Des objets concurrents aux agents autonomes

Zahia Guessoum, Jean-Pierre Briot et Michel Dojat*
LIP6, Boîte 169, 4, place de Jussieu, 75252 PARIS Cedex 5
INSERM, 8, avenue du Général Sarrail, F-94010 Créteil
e-mail: {guessoum, briot, dojat}@laforia.ibp.fr

Cet article étudie une extension du concept d'objets actifs en une structure d'agents autonomes. Nous présentons d'abord les caractéristiques des agents autonomes qui ne sont pas assurées par de simples objets actifs. Nous proposons ensuite une extension du comportement simple d'un objet actif en un ensemble de comportements et un méta-comportement pour superviser leurs interactions. Comme application concrète de ces idées, nous décrivons l'extension d'une plate-forme d'objets actifs (appelée *Actalk*), en une plate-forme multi-agents générique (appelée *DIMA*). Nous discutons de l'implémentation de cette extension. Finalement, nous rapportons des expérimentations de *DIMA* sur le contrôle de la ventilation artificielle des patients en rénamiation.

1 Introduction

La programmation orientée-objets concurrente (POOC) est l'outil le plus naturel pour implémenter les agents. L'objet actif, élément de base de la POOC, peut être considéré utilement comme une brique de base pour l'implémentation de modèles d'agents. Par ailleurs, l'intégration du concept d'agents et du paradigme d'objets mène à la "programmation par agents" [Sho91] [Sho93]. Ainsi, nous nous inscrivons dans la voie de la "programmation par agents" en nous plaçant dans un environnement où la communication entre agents s'effectue par envoi de messages et où les canons de la programmation par objets, tels que l'encapsulation et l'héritage, sont bien respectés.

Un agent est naturellement implémenté comme un objet concurrent. Ce dernier offre une certaine autonomie du fait qu'il ne nécessite pas de ressources externes pour être activé, mais son comportement reste procédural en réaction aux requêtes de messages. Pour améliorer son autonomie, différents travaux proposent d'enrichir cet objet actif d'une fonction lui permettant de contrôler le traitement des messages reçus en considérant son état interne (voir par exemple [MIT90] [Bri94]). Toutefois, l'activité d'un agent n'est pas réduite à la réception/transmission de messages. Pour être autonome, l'objet actif doit avoir différentes activités (perception, raisonnement, communication, action, ...) et doit pouvoir contrôler toutes ces activités.

Les questions de base pour construire une passerelle entre les exigences de l'implémentation et de la modélisation des systèmes d'IAD (voir par exemple [CL90] [Gas92] [Cas95]), et les facilités d'implémentation et de modélisation fournies par la POOC [GB92] sont :

- quelle est l'architecture d'agents la plus générique et la mieux appropriée pour représenter les principales caractéristiques d'un agent autonome ?
- comment étendre le mécanisme de représentation hautement-structuré de la POOC en une structure d'implémentation d'agents relativement générique [GB92] ?

Cet article est une tentative de réponse à ces deux questions. Il traite 1) des exigences de la modélisation des systèmes d'IAD en proposant une architecture générique et modulaire d'agents et 2) de l'extension des facilités de modélisation et d'implémentation de la POOC. Nous voudrions rendre la programmation par objets concurrente plus appropriée aux problèmes d'IAD en lui incorporant des représentations spécifiques et des structures de contrôle appropriées aux exigences de l'IAD. L'article présente une structure d'agents appropriée au contrôle et à la coordination des actions d'un agent autonome dans un univers dynamique multi-agents. Plus concrètement, dans cet article nous décrivons l'extension d'un modèle d'objets actifs (incarné par la plate-forme *Actalk*), en un modèle d'agents générique et modulaire (incarné par la plate-forme *DIMA*).

L'article comprend cinq paragraphes. Dans le paragraphe 2, nous présentons brièvement *Actalk*, quelques limitations des objets actifs ainsi que les principales exigences de l'IAD. Dans le paragraphe 3, nous décrivons une architecture générique d'agents et leur mécanisme d'auto-contrôle. Le paragraphe 4 décrit l'implémentation de cette architecture d'agents à l'aide d'*Actalk*. Dans le paragraphe 5, nous rapportons des expérimentations de *DIMA* sur le contrôle de la ventilation artificielle de patients en milieu hospitalier. Nous terminons l'article par une discussion pour souligner l'intérêt de cette extension et quelques perspectives.

2 Objets actifs

Le concept d'objets actifs (appelés aussi acteurs) a été introduit par C. Hewitt [Hew77] pour décrire un ensemble d'entités qui coopèrent et communiquent par envoi de messages. Ce concept apporte les avantages de la programmation par objets aux environnements distribués et fournit aux langages à objets quelques caractéristiques des systèmes ouverts [AH85]. Différents langages basés sur ces caractéristiques ont été proposés.

2.1 Actalk

Actalk est une plate-forme qui implémente différents types de modèles d'objets actifs [YT87]. Elle est fondée sur l'existence d'un noyau qui modélise des objets actifs communiquant par envoi de messages asynchrones. La communication asynchrone, principe de base des langages d'objets actifs, est implémentée par une communication synchrone puis dépôt du message dans un tampon ("mail-Box") ; il y a ainsi dissociation entre envoi et interprétation du message.

Les principales classes qui décrivent les trois composantes de base d'un objet actif sont les suivantes (voir Figure 1) :

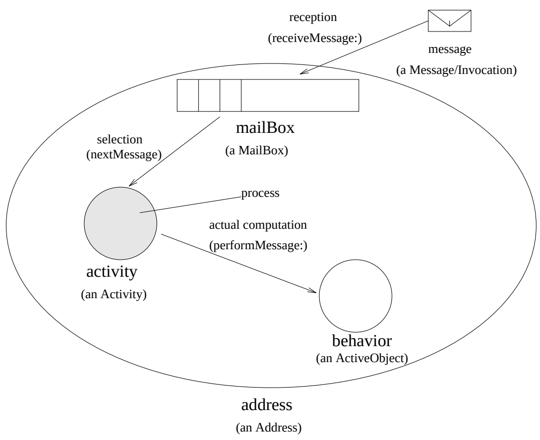

Figure 1 - *Le modèle d'objets actifs dans Actalk*

- La classe `Address` représente l'adresse d'un objet actif. Elle définit la politique de réception de toutes les requêtes faites à l'objet actif. Elle décrit donc les différents types de communication.
- La classe `Activity` représente l'activité interne d'un objet actif. Elle décrit la manière dont les messages sont sélectionnés, séquencés et activés en contrôlant les messages placés dans la boîte aux lettres. Elle spécifie la politique de gestion des messages. Elle définit un processus pour retirer

continuellement les messages présents dans la boîte aux lettres et lancer leur interprétation spécifiée par le comportement de l'objet actif.
- La classe `ActiveObject` décrit le comportement de l'objet actif et la réponse aux requêtes qui lui sont transmises par l'objet décrivant son activité.

Ces trois classes de base sont ensuite progressivement spécialisées (sous-classées) pour définir différents modèles d'objets actifs (modèles de communication, modèles d'activités et de synchronisation).

2.2 Limitations des objets actifs

La POOC nous fournit des fondations puissantes pour modéliser et implémenter des agents. Cependant, ces puissantes et utiles fondations ne fournissent pas réellement une structure d'agents autonomes. Les objets actifs sont monolithiques et ils ont un comportement procédural. Si un objet actif ne reçoit pas de messages des autres objets, il reste inactif. On souhaiterait alors leur fournir un certain degré de proactivité. De plus, comme cela a été constaté par J. Ferber [Fer89], malgré leur apparence de sujets communiquants, ces objets ne savent pas réfléchir sur leur comportement, sur les liens qu'ils entretiennent avec d'autres objets. Afin de dépasser ces limites, il faut donner aux objets actifs les moyens d'avoir différents comportements et les moyens de contrôler ces comportements. Il faut passer de la notion d'objets à celle d'agents.

Différents chercheurs ont enrichi le concept d'objets actifs pour définir des agents :

- T. Maruichi a défini les concepts à rajouter aux objets actifs pour avoir des agents autonomes [MIT90]. Il a introduit un interpréteur de messages pour réaliser l'autonomie et la notion d'environnement pour former des groupes d'agents.
- T. Bouron a utilisé la théorie des actes de langages pour améliorer la communication entre objets actifs [Bou92].
- Y. Shoham a introduit les états mentaux pour avoir un comportement basé sur les interactions [Sho91] [Sho93].
- etc.

Ces travaux sont des propositions très intéressantes. Cependant, ils n'offrent pas une structure d'agents générique qui couvre entièrement le spectre des exigences de l'IAD [GB92]. Ainsi, ils ne permettent pas d'intégrer différentes capacités (exemple : perception, raisonnement, communication) de manière modulaire et sélective. Enfin, l'unique comportement de ces agents reste souvent non déclaratif.

2.3 Quelques exigences de l'IAD

La POOC fournit un support de base pour l'IAD sans pour autant offrir la structure d'agents générique requise. Pour mieux illustrer l'insuffisance de la POOC, nous présentons brièvement quelques caractéristiques des agents qui ne sont pas fournies par les objets actifs :

- Un agent peut avoir différents comportements. Des exemples de comportements sont : 1) un agent interagit de façon asynchrone et concurrente avec son environnement pour avoir un modèle des autres ; 2) il traite les requêtes des autres agents et génère les réponses adéquates pour ces requêtes ; et 3) il raisonne pour déterminer les actions les mieux appropriées dans le contexte courant.
- Un agent possède des compétences qui peuvent varier de simples opérations à des raisonnements divers. Ces derniers permettent à chaque agent de représenter et raisonner sur les actions, les plans, les croyances, les intentions et les connaissances des autres agents afin de coordonner son action avec celles des autres. En effet, certains comportements de l'agent doivent intégrer des mécanismes d'IA.
- Un agent est une entité autonome. Il agit sans l'intervention des humains ou des autres agents, et il contrôle ses actions en fonction de son état interne et de son environnement [Cas90] [Cas95] [DM90]. La notion d'autonomie nécessite la résolution de multiples problèmes [Vig92] : 1) comment "comprendre" et "s'adapter" à la "réalité"? 2) comment élaborer en conséquence des plans d'actions? et surtout 3) comment prendre en compte non seulement les pannes, mais les perturbations dans la perception de l'environnement et l'action sur cet environnement? Pour être autonome, chaque agent doit donc être doté d'un mécanisme d'auto-contrôle afin de pouvoir gérer ses différents comportements en fonction de son état interne et celui de son monde externe.

Dans le paragraphe suivant, nous présentons une architecture générique d'agents qui répond à ces exigences.

3 Une architecture générique d'agents

L'architecture d'agents que nous proposons (voir Figure 2) comprend une première strate composée de modules décrivant les différentes comportements d'un agent et une deuxième strate représentant le méta-comportement de l'agent. Elle permet à l'agent de raisonner et d'agir sur ses différents comportements.

3.1 Les comportements de l'agent

La modélisation de systèmes complexes nécessite des agents qui combinent des capacités cognitives pour raisonner sur des situations complexes et des

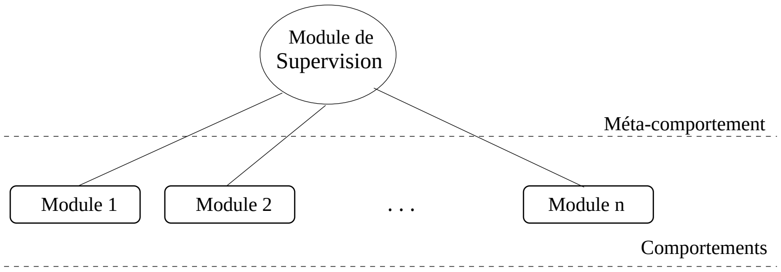

Figure 2 - *Le modèle d'agents proposé*

capacités réactives pour respecter des échéances. Ainsi, un agent peut avoir deux types de comportements : procéduraux (stimuli-réponses) et à base de connaissances (cognitifs).

Dans ce paragraphe, nous donnons trois exemples de modules : le module de perception (comportement procédural), le module de raisonnement (comportement à base de connaissances) et le module de communication (comportement procédural ou à base de connaissances en utilisant par exemple les actes de langages).

- *Le module de perception* gère les interactions entre l'agent et son environnement. Le rôle principal de ce module est d'initialiser et de mettre à jour un ensemble de données qui regroupe les valeurs de variables externes accessibles à l'aide de capteurs.
- *Le module de raisonnement* représente les différentes compétences de l'agent. Il définit la réponse adaptée, soit à la requête reçue via le module de communication, soit à l'évolution de l'environnement perçue via le module de perception. La réponse de l'agent est fonction de son savoir-faire et de ses connaissances. Ainsi, le module de raisonnement intègre un système à base de connaissances.
- *Le module de communication* réalise deux fonctions : 1) la réception de messages en provenance d'autres agents et 2) l'envoi de messages pour communiquer avec les accointances de l'agent. Ce module peut réutiliser le mécanisme de bas-niveau de communication des modèles d'objets actifs. Dans ce cas, il est procédural. Mais les agents utilisent souvent des mécanismes de communication de plus haut-niveau en utilisant, par exemple, les actes de langages [Bou92]. Ce module peut alors intégrer un système à base de connaissances.

Ces trois modules s'avèrent suffisants pour de nombreuses applications (voir le paragraphe 5). Par ailleurs, le choix d'une approche modulaire facilite l'intégration de nouveaux modules tels que le module de planification, le module d'apprentissage et le module d'action.

3.2 Le méta-comportement de l'agent

La première strate de l'architecture proposée comprend un ensemble de modules des comportements qui peuvent être de nature différente (procédurale ou à base de connaissances). Ces différents modules interagissent de façon asynchrone et concurrente. Le problème est alors de gérer ces interactions et d'adapter les comportements à base de connaissances aux comportements procéduraux. Il s'agit donc de définir les mécanismes et les stratégies nécessaires pour gérer les interactions entre ces différents modules, soit d'imposer un séquencement temporel global interne à l'agent. La solution que nous proposons consiste à introduire une deuxième strate qui représente le méta-comportement de l'agent. Ce méta-comportement permet à chaque agent d'adapter les comportements à base de connaissances aux comportements procéduraux. Nous avons ainsi introduit un *module de supervision*.

Le module de supervision repose sur les notions d'*état* et de *transition*. Ces états et ces transitions définissent un ATN (Augmented Transition Network) [Woo70] qui est une représentation synthétique et déterministe du comportement de l'agent. Les états marquent les étapes importantes de la vie d'un agent : ils représentent une phase temporaire de sa vie. Ils sont reliés par des transitions qui représentent les changements d'états. Un agent ne change d'état qu'après l'apparition d'un événement (réception d'un nouveau message, perception d'une alarme, ...). Les états définissent ainsi le contexte dans lequel prennent place les événements qui marquent la vie de l'agent.

3.3 Auto-contrôle

L'auto-contrôle est géré à deux niveaux (voir Figure 3) : 1) au niveau du module de supervision, un ATN permet de spécifier le comportement de l'agent en fonction des états internes observés et 2) au niveau du module de raisonnement, une architecture de niveau méta permet de spécifier de façon déclarative les stratégies de raisonnement de l'agent.

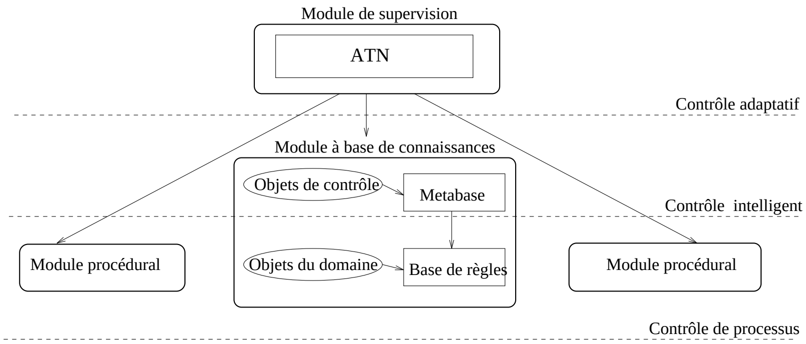

Figure 3 - *L'architecture d'auto-contrôle*

3.3.1 *Module de supervision : un contrôle adaptatif*

Le module de supervision permet à l'agent d'adapter dynamiquement son comportement aux changements qu'il détecte dans son univers. Ce module représente le *scheduler* des activités de l'agent.

Le découplage entre les comportements procéduraux (tel que la perception) et les comportements à base de connaissances (tel que le raisonnement), et l'utilisation de processus concurrents pour représenter ces différents comportements offrent une réactivité aux agents ainsi définis. Chaque agent peut détecter en temps réel les modifications de son univers. A chaque transition, le module de supervision évalue les conditions de transition pour déterminer les actions les mieux appropriées. A chaque transition le module de supervision peut ainsi adapter les comportements à base de connaissances aux comportements procéduraux. Autrement dit, l'agent peut adapter son comportement aux changements de son univers.

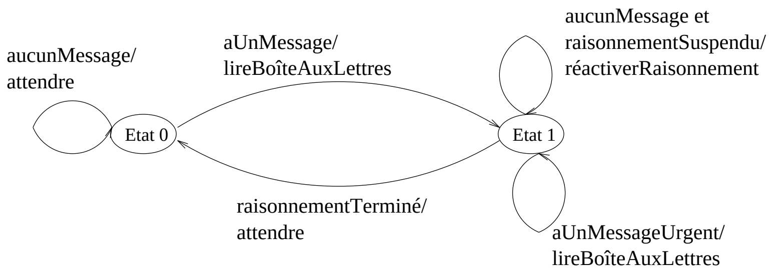

Figure 4 - *Exemple d'ATN*

La figure 4 donne l'exemple d'un ATN qui gère deux modules (communication et raisonnement). A partir de *Etat0*, la condition "n'a aucun message" conduit à l'action "attendre" et au maintien dans *Etat0* alors que la condition "a un message" entraîne l'action "lire la boîte aux lettres" et le passage à *Etat1*. Quel que soit l'état courant, la condition "a un message urgent" entraîne toujours l'action "lire la boîte aux lettres". La stratégie de l'agent qui possède cet ATN consiste à privilégier le traitement de messages urgents à toute autre activité.

3.3.2 *Niveau de raisonnement : un contrôle intelligent*

Nous considérons que si les stratégies de contrôle du raisonnement de l'agent doivent être séparées de ses connaissances du domaine, elles ne doivent pas pour autant lui échapper. Nous avons donc choisi d'associer au sein de l'agent ces deux types de connaissances contrairement à certaines approches de contrôle à base de tableau noir [HR85].

Le système *NéOpus* que nous utilisons pour doter nos agents d'une capacité de raisonnement, offre une architecture déclarative de contrôle qui repose sur le principe de réflexivité suivant (voir Figure 3) : le contrôle de l'exécution d'une base de règles est vu comme un problème qui nécessite des connaissances. L'exécution d'une base de règles est entièrement contrôlée par l'exécution d'une autre base de règles de niveau méta appelée métabase [Pac95]. Des objets spécifiques (évaluateurs) qui représentent à chaque instant l'état courant du raisonnement, sont utilisés pour réifier le contrôle. Ces objets peuvent aussi être utilisés par l'ATN pour agir sur le raisonnement. Des métarègles de contrôle peuvent permettre de spécifier des stratégies de coordination inter-agents (échange d'informations en cours de raisonnement par exemple)

4 Implémentation

Pour implémenter le modèle d'agents proposé, nous avons réutilisé le noyau de base de la dernière version d'*Actalk*. Dans ce noyau, chaque objet actif est défini par une activité, un comportement et une adresse. Son activité est procédurale et non flexible.

Nous avons enrichi ce noyau pour implémenter l'architecture d'agent proposée et offrir une structure d'agents générique fidèle à cette architecture. Pour rendre flexible l'activité de l'agent, nous avons séparé la partie procédurale de l'ATN (son interpréteur) de sa partie déclarative (la spécification de l'ATN). Dans *DIMA*, un agent est un objet actif composé (voir Figure 5) :

- d'un objet (instance de la classe `AgentActivity` sous-classe de `Activity`) qui réifie l'interpréteur de l'ATN,
- d'un objet (instance de la classe `BasicAgent` sous-classe de `ActiveObject`) qui décrit le méta-comportement de l'agent,
- d'un ensemble d'objets qui décrivent les modules de la première strate de l'agent (perception, raisonnement, communication, ...).

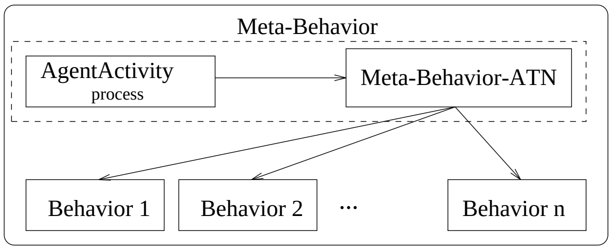

Figure 5 - *Différents objets qui implémentent un agent*

4.1 L'activité d'un agent

Dans *Actalk*, un objet `Activity` gère et transmet les messages qui sont interprétés par un autre objet `ActiveObject`. La méthode d'instance `body` utilisée dans `createProcess` consiste à créer un processus pour continuellement retirer et traiter les messages présents dans la boîte aux lettres.

```
!Activity methodsFor: 'activity setting'!
 body
  [true] WhileTrue: [self acceptNextMessage].

 createProcess
  [self body] newProcess
```

Dans DIMA, l'activité d'un agent est décrite par un ATN qui permet de contrôler ses différents comportements. Nous avons alors redéfini la méthode d'instance `body`. La nouvelle fonction de ce processus consiste à interpréter l'ATN de l'agent.

```
!AgentActivity methodsFor: 'activity setting'!
  body
  [true] whileTrue: [self atnInterpreter].

!AgentActivity methodsFor: 'atn'!
 atnInterpreter
  |state atn|
  atn := self meta-behavior atn.
  state := atn initialState.
  state = atn finalState whileFalse:
               [state := atn transitionAt: state].
```

4.2 Les comportements et le méta-comportement de l'agent

Le contrôle ou la simulation de processus dynamiques complexes tels que le monitorage de procédés industriels, la surveillance de patients en unités de soins intensifs ou l'évolution économique des entreprises, nécessitent des agents qui peuvent varier de simples entités à des entités complexes. Pour doter it DIMA d'agents de différentes granularités, nous avons implémenté une hiérarchie de classes décrivant le méta-comportement de ces agents.

La figure 6 donne un exemple de hiérarchie de méta-comportements. Nous avons considéré trois comportements : perception, raisonnement et communication. Par exemple, `CommunicatingPerceivingReasoning` supervise les trois comportements.

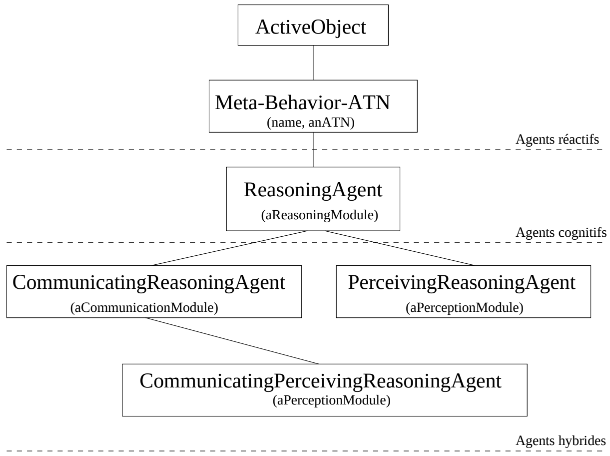

Figure 6 - *Exemple de hiérarchie de classes d'agents*

A chaque classe est associé un ATN pour définir un méta-comportement. Elle implémente : 1) les conditions et les actions des transitions de l'ATN et une ou plusieurs méthodes de création d'agents.

Dans *DIMA*, l'implémentation d'un agent est souvent limitée à la définition de ses bases de connaissances pour définir les comportements à base de connaissances, et à l'instanciation de classes existantes pour décrire les autres comportements.

DIMA implémente également des hiérarchies de classes décrivant les modules de la première strate. Les classes les plus génériques sont :

– `ProceduralBehavior` qui décrit les modules procédaux,
– `KnowledgeBasedBehavior` qui implémente les modules à base de connaissances. Elle utilise une base de règles implémentée avec le système à base de règles *NéOpus* [Pac92].

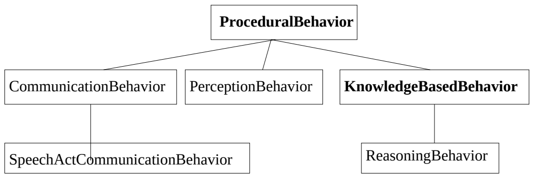

Figure 7 - *Exemples de classes décrivant les comportements de l'agent*

Les classes décrivant le module de communication implémentent une fonction qui contrôle l'exécution de messages en considérant l'état interne de l'agent. Elle a une variété d'opérations `send` qui permettent de modéliser la notion de *groupe* telle qu'elle a été proposée dans [MIT90]. Chaque groupe est réifié comme un simple agent dont la fonction principale est de retransmettre les messages aux agents associés.

5 Exemple d'application : le contrôle de la ventilation artificielle

NéoGanesh [Doj94] est utilisé pour contrôler la ventilation mécanique de patients hospitalisés dans des unités de soins intensifs à l'hôpital Henri Mondor (Créteil). *NéoGanesh* acquiert en temps réel différents signaux ventilatoires (débit inspiratoire, fréquence respiratoire et pression de CO2 expiré), diagnostique l'état courant du patient et adapte en conséquence la thérapie, soit l'assistance mécanique fournie par le respirateur. Pour réaliser cela, *NéoGanesh* développe un raisonnement relativement complexe pour construire à différents niveaux d'abstraction une image de l'évolution de l'état du patient [DS96].

Pour améliorer la réactivité et l'adaptabilité de *NéoGanesh* et pour faciliter l'introduction de nouvelles connaissances et la distribution sur plusieurs machines, nous l'avons remodélisé avec *DIMA*. Le nouveau système baptisé *NéoGanDi* réutilise la totalité des connaissances de *NéoGanesh*. Il comprend sept agents (voir la figure 8).

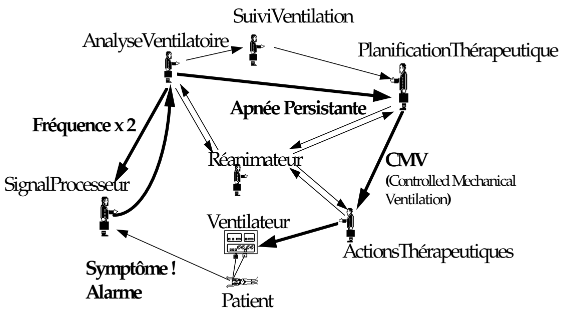

Figure 8 - *NéoGanDi : Une société d'agents*

Une comparaison des différentes versions de ce système de contrôle de ventilation artificielle a été effectuée. Pour cette comparaison, nous avons mesuré :

- Le temps de réactivité qui est le temps entre la date d'apparition d'une alarme et la date de sa perception par le système.
- Le temps de réponse qui est le temps entre la date de perception de l'alarme et le temps d'action. Ce temps comprend le temps d'analyse, le temps de planification et le temps d'action.

Nous avons lancé 1000 expérimentations (sur une station SUN SPARC 10) pour mesurer ces différents intervalles de temps. Le résultat montre que le temps de réactivité de *NéoGanDi* est nettement inférieur à celui de *NéoGanesh*. Cette amélioration est principalement due au mécanisme d'auto-contrôle proposé (voir paragraphe 3.2). Ce résultat montre également l'intérêt de distribuer le système réellement sur plusieurs machines. Par exemple, le temps de moyen de réactivité (nombre d'agents * temps d'une transition / 2) est de 37 ms sur une seule machine, mais il est réduit à 12 ms (une seule transition) si l'on implémente l'agent SignalProcesseur sur une première machine et les autres sur une deuxième.

Verions	temps de réactivité	Temps de réponse Sit. Normale	temps de réponse Sit. Alarmante
NéoGanesh (une machine)	1000 ms	4232 ms	4067 ms
NéoGanDi (une machine)	37 ms	4738 ms	3475 ms
NéoGanDi (deux machines)	12 ms	4240 ms	2156 ms

En situation normale, le temps de réponse de *NéoGanesh* est légèrement inférieur à celui de *NéoGanDi*. Cette dégradation est principalement due à l'élimination de variables globales et l'utilisation des messages asynchrones. Par contre, en situation d'alarme, le temps de réponse de *NéoGanDi*, notamment sa version distribuée, est nettement inférieur.

Le temps de réponse total du système *NéoGanesh* en situation alarmante (temps de réactivité + temps de réponse) est ainsi amélioré de 57 % en utilisant la version distribuée de *NéoGanDi*.

6 Conclusion

Dans cet article, nous avons présenté une extension des objets actifs pour décrire une structure d'agents autonomes. Il propose une architecture méta-niveau pour décomposer le comportement d'un agent en une organisation de comportements.

La plate-forme développée (*DIMA*) intègre différents *frameworks* à base de classes *Smalltalk*. En plus du *framework* d'objets actifs (*Actalk*), elle utilise le *framework* de simulation à événements discrets pour représenter et gérer l'évolution temporelle des agents. Elle utilise également le *framework NéOpus* pour implémenter les bases de connaissances des agents.

Pour valider *DIMA*, nous avons développé trois applications : 1) *Eddaks*, un simulateur de procédés industriels de fabrication [GD95], 2) *NéoGanDi* [GD96a], un système multi-agents pour le contrôle de la ventilation artificielle et 3) *Meveco* [GD96b], un système multi-agents pour la modélisation de l'évolution des entreprises.

DIMA, le résultat d'extension d'*Actalk*, offre un *framework* intéressant pour étudier les problèmes multi-agents. Par exemple, les agents implémentés avec *DIMA* sont principalement caractérisés par la réactivité et l'adaptabilité aux changements de leur environnement. Pour avoir des agents temps-réel, nous étudions actuellement une technique de raisonnement *anytime* [ZR96]. Les expérimentation réalisées [Gue96] offrent des résultats intéressants. Ces expérimentations étudient uniquement le temps réel au niveau de l'agent. Un fonctionnement temps réel global d'un système multi-agents est subordonné à un fonctionnement temps réel local à un agent. Cependant, ceci n'est pas suffisant pour garantir un système multi-agents temps réel. Il semblerait donc intéressant d'étendre cette étude au niveau multi-agents.

Références

[AH85] G. Agha and C. Hewitt. Concurrent programming using actors: Exploiting large scale parallelism. In *5th Conference On Foundations of Software Technology & Theoritical Computer Science*, pages 19–41, 1985.

[Bou92] T. Bouron. *Structures de communication et d'organisation pour la coopération dans un univers multi-agents.* PhD thesis, Université Paris-6, Laforia, 1992.

[Bri94] J-P Briot. Modélisation et classification de langages de programmation concurrente à objets : l'expérience actalk. In *LMO'94*, pages 103–125, Grenoble, 1994.

[Cas90] C. Castelfranchi. *Decentralized AI*, chapter A Point Missed in Multi-Agent, DAI and HCI, pages 49–62. Elsevier, Amsterdam, 1990.

[Cas95] C. Castelfranchi. Guarantees for autonomy in cognitive agent architecture. In *Intelligent Agents: Theories, Architectures and Languages, LNAI Volume 890*, pages 56–70, 1995.

[CL90] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(1):225–253, 1990.

[DM90] Y. Demazeau and J-P Müller. *Decentralized AI*, chapter Decentralized Artificial Intelligence, pages 3–16. Elsevier, Amsterdam, 1990.

[Doj94] M. Dojat. *Contribution à la représentation d'expertises médicales dynamiques : application en réanimation médicale.* PhD thesis, Université de Compiègne, 1994.

[DS96] M. Dojat and C. Sayettat. A realistic model for temporal reasoning in real-time patient monitoring. *Applied Artificial Intelligence*, 10(2):121–143, 1996.

[Fer89] J. Ferber. *Objets et agents : une étude des structures de représentations en intelligence artificielle.* PhD thesis, Université Paris 6, 1989.

[Gas92] L. Gasser. *An Overview of DAI.* Kluwer Academic Publisher, Boston, 1992.

[GB92] L. Gasser and J-P. Briot. *Distributed Artificial Intelligence: Theory and Praxis*, chapter Object-Oriented Concurrent Programming and Distributed Artificial Intelligence, pages 81–108. Kluwer Academic Publisher, 1992.

[GD95] Z. Guessoum and P. Deguenon. A multi-agent approach for distributed discrete-event simulation. In *DISMAS'95*, pages 183–190, Poland, November 1995.

[GD96a] Z. Guessoum and M. Dojat. A real-time agent model in an asynchronous object environment. In *MAAMAW'96*, pages 190–203, NL, January 1996. Springer Verlag.

[GD96b] Z. Guessoum and R. Durand. Un système multi-agents pour la modélisation de l'évolution économique. In J-P Muller et J. Quinqueton, editor, *JFIADSMA'96*, pages 47–57. Hermès, 1996.

[Gue96] Z. Guessoum. *Un environnement opérationnel de conception et de réalisation de systèmes multi-agents.* PhD thesis, Laforia, Université Paris 6, 1996.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[HR85] B. Hayes-Roth. Blackboard architecture for control. *Artificial Intelligence*, 1(26):251–321, 1985.

[MIT90] T. Maruichi, M. Ichikawa, and M. Tokoro. *Decentralized AI*, chapter Modeling Autonomous Agents and their Groups, pages 215–134. Elsevier Science, 1990.

[Pac92] F. Pachet. *Représentation de connaissances par objets et règles : le système NéOpus.* PhD thesis, Université Paris 6, Laforia, 1992.

[Pac95] F. Pachet. On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming 8(4)*, pages 19–24, 1995.

[Sho91] Y. Shoham. Agent0: An agent-oriented programming language and its interpreter. In *AAAI-91*, pages 704–709, 1991.

[Sho93] Y. Shoham. Agent-oriented programming. *AI*, 60(1):139–159, 1993.

[Vig92] G. Vignaux. *Les sciences cognitives, une introduction.* Découverte, Paris, 1992.

[Woo70] W. Woods. Transition network grammar for natural language analysis. *Communication of Association of Computing Machinery*, 10(13):591–606, 1970.

[YT87] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming.* The MIT Press, 1987.

[ZR96] S. Zilberstein and S. Russel. Optimal composition of real-time system. *AI*, 1(82):181–213, 1996.