II - L'Intelligence Artificielle

Le nom "Artificial Intelligence" a été inventé en 1956 à l'issue d'une réunion où deux chercheurs NEWELL et SIMON présentent un démonstrateur de théorème, ordinateur ayant des possibilités de manipulation de symboles [PITRAT 92]. Depuis cette époque le développement de matériels de plus en plus puissants a contribué à de grandes avancées dans ce domaine. Les recherches se sont orientées aussi bien sur le plan technologique que dans la voie de création de nouveaux langages de programmation.

L'engouement suscité par cette nouvelle discipline a été vif, mais les difficultés liées à l'expérimentation de concepts nouveaux, et aux limites technologiques en a réduit les perspectives et les développements. L'Intelligence Artificielle actuellement en est au stade d'une intégration dans un développement industriel. En effet les prétentions de cette discipline et ses

chances de survie ont été d'être intégrées dans le processus industriel avec notamment des perspectives intéressantes en ingénierie de la connaissance. Ce repositionnement dans des projets industriels est prometteur et de grands groupes industriels l'ont intégré dans leur recherche et développement [JEANNE 93].

Pour donner une définition très succincte de l'Intelligence Artificielle on peut dire que l'intelligence étant ce qui caractérise l'homme, ce qui le différencie du reste du monde :

"L'Intelligence Artificielle
cherche a reproduire l'activité humaine de cognition au sein d'une machine."

Tenant compte de la réalité technologique une définition plus pragmatique peut être donnée :

"L'Intelligence Artificielle
c'est l'implantation de méthodes qui donneront à un calculateur
le "comportement" d'un humain, dans des activités
de perception, de compréhension, de décision."

Dans un premier temps on a donc tenté d'identifier l'être humain à un programme unique représentant directement un "expert" capable de résoudre un problème par lui-même. Cette manière de penser se retrouve dans la notion de "*systèmes experts*", catégories de programmes informatiques censés être capables de remplacer l'être humain dans ses tâches réputées les plus complexes qui, pour leur résolution réclament de l'expérience, du savoir-faire, et une certaine forme de raisonnement [ERCEAU 93].

Les systèmes experts conçus comme "*connaissances + raisonnement + contrôle*", ayant l'avantage de posséder un pilote qui organise les activités séquentiellement, sont néanmoins des systèmes centralisés et se sont trouvés confrontés à des difficultés inhérentes à cette centralisation.

D'autre part la complexité des systèmes experts n'a cessé d'augmenter, simultanément avec les capacités de traitement et consécutivement à l'expérience qui en a été retirée.

Enfin les domaines d'application sont de plus en plus diversifiés et les connaissances mises en oeuvre sont de plus en plus variées.Une résolution de problème complexe est la somme de compétences possédées non pas par un expert mais par des "experts" différents qui doivent discuter de leurs points de vue et prendre des décisions conséquences de leurs dialogues.

On passe donc d'une entité possédant le savoir et les connaissances nécessaires à la résolution d'un problème, à un ensemble d'entités en interactions, chacune d'entre elles ayant des connaissances propres à un domaine particulier du problème à résoudre et n'ayant pas la vue globale de ce problème.

L'Intelligence Artificielle Distribuée est le domaine qui s'occupe de cette approche. Schématiquement et en première approximation nous dirons que l'Intelligence Artificielle

s'intéresse à modéliser un individu alors que l'Intelligence Artificielle Distribuée s'occupe de modéliser un groupe d'individus. Ce schéma découle de plusieurs constatations, certaines limitations de l'Intelligence Artificielle classique, de nouvelles exigences nées de l'évolution de ces disciplines et une "utilisation" de comportements sociologiques et biologiques.

III - L'Intelligence Artificielle Distribuée

Une première raison à la "distribution" est que certaines applications sont naturellement distribuées, soit géographiquement comme en témoignent l'application du projet européen ARCHON qui s'occupe de l'exploitation de grands réseaux de production et de distribution d'énergie électrique, soit fonctionnellement pour respecter l'autonomie de certaines expertises comme dans la plannification de missions spatiales par exemple.

Une deuxième raison est qu'il peut être intéressant voire nécessaire de réutiliser des composants informatiques existants, leurs combinaisons et leurs interactions avec des systèmes en développement devenant difficiles, car il n'existe pas de plate-formes hétérogènes pour intégrer ces possibilités. Il devient donc nécessaire de disposer des moyens de s'abstraire de ces difficultés et de passer à un niveau supérieur d'expression et de manipulation des ressources [HAYES-ROTH 91].

Une autre raison est que les connaissances peuvent être très diverses d'un domaine à l'autre limitant la centralisation, l'exploitation de ces connaissances variées ayant conduit à l'existence de systèmes spécialisés.

L'Intelligence Artificielle Distribuée propose donc de répartir les différentes capacités entre des "experts" bien adaptés à un problème où à une situation donnée, la difficulté étant alors d'établir une conversation cohérente et constructive pour obtenir une coopération efficace.

Parmi les trois voies envisagées par l'Intelligence Artificielle Distribuée [BOND 88] nous trouvons :

- une première voie, appelée DPS pour Distributed Problem Solving, est une solution plutôt logicielle qui consiste à décomposer un problème en sous tâches coopérant à la résolution du problème global,
- une deuxième voie plus axée sur le développement d'architectures parallèles est appelée PAI pour Parallel Artificial Intelligence et privilégie la création de langages et algorithmes pour ces architectures,
- une troisième voie très explorée qui nous intéresse est celle des systèmes Multi-Agents. Elle concerne la coopération d'entités intelligentes appelées "agents" qui partagent des connaissances et sont douées de raisonnement, mais n'ont pas de vue globale du problème à résoudre.

Pour répondre aux exigences de dynamique de participation d'un individu dans un groupe, l'Intelligence Artificielle Distribuée a emprunté des concepts à la sociologie, la psychologie [ERCEAU 91], comme en témoignent les termes employés pour la communication et le contrôle entre "agents". On parle de "négociation de contrats", d'"actes de langages", de "planning", de "tableau noir" ... pour "allouer des tâches", "communiquer", "coordonner", "gérer les conflits" ...

Un autre emprunt enfin est fait cette fois-ci à la biologie, c'est la métaphore des fourmis. Les "agents" fourmis en très grand nombre n'ont pas individuellement d'intelligence suffisante pour couvrir l'ensemble du problème. Mais le résultat va émerger de l'interaction de ces agents entre eux. On dit qu'il s'agit d'émergence de résultats par mécanismes de réaction aux événements. On pourrait comparer ceci à des influences de types "effets de bords". Cette approche a donné naissance à l'école "réactive", les agents sont peu ou pas "intelligents", on dit qu'ils sont de faible granularité.

L'approche sociale duale décrite précédemment étant l'école "cognitive" et les agents y sont de forte granularité.

Les études des comportements sociologiques humains, et des formidables émergences dynamiques d'une fourmilière, termitières et autres ruches, ont eu un fort impact sur le développement des systèmes Multi-Agents. L'école cognitive étant à l'heure actuelle celle qui a donné lieu au plus grand nombre d'applications, mais l'école réactive se développe rapidement.

IV - Les Systèmes Multi-Agents

1 - Définitions

De la même façon schématique que lors de la comparaison Intelligence Artificielle classique et Distribuée, on peut dire que les Systèmes Experts sont à l'Intelligence Artificielle ce que les Systèmes Multi-Experts et plus généralement les Systèmes Multi-Agents sont a l'Intelligence Artificielle Distribuée.

Donnons en premier lieu quelques définitions [ERCEAU 93].

Systèmes Multi-Agents : On appelle Systèmes Multi-Agents un système <O,E,A>, où O est un ensemble d'objets, A est un ensemble d'agents, O et A étant immergés dans un environnement E.

Agent : On appelle agent une entité physique ou abstraite qui est capable d'agir sur elle-même et sur son environnement, qui ne dispose que d'une représentation partielle de cet environnement, qui peut communiquer avec d'autres agents, qui poursuit un objectif individuel, et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions qu'il peut avoir avec d'autres agents et l'environnement.

Ces définitions mettent en évidence les points suivants qui apparaîtront au long de cette étude :

- la notion de ressources, ce sont les connaissances sur le domaine d'application , c'est-à-dire les compétences de l'agent. Ces connaissances sont propres à l'agent indépendamment de la société dans laquelle il opère.

- la notion de connaissances sur soi et les autres (accointances), c'est la partie qui permet à l'agent de gérer ses relations avec les autres agents. Ces connaissances portent sur les interactions avec les autres, la participation dans la société.

- la notion de communication, ce sont les protocoles de communication entre un agent et son environnement (les autres agents).

- la notion de contrôle (capacité de raisonnement, comportements). Le raisonnement est la façon dont on enchaîne les inférences. La structure de contrôle donne les modes d'exploitation des connaissances. Le comportement est la manière dont réagit un agent à un événement.

L'introduction de ces notions amène à une classification comme celle proposée par [ERCEAU 91] qui différencie des agents réactifs, cognitifs communiquants, rationnels, spécialistes, et intentionnels. Dans cette hiérarchie l' "intelligence" va croissante, celle-ci étant progressivement introduite comme capacité à réagir sur des événements extérieurs, et comme capacités à résoudre des problèmes. De cette classification découle les deux grandes classes de systèmes multi-agents évoquées précédemment. L'école réactive qui prévoit l'émergence des résultats comme provenant de la coopération de multiples agents très simples et peu ou pas "intelligents", c'est la métaphore de la fourmilière. L'autre tendance cognitive, envisage peu d'agents mais leurs capacités de raisonnement sont importantes, c'est la métaphore du groupe de personnes.

Ces définitions font apparaître deux niveaux de description [PLEIAD 92]. Un macro-niveau, le système, vu comme une société d'agents, construit à partir d'éléments ayant des connaissances, des capacités de résolution et qui interagissent et un micro-niveau, celui de l'agent, entité de base ayant un problème à résoudre.

Dans les deux paragraphes suivants nous allons aborder les deux niveaux de description d'un système Multi-Agents. Le niveau société d'agents relève d'une étape d'abstraction importante puisqu'elle s'intéresse à la répartition et à la décomposition de l'application en termes d'agents et de leurs ressources, capacités et fonctionnalités. Le niveau agent est plus proche de l'implantation car c'est là que se réalise l'écriture de l'application. C'est en ces termes que nous allons effectuer cette description.

2 - Le point de vue société

Lorsqu'un système est développé des choix doivent être faits au niveau de l'organisation des agents, en fonction de la distribution et de la granularité des agents que l'on désire, au

niveau des protocoles de communications dont on dispose et au niveau de la façon d'entreprendre une résolution de problème.

Néanmoins deux critères semblent importants pour classifier les architectures des systèmes Multi-Agents : le partage des tâches et des connaissances et le mode d'interaction et de communication entre agents.

le partage des tâches et des connaissances

Dans la conception d'un système Multi-Agents, la détermination du rôle de chaque agent et le partage des tâches et des informations constituent des points fondamentaux [HATON 89]. Deux grandes approches sont possibles suivant le mode de coopération que l'on entreprend.

La première approche de la coopération est dite volontaire, chaque agent possède alors des capacités dans un sous-domaine et contribue à une partie de la résolution d'un problème. Il n'y a pas de conflit entre des agents complémentaires.

Selon [HAUTIN 86] la coopération volontaire s'applique bien à la résolution distribuée de problèmes, approche dite DPS, et il envisage trois modes de coopération dans ce cas :

- Le mode "commande" où un agent décompose un problème en sous-problèmes qu'il répartit entre d'autres agents. Ceux-ci lui renvoie leurs résultats.
- Le mode "Appel d'offre", un agent après décomposition d'un problème en sous-problèmes, les propose à un certain nombre d'agents. Ceux-ci lui font une offre et il choisit sa distribution.
- Le mode "compétition", un agent décompose un problème en sous-problèmes et en diffuse la liste. Les agents intéressés répondent et il choisit parmi les résultats.

Ces trois modes de fonctionnement s'implantent sur un ensemble d'agents organisé en hiérarchie. Si elle est absente, la coopération se fera par négociation entre agents. Enfin dans le cas où le problème ne peut être découpé, la coopération peut se faire par échanges de résultats partiels.

La seconde grande approche envisage la concurrence entre des agents. Plusieurs agents ayant des expertises différentes proposent des solutions à un même problème. Il faut alors résoudre les conflits apparaissant dans de tels systèmes.

Le mode d'interaction et les communications entre agents

Le point de vue de la communication s'oriente vers deux grandes directions, le partage de ressources par tableau noir et les envois de messages [HATON 89]

- Dans le modèle "tableau noir" les informations sont échangées via une structure de donnée commune à tous les agents : le tableau. Il contient au départ les données du problème et

s'enrichit progressivement des résultats partiels jusqu'à contenir la solution globale du problème posé.

- Dans l'envoi de messages les informations sont échangées directement entre des agents autonomes en données et connaissances. Deux agents établissent une communication selon un protocole clairement établi.

La figure 1.1 suivante résume les deux approches :

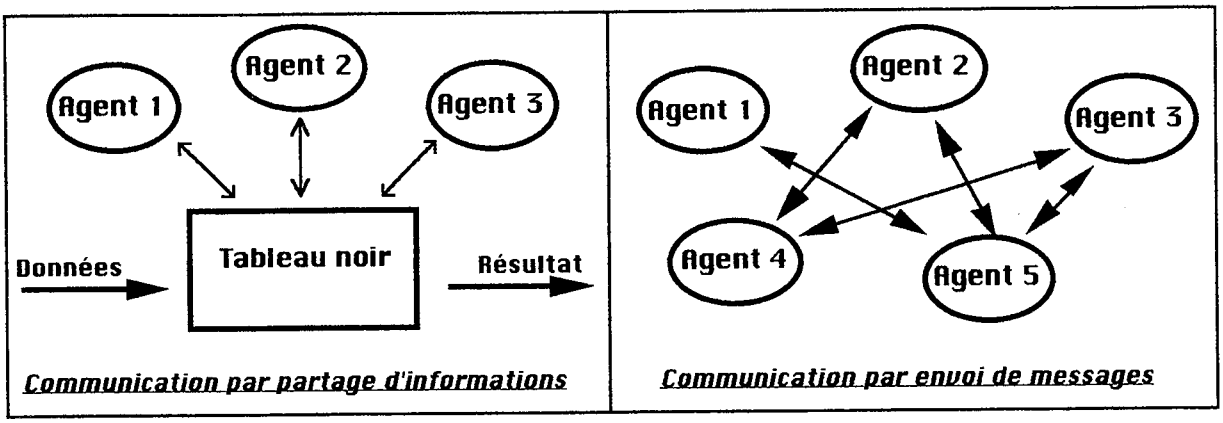

Figure 1.1 : Deux types de communication.

L'intérêt de l'approche par tableau noir est que les possibilités de contrôle centralisé y sont plus importantes, la vision du problème y est globale. Par contre l'approche par envoi de messages permet plus de souplesse et d'autonomie des agents mais le langage de communication doit être plus élaboré et le contrôle est plus difficile à mettre en place, la vision globale étant absente.

En conclusion, la méthode de résolution de problème est grandement influencée par l'architecture du système Multi-Agents choisi. Une centralisation conduit plus facilement à une plannification globale alors qu'une grande autonomie des agents conduit à beaucoup d'échanges entre eux. Les protocoles étant alors plus ou moins évolués suivant que les échanges porteront sur des données brutes ou sur des commandes.

3 - Le point de vue agent

Nous allons aborder ici uniquement les caractéristiques évoquées au paragraphe 1 des définitions, des agents cognitifs communiquants. En effet le système MAPS (Multi-Agents Problem Solver), qui est central dans notre travail, utilise ce type d'agents.

Les ressources

Au point de vue de l'implantation, l'influence des représentations orientés objet (structure de donnée ayant des actions associées) est importante. On retrouvera ainsi souvent le principe des bases de faits et de règles, le principe de frames, mais aussi les règles de production.

connaissances sur soi et les autres

Ici un compromis doit être entrepris, il s'agit soit de connaissances statiques qui vont faciliter le raisonnement mais réduisent l'autonomie de l'agent, sa capacité d'apprentissage étant réduite, soit un agent est capable de faire évoluer sa représentation des autres, le dialogue donc les protocoles de communication seront alors plus évolués.

communications

Elles sont basées sur des protocoles, ceux-ci peuvent être synchrones ou asynchrones. Si les protocoles sont très évolués, les communications seront simplifiées car le langage de communication est de haut niveau. Si à l'opposé on implante des protocoles peu évolués, les communications seront alors directement "codables" (dans des règles par exemple) au niveau de la représentation interne de l'agent.

contrôle, comportement

L'implantation du contrôle et des comportements peut être réalisée par des méthodes de réaction aux événements et dans un moteur d'inférence opérant sur des règles d'action et/ou de stratégie.

V - Conclusion

Cette description rapide en deux niveaux, société et agents des systèmes multi-agents nous a permis de définir les termes et aborder les problématiques.

Le contexte qui nous intéressera est celui de résolution de problème par coopération d'agents, les systèmes multi-agents étant alors une approche technologique intéressante.

Dans le chapitre suivant la description du logiciel MAPS base de notre travail va être abordée dans les termes définis.

Nous verrons ensuite rapidement les avantages du Génie Logiciel dont nous essayerons de retenir les principaux atouts. Puis nous entreprendrons un panorama d'outils susceptibles de nous intéresser, il s'agit notamment de la conception orientée objet. Enfin nous élargirons notre horizon à l'étude d'autres réalisations, en essayant de déterminer quelles sont les particularités qui nous intéressent.

Chapitre 2 - Etude de l'existant : le système Multi-Agents MAPS

I - Introduction

MAPS (Multi-Agents Problem Solver) est un produit logiciel de l'équipe SIC du laboratoire TIMC de l'IMAG. C'est le résultat d'un travail de thèse réalisé par Olivier Baujard entre 1989 et 1992 [BAUJARD 92]. Les principales personnes qui ont participées à la définition de ce logiciel sont Catherine Garbay [GARBAY 88] et Sylvie Pesty [PESTY 89].

Première définition générale :

MAPS est dédié à la conception de systèmes Multi-Agents,
il a été développé dans le but de fournir un outil de développement (langage + interface)
permettant de décrire et résoudre un problème de façon distribuée.

Les principales applications concernent la vision par ordinateur [BAUJARD 92], le diagnostic biomédical [OVALLE 91], la reconnaissance de la parole [CAILLAUD 91], ainsi que l'acquisition des connaissances [HUGONNARD 93].

Dans la suite de ce chapitre nous aborderons le système MAPS du point de vue de l'utilisateur. La description en deux niveaux vue dans le chapitre précédent va être reprise pour détailler le fonctionnement de MAPS. Le point de vue société sera exploité simplement sur un plan architecture qui relève du choix de l'utilisateur pour son application. Le point de vue agent concernera essentiellement le détail du contenu des agents génériques. Nous verrons ensuite l'implantation qui en a été faite et enfin nous effectuerons une analyse critique et verrons quels sont les points à faire évoluer.

II - Des concepts à l'implantation

1 - L'approche réseau

Sans pour autant interférer sur la prochaine description des agents MAPS, on peut déjà dire que le principe de base vient de la distinction, classique en intelligence artificielle, entre deux types de connaissances : les connaissances figuratives et les connaissances opératoires, que nous détaillerons dans le prochain paragraphe. De ce principe est née l'idée d'encapsuler ces connaissances dans deux types d'agents, KS (Knowledge Server) et KP (Knowledge

Processor). Les agents KS manipulant les connaissances figuratives, les agents KP manipulant les connaissances opératoires. Ces agents étant connectés en réseau, en alternant KS et KP

La figure 2.1 présente un réseau minimal.

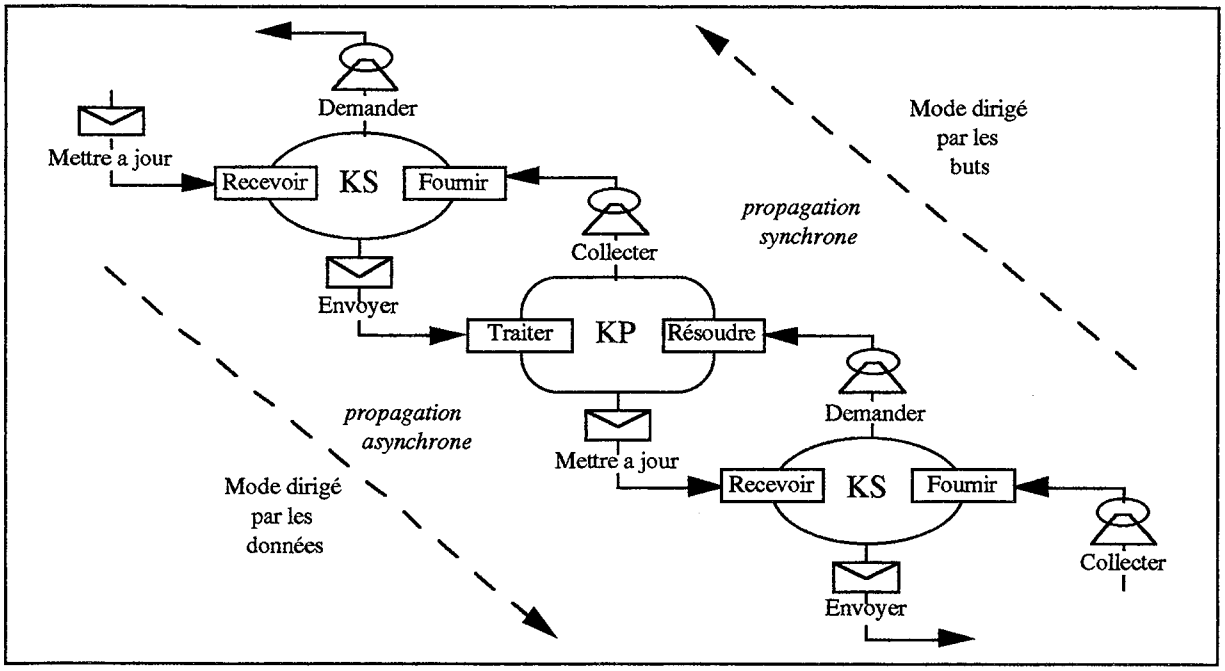

Figure 2.1 : Représentation minimale d'un réseau d'agents KP et KS.

Les principales architectures que l'on peut considérer avec un réseau d'agents MAPS sont de deux sortes, celles fondées sur des groupes centrés connaissances et celles fondées sur des groupes centrés tâches.

Un groupe centré connaissances concerne un certain nombre d'agents KP réalisant des traitements sur les connaissances détenues par un seul agent KS. (figure 2.2)

Ce dernier correspond alors à un niveau d'abstraction sur lequel opère des traitements locaux. Chaque niveau d'abstraction étant lié à un style de représentation, un mécanisme de traduction est mis en oeuvre pour que les messages soient compréhensibles d'un niveau à un autre. Cette traduction est réalisée par un agent KP qui traite et met en forme ces informations.

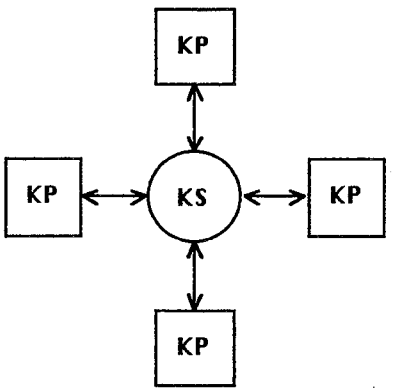

Figure 2.2 : Architecture centrée connaissances.

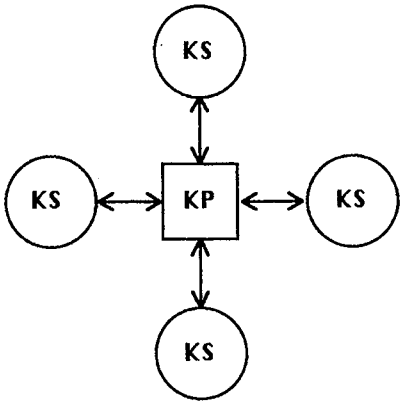

Figure 2.3 : Architecture centrée tâches.

L'approche duale est la constitution de groupes centrés tâches, ceux-ci étant constitués d'un agent KP qui effectue une tâche particulière sur un ensemble de connaissances distribuées au sein de plusieurs agents KS (ou de groupes centrés connaissances), qui devront maintenir les données, les hypothèses et les résultats produits. Chaque groupe constitué devra réaliser une tâche précise, l'échange entre groupe concernera des résultats intermédiaires. (figure 2.3.)

La figure 2.4 suivante résume une architecture où de tels groupes sont mis en évidence.

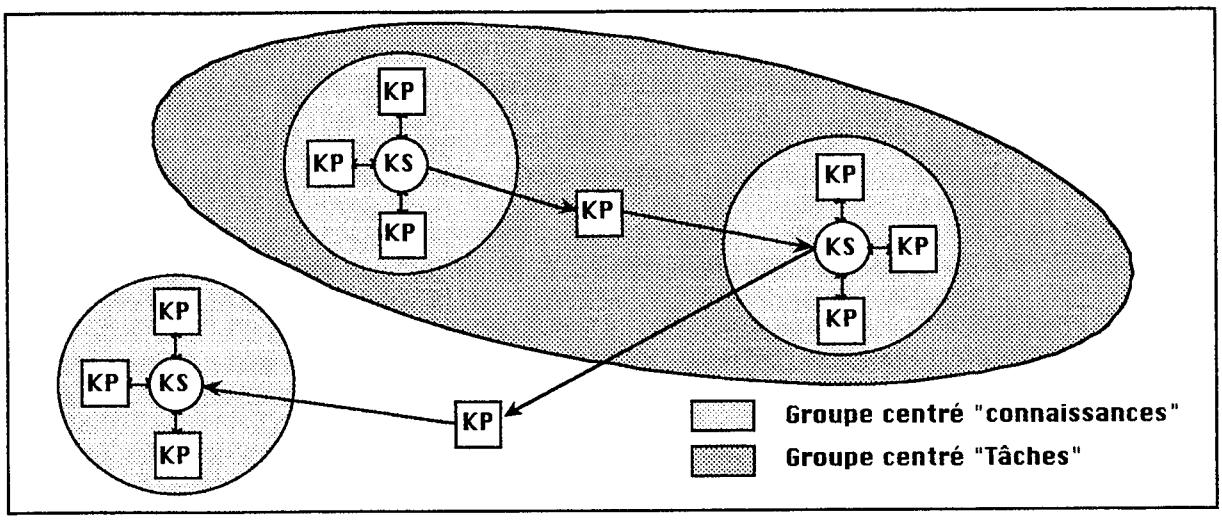

Figure 2.4 : Architecture centrée "tâches" et centrée "connaissances".

Cette dualité d'architecture permet d'obtenir deux types de "dialogues":

- un dialogue entre niveaux d'abstraction introduisant une coopération orientée vers la progression et la complétion de la connaissance.
- un dialogue entre groupes introduisant une coopération orientée partage de tâches.

Notons sans nous étendre une certaine ressemblance avec une approche tableau noir distribué. Le tableau noir est ici distribué au sein des agents KS.

La distribution porte aussi sur le contrôle, un agent connaissant ses accointances (agents avec lesquels il est connecté) et leurs actions. Le contrôle étant codé dans les règles au sein de chaque agent et dans les moteurs d'inférence agissant sur ces règles comme nous le verrons dans le prochain paragraphe.

Nous voyons apparaître certains principes des systèmes Multi-Agents énoncés précédemment :

- populations d'agents,
- interactions, communications entre agents,
- connaissances sur les autres,
- distribution des capacités de travail et de contrôle.

et il se dégage le fait que MAPS effectue de "la résolution de problème par coopération d'agents".

2 - Les agents génériques

2.1 - Principe

Deuxième définition générale :

Un agent MAPS est une entité autonome, un système à base de connaissances capable de réagir à la réception d'événements extérieurs, c'est-à-dire à des requêtes d'autres agents, grâce à un comportement spécifique.

La présence de deux types d'agents résulte de la distinction que l'on peut faire sur les types de connaissances.

Une connaissance peut être définie comme une information et une méta-information qui est liée à la manière dont nous rangeons et utilisons cette information. Ce principe est celui qui est à la base des systèmes experts qui évoquent des connaissances factuelles (faits) et des connaissances déductives (règles). De façon plus générale on emploie les termes d'objet et action dont on peut donner une définition d'après [LAURENT 85] :

- objet, toute connaissance dont la représentation implique une interprétation factuelle de la connaissance abstraite correspondante. Un objet est une connaissance que l'on choisit statique (de définition). C'est une unité de savoir, une connaissance figurative dans notre vocabulaire.
- action, toute connaissance dont la représentation implique une interprétation dynamique de la connaissance abstraite correspondante. C'est une unité de savoir-faire, une connaissance opératoire dans notre vocabulaire.

A partir de cette distinction il a été développé deux types d'agents génériques :

- Un agent appelé KS (Knowledge Server), le serveur de connaissances qui a essentiellement une fonction de maintien et de transmission des connaissances figuratives qu'il détient.
- Un agent appelé KP (Knowledge Processor), le processeur de connaissances qui a une fonction d'exploitation et de mise en oeuvre des connaissances opératoires qu'il détient.

A cette étape on peut donner une définition plus technique :

MAPS permet de concevoir un réseau d'agents cognitifs communicant, chaque agent étant défini comme un système à base de connaissance à part entière, communicant par envoi de message en mode commande.

Les éléments de la définition de l'agent MAPS sont répartis dans trois couches différentes définissant sa structure :

- la couche interne comprend les ressources dont est doté l'agent : ces ressources définissent son expertise, son domaine de compétence pour l'application, et son savoir : savoir figuratif ou savoir opératoire;
- la couche intermédiaire définit les capacités de contrôle de l'agent; ces capacités expriment les formes génériques d'exploitation des ressources internes (lire, écrire, rechercher, sélectionner, planifier...);
- la couche externe définit le comportement de l'agent vis a vis des autres agents.

Nous allons maintenant analyser les principales caractéristiques de chaque type d'agent en gardant une vision utilisateur.

2.2 - Agent KS

La figure 2.5 suivante résume les différentes composantes d'un agent KS :

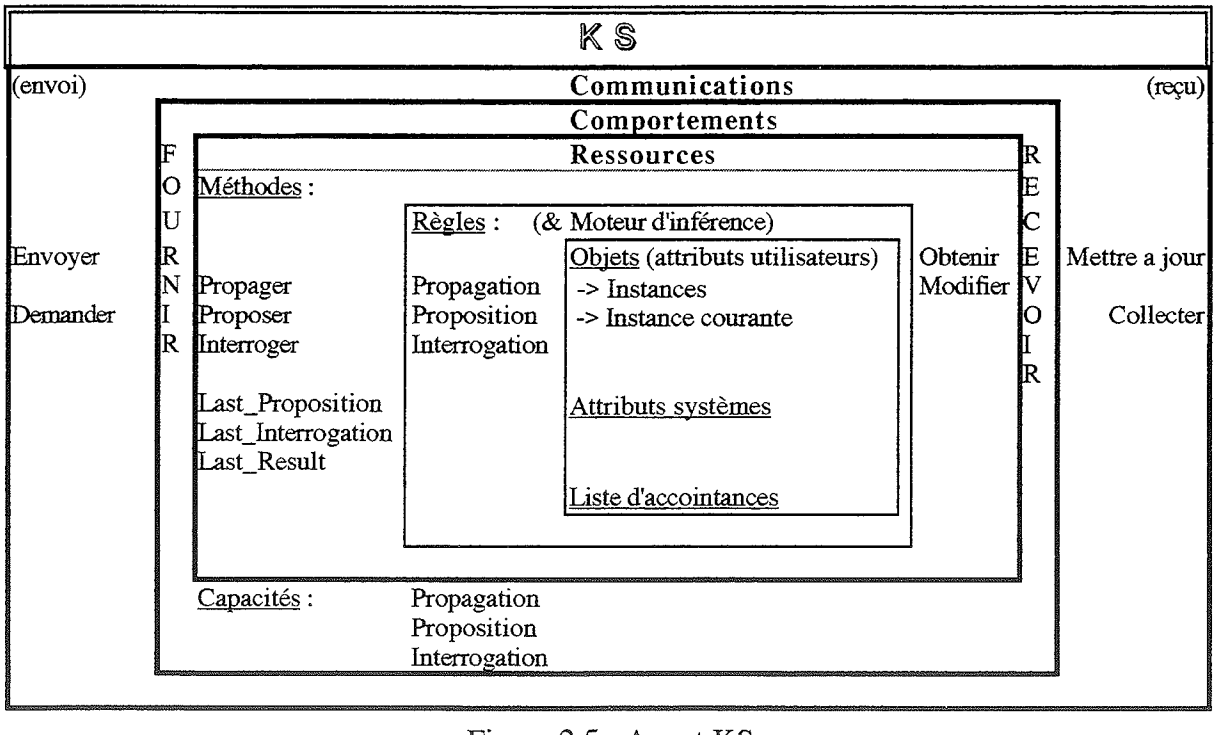

Figure 2.5 : Agent KS.

2.2.1 - Les ressources

Elles sont définies sous forme d'objets. Ceux-ci sont exploités par plusieurs jeux de règles. A l'intérieur de ces règles, ce sont des méthodes qui permettent l'accès aux objets. Ces règles sont elles-mêmes exploitées par un moteur d'inférence.

- ***les objets*** sont de deux types :

-- *définis par l'utilisateur*, ils définissent un modèle pour la structure de leur représentant physique, les instances, et regroupent des attributs ou variables d'instances. Une instance peut être assimilée à un monde de pensée et plusieurs instances peuvent être maintenues en parallèle. Chaque objet maintient la liste de ses instances et de son instance courante car c'est sur celle-ci que s'effectuent les accès.

-- *attributs de contrôle* ou attributs systèmes, ajoutés à chaque objet en plus des attributs définis par l'utilisateur. Ils sont de deux types :

--- Trois indicateurs PROPOSE_FLAG, INTERROGATE_FLAG et RESULT_FLAG qui donnent des informations sur le dernier agent interrogé, auquel il a été proposé une information et le résultat (succès ou échec) de ce transfert ou demande d'information. Ces informations permettent ainsi de raisonner sur les capacités de réception de l'agent KP. De plus ces attributs servent de trace des actions effectuées par l'agent lui-même.

--- une liste des instances de chaque objet.

- ***les règles*** sont de trois types :

-- *propagation* dont le rôle est de propager localement une information ou d'effectuer des opérations de vérification de la cohérence.

-- *proposition* dont la fonction consiste à sélectionner une information susceptible d'être transmise ou requise vis-à-vis de l'extérieur (c'est-à-dire d'un agent KP).

-- *interrogation* pour requérir une information vis-à-vis de l'extérieur (agent KP).

Un ensemble de ***méthodes*** de deux types sont disponibles :

-- des méthodes pour manipuler les jeux de règles, ce sont Propager, Proposer et Interroger.

-- des méthodes pour manipuler les objets, leurs instances et les attributs systèmes, telles que Obtenir (objet ou instance), Modifier (objet ou instance), Last_Proposition, Last_Interrogation, Last_Result.

2.2.2 - Les connaissances sur soi et les autres

Elles sont de deux sortes :

- statiques, contenues dans une liste d'accointances détenue par l'agent KS, c'est-à-dire les agents KP auxquels il est connecté.

- dynamiques, par l'intermédiaire des attributs système qui fournissent une trace des dernières actions entreprises.

2.2.3 - Les communications

Les messages reçus par un agent KS sont de trois types :

- deux messages d'informations,
  - -- la requête "*mettre à jour*" lui communique de nouvelles informations
  - -- la requête "*collecter*" lui demande des compléments d'informations.
- trois messages de contrôle qui concernent les attributs systèmes évoqués ci-dessus. Ils activent les méthodes d'accès aux attributs de contrôle et donnent en retour à l'agent KP des informations sur les actions entreprises par l'agent KS.
- trois messages de focalisation sur les instances d'objets, ces messages activent les méthodes de manipulation d'instances et permettent à un agent KP de travailler sur tel ou tel monde de pensée.

La requête Collecter est une communication asynchrone, la requête Mettre à jour est synchrone. Toutes les autres requêtes s'effectuent en mode synchrone.

2.2.4 - Le contrôle

Il est présent sur trois niveaux :

- niveau ***comportement*** qui définit la manière de réagir face à la réception ou à la demande d'une information impliquée par les requêtes "Mettre à jour" et "Collecter", ce sont *Recevoir* et *Fournir* qui peuvent se résumer comme suit :

Recevoir :

```
(Et Modifier
   (Ou Proposer
      (Et Interroger
         (Envoyer Self Recevoir))))
```

Fournir :

Obtenir

- niveau ***compétence*** , ce sont ses capacités de Propagation, Proposition et Interrogation. Mais également toutes les capacités exprimées par l'utilisateur dans les règles.
- niveau ***moteur d'inférence*** , pour le fonctionnement des règles de l'agent KS un moteur d'inférence fonctionne en chaînage avant de la manière suivante pour chaque type de règles :

1- Recherche d'un ensemble de conflit parmi l'ensemble des règles de propagation.

2- Activation des règles de l'ensemble de conflit.

3- Reprise du cycle jusqu'à l'obtention d'un ensemble de conflit vide.

Au cours des cycles, le moteur garantit qu'une règle n'est exécutée qu'une fois, évitant ainsi les bouclages possibles. Le moteur fonctionne ici par saturation, il s'arrête lorsque toutes les règles applicables ont été exécutées.

2.3 - Agent KP

La figure 2.6 suivante résume les différentes composantes d'un agent KP :

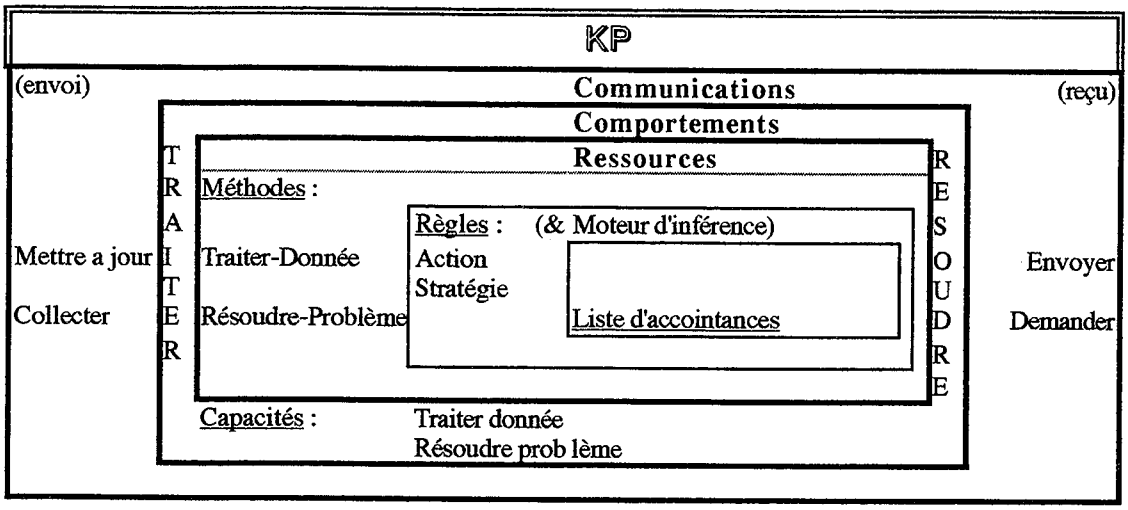

Figure 2.6 : Agent KP.

2.3.1 - Les ressources

Ce sont uniquement des règles pour manipuler les informations émanant des agents KS (valeurs d'attributs d'instances). Ces règles sont de deux types :

- règles d'action
- règles de stratégie pour contrôler la sélection des règles d'action.

Des procédures peuvent être appelées depuis les règles d'action.

2.3.2 - Les connaissances sur soi et les autres

Elles sont de deux sortes :

- statiques; on en distingue trois types :

-- contenues dans une liste d'accointances détenue par l'agent KP, c'est-à-dire les agents KS auxquels il est connecté.

-- les connaissances codées par l'utilisateur dans les règles de stratégie.

-- la possibilité d'accéder aux informations contenues dans un agent KS.

- dynamiques par l'intermédiaire des attributs systèmes des agents KS qui lui fournissent une trace des dernières actions entreprises.

2.3.3 - Les communications

Un agent KP ne reçoit que des messages d'informations,

- la requête "*Envoyer*" lui fournit une information à traiter,
- la requête "*Demander*" lui soumet un problème à résoudre.

La requête "Envoyer" est une communication asynchrone. La requête "Demander" est synchrone.

2.3.4 - Le contrôle

Il est présent sur trois niveaux :

- niveau ***comportement*** , les deux comportements "Traiter" et "Résoudre" sont simples, du type stimulus / réponse.
- niveau ***compétence*** , ce sont ses capacités à traiter une donnée et à résoudre un problème. Mais également toutes les capacités exprimées par l'utilisateur dans les règles.
- niveau ***moteur d'inférence*** , pour l'activation de ces règles un moteur d'inférence fonctionne en chaînage avant et arrière de la façon suivante :

Règles de stratégie :

1- Définition de l'ensemble de conflit parmi les règles de stratégie.

2- Application des règles de l'ensemble de conflit et obtention des restrictions.

3- Définition du sous ensemble de règles d'action.

4- Arrêt et paramétrage du moteur d'inférence avec le sous ensemble de règles d'action.

Règles d'action :

1- Définition de l'ensemble de conflit parmi les règles d'action retenues.

2- Application des règles de l'ensemble de conflit.

3- Arrêt.

3 - L'implantation

3.1 - Le matériel

Une première version de MAPS a été développée sur station APOLLO, ensuite d'autres fonctionnalités et évolution dans le langage et dans le traitement du parallélisme ont nécessitées un portage vers des stations de travail plus "rapides".

La dernière version de MAPS s'utilise sur une ou plusieurs stations de travail de type SUN 4 (SPARC II). Le système d'exploitation est celui des stations, à savoir SUN OS 4.1 compatible UNIX System V. Le développement a été entièrement réalisé en langage C++. Une interface graphique utilise les standards X Window et Open Look.

3.2 - le principe

Une dernière définition peut être donné à ce niveau :

MAPS a des agents de type "agents prédéfinis".
Ce sont des agents spécialisables à l'aide d'un langage de programmation.

Chaque agent est un processus UNIX pouvant être implanté sur une station de travail indépendamment les uns des autres. Il est entièrement décrit à l'écriture dans un fichier de ressource avec un langage de programmation proche du C++.

Des classes C++, (figure 2.7,) représentant les divers composants d'un agent et organisées en une hiérarchie servent par composition à créer une classe générique KS ou KP. On trouve les classes Common (méthodes communes à toutes les classes), Attribute (attributs d'un objet), Object (un objet d'un agent KS), Element (les objets d'un agent KS), Rule (les règles), Activity (les interpréteurs de règles), Engine, KS-Engine et KP-Engine (les moteurs d'inférence) et KS-Interface et KP-Interface (les protocoles de communication entre les agents). D'autres classes nécessaires au fonctionnement ont été implantées.

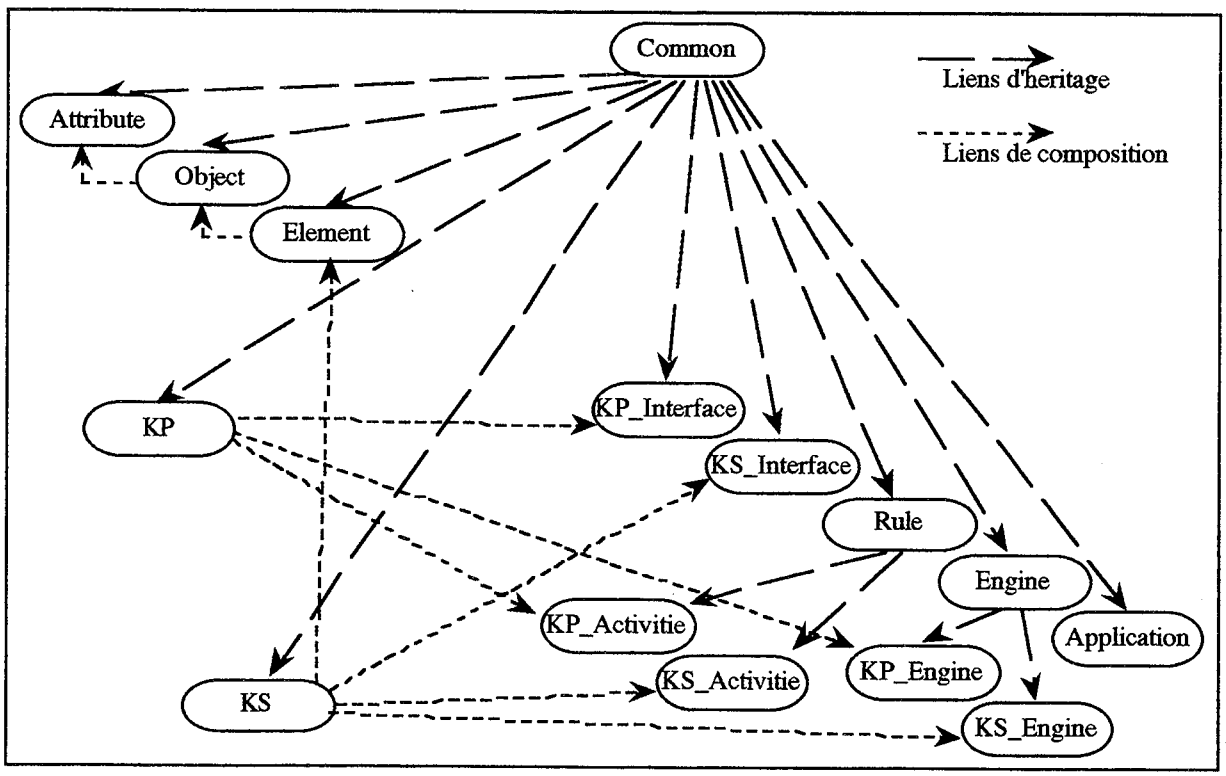

Figure 2.7 : Hiérarchie des classes C++.

Un précompilateur traduit les fichiers de ressources en autant de fichiers exécutables. Ce précompilateur a été développé avec les outils LEX et YACC disponibles sous UNIX, et, produit à partir des fichiers de ressources un ensemble de tableaux de dépendance pour les

divers composants de chaque agent et une arborescence pour chaque règle. C'est cet arbre qui sera parcouru à l'exécution par un interpréteur généré par le précompilateur.

Les agents sont créés à l'exécution par instanciation des classes génériques.

Chaque processus agent communique avec les autres par envoi de messages. Le mécanisme de communication interprocessus est le mécanisme des "sockets" de UNIX. Les messages sont stockés en file d'attente et traités un par un.

Un contrôleur d'application, un autre processus, est chargé, après compilation des agents, de lancer leur exécution et de faire le lien entre l'utilisateur et les agents par l'intermédiaire de l'interface, (figure 2.8).

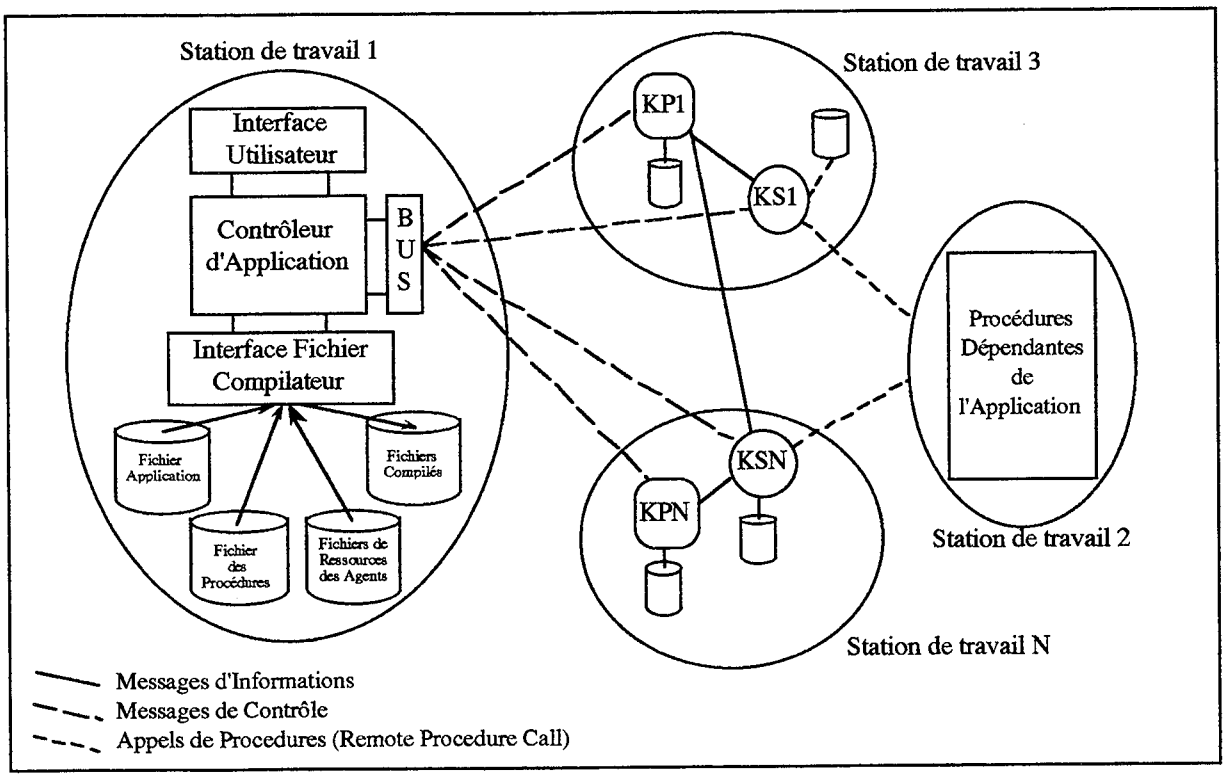

Figure 2.8 : Implantation d'une application MAPS.

L'interface permet de visualiser l'application et tout ou partie du contenu d'un agent.

Des procédures externes peuvent être utilisées. Un serveur permet de décrire ces procédures dans un fichier de ressource qui permettra lors de l'exécution de faire appel a ces procédures par un mécanisme d'appel de procédures à distance (RPC, Remote Procedure Call).

3.3 - Exemples

Nous donnons dans les figures 2.9 et 2.10 suivantes un exemple de chaque type d'agent pour illustrer les capacités de MAPS. Ces exemples sont schématiques et ne sont pas tirés d'application réelles. Le but est de donner un aperçu de la façon de développer le contenu des agents. La syntaxe est proche du langage C++.

Un agent KS est composé d'objets, eux mêmes composés d'attributs, d'une liste de connections et d'un ensemble de règles.

Un agent KP est composé d'une liste de connections et d'un ensemble de règles.

Dans les règles les accès se font sous forme de référence objet-attribut ou agent-objet-attribut.

Ces règles sont de la forme règle de production "IF prémisse THEN conclusion" , la partie conclusion ne s'applique que si la partie prémisse est vérifiée.

Un agent KS nommé KS1 composé de deux objets O1 et O2. Les attributs de ces objets peuvent être de type entier, flottant, chaîne de caractères ou liste. Cet agent est connecté a un agent KP1 de type KP (obligatoirement). Il est doté de trois règles, une de propagation, une de proposition et une d'interrogation.
Les règles illustrent les possibilités de manipulation des instances d'objets. Par défaut il s'agit de l'instance courante. On peut en créer (MAKE INSTANCE OF), lire (GET), écrire (SET), envoyer (SEND), demander (ASK)... On peut également tester l'état des attributs de contrôle (LAST PROPOSITION, LAST INTERROGATION, LAST RESULT).

Dans la règle de propagation, il y a test de cohérence (par exemple sur attributs utilisateur avec GET et sur attributs de contrôle) puis mise à jour de la base de connaissance.

Dans la règle de proposition une instance d'objet est envoyée à un agent (KP1 faisant partie des accointances).

Dans la règle d'interrogation c'est une information qui est demandée à un agent.

```
AGENT  KS1 : KS {

OBJECT O1 {
INT X;
LIST L1;
};

OBJECT O2 {
FLOAT F1;
INT Y;
};

CONNECTION  KP1;

RULE R1 : PROPAGATION {
IF
   (GET(object:O1,attribute:X) != 0) &&
   {
   FLOAT FF1;
   LIST LL1 = GET(object:O1,attribute:L1);
   FF1 = GET(object:O2,attribute:F1);
   } &&
   (LAST PROPOSITION OF (Object:O2) == "NO")
THEN
   INT I = MAKE INSTANCE OF (object:O2);
   & &
   SET(object:O1,attribute:X) WITH  I;
};

RULE R2 : PROPOSITION {
IF
   (GET(object:O1,attribute:X) == 0)
THEN
   SEND(object:O2) TO KP1
};
RULE R3 : INTERROGATION {
IF
   (GET(object:O2,attribute:F1) == 3.4) &&
   (LAST INTERROGATION OF (object:O2) == "YES")
THEN
   ASK(Object:O1,attribute:X) TO KP1
};
};
```

Figure 2.9 : Exemple d'agent KS.

```
AGENT  KP1 : KP {

CONNECTION  KS1;

RULE R1 : STRATEGY {
IF
   (GET(object:O2,attribute:Y) != 0)
THEN
   '(R2)
};

RULE R2 : ACTION {
IF
{
   INT   X = (GET(object:O1,attribute:X) IN agent:KS1);
   FLOAT Y = (GET(object:O2,attribute:F1) IN agent KS1);
}
THEN
   SET(object:O2,attribute:Y) WITH  3  IN KS1
};
```

Figure 2.10 : Exemple d'agent KP.

Dans un agent KP on retrouve le même principe d'écriture que pour un agent KS. Les types de règles sont Action et Stratégie. Dans une règle de stratégie il y a test dans la partie prémisse, et conclusion sur un nom de règle. Les règles de stratégie servent à sélectionner un ensemble de règles d'action.

Voilà brièvement énoncées les règles d'écriture de MAPS. Ceci conclut la présentation générale de MAPS qui nous servira de base pour évoluer vers l'environnement de programmation COALA, but que l'on s'est fixé. Mais auparavant et afin d'introduire les dites évolutions, nous allons énoncer les différents points à faire évoluer.

III - Discussion

Consécutivement à l'utilisation du système MAPS est apparu le besoin de faire évoluer certaines fonctionnalités aussi bien que le concept même de Multi-Agents utilisé. Notre volonté est de faire progresser MAPS vers un environnement de programmation plus complet dans lequel nous incluons de nouvelles possibilités pour aider au développement, et de nouvelles capacités devant améliorer l'efficacité d'un système Multi-Agents.

MAPS possède un certain nombre d'avantages mais aussi des inconvénients. Les avantages portent premièrement sur la distinction et le développement des connaissances au sein de deux types d'agents. Ce principe force un utilisateur à distinguer les actions des connaissances sur lesquelles elles portent. De plus les possibilités d'architecture autour de ces types d'agents sont étendues, groupes centrés tâches, centrés connaissances, traitement dirigé par les données, les buts. Deuxièmement le développement de ce principe sous formes d'objets et de règles dans un langage de programmation propre permet de rester proche de son application sans toutefois avoir à découvrir toutes les finesses du langage C++ sous-jacent.

L'expression du parallélisme à l'exécution des agents est important et correspond bien à l'idée de collaboration d'agents dans une société. Cette possibilité est mise en œuvre par l'utilisation des capacités multi-processus d'UNIX.

Les faiblesses de MAPS se situent notamment dans le manque de niveaux d'abstraction, la conception "à plat" d'une application, dans la granularité des agents qui est trop figée, dans le fait qu'il y a peu ou pas de possibilité de sortir du cadre fixé par les agents, et enfin qu'il n'y a pas d'évolution dynamique.

Un reproche souvent formulé à l'égard de MAPS est qu'une application est figée à sa conception, il n'y a pas en effet de création dynamique d'agent en cours d'exécution. Mais MAPS propose des gestions dynamiques des différentes instances d'objets. Ces instances peuvent être assimilées à des mondes qui grâce au parallélisme peuvent être traitées simultanément et évoluer jusqu'à une prise de décision sur l'opportunité et la validité de chacune des instances créées.

Une autre remarque est que le principe d'encapsulation est faussé dans la réalité. Le principe de communication KS-KP est basé sur le principe que les messages envoyés d'un agent à un autre provoque une "réflexion" de la part de l'agent receveur sur l'opportunité de la demande qui lui parvient. Cependant certains messages de la part d'un agent KP accèdent directement aux informations détenues par un agent KS. Une évolution serait d'attacher à toutes les réceptions de messages un comportement de l'agent receveur pour lui inculquer une capacité de réflexion en toute circonstance.

Mais c'est surtout sur le principe d'abstraction que se portera notre attention. La volonté de penser en termes d'agents nous a fait évoluer vers un langage d'agent appelé COALA (COoperative Agent LAnguage). Pour implanter cette possibilité, les classes de base du langage C++ sous-jacent du logiciel ont été entièrement repensées. De plus une réorganisation complète sous forme d'un environnement de programmation a été envisagée. Dans cette remise à niveau sontt inclues des évolutions devant améliorer le temps d'exécution, et le développement d'une interface graphique, la version actuelle de MAPS n'ayant plus d'interface, le portage lors du passage des stations APOLLO à SUN n'ayant pas été fait.

Chapitre 3 - Etude bibliographique

I - Introduction

L'objet du présent chapitre n'est pas de faire un parcours exhaustif des techniques informatiques disponibles, mais de découvrir quelles sont les possibilités pour faciliter, fiabiliser la tâche que l'on a entreprise. En cela nous verrons d'abord les apports du Génie Logiciel, car même si le contexte ne se prête pas à utiliser une stratégie complète de développement, on peut retirer une certaine rigueur menant à une qualité logicielle. Ensuite l'optique des langages objet sera explorée, pour deux raisons, la première est toujours dans un souci de qualité, la deuxième est que les objets sont à la base du travail du système Multi-Agents en développement. Enfin nous effectuerons un rapide panorama de systèmes voisins de notre approche, en décrivant trois plate-formes industrielles et une approche orientée langage d'acteurs. Le but étant de retenir dans chacun de ces systèmes une ou plusieurs particularités intéressante pour notre approche.

II - Génie Logiciel

Les informations qui ont servi à l'élaboration de ce paragraphe ont été retirées principalement de deux ouvrages, "Le Génie Logiciel" de Ian Sommerville [SOMMERVILLE 88] et "Conception et programmation par objets" de Bertrand Meyer [MEYER 90], ainsi que divers articles de la presse informatique.

1 - Introduction

Un système informatique est un ensemble composé de deux éléments principaux, une partie matérielle et une partie logicielle, plus communément appelés "hard" et "soft".

Le matériel a connu un développement important lié à l'évolution des techniques et des technologies. Ces évolutions ont été largement maîtrisées par le fait que les moyens utilisés relèvent de la physique, de la chimie, de la mécanique, bref de tout domaine où le terme "Génie" avait un sens. Le "Génie" désigne un ensemble de techniques appliquées à un domaine donné.

La partie logicielle des systèmes informatiques a nécessairement suivi , mais du fait de sa nouveauté en tant que technique n'a pas vu les mêmes concentrations d'efforts dans sa production.

Une caractéristique des logiciels par rapport à d'autre produits, c'est qu'un logiciel ne s'use pas : ainsi un composant qui ne remplit pas sa fonctionnalité ne pourra pas être remplacé par un composant identique obtenue par réplication. La maintenance n'est pas du même niveau d'activité comme nous le verrons, ce n'est pas de l'"entretien", mais relève plutôt de la correction d'erreurs, de l'amélioration de la réalisation et de l'augmentation des fonctionnalités.

Une autre constatation est celle de la part du logiciel dans le coût d'un système informatique. Celle-ci est estimé à 70 à 90% du coût total, d'où une attention importante à cette part. Toujours dans les coûts, durant le cycle de vie (nous préciserons plus loin ce que nous entendons par "cycle de vie") d'un logiciel on estime à 2% la part de conception, 20% pour l'écriture, 3% pour l'intégration et la mise en œuvre, c'est-à-dire en moyenne de 30 à 40% pour son développement, les 60 à 70% restants étant consacrés à la maintenance, d'où une attention accrue sur le développement pour réduire les interventions de maintenance. Autre résultat à propos de la maintenance, plus de 40% des activités sont des extensions et des modifications demandées par l'utilisateur, 17% sont pour la prise en compte des modifications de formats de données et 21% sont pour des corrections d'erreurs, d'où une modularité importante pour faciliter l'extensibilité et les interventions. Encore dans les coûts , le rapport va de 5 à 600 entre le prix de vente d'une ligne de code et son coût de production d'où une réutilisation maximum du code déjà écrit.

Pour pallier à tous ces inconvénients est apparu le Génie Logiciel. Le terme Génie Logiciel est apparu à la fin des années soixante et est devenu une "science" à part entière.

Le Génie Logiciel s'occupe de la conception et de l'écriture des programmes qui constituent le logiciel. Il regroupe l'ensemble des méthodes, concepts et outils utilisés pour la production industrielle des logiciels.

Son champ d'action va de la définition à la mise en œuvre d'outils , de techniques et de méthodes de conception, de fabrication et de documentation assistées par ordinateur, de simulation, de création de maquettes, de test automatique, de spécification et de contrôle de la qualité.

Son but est la maîtrise industrielle de l'abstrait et du complexe en vue d'une production massive de produits logiciels de qualité à un coût minimal.

Le Génie Logiciel s'est largement inspiré des méthodes industrielles pratiquées par d'autres "Génies". Mais deux différences essentielles apparaissent :

- premièrement chaque programme est unique, même dans le cas de ventes de multiples exemplaires d'un logiciel, chacun d'entre eux étant une réplique exacte de l'original. Le problème, en fait est celui de la production à grande échelle de logiciels très sophistiqués mais en seul exemplaire.

- deuxièmement tout au long de son développement, le logiciel n'a aucune apparence matérielle ou concrète, sinon sous la forme de textes. Cette dernière exigence en entraîne une autre qui est celle de la mesure de la qualité d'un logiciel. Ce sont des critères purement abstraits qui sont introduits ici.

2 - Le cycle de vie

Le cycle de vie d'un logiciel s'étend depuis les étapes de définition du produit jusqu'à son exploitation. On y retrouve grossièrement trois étapes, une étape de définition, une étape de réalisation et une troisième étape de suivi après mise en service. Les frontières ne sont pas toujours forcément aussi bien marquées suivant les subdivisions que l'on applique aux différentes étapes ci-dessus. Il y a des passages d'informations d'une étape à l'autre, une mention particulière pour l'étape de maintenance qui peut influer sur tous les niveaux. Voir figure 3.1.

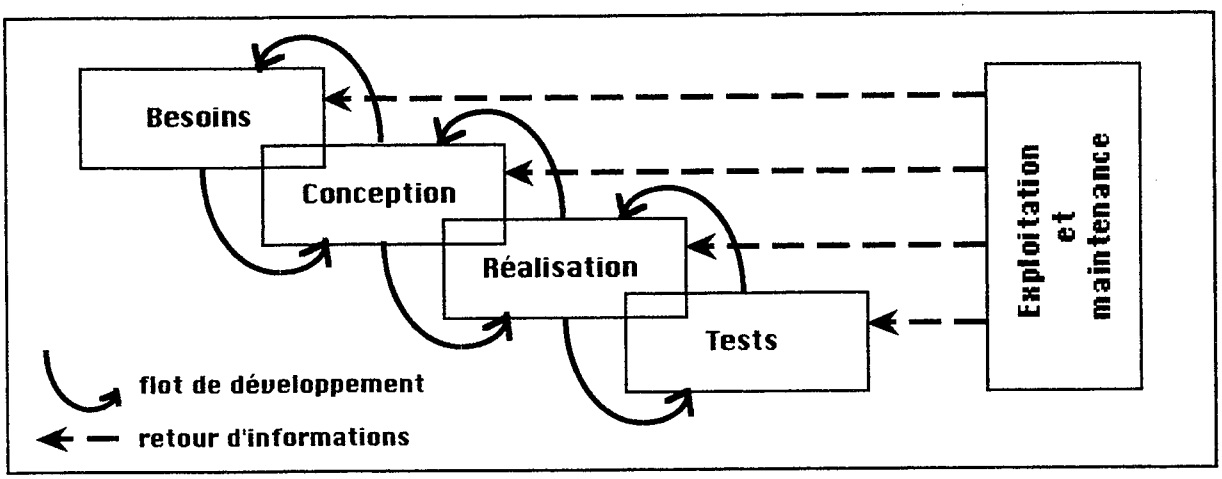

Figure 3.1 : Les étapes du cycle de vie d'un logiciel.

Dans la première étape de définition on retrouve fréquemment une **analyse et définition des besoins** ou un établissement de cahier des charges. On trouve plusieurs niveaux suivant la manière de procéder. Certains clients procèdent à l'établissement d'un cahier des charges rigoureux et détaillé, la spécification étant complètement détachée, d'autres effectuent une spécification conjointement avec le fournisseur, la limite entre définition des besoins et spécifications étant plus floue.

On arrive ainsi à une **spécification** qui par rapport à une définition des besoins est orientée vers le concepteur, c'est-à-dire qu'elle décrit ce que doit faire le système et quand. C'est le système vu par l'utilisateur, mais également les fonctionnalités à introduire par le concepteur pour parvenir à cet état. La spécification est établie par le fournisseur pour le concepteur. C'est parfois une tâche ardue, la séparation entre spécification et conception n'est pas toujours très nette. Etablir la spécification comme une représentation abstraite du processus de conception peut être difficile et peut sembler inopportune. Mais l'avantage de la spécification est de rester comme une référence à atteindre, un objectif.

Il y a plusieurs façons d'écrire une spécification, la première communément employée est d'utiliser un langage naturel, la deuxième est d'employer un langage de spécification formel avec une syntaxe et une sémantique propre. D'autres possibilités existent entre ces deux extrêmes consistant à utiliser un langage structuré avec quelques règles mais sans syntaxe et sémantique rigoureuses. L'utilisation de langage naturel est la plus compréhensible par un grand nombre d'utilisateurs mais elle reste de haut niveau et doit être très découpée en ce sens

que chaque notion utilisée doit être soigneusement détaillée, séparément les unes des autres. Moyennant ces quelques précautions le langage naturel reste clair.

Dans la deuxième phase de développement arrive la **conception** qui peut être décomposée suivant la complexité du projet en conception préliminaire et conception détaillée, suivi du codage. Toutes ces étapes peuvent parfois être confondues en une seule. La plupart du temps, à partir des spécifications on établit un découpage des tâches, et on répartit le travail entre équipes. Plusieurs possibilités s'offrent au concepteur, Conception Descendante, Ascendante, Orientée objets. Le plus souvent il s'agit de programmation modulaire.

A la suite vient la **validation**, étape pour détecter les éventuelles erreurs de conception. Le but est de montrer que la conception est correcte, en ce sens qu'elle traduit bien les intentions du concepteur et que le logiciel est valide, c'est-à-dire qu'il répond à la définition des besoins. Dans le premier cas on parle souvent de **vérification**. La vérification permet d'établir que le produit en cours de réalisation répond à la définition des besoins, la validation que les fonctionnalités du produit sont bien celles que veut le client. Schématiquement :

Vérification = "Est-ce-que nous construisons bien le produit ?

Validation = "Est-ce-que nous construisons le bon produit ?

L'opération de validation peut être répercutée tout au long du cycle de vie, au moyen de tests. Ceux-ci peuvent être présents sous forme de tests unitaires pour chaque procédure et sous forme de test d'intégration lorsque le système complet est assemblé. On retrouve également des tests de modules pour tester un certain nombre de fonctions qui coopèrent, de tests de sous-systèmes. Les tests sont réalisés par les programmeurs au fur et à mesure du développement du code.

Lorsque le système est complet et valide, il est proposé au client qui détermine au cours d'une recette si le logiciel correspond à ses souhaits. La recette est un ensemble de tests mais aussi de contrôle de tous les éléments, le plus souvent dans le contexte précis de travail du système. Les éléments correspondent à tout ce qui est nécessaire pour le fonctionnement du logiciel, c'est-à-dire code, documentation, commentaire, manuel de mise en œuvre, d'utilisation.

Le logiciel entre alors dans sa phase de suivi après mise en service, et de **maintenance**. La maintenance évoquée ici est de trois types. Elle peut être perfective, pour effectuer des modifications à la demande des utilisateurs, c'est la plus fréquente, adaptative pour adapter le logiciel aux changements de son environnement et corrective pour corriger les erreurs.

La documentation d'un logiciel est de deux types, premièrement pour l'utilisateur, deuxièmement pour la maintenance. Généralement on a : un manuel de l'utilisateur, et un manuel d'installation et de mise en œuvre, et pour la maintenance, les documents relatifs à la réalisation du système. Ceux-ci vont des spécifications aux tests de validation en passant par le code commenté.

Voilà rapidement évoqué un cycle de vie logiciel. Les éléments cités se retrouvent souvent mais dépendent fortement des types de logiciels, de leurs tailles et des méthodes employées. Il faut néanmoins noter qu'une certaine rigueur parfois contraignante est indispensable pour produire un logiciel satisfaisant à des normes de qualité apte à une production industrielle.

3 - Critères de qualité

On trouvera en annexe D une définition des termes énoncés dans ce paragraphe.

Le but du génie logiciel est d'aider à produire du logiciel de qualité. Mais un "bon" cycle de vie ne va pas sans une forte imprégnation de qualité. Ainsi, l'un ne va pas sans l'autre, et le souci de toute personne désireuse de mener à bien un travail logiciel de qualité est de se situer à l'intersection de ces deux notions.

Selon Meyer deux types de facteurs de qualité sont à considérer. Les facteurs externes, ce sont ceux vus par l'utilisateur du produit et les facteurs internes, ce sont ceux vus par les informaticiens "générateurs" du produit. En final seuls les facteurs externes comptent mais ils ne peuvent être perçus sans la prise en compte des facteurs internes.

Les plus importants de ces facteurs sont :

Validité, Robustesse, Fiabilité, Extensibilité, Réutilisabilité, Compatibilité, Efficacité, Portabilté, Vérifiabilité, Intégrité, Facilité d'utilisation.

Ces qualités là sont nécessaires dès que l'on désire qu'un logiciel non seulement réponde à ses spécifications mais puisse également recouvrir sa zone de vie appelée maintenance. En effet comme nous l'avons vu les activités de maintenance gèrent plus d'évolutions que de corrections. Il est donc indispensable qu'un minimum de **modularité** soit présente dans un logiciel, cette modularité n'étant possible que si l'on prend en compte les paramètres évoqués ci-dessus.

Cette modularité nécessite les critères suivants :

Décomposabilité modulaire, Composabilité modulaire, Compréhensibilité modulaire, Continuité modulaire, Protection modulaire.

De ces critères B. Meyer déduit cinq principes :

Unités modulaires linguistiques, Peu d'interfaces, Petites interfaces, Interfaces explicites, Masquage de l'information.

Mais la difficulté dans les critères de qualité sont dans la métrique qu'il faut leur appliquer. La difficulté est dans l'abstraction du logiciel qui ne peut être "vérifié" qu'en phase finale. Pour pallier à ces difficultés des outils inclus dans des méthodes de développement ont été créés qui permettent de modéliser et représenter les concepts les plus abstraits. Ces outils sont souvent assez importants dans leurs investissements et ne sont présents qu'au sein de grosses sociétés de logiciel.

4 - Méthodes et Ateliers de Génie Logiciel

Afin de parvenir à la qualité de logiciel recherché, des méthodes de conception, de spécification et de conduite de projet sont apparues. Ces méthodes font souvent l'objet de dépôt de brevet et ne sont plus alors accessibles au public le plus large. On se retrouve fréquemment avec le schéma suivant : chaque société a sa méthode interne qu'elle conserve jalousement et qui doit être apprise par chaque nouveau venu dans la société.

Néanmoins on retrouve des outils fréquemment utilisés au niveau de l'aide à la programmation, qui vont des éditeurs, compilateurs, déboggeurs à des outils de gestion de configuration par exemple MAKE et SCCS sous UNIX. Lorsque un ensemble d'outils est regroupé au sein d'un même système on parle d'environnement de programmation ou d'atelier de Génie Logiciel.

Un atelier de Génie Logiciel est un système informatique dédié aux seules tâches de développement et de maintenance du logiciel (à l'exclusion des tâches d'exploitation). Un atelier concentre l'ensemble des outils d'analyse, de conception, de programmation et de test bien sûr mais aussi outils aidant à la plannification et à la gestion des projets logiciels, à l'assurance et au contrôle de la qualité ainsi qu'au suivi après mise en service.

Parmi les environnements nous pouvons citer UNIX qui, sans être un atelier de Génie Logiciel, regroupe un ensemble d'outils d'aide à la mise au développement, et a l'avantage de fournir une interface utilisateur identique à celle proposée lors de l'utilisation du produit. Néanmoins il n'intègre pas de méthodes de spécification particulière. Parmi les méthodes les plus connues nous pouvons nommer MERISE, SART ou SADT. Par exemple SADT (Structured Analysis and Design Technique) permet de modéliser un problème sous forme d'une boîte noire qui est ensuite décomposée en sous-boîtes (sous-problème ou sous système). Les boîtes représentant un traitement sont alors reliées entre elles par des "objets" (ou données), pouvant être des commandes, des entrées, des sorties, ... La même décomposition est appliquée à chaque sous-boîte jusqu'à l'obtention du niveau de détail requis pour que le problème soit parfaitement maîtrisé. Parmi les méthodes orientées objets on trouve HOOD (Hierarchical Object Oriented Design) [HEITZ 91] qui intervient après les activités d'analyse des besoins logiciels et couvre les activités de définition d'architecture, conception détaillée et codage. HOOD est basé sur des formalismes graphiques et textuels qui s'utilisent de manières complémentaires :

- la notation graphique permet l'expression semi-formelle de la structure statique d'une solution de manière claire et accessible aux non-spécialistes.
- la notation textuelle permet de compléter la solution graphique d'abord au moyen de squelette de description d'objets, puis avec des notations d'expression du contrôle et de description de programme, enfin par l'utilisation d'un langage cible.

5 - Conclusion

Le développement de "label qualité" est devenu important en matière de production de logiciel, une qualité de conception ne va pas sans appliquer une certaine méthodologie dans la conception d'un logiciel. Même si l'on ne dispose pas d'outil spécifique d'aide aux spécifications, conception, suivi de projet il est essentiel de garder à l'esprit les critères de qualité présentés précédemment pour produire du logiciel fiable, maintenable et évolutif.

III - Objet, Acteur, Langages

1 - Introduction

Ce Chapitre traite de l'approche objet qui a donné lieu à de nombreux articles et livres. L'essentiel de ce chapitre a été retiré de deux ouvrages, "Les Langages à Objets" de G.Masini, A.Napoli, D.Colnet, D.Léonard, K.Tombre [MASINI 90] et "Conception et Programmation par Objets" de B.Meyer [MEYER 90]. Le premier présente un état et une classification des différents langages existants, le second est une approche Génie Logiciel du concept Objet.

Le développement de l'approche Objet, de plus en plus utilisée (notamment avec l'emploi de C++ et SMALLTALK), est issue d'une constatation et de nouvelles exigences en matière de programmation.

Dans tous les cas il s'avère qu'un programme informatique est une suite d'actions sur une structure de données. La technique objet dans son principe consiste à associer au sein d'un même objet les données et les procédures qui doivent manipuler les dites données, ce qui évite un complet bouleversement du programme lorsque les données évoluent dans leur structuration. Ainsi les objets ne sont vus qu'à travers une interface qui reste inchangée, les données sous-jacentes pouvant alors évoluer librement, on parle alors du principe d'encapsulation.

De plus les exigences du Génie Logiciel telles que modularité, réutilisabilité s'accordent bien avec le principe d'objet. En effet une structuration d'un logiciel sous forme d'objets conduit en outre à la modularité recherchée. La vision d'un objet à travers son interface permet de le réutiliser aisément, on ne se soucie pas de la façon dont il est construit. Cette réutilisation est facilitée par les possibilités d'héritage des objets entre eux. De plus l'encapsulation permet d'obtenir une abstraction supplémentaire sur les données ce qui induit une meilleure lisibilité et compréhensibilité. La programmation est passée de "dirigée par les traitements" à "dirigée par les données".

2 - Objet

2.1 - Définitions

Des différentes définitions données par les dictionnaires (voir annexe F) apparaît le coté tangible et d'existence d'un objet vis-à-vis d'une activité humaine, c'est-à-dire que l'on trouve les notions de perception et de manipulation.

En informatique, la notion d'objet reprend effectivement ces éléments, en définissant un objet comme un type de données, donc comme une chose concrète et tangible, les données, et perceptible en le classifiant, le type. Cet objet là est le résultat d'une activité humaine, l'abstraction de données, pour rendre ces données accessibles à un traitement informatique, il s'agit en quelque sorte d'un prétraitement. Un objet n'est donc pas une chose stérile mais un élément de base servant à la réflexion.

De façon plus technique, un objet est un ensemble de données et un ensemble de procédures (Voir figure 3.2). Autour de ce principe et suivant les orientations prises par les différents concepteurs de langages on retrouve les éléments suivants : l'abstraction de données, l'héritage, l'envoi de messages mais aussi la modularité, l'homogénéité et le parallélisme.

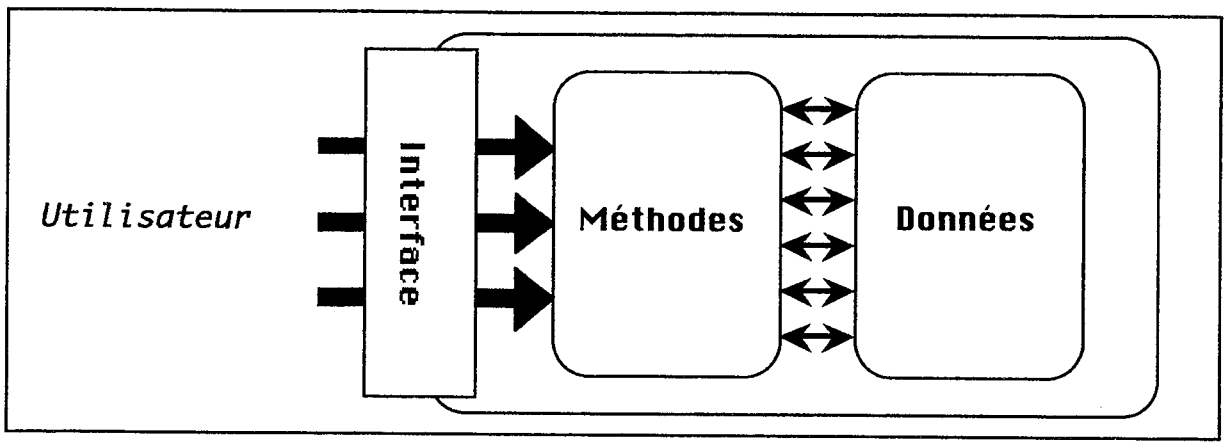

Figure 3.2 : Objet.

2.2 - L'abstraction de données

L'encapsulation conduit à ne voir un objet qu'à travers une interface qui cache l'organisation interne, les données ne sont manipulables que par les actions permises par l'interface. On a donc une abstraction sur les données. Ceci induit une protection possible des données en limitant ou en autorisant que certains types d'actions cohérentes avec l'usage que l'on veut établir pour cet objet.

Ces abstractions et protections sont réalisées diversement suivant les langages choisis, soit pas de protection, visibilité complète de tous les objets, soit une complète abstraction, ou

des étapes intermédiaires possibles en permettant à certains types d'objets de s'accéder mutuellement.

Pour le coté dynamique, les méthodes d'un objet permettent de définir un comportement. Celui-ci est mis en oeuvre de façons différentes suivant le type langage utilisé. Dans les langages de classes les méthodes sont accédées par des sélecteurs (couramment le nom de la méthode), dans les langages d'acteurs les méthodes, on parle ici de "script", sont déclenchées à travers un filtre (ensemble de sélecteur, un même sélecteur pouvant sélectionner des méthodes différentes suivant l'instant où se produit l'action), dans les langages de frames ce sont les opérations d'accès (lecture-écriture) sur les données qui déterminent un comportement "réflexes" (vérification, permission par exemple).

2.3 - L'héritage

L'héritage est une caractéristique importante des langages à objets. Cette possibilité est axée sur le fait qu'un objet peut être vu comme un type de base, qui peut être générateur d'objets identiques. Plusieurs principes sont utilisés suivant le type de langage. Dans les langages de classes, une classe est un moule que l'on peut spécialiser pour créer d'autre classes. Dans les langages de frames, un frame est un prototype et l'on peut créer un objet par copie différentielle de ce prototype. Dans les langages d'acteurs il n'y a pas vraiment d'héritage, le principe de délégation pouvant simuler l'héritage.

Le principe d'héritage est dans certains langages poussé jusqu'à la généricité. Avec les classes par exemple on obtient des classes dont le rôle est de générer des classes. De même l'héritage peut être multiple et permet une réutilisation maximum en créant des objets par héritage des propriétés de plusieurs autres.

2.4 - L'envoi de messages

Dans le monde objet, le principe de communication entre objets est l'envoi de messages. Plusieurs niveaux de complexité ont étés adoptés dans différents langages. Depuis un appel de procédure lors d'une exécution où les communications sont synchrones, jusqu'aux dialogues asynchrones avec les acteurs.

2.5 - La modularité

Un objet pouvant être décrit par spécialisation ou différenciation suivant les principes énoncés ci-dessus, une modularité est ainsi possible qui produit une réutilisation avec par exemple la constitution d'une bibliothèque. Dans un environnement particulier on adoptera en général un modèle d'objet adéquat qui par sa structure et ses moyens de dialogue produisent une homogénéité qui facilite le travail de conception.

3 - Langages

Des différentes manières d'envisager un objet, en accentuant tel ou tel point de son principe, et en fonction des différentes utilisations, sont nés divers types de langages. Dans le paragraphe qui suit nous allons voir une classification de ces langages en fonction de leurs caractéristiques, ceci nous permettra de nommer les principaux.

Une classification à partir des trois différents points de vue d'un objet donnée par Ferber [FERBER 87] peut être élaborée. Ce sont :

- le point de vue structurel : un objet est un type de donnée lié à un modèle. Il sera à la base des langages de classes.
- le point de vue conceptuel : un objet est une unité de connaissance. Il générera les langages de frames.
- le point de vue acteur : un objet est une entité autonome et active. Il donnera naissance aux langages d'acteurs.

De différentes constatations, est apparu par "mélange" des précédents concepts les langages hybrides.

3.1 - Les langages de Classes

Principes

Une classe est une implantation d'un type abstrait de données. Les objets sont générés par instanciation de cette classe. La classe va décrire une famille d'objets de même structure et même comportement.

Dans une classe on aura donc la définition de la partie statique : les données, champs ayant une valeur et la définition de la partie dynamique : les méthodes représentant le comportement des objets de la classe. Ces méthodes opèrent sur les seuls champs de l'objet. Les objets sont des instances particulières d'une classe. Les méthodes restent dans la classe car elles ne changent pas d'un objet à l'autre pour une même classe, les champs étant également dans la classe mais les instances de ces champs appartenant aux objets générés.

Les classes permettent un partage des connaissances par le mécanisme d'héritage. Des connaissances générales peuvent être mises dans une classe qui servira ensuite par spécialisation en sous-classe d'obtenir des classes plus spécifiques. Ici deux possibilités, spécialiser par enrichissement, en améliorant une sous-classe en lui apportant des champs et méthodes supplémentaires ou spécialiser par substitution, on donne un nouvelle définition dans une sous-classe à une méthode héritée. Ces possibilités permettent de créer des classes dont le rôle est de générer des classes, c'est la notion de métaclasse. Les classes sont alors des objets instances de métaclasses.

Pour communiquer les objets s'envoient des messages. Ceux-ci doivent contenir l'objet receveur, le sélecteur à activer c'est-à-dire la méthode utilisée, et les arguments à fournir. Dans la plupart des langages de classes, les envois de messages sont des appels de procédures.

Quelques langages

L'évolution des langages de classes est issue de deux courants SIMULA et SMALLTALK. Ce sont les premiers à avoir utilisé les concepts de classes et d'héritage, SMALLTALK héritier de SIMULA ayant en plus la notion d'envoi de messages.

SMALLTALK introduit le concept de métaclasse qui sera affiné dans LOOPS puis encore amélioré avec OBJVLISP. Ce dernier a complètement unifié le principe de métaclasse, classe et instance. Ainsi toute classe est un objet à part entière instance d'une autre classe. Au départ une classe CLASS instance d'elle même est à la racine de l'arbre d'instanciation, et une classe OBJET instance de classe à la racine du graphe d'héritage.Un bootstrap permet d'initialiser le système. De plus la sémantique opérationnelle est exprimé en LISP, langage puissant de manipulation de liste. Dans le même esprit le langage CLOS est une extension objet de COMMON LISP.

Le principe d'encapsulation est à la base de langage objet comme MODULA 2, CLU et surtout ADA qui possède en plus la généricité, la surcharge et le parallélisme. ADA est maintenant très répandu. L'encapsulation est résumé dans des paquetages comme ensemble de constantes, de routines et de variables. C'est un langage statique très fortement typés qui ne possède pas à l'heure actuelle de possibilité d'héritage.

SIMULA est à la base de langage à objets dite de l'école scandinave. Ces langages ont tous un typage statique et sont tous compilés. On retrouve ici C++ et EIFFEL.

C++ est une extension de C, et étant implanté couramment sur système d'exploitation UNIX, ce langage est certainement le plus répandu dans le monde industriel. Dans C++ une classe est un ensemble de données pouvant être accessible ou inaccessible à partir d'autre objets à divers niveaux de protection (notion d'amis), et un ensemble de méthodes pour accéder ces données. Toute classe possède un constructeur pour créer ses instances. L'héritage simple ou multiple est possible ainsi que la surcharge de toute fonction ou opérateur prédéfinis ou non (fonctions virtuelles). De plus une version récente permet de créer des classes génériques c'est-à-dire paramétrées par un type. Son défaut est que le niveau de protection maximum est la classe et non pas l'instance de cette classe et que ayant gardé sa compatibilité avec C, C++ est réservé à des programmeurs avertis. Cependant cette compatibilité et ses accointances avec UNIX l'ont beaucoup répandu et de très nombreuses bibliothèques existent, l'environnement de programmation couramment utilisé facilitent son succès.

EIFFEL conçu par B. Meyer est très attaché aux critères de qualité du Génie Logiciel. Il offre également un cadre de programmation rigoureux. C'est un langage et un système industriel à typage statique avec possibilité d'héritage multiple. On peut définir des classes génériques et l'insertion d'assertion avec pré-condition et post-condition est possible dans un programme permettant des vérifications supplémentaires optionnelles en cours de développement, puis éventuellement supprimées à l'exécution. EIFFEL est un langage attrayant dans son utilisation simple qui possède un environnement de programmation complet.

D'autre langage de classe comme CEYX, HYBRID, FLAVORS, OBJECTIVE C sont moins répandus ou pas enseignés et ont des orientations différentes suivant leur typage, statique ou dynamique, leur niveau de protection des données et leur niveau d'héritage.

Le principe de base des classes est très répandu et a donné naissance à la majorité des langages à objets.

3.2 - Les langages de Frames

Principes

Les langages de frames sont orientés sur la représentation des connaissances et ont pour origine les réseaux sémantiques. Ces langages ont été exploités notamment en Intelligence Artificielle. C'est dans ce domaine que les recherches sont les plus anciennes, les travaux de Minsky et Hewitt datent de la fin des années 60. Des systèmes comme DENDRAL et MYCIN ont été crées dans les années 70 pour résoudre des problèmes dans des domaines bien précis. C'est à cette époque qu'est apparue la dualité entre déclaratif et procédural. Une connaissance peut être déclarative, ce sont des faits sans caractère opératoire et peuvent être jugés comme tels, ou la connaissance est procédurale, elle est alors pertinente qu'après une exécution.

Une des méthodes les plus employées est celle des prototypes. Un prototype est une référence pour une catégorie, c'est un représentant type et les autres éléments de cette catégorie seront décrits par rapport à lui par spécialisation. Il est créé alors des liens d'héritage "une sorte de" et des liens d'instanciation "est un". Toute création d'objets se fera par copie différentielle du prototype. Les liens "sorte de" et "est un" se confondent, tout objet peut être générateur de nouveaux objets.

Les frames sont des prototypes organisés en une hiérarchie d'héritage. Plus précisément un frame est une structure de données qui représente un objet typique. Cette structure est composée d'attributs décrivant les différentes propriétés du concept représenté. Un attribut étant à son tour représenté par un certain nombre de facettes possédant des valeurs. Ces facettes peuvent être de deux types, déclaratives qui définissent le type de l'attribut, sa valeur et sa valeur par défaut, ou procédurales pour spécifier des comportements locaux aux attributs. Si les facettes déclaratives sont les valeurs retournées par une opération de lecture sur un attribut, les facettes procédurales mettent en jeu des opérations plus complexes. Ces opérations sont plutôt des réflexes déclenchés lors d'accès à des attributs et qui permettent des vérifications, modifications, création de valeurs de cet attribut.

Quelques langages

KRS n'est pas un langage spécifiquement de frames mais a beaucoup oeuvré dans la représentation des connaissances. Il en est de même de KL-ONE. KRL quant à lui est à la fois un langage de programmation et un langage de représentation des connaissances. FRL est un langage simple présenté comme une bibliothèque pour définir des frames. SHIRKA s'appuie sur des schémas de deux types, schéma de classe et schéma d'instance. L'originalité est

également dans une facette particulière de filtrage qui permet une exploration systématique d'un certain nombre de facettes.

Les langages de frames sont typiquement des langages de représentation des connaissances. La base est une structuration sous forme d'objets c'est-à-dire encapsulation de données et de procédures, les procédures sont simplement des réflexes attachés aux données ou attributs, qui font office de comportements.

3.3 - Les langages d'Acteurs

Principes

Carl HEWITT [HEWITT 73] est à l'origine du modèle d'acteur. Les acteurs sont des objets actifs, autonomes et pouvant communiquer librement entre eux. La notion d'acteur met en évidence le principe d'expert et de société. Cet objet possède des données locales et un comportement défini par un script qui procède par filtrage des messages reçus. Un acteur est connu par le monde extérieur par son intention, c'est le contrat dont il est responsable. Il y a donc abstraction. Les acteurs communiquent par envoi de message, c'est le seul type d'événement qui peut se produire. Chaque message est lui-même un acteur et contient une continuation, c'est-à-dire l'acteur auquel doit être transmis le résultat du message, permettant ainsi des communications asynchrones. De plus aucune connaissance n'étant globale le parallélisme entre acteurs en est facilité. Chaque acteur a son propre comportement, le comportement global du système étant difficile a appréhender.

Quelques langages

De la description de Carl HEWITT est né PLASMA puis ACT1 et 2. PLASMA a ajouté au modèle d'acteur la notion de "packager", regroupement de plusieurs acteurs au script identique qui servent de modèle pour créer d'autres acteurs. ACT1 a développé la notion de délégation comme un mode d'héritage. La délégation est le fait qu'un acteur connaît un autre acteur auquel il délégue les messages qu'il ne sait pas traiter, cette communication est asynchrone. Pour chaque acteur ce principe est dynamique, il peut en effet choisir dynamiquement de nouveaux acteurs pour déléguer. Ceci est mieux que dans les classes qui ont un graphe d'héritage figé.

Gul AGHA a fait évoluer ce modèle d'acteur en introduisant le principe de changement de comportement au lieu de changement d'état. Un acteur traite des messages stockés dans une boîte aux lettres par ordre d'arrivée, un comportement pour un message doit spécifier un comportement de remplacement pour le prochain message. Les langages PRACT et ACORE ont été développés dans ce principe.

Ce pur concept d'acteur a été repris mais de façon plus pragmatique par de langages comme ABCL/1 basé sur LISP, et HYBRID. Ce dernier allie langage de classes et d'acteur au

sein d'objets actifs. Le parallélisme est traité de façon intéressante par regroupement d'un ensemble d'acteur en un seul processus. ces derniers sont à la charge du programmeur qui peut ainsi disposer d'une granularité variable. ACTALK allie aussi classes et acteurs. D'autres langages allient programmation logique et les acteurs comme Concurrent PROLOG et PARLOG.

Les langages d'acteurs sont souvent des langages interprétés. Leur grande orientation est dans l'expression du parallélisme au sein d'entités : les Acteurs. L'abstraction des données et l'encapsulation reste prépondérantes.

3.4 - Les langages hybrides

A l'origine une constatation : aucun formalisme de représentation des connaissances n'est universel, chaque formalisme n'est bien adapté qu'à un certain type de connaissances.

Les langages hybrides sont nés de la nécessité d'intégrer divers styles de programmation. L'Intelligence Artificielle qui manipule des connaissances hétérogènes a besoin de ces divers formalismes et a souvent développé des langages adaptés à son domaine de prédilection.

On retrouve ainsi des langages comme LOOPS basé sur la programmation objet, les classes et la programmation logique LISP, YAFOOL basé sur les prototypes ou frames, OBJLOG est basé sur des frames et le langage Prolog, ART et KEE sont utilisés dans l'industrie et sont à base de frames. Une mention particulière est faite a MERING [FERBER 89] qui est à l'origine de nombreux langages (LOOPS, LORE, YAFOOL) et bien qu'expérimental a permis avec ses quatre versions de valider des techniques de représentation des connaissances.

Les langages hybrides sont donc souvent des exigences de l'Intelligence Artificielle pour intégrer diverses possibilités, pour les adapter à un besoin spécifique.

4- Conclusion

Dans ce paragraphe nous sommes partis d'une notion importante en informatique: l'Objet. Puis après analyse du contenu de ces objets, de ses différents aspects et des différentes orientations voulues par les utilisateurs, nous sommes arrivés à la découverte des langages crées à cet effet.

Parmi les techniques utilisées, les langages de classes se révèlent plus pragmatiques et adaptés à réaliser des implantations variées, alors que les langages de frames sont orientés vers la représentation des connaissances. Les langages d'acteurs, de par leur dynamique sont adaptés à l'expression de problèmes parallèles. Mais pour l'Intelligence Artificielle et les problèmes de représentation des connaissances les langages hybrides sont souvent les plus adéquats.

Voici maintenant, en guise de synthèse, quelques questions que l'on peut se poser en matière d'Objet et les réponses à ces questions :

Pourquoi les objets ?

Pour la modularité, l'abstraction : c'est l'aspect réutilisation du Génie Logiciel, et meilleur "souci de réalisme", c'est l'aspect pratique de la programmation.

Comment les manipuler ?

A partir d'un langage à objets, langage de classes, langage de frames, langage d'acteurs et/ou langage hybride

Lequel choisir ?

En fonction du domaine de l'application. par exemple pour un développement de logiciel "standard" un langage de classes comme C++, pour une application d'Intelligence Artificielle ou la Représentation des Connaissances un langage de frames ...

Pourquoi passer d'objets à acteurs ?

Lorsqu'un besoin d'autonomie, de parallélisme d'exécution est nécessaire de par l'application. Et/ou besoin d'un système ouvert.

... Et pour aller plus "loin" ?

Passer des acteurs aux agents. Très schématiquement, ajouter de l'"intelligence" à des acteurs.

IV - Systèmes voisins de notre approche

1 - Introduction

Le but de ce paragraphe n'est pas de faire une étude exhaustive de systèmes existants, mais de parcourir ceux dont la "philosophie" est proche de celui que nous devons développer. Pour cela nous tâcherons de mettre en avant les caractéristiques qui nous semblent intéressantes voir essentielles à la suite de notre étude.

Parmi ces systèmes appelés souvent plates-formes nous pouvons citer MACE [GASSER 87], MAGES [BOURON 91], RATMAN [BURCKERT 91], ABE [HAYES-ROTH 91], ARCHON [LOINGTIER 92], CONSTRUCT [DAVID 92].

MACE est un système expérimental pour construire et simuler des systèmes d'Intelligence Artificielle Distribuée d'approches différentes. Il permet des architectures variées (granularité d'agent cognitif et réactif) et divers modèles de résolution (tableaux noirs, envoi de messages).

MAGES est une plate-forme générique de systèmes d'Intelligence Artificielle Distribuée. Un de ses objectifs est de rendre possible la composition de réseaux d'agents hétérogènes qui puissent communiquer entre eux.

RATMAN est une plate-forme pour la définition et le test d'agents rationnels dans un environnement Multi-Agents.

Mais c'est avec ABE, ARCHON et CONSTRUCT dont la connotation industrielle est la plus forte que nous allons nous étendre. Ces plates-formes bien qu'ayant des buts différents et n'étant pas classées purement en Intelligence Artificielle Distribuée, ont des caractéristiques nous intéressant.

Un autre aspect, important dans notre étude, est celui du langage. Si le concept d'objet est maintenant bien répandu en informatique et permet une focalisation en termes de données, il manque la dynamique qui a été introduite dans les langages d'acteurs, plus récents en termes d'implantation que les langages à objets. Dans cette vision nous aborderons PLASMA II récemment développé à l'université de Toulouse.

2 - Les plates-formes

2.1 - Le projet ARCHON

Le cadre

Dans le domaine de l'Intelligence Artificielle Distribuée deux types de plates-formes ont été construites, celles qui s'appliquent à un problème particulier et celles qui se veulent plus générales. Le constat est que soit leur trop grande amplitude dans la résolution de problèmes induit un manque de puissance, soit leur spécificité les réserve à un domaine trop étroit.

C'est pourquoi un des buts d'**ARCHON** (**AR**chitecture for **C**ooperative **H**eterogenous **ON**-line Systems) est de donner un cadre qui permette d'être à la fois général tout en gardant assez de puissance pour s'appliquer à des problèmes industriels réels.

Le principe

Le projet ARCHON propose une architecture distribuée, une plate-forme logiciel et une méthodologie d'intégration de logiciels pré-existants. L'objectif est d'une part d'améliorer la résolution de problème, en facilitant le partage des connaissances disparates, en améliorant le dialogue entre l'opérateur et les systèmes en coopération, d'autre part d'augmenter la fiabilité, tout en gardant à chaque système leur indépendance de développement et leur autonomie [LOINGTIER 92].

Le principe repose sur une encapsulation des "systèmes intelligents" (ce sont les systèmes autonomes dans leurs domaines, à faire coopérer) dans une couche permettant un dialogue de même niveau pour former des Agents.

Chaque système encapsulé dans une couche ARCHON formera un agent. Chaque agent aura un comportement individuel, typiquement l'activité du système encapsulé, et un rôle de membre actif d'une communauté, apporté par la couche ARCHON [JENNINGS 91].

Cette couche comprend (voir figure 3.3) :

- au plus bas niveau, un **moniteur** qui filtre et traite les informations en entrée/sortie du système local,

- au plus haut niveau, un **planificateur** qui gère et planifie les interactions avec les autres agents,

- un **module de communication haut niveau** pour le dialogue entre les agents,
- un **module de gestion de l'information** qui permettra de modéliser les autres agents (ce sont les accointances) et de fournir un modèle de soi.

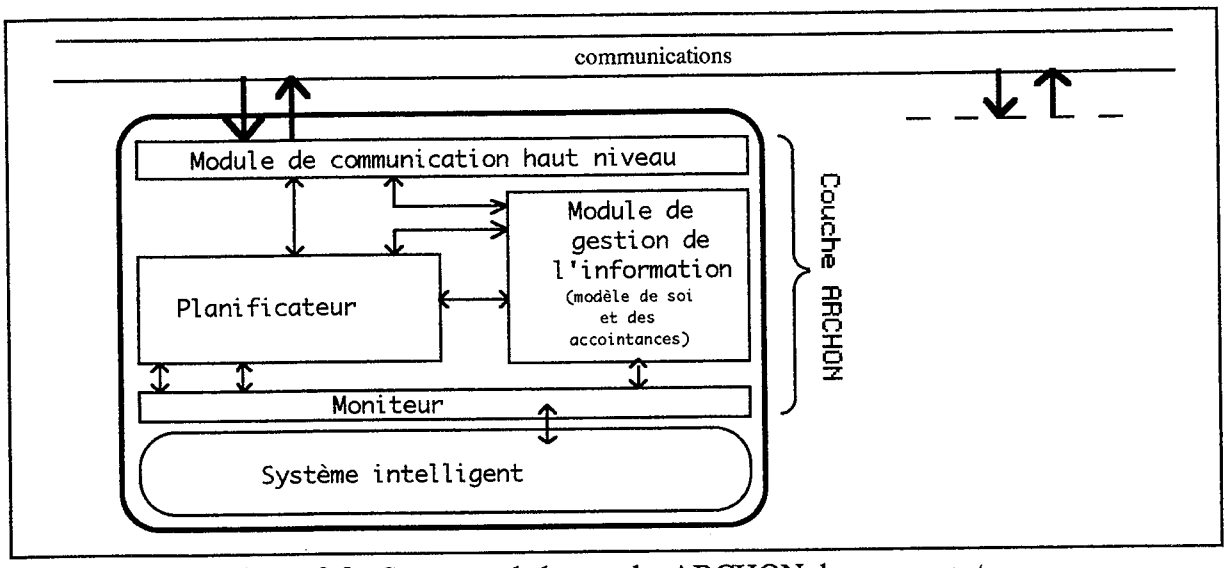

Figure 3.3 : Structure de la couche ARCHON dans un agent.

La combinaison d'un système intelligent et de sa couche ARCHON, constitue un agent situé à la frontière entre agent cognitif et réactif : réactif car certaines des actions sont guidées par les événements provenant d'autres agents, et cognitif, car un agent peut manipuler une connaissance de soi et une partie de connaissance sur les autres, ainsi qu'une notion d'intentionnalité laissé au planificateur.

L'ensemble des outils ARCHON est intégré dans un atelier de développement. Parmi ces outils un langage d'acteur **ALAN** (Actor **LAN**guage) a été développé. Il permet de définir des événements et les actions associées comme étant des ordres partiels sur ces événements. La communication se fait par envoi de message; les messages sont filtrés dans chaque agent. Un système de gestion de base de données utilisé pour la transformation automatique de données (pour contribuer à la prise en compte de systèmes hétérogènes) a été développé suivant un modèle objet.

Du point de vue matériel, le logiciel ARCHON s'exécute sur des stations de travail UNIX reliées par un réseau Ethernet. Les communications interprocessus s'effectuent grâce au mécanisme des "sockets" du système UNIX.

En conclusion, on retrouve dans ce projet une des préoccupations du génie logiciel, celle de la réutilisabilité. L'approche proposée s'appuie sur les concepts d'agent, d'acteur et plus généralement d'objet.

2.2 - Le projet CONSTRUCT.

Le cadre

Construct qui nous intéresse du point de vue méthodologie, n'est pas directement classé dans les systèmes Multi-Agents. Il a pour cadre les Systèmes à Base de Connaissances (SBC); Son optique étant la génération d'outils pour la réalisation de SBC accessibles à des non spécialistes.

Il en ressort essentiellement une méthodologie de conception : **COMMET** et un outil la supportant : le **Workbench** [DAVID 92].

Dans ce projet, l'idée initiale a été la Réutilisabilité. Cette notion importante a pour but de permettre à des non-programmeurs de configurer des applications de type Système à Base de Connaissances. Première constatation : un non-programmeur n'est pas nécessairement quelqu'un qui ne sait pas programmer mais quelqu'un qui n'a pas le temps ou les connaissances détaillées nécessaire à un domaine particulier. Deuxième constatation, il n'y a pas ou peu d'outils pour ces cas là. Le projet était de développer une méthodologie qui ne reste pas confinée à une approche de développement spécifique.

D'après [STEELS 92] toutes les approches de développement sont valables. Par exemple l'approche encapsulation type Langage Orienté Objet ou l'approche Librairie Logiciel ne sont pas "mauvaises" mais trouvent leurs limites dans des domaines particuliers. Telle une librairie, qui se restreint lorsqu'on s'attaque à un très large jeu de structure de données, ou la complexité s'accroit proportionnellement au niveau d'encapsulation.

Partant de là, Steels argumente sur une approche "boîte blanche" (par opposition à "boîte noire" où la réutilisabilité se fait sans forcément connaître la structure de données ou les algorithmes utilisés) mais qui permet le réemploi à un niveau de granularité variable, par description des composants de l'application de façon plus abstraite que la description informatique, en gardant une liaison entre ces deux niveaux.

Le principe

la méthodologie proposée dans le projet CONSTRUCT est appelée COMMET (pour COMponential METhodology) et basée sur la différenciation entre le **niveau connaissance** (knowledge level de [STEELS 92]) et le **niveau symbole**. Le niveau connaissance relève des taches indépendamment de la façon dont sont structurées les connaissances. Le niveau symbole est le niveau logiciel qui se compose du **niveau code**, textuelle ou se réalise l'écriture et le **niveau exécution** qui est une translation du niveau code par compilation, chargement ou interprétation.

La méthodologie COMMET (COMponential METhodologie) [JONCKERS 92], permet la modélisation d'une application au niveau connaissance par itérations sur trois étapes, sans en privilégier une par rapport aux autres :

- étape **Modèles**, décrivant les connaissances à utiliser : il faut identifier les objets, les classifier, déterminer les relations. Ici on distingue les **Modèles du domaine**, décrivant les connaissances très générales (connaissances applicable d'une application à l'autre) et les **Modèles du cas** décrivant les connaissances spécifiques du domaine d'application. De même on distingue la **forme**, structure du modèle, du **contenu** se référant à l'intérieur du modèle.

- étape **Tâches**, décrivant ce qui doit être réalisé : permet d'établir un **diagramme de dépendance des modèles** qui représente les flots entre modèles et tâches. Ici on distingue le **domaine d'acquisition** du **domaine d'application** des tâches. Les tâches se décomposent en deux types, tâches solutions (sans sous-tâches) et tâches décomposables (en sous-tâches) donnant une structure de tâches.

- étape **méthodes**, décrivant comment les tâches se réalisent : ici on crée un **diagramme de contrôle** qui est un automate d'états finis.

Lors de l'itération sur ces trois points le choix d'un type de tâches peut orienter vers un modèle, ou une méthode particulière, privilégier telle tâche, etc...

L'approche COMMET est donc de ne pas privilégier une méthode particulière parmi d'autres, en considérant qu'elles sont nécessaires et complémentaires. La réutilisabilité en est accrue en permettant d'intégrer des choses existantes très variées.

Un couplage important est établi entre le niveau connaissance et le niveau symbole, chaque composant d'un niveau à un composant dans l'autre niveau. Ce lien permet une configuration automatique du niveau symbole d'après le niveau connaissance et permet une construction progressive d'une librairie de composants réutilisables. Ce couplage est reporté sur le Workbench.

Le **Workbench** est un environnement d'ingénierie de la connaissance supportant COMMET. Il permet de réaliser un SBC (Système à Base de Connaissance) suivant un cycle en quatre étapes :

- **analyse/modélisation du SBC** : identifier les différents modèles et tâches donnant le diagramme de dépendances des modèles et la structure des tâches.

- **conception** : choix des méthodes et de la forme des modèles. A cette étape la nécessité de modèles intermédiaires peut apparaître ainsi que le besoin de tâches supplémentaires influençant l'étape d'analyse précédente.

- **acquisition des connaissances** : "remplir" les modèles du domaine et stocker les informations dans les structures de données.

- **génération de l'application** : c'est l'étape d'exécution de l'application.

Le **Workbench** a permis de mettre en oeuvre la gestion d'un cycle complet de production d'un SBC, de la modélisation jusqu'à l'application finale.

CONSTRUCT permet une intervention à toute étape du développement d'une application pour accéder, changer tous composants évoqués.

Suivant les capacités (spécialiste ou non) et les connaissances (avec et sans expérience) du développeur le point d'entrée peut être différent.

Un prototype du Workbench a été développé en LISP sur Macintosh, puis une implantation a été réalisée en langage CLOS et d'autres langages sont prévus.

2.3 - La plate-forme de développement ABE

Le cadre

ABE (**A Better Environment**) [HAYES-ROTH 88, HAYES-ROTH 91] a pour origine le constat suivant : les trois principaux axes de développement système experts, systèmes tableaux noirs et systèmes orientés objets basés chacun sur un modèle sous-jacent d'ordinateur et une méthode d'organisation des connaissances ont montré leurs faiblesses. Notamment au niveau de la modularité, la réutilisabilité, l'indépendance vis-à-vis de l'implantation bref de la souplesse des différentes configurations et des outils proposés.

D'autre part dans un souci d'efficacité des développements, le besoin de coopération entre processus concurrent sur des plates-formes hétérogènes se fait plus en plus sentir. Par hétérogénéité on entend non seulement systèmes matériels et logiciels différents ayant impliqué des méthodologies de développement différentes mais aussi une distribution logique et géographique large.

Les buts qui ont dirigés ABE ont été d'étendre l'état de la technologie de traitement des connaissances pour s'occuper de l'exigence des systèmes coopératifs intelligents, puis de "modulariser" les composants afin de procurer des modules standards.

En définitive ABE veut développer un cadre de développement modulaire, et associer des outils d'analyse pour construire des systèmes intelligents.

Les besoins d' ABE :

- Etablir un concept de machine virtuelle qui permette de décrire une application indépendamment d'une plate-forme. C'est l'idée de machine fédérative. Que l'on retrouve typiquement dans les systèmes d'exploitations distribués.
- Posséder un environnement qui permette plusieurs niveaux d'abstraction.
- Permettre la composition de systèmes / sous-systèmes, c'est-à-dire une interopérabilité et une composition uniforme par module.

Le principe

La méthodologie d'ABE est centrée sur le concept appelé MOP (Module Oriented Programming).

Une application est un ensemble de modules, distribués, communiquant de façon asynchrone et qui peuvent s'exécuter concurremment.

Dans ce principe de modularité, chaque module est soit une "primitive" (module de base), soit est composé d'autres modules. La communication se fait entre modules à travers des ports de communication associés à chaque module : un contrôleur local dirige les communications internes.

La construction d'une application se fait par définition des différents modules qui la composent et par définition des communications entre eux.

S'il s'agit d'un module de base ou primitive, il est considéré comme une "boîte noire", il peut être soit construit dans un langage quelconque, soit récupéré. ABE n'a pas de visibilité sur sa structure interne et n'a de vue que par ses ports d'entrées / sorties.

L'évolution vers d'autres modules se fait récursivement par inclusion de modules jusqu'à l'obtention d'une application sous forme d'un ensemble de modules. La figure 3.4 montre cette composition modulaire et la récursivité employée.

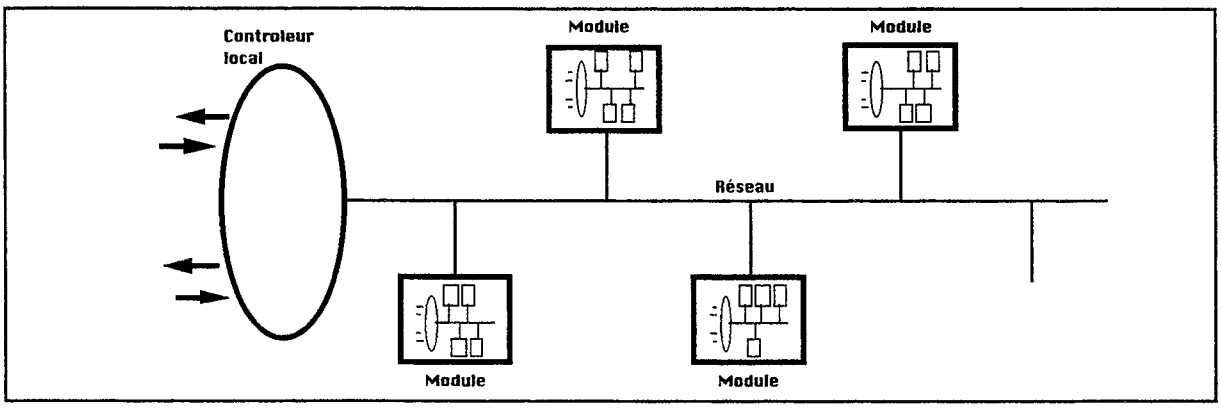

Figure 3.4 : Structure d'un module ABE.

Les échanges entre modules se font sur la base de types de données abstraites ADT (Abstract Data Types) qui sont des structures de données haut niveau, c'est-à-dire des données organisées en vue d'un but précis.

Lors de "l'assemblage du module" l'utilisateur définit le mécanisme d'échange entre les composants. Pour Cela ABE propose un ensemble d'outils, environnement de développement appelé Framework.

Un framework permet de développer et d'exécuter le système. Chaque framework procure un environnement de développement et un environnement d'exécution. Le développement permet de créer des modules comme boîte noire ou par inclusion, avec l'aide d'un langage graphique et d'un éditeur. Pour l'exécution ABE, propose un contrôle local des communications

La plate-forme ABE est implantée sur une machine virtuelle qui possède les principales caractéristiques du système d'exploitation temps réel distribué sous-jacent. Plusieurs systèmes d'exploitation ont pu être utilisés tel que UNIX/MACH sur station de travail SUN, et MS-DOS

sur machine IBM PC. La couche d'interaction avec l'utilisateur est celle des frameworks qui proposent des outils pour développer et exécuter une application.

ABE est un système dans un environnement très complet en vue de développer des applications distribuées. Son domaine d'application est général et peut s'appliquer à celui plus particulier de l'Intelligence Artificielle.

3 - L'approche langage

3.1 - Langage d'acteur

Avec le modèle d'acteur de [HEWITT 73] l'ensemble des connaissances nécessaires à un problème est distribué au sein d'acteurs qui possèdent des accointances, données locales sur les autres acteurs et un comportement qui définit les actions en fonction d'événements survenant à la suite d'envoi de messages, seul moyen de communication. Un acteur est défini par un comportement et par un mode d'exécution concurrent par rapport à des objets simples qui restent d'exécution séquentielle [BARTHES 90].

Ce principe se retrouve dans **MERING IV** [FERBER 89], langage d'acteur qui permet à la fois de représenter des connaissances et de définir des modèles d'agents. L'originalité du projet réside dans une architecture totalement réflexive, l'ensemble du langage s'auto-définit tant dans ses aspects statiques (héritage, instanciation ...) que dynamique (le mode de réaction des acteurs est défini à un niveau méta).

Comme autre exemple, plus récent, de langage d'acteur pour la construction de systèmes Multi-Agents, nous abordons PLASMA II.

3.2 - Le langage d'acteur PLASMA II

Le cadre

PLASMA II [ARCANGELI 93] est un langage de programmation par acteurs développé à l'IRIT (Institut de Recherche en Informatique de Toulouse). Les acteurs sont des entités communicantes qui coopèrent à un but commun. Ils sont proches des objets qu'ils doivent modéliser, leurs communications sont asynchrones et transmettent des informations, demandes ou réponses. Un système d'acteur peut évoluer dynamiquement par création d'acteurs et par évolution des comportements de chaque acteur en fonction des messages reçus.

PLASMA II a axé sont développement sur la similarité entre acteur et agent et a développé un outil d'implantation des systèmes Multi-Agents.

PLASMA II se présente sous forme d'un interprète disponible sur plusieurs environnements multi-processeurs telles que réseau de stations de travail (SUNs), réseau de transputer (T-Node) et machine à mémoire partagée (BBN).

Le principe

L'idée de base des acteurs se retrouve à plusieurs niveaux, par exemple les comportements d'un acteur sont eux-mêmes décrits en termes d'acteurs. Cette notion de

récursivité se retrouve également au niveau de l'application elle même, un système pouvant être recomposé en un seul acteur d'un système plus important. On trouve ainsi une notion de granularité variable assez importante.

Pour définir un système Multi-Agents un agent sera décrit par un acteur. Son comportement est décrit comme un aiguillage vers des traitements adéquats. Ce comportement évolue sans cesse par réorganisation périodique selon la fréquence des services demandés.

Au point de vue communication, le protocole de base est asynchrone, non bloquant. Les messages sont stockés dans une boîte aux lettres pour chaque agent et sont traités en file. Cependant d'autres protocoles ont été implantés comme les transmission bloquantes (attente de réponse), et la diffusion à un groupe ou par "cri" (pas de précision sur le destinataire).

Un tel langage, qui évolue encore, est intéressant pour implanter des systèmes Multi-Agents car il possède des possibilités de reconfiguration et des capacités d'apprentissage qui permettent un dynamisme important dans une application.

4 - Conclusion

Nous présentons une brève synthèse de ces quatre projets.

Type de projet :

- deux sont des produits de coopération entre industrie et recherche, il s'agit de ARCHON et CONSTRUCT, ce sont des projets à l'échelle européenne,
- un est un produit industriel, ABE,
- et un est un produit de laboratoire de recherche, PLASMA II.

Domaines d'application :

- Pour ARCHON, c'est dans l'univers industriel que se situe son domaine d'application.
- Pour CONSTRUCT il s'agit de la génération d'outils pour la réalisation de système a base de connaissances.
- Pour ABE, le logiciel le plus ancien, c'est un environnement de développement d'application distribuées.
- Et pour PLASMA II, langage d'acteurs, une des applications est l'écriture de système Multi-Agents.

Objectif :

L'objectif d'ARCHON est double, procurer un atelier de développement dans une optique Multi-Agents et assurer le développement dans une optique Génie Logiciel. La tendance objet-acteur-agent s'inscrit dans l'orientation "Multi-Agents", alors que les points suivants s'inscrivent dans l'optique "Génie Logiciel" :

- Réutilisabilité, par intégration de systèmes existants simples ou complexes,
- cette réutilisabilité s'étend à des domaines très hétérogènes
- Fiabilité dans l'introduction d'une couche ARCHON.

L'intérêt du projet CONSTRUCT réside dans sa méthodologie et dans l'application de celle-ci sur un outil. L'intérêt du Génie Logiciel a été pris en compte pour fournir des interventions possibles tout au long d'un développement.

ABE est un environnement de développement c'est-à-dire qu'il possède tout ce qui est nécessaire à la conception d'une application. L'ensemble de ses outils permet, en forçant l'utilisateur dans un schéma précis en fonction des données échangées, d'évoluer vers une application finale. Cette méthodologie de conception est basée sur l'homogénéité : tous les modules conçus sont vus de la même façon, sur la modularité : tout se passe comme des échanges entre modules et sur la réutilisation qui découle des deux éléments précédents.

La présentation de PLASMA II n'est pas là en tant que logiciel complet ou d'environnement de développement, mais pour démontrer que l'idée de langage est nécessaire pour décrire des concepts d'agents et de société d'agents. En effet un agent est un niveau d'abstraction supplémentaire, et fournir un langage pour penser en ce sens aide a formaliser les idées.

En conclusion, bien que de but et d'origine diverses, ces travaux ont tous en commun la notion d'entité de travail, plus ou moins finalisée comme module ou agent :

- Dans ARCHON c'est bien la notion d'agent qui sous-tend toute la conception.
- Pour CONSTRUCT l'idée de module n'est présente que sous la forme de principe de "boîte blanche" dans un schéma de conception de systèmes à base de connaissances.
- Pour ABE cette idée de module est présentée pour faciliter le travail de conception d'une application. Il s'agit là de faciliter un développement en proposant une récursivité basé sur des techniques de communication.
- Pour PLASMA II la tendance objet-acteur-agent est complètement utilisée. L'acteur comme objet actif et comme outil d'implantation du concept d'agent.

Nous voyons également qu'une méthodologie est impliquée dès que la prétention d'un produit logiciel est d'être industriel c'est-à-dire devant répondre à des normes de qualité, de fiabilité, de réutilisabilité ...

V - Conclusion

En conclusion notre point de vue est de positionner COALA entre les langages d'acteurs et les plates-formes de développements. Des différents travaux cités nous avons gardé l'approche Génie Logiciel en ce sens que nos désirs étaient d'augmenter la modularité et le découpage des tâches aussi bien dans le projet lui-même que lors de l'utilisation de l'environnement de COALA. L'idée de méthodologie a sous-tendu notre volonté lors de l'établissement d'un langage d'agent ainsi que pour le développement d'une application.

Même si la base du projet reste le logiciel MAPS, nous avons voulu lui faire prendre une nouvelle dimension dans la résolution de problème en tâchant d'élever à un niveau d'abstraction supplémentaire la façon de penser un développement logiciel.

Chapitre 4 - Réalisation, l'environnement de développement Multi-Agents : COALA

I - Présentation

COALA (COoperative Agent LAnguage) se définit dans trois directions majeures, une méthodologie de conception, un environnement de programmation et un langage d'agents. Nous positionnons effectivement COALA à l'intersection de ces trois axes pour profiter des moyens développés par le génie logiciel et des capacités développées par la recherche. Notre ambition pour un langage d'agents est, par rapport à un langage d'acteurs, d'apporter une méthodologie et de fournir les outils nécessaires à cette perspective. L'ambition n'est pas de faire un langage universel mais de pouvoir s'"attaquer" à des applications dédiées à la formulation et résolution de problème par coopération d'agents. Pour cela nous utilisons le travail précédemment réalisé avec MAPS en restant dans la philosophie de coopération et de découpage des tâches en fonctions des types de connaissances.

Les raisons qui nous ont amené à développer COALA comme un langage d'agent coopératif sont multiples. Elles résultent aussi bien de la prise en compte des faiblesses du système MAPS, que de la volonté de le faire évoluer pour se placer dans une optique résolument Génie Logiciel.

Il faut noter ici que si l'on parle de langage, il ne s'agit pas de langage de communication entre agents mais plutôt de faciliter l'écriture du développement d'une application d'un système multi-agents en se plaçant dans un monde de pensée "agents". Ce langage s'introduit comme un des outils de l'environnement COALA permettant le développement et l'implantation d'applications Multi-Agents. Ce langage doit donc développer des notions claires et précises du concept d'agent.

L'optique Génie Logiciel apporte des notions de modularité, extensibilité, réutilisabilité ainsi que facilité d'utilisation, efficacité ... Ces notions sont introduites au niveau de l'environnement COALA dans un souci de qualité logicielle.

Nous avons vu que certaines faiblesses de MAPS se situaient au niveau de son manque d'abstraction et du peu de "dynamisme" dans une application. Le problème de l'abstraction trouve un remède dans une méthodologie qui par une série d'étapes propose à l'utilisateur un mode de pensée évolutif, depuis une vue globale, puis plus restreinte sur un agent, et enfin sur l'exécution. Mais le langage d'agent est également prévu pour apporter une vision plus globale

au sein même d'un agent. Les dialogues entre agents se font par échanges de données, d'instances, de mondes, ces deux derniers types d'échanges apportant le dynamisme recherché et le langage offrant des instructions adéquates de manipulation.

Le travail pour parvenir à une méthodologie, un environnement de programmation, et un langage d'agent s'est effectué en plusieurs temps.

Une première phase a vu l'étude du système MAPS et des applications existantes ainsi qu'un dialogue avec les utilisateurs et nous a servis de phase d'analyse des besoins. En effet à partir d'une expérience dans plusieurs applications développées, ainsi que le désir de faire évoluer le concept de multi-agents, a permis de déterminer les trois directions ci-dessus.

A partir de ceci la seconde phase de travail a été de produire des spécifications. Pour ce travail nous nous sommes appuyés sur l'expérience de la société TITN-Alcatel, partenaire de l'équipe : pour des raisons de confidentialité nous ne développerons que certains des éléments de la méthode de spécifications de logiciels propre à cette société.

La spécification est vue comme suit : elle définit ce que fait le système et quand il le fait sans se préoccuper du comment il le fait. Cela revient à décrire le système tel qu'il est vu par l'utilisateur : c'est la référence à atteindre. Trois axes de modélisation peuvent être dégagés :

- un axe fonctionnel, c'est le traitement appliqué aux données, le système est vu comme un ensemble qui transforme les données qui lui sont présentées en entrées.
- un axe dynamique, c'est la réponse aux informations de contrôle, le système peut être vu comme un ensemble d'automates à états finis qui agissent les uns sur les autres (partie opérative et partie contrôle-commande).
- un axe des informations échangées entre le système et l'environnement, modélisation décrivant les relations entre les données et leurs attributs, ainsi que la hiérarchie entre les données.

Cette méthode de spécifications a notamment permis de définir l'architecture finale ainsi que la documentation à fournir.

Enfin la troisième phase de travail a vu le développement effectif des divers éléments cités. Ce travail s'est effectué sur station de travail SUN et SONY. La station SONY étant réservée pour le développement de l'interface graphique comme nous le verrons.

Dans la suite du chapitre nous allons voir en détail les trois directions de développement : la méthodologie est d'abord détaillée comme l'enchaînement de l'utilisation des outils de l'environnement proposé, puis cet environnement est vu d'une façon globale sous forme d'une architecture, enfin les principaux outils sont détaillés et parmi ceux-ci le langage d'agents.

II - Méthodologie

1 - Généralités sur la méthodologie

La méthodologie dont nous parlons ici, décrit la façon dont un utilisateur "COALA" va concevoir l'architecture décrivant son application. Notre intention est de faciliter la tâche de cet utilisateur dans le cadre du développement d'un système Multi-Agents pour la résolution distribuée de problème. Mais nous ne voulons en aucun cas restreindre ce processus à la seule utilisation de notre méthodologie, nous voulons proposer un univers ouvert aussi bien à des utilisateurs sans une grande connaissance de la programmation, qu'à des programmeurs compétents.

De manière plus précise, le but d'un utilisateur de COALA est de concevoir un réseau d'agents. Il doit "distribuer" son problème au sein d'agents de deux types, et établir une coopération entre ces agents.

La méthodologie reprend le principe suivant :

- une application est un réseau d'agents,
- un agent est construit à partir de briques de base.
- ces éléments de base sont dans une librairie de classes C++, (c'est un ensemble de classes génériques décrivant les différents constituants d'un agent),
- un agent est construit par instanciation d'une classe générique (KS et KP par exemple) elle même construite par composition d'instanciation des classes génériques des composantes de cet agent.

Pour résoudre son application l'utilisateur procède par étapes en ayant à sa disposition des outils lui permettant d'affiner son application depuis la conception jusqu'à l'exécution.

2 - Phases de la méthodologie

Dans un développement on retrouve de façon générale trois étapes : la génération de l'application, la mise au point et l'exécution. L'originalité de notre méthodologie provient de l'étape de génération de l'application, qui comme nous allons le voir accorde une place prépondérante à une installation "graphique" ainsi qu'à l'utilisation d'un langage d'agents. Dans cette étape de génération on trouve les différentes phases de la méthodologie portant sur deux niveaux d'abstractions : le niveau réseau et le niveau agent. Une troisième phase de transformation permet l'obtention d'une application exécutable. La figure 4.1 suivante résume ce schéma de développement.

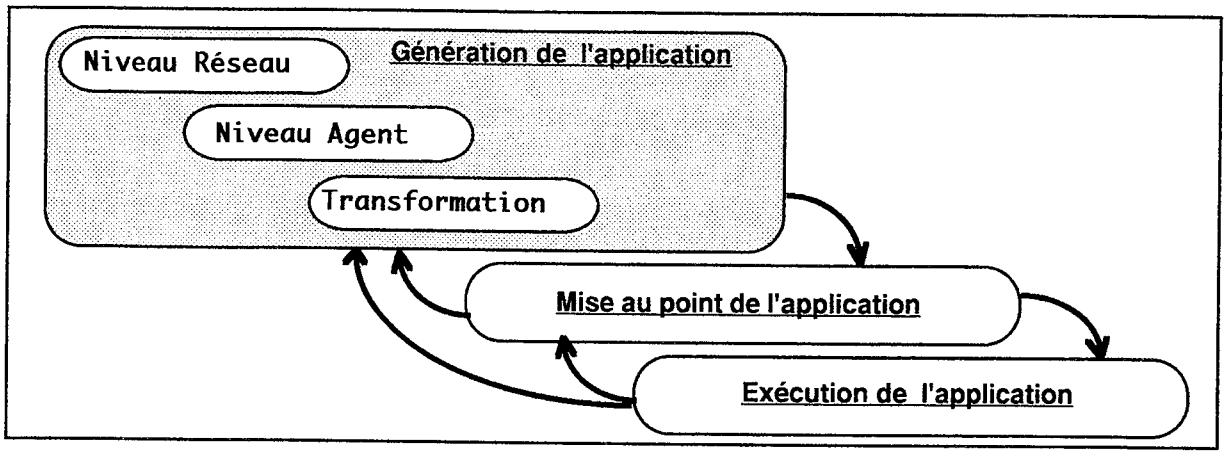

Figure 4.1 : Étapes de développement d'une application.

Lors de la première *phase* en effet, la volonté est de procurer une abstraction de haut niveau. La solution est de visualiser l'ensemble du réseau en cours de conception de façon graphique. L'utilisateur doit penser en termes d'agents réunis en réseau. Un seul "coup d'oeil" doit pouvoir englober l'ensemble de l'application. On met tout d'abord en place un réseau d'agents; on peut alors différencier les agents, les nommer, les répartir, les interconnecter, établir les liaisons. Une aide graphique a donc été envisagée. Dans ce but une interface que nous avons appelée **Editeur Graphique d'Application** a été conçue.

Le réseau étant établi, il convient de le préciser. La deuxième *phase* consiste en un affinement du réseau, en "remplissant les cases" que sont les agents. On est passé au niveau d'abstraction immédiatement inférieur qui consiste à développer chaque agent, c'est-à-dire a développer leurs structures, statiques et dynamiques. Pour cette réalisation, il s'agit en fait d'éditer un contenu textuel. Nous avons envisagé des éditeurs qui peuvent être des éditeurs de textes standards ou des éditeurs plus spécialisés appelés **éditeurs d'agents** dans une prochaine évolution. Ici l'abstraction est réalisée par le **langage d'agents** appelé COALA. Il a été conçu en tenant compte du précédent langage : celui du système Maps. Ce langage permettra d'exprimer un agent tel qu'il doit être construit en termes d'objets et de règles. Il est un niveau d'abstraction au dessus du langage de programmation C++ sous-jacent à COALA. Mais on peut également développer directement en code C++. En effet un élément important de COALA est une **librairie de classes**, qui prédétermine l'essentiel d'une application et d'un agent. Cette librairie donne tous les éléments de base pour produire un agent par héritage des classes préétablies. Le langage d'agent permet même si sa syntaxe est proche de celle de C++ de s'affranchir de la connaissance de C++. Mais c'est surtout en terme d'abstraction que le gain est le plus important.

Nous avons voulu imposer un passage obligé par une étape C++ pour profiter d'une compilation complète de l'application dans un souci d'efficacité lors de l'exécution.

Ceci implique une *phase* de transformation qui convertit les fichiers "agents" (écrits en langage d'agents) en fichiers C++ puis en fichiers exécutables. ces passages se font

premièrement grâce à un **Précompilateur** spécifique puis avec compilation et éditions de liens "classiques" de l'environnement UNIX.

Le précompilateur effectuera un certain nombre de vérifications de type et de cohérence et le compilateur de par son typage statique garantira une fiabilité à l'exécution. Ces phases sont résumées dans le schéma figure 4.2.

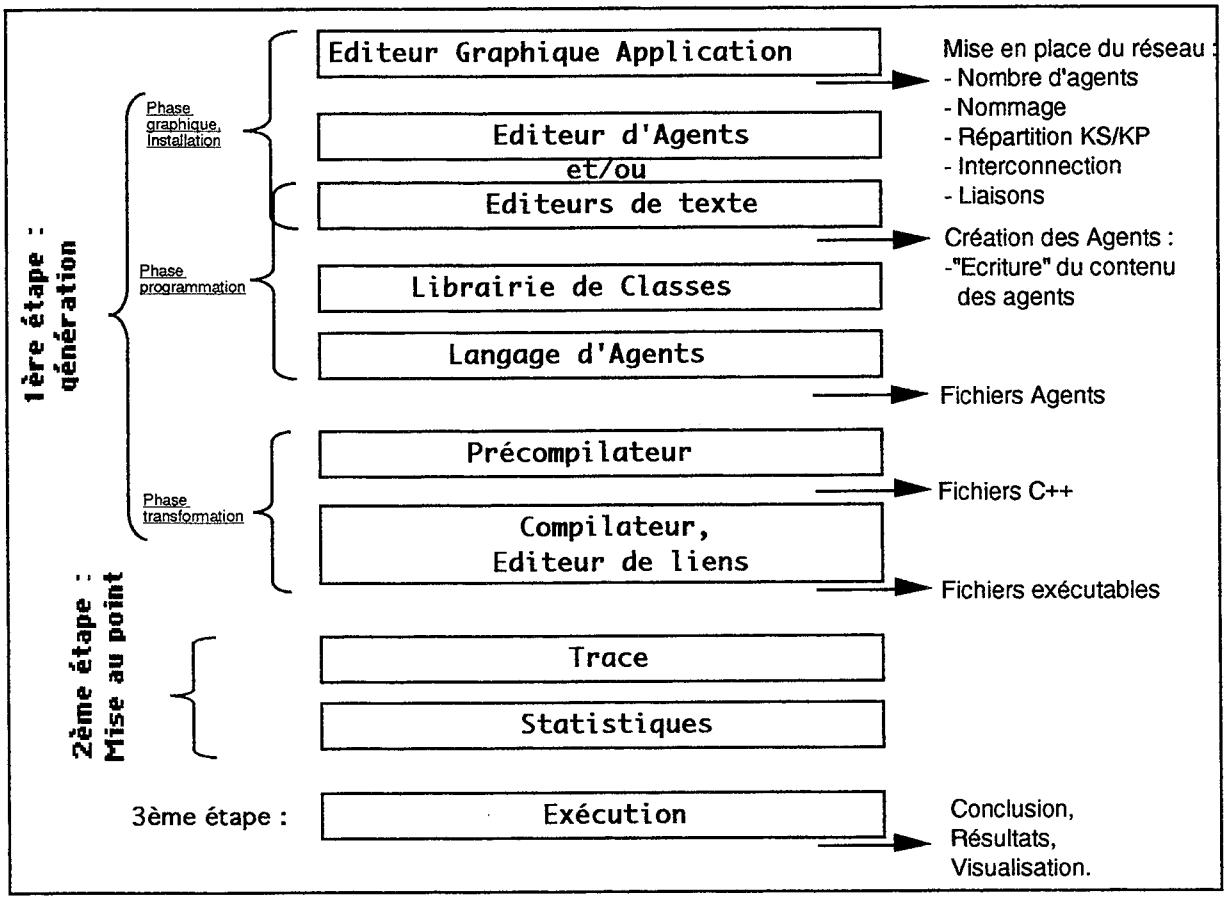

Figure 4.2 : COALA, une succession d'étapes et de phases.

Les deux dernières étapes sont la mise au point et l'exécution. On visualise l'application en cours d'exécution sur le réseau d'agents, des outils statistiques permettent de recueillir des informations sur des paramètres tels que, les temps d'exécution, d'occupation, de communication ... qui aident à valider le bon choix quant à la répartition opérationnelle et fonctionnelle entre les agents du réseau. De même un outil de trace permet de suivre le déroulement d'une application.

3 - De la méthodologie à l'environnement

On voit que la méthodologie proposée est basée sur un ensemble d'outils : ils ont été regroupés au sein d'un environnement dont nous donnons ci-dessous l'architecture.

III - Architecture

L'environnement de programmation COALA doit procurer des facilités pour l'utilisation des différents outils (éditeurs, langage, précompilateur, ..). Une architecture, présentée figure 4.3, pour regrouper ces outils a été développée et peut se décrire suivant quatre niveaux :

- niveau Matériel,

il consiste en un ensemble de stations de travail reliées par un réseau Ethernet. Couramment des stations SUN 4 (SPARC II) sont utilisées mais l'environnement peut s'exécuter sur matériel HP, DEC, SONY, Silicon Graphics.

- niveau Système d'Exploitation,

actuellement le système utilisé est l'environnement SUN OS 4.1 compatible UNIX System V assurant la portabilité vers d'autres matériels. Le système d'exploitation assure notamment les communications par l'intermédiaire du principe sockets UNIX.

- niveau Système de Gestion des Agents,

il regroupe l'ensemble des outils décrits ci-dessus. Ils sont écrits en C++ compilés, soit utilisent les capacités UNIX, tel que le langage Shell.

- niveau Interface Utilisateur,

il est basé sur le standard MOTIF pour des raisons de standardisation. MOTIF étant couramment implanté sur les stations de travail citées précédemment. Cette Interface regroupe dans un système de fenêtrage l'accès aux différents outils proposés et ceci à n'importe quelle étape dans la génération d'une application.

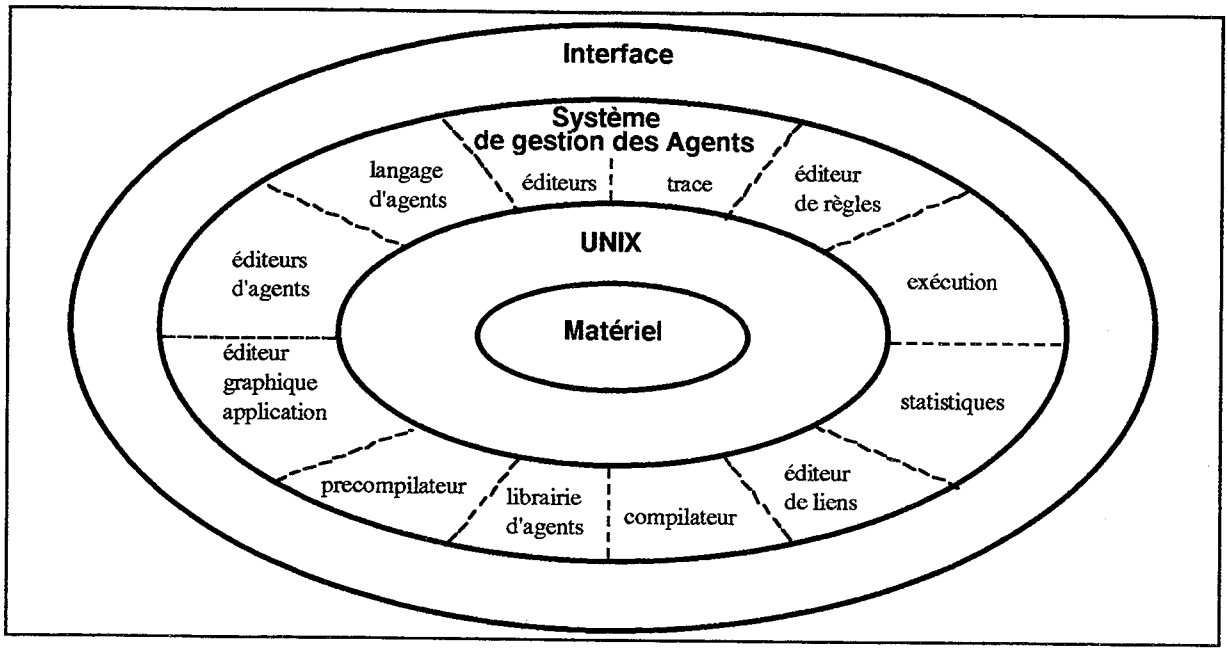

Figure 4.3 : Architecture de COALA.

L'environnement COALA est implanté comme une couche sur UNIX dont il utilise les capacités. Les outils sont accessibles par l'interface. Les fonctionnalités d'UNIX sont également accessibles, l'utilisateur ayant à tous moments la possibilité "d'ouvrir un Shell". Nous avons voulu garder une complète transparence pour l'utilisateur.

IV - Implantation

Les paragraphes suivants présentent les principaux éléments de COALA qui ont été développés. Ces éléments que sont la Librairie de classes C++, le Langage d'agent, le Précompilateur et l'Interface graphique ont été développés dans un certain ordre. L'influence du principe même des agents KS et KP a déterminé la structure de l'ensemble. Le langage sous-jacent étant C++, le travail a commencé par la réimplantation de la librairie C++, en profitant des acquis de MAPS et des expériences acquises par le développement d'applications. A partir de cette librairie a été développé le langage en tenant compte de l'existant et des nouvelles fonctionnalités à implanter en tant qu'abstraction supplémentaire. Ici nous avons tenu compte de la librairie C++ pour établir une adéquation étroite entre langage et librairie pour un maximum d'efficacité. Ensuite le précompilateur a été développé pour permettre de traduire le langage d'agent en C++. L'interface graphique a été défini de façon générale et toutes fonctionnalités nouvelles peuvent y être introduites.

1 - Librairie de classes C++

Je remecie ici Olivier BAUJARD qui m'a accepté comme collaborateur dans cette délicate mission de conception de la librairie de classes C++.

1.1 - Généralités sur la librairie de classes C++

On appelle librairie l'ensemble des classes qui par héritage, composition puis instanciation donneront les agents s'exécutant comme processus dans l'application. Pour des raisons d'effcicacité ces classes doivent exprimer tous les principaux éléments apparaissant au sein d'un agent, ainsi que les concepts liés aux communications.

Le principe est de créer des classes génériques, dont il suffira d'hériter et de spécialiser pour créer les éléments de l'agent puis l'agent lui-même.

Le langage C++ offre comme outils de conception, l'abstraction de données, l'héritage, et une certaine généricité nécessaire à ce projet.

exemple :
créer un agent KS nommé Essai se fait de la façon suivante,

```
class Essai : public KS {
...
}
```

KS étant une classe générique contenant tous les éléments d'un agent KS, objets,règles ....
On a ainsi créé une classe Essai qui lors de l'instanciation génèrera un agent KS.

La composition de la librairie de classes C++ est donc calquée sur la composition d'un agent. A titre de rappel nous redonnons la composition des agents :

Agent KS = liste d'objets + liste d'accointances + liste de règles
+ moteur d'inférence + liste éventuelle de procédures
+ redéfinition éventuelle des comportements qui lui sont propres.

Agent KP = liste d'accointances + liste de règles + moteur d'inférence
+ liste éventuelle de procédures
+ redéfinition éventuelle des comportements qui lui sont propres.

Les figures 4.4 et 4.5 résument quelles sont les classes créées a partir d'un agent KP et d'un agent KS et de leurs composants, ainsi que les méthodes des classes créées a partir des méthodes et comportements attribués à ces agents.

De plus, comme nous le verrons plus loin, des classes regroupant des données et méthodes communes à certaines classes sont créées pour servir de racine d'héritage à certaines classes . De même dans ces figures apparaissent également certains liens de composition, notamment qu'un agent est composé de "Element", "Connection", ...etc, de même un "Element" est composé d'"Object", ...

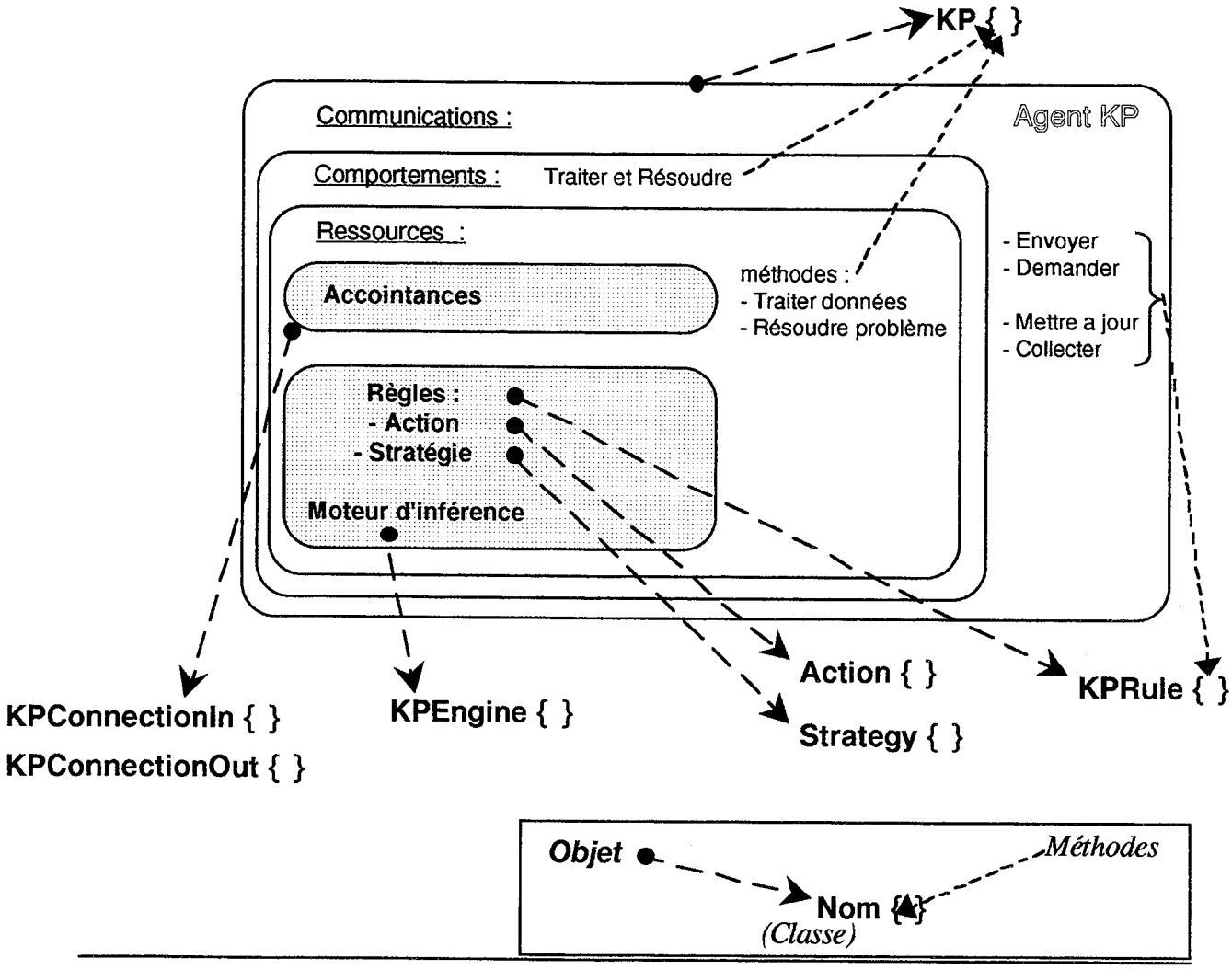

Figure 4.4 : classes d'un agent KP.

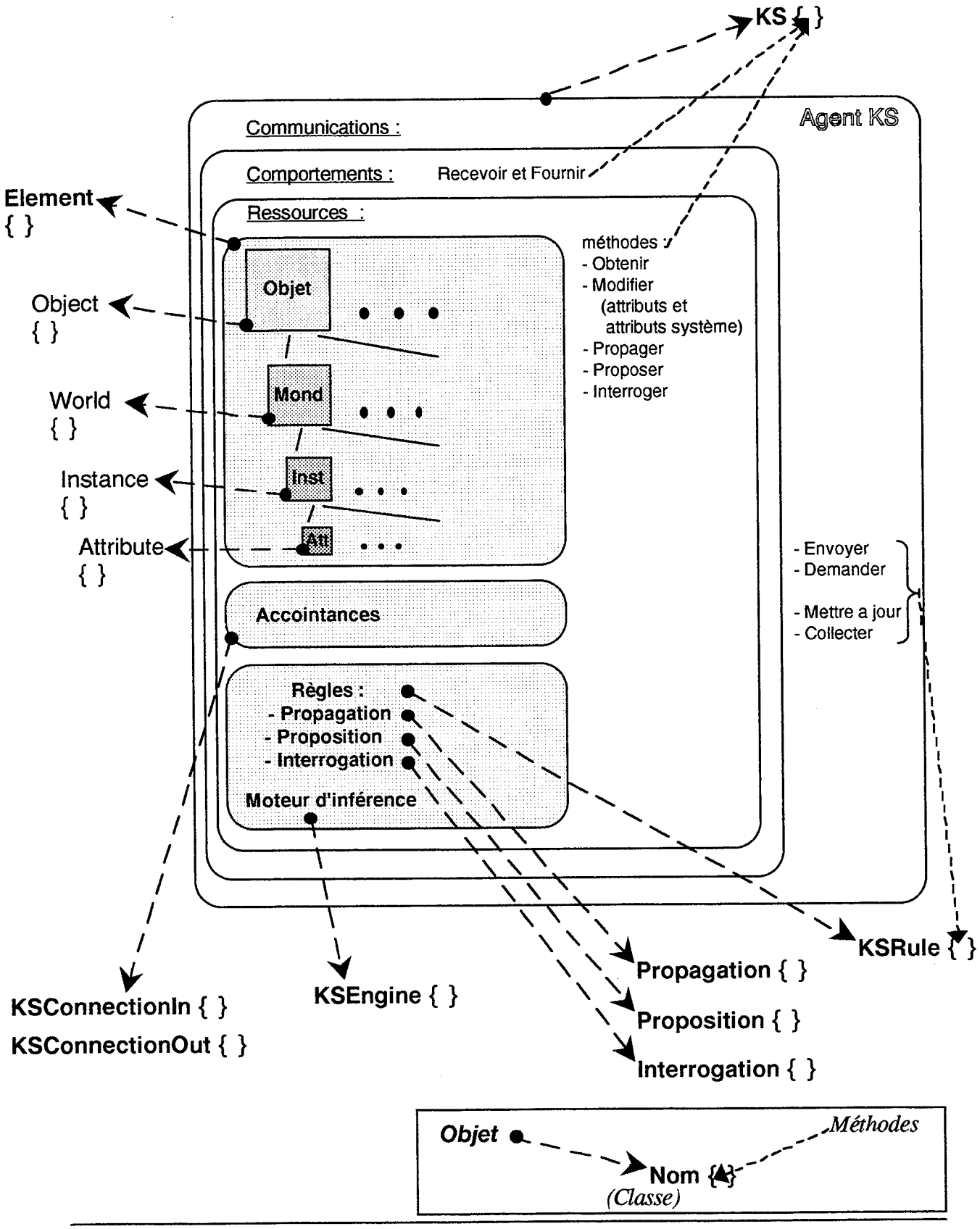

Figure 4.5 : classes d'un agent KS.

D'une façon générale les classes sont les suivantes :

- **KS, KP** pour décrire les agents.
- **Element, Object, World, Instance, Attribute, AttributeInt, AttributeFloat, AttributeString, AttributeListInt, AttributeListFloat, AttributeListString,** pour exprimer les objets d'un agent (KS en l'occurence).
- **RPC, Connection, ConnectionIn, ConnectionOut, KSConnectionIn, KPConnectionIn, KSConnectionOut, KPConnectionOut,** pour exprimer les connexions d'un agent (accointances) et ses capacités de communications
- **Rule, KSRule, KPRule, RuleSet, Proposition, Interrogation, Result, Action, Strategy,** pour exprimer les règles d'un agent.
- **Engine, KSEngine, KPEngine,** pour exprimer le moteur d'inférence d'un agent.

Dans ces classes beaucoup de méthodes sont déclarées virtuelles, ceci afin de laisser à l'utilisateur la possibilité de redéfinir ses propres méthodes. Il s'agit ici de l'ouverture vers des perspectives importantes qui peut devenir une des raisons essentielles pour l'évolution du système Multi-Agents qu'est COALA.

Dans le reste de ce paragraphe nous allons aborder une description rapide des principales classes de la librairie.

1.2 - Les principales classes

1.2.1 - Les classes "KS" et "KP"

Ces classes génériques contiennent tous les éléments nécessaires à l'installation d'un agent. Celui-ci est créé par instanciation de classe et correspond à un processus UNIX.

Cette classe est composée d'autres classes de base représentant les éléments d'un agent, le schéma figure 4.6 et 4.7 exprime ce fait.

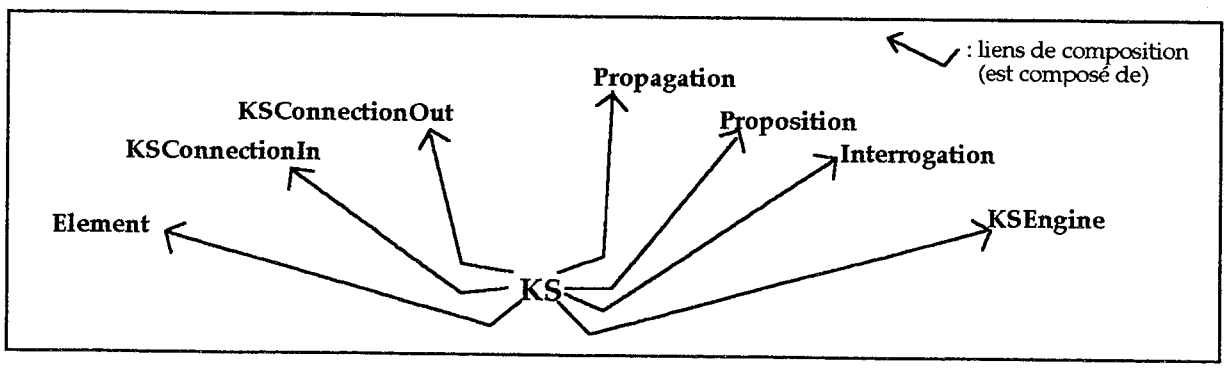

Figure 4.6 : Composition de la classe KS générique.

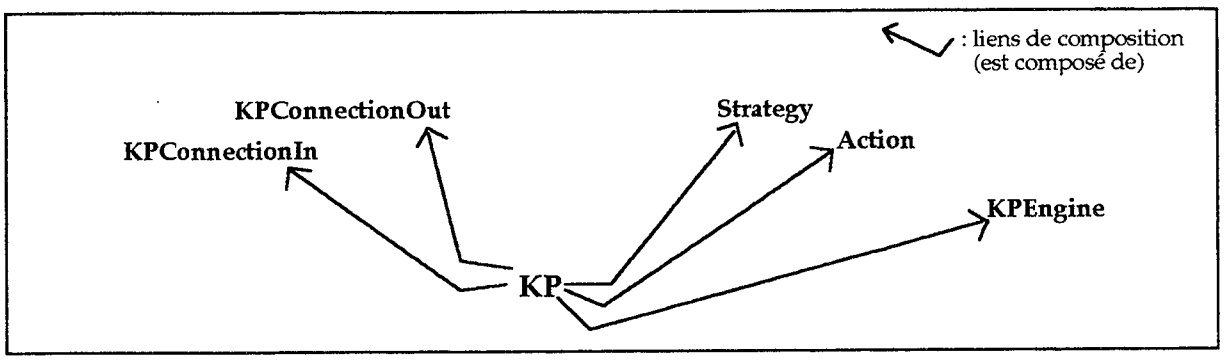

Figure 4.7 : Composition de la classe KP générique.

Contenu des classes KS et KP

Données :

En plus d'un nom et d'un commentaire associé à un agent KS sous forme de chaînes de caractères, on retrouve parmi les données membres les éléments évoqués dans la figure 4.4 et 4.5, c'est-à-dire :

KS :	KP :
- un ensemble des éléments de l'agent, ce sont ses objets,	
- des ensembles de règles de propagation, de proposition et d'interrogation,	- des ensembles de règles d'action et de stratégies,

- un ensemble des connections en écriture et un ensemble des connections en lecture c'est-à-dire les agents avec lesquels il est connecté (et les divers éléments inhérents à ces connections que nous verrons lors des classes 'Connections').
- un moteur d'inférence.

Méthodes :

Au niveau des méthodes de classe on retrouve plusieurs types de fonctions :

- premièrement les fonctions nécessaires à la création de l'agent ce sont :
  - un constructeur
  - et des fonctions 'Add...' qui permettent de créer les objets, connections et règles propres de l'agent.
- deuxièmement les fonctions de "fonctionnement" telles que attente de message, contrôle de boucle.
- troisièmement les fonctions "type" d'un agent vu par l'utilisateur. Ces fonctions sont toutes établies comme fonctions virtuelles, c'est-à-dire pouvant être redéfinies, ceci pour deux raisons:
  - leur fonctionnement standard type est un appel des fonctions de mêmes noms mais de l'élément auxquels se rapporte l'action effectuée, ce qui parfois implique un appel en chaîne de fonctions de même nom. Par exemple pour lire un attribut d'un objet, on utilise une fonction GET. Cette fonction Get de l'agent KS appelle la fonction Get de l'objet (objet au sens instance de classe) Element qui fait appel à la fonction à la fonction Get de l'objet Object ...etc jusqu'à la fonction Get de l'objet Attribute (voir exemple)

- la redéfinition peut être effectuée par l'utilisateur qui a ainsi la possibilité de spécifier des actions très personnalisées pour son application. Il s'agit ici d'un utilisateur expert en programmation C++ et connaissant bien le fonctionnement d'une application COALA.

L'exemple suivant illustre le principe d'écriture de l'ensemble des classes.

```
Classe Ks
   virtual UNSIGNED Get(int i,int j,int& v)
   {
      return Elements.Get(i, j, v);
   }
=>Classe Element
      virtual UNSIGNED Get(int i,int j,int k,int& v)
      {
         return (i<NbObjects) ? Objects[i]->Get(j,k,v) : 0;
      }
   =>   Classe Object
            virtual UNSIGNED Object::Get(int i,int& v)
            {
               UNSIGNED Result=(CurrentWorld) ? CurrentWorld->Get(i,v) : 0;
               _
            }
      =>    Classe World
                  virtual UNSIGNED Get(int i,int& v)
                  {
                     return (CurrentInstance) ? CurrentInstance->Get(i,v) : 0;
                  }
         =>    Classe Instance
                        virtual UNSIGNED Get(int i,int& v)
                        {
                           return (i<NbAttributes) ? Attributes[i]->Get(v) : 0;
                        }
            =>    Classe Attribute
                              virtual UNSIGNED Get(int&)=0;
               =>    Classe AttributeInt (hérite de la classe Attibute)
                                 UNSIGNED AttributeInt::Get(int& a)
                                 {
                                    ... lecture de la valeur attribut 'a' ...
                                 }
```

Exemple : Lecture d'un attribut i de l'instance courante

Mode d'exécution des classes KS et KP

Après instanciation de l'agent KS ou KP, (figure 4.8) le processus est en attente de message dans une boucle. Sur la réception d'un message, il y a appel d'une fonction 'ProcessMessage', cette fonction fait elle même appel à une fonction FilterMessage devant permettre d'associer à chaque message un comportement. Puis suivant le type de message reçu il y a sélection de la méthode adéquate à l'opération demandée ou l'activation des règles sélectionnées. Enfin s'il y a une réponse à fournir, l'appel d'une fonction 'Reply' est effectuée et l'agent se remet en attente .

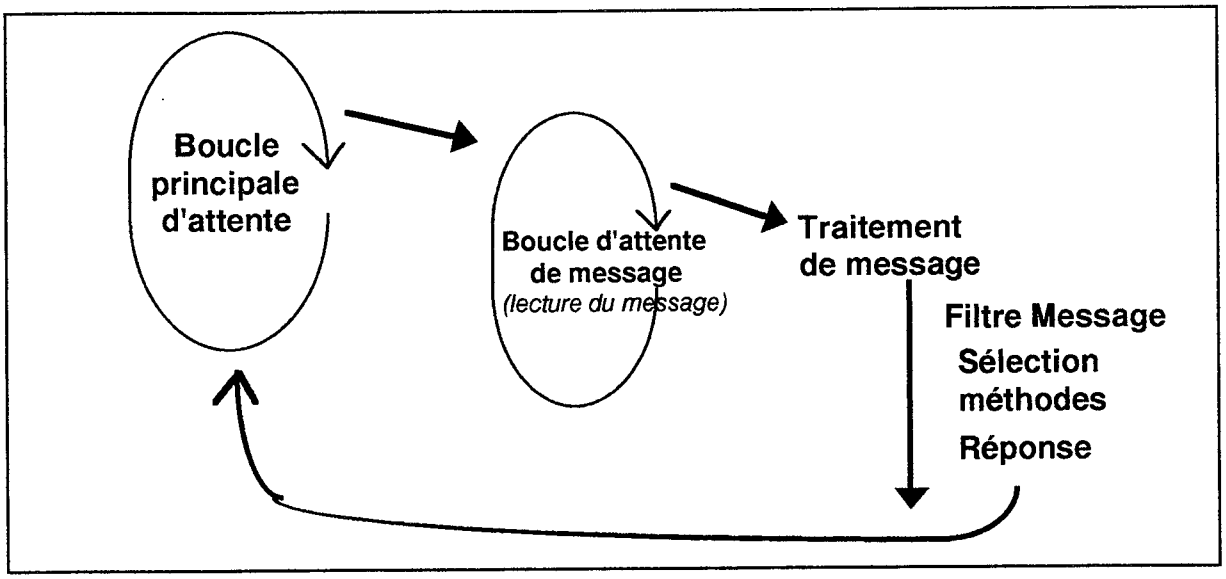

Figure 4.8 : Mode d'exécution d'un agent.

1.2.2 - Les classes relatives aux objets d'un agent KS

Les classes concernées sont **'Element'**, **'Object'**, **'World'**, **'Instance'**, **'Attribute'**. Ces classes définissent les objets d'un agent KS c'est-à-dire une partie des données figuratives que l'on peut disposer dans une application, l'autre partie étant représentée par les règles de propagation, proposition et interrogation agissant sur ces objets. Pour comprendre l'utilisation de ces classes, il nous faut expliquer comment sont construits les "objets" d'un agent KS.

Les **objets** définissent un modèle de la structure de leur représentants physiques (**instances**). Ils regroupent des **attributs** ou variables d'instances. Ces objets peuvent également être considérés du point de vue conceptuel comme des unités de connaissances, représentant le prototype d'un concept, voire d'un cadre de pensée; les instances peuvent alors être considérées comme des mondes de pensée. La notion d'instance rejoint ici la notion d'environnement de travail. Ce dernier est construit comme un modèle dynamique du monde au fur et à mesure des traitements effectués par les agents. Cet environnement constitue le contexte effectif de travail des agents.

Le parallélisme au niveau du traitement des requêtes permet à un agent de maintenir plusieurs mondes de pensée en parallèle : les instances d'un objet peuvent ainsi appartenir à des **mondes** différents. A ce niveau, tout se passe comme s'il y avait multiplication virtuelle et dynamique des ressources de l'agent. De cette façon, tout agent explore l'environnement de manière parallèle et simultanée. Chaque instance correspond à une perception ou une inférence qui correspond à la détection ou la compréhension, par un agent KP, d'un événement de l'environnement. Le parallélisme permet d'émettre plusieurs hypothèses sur ces événements. Ce mécanisme correspond au mécanisme des mondes multiples. Les méthodes de manipulation des instances permettent ainsi de créer, de modifier ou de détruire des mondes de pensée en parallèle. La figure 4.9 résume cette description.

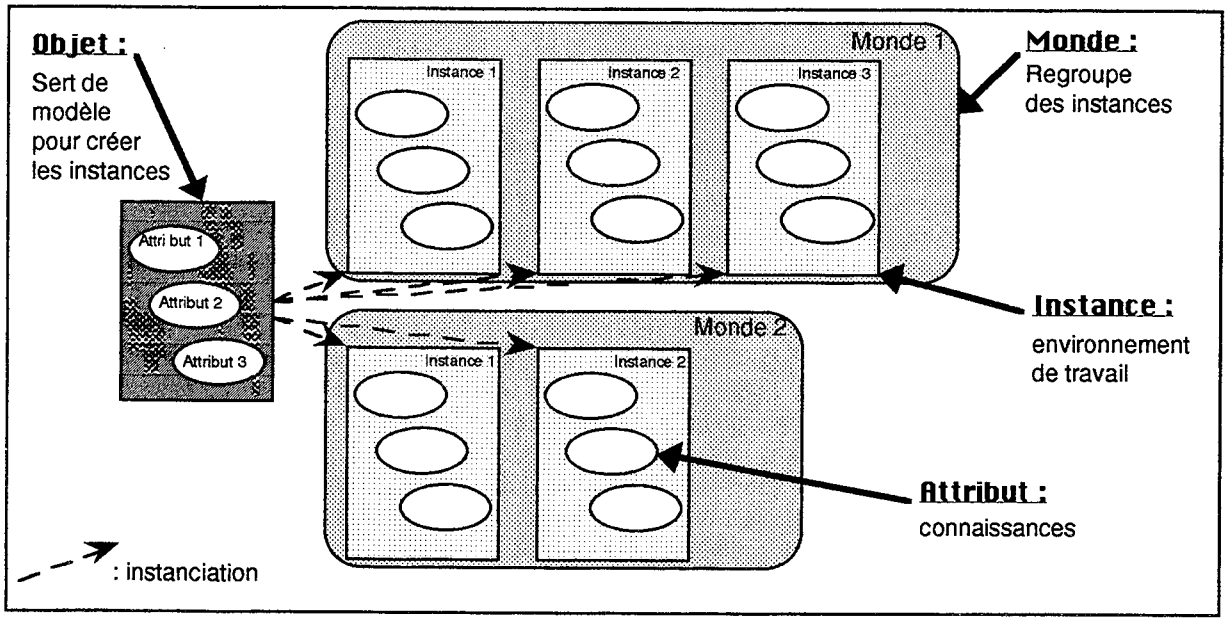

Figure 4.9 : Les objets d'un agent KS.

Aux attributs d'information définis par le concepteur des agents (attributs "utilisateur"), le système adjoint les attributs de contrôle (attributs "système") PROPOSE_FLAG, INTERROGATE_FLAG et RESULT_FLAG. Ces attributs sont ajoutés à la liste des attributs de chaque objet et hérités par leurs instances. Une liste d'instances et une instance courante (ou instance de travail) sont également maintenues pour chaque objet, il en sera de même pour les mondes.

Tout accès à un attribut s'effectue sur une instance particulière, l'instance courante ou instance de travail. Des méthodes internes contrôlent l'accès en lecture et en écriture aux attributs d'information de cette instance. Des méthodes dédiées permettent d'accéder en lecture aux attributs de contrôle de cette instance.

Un ensemble de méthodes dédiées est également fourni pour manipuler les instances d'objets. Elles permettent de créer ou de détruire des instances et également de changer l'instance courante. Ces méthodes permettent ainsi de créer ou de détruire des mondes de pensée, ou de se focaliser sur un monde de pensée particulier. Ces méthodes qui visent à modifier ou mettre à jour l'instance de travail constituent des moyens de focalisation dans l'environnement de travail. Un autre ensemble de méthodes dédiées est également fourni pour manipuler les mondes. Elles permettent de créer ou de détruire des mondes et également de changer le monde courant, ceci en parallèle.

En fin de compte on peut manipuler tout attribut (utilisateur et système) d'une quelconque instance et dans un monde quelconque, par défaut les manipulations se faisant sur l'instance courante et sur le monde courant.

De cette description est issue logiquement le schéma de composition figure 4.10. La classe 'Element' est créée pour manipuler les objets.

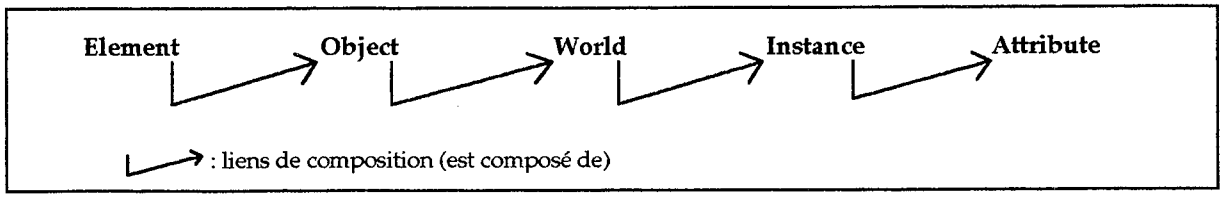

Figure 4.10 : relations entre éléments et ses composantes.

L'ensemble des objets d'un agent KS est décrit sous forme d'un tableau dans la classe 'Element'. Ce tableau est construit et figé par l'utilisateur lors de la création de l'application, c'est l'utilisateur qui détermine ses besoins. Il en est de même pour les attributs a l'intérieur des objets. Par contre le nombre de mondes et d'instances pouvant évoluer dynamiquement suivant les besoins du système, en cours d'exécution de l'application, ils sont stockés dans une liste. Les objets détiennent ainsi une liste de mondes et les mondes une liste d'instances.

On peut noter ici l'utilisation de la classe générique de la bibliothèque C++ SLList<type> qui permet d'établir une liste d'objets de type défini par l'utilisateur.

Chaque instance d'une classe est manipulée par l'instance de la classe auquelle elle appartient. Par exemple les instances de la classe 'World' sont manipulées par les instances de la classe 'Object' (création = méthode MakeWorld), les instances de la classe 'Instance' sont manipulées par les instances de la classe 'World' (création = méthode MakeInstance). Toutes les méthodes de manipulation se retrouvant au plus haut niveau, dans la classe élément.

Les classes "Attribute"

Ces classes sont constituées d'une classe générique "Attribute" regroupant tous les éléments communs et de classes héritant de cette classe générique et spécialisée pour manipuler les types autorisées dans le système COALA, à savoir Entier, Réel, Chaîne de Caractères, Liste d'Entiers, Liste de Réels et Liste de Chaîne de Caractères. Le schéma de la figure 4.11 explicite cet héritage.

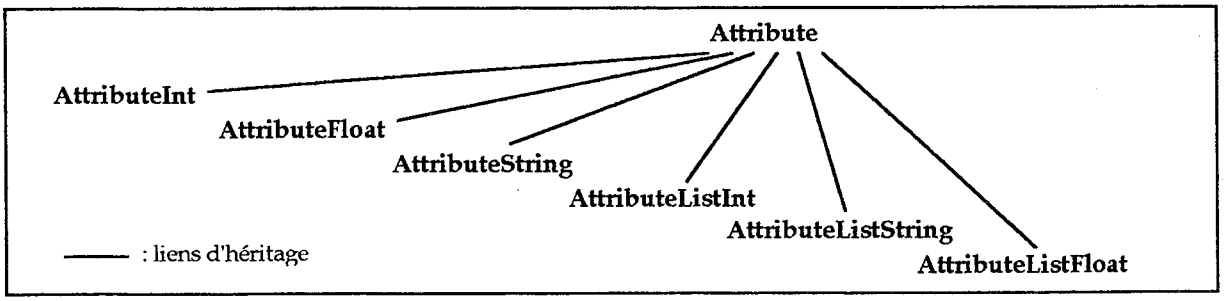

Figure 4.11 : liens entre classes attributs.

La classe 'Attribute" :

```
class Attribute {
protected :
    String              Name;
    String              Comment;
    UNSIGNED    IsValued;
    UNSIGNED    IsDefaulted;
    UNSIGNED    IsRanged;
public :
    virtual UNSIGNED Set (int) = 0;
    virtual UNSIGNED Get (int&) = 0;
    virtual UNSIGNED Set (float) = 0;
    ...
    void SetComment (String& c);
    void SetInitFlag ( );
}
```

La classe Attribute a, en données membres, un nom, un commentaire sous forme de chaîne de caractères et trois indicateurs pour déterminer si l'attribut est valué, s'il possède une valeur par défaut et s'il détient une gamme de valeur. Parmi les fonctions membres on trouve une initialisation des indicateurs, du commentaire et une méthode pour prédéfinir les fonctions de base Set et Get pour chaque type d'attribut possible à savoir entier, réel, chaîne de caractères, liste d'entiers, liste de réels, liste de chaîne de caractères. Cette classe est abstraite et ne peut être utilisée directement (pas d'instance possibles de cette classe). On doit obligatoirement l'utiliser via un héritage de classe. C'est ce qui est fait avec les classes 'Attribute' spécialisées.

Les classes 'Attribute' "spécilisées" :

Ce sont **AttributeInt** pour les attributs de type entier, **AttributeFloat** pour les attributs de type réel, **AttributeString** pour les attributs de type chaîne de caractères, **AttributeListInt** pour les attributs de type liste d'entiers, **AttributeListFloat** pour les attributs de type liste de réels et **AttributeListString** pour les attributs de type liste de chaînes de caractères. Nous donnons en exemple la classe AttributeInt.

```
class AttributeInt : public Attribute {
protected :
    int                         Value;
    int                         Default;
    Interval<int>*      Range;
public :
    AttributeInt(String&) ;
    ~AttributeInt(String&) ;
    virtual UNSIGNED Set (int a);
    virtual UNSIGNED Get (int& a);
    UNSIGNED Set (float a) {return 0};
    ...
    void SetRange (int a, int b);
    void SetDefault (int a);
}
```

La classe AttributeInt hérite de Attribute et contient en données membres, sa valeur, sa valeur par défaut, sa gamme de valeur. On trouve la redéfinition des fonctions Set et Get spécifiques à un attribut, entier en l'occurence. Ces méthodes sont également virtuelles pour laisser a l'utilisateur la possibilité de les redéfinir si le besoin s'en fait sentir. Les autres fonctions Set et Get renvoyant la valeur 0 et n'étant là que pour permettre la présence de toutes les méthodes virtuelles dans la classe de base.

L'affectation des valeurs se faisant par un constructeur pour la valeur, et par les méthodes adéquates pour la valeur par défaut et la gamme.

1.2.3 - Les classes "Connection"

Ces classes servent à exprimer les accointances d'un agent. Ce sont en fait les agents auxquels il est connecté, les informations réunies dans ces classes étant outre le nom des agents, des informations inhérentes au mode de fonctionnement telles que numéro de port et machine. Les communications sont actuellement basées sur le principe des sockets du système UNIX. Le schéma figure 4.12 résume les liens entre ces différentes classes.

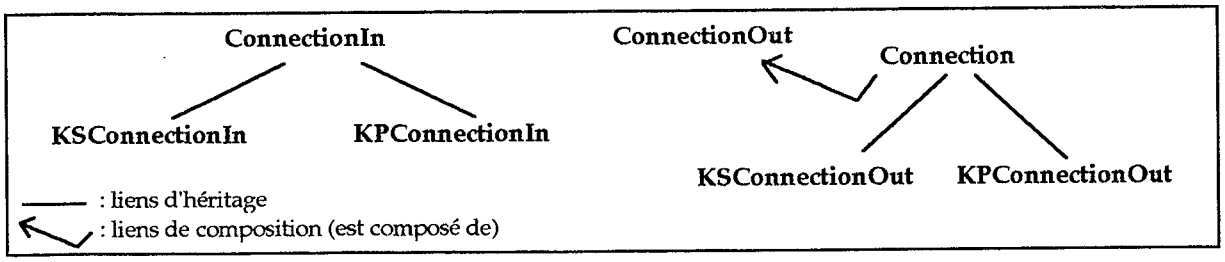

Figure 4.12 : les classes 'Connection'.

Les classes '...In' représentent l'aspect serveur de l'agent, ce sont les réceptions de messages qui lui sont destinées. Les classes '...Out' représentent l'aspect Client, se sont les messages expédiés vers d'autres agents.

Les classes 'ConnectionIn' et 'ConnectionOut' sont des classes génériques regroupant toutes les informations (données et méthodes) inhérentes aux types de communication (sockets). 'ConnectionIn' permet de maintenir les sockets d'écoute dans une liste d'attente avec toutes leurs caractéristiques (Port, Canal, ...). 'ConnectionOut' permet d'établir un point de communication (socket) avec toutes leurs caractéristiques (Port, Canal, ...), avec le ou les agents destinations.

La classe 'Connection' est une classe abstraite qui détient tous les appels aux protocoles liés aux types de messages possibles échangés entre agents.

Les communications :

Elles s'effectuent sur le domaine INTERNET en mode connecté. Le domaine Internet permet l'implantation d'une application COALA sur un réseau de stations de travail (sous réserve des droits d'accès) et le mode connecté assure une fiabilité dans les comunications. Le protocole utilisé par le système sous-jacent est un protocole de la couche transport appelé TCP (Transport Control Protocol) qui offre un service sûr de transport de flots d'octets [RIFFLET 90]. Les octest émis d'un coté de la connexion sont délivrés dans le même ordre de l'autre coté de celle-ci. Ce flot d'octets n'a par ailleurs aucune structure. La connexion est réalisée en mode duplex : elle supporte donc une communication simultanée dans les deux sens. Cependant la connexion s'établit suivant un ordre bien précis assurant une fiabilité et un flot continu d'informations. Cet ordre schématiquement est le suivant, vu à la fois coté client et serveur :

Serveur		Client
création d'une socket d'écoute,		Création d'une socket
Ouverture du service,		construction de l'adresse serveur
attente de demande de connexion		demande de connexion
	connexion	*accepté*
	DIAL	OGUE
fin de la connexion ou attente		fin de la connexion

1.2.4 - Les classes "Rule"

Ces classes expriment l'ensemble des règles qui pourront être utilisées dans le langage d'agent décrit dans le paragraphe 2 suivant. Ces classes sont représentées dans le schéma figure 4.13.

Toutes les primitives du langage d'agent seront traduites par des méthodes de la classe générique 'Rule'. C'est une classe purement abstraite, toutes les fonctions qui représentent les primitives du langage d'agent sont abstraites.

Les classes 'KSRule' et 'KPRule' redéfinissent ces fonctions. La différence entre ces deux classes étant due au fait que les manipulations de mondes, d'instances, d'attributs depuis un agent KP se font obligatoirement vers les éléments d'un agent KS, alors qu'un agent KS peut envoyer des informations vers un autre agent KS, vers un agent KP ou à lui-même.

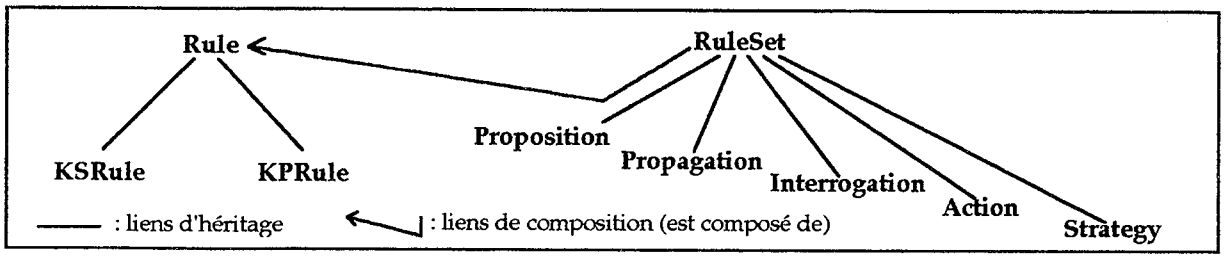

Figure 4.13 : Les classes 'Rule'.

Lorsqu'il y a envoi d'un agent vers un autre agent les méthodes des classes 'Rule' activent les méthodes des classes 'Connections'. Lorsqu'il y a envoi en interne (KS) les méthodes de la classe 'KSRule' activent les méthodes de la classe 'Element'.

La classe 'RuleSet' est une classe générique pour la construction d'une liste de règles conforme à l'application lors de la construction des agents. Elle contient les méthodes nécessaires telle que 'AddRule' qui lors de l'instanciation d'un agent construirons la liste de règles exploitables par le moteur d'inférence. Les classes 'Proposition' 'Propagation' 'Interrogation' 'Action' 'Strategy' ne sont là que pour spécialiser la classe générique.

1.2.5 - Les classes "Engine"

Ces classes expriment les moteurs d'inférences des deux types d'agents. Une hiérarchie (figure 4.14) détermine une classe générique 'Engine', qui contient deux listes : une liste de l'ensemble des régles et une liste des règles de l'ensemble de conflit. Les méthodes assurent le fonctionnement de l'ensemble. Les classes 'KSEngine' et 'KPEngine' se différencient par le fait que l'une manipule des règles de propagation, proposition et interrogation, et l'autre des d'actions et de stratégie. De plus le fonctionnement des moteurs est différent.

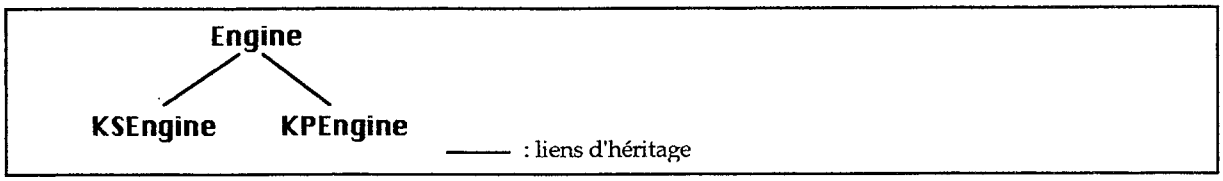

Figure 4.14 : Les classes 'Engine'.

Le moteur d'inférence d'un agent KS fonctionne de la manière suivante pour ses trois types de règles, propagation, proposition, interrogation :

1- Recherche d'un ensemble de conflit parmi l'ensemble des règles de propagation.
2- Activation des règles de l'ensemble de conflit.
3- Reprise du cycle jusqu'à l'obtention d'un ensemble de conflit vide.

Le moteur d'inférence d'un agent KP fonctionne de la manière suivante :
-pour ses règles de stratégie :

1- Définition de l'ensemble de conflit parmi les règles de stratégie.
2- Application des règles de l'ensemble de conflit et obtention des restrictions.
3- Définition du sous ensemble de règles d'action.
4- Arrêt et paramétrage du moteur d'inférence avec le sous ensemble de règles d'action.

-pour ses règles d'action :

1- Définition de l'ensemble de conflit parmi les règles d'action retenues.
2- Application des règles de l'ensemble de conflit.
3- Arrêt.

1.3 - Finalité de la librairie de classes C++

L'objet même de cette librairie est de procurer un outil qui peut être exploité par l'utilisateur de deux façons différentes. Premièrement de façon complètement transparente a travers le langage d'agent vu au paragraphe 2 suivant, un exemple de la correspondance entre exploitation de la libraire et langage y est donné. Deuxièmement par écriture directement en langage C++, en utilisant le mécanisme d'héritage de ce langage à partir des classes génériques proposées. Nous espérons que l'abstraction fournie par le langage d'agent sera suffisante et que la description des classes permettra à un utilisateur de profiter pleinement de tous ces avantages dans le développement d'une résolution de problème en univers Multi-Agents.

2 - Langage d'agents

2.1 - Introduction

D'un point de vue strictement programmation le concept d'objet a vu la création de langage pour manipuler ces entités, le concept d'acteur comme objet actif a également vu paraître des langages d'acteurs. Il est logique que le principe d'agent voit surgir des langages d'agents qui permettent de penser en termes d'agents. Les langages d'acteurs sont utilisés pour la spécification d'agents, ils font figure d'outils pour l'implantation des agents. Mais un modèle d'acteur est minimal, il manque des notions de représentation des autres agents, de puissances de raisonnement sur soi et les autres, toute chose qui doivent être présentes pour parvenir à la notion d'agents.

Le langage doit donc permettre, par l'intermédiaire d'un ensemble de mots-clés et de primitives, de développer une application, en l'occurrence un certain nombre d'agents.

Le développement de ce langage est donc lié à la structure des agents COALA.

2.2 - Rappel sur les agents COALA

Une application est un réseau d'agents KS et KP. Les agents sont composés des éléments suivants :

Agent KS	Agent KP
- une liste d'objets, - une liste d'accointances, - une liste de règles, - un moteur d'inférence, - une liste de procédures (éventuelles) - une redéfinition (éventuelle) des comportements qui lui sont propres.	- une liste d'accointances, - une liste de règles, - un moteur d'inférence, - une liste de procédures (éventuelles) - une redéfinition (éventuelle) des comportements qui lui sont propres.

Tableau xx.

Remarques :

Les objets sont eux-mêmes constitués d'attributs. Les instances des objets peuvent constituer des mondes. Plusieurs mondes peuvent être maintenus dans un agent.

Les accointances sont les agents avec lesquels l'agent peut communiquer. Il doit connaître la situation de ces agents en termes de localisation (emplacement physique sur une machine) et les canaux de communication nécessaire pour établir ces communications.

Les règles sont du type règles de production et s'expriment donc sous la forme d'une prémisse et d'une conclusion.

Le moteur d'inférence est transparent à l'utilisateur, car explicitement précodé.

Les procédures sont ici des procédures "classiques" internes à un agent, auxquelles il peut faire appel depuis ses règles.

Les comportements sont prédéfinis,mais on laisse à l'utilisateur la possibilité de les redéfinir suivant ses propres besoins.

En tenant compte de ces remarques et de la distinction agent KS, agent KP, le langage COALA a été développé.

2.3 - Généralités sur le langage

La syntaxe de COALA a été voulue proche de celle du C++. Celui-ci étant le langage sous-jacent, l'ensemble des classes a déterminé les mots-clés du langage. On retrouvera une correspondance assez exact entre les termes du langage et les noms des classes de la librairie et de leurs méthodes.

On peut subdiviser la description (écriture) d'un agent en deux parties :

- une exprime le coté statique, purement descriptif, que sont les objets, leurs instances et les mondes, et les accointances
- l'autre exprime le coté dynamique, dans les règles.

C'est dans les règles que se verra la puissance d'expression du langage qui permet d'exprimer des stratégies diverses évoquées lors du chapitre sur MAPS.

De façon générale tous les éléments d'un agent y compris lui-même seront décrits de la façon suivante :

```
ENTITE Nom < TYPE > Options {
Commentaire
Contenu...
} ;
```

Ainsi **ENTITE** correspondant à un agent, un objet, un attribut, une règle, une accointance (définie par une classe C++). Chacun de ces éléments possède un **type** (également une classe). **Nom** est le nom donné par l'utilisateur, sous forme d'une chaîne de caractères à l'entité et définira une classe nouvelle qui héritera des classes précédente. Cette classe aura des éléments déterminés par **Options**, **Commentaire**, et le **Contenu**. Le commentaire est une possibilité que l'utilisateur a de donner à chaque objet qu'il créera lors de l'instanciation de la classe et qui pourra être utilisé lors du déroulement de l'application.

Le contenu des règles sera exprimé de la façon suivante :

IF Prémisse THEN Conclusion

Les paragraphes suivants décrivent les principales entités et les règles d'écriture des règles. On trouvera en annexe B la grammaire complète du langage et en annexe A le détail du langage d'agents.

2.4 - Description d'un agent

2.4.1 - Entité Agent

La syntaxe suivante exprime qu'un agent est constitué d'un nom qui le référence dans toute l'application, d'un type (deux types d'agents prédéfinis KP et KS) et d'un contenu :

AGENT nom de l'Agent *<type >* { contenu };

Le contenu est le suivant :

pour un agent KS :
- liste d'objets,
- liste d'accointances,
- liste de règles,
- redéfinitions éventuelles de comportements,
- liste de procédures.

pour un agent KP :
- liste d'accointances,
- liste de règles,
- redéfinitions éventuelles de comportements
- liste de procédures.

2.4.2 - Entité Objet

Un objet est connu sous un nom et s'exprime comme suit :

OBJECT nom de l'Objet { contenu };

Le contenu d'un objet est une liste d'attributs.

Pour un objet on peut prédéfinir les mondes et les instances de la façon suivante :

```
WORLD Nom de l'objet {
    INSTANCE {
        Attribut1 Valeur1;
        Attribut2 Valeur2;
        ...
    } ;
    Instance suivante ...
} ;
```

Attribut1,2,... sont les attributs de l'objet.

Si on ne précise pas le monde ce seront les instances du monde courant qui seront prises en compte. Le but est de pouvoir démarrer une application avec une initialisation correspondant à un cadre précis, par exemple le contexte d'une précédente application.

2.4.3 - Entité Attribut

Un attribut est connu sous un nom est peut être défini avec une valeur par défaut, une gamme qui délimite l'excursion de sa valeur et un type :

ATTRIBUTE NomAttribut *<type* > (*Valeur*) [*Limite basse ..Limite haute*] { contenu } ;

Les types sont les suivants : entier, réel, chaîne de caractères, liste d'entiers, liste de réels et liste de chaîne de caractères.

2.4.4 - Accointances

Les accointances d'un agent sont réunies dans une liste de nom des agents avec lequel il peut communiquer.

ACQUAINTANCE NomConnection1,...;

2.5 - Description des règles

2.5.1 - Généralités sur les règles

Les règles servent a exprimer la façon dont sont exploitées les données. Ces données sont modélisées dans des *attributs*. Ces attributs sont détenus par des *objets* dans un *agent* KS. On manipule des *instances* de ces objets. Ces instances peuvent être regroupées au sein de *mondes*.

Les instructions portent donc sur des manipulations de ces éléments (agent, objet, attribut, instance, mondes). Les opérations sont des créations, destruction, affectation, lecture de ces éléments.

Une règle s'exprime par :

RULE Nom < type > { **IF** Prémisse **THEN** Conclusion };

Elles sont nommées, Nom est une chaîne de caractères qui servira à référencer la règle, et possèdent un type qui est :

- Propagation, Proposition et Interrogation, pour un agent KS,
- Action et Stratégie, pour un agent KP.

Le fonctionnement général d'une règle est le suivant, Prémisse et Conclusion retournent la valeur 0 ou 1. La partie Conclusion n'est exécutée que si la partie Prémisse rend la valeur 1.

Une prémisse ou une conclusion de règle est constituée d'une instruction ou d'un ensemble d'instructions (expressions) reliées par les opérateurs logiques && (ET) et || (OU). Il est également possible d'utiliser la négation ! (NON) en prémisse d'une règle.Il est enfin possible de définir une priorité avec des parenthèses.

Les différentes instructions possibles sont :

- Définition de paramètre,
- Affectation de paramètre,
- Accès aux méthodes,
- Appel de procédures internes ou externes,
- Méthode de recherche,
- Les envois de messages,

Ces instructions peuvent être utilisées a l'intérieur de boucles et d'expressions booléennes.
On peut regrouper des instructions dans un bloc.

A l'intérieur d'une règle toute référence à un agent, objet, attribut, paramètre... se fera de deux façons :

- soit par le nom qu'on lui a donné,
- soit par un paramètre que l'on aura initialisé avec la valeur de l'index de cet élément. Cet index doit être de type entier.

Dans la suite de ce paragraphe il faut noter que toute référence à un objet dans une règle quelconque se fera sous la forme référence Agent-Objet.
Pour un agent KS on aura (Agent , Objet) exprimé de la façon suivante :

- (Self , O) pour un accès interne à l'objet O,
- (A , O) pour un envoi de l'objet O vers un agent KS de nom A

Pour un agent KP l'expression (Agent , Objet) sera toujours exprimée :

- (A , O) pour un accès à l'objet O d'un agent KS de nom A. (Puisque l'agent KP ne peut pas, par construction, posséder en propre d'objets).

exemples :

```
FLOAT  A = GET ( NomAgent,NomObjet,NomAttribut);
        ( A prend la valeur détenue par l'attribut spécifié )
INT  B;
        ( on crée un entier B non initialisé )
B = MAKE INSTANCE OF ( Self,NomObjet );
        ( B prend le numéro de l'instance créée  en interne)
INT  C = ( NomAgent );
        ( C prend la valeur de l'index représentant de manière interne l'agent )
```

2.5.2 - Types de base

Les types de base que l'on peut manipuler sont les entiers, les réels, les chaînes de caractères et les listes homogènes des types précédents.

exemples :

```
INT A = 123;                                  (entier)
FLOAT B = 45.67;                              (réel)
STRING = "chaîne de caractères";              (chaîne de caractères)
INTLIST = ( 1 23 4 567 );                     (liste d'entiers)
FLOATLIST = ( 12.3 45.67 8.9 );                    (liste de réels)
STRINGLIST = ( "abc" "defg"  "hijklm" );      (liste de chaînes de caractères)
```

2.5.3 - Manipulation de listes

Les listes sont particulièrement intéressantes à manipuler, elles permettent de stocker des listes d'agent , d'objets, d'attributs, d'instances, de mondes pour une exploration systématique à l'intérieur d'un boucle par exemple. Pour la manipulation de ces listes, une syntaxe proche de LISP a été retenue, LISP est en effet un langage pour manipuler des listes. Notre but n'est pas de fournir toute l'étendue d'un langage de liste mais de faciliter la tâche d'un utilisateur pour écrire des règles aisément, seuls quelques opérateurs ont été gardés.

Les paragraphes suivants décrivent les opérateurs qui ont étés retenus.

exemples si a, b, c sont des listes d'entiers :

Constructeur

```
INTLIST a;              une liste d'entiers 'a' vide.
INTLIST a = ( 1 2 3 );  une liste d'entiers 'a' initialisée avec trois valeurs.
```

Affectation

```
a = b;          affectation de la liste 'b' à la liste 'a', résultat : deux listes identiques
```

Opérateur

```
CAR ( a );      tête de la liste 'a'
```

CDR (a); *queue de la liste 'a'*

Status

NULL (a); *rend la valeur 1 si la liste 'a' est vide, 0 sinon*

Opération

APPEND (a b); *la liste 'b' est concaténé à la fin de la liste 'a'*

2.5.4 - Paramètres des règles

Un paramètre est défini comme une variable d'un des types standard, entier, réel, chaîne de caractères ou liste d'un de ces types, qui peut éventuellement être initialisée . Il est également possible d'initialiser un paramètre avec la valeur retournée par les instructions GET, par les différentes méthodes de manipulation (GET INSTANCE OF, MAKE INSTANCE OF, ...) et avec la valeur d'un attribut déjà défini. Il est également possible d'initialiser des paramètres de type INT avec l'index représentant de manière interne un agent, un objet ou un attribut. Ce mécanisme est particulièrement utile pour la paramétrisation des instructions manipulant des noms d'agents, d'objets et d'attributs, comme GET ou SET.

2.5.5 - Requêtes

Il s'agit des requêtes qui déclenchent des comportements spécifiques chez l'agent receveur.

Une requête SEND (envoyer) de KS à KP déclenche le comportement Traiter, une requête ASK (demander) déclenche le comportement Résoudre.

Une requête GET (Collecter) de KP à KS déclenche le comportement Fournir, une requête SET (Mettre à jour) déclenche le comportement Recevoir.

Requêtes de KP à KS : GET et SET

Les instructions GET et SET permettent de manipuler des attributs des différentes instances des différents mondes créés pour un objet,

en lecture :

GET (Ag,Ob,Att)
de l'attribut Att de l'objet Ob de l'agent Ag de l'instance courante du monde courant
GET (Ag,Ob,Att) **FROM** (Ins)
de l'attribut Att de l'objet Ob de l'agent Ag de l'instance Ins du monde courant
GET (Ag,Ob,Att) **FROM** (Ins) **IN** (Mond)
de l'attribut Att de l'objet Ob de l'agent Ag de l'instance Ins du monde Mond

en écriture, avec la valeur Val :
(Val peut être une valeur d'un des types de base ou un paramètre).

SET (Ag,Ob,Att) **WITH** (Val)
de l'attribut Att de l'objet Ob de l'agent Ag de l'instance courante du monde courant
SET (Ag,Ob,Att) **FROM** (Ins) **WITH** (Val)

de l'attribut Att de l'objet Ob de l'agent Ag de l'instance Ins du monde courant
SET (Ag,Ob,Att) **FROM** (Ins) **IN** (Mond) **WITH** (Val)
de l'attribut Att de l'objet Ob de l'agent Ag de l'instance Ins du monde Mond

Requêtes de KS à KP : SEND et ASK

SEND (Ob) **TO** (Ag)
demande d'un agent KS de traiter une information Ob à un agent KP Ag
SEND (Ob,Att) **TO** (Ag)
demande d'un agent KS de traiter une information Ob,Att à un agent KP Ag

ASK (Ob) **TO** (Ag)
demande d'un agent KS de résoudre un problème Ob à un agent KP Ag
ASK (Ob,Att) **TO** (Ag)
demande d'un agent KS de résoudre un problème Ob ,Att à un agent KP Ag

2.5.6 - Méthodes

Il s'agit des méthodes de manipulation d'attributs (pour un agent KS en interne), d'instances, de mondes, d'attributs système, de recherche d'instances, qui ne font pas appel à des comportements de la part de l'agent receveur.

Méthodes de manipulation d'attributs

Les attributs se manipulent de façon interne à un KS par les mêmes instructions GET et SET vues ci-dessus, le référence Agent étant dans ce cas lui-même (Self).

exemples :
GET (Self,Ob,Att) **FROM** (Ins) **IN** (Mond)
GET (Self,Ob,Att) **FROM** (Ins) **IN** (Mond) **WITH** (Val)

Méthodes de manipulation de monde

ces manipulations permettent de créer, détruire et se positionner sur un monde, ceci afin de "faire vivre" des mondes d'hypothèses en parallèle. Cette possibilité va permettre d'explorer différents axes d'un même problème et de pouvoir choisir une solution de façon opportune.

exemples :
MAKE WORLD (Ag,Ob)
(pour créer un monde entièrement nouveau)
DELETE CURRENT WORLD (Ag,Ob)
(pour détruire un monde courant)
SET CURRENT WORLD (Ag,Ob) **WITH** (Mond)
(pour se positionner sur un monde particulier)

Méthodes de gestion des instances d'objets

Ces méthodes permettent de gérer des instances courantes ou particulières.

exemples :
Création :
MAKE INSTANCE OF (Ag,Ob) **IN** (Mond)
(créer une instance du monde Mond de l'objet Ob de l'agent Ag)

Mise à jour :
SET CURRENT INSTANCE OF (Ag,Ob) **WITH** (Val)
(met à jour avec la valeur Val l'instance courante du monde courant de l'objet Ob de l'agent Ag). Val peut être un entier ou un paramètre.
destruction :
DELETE CURRENT INSTANCE OF (Ag,Ob)
(détruit l'instance courante du monde courant de l'objet Ob de l'agent Ag)

Méthodes de recherche des instances d'objets

Plusieurs méthodes ont été développées, pour faire une recherche dans des mondes courants, particuliers, et avec les critères de sélection de l'utilisateur.

exemples :
GET INSTANCE OF (Ag,Ob) **IN** (Mond) **WHERE** { cond }
(effectue une recherche de l'instance du monde Mond de l'objet Ob de l'agent Ag et qui vérifie la condition cond)
GET INSTANCE OF (Ag,Ob) **WITH LOWER** (Att) **WHERE** { cond }
(effectue une recherche de l'instance du monde courant de l'objet Ob de l'agent Ag dont l'attribut Att a la plus petite valeur et qui vérifie la condition cond)
GET INSTANCE OF (Ag,Ob) **WITH UPPER** (Att)
(effectue une recherche de l'instance du monde courant de l'objet Ob de l'agent Ag dont l'attribut Att a la plus grande valeur)

Méthodes d'accès aux attributs système

Les attributs système permettent de modéliser l'activité des agents KS. Ils servent à un agent KP pour connaître l'état de l'agent KS concerné et pour l'agent KS lui-même de savoir comment a réagi l'agent KP à son dernier envoi.

exemples :
GET PROPOSITION OF (Ag,Ob)
GET INTERROGATION OF (Ag,Ob)
GET RESULT OF (Ag,Ob)
(pour une lecture des attributs système de l'instance courante du monde courant de l'objet Ob de l'agent Ag)

2.5.7 - Appel de procédures

Deux cas se présentent.

1er Cas :

qui implique l'appel de procédure externes, réutilisation possible de procédures existantes, cas de bibliothèques spécialisées par exemple. C'est un mécanisme de RPC (Remote Procedure Call ou Appel de procédure à distance) qui est utilisé.

L'appel s'effectue de façon simple par le nom de la procédure et une liste de paramètre conforme à la définition des paramètres formels de la procédure :

NomProcédure (Paramètre1, Paramètre2, ...)

2ème Cas :

celui de procédures internes à un agent , écrite dans le corps de l'agent mais en C++. Ce sont des procédures qui ne peuvent être traduites dans des règles.

Appel :
NomProcédure1 (Paramètre1, Paramètre2, ...)

Déclaration :
```
DEFINE PROCEDURE {
        NomProcedure1(a,b,...) {
        ... Corps de la procédure
        }
        NomProcedure2(...) {
        ...
        }
...
};
```

2.5.8 - Les boucles

L'intérêt des boucles est de pouvoir faire des explorations conditionnelles sur des critères définis par l'utilisateur.

Les boucles DO et WHILE

La syntaxe est la suivante :

DO Expression1 **WHILE** Expression2

ou

WHILE Expression1 **DO** Expression2

L'utilisateur peut utiliser des sélections simples ou complexes dans les expressions.

Les boucles FOR

Deux possibilités, itération sur Liste et itération entière.

Itération sur liste :

FOR (i **IN** Liste) { bloc };

i est du type entier, réel ou string, Liste une référence à une liste appropriée (entier, réel ou string), ou une liste explicitement donnée. Pour tout élément de la liste, affecté à i il y a exécution du bloc.

Itération entière :

FOR (i, condition) { bloc };

i est un entier qui s'incrément de 1 à partir de sa valeur initiale, tant que la condition est vraie, le bloc est exécuté.

2.5.9 - Les opérations arithmétiques

Pour les entiers les opérations permises sont : + , - , * , / , % et le - **unaire.**
Pour les réels se sont + , - , * , / et le - **unaire.**

Les expressions arithmétiques sont parenthésées et utilisées lors d'une affectation.

2.5.10 - Les expressions booléennes

Les expressions booléennes sont des comparaisons de deux opérandes et complètement parenthésées. Les opérateurs de comparaisons sont : == , != , < , > , <= , >=, **ELEMENT** , **NON ELEMENT** . Les comparaisons portent sur des Valeurs de type de base, des expressions arithmétiques, des résultats de méthodes ou sur l'appartenance de valeur dans une liste.

2.6 - Correspondance entre langage et librairie

Nous avons déjà évoqué la correspondance étroite entre les termes du langage et les noms des classes de la librairie et de leurs méthodes, nous allons donner un exemple pour exprimer cette correspondance.

Cet exemple n'est en aucun cas tiré d'une application réelle, le but étant seulement de donner un aperçu de la façon de programmer et non pas de montrer les capacités du système COALA en tant que résolution de problème en univers Multi-Agents.

Soit un agent KS nommé 'Agks1' qui contient un objet contenant lui-même trois attributs, il a trois règles et n'est connecté qu'à un agent KP 'Agkp1'. La description de cet agent est développée dans un fichier de ressources portant le nom de l'agent.

Nous ne donnons dans la suite qu'une écriture "allégée" du fichier de ressources, et nous nommons simplement les classes C++ qui sont développées à partir de ce fichier agent. Le fichier est développé en annexe C ainsi que les classes générées.

Fichier de ressources Agent AgKS en langage COALA	fichier et classes C++
AGENT Agks1 <KS> {	---> un fichier agent
OBJECT Ob1 {	---> une classe C_Agks1_Ob1
ATTRIBUTE Att1	---> une classe C_Agks1_Ob1_Att1
ATTRIBUTE Att2	---> une classe C_Agks1_Ob1_Att2
ATTRIBUTE Att3	---> une classe C_Agks1_Ob1_Att3
};	
ACQUAINTANCE Agkp1;	
RULE R1 <PROPAGATION> {	---> une classe C_Agks1_R1
....	
};	
RULE R2 <PROPOSITION> {	---> une classe C_Agks1_R2
....	
};	
RULE R3 <INTERROGATION> {	---> une classe C_Agks1_R3
...	
};	
};	

On trouve donc, une classe pour chaque objet, pour chaque attribut et pour chaque règle. L'agent est développé comme un fichier et fait appel au méthodes d'ajout des différents éléments qui le constituent, à savoir Objet, règle et connection, voir ci-dessous.

```
Main()
{
   KS   agent ("AgKs1", ...);
   (void)  agent. AddObject(new C_Agks1_Ob1 );
   (void)  agent. AddPropagation (new C_Agks1_R1 );
   (void)  agent. AddProposition (new C_Agks1_R2 );
   (void)  agent. AddInterrogation (new C_Agks1_R3 );
   (void)  agent. AddConnection (...);
   (void) agent.KSMainLoop();
};
```

2.7 - Ouverture vers C++

L'exemple précédent nous montre que le gain est substantiel en terme d'écriture, dans un rapport de un à deux. Mais surtout il est important au niveau de l'abstraction, les manipulations portent sur des éléments en rapport avec le concept même du système Multi-Agents COALA.

Cependant dans un souci d'évolution des concepts du système Multi-Agents COALA nous avons voulu laisser une certaine ouverture vers C++. Pour cela l'emploi des méthodes virtuelles de C++ laisse l'opportunité à l'utilisateur de redéfinir les principes mêmes de COALA en réécrivant par exemple les comportements d'un agent. Si cela semble une "porte ouverte" à de multiples possibilités pour transgresser et dénaturer les fondements du système Multi-Agents COALA, il n'en reste pas moins une grande possibilité d'apprentissage.

En définitive ce langage doit permettre d'explorer avec efficacité les solutions à un problème posé en termes de répartition de capacités et de connaissances.

3 - Précompilateur

3.1 - Introduction

Resituons le problème. Parmi le travail que doit effectuer le système COALA en tant qu'environnement de programmation, il est une étape qui permet le passage du niveau d'abstraction agent à celui du niveau d'abstraction C++. La figure suivante 4.15 résume ce passage. Ce passage est effectué par un outil que nous appellerons "précompilateur" ou "traducteur". Précompilateur, car il prépare à la compilation ultérieure qui sera effectuée par le compilateur C++ sous système UNIX. Traducteur, car il effectue la traduction des fichiers de ressources écrits en langage d'agents COALA, en fichiers C++.

La traduction s'accompagne d'un certain nombre de vérification qui assurent l'utilisateur d'une syntaxe correcte dans l'écriture de ses fichiers agents et de cohérence notamment au niveau des différents types employés.

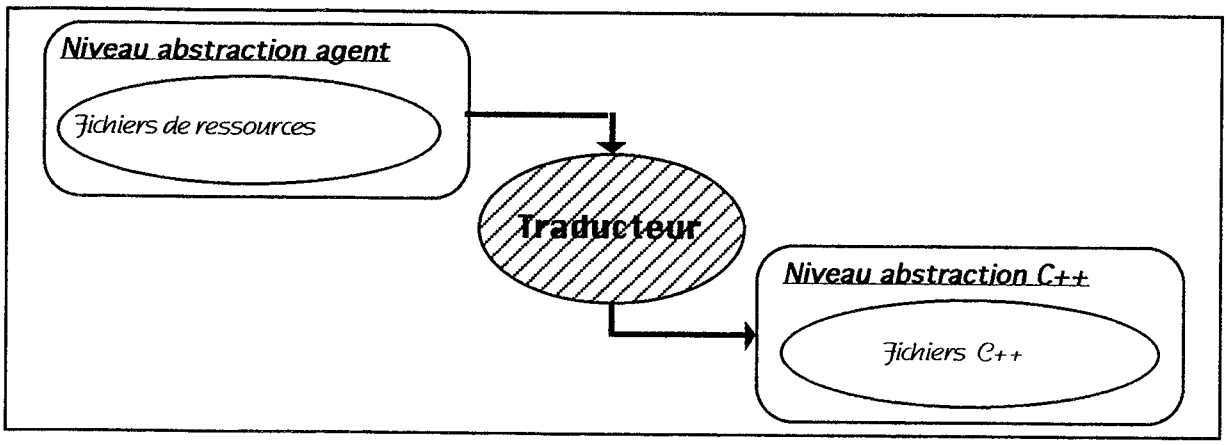

Figure 4.15 : L'étape de précompilation.

Remarque : le compilateur de MAPS :

Dans la dernière version de MAPS il existait un compilateur dont nous allons donner brièvement le fonctionnement [BRUNO 91]. Citons :

"Ce compilateur diffère des compilateurs de langage de programmation. A l'inverse de ces derniers, son but n'est pas la génération d'un code exécutable par l'ordinateur, mais en tant que générateur d'applications pour un autre programme, *la traduction du code* (fichiers de ressources) sous une forme exécutable par l'interpréteur du système MAPS, et non par l'ordinateur".

Certains éléments de ce compilateur seront repris, soit complètement (la table des symboles) soit en partie (l'arbre abstrait) soit dans le principe (la table des dépendances). Nous verrons plus loin comment sont intégrés ces éléments.

Le but ayant changé, il s'agit maintenant d'opérer une traduction d'un langage dans un autre langage et non plus de générer un code "exécutable", le fonctionnement a donc changé.

Auparavant lors de l'exécution d'une application, pour chaque agent et pour chacune de ses règles il y avait interprétation par parcours d'un arbre représentant chaque règle. A l'exécution les règles étaient interprétées. Maintenant le besoin d'obtenir une application entièrement compilable nécessite un résultat de "compilation" sous forme de fichiers C++.

3.2 - Les étapes de précompilation

Tout ce travail se déroule en plusieurs étapes que nous allons énumérer :

- une étape dite lexicographique qui est chargée de lire le code source (langage d'agents COALA) pour reconnaître les éléments du langage. Cette étape différencie les "mots-clés" des autres éléments tels que variables, paramètres, littéraux ou commentaires. Les éléments reconnus sont des chaînes de caractères sans valeur particulière. Ils sont appelés "tokens", syntaxe anglaise de l'outil LEX utilisé (voir plus loin).

- une étape dite "grammaticale" ou syntaxique qui vérifie l'enchaînement des mots reconnus selon une grammaire établie, celle du langage. Dans cette étape il y a stockage des tokens en leur attribuant un sens qui servira pour effectuer des contrôles par la suite.

- une étape de construction d'une structure arborescente correspondant à une représentation de l'application complète. Cet arbre exprime les dépendances de construction de l'application en agents, des agents en objets, règles ... A chaque noeud ou feuille de l'arbre sont associées des informations sur la dépendance des éléments, sur l'enchaînement des actions, sur les actions elles mêmes ... De plus tout au long de cette étape de construction un certain nombre de vérifications sont effectuées, notamment sur l'unicité des déclarations, des cohérences d'utilisation, d'affectation... chaque fois que cela est possible.

- une étape d'analyse de l'arbre qui effectue les vérifications de cohérence qui ne peuvent être faites qu'une fois la totalité de l'arbre construit, c'est-à-dire sur l'application complète. Il s'agit de vérifier que tous les éléments sont bien définis, que toutes les affectations sont correctes ...

- Enfin la dernière est celle de traduction proprement dite. Elle effectue un parcours de l'arbre et construit au fur et a mesure un ensemble de fichiers correspondant aux principaux constituants d'un agent. Le code produit est du C++.

Toutes ces étapes seront effectuées au cours de trois passages :

- le premier passage regroupe les trois premières étapes citées ci-dessus, à savoir analyse lexicale, syntaxique et construction de l'arbre,
- le deuxième passage réalise l'analyse de cohérence,
- et le troisième passage terminera avec la traduction.

La figure 4.16 résume ces opérations.

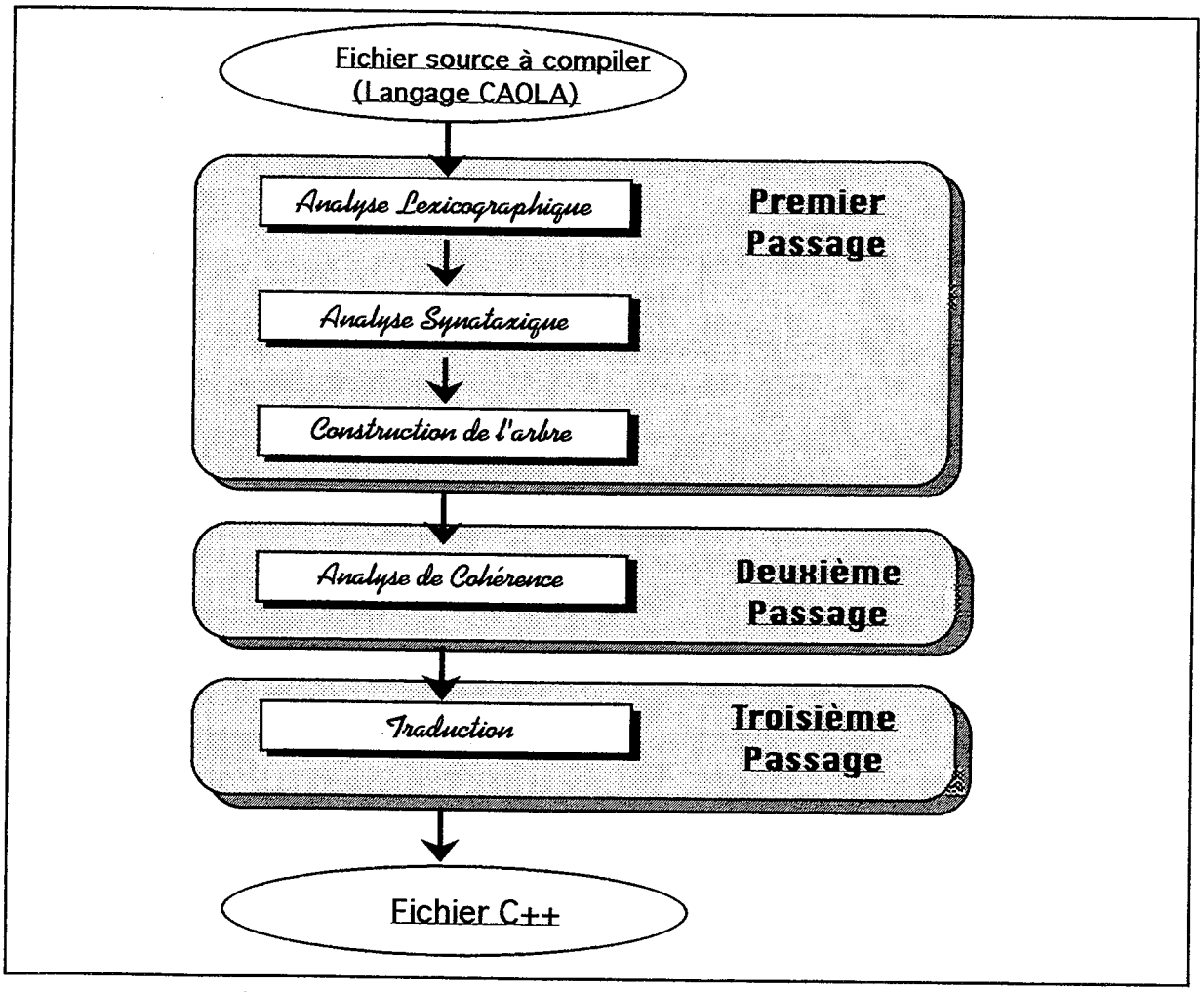

Figure 4.16 : étapes et passages faits par le précompilateur.

3.3 - Les structures du précompilateur

Le précompilateur s'articule autour de deux structures essentielles, une table des symboles et l'arbre application qui contient entre autre deux sous-arbres décrivant les règles dont une partie a été reprise du compilateur MAPS. La table des symboles a été reprise telle quelle.

3.3.1 - La table des symboles

Dans un formalisme objet, c'est un objet 'Table' stockant des chaînes de caractères associées à un indice. Tous les accès à cette table se font par des méthodes dédiées qui effectuent des opérations de vérifications éventuelles. La figure 4.17 suivante montre cette encapsulation.

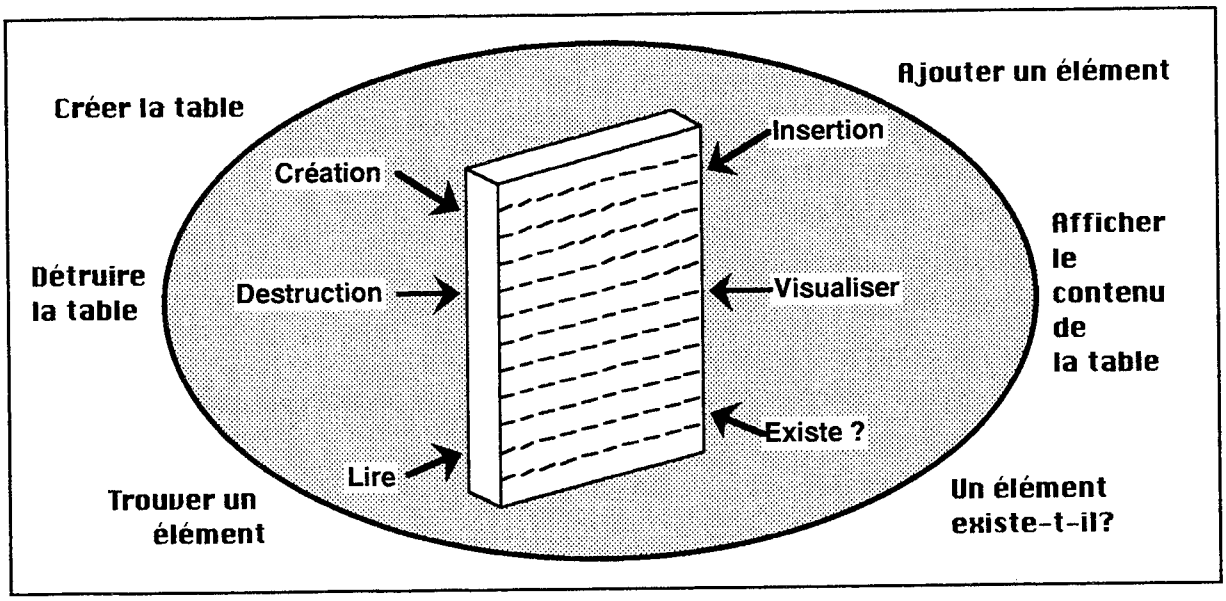

Figure 4.17 : La table des symboles.

Tout accès à la table des symboles (TS) se fait moyennant un indice (entier), ou une chaîne de caractères. Outre la création et la destruction de la table les méthodes autorisent les accès suivants :

- Insérer un nouvel élément : Au premier accès en écriture d'une nouvelle chaîne de caractères, on la fournit et TS rend un indice. Cet indice est calculé par une fonction de ** "Hashcode". Cette fonction est une méthode de stockage de chaînes de caractères permettant un accès direct à la case du tableau contenant une chaîne donnée. A chaque accès suivant pour la même chaîne, une vérification de l'existence de la chaîne est effectuée garantissant l'unicité de celle-ci.

- Trouver un élément : on fournit un indice et TS rend la chaîne de caractères correspondante.

- Existence d'un élément : on donne une chaîne de caractères et la table des symboles TS rend l'indice s'il existe sinon la valeur zéro.

- Afficher la totalité de la table, chaîne de caractères et indice.

De plus des vérifications sur la place disponible sont faites et des vérifications de la validité des chaînes de caractères (exemple, refus d'insérer des chaînes vides).

Cette structure garantit que les chaînes dupliquées sont insérées une seule fois, qu'un indice et un seul correspond à une chaîne et une seule, et surtout permet de ne manipuler que des indices sous forme d'entier ce qui facilite les comparaisons et les manipulations de façon générale.

** Hascode : mot anglais signifiant hacher.
Une fonction de hachage calcule une valeur numérique à partir d'une chaîne de caractères, et sert à déterminer l'endroit où sont stockées les informations ayant cette chaîne pour clé.
exemple : indice i = f(chaîne), ou f() est une fonction mathématique surjective calculée à partir de la chaîne.

3.3.2 - L'arbre application

C'est la structure la plus complexe car elle regroupe toutes les informations relatives à une application. Elle stocke les informations de dépendances, de vérification.

Elle est basée sur le schéma même d'une application :

- une application est un ensemble d'agents
- un agent est constitué :
  -- d'objets eux mêmes composés d'attributs
  -- de règles qui regroupent des paramètres, une prémisse et une conclusion
  -- des accointances.

On aura donc l'arbre suivant figure 4.18

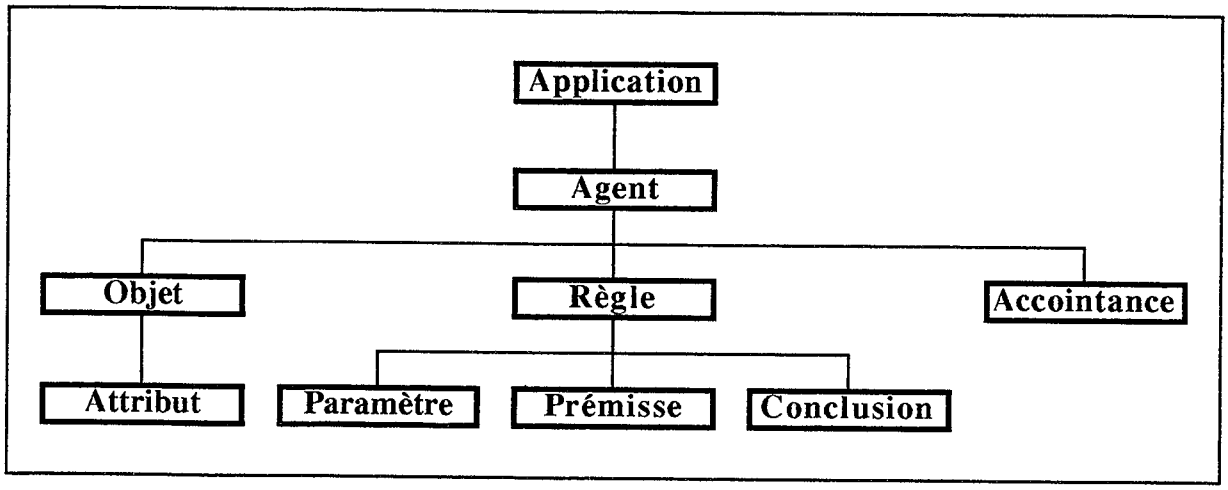

Figure 4.18 : L'arbre application.

Les noeuds Agent, Objet, Attribut, Régle, Paramètre et Accointances sont en fait des listes chaînées.

A la racine le noeud Application détient l'information sur l'état général, sur les noeuds immédiatement inférieurs que sont les agents dont un pointeur sur le premier noeud de la liste Agent et une série de pointeurs qui permettent de mémoriser des pointeurs pour faciliter le parcours de l'arbre.

Les noeuds de la liste 'Agent' ont trois pointeurs vers les listes 'Objet', 'Règle' et 'Accointance'.

Les noeuds de la liste 'Règle' ont trois pointeurs vers une liste de paramètres,et vers les sous-arbres Prémisse et Conclusion.

Les noeuds Prémisse et Conclusion sont des pointeurs vers la racine du sous-arbre représentant les deux parties d'une règle (voir plus loin).

Le détail de chaque noeud est donné dans la figure 4.19. Les informations en gras exprime le chaînage dans l'arbre, les informations en souligné expriment le chaînage dans les listes chaînées.

Application

Etat	Nombre agent	Nombre agents définis	Pointeur agent courant	Pointeur agent préced.	Pointeur objet courant	Pointeur attribut courant	Pointeur règle courante	Pointeur param. courant	Pointeur accoint. courante	**Pointeur liste agent**

Agent

Etat	Indice agent	Type	Numéro	**Pointeur liste objet**	Nombre objet	Nombre objets définis	**Pointeur liste règle**	Nombre règles	Nombre règle Ok	**Pointeur liste accoint.**	Nombre accoint.	Nombre accoint. définies	Pointeur agent Suivant

Objet

Etat	Indice objet	Type	Numéro	**Pointeur liste attribut**	Nombre attribut	Nombre attribut définis	Pointeur objet Suivant

Attribut

Etat	Indice attribut	Type	Numéro	Pointeur Valeur défaut	Pointeur Gamme	Pointeur attribut Suivant

Valeur par défaut	Limite basse	Limite haute

Règle

Etat	Indice regle	Type	Numéro	**Pointeur param.**	Nombre param.	Nombre param. définis	**Pointeur Premise**	**Pointeur Conclusion**	Pointeur regle Suivant

Paramètre

Etat	Indice param.	Type	Numéro	Valeur	Pointeur param. Suivant

Accointances

Etat	Indice accoint.	Type	Numéro	port	machine	Pointeur accoint. Suivant

Figure 4.19 : Structure des noeuds de l'arbre.

3.3.3 - Les sous-arbres Prémisse et Conclusion.

Les noeuds de ces arbres sont constitués de la manière suivante :

Type	Nom	Pointeur 1	Pointeur 2	Valeur	Etat

Figure 4.20 : Noeud des sous-arbres Prémisse et Conclusion.

Type : différencie des noeuds de même syntaxe,
Nom : renseigne sur la sémantique du noeud (action),
Pointeur 1 & 2 : pour accéder aux fils du noeud,

Valeur : contient des littéraux, des indices, des valeurs d'affectations.

A chaque mot-clé de la règle est associé un noeud distinct.

Etat : indique l'état de toutes les vérifications effectuées pour le noeud.

3.4 - Les vérifications effectuées par le précompilateur

Le tableau 4.20 suivant donne la dépendance des différents éléments dans l'arbre application :

Etat de	dépend de	
Agent	- Agent a été défini	#
	- si KS le nombre d'objet n'est pas nul	#
	si KP le nombre d'objet est nul	#
	- Etat de tous les objets ok	*
	- le nombre de règle n'est pas nul	#
	- si KS type règle est Propagation, Proposition, Interrogation	#
	si KP type règle est Action, Stratégie	#
	- Etat de toutes les règles ok	*
	- le nombre d'accointances n'est pas nul	#
	- toutes les accointances existent	*
	- Etat de toutes les accointances ok	*
Objet	- Objet a été défini	#
	- le nombre d'attribut n'est pas nul	#
	- Etat de tous les attributs ok	*
Attribut	- Attribut a été défini	#
	- Valeur par défaut correspond au type déclaré (éventuellement)	#
	- Type des valeurs de gamme correspond au type déclaré (éventuellement)	#
	- Valeur basse < Valeur Haute de gamme (éventuellement)	#
	- Valeur par défaut cohérent avec gamme (éventuellement)	#
Règle	- Règle a été définie	#
	- Prémisse non nulle	#
	- Conclusion non nulle	#
	- Etat de tous les paramètres ok	*
	- Prémisse ok	*
	- Conclusion ok	*
Accointance	- Accointance a été définie	#
	- Port et Machine ont été renseignés	*
	- type accointance cohérent avec type de l'agent	*
Paramètre	- Paramètre a été défini	*
Prémisse	- Tous les noeuds sont ok	*
Conclusion	- Tous les noeuds sont ok	*

nb : # : à vérifier en cours de construction de l'arbre
* : à vérifier après construction de l'arbre complet.

Tableau 4.20 : dépendance des états des différents éléments.

Le champ Etat de chaque noeud de l'arbre (voir arbre application) est initialisé à zéro et incrémenté de un chaque fois qu'une vérification est effectuée.

Par exemple pour l'attribut suivant :

ATTRIBUTE NomAtt < INT > (123) [1..256];

on a Etat = 0 à la création de l'attribut dans l'arbre application,
puis Etat est incrémenté de un car il s'agit d'une déclaration,
puis Etat est incrémenté de un car on vérifie que la valeur par défaut "123" est du type entier
puis Etat est incrémenté de un car les vérifications sur la gamme sont valides à savoir, le type de "1" et "256" est entier, "1 < 245", "123" est dans la gamme "1..256".
En définitive le champ Etat doit avoir une valeur égale à 3 pour être correct dans le cas d'un attribut.

3.5 - Le fonctionnement

3.5.1 - Utilisation de LEX++ et YACC++

Nous avons vu que que le déroulement d'une précompilation s'effectue en trois passages et regroupe cinq étapes. Les deux premières étapes ont été écrites à l'aide des outils UNIX, LEX++ et YACC++ (LEX pour lexicographique et YACC pour "yet another compiler of compiler"). Ce sont deux outils complémentaires qui permettent de faire l'analyse lexicographique et syntaxique d'un "texte". Le suffixe "++" exprime qu'il s'agit des versions objets de LEX et YACC.

LEX++ permet la reconnaissance de chaînes de caractères et fournit à partir des fichiers ressources et d'une description du langage, une description du fichier traité sous forme de tokens (code numérique).

YACC++ effectue une reconnaissance de la grammaire des fichiers traités. YACC++ procède à partir des tokens reconnus par LEX++ et d'une description de la grammaire.

En fait LEX++ et YACC++ sont des outils UNIX qui permettent la génération de fichiers sources d'analyse lexicale et syntaxique. LEX++ donne un fichier source d'analyse lexicale et YACC++ donne un squelette de compilateur, fichiers sources d'analyse syntaxique.

La démarche est la suivante :

- écriture d'un source "compilé" par LEX++, donnant un fichier "lex.yy.c"
- écriture d'un source "compilé" par YACC++, donnant un fichier "y.tab.c" et un fichier "y.tab.h". Le fichier lex.yy.c est inclus dans le code de y.tab.c .
- compilation par le compilateur C++ "CC" de "y.tab.c" pour obtenir un fichier exécutable qui réalisera les analyses lexicale et syntaxique.

De plus, en même temps que s'effectuent les étapes précédentes il a été ajouté un contrôle sémantique et une construction d'un arbre en mémoire représentant l'application complète.

3.5.2 - Algorithmes utilisés

Nous donnons comme exemple, l'algorithme utilisé pour l'insertion d'éléments dans l'arbre.

Lors du parcours du fichier en cours de traitement, chaque fois que l'on trouve une référence Agent, Objet, Attribut ... par exemple dans une déclaration,

OBJECT NomObjet { ... };

ou dans une règle,

GET (NomAgent,NomObjet)

on utilise l'algorithme suivant pour rechercher et ajouter l'élément dans l'arbre :

```
- insertion dans la table des symboles
- exploration de l'arbre :
      si l'élément est non trouvé :
            Créer nouvel élément en fin de liste
            L'initialiser avec les paramètres éventuels
            s'il s'agit d'une déclaration :
                  Etat de l'élément ajusté à déclaré
            sinon (il s'agit d'une référence dans une règle)
                  Etat de l'élément ajusté à référence
      si l'élément est trouvé
            s'il s'agit d'une déclaration :
                  si Etat = déjà déclaré
                        alors erreur (c'est une double définition)
                  sinon (Etat = référencé)
                        alors Etat ajusté à déclaré et référencé
            sinon (il s'agit d'une référence dans une règle)
                  ok.
```

3.6 - Bilan

Lors de la rédaction du présent mémoire, le précompilateur n'est pas complètement terminé. Il reste à vérifier que le fonctionnement global est bien celui attendu, en réalisant une série de tests qui permettent de contrôler que la correspondance entre le code source en langage d'agents et code C++ est correcte.

4 - L'interface graphique

4.1 - Introduction

L'interface dont nous parlons ici est une interface utilisateur, c'est le premier contact que l'utilisateur aura avec l'environnement COALA. Elle doit donc offrir un certain nombre de fonctionnalités pour que l'utilisateur puisse accéder à l'ensemble des outils proposés.

Nous verrons dans un premier temps les objectifs que nous nous sommes fixés, puis les choix qui ont du être fait, en fonction de ces buts et des moyens dont nous disposons, avant d'aborder une vue de l'ensemble réalisé. Nous finirons par une évaluation des problèmes rencontrés, très spécifiques au développement de ce travail.

Lors de la réalisation, j'ai eu à apprécier la collaboration de Rémi PINCK, stagiaire de l'IUT informatique de Grenoble que je remercie au passage.

Parmi les outils de l'environnement COALA, tous n'ont pas été développés. L'accent a été mis sur les outils permettant l'évolution dans le cadre de la méthodologie proposée.

4.2 - But

L'interface envisagée est la première couche de l'architecture vue au paragraphe III de ce chapitre, elle doit permettre l'accès à tous les outils de l'environnement COALA. Mais de plus elle doit être calquée sur la méthodologie proposée de façon à guider un utilisateur et garder en vue le fait que COALA peut aussi être utilisé par des personnes compétentes en programmation C++.

La conception de COALA comme un environnement plus orienté utilisateur qu'expérimentation nécessite de concevoir une interface agréable et aisée à utiliser, et fonctionnant comme d'autres interfaces actuelles (présences de menus déroulants, multifenêtrage, graphisme...).

On a donc une exigence sur une accessibilité rapide et pratique des outils, une exigence sur l'organisation de ces outils en fonction de la méthodologie et une troisième exigence d'ouverture potentielle vers les outils et l'environnement UNIX. Le fait que la vision du système COALA se fait à travers l'interface pose des exigences de présentation pour ne pas rebuter l'utilisateur.

4.3 - Choix

Les choix se sont faits à partir des exigences citées, et en fonction de contraintes et de moyens disponibles .

Les contraintes sont de plusieurs ordres. Premièrement nous travaillons dans un domaine d'applications multitâches qui exploitent les fonctionnalités du système d'exploitation UNIX, il faut prendre en considération le matériel à notre disposition. Deuxièmement comme il ne s'agit en aucun cas de réécrire une interface graphique à partir de rien, il faut envisager les interfaces graphiques existantes.

Nous entrons ainsi dans le domaine des moyens à notre disposition. Le matériel est le suivant : les stations de travail sont de marque SUN, le type étant SPARC II et IPC et une station de marque SONY (voir annexe E).

Parmi les "standards graphiques" disponibles sur le marché on trouve OPENLOOK et OSF/MOTIF qui se disputent la suprématie de l'interface graphique dans le monde UNIX. Si les efforts de SUN MicroSystèmes se sont portés sur OPENLOOK de récents accords commerciaux orientent SUN vers MOTIF. Ce dernier est actuellement un "standard" plus répandu qu'OPENLOOK.

Cependant si les stations SUN sur lesquelles est implanté OPENLOOK possèdent un générateur d'interface graphique, la station SONY ne dispose pas d'interface mais d'un langage de programmation qui manipule des objets graphiques.

La programmation sur station SUN s'effectue de deux manières. Soit par l'intermédiaire du générateur, une application étant alors figée très rapidement, il est difficile de faire évoluer un élément sans toucher à l'ensemble. Cet inconvénient est un peu compensé par la facilité d'utilisation de l'interface. Soit par un programme, en langage C par exemple, faisant appel aux outils de la boîte à outils XView élaboré par SUN. La programmation est alors laborieuse.

Sur la station SONY l'interface Motif s'utilise avec un langage de programmation appelée UIL (User Interface Language) qui permet de manipuler des objets graphiques procurant une abstraction. Ces objets graphiques sont à "lier" par du code C ou C++. Le résultat produit en terme de code est très modulaire et facile à faire évoluer. Il reste néanmoins le fait que l'apprentissage est long.

Toutes ces contraintes nous ont amenés à faire le choix de MOTIF pour sa plus grande expansion dans le monde informatique, augmentant les capacités de portages futurs éventuels vers d'autres matériels. La disponibilité de la machine SONY est également plus importante.

De plus son style de programmation, objet (langage UIL), est dans une optique de modularité donc de réutilisation.

4.4 - Réalisation

L'utilisation de Motif.

MOTIF est une couche au dessus des couches X Windows et permet de manipuler des objets graphiques tels que fenêtre, menu, ascenseur ..., en fait des objets représentant des concepts précis d'une application. Les objets MOTIF appelés "Widgets" et "Gadjets" font appel aux éléments des couches inférieures. Le schéma figure 4.21 montre cet empilement.

Motif	Motif Window Manager	(MWM)
	bibliothèque Widgets	(Xm)
X Windows	Xt Intrinsics	(Xt)
	Xlib	(Xlib)
Système d'exploitation	UNIX	

Figure 4.21 : Les couches des interfaces graphiques.

Les objets graphiques de Motif sont utilisés avec le langage UIL (User Interface Language) qui permet de décrire une interface graphique MOTIF. Ce langage est "géré" par un programme en code C qui notamment décrit les actions de l'utilisateur par des fonctions appelées "CallBack".

Schéma global

Des exigences citées précédemment, notamment sur la méthodologie, nous en avons retiré le schéma suivant figure 4.22.

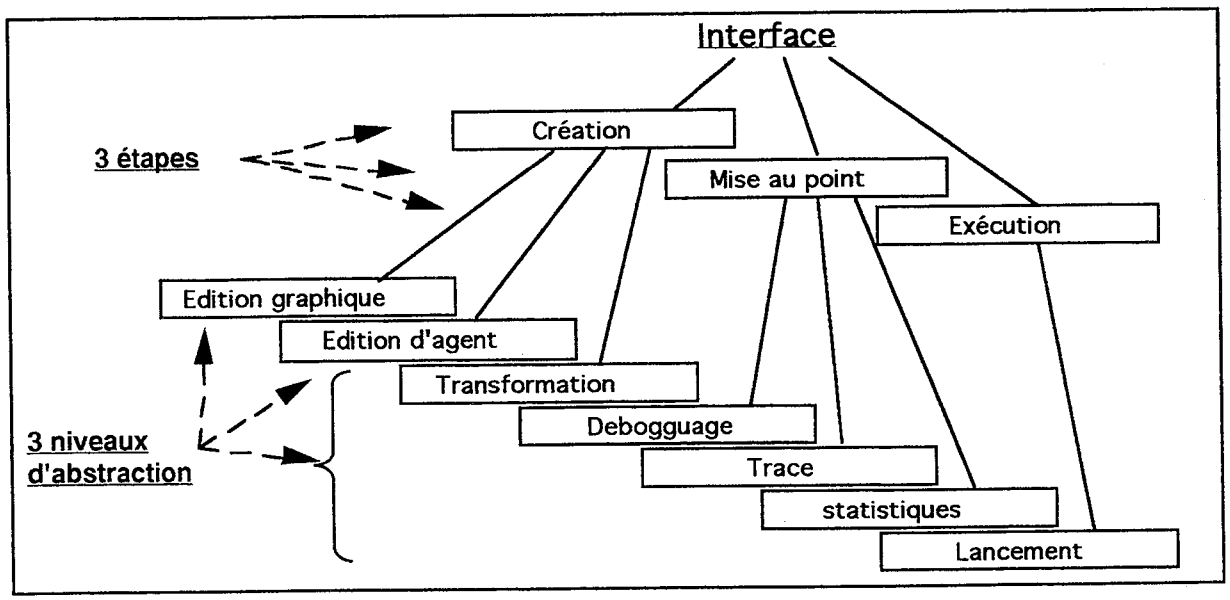

Figure 4.22 : Structuration de l'interface.

Ce schéma fait apparaître les différentes étapes d'utilisation que nous avons développées sous la forme des menus suivants figure 4.23 :

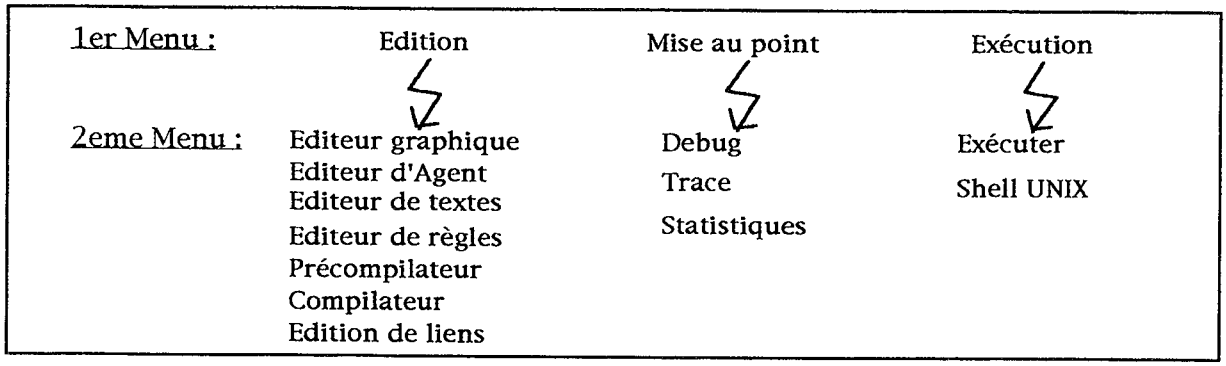

Figure 4.23 : Les menus de l'interface.

Nous allons maintenant présenter l'éditeur graphique d'application (EGA) qui constitue le premier niveau d'abstraction.

L'editeur graphique d'application

Le but de cet éditeur est de permettre à l'utilisateur d'implanter son réseau d'agents sur un ensemble de stations de travail reliées en réseau. Il doit permettre de concrétiser le désir d'un utilisateur en lui proposant différentes actions pour tracer graphiquement son réseau d'agents sur un treillis tout en effectuant un certain nombre de vérifications de cohérence.

On trouve deux sortes de traitements, suivant les commandes exécutées :
- les traitements relatifs aux commandes de l'utilisateur, ce sont les actions suivantes :
  -- Ajouter un agent (KS ou KP)
  -- Positionner un agent, dans le treillis logique,
  -- Positionner un agent sur une machine spécifique,
  -- Nommer (ou renommer) un agent,
  -- Supprimer un agent,
  -- Déplacer un agent,
  -- Saisir les paramètres agent
  -- Ajouter une connexion,
  -- Supprimer une connexion,
  -- Saisir les paramètres connexion,
  -- Annuler la dernière commande,
  -- Accéder en visualisation :
    --- aux contenu des agents,
    --- à tous les éléments saisis relatifs au réseau,
  -- Valider et mettre à jour le réseau,

-- Quitter / Sauver,

- et les traitements induits par l'action des commandes précédentes, essentiellement vérification de l'implantation réalisée, sur :

--- validité des connexions,

--- validité des noms (port, stations, agents),

--- unicité des noms,

On trouve également la visualisation de tout le réseau,

A l'issue du travail d'implantation on récupère les informations suivantes :

- Informations textuelles :

-- Nom du fichier application,

-- Nombre d'agents dans le réseau,

-- pour chaque agent :

--- Nom,

--- Type,

--- Coordonnées graphiques,

--- Numéro de port de communication,

--- Nom de la station où il est implanté,

--- Nombre de connexions,

--- pour chaque connexion :

---- Nom de l'agent avec lequel il est connecté,

---- Numéro de port de communication,

---- Nom de la station de l'agent avec lequel il est connecté,

- Informations graphiques :

- schéma complet du réseau,

Ces informations seront utilisées lors de précompilations notamment pour établir les connexions entre agents. De plus la fenêtre d'exécution réutilise le treillis d'implantation.

4.5 - Conclusion

L'interface n'est pas complètement finie, les inclusions des outils dans l'interface sont en partie à réaliser, mais la structure complète est faite.

L'utilisation de Motif est un choix intéressant, son orientation objet en fait un outil modulaire facilitant la *réutilisabilité*, la *portabilité* et l'*extensibilité*. Néanmoins sa programmation est laborieuse et l'apprentissage est long. L'efficacité d'un générateur d'interface est là pour démontrer que l'abstraction des couches Motif n'est encore pas suffisante pour obtenir une programmation très rapide. Malheureusement l'absence de cet outil a rendu la réalisation difficile, mais nous espérons que les avantages cités primeront.

Conclusion

Au cours de cette conclusion nous allons aborder successivement la place de l'approche "agent" par rapport à l'approche "objet", puis les apports du système COALA , ainsi que son état d'avancement, enfin les évolutions et perspectives plausibles à ce travail.

L'approche agent par rapport à l'approche objet

Le domaine que nous avons eu à traiter est celui du développement de logiciels de résolution de problème, et les moyens envisagés ont été ceux de l'approche Multi-Agents. Dans cette approche nous avons eu comme directives celles du Génie Logiciel.

Les systèmes Multi-Agents en tant que branche de l'Intelligence Artificielle Distribuée ont comme but de "distribuer" connaissances et raisonnements nécessaires à la résolution d'un problème au sein de plusieurs "agents", la vision du problème n'étant jamais globale.

Penser en termes d'agents permet de s'affranchir de l'effort permanent que doit effectuer un programmeur pour essayer de "coller" un modèle objet à un problème qui est d'un niveau d'abstraction différent. On peut voir cette approche comme une sur-couche à une approche objet, celle-ci étant le modèle sous-jacent du développement. L'approche objet a permis de structurer un logiciel autour des objets plutôt qu'autour des actions, l'approche agent permet de structurer un logiciel autour des agents en tant "qu'objets actifs intelligents".

Le génie Logiciel a pour but de fiabiliser la tâche de développement de logiciels, un des moyens pour atteindre ce niveau de qualité étant la modularité.

En Intelligence Artificielle Distribuée on cite souvent la modularité comme indispensable à la distribution.

Ainsi cette modularité rejoint la "distribution" des systèmes Multi-Agents, l'"agent" devenant l'élément de base.

Les apports du système COALA

COALA a été développé selon trois axes : une *méthodologie* incluse dans un *environnement* dont une étape importante est le *langage* d'agents. En effet les apports du langage proposé renforcent l'intérêt de l'approche "agent". Un but qui a également été

recherché, est que cet environnement puisse servir à des utilisateurs pour valider le concept du Multi-Agents utilisé. De plus, COALA permettra de faire évoluer le concept en fonction des dernières expérimentations réalisées et de les inclure dans un cadre de développement facile à utiliser.

Par rapport au système MAPS antérieur, les apports de COALA sont de trois sortes : apport conceptuel, apport de principe et apport fonctionnel.

Dans les apports conceptuels on trouve :

- l'extension des possibilités d'envoi de message d'un agent à lui-même,
- la manipulation de mondes.

Dans les apports de principe on trouve :

- la structuration importante dans la conception d'une application, liée à une modularisation des différents éléments constituant l'environnement, (modularité permettant une évolution du concept de base),
- la librairie de classes C++ permettant le développement de nouveaux concepts,
- l'ouverture vers le langage C++,
- un nouveau langage d'agents par homogénéisation et simplification du langage précédent,
- de fortes liaisons entre la méthodologie et l'utilisation des outils dans une interface graphique.

Dans les apports fonctionnels on trouve :

- le passage de règles interprétées à des règles compilées, donc une meilleure efficacité à l'exécution,
- des vérifications supplémentaires lors de la précompilation,
- l'étape C++,
- une portabilité accrue.

Une résolution de problème peut alors se résumer comme suit :

1) soumettre le problème : c'est la "philosophie" de COALA, penser en terme de réseau d'agents.

2) poser le problème : c'est appliquer la méthodologie de développement de COALA.

3) visualiser le problème : c'est utiliser les outils de COALA

Etat d'avancement du projet

Il reste à développer quelques outils d'aide à la mise au point (débogue, trace, statistiques), de même que des outils pressentis par les utilisateurs, tels que éditeur de règles et bien sûr éditeur d'agent.

Trois concepts ont été étudiés, mais il reste à les implanter, ce sont : les agents réactifs, la technique frame, un comportement associé à chaque message.

Du point de vue des vérifications, il reste à vérifier, notamment la portabilité sur divers matériels. En particulier l'interface graphique a été développée sur station SONY et doit être porté sur SUN et intégrée totalement au système COALA.

D'autre part le précompilateur nécessite encore des améliorations telles que des vérifications supplémentaires, la réorganisation des fichiers produits.

Evolution et perspectives

Dans un premier temps c'est le développement d'applications qu'il semble important de réaliser. Ceci permettrait de valider les choix effectués et de déterminer les futurs évolutions. Ces évolutions sont d'ordre conceptuelles (concept Multi-Agents) et techniques.

Evolution du concept Multi-Agents :

Un certain nombre d'améliorations structurelles, déjà présenties dans [BAUJARD 92], concernent deux directions importantes. Ce sont d'une part la présence d'agents de granularité variables et d'autre part une structuration hiérarchique des agents d'une application par la définition d'architectures multicouches.

Sans entrer dans des considérations théoriques importantes, nous pouvons dire que la présence d'agents réactifs est nécessaire et/ou suffisantes pour des analyses bas niveau, alors que des agents cognitifs évoluent dans des niveaux où des stratégies complexes sont nécessaires. cependant le positionnement d'un agent sur cette échelle réactif-cognitif n'est pas aisé. En effet il ne suffit pas de leur donner un comportement réactif ou cognitif, il faut aussi penser que les interactions entre agents sont différentes. Un agent réactif "réagit" de façon générale à son environnement, alors qu'un agent cognitif doit être capable de raisonner sur les connaissances qu'il possède sur les autres. Pour parvenir à cette réactivité il faut implanter la notion de type "frame" dans un agent KS. Un frame est un objet dont les attributs sont dotés de "facettes". Ces facettes sont par exemple des appels de procédure. Ces objets sont activés dès que l'on modifie les attributs, ce sont des activations de type "si-besoin", "si-ajout". Par exemple la lecture d'une valeur d'attribut peut déclencher une fonction de recherche d'une valeur par défaut si cet attribut n'a pas été initialisé. Un agent KS peut alors se voir ôter de toute possibilité de communication, il est purement réactif. De même un agent KP peut n'être qu'une

règle qui se déclenche dans certaines conditions sans qu'il y ait envoi de message de la part d'un agent KS, par exemple déclenchement d'une règle suite à un événement.

L'autre direction, "hiérarchie multicouches d'agents hétérogènes" est intéressante pour permettre d'acquérir des niveaux d'abstractions au sein d'une application. Le but est de disposer les agents en couches suivant le travail à effectuer. Les agents d'une même couche collaborent par l'intermédiaire de messages d'informations, et pourront diriger et organiser des agents d'une couche inférieure par l'intermédiaire de messages de contrôle. Cette approche peut être mixée avec l'approche de granularité variable précédente. En effet le travail bas niveau peut être effectué par des agents réactifs sur une couche basse, et sont dirigés par des agents cognitifs des couches supérieures. Cette approche multicouche a été envisagée par la création d'agents "méta". Suite à de nombreuses discussions est apparue la difficulté de formalisation de ces agents, résumée dans les questions suivantes :

- Qu'est-ce qu'un agent "méta-KS" ? "méta-KP" ?
- Que pilote un agent "méta-KS ? des agents KS d'une couche inférieure ? est-il en relation avec des agents KP d'une couche inférieure ?
- Mêmes questions pour un agent "méta-KP" ?
- Doit-on avoir des dialogues "méta" entre agent KS et KP, KS et KS, KP et KP ?

Ce sont en fait des questions sur l'évolution du concept Multi-Agents de COALA.

S'il ne m'appartient pas de décider sur l'orientation de ces évolutions, je pense que la solution réside dans la programmation de diverses applications mettant en oeuvre ces principes. En effet le langage d'agents introduit permet d'établir des dialogues non exclusifs entre agents KS et KP et l'ouverture vers le langage C++ permet la redéfinition de comportements et de méthodes. L'implantation d'agents "réactifs" est possible. La structure de base d'un frame est déjà introduite dans la librairie de classes, son développement demande seulement l'écriture des algorithmes adéquats. De plus l'existence d'un indicateur d'état réactif-cognitif au sein d'un agent KS permettra de jouer sur la granularité en cours d'exécution, laissant à un agent KP l'opportunité du passage réactif-cognitif.

Pour faciliter le développement de ces évolutions, on peut utiliser le schéma suivant :

- le développement d'application en standard avec le langage d'agents,
- le travail sur fichier C++,
- la prise en compte des évolutions pour la librairie,
- l'évolution possible du langage.

Evolution technique :

Parmi les évolutions techniques, l'amélioration des communications doit être envisagée. Actuellement tous les dialogues entre agents s'effectuent selon le mécanisme UNIX des "sockets". Ceci peut être pénalisant dans le cas d'échanges très fréquents. En effet tel que cela a été programmé chaque échange nécessite l'établissement de la communication, avec acceptation de la part de l'agent receveur, puis le dialogue et enfin rupture de la communication. Ce principe

est indispensable lorsque le dialogue s'effectue entre deux agents implantés sur deux machines différentes, mais est inutilement lourd lorsque les agents sont deux processus de la même machine. Dans ce cas on peut envisager une communication par "pipe", pour accélérer les communications.

Avec ce principe on peut proposer une évolution plus générale des possibilités de communication en étendant les capacités des agents KS. Par exemple, un agent KS pourrait propager les informations qui lui sont communiquées, puis les proposer en interne, mais aussi à d'autres agents KS. Ceci permettrait d'éviter l'apparition d'agents KP qui ne servent qu'à transmettre des informations entre KS, ce qui arrive "naturellement" dans la description d'une application, on associe fréquemment un KS et un KP.

Un autre problème pour le traitement de messages par le mécanisme des sockets, vient de la limitation de la file d'attente des messages reçus par un agent. Cette limitation est de 5 messages. Une solution est de créer un processus fils pour le traitement de chaque message. Mais le problème réside dans la duplication des données (principe du "fork" UNIX), qui peut entraîner un traitement sur des données erronées, celles-ci ayant évolué entre l'instant de la réception du message et l'instant où elles peuvent être traitées. Une solution peut être envisagée avec la future version du système d'exploitation SOLARIS II sur les nouvelles machines SUN (SPARC 10). Cette version introduit notamment le "multi-threading", c'est-à-dire la création de processus sur des processeurs virtuels, qui sont concurrents et partagent les données, on utilise un nouveau "fork" qui ne duplique pas systématiquement les données du processus père.

Parmi d'autre évolutions, entre concept et technique on peut citer :

- pousser le concept de classe générique jusqu'à obtenir des agents "type". Une évolution possible est alors, si l'expérimentation le confirme, de créer des agents avec un rôle précis qui permettent de ne plus se préoccuper de leurs contenus (agent observateur par exemple) . Dans une première étape on peut développer une base d'agents réutilisables d'une application à une autre.
- établir des classes génériques "protocole". Actuellement les protocoles sont codés dans les règles, donc programmés par les utilisateurs, la présence de classes génériques permettra de s'affranchir de ce travail.

Si , comme nous l'espérons, le développement d'applications valide les choix sur le langage d'agent et l'implantation de la librairie de classes, la conception de logiciels de résolution de problème par coopération d'agents aura fait un pas vers une utilisation plus pragmatique de ce concept.

ANNEXE A : Langage d'agents.

Types standards

Dans COALA les variables manipulées sont de six types possibles, 3 types de base et 3 types de liste.
Les types de base sont Entier, Réel, Chaîne de caractères.
Les listes sont Liste d'entiers, Liste de réels et Liste de chaînes de caractères.

Mot clés :

entier	:	**INT**
réel	:	**FLOAT**
chaîne de caractères	:	**STRING**
Liste d'entiers	:	**INTLIST**
Liste de réels	:	**FLOATLIST**
Liste de chaînes de caractères	:	**STRINGLIST**

Exemples :

entier	:	123 9876
réel	:	12.34 0.567
chaîne de caractères	:	coala Voila une chaîne
Liste d'entiers	:	(123 456 78)
Liste de réels	:	(12.34 67.3 0.45)
Liste de chaînes de caractères	:	(bonjour coala)

Opérations applicables :

entier	:	addition +, soustraction (binaire ou unaire) -, multiplication *, division entière / et modulo %
réel	:	addition +, soustraction (binaire ou unaire) -, multiplication * et division /
chaîne de caractères	:	addition ou concaténation +,
Liste	:	CAR, CDR, CONS, APPEND, ELEMENT, NELEMENT, NULL, (la sémantique est celle des opérateurs de LISP)

Définition d'un agent

syntaxe :

```
AGENT NomAgent <type >{
COMMENT "chaîne de caractères";
... (contenu)
};
```

NomAgent : Chaîne de caractères identifiant l'agent.

type = KS ou KP

Mot clés : **AGENT, KS, KP, COMMENT.**

Commentaires :

Comment est optionnel.

Contenu d'un agent KS : - liste d'objets,
- initialisations éventuelles de mondes,
- initialisations éventuelles d'instances,
- liste d'accointances,
- liste de règles,
- redéfinitions éventuelles de comportements,
- liste de procédures.

Contenu d'un agent KP : - liste d'accointances,
- liste de règles,
- redéfinitions éventuelles de comportements
- liste de procédures.

Définition d'un objet

syntaxe :

```
OBJECT NomObjet {
COMMENT "chaîne de caractères";
... (contenu)
};
```

NomObjet : Chaîne de caractères identifiant l'objet.

Mot clé : **OBJECT, COMMENT.**

Commentaires :

Comment est optionnel.

Contenu d'un objet : - liste d'attributs,
- initialisation de mondes,
- initialisation d'instances.

(initialisation de mondes et d'instances sont optionnelles)

Initialisation d'un monde

syntaxe :

```
WORLD NomObjet {

... (contenu)
};
```

NomObjet : Chaîne de caractères identifiant l'objet.

Mot clé : **WORLD.**

Commentaires :

'Nomobjet' est le nom d'un objet précédemment défini.

le contenu est une intialisation d'instances.

Initialisation d'une instance

syntaxe :

```
INSTANCE NomObjet {

        NomAttribut1 = ValeurAttribut1;
        NomAttribut2 = ValeurAttribut2;
        ...
};
```

NomObjet : Chaîne de caractères identifiant l'objet.

Mot clé : **INSTANCE.**

Commentaires :

- Une initialisation d'instance est une suite d'initialisation d'un nombre quelconque d'attributs de l'objet.

- L'initialisation d'un attribut est une affectation de valeur de l'attribut nommé par une valeur du type de l'attribut.

- Une initialisation d'instance se fait soit dans une initialisation de monde soit dans le monde courant.

- Le nom de l'objet 'NomObjet' est optionnel si l'initialisation se fait dans une initialisation de mondes.

Définition d'un attribut

syntaxe :

ATTRIBUTE NomAttribut <*type* > (*Valeur*) [*Limite basse* ..*Limite haute*]
{ COMMENT "chaîne de caractères"; } ;

NomAttribut : Chaîne de caractères identifiant l'attribut.

type = INT, FLOAT, STRING, LISTINT, LISTFLOAT, LISTSTRING. (entier, réel, chaîne de caractères, liste d'entiers, liste de réels, liste de chaîne de caractères).

Mot clés : **ATTRIBUTE, COMMENT,**
INT, FLOAT, STRING, LISTINT, LISTFLOAT, LISTSTRING.

Commentaires :

Valeur, Limite basse et haute, et comment sont optionnels.

Valeur = Valeur par défaut,
Limite basse et haute = Gamme de Valeur ou doit se trouver l'attribut,

Définition d'une accointance

syntaxe :

```
ACQUAINTANCE NomConnection1,...;
```

NomConnection : Chaîne de caractères identifiant les connections.

Mot clé : **ACQUAINTANCE**

Commentaires :

Contenu d'une accointance : - liste de noms d'agents.

Redéfinition d'un comportement

syntaxe :

```
BEHAVIOR NomComportement {
...(contenu)
} ;
```

NomComportement: : Chaîne de caractères identifiant le comportement.

type de comportement = RECEIVE, SUPPLY, PROCESS, SOLVE.

Mot clé : **BEHAVIOR, RECEIVE, SUPPLY, PROCESS, SOLVE.**

Commentaires :

Pour un Agent KS, les comportements sont : RECEIVE et SUPPLY,

et pour un Agent KP, les comportements sont : PROCESS et SOLVE.

Le contenu est écrit en C++

Définition de procédures internes

syntaxe :

```
DEFINE PROCEDURE {
        Procedure1(parameter1,parameter2,...) {
                ...contenu1
        }
        Procedure2(parameter...) {
                ...contenu2
        }
        ...
} ;
```

Mot clé : **DEFINE PROCEDURE.**

Commentaires :

Cette instruction permet de définir des procédures écrites en C++ et pouvant être appelées depuis une règle (prémisse ou conclusion) de l'agent.

Définition d'une règle

syntaxe :

```
RULE NomRègle <type > {
COMMENT "chaîne de caractères"
IF      < Prémisse >
THEN < Conclusion >
} ;
```

NomRègle : Chaîne de caractères identifiant la règle.

type = PROPAGATION, PROPOSITION, INTERROGATION, ACTION, STRATEGY

Mot clés : **RULE, PROPAGATION, PROPOSITION, INTERROGATION, ACTION, STRATEGY, IF, THEN, COMMENT.**

Commentaires :

Comment est optionnel.

Pour un Agent KS, les types sont : PROPAGATION, PROPOSITION et INTERROGATION, et pour un Agent KP, les types sont : ACTION et STRATEGY.

Définition de Prèmisse et Conclusion

Généralités

Une prémisse ou une conclusion de règle est constituée d'une instruction ou d'un ensemble d'instructions (expressions) reliées par les opérateurs logiques && (ET) et || (OU). Il est également possible d'utiliser la négation ! (NON) en prémisse d'une règle. Il est enfin possible de définir une priorité avec des parenthèses.

Les différentes instructions possibles sont :

- Définition de paramètre,
- Affectation de paramètre,
- Accès aux méthodes,
- Appel de procédures internes ou externes,
- Méthode de recherche,
- Envois de messages,

Ces instructions peuvent être utilisées a l'intérieur de boucles et d'expressions booléennes,

On peut regrouper des instructions dans un bloc.

(Pour la syntaxe exacte se reporter à la grammaire).

Dans les paragraphes suivants sont détaillées les différentes possibilités des expressions énoncées ci-dessus.

Opérateurs logiques

Ce sont && (ET) , || (OU) , ! (NON)

Exemples :
```
A && B
A || B
!A
!(A || ((B && C) || D))
```
où A, B, C et D sont des instructions.

Blocs

Un bloc regroupe des instructions, séparées par des points-virgules (;).

Exemple :
```
{
InstA1;
InstA2;
...
}
```

Paramètre

Un paramètre est défini comme une variable d'un des types standard, entier, réel, chaîne de caractères ou liste d'un de ces types, qui peut éventuellement être initialisée . Il est également possible d'initialiser un paramètre avec la valeur retournée par les instructions GET, par les différentes méthodes de manipulation (GET INSTANCE OF, MAKE INSTANCE OF, ...) et avec la valeur d'un attribut déjà défini.

exemples :
```
STRING NomC;
FLOATLIST NomFL = (1.9 0.67);
INT NomI = 123;
STRING NomCH = GET ( Agent, Objet, Attribut );
```

Il est également possible d'initialiser des paramètres de type INT avec l'index représentant de manière interne un agent, un objet ou un attribut. Ce mécanisme est particulièrement utile pour la paramétrisation des instructions manipulant des noms d'agents, d'objets et d'attributs, comme GET ou SET.
exemple : INT NomI = NomAgent;

Opérations sur les listes

les opérations sur les listes se font à partir d'opérateurs inspirés de LISP. Ces opérateurs peuvent être appliqués sur des valeurs liste ou sur des paramètres de type liste.

Constructeur et affectation

exemples : INTLIST LI1 ;
FLOATLIST LF1 = NomListe;
STRINGLIST LC1 = ("coala" "liste initialisée" "abc");
le premier cas crée une liste d'entiers vide,
le deuxième cas crée une liste de réels initialisés avec le contenu d'une autre liste,
le troisième cas crée une liste de chaînes de caractères initialisée avec les 3 chaînes citées.

Status

exemple : NULL(Liste);
teste si une liste 'Liste' est vide, rend la valeur vrai si oui, faux sinon.

Tête et queues de liste

exemples : Element = CAR (Liste);
Liste1 = CDR (Liste2);
le premier cas affecte à 'Element' la valeur du premier élément de la liste 'Liste',
le deuxième cas affecte à la liste 'Liste1' le contenu de la queue de la liste 'Liste2', c'est-à-dire la sous liste constituée de la liste 'Liste2' moins le premier élément.

Concaténation

exemples : APPEND (Liste1 , Element);
APPEND (Liste1 , Liste2);
APPEND (Liste1, Liste2, Liste3);
le premier cas ajoute l'élément 'Element' en queue de la liste 'Liste1',
le deuxième cas concatène la liste 'Liste2' à la liste 'Liste1', le résultat est dans la liste 'Liste1'.
le troisième cas concatène la liste 'Liste3' à la liste 'Liste2' et met le résultat dans la liste 'Liste1'.

Opérations arithmétiques

Les opérations arithmétiques sont +,-,*,/,% et le -unaire pour des entiers, +,-,*,/ et le -unaire pour les réels. Une opération arithmétique est encadrée par des parenthèses.

Expressions booléennes

Une expression booléenne est constituée par la comparaison de deux opérandes. Les opérateurs de comparaison sont == , != , < , <= , > , >=, ELEMENT. Une opération booléenne est encadrée par des parenthèses.

Boucles

Il est possible de définir dans une règle des boucles sur un ensemble d'instructions. Les instructions de boucle sont FOR, WHILE et DO

boucle WHILE et DO :
la syntaxe est la suivante, WHILE Expression DO Expression
DO Expression WHILE Expression

boucle FOR :
plusieurs syntaxes sont possibles suivant que l'itération se fait sur une liste ou sur un entier.
FOR (Element IN Liste) DO Expression
la boucle est effectuée pour chaque élément de la Liste, cet élément étant affecté à Element.
FOR (entier , Condition) DO Expression
la boucle est effectué sur l'entier si la condition est réalisé, l'incrément de boucle est 1.

Remarques :
A l'intérieur d'une règle toute référence à un agent, objet, attribut, paramètre... se fera de deux façon :
- *soit par le nom qu'on lui a donné,*
- *soit par un paramètre que l'on aura initialisé avec la valeur de l'index de cet élément. Cet index doit être de type entier.*

Dans la suite de ce paragraphe il faut noter que toute référence à un objet dans une règle quelconque se fera sous la forme référence Agent-Objet.
Pour un agent KS on aura (Agent , Objet) exprimé de la façon suivante :
(Self , O) pour un accès interne a l'objet O,
(A , O) pour un envoi de l'objet O vers un agent KS de nom A.
Pour un agent KP l'expression (Agent , Objet) sera toujours exprimé :
(A , O) pour un accès a l'objet O d'un agent KS.

Accès aux objets

GET (Ag,Ob,Att)
ou
GET (Ag,Ob,Att) **FROM** (Ins)
ou
GET (Ag,Ob,Att) **FROM** (Ins) **IN** (Mond)

Cette instruction retourne la valeur l'attribut Att de l'objet Ob de l'agent Ag
de l'instance courante du monde courant dans le premier cas,
de l'instance Ins du monde courant dans le deuxième cas,
de l'instance Ins du monde Mond dans le troisième cas.

SET (Ag,Ob,Att) **WITH** (Val)
ou
SET (Ag,Ob,Att) **FROM** (Ins) **WITH** (Val)
ou
SET (Ag,Ob,Att) **FROM** (Ins) **IN** (Mond) **WITH** (Val)

Cette instruction affecte la valeur Val à l'attribut Att de l'objet Ob de l'agent Ag
de l'instance courante du monde courant dans le premier cas,
de l'instance Ins du monde courant dans le deuxième cas,
de l'instance Ins du monde Mond dans le troisième cas.
Val peut être une valeur d'un des types de base, un littéral ou un nom.

Méthode de manipulation de monde

Création :

MAKE WORLD OF (Ag,Ob)
ou
MAKE WORLD OF (Ag,Ob) **FROM** (Mond)

cette instruction crée un monde
dans l'objet Ob de l'agent Ag et retourne la valeur de l'index du monde créé dans le premier cas,
a partir du monde Mond dans l'objet Ob de l'agent Ag et retourne la valeur de l'index du monde crée dans le deuxième cas,

destruction :

DELETE CURRENT WORLD OF (Ag,Ob)
ou
DELETE WORLD OF (Ag,Ob,Mond)
ou
DELETE ALL WORLD OF (Ag,Ob)
ou
DELETE CURRENT WORLD OF (Ag,ListeOb)
ou
DELETE WORLD OF (Ag,ListeOb,Mond)
ou
DELETE ALL WORLD OF (Ag,ListeOb)

cette instruction détruit
le monde courant de l'objet Ob de l'agent Ag dans le premier cas,
le monde Mond de l'objet Ob de l'agent Ag dans le deuxième cas,
tous les mondes de l'objet Ob de l'agent Ag dans le troisième cas,
le monde courant de la liste d'objet ListeOb de l'agent Ag dans le quatrième cas,
le monde Mond de la liste d'objet ListeOb de l'agent Ag dans le cinquième cas,
tous les mondes de la liste d'objet ListeOb de l'agent Ag dans le sixième cas.

lecture :

GET FIRST WORLD NUMBER OF (Ag,Ob)
ou
GET NEXT WORLD NUMBER OF (Ag,Ob)
ou
GET CURRENT WORLD NUMBER OF (Ag,Ob)

cette instruction retourne
le premier monde de l'objet Ob de l'agent Ag dans le premier cas,
le monde suivant de l'objet Ob de l'agent Ag dans le deuxième cas,
le monde courant de l'objet Ob de l'agent Ag dans le troisième cas.

mise à jour :

```
SET CURRENT WORLD OF ( Ag,Ob ) WITH ( Mond )
```

cette instruction positionne le monde courant de l'objet Ob de l'agent Ag avec Mond.

Méthodes de gestion des instances d'objets

Création :

```
MAKE INSTANCE OF ( Ag,Ob )
ou
MAKE INSTANCE OF ( Ag,Ob ) IN ( Mond )
ou
MAKE INSTANCE OF ( Ag,Ob ) FROM ( Ins ) IN ( Mond )
```

Cette instruction crée une instance,
du monde courant de l'objet Ob de l'agent Ag dans le premier cas,
du monde Mond de l'objet Ob de l'agent Ag dans le deuxième cas,
du monde Mond de l'objet Ob de l'agent Ag à partir de l'instance Ins dans le troisième cas,
et retourne le numéro de l'instance créée.

lecture :

```
GET FIRST INSTANCE NUMBER OF ( Ag,Ob )
ou
GET FIRST INSTANCE NUMBER OF ( Ag,Ob ) IN ( Mond )
ou
GET NEXT INSTANCE NUMBER OF ( Ag,Ob )
ou
GET NEXT INSTANCE NUMBER OF ( Ag,Ob ) IN ( Mond )
ou
GET CURRENT INSTANCE NUMBER OF ( Ag,Ob )
ou
GET CURRENT INSTANCE NUMBER OF ( Ag,Ob ) IN ( Mond )
```

cette instruction retourne
la première instance du monde courant de l'objet Ob de l'agent Ag dans le premier cas,
la première instance du monde Mond de l'objet Ob de l'agent Ag dans le deuxième cas,
l'instance suivante du monde courant de l'objet Ob de l'agent Ag dans le troisième cas,
l'instance suivante du monde Mond de l'objet Ob de l'agent Ag dans le quatrième cas,
l'instance courante du monde courant de l'objet Ob de l'agent Ag dans le cinquième cas,
l'instance courante du monde Mond de l'objet Ob de l'agent Ag dans le cinquième cas.

Mise à jour :

SET CURRENT INSTANCE OF (Ag,Ob) **WITH** (Val)
ou
SET CURRENT INSTANCE OF (Ag,Ob) **IN** (Mond) **WITH** (Val)

Cette instruction met à jour avec la valeur Val l'instance courante,
du monde courant de l'objet Ob de l'agent Ag dans le premier cas,
du monde Mond de l'objet Ob de l'agent Ag dans le deuxième cas.
Val peut être un entier ou un paramètre.

destruction :

DELETE CURRENT INSTANCE OF (Ag,Ob)
ou
DELETE CURRENT INSTANCE OF (Ag,Ob) **IN** (Mond)
ou
DELETE INSTANCE OF (Ag,Ob,Ins)
ou
DELETE INSTANCE OF (Ag,Ob,Ins) **IN** (Mond)
ou
DELETE ALL INSTANCE OF (Ag,Ob)
ou
DELETE ALL INSTANCE OF (Ag,Ob) **IN** (Mond)
ou
DELETE CURRENT INSTANCE OF (Ag,ListeOb)
ou
DELETE CURRENT INSTANCE OF (Ag,ListeOb) **IN** (Mond)
ou
DELETE INSTANCE OF (Ag,ListeOb,Ins)
ou
DELETE INSTANCE OF (Ag,ListeOb,Ins) **IN** (Mond)
ou
DELETE ALL INSTANCE OF (Ag,ListeOb)
ou
DELETE ALL INSTANCE OF (Ag,ListeOb) **IN** (Mond)

Cette instruction détruit,
l'instance courante du monde courant de l'objet Ob de l'agent Ag dans le premier cas,
l'instance courante du monde Mond de l'objet Ob de l'agent Ag dans le deuxième cas,
l'instance Ins du monde courant de l'objet Ob de l'agent Ag dans le troisième cas,
l'instance Ins du monde Mond de l'objet Ob de l'agent Ag dans le quatrième cas,
toutes les instances du monde courant de l'objet Ob de l'agent Ag dans le cinquième cas,
toutes les instances du monde Mond de l'objet Ob de l'agent Ag dans le sixième cas,
l'instance courante du monde courant de la liste d'objet ListeOb de l'agent Ag dans le septième cas,
l'instance courante du monde Mond de la liste d'objet ListeOb de l'agent Ag dans le huitième cas,
l'instance Ins du monde courant de la liste d'objet ListeOb de l'agent Ag dans le neuvième cas,
l'instance Ins du monde Mond de la liste d'objet ListeOb de l'agent Ag dans le dixième cas,
toutes les instances du monde courant de la liste d'objet ListeOb de l'agent Ag dans le onzième cas,
toutes les instances du monde Mond de la liste d'objet ListeOb de l'agent Ag dans le dernier cas.

Méthodes de recherche des instances d'objets

GET INSTANCE OF (Ag,Ob) **WHERE** { cond }
ou
GET INSTANCE OF (Ag,Ob) **IN** (Mond) **WHERE** { cond }

Cette instruction effectue une recherche de l'instance du monde
courant de l'objet Ob de l'agent Ag et qui vérifie la condition cond dans le premier cas,
Wor de l'objet Ob de l'agent Ag et qui vérifie la condition cond dans le deuxième cas.

GET INSTANCE OF (Ag,Ob) **WITH LOWER** (Att)
ou
GET INSTANCE OF (Ag,Ob) **IN** (Mond) **WITH LOWER** (Att)
ou
GET INSTANCE OF (Ag,Ob) **WITH LOWER** (Att) **WHERE** { cond }
ou
GET INSTANCE OF (Ag,Ob) **IN** (Mond) **WITH LOWER** (Att)**WHERE** { cond }

Cette instruction effectue une recherche de l'instance du monde
courant de l'objet Ob de l'agent Ag dont l'attribut Att a la plus petite valeur dans le premier cas,
Wor de l'objet Ob de l'agent Ag dont l'attribut Att a la plus petite valeur dans le deuxième cas.
courant de l'objet Ob de l'agent Ag dont l'attribut Att a la plus petite valeur et qui vérifie la condition cond dans le troisième cas,
Wor de l'objet Ob de l'agent Ag dont l'attribut Att a la plus petite valeur et qui vérifie la condition cond dans le quatrième cas.

GET INSTANCE OF (Ag,Ob) **WITH UPPER** (Att)
ou
GET INSTANCE OF (Ag,Ob) **IN** (Mond) **WITH UPPER** (Att)
ou
GET INSTANCE OF (Ag,Ob) **WITH UPPER** (Att) **WHERE** { cond }
ou
GET INSTANCE OF (Ag,Ob) **IN** (Mond) **WITH UPPER** (Att)**WHERE** { cond }

Cette instruction effectue une recherche de l'instance du monde
courant de l'objet Ob de l'agent Ag dont l'attribut Att a la plus grande valeur dans le premier cas,
Wor de l'objet Ob de l'agent Ag dont l'attribut Att a la plus grande valeur dans le deuxième cas,
courant de l'objet Ob de l'agent Ag dont l'attribut Att a la plus grande valeur et qui vérifie la condition cond dans le troisième cas,
Wor de l'objet Ob de l'agent Ag dont l'attribut Att a la plus grande valeur et qui vérifie la condition cond dans le quatrième cas.

Méthodes d'accès aux attributs système

```
GET PROPOSITION OF ( Ag,Ob )
GET INTERROGATION OF ( Ag,Ob )
GET RESULT OF ( Ag,Ob )
ou
GET PROPOSITION OF ( Ag,Ob ) FROM ( Ins )
GET INTERROGATION OF ( Ag,Ob ) FROM ( Ins )
GET RESULT OF ( Ag,Ob ) FROM ( Ins )
ou
GET PROPOSITION OF ( Ag,Ob ) FROM ( Ins ) IN ( Mond )
GET INTERROGATION OF ( Ag,Ob ) FROM ( Ins ) IN ( Mond )
GET RESULT OF ( Ag,Ob ) FROM ( Ins ) IN ( Mond )
```

Ces instructions effectuent une lecture des attributs système de l'instance courante du monde courant de l'objet Ob de l'agent Ag dans le premier cas, Ins du monde courant de l'objet Ob de l'agent Ag dans le deuxième cas, Ins du monde Wor de l'objet Ob de l'agent Ag dans le troisième cas.

Communication KS à KP

```
SEND ( Ob ) TO ( Ag )
ou
SEND ( Ob,Att ) TO ( Ag )
```

Cette instruction effectue une demande d'un agent KS de traiter une information Ob à un agent KP Ag dans le premier cas, une information Ob,Att à un agent KP Ag dans le deuxième cas.

```
ASK ( Ob ) TO ( Ag )
ou
ASK ( Ob,Att ) TO ( Ag )
```

Cette instruction effectue une demande d'un agent KS de résoudre un problème Ob à un agent KP Ag dans le premier cas, un problème Ob,Att à un agent KP Ag dans le deuxième cas.

Appel de procédure

Deux cas se présentent. Celui qui implique l'appel de procédure externes, réutilisation possible de procédures existantes, cas de bibliothèques spécialisées par exemple. C'est un mécanisme de RPC (Remote Procedure Call ou Appel de procédure à distance) qui est utilisé. Le deuxième cas est celui de procédures internes à un agent , écrites dans le corps de l'agent mais en C++. Ce sont des procédures qui ne peuvent être traduites dans des règles.

1er Cas :

```
NomProcédure ( Paramètre1, Paramètre2, ... )
```

L'appel s'effectue de façon simple par le nom de la procédure et une liste de paramètre conforme à la définition des paramètres formels de la procédure. Ceci suppose l'existence d'un serveur de procédures.

2eme Cas :

```
Appel :
NomProcédure1 ( Paramètre1, Paramètre2, ... )

Dèclaration :
DEFINE PROCEDURE {
        NomProcedure1(a,b,...) {
        ... Corps de la procédure
        }
        NomProcedure2(...) {
        ...
        }
...
} ;
```

ANNEXE B : Grammaire du langage d'agents.

Remarque : la grammaire suivante utilise une forme Backus Nohr Form (BNF) modifiée, car le langage utilise des caractères (exemple : { , °, |) qui servent à la description BNF pour modéliser les répétitions, conditions, etc... .

< ... > ::= marque le début d'une règle,
| indique une alternative,
les mots-clés sont soulignés,
les caractères ayant une importance dans la syntaxe sont en gras ce sont : '&&' , 'II', '!' , '{' , '}' , ';' , '(' , ')' , ' =', ',' .

<Expression> ::= <Expression> **&&** <Expression>
| <Expression> II <Expression>
| ! <Expression>
| { <Block> }
|<Instruction>
| (<Expression>)

<Block> ::= <Instruction> ;
| <Block> <Instruction> ;

<Instruction> ::= <Define-Parameter>
| <Affect-Parameter>
| <Method>
| <Loop>
| <Boolean-Expression>
| <Procedure>
| <Set>
| <Send>
| <Ask>

<Define-Parameter> ::= <Type> <Name>
| INT <Name> = <Integer>
| FLOAT <Name> = <Float>
| STRING <Name> = <String>
| INTLIST <Name> = <Integer-List>
| FLOATLIST <Name> = <Real-List>
| STRINGLIST <Name> = <String-List>
| INT <Name> = CAR (<Integer-List>)
| INT <Name> = CAR (<Name2>)
| FLOAT <Name> = CAR (<Real-List>)
| FLOAT <Name> = CAR (<Name2>)
| STRING <Name> = CAR (<String-List>)
| STRING <Name> = CAR (<Name2>)
| INTLIST <Name> = CDR (<Integer-List>)
| INTLIST <Name> = CDR (<Name2>)
| FLOATLIST <Name> = CDR (<Real-List>)
| FLOATLIST <Name> = CDR (<Name2>)
| STRINGLIST <Name> = CDR (<String-List>)
| STRINGLIST <Name> = CDR (<Name2>)
| <Type> <Name> = <Get>
| <Type> <Name> = <Name2>
| INT <Name> = <RefAgent>
| INT <Name> = <RefObject>
| INT <Name> = <RefAttribute>
| INT <Name> = <RefInstance>
| INT <Name> = <RefWorld>
| INT <Name> = <Search>
| INT <Name> = <Create>

| INT <Name> = <Get-Number>

<Type> ::= INT | FLOAT | STRING | INTLIST | FLOATLIST | STRINGLIST

<Affec-Parameter> ::= <Name> = <Litteral>
| <Name> = <List>
| <Name> = <Name2>
| <Name> = <Arithmetical-Expression>
| <Name> = CAR (<Integer-List>)
| <Name> = CAR (<Real-List>)
| <Name> = CAR (<String-List>)
| <Name> = CAR (<Name2>)
| <Name> = CDR (<Integer-List>)
| <Name> = CDR (<Real-List>)
| <Name> = CDR (<String-List>)
| <Name> = CDR (<Name2>)
| <Append-List>
| <Name> = <RefAgent>
| <Name> = <RefObject>
| <Name> = <RefAttribute>
| <Name> = <RefInstance>
| <Name> = <RefWorld>
| <Name> = <Get>
| <Name> = <Search>
| <Name> = <Create>
| <Name> = <Get-Number>

<Append-List> ::= APPEND (<Name> , <Litteral>)
| APPEND (<Name> , <Name2>)
| APPEND (<Name> , <Integer-List>, <Integer-List>)
| APPEND (<Name> , <Name2> , <Integer-List>)
| APPEND (<Name> , <Integer-List> , <Name2>)
| APPEND (<Name> , <Real-List>, <Real-List>)
| APPEND (<Name> , <Name2> , <Real-List>)
| APPEND (<Name> , <Real-List> , <Name2>)
| APPEND (<Name> , <String-List>, <String-List>)
| APPEND (<Name> , <Name2> , <String-List>)
| APPEND (<Name> , <String-List> , <Name2>)
| APPEND (<Name> , <Name2> , <Name3>)

<Get> ::= GET (<RefAgent> , <RefObject> , <RefAttribute>)
| GET (<RefAgent> , <RefObject> , <RefAttribute>) FROM (<RefInstance>)
| GET (<RefAgent> , <RefObject> , <RefAttribute>) FROM (<RefInstance>) IN (<RefWorld>)

<Set> ::= SET (<RefAgent> , <RefObject> , <RefAttribute>) WITH <Value>
| SET (<RefAgent> , <RefObject> , <RefAttribute>) FROM (<RefInstance>) WITH <Value>
| SET (<RefAgent> , <RefObject> , <RefAttribute>) FROM (<RefInstance>) IN (<RefWorld>) WITH <Value>

<Send> ::= SEND (<RefObject>) TO <RefAgent>
| SEND (<RefObject> , <RefAttribute>) TO <RefAgent>

<Ask> ::= ASK (<RefObject>) TO <RefAgent>
| ASK (<RefObject> , <RefAttribute>) TO <RefAgent>

<Search> ::= GET INSTANCE OF (<RefAgent> , <RefObject>) WHERE <Expression>
| GET INSTANCE OF (<RefAgent> , <RefObject>) IN (<RefWorld>) WHERE <Expression>
| GET INSTANCE OF (<RefAgent> , <RefObject>) WITH LOWER (<RefAttribute>)
| GET INSTANCE OF (<RefAgent> , <RefObject>) IN (<RefWorld>) WITH LOWER (<RefAttribute>)
| GET INSTANCE OF (<RefAgent> , <RefObject>) WITH LOWER (<RefAttribute>) WHERE <Expression>
| GET INSTANCE OF (<RefAgent> , <RefObject>) IN (<RefWorld>) WITH LOWER (<RefAttribute>) WHERE <Expression>
| GET INSTANCE OF (<RefAgent> , <RefObject>) WITH UPPER (<RefAttribute>)
| GET INSTANCE OF (<RefAgent> , <RefObject>) IN (<RefWorld>) WITH UPPER (<RefAttribute>)

| GET INSTANCE OF (<RefAgent> , <RefObject>) WITH UPPER (<RefAttribute>) WHERE <Expression>
| GET INSTANCE OF (<RefAgent> , <RefObject>) IN (<RefWorld>) WITH UPPER (<RefAttribute>)
WHERE <Expression>

<Get-number>::= GET FIRST INSTANCE NUMBER OF (<RefAgent> , <RefObject>)
| GET NEXT INSTANCE NUMBER OF (<RefAgent> , <RefObject>)
| GET FIRST INSTANCE NUMBER OF (<RefAgent> , <RefObject>) IN (<RefWorld>)
| GET NEXT INSTANCE NUMBER OF (<RefAgent> , <RefObject>) IN (<RefWorld>)
| GET CURRENT INSTANCE NUMBER OF (<RefAgent> , <RefObject>)
| GET CURRENT INSTANCE NUMBER OF (<RefAgent> , <RefObject>) IN (<RefWorld>)
| GET FIRST WORLD NUMBER OF (<RefAgent> , <RefObject>)
| GET NEXT WORLD NUMBER OF (<RefAgent> , <RefObject>)
| GET CURRENT WORLD NUMBER OF (<RefAgent> , <RefObject>)

<Create> ::= | MAKE INSTANCE OF (<RefAgent> , <RefObject>)
| MAKE INSTANCE OF (<RefAgent> , <RefObject>) IN (<RefWorld>)
| MAKE INSTANCE OF (<RefAgent> , <RefObject>) FROM (<RefInstance>) IN (<RefWorld>)
| MAKE WORLD OF (<RefAgent> , <RefObject>)
| MAKE WORLD OF (<RefAgent> , <RefObject>) FROM (<RefWorld>)

<Method> ::= GET PROPOSITION OF (<RefAgent> , <RefObject>)
| GET PROPOSITION OF (<RefAgent> , <RefObject>) IN (<RefWorld>)
| GET PROPOSITION OF (<RefAgent> , <RefObject>) FROM (<RefInstance>) IN (<RefWorld>)
| GET INTERROGATION OF (<RefAgent> , <RefObject>)
| GET INTERROGATION OF (<RefAgent> , <RefObject>) IN (<RefWorld>)
| GET INTERROGATION OF (<RefAgent> , <RefObject>) FROM (<RefInstance>) IN (<RefWorld>)
| GET RESULT OF (<RefAgent> , <RefObject>)
| GET RESULT OF (<RefAgent> , <RefObject>) IN (<RefWorld>)
| GET RESULT OF (<RefAgent> , <RefObject>) FROM (<RefInstance>) IN (<RefWorld>)
| SET CURRENT INSTANCE OF (<RefAgent> , <RefObject>) WITH <Value>
| SET CURRENT INSTANCE OF (<RefAgent> , <RefObject>) IN (<RefWorld>) WITH <Value>
| DELETE CURRENT INSTANCE OF (<RefAgent> , <RefObject>)
| DELETE CURRENT INSTANCE OF (<RefAgent> , <List-Object>)
| DELETE CURRENT INSTANCE OF (<RefAgent> , <RefObject>) IN (<RefWorld>)
| DELETE CURRENT INSTANCE OF (<RefAgent> , <List-Object>) IN (<RefWorld>)
| DELETE INSTANCE (<RefAgent> , <RefObject> , <RefInstance>)
| DELETE INSTANCE (<RefAgent> , <List-Object> , <RefInstance>)
| DELETE INSTANCE (<RefAgent> , <RefObject> , <RefInstance>) IN (<RefWorld>)
| DELETE INSTANCE (<RefAgent> , <List-Object> , <RefInstance>) IN (<RefWorld>)
| DELETE ALL INSTANCE OF (<RefAgent> , <RefObject>)
| DELETE ALL INSTANCE OF (<RefAgent> , <List-Object>)
| DELETE ALL INSTANCE OF (<RefAgent> , <RefObject>) IN (<RefWorld>)
| DELETE ALL INSTANCE OF (<RefAgent> , <List-Object>) IN (<RefWorld>)
| DELETE CURRENT WORLD OF (<RefAgent> , <RefObject>)
| DELETE CURRENT WORLD OF (<RefAgent> , <List-Object>)
| DELETE WORLD (<RefAgent> , <RefObject> , <RefWorld>)
| DELETE WORLD (<RefAgent> , <List-Object> , <RefWorld>)
| DELETE ALL WORLD OF (<RefAgent> , <RefObject>)
| DELETE ALL WORLD OF (<RefAgent> , <List-Object>)
| SET CURRENT WORLD OF (<RefAgent> , <RefObject>) WITH (<RefWorld>)

<List-Object> ::= <RefObject>
| <RefObject> , <List-Object>

<Procedure> ::= <InternalProc>
| <ExternalProc>

<InternalProc> ::= <Name> (<ParameterList>)

<ExternalProc> ::= <Name> (<ParameterList>)

<ParameterList> ::= <Parameter>
| <ParameterList> , <Parameter>

<Parameter> ::= <Name>
| <Litteral>
| <List>
| <Get>

```
<Loop>                ::=  DO <Expression> WHILE <Expression>
                         | WHILE <Expression> DO <Expression>
                         | FOR ( <Iteration-List> ) DO <Expression>
                         | FOR ( <Iteration-Ent> ) DO <Expression>

<Iteration-List>      ::=  <Name> IN <Name>
                         | <Name> IN <List>
                         | INT <Name> IN <Name>
                         | INT <Name> IN <Integer-List>
                         | FLOAT <Name> IN <Name>
                         | FLOAT <Name> IN <Real-List>
                         | STRING <Name> IN <Name>
                         | STRING <Name> IN <String-List>

<Iteration-Ent>       ::=  <Name> , <Condition>
                         | INT <Name> , <Condition>

<Condition>           ::=  <Boolean-Expression>

<Arithmetical-Expression> ::=  ( <TermA> <Operator> <TermA> )
                             | ( - <TermA> )

<TermA>               ::=  <Arithmetical-Expression>
                         | <Name>
                         | <Integer>
                         | <Float>
                         | <Get>

<Operator>            ::=  + | - | * | / | %

<Boolean-Expression>  ::=  ( <Term> <Comparator> <Term> )
                         | ( <Integer> <iselement> <Integer-List> )
                         | ( <Name> <iselement> <Integer-List> )
                         | ( <Integer> <iselement> <Name> )
                         | ( <Float> <iselement> <Real-List> )
                         | ( <Name> <iselement> <Ral-List> )
                         | ( <Float> <iselement> <Name> )
                         | ( <String> <iselement> <String-List> )
                         | ( <Name> <iselement> <String-List> )
                         | ( <String> <iselement> <Name> )
                         | ( <Name> <iselement> <Name> )

<Term>                ::=  <Value>
                         | <Get>
                         | <Arithmetical-Expression>
                         | <Method>
                         | NULL ( <list> )
                         | NULL ( <Name> )

<Comparator>          ::=  < | > | <= | >= | == | !=

<iselement>           ::=  ELEMENT | NELEMENT

<RefAgent>            ::=  <Name>
                         | SELF
                         | <Name>

<RefObject>           ::=  <Name>
                         | <parameter>

<RefAttribute>        ::=  <Name>
                         | <parameter>

<RefInstance>         ::=  <Integer>
                         | <parameter>

<RefWorld>            ::=  <Integer>
```

| <parameter>

<Value> ::= <Name>
| <Litteral>
| <LogicValue>
| <List>

<LogicValue> ::= TRUE
| FALSE

<Litteral> ::= <Integer>
| <Float>
| <String>

<List> ::= <Integer-List>
| <Real-List>
| <String-List>

<Integer-List> ::= ( )
| (<Body-Integer>)

<Body-Integer> ::= <Integer>
| <Integer> <Body-Integer>

<Real-List> ::= ( )
| (<Body-Real>)

<Body-Real> ::= <Float>
| <Float> <Body-Real>

<String-List> ::= ( )
| (<Body-String>)

<Body-String> ::= <String>
| <String> <Body-String>

Annexe C : Exemple, correspondance entre langage et librairie

Cet exemple n'est en aucun cas tiré d'une application réelle, le but étant seulement de donner un aperçu de la façon de programmer et non pas de montrer les capacités du système COALA en tant que résolution de problème en univers Multi-Agents.

fichier agent développé en langage d'agent : (fichier Agks1.co)

```
AGENT Agks1 <KS> {
     COMMENT "voici un agent ks";
     OBJECT Ob1 {
          COMMENT "objet numero 1";
          ATTRIBUTE Att1 <INT> (123) [1..500] {
          COMMENT "attribut numero 1";
          };
          ATTRIBUTE Att2 <FLOAT> (125.21);
          ATTRIBUTE Att3 <STRINGLIST>;
     };
     ACQUAINTANCE Agkp1;
     RULE R1 <PROPAGATION> {
          COMMENT "règle 1";
          IF
               INT A = GET ( SELF,Ob1,Att1) < 500;
          THEN
               SET ( SELF,Ob1,Att2) WITH ( 1235.5);
     };
     RULE R2 <PROPOSITION> {
          COMMENT "règle 2";
          IF
               INT A = GET ( SELF,Ob1,Att1) < 500;
          THEN
               SEND ( Ob1,Att1) TO ( Agkp1 );
     };
     RULE R3 <INTERROGATION> {
          COMMENT "règle 3";
          IF
               INT B = 0;
               (
               STRING A = GET INTERROGATION OF ( SELF, Ob1 )
               &&  A == "YES"
               &&  B = GET ( SELF, Ob1, Att1)
               &&  B < 500;
               );
          THEN
               FLOAT C = 0;
               (
               ASK ( Ob1, Att2 ) TO ( Agkp1 )
               &&  SET ( SELF, Ob1, Att2 ) WITH ( C )
               );
     };
};
```

L'agent KS 'Agks1' contient un objet contenant lui-même trois attributs, il a trois règles et n'est connecté qu'à un agent KP 'Agkp1'.

La description de cet agent est développée dans un fichier de ressources portant le nom de l'agent, avec l'exension '.co'. L'"équivalent" C++ de ce fichier de ressources est dans une série de classes décrites à partir des différents composants de ce fichier.

Nous obtiendrons :

- une classe pour chaque objet,
  - C_Agks1_Ob1
- une classe pour chaque attributt,
  - C_Agks1_Ob1_Att1
  - C_Agks1_Ob1_Att2
  - C_Agks1_Ob1_Att3
- une classe pour chaque règle,
  - C_Agks1_R1
  - C_Agks1_R2
  - C_Agks1_R3

ainsi qu'un fichier principal qui contiendra la description de l'agent lui même.

La suite donne la description des classes créées :

Constantes :

Les constantes développées ici servent à repérer les différents composants (Agent, Objet, Attribut) dans les tableaux créés lors de l'application

```
const    int Agks1 = 0;
const    int Agks1_Ob1 = 0;
const    int Agks1_Ob1_Att1 = 0;
const    int Agks1_Ob1_Att2 = 1;
const    int Agks1_Ob1_Att3 = 2;
const    int Agkp1 = 1;
```

Classes 'Attribut' :

Suivant le type de l'attribut la classe créée hérite de 'AttributeInt' pour entier, 'AttirbuteFloat' pour réel ...Le constructeur appelé lors de l'instanciation de la classe fait appel au constructeur approprié de la classe générique suivant les paramètres d'initialisation.

```
class C_Agks1_Att1 : public AttributeInt {
public:
    C_Agks1_Att1() : AttributeInt("Att1",123,1,500,"attribut
numero 1") {}
    C_Agks1_Att1(AttributeInt* a) : AttributeInt(a) {}
};
class C_Agks1_Att2 : public AttributeFloat {
public:
    C_Agks1_Att2() : AttributeFloat("Att2",125.21) {}
    C_Agks1_Att2(AttributeFloat* a) : AttributeFloat(a) {}
};
class C_Agks1_Att3 : public AttributeListString {
public:
    C_Agks1_Att3() : AttributeListString("Att3") {}
    C_Agks1_Att3(AttributeListString* a) : AttributeListString(a) {}
};
```

Classe 'Objet' :

De même que pour un attribut il y a héritage et appel du constructeur de la classe générique. Mais ici le constructeur est complété par la construction du tableau d'attributs dépendant de l'application. De même il y a élaboration d'une méthode de création et duplication d'instances. Mais ces méthodes font appel à des méthodes de la classe générique.

```
class C_Agks1_Ob1 : public Object {
public:
    C_Agks1_Ob1() : Object("Ob1",3,"objet numero 1")
    {
        Attributes.AddAttribute(new C_Agks1_Att1);
        Attributes.AddAttribute(new C_Agks1_Att2);
        Attributes.AddAttribute(new C_Agks1_Att3);
    }
    UNSIGNED CreateInstance()
    {
        Ptr_Instance i;
        if (CurrentInstance(i))
        {
            i->AddAttribute(new C_Agks1_Att1);
            i->AddAttribute(new C_Agks1_Att2);
            i->AddAttribute(new C_Agks1_Att3);
            return 1;
        }
        return 0;
    }
    UNSIGNED DuplicateInstance(Ptr_Attribute* TabAttributes)
    {
        Ptr_Instance i;
        if (CurrentInstance(i))
        {
            i->AddAttribute(new C_Agks1_Att1((AttributeInt*)

    TabAttributes[Agks1_Ob1_Att1]));
            i->AddAttribute(new C_Agks1_Att2((AttributeFloat*)

    TabAttributes[Agks1_Ob1_Att2]));
            i->AddAttribute(new C_Agks1_Att3((AttributeListString*)

    TabAttributes[Agks1_Ob1_Att3]));
            return 1;
        }
        return 0;
    }
};
```

Classes 'règles' :

Toujours sur le même principe, il y a création d'une classe par règle. Ici les seules méthodes sont les redéfinitions des méthodes abstraites 'Premise' et 'Conclusion' Qui sont dépendantes évidement de l'application. On peut voir la correspondance entre les mots-clés du langage et les méthodes de la classe, par exemple ici 'GET' et 'Get', les paramètres de la méthodes étant les paramètres développés dans la régle y compris la valeur de retour, les valeurs retournées par les méthodes des classes étant toujours 0 ou 1. On peut remarquer

l'utilisation de variables de classes (Tmp.) créées lors d'utilisation de variables dans les règles du langage.

```
class C_Agks1_R1 : public KSRule {
protected:
    int Tmp;
public:
    C_Agks1_R1() : KSRule("regle 1") {Tmp=0;}
    ~C_Agks1_R1() {}
    UNSIGNED Premise()
    {
        return Get(SELF, Agks1_Ob1, Agks1_Ob1_Att1, Tmp)
                                    && LT(Tmp, 500);
    }
    UNSIGNED Conclusion()
    {
        return Set(SELF, Agks1_Ob1, Agks1_Ob1_Att2, 1235.5);
    }
};
class C_Agks1_R2 : public KSRule {
protected:
    int Tmp;
public:
    C_Agks1_R2() : KSRule("regle 2") {Tmp=0;}
    ~C_Agks1_R2() {}
    UNSIGNED Premise()
    {
        return Get(SELF, Agks1_Ob1, Agks1_Ob1_Att1, Tmp)
                                && LT(Tmp, 500);
    }
    UNSIGNED Conclusion()
    {
        return Send(Agkp1, Agks1_Ob1, Agks1_Ob1_Att1);
    }
};
class C_Agks1_R3 : public KSRule {
protected:
    float Tmp;
    int Tmp2;
    String Tmp3;
public:
    C_Agks1_R3() : KSRule("regle 3") {Tmp=0;Tmp2=0;}
    ~C_Agks1_R3() {}
    UNSIGNED Premise()
    {
        return GetInterrogationFlag(SELF, Agks1_Ob1, Tmp3)
                && EQ(Tmp3, "YES")
                && Get(SELF, Agks1_Ob1, Agks1_Ob1_Att1, Tmp2)
                && LT(Tmp2, 500);
    }
    UNSIGNED Conclusion()
    {
        return Ask(Agkp1, Agks1_Ob1, Agks1_Ob1_A2, Tmp)
            && Set(SELF, Agks1_Ob1, Agks1_Ob1_A2, Tmp);
    }
};
```

L'agent KS est développé dans un fichier de même nom, mais avec une extension '.C' qui inclut les fichiers contenant les classes précédentes et sera compilées "classiquement" avec le compilateur C++.

Fichier Agks1.C :

L'agent est développé comme un fichier et fait appel aux méthodes d'ajout des différents éléments qui le constituent, à savoir Objet, règle et connection.

```
Main()
{
     KS     agent ("AgKs1", ...);
     (void)   agent. AddObject(new C_Agks1_Ob1 );
     (void)   agent. AddPropagation (new C_Agks1_R1 );
     (void)   agent. AddProposition (new C_Agks1_R2 );
     (void)   agent. AddInterrogation (new C_Agks1_R3 );
     (void)   agent. AddConnection (...);
     (void)   agent.KSMainLoop();
} ;
```

ANNEXE D : Critères de qualité.

Le but de cette annexe est de donner au lecteur une définition des critères de qualité dans le but de produire du logiciel de qualité, et les principes en découlant. Les informations suivantes ont été retirées de l'ouvrage de Bertrand Meyer [MEYER 90].

Deux types de facteurs de qualité sont à considérer. Les facteurs externes, ce sont ceux vus par l'utilisateur du produit et les facteurs internes, ce sont ceux vus par les informaticiens "générateur" du produit. En final seul les facteurs externes comptent mais ils ne peuvent être perçus sans la prise en compte des facteurs internes. Les plus importants de ces facteurs sont :

Validité :
c'est l'aptitude d'un produit logiciel à réaliser exactement les tâches définies par sa spécification.

Robustesse :
c'est l'aptitude d'un logiciel à fonctionner même dans des conditions anormales.

Fiabilité :
concept général qui couvre à la fois validité et robustesse.

Extensibilité :
c'est la facilité d'adaptation d'un logiciel aux changements de spécification.

Réutilisabilité :
c'est l'aptitude d'un logiciel a être réutilisé en tout ou en partie pour de nouvelles applications.

Compatibilité :
c'est l'aptitude des logiciels à pouvoir être combinés les uns avec les autres.

Efficacité :
c'est la bonne utilisation des ressources dans l'espace et dans le temps.

Portabilté :
c'est la facilité avec laquelle un produit logiciel peut être adapté a différents environnements matériels et logiciels.

Vérifiabilité :
c'est la facilité de préparation des procédures de recettes et de certification, notamment des tests et des procédures de déboggages.

Intégrité :
c'est l'aptitude des logiciels à protéger leurs différentes composantes contre des accès ou des modifications non autorisés.

Facilité d'utilisation :
c'est la facilité avec laquelle les utilisateurs d'un logiciel peuvent apprendre comment l'utiliser, comment le faire fonctionner, comment préparer les données, mais aussi comment interpréter les résultats et réparer les effets en cas d'erreur.

Ces qualités là sont nécessaires dés que l'on désire qu'un logiciel non seulement réponde à ses spécifications mais puisse également recouvrir sa zone de vie appelé maintenance. En effet les activités de maintenances gèrent plus d'évolutions que de corrections. Il est donc indispensable qu'un minimum de **modularité** soit présente dans un logiciel, cette modularité n'étant possible que si l'on prend en compte les paramètres évoqués ci-dessus. Cette modularité nécessite les critères suivants :

Décomposabilité modulaire :
réduire la complexité apparente de la description d'un système initial en la décomposant en un ensemble de sous-systèmes moins complexes organisés en une structure simple. Ce processus est répétitif : les sous-systèmes doivent eux-mêmes être décomposés.

Composabilité modulaire :
favorise la production d'éléments de logiciel qui peuvent être combinés librement les uns avec les autres pour produire de nouveaux systèmes, éventuellement dans un environnement très différent de celui pour lequel ils ont été initialement développés.

Compréhensibilité modulaire :
aide à produire des modules dont chacun peut être compris isolément par un lecteur humain. Au pire, le lecteur ne doit avoir à consulter que quelques modules voisins.

Continuité modulaire :
une petite modification de la spécification du problème n'amène à modifier qu'un seul module, ou peu de modules, d'un système produit grâce à la méthode issue de cette spécification. De telles modifications ne doivent pas avoir d'impact sur l'architecture du système, c'est-à-dire sur les relations entre les modules.

Protection modulaire :
conduit à des architectures dans lesquelles l'effet d'une condition anormale, se produisant à l'exécution d'un module, reste localisé à ce module, tout au moins ne se propage qu'à quelques modules voisins.

De ces critères B. Meyer déduit cinq principes pour obtenir une bonne modularité :

Unités modulaires linguistiques :
les modules doivent correspondre à des unités syntaxiques du langage.

Peu d'interfaces :
tout module doit communiquer avec aussi peu d'autres que possible.

Petites interfaces :
si deux modules communiquent, ils doivent échanger aussi peu d'informations que possible.

Interfaces explicites :
chaque fois que deux modules A et B communiquent, cela doit ressortir clairement du texte de A, de B ou des deux.

Masquage de l'information :
toute information doit être privée, sauf si elle est explicitement déclarée publique.

ANNEXE E : Informations techniques.

Le laboratoire dispose de plusieurs stations de travail de marques différentes, (SUN, APOLLO, SILICON GRAPHICS, SONY, Compatible PC), ainsi qu'un ensemble bureautique constitué de MacIntosh Apple. Ces derniers sont reliés entre eux par un réseau Appleshare. Plusieurs réseaux internes Ethernet reliés par des passerelles interconnectent le tout avec le réseau régional GRENET lui même connecté au réseau international.

Le Matériel :

Le développement s'est effectué sur les stations de travail SUN et SONY. Les caractéristiques sont les suivantes :

- 2 stations "SUN 4 SPARC 2", permettant une puissance de 28,5 MIPS :
  - processeur RISC cadencé à 40 Mhz avec une ante-mémoire de 64 Koctets,
  - une mémoire RAM de 16 Moctets,
  - 2 disques durs de 207 Moctets,
  - écran graphique couleur de 19 pouces à haute résolution de 1152 x 900 points,
- 3 stations "SUN SPARC IPC", permettant une puissance de 15,8 MIPS :
  - processeur RISC cadencé à 40 Mhz avec une ante-mémoire de 6 Koctets,
  - une mémoire RAM de 12 Moctets,
  - 1 disque dur de 207 Moctets, (1 station possédant en plus un disque dur externe de 424 Moctets)
  - écran graphique couleur de 16 pouces à haute résolution de 1152 x 900 points,
- 1 station "SONY" NWS 3865, permettant une puissance de 17 MIPS :
  - processeur RISC R3000 cadencé à 20 Mhz avec une ante-mémoire données de 64 Koctets et ,une ante-mémoire instructions de 64 Koctets,
  - une mémoire RAM de 8 Moctets,
  - 1 disque dur de 240 Moctets,
  - écran graphique couleur de 19 pouces à haute résolution de 1280 x 1024 points,

- Plusieurs périphériques :
  - imprimantes Listing, Laser, Couleur,
  - Streamer,
  - CD ROM,
  - floppy disque.

Le Logiciel :

Le système d'exploitation sur les stations SUN est SUN OS 4.1.2 compatible UNIX System V. SUN OS 4.1.2 est le premier pas vers le système d'exploitation SOLARIS, et correspond à la version 1.1. Le futur est l'utilisation de SOLARIS 2.1.

Les développements ont été effectués avec le langage C++, dans un premier temps avec la version 2.0 puis avec la version 3.0 qui est à la norme ANSI. Cette version a été utilisée avec le compilateur 'gcc'. Il s'agit d'un compilateur natif de GNU (UNIX du domaine public) et avec les bibliothèques de classes s'y rapportant, notamment les classes génériques de type Liste.

Il a été fait usage des outils du système d'exploitation UNIX, notamment LEX++ et YACC++ qui sont les versions C++ de LEX et YACC.

L'interface graphique utilisé est MOTIF. C'est une sur-couche sur XWindows et X11.

ANNEXE F : Définitions Générales.

Accointance
Liaison familière, fréquentation, lien, Connaissance, relation

Acteur
- Artiste dont la profession est de jouer un rôle à la scène ou à l'écran.
- Personne qui prend une part active, joue un rôle important

Agent
- Celui qui agit.
- Ce qui agit, opère. Force, corps, substance intervenant dans la production de certains phénomènes.

Classe
- Dans un groupe social, ensemble des personnes qui ont en commun une fonction, un genre de vie, une idéologie, etc...
- Ensemble d'individus ou d'objets qui ont des caractères communs.

Communication
- Le fait de communiquer, d'établir une relation avec quelqu'un.
- Toute relation dynamique qui intervient dans un fonctionnement
- Action de communiquer quelque chose à quelqu'un. Résultat de cette action.
- Moyen technique par lequel des personnes communiquent; messages qu'elles se transmettent.
- Ce qui permet de communiquer. passage d'un lieu à un autre.

Connaissance
- Le fait ou la manière de connaître.
- Faculté de connaître propre à un être vivant.
- Le fait de sentir de percevoir.
- Connaissances : ce qui est connu; ce que l'on sait, pour l'avoir appris.
- Relation qui s'établit entre personnes.

Environnement
- Action d'environner (mettre autour de); Son résultat.
- Enceinte. Environs d'un lieu.
- Ensemble des conditions naturelles (physiques, chimiques, biologiques) et culturelles (sociologiques) susceptibles d'agir sur les comportements et les activités humaines.
- Conditions extérieures susceptibles d'agir sur le fonctionnement d'un système, d'un dispositif.

Intelligence
- Faculté de connaître, de comprendre.
- L'ensemble des fonctions mentales ayant pour objet la connaissance conceptuelle et rationnelle.
- Aptitude d'un être vivant à s'adapter à des situations nouvelles.
- Qualité de l'esprit qui comprend et s'adapte facilement

Koala

KOALA (phascolarctos),

Classification

règne	:	**Animal,**
embranchement	:	**Vertébrés,**
classe	:	**Mammifères,**
sous-classe	:	**Méthatériens,**
ordre	:	**Marsupiaux,**
sous-ordre	:	**Diprotodontes,**
groupe	:	**Phalangéroïdes,**
famille	:	**Phalangéridés.**

Ce mammifère dont le nom signifie "qui ne boit pas" ("koolahs" en dialecte arborigène) est cantonné au continent australien. La niche écologique qu'il a occupé au fil des âges en a fait un animal tout à fait original. Tout d'abord il ne se nourrit que de feuilles d'eucalyptus, en sélectionnant seulement une douzaine d'espèces parmi les 500 existantes. Il en retire toute sa nourriture et l'eau nécessaire à son alimentation. Ensuite c'est un animal lent, il ne vit que dans les arbres. Il n'avait pas de prédateur, avant que l'homme ne vienne bouleverser son milieu naturel. Son mode de reproduction qui a contribué à le classer dans les marsupiaux (du latin marsupium qui signifie bourse) est également original. Il possède une poche dorsale, contrairement à la plupart des autres marsupiaux, chez qui elle est ventrale, dans laquelle est élevé le petit Koala. Il ne naît qu'un seul petit à la fois après une brève gestation de 4 à 5 semaines. L'embryon n'étant pas fixé dans un placenta, il est expulsé prématurément, ce n'est alors qu'une larve de 5 à 6 grammes qui se traîne jusqu'à la poche. Il s'y alimente du lait maternel mais aussi peu à peu d'une substance verte constituée d'aliments prédigérés par sa mère qui lui communique ainsi la bactérie essentielle à la digestion des coriaces feuilles d'eucalyptus.

Ces curieux animaux ont eu à souffrir des déforestations intensives (leur cadre de vie) et de l'attrait qu'a suscité leur fourrure. Ces ingérences de l'homme dans son univers a considérablement réduit le nombre d'individus. Il a fallu des lois draconiennes et la création de parcs nationaux pour qu'il puisse survivre. L'image montrant une mère Koala transportant sa minuscule réplique sur son dos, toutes deux avec un museau aplati et des oreilles touffues, l'a rendu extrêmement populaire.

Langage

- Fonction d'expression de la pensée et de communication entre les hommes, mise en oeuvre au moyen d'un système de signes vocaux et éventuellement de signes graphiques qui constitue une langue.
- Tout système de signes vocaux qui remplit cette fonction.
- Tout système secondaire de signes crée à partir du langage d'une langue.
- Tout système d'expression et de communication que l'on assimile au langage naturel.
- Façon de s'exprimer. Usage qui est fait quant à la forme, de cette fonction d'expression. Usage propre à un groupe d'individus.

Méthodologie

Etude des méthodes scientifiques, techniques. Subdivision de la logique.

Objet

- Toute chose concrète, perceptible aux sens (notamment à la vue, le toucher).
- Chose quelconque produit de l'activité humaine destinée à un certain usage.
- Tout ce qui occupe l'esprit, toute matière à réflexion, à étude, à observation.
- En philosophie : par opposition au moi, au sujet, tout ce qui existe en dehors de l'esprit.
- But d'une action, d'un comportement.

Bibliographie

[ARCANGELI 93] J.P. Arcangeli, A. Marcoux, C. Maurel, P. Sallé
"Le Langage d'Acteur PLASMA II et les Systèmes Multi-Agents ".
Premiéres journées francophones IAD & SMA, (pp 181-191)
Toulouse Avril 1993

[BARTHES 90] J.P. Barthès,
"Objet et Intelligence Artificielle".
Actes des 3emes journées PRG-GDR Intelligence Artificielle
CNIT Paris la défense Mai 1990.

[BAUJARD 92] O. Baujard,
"Conception d'un environnement de développement pour la résolution de problèmes: apport de l'intelligence articielle distribuée et application à la vision".
Thèse de l'université Joseph Fourier Grenoble 1992

[BOND 88] H. Bond, L. Gasser,
"An analysis of problems and research in DAI".
Reading in Distributed Artificial Intelligence
Ed Bond&Gasser(pp3-35), Morgan Kaufman Publishers, 1988

[BOURON 91] T. Bouron, J. Ferber, F. Samuel,
"MAGES: a Multi-Agents testbed for heterogenous agents".
Decentralized A.I 2 (pp 195-214)
Demazeau & Müller Eds. Elsevier Science Publishers 1991

[BRUNO 91] G. Bruno,
" MAPS : un nouveau compilateur".
Rapport de stage IUT II Grenoble.

[BURCKERT 91] H.J. Burckert, J. Müller,
"RATMAN : Rational Agents Testbed for Multi-Agents Network".
Decentralized A.I 2 (pp 217-230)
Demazeau & Müller Eds. Elsevier Science Publishers 1991

[CAILLAUD 91] B. Caillaud,
"Architecture Multi-Agents pour la reconnaissance de la parole".
Rapport de DEA Signal Image Parole Grenoble 1991

[DAVID 92] J.M. David, A. Nguyen,
"Projet ESPRIT: CONSTRUCT".
Bulletin de l'AFIA, 10 (pp19-22) juillet 1992.

[ERCEAU 91] J. Erceau, J. Ferber,
"L'Intelligence Artificielle Distribuée".
La Recherche 233 vol 22 (pp 750-758) 1991

[ERCEAU 93] J. Erceau, J. Ferber,
"Introduction aux premières journées francophones d'Intelligence Artificielle Distribuée et Systèmes Multi-Agents".
Premières journées IAD & SMA Toulouse Avril 1993.

[FERBER 87] J. Ferber,
"Des objets aux agents: une architecture stratifiée".
AFCET/RFIA, Antibes 1987. Dunod Informatique.

[FERBER 89] J. Ferber,
"Objets et agents: une étude des structures de représentation et de communications en Intelligence Artificielle".
Thèse de doctorat d'état,
Université Pierre et Marie Curie, Paris 1989.

[GARBAY 88] C. Garbay, S. Pesty,
"MAPS un système Multi-Agents pour la résolution de problème".
Actes du 6eme congrès RFIA-AFCET, Paris 1988.

[GASSER 87] L. Gasser, C. Braganza, N. Herman,
"MACE: a flexible testbed for distributed AI research".
Distrbuted Artificial Intelligence Vol 1 (pp119-154), Morgan Kaufman Publishers.

[HATON 89] J.P Haton,
"Panorama des systèmes Multi-Agents".
Onzième journée francophones sur l'informatique: Architecture avancées pour l'Intelligence Artificielle,
EC2 Editeur (pp 247-261) 1989.

[HAUTIN 86] F. Hautin, A Vailly,
"La coopération entre systèmes experts".
Actes des journées nationales des PRC-GRECO "Intelligence Artificielle", Cepadues Edition 1986

[HAYES-ROTH 88] F. Hayes-Roth, L.D Erman, S. Fouse, J.S Lark, J. Davidson,
"ABE: a cooperative operating system and development environment".
Readings in distributed Artificial Intelligence (pp 457-488)
A BOND & L GASSER Eds 1988.

[HAYES-ROTH 91] F. Hayes-Roth, J. Davidson,L.D Erman, J.S Lark,
"Framework for developing intelligent systems: the ABE systems engineering environment".
IEEE Expert (pp 30-40) IEEE 1991.

[HEITZ 91] M.Heitz,
" Structuration des classes, objets, types et données dans la méthode HOOD".
AFCET/INTERFACES numéro spéciale n°103-104 Mai-Juin 1991.

[HEWITT 73] C.E. Hewitt, P. Bishop, R. Steiger,
"A Universal Modular Actor Formalism for Artificial Intelligence".
Proceeding of 3rd IJCAI, (pp 235-245)
Standford, California 1973

[HUGONNARD 93] E. Hugonnard,
"Modèles multi-agent pour l'acquisition des connaissances en biomédecine".
Thèse de Doctorat en Génie Biomédicale,
Université Scientifique et Technologique de Compiegne, 28 juin 1993.

[JEANNE 93] F. Jeanne,
" Un repositionnement vital pour l'IA".
Dossier Intelligence Artificielle
Le Monde Informatique n°546 24Mai 1993

[JENNINGS 91] N.R Jennings,
"Cooperation in industrial systems".
ESPRIT conference Brussels 25-29 Novembre 1991.

[JONCKERS 92] V.Jonckers, S. Geldof, K. De Vroede,
"The COMMET methodology and Workbench in practice".
VUB AI Lab Memo Avril 92

[LAURENT 85] J.P Laurent,
"La structure de contrôle dans les systèmes experts".
Techniques et Sciences Informatiques Vol 1; N° 1 & 2, 1985

[LOINGTIER 92] J.M. Loingtier,
"Le projet ARCHON".
Bulletin de l'AFIA, 12 (pp37-41) avril 1992.

[MASINI 90] G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre,
"Les langages à objets".
IIA.InterEditions. 1990.

[MEYER 90] B. Meyer,
"Conception et programmation par objets".
IIA.InterEditions. 1990.

[OVALLE 91] D.A Ovalle,
"Contribution à l'étude du raisonnement en univers Multi-Agent: KIDS, une application pour l'interprétation d'images biomédicales".
Thèse de l'université Joseph Fourier Grenoble 1991

[PESTY 89] S. Pesty, C. Garbay,
"Maps : A Multi-Agent Problem Solving Environment".
Proc IASTED Int. Symp. on Expert System Theroy and Application,
(pp 110-113) Acta Press 1989

[PITRAT 92] J. Pitrat,
" Allen Newell et les débuts de l'Intelligence Artificielle".
Bulletin de l'AFIA, n°11 (pp52-55) novembre 1992.

[PLEIAD 92] Pôle et Lieu d'Echanges en Intelligence Artificielle Distribuée,
"Vers une taxinomie du vocabulaire pour les Systèmes Multi-Agents".
PRC-GDR Intelligence Artificielle journe Systèmes Multi-Agents
Nancy 18 Décembre 1992.

[REEVES 90] H. Reeves,
"Malicorne".
Collection Seuil "science ouverte" 1990

[RIFFLET 90] J.M. Rifflet,
"La communication sous UNIX".
Mc GRAW HILL , Paris 1990

[SOMMERVILLE 88] I. Sommerville,
"Le génie logiciel et son application".
IIA. InterEditions. 1988.

[STEELS 92] L.Steels,
"Reusability and configuration of applications by non-programmers ".
VUB AI Lab Memo 92-4 Avril 92

[SWADE 93] D. Swade,
" Le calculateur mécanique de Charles Babbage".
Pour La Science n° 186 pp78-84 Avril 1993.