Rapport ASA
Année 2001-2002
Le problème de la Vente aux Enchères

Groupe ASA, PRC IA
Mise en forme : P Mathieu

Résumé

Depuis plusieurs années, des équipes françaises, à l'instar des équipes américaines, ont proposé puis développé des plateformes d'aide à la conception de systèmes multi-agents. Bien que toutes ces plateformes se réfèrent à la même notion générale d'"agent"[1], il est néanmoins difficile de distinguer les avantages et inconvénients de chacune. Des tableaux récapitulatifs de caractéristiques ont été publiés mais jamais personne n'avait tenté d'aborder un même problème à l'aide de chacune de ces plateformes. L'objectif de cet article collectif est donc de montrer par l'exemple comment utiliser chacune de ces plateformes. Afin d'aider le lecteur dans sa comparaison, un "toy problem" fédérateur a été choisi : la simulation de vente aux enchères par des agents.

1 Introduction

Depuis la démocratisation de l'Internet et plus précisément du commerce électronique, les systèmes informatiques de ventes aux enchères ont acquis un regain d'intérêt conséquent. `Ibazar.fr`, `Kelkoo.fr`, `Yahoo.fr` et beaucoup d'autres, proposent des environnements à cet effet. Des chercheurs comme P. Maes avec le système Kasbah s'y sont d'ailleurs intéressés de manière approfondie. Dans ce problème il existe deux sortes d'agents parfaitement identifiés (les acheteurs et les vendeurs). Chaque acheteur peut avoir son propre raisonnement. Les agents acheteurs communiquent obligatoirement avec le vendeur et le fonctionnement du système est naturellement distribué. Ce problème semblait donc tout à fait adapté à notre objectif. Néanmoins, afin d'aligner tout le monde sur la même ligne

[1] La définition suivante semble assez consensuelle : Entité automone, douée de raisonnement et capable de communiquer avec ses semblables

de départ tout en restant “Toy Problem”, il nous fallait nous contraindre à l’aide de spécifications précises.

Tout d’abord notons que la vente aux enchères de type “Salle de vente” est différente de la vente de type “Internet”. Dans une salle de vente les produits sont vendus un par un tandis que sur Internet un acheteur peut enchérir sur de nombreux produits simultanément. Dans une salle de ventes, l’enchère sur un produit se termine quand plus personne ne renchérit tandis que sur Internet c’est à une date fixe que l’enchère s’arrête. Ces différences fondamentales impliquent notamment que dans une salle de vente un acheteur peut raisonner sur l’argent qu’il lui reste pour acheter les produits qui seront présentés tandis que cela est impossible sur Internet. Nous choisirons donc l’approche “Salle de Vente” pour notre problème.

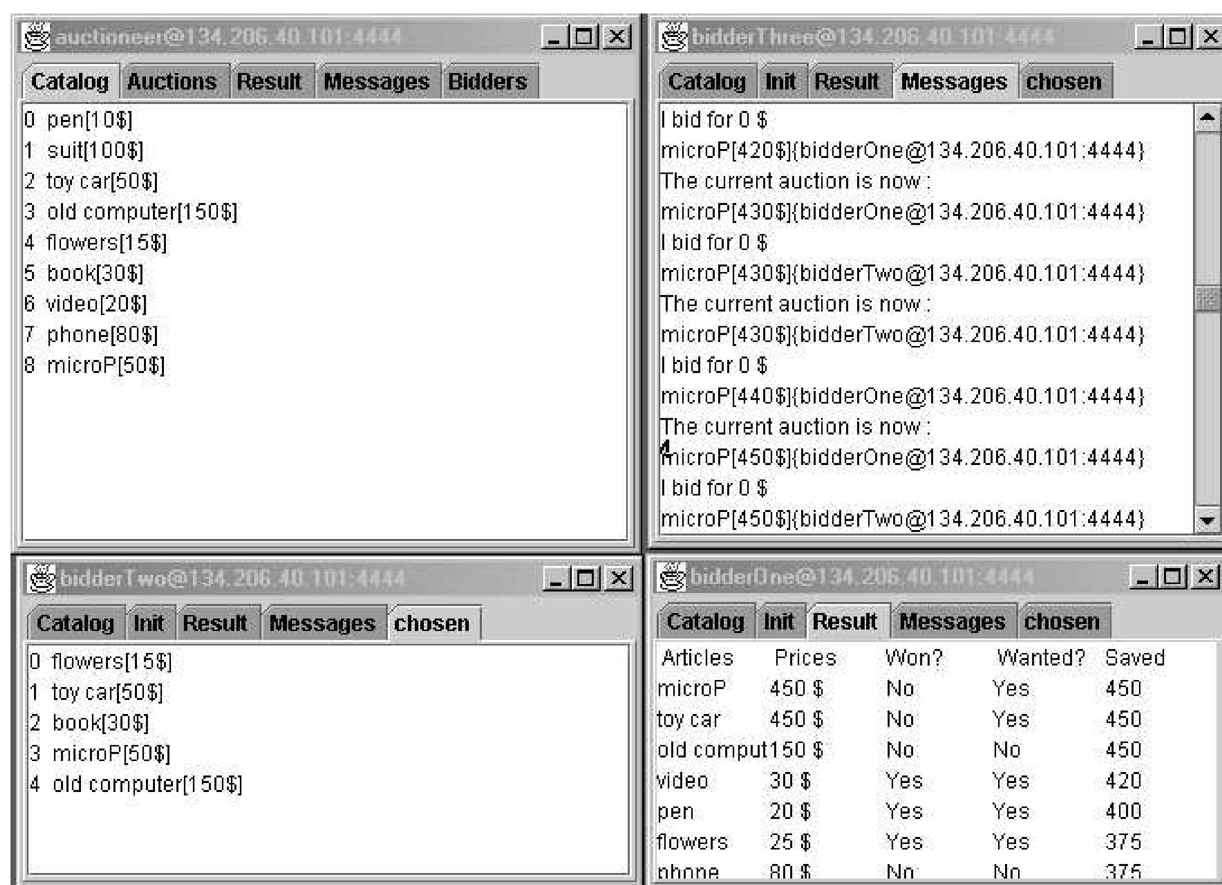

Fig. 1 – Système avec 1 vendeur et 3 acheteurs

Enoncé du Toy Problem. *On considère donc un agent vendeur et plusieurs agents acheteurs. Au début de la session le vendeur envoie le catalogue d’articles à chaque acheteur. Ensuite il lance les enchères séquentiellement dans un ordre non déterminé. De son*

côté, chaque acheteur décide d'acheter n produits du catalogue aléatoirement. Il dispose d'une somme limitée d'argent et a pour but d'acquérir le maximum d'articles dans l'ensemble qu'il a tiré au sort. A chaque enchère, le vendeur donne le nom de l'acheteur actuel avec le nouveau montant du produit à tous les autres acheteurs. Chaque agent implémente sa propre stratégie d'achat qui peut être plus ou moins agressive et/ou complexe. Pour rester conforme au modèle "salle de ventes", à chaque début de vente, le vendeur initialise un Timer. Si le Timer se termine sans que personne n'ait renchérit, la vente est terminée. A chaque renchérissement, le Timer est réinitialisé. Le système doit contenir un ou plusieurs agents humains qui permettent à des personnes humaines de renchérir si elles le souhaitent.

On pourrait bien sûr aller beaucoup plus loin en offrant pour chaque agent des stratégies d'achat plus ou moins agressives. On pourrait aussi permettre aux agents d'effectuer une étude du comportement des autres acheteurs afin de s'adapter au profil de chacun mais cela sort du cadre de notre comparaison.

Les différentes plateformes françaises que nous abordons dans ce papier sont :

- Magique du LIFL, Lille (rédacteur P Mathieu)
- Geamas de l'Irémia, La réunion (redacteur R Courdier)
- SMAS de Compiègne (rédacteur JP Barthes)
- Dima du LIP6, Paris (rédacteur Z Guessoum)
- oRis de l'ENIB, Brest (rédacteur P Chevaillier)
- Mask du Leibniz, Grenoble (rédacteur M Occello)
- Madkit du LIRMM, Montpellier (rédacteur O Gutknecht)

2 Le système Magique

2.1 Description générale de la plateforme

Magique est une plateforme de développement d'applications multi-agents développée à l'Université de Lille dans l'équipe SMAC[2]. Elle se base sur une communauté de hiérarchies d'agents qui permet une totale maîtrise de la granularité du contrôle [MRS00, BM95, BM97, MT00]. Une originalité de ce modèle est de permettre aux agents d'exploiter de manière transparente (ie. sans qu'ils sachent qui l'accomplira) toutes les compétences des différents agents du système. Ajouté aux possibilités d'évolution dynamique des agents, de la structure des communications et de l'organisation, ce mécanisme contribue à la souplesse de développement et d'exécution d'applications multi-agents. Un agent Magique est une coquille vide dans laquelle on greffe des compétences qui peuvent ensuite être échangées entre les différents agents. Cette plateforme est écrite en Java, ce qui permet une distribution des agents sur un réseau hétérogène.

2.2 Résolution du problème

Tout naturellement, la racine de la hiérarchie est constitué de l'agent vendeur sur lequel les agents acheteurs se connectent. L'agent vendeur possède la compétence `AuctioneerSkill` et les agents acheteurs la compétence `BidderSkill`. Tout cela s'écrit très simplement en Magique : une compétence est décrite par une classe qui hérite de `Skill` tandis qu'un agent hérite de la classe `Agent`. Le fait d'hériter d'agent permet à chacun de communiquer directement avec les autres de manière centralisée ou distribuée avec une gestion multi-threadée des messages. La connexion au niveau hiérarchique se fait par la méthode `connectToBoss` et l'appel d'une compétence chez un autre agent se fait à l'aide des méthodes `perform` ou `askNow`. Commençons par décrire l'interface du service de vente :

```
public class AuctionneerSkill
       extends DefaultSkill
{
private Catalog catalog;
Private Timer t;

    // Constructeur qui construit le
    // catalogue à la vente.
    public AuctionneerSkill(){...}

    // Choisit un article au hasard pour
    // la vente.
    public Article chooseArticle(Catalog c)
```

[2] http ://www.lifl.fr/SMAC

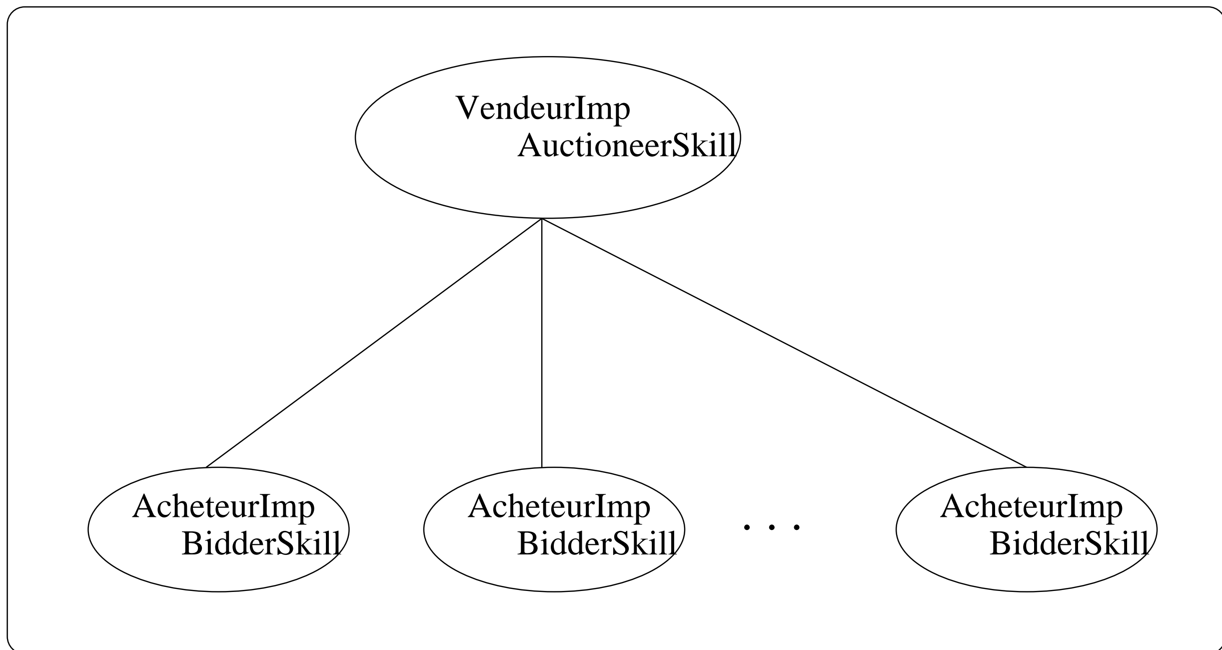

FIG. 2 – hiérarchie des agents dans le SMA

```
    // permet la vente entière du catalogue
    public void sellAll()

    // Permet la vente d'un article et
    // initialise le Timer de vente.
    public void sellArticle(Article a)

    // Envoie une enchère à tous les
    // acheteurs connectés à chaque fois
    // que quelqu'un renchérit.
    public void sendArticleDescription(Auction a)

    // Envoie le catalogue à chaque agent
    // qui se connecte
    public void sendCatalog(String who)

    // Méthode appelée par les acheteurs
    // pour proposer une enchère.
    void setLastAuction(Auction a)
}
```

L'ensemble de ces méthodes est décrit très classiquement si ce n'est que l'invocation d'une compétence d'un agent vendeur se fait par `perform` ou `askNow`.

Par exemple `perform("agent","competence",args...)` appellera la méthode nommée `competence` de l'agent `agent` avec les arguments correspondants, tandis que `perform("competence",args...)` recherchera automatiquement le premier agent possédant cette compétence. Il existe en réalité trois méthodes d'invocation de compétences en `Magique` : `perform` pour les appels de services sans valeur de retour, `ask` our les appels non synchronisés avec retour et `askNow` pour les appels synchronisés avec valeur de retour. L'appel de ces trois méthodes se fait de manière identique, soit sous la forme d'un appel nommé (on sait à qui l'on s'adresse), soit par appel à la cantonnade et c'est `Magique` qui recherche quelqu'un capable de réaliser ce service. Cette dernière invocation est bien sûr préférable puisqu'elle est intépendante des agents présents dans le système. Il suffit que la compétence soit quelque part pour qu'elle puisse être réalisée.

Une fois le service implémenté, sa greffe dans un agent est très aisée. Il suffit de créer une plateforme d'hébergement de l'agent qui peut éventuellement en contenir plusieurs, puis de greffer la compétence `AuctionneerSkill` à l'agent.

```
public class VendeurImp {
  public static void main() {
      Platform p = new Platform();
      Agent a = p.createAgent("vendeur");
      a.addSkill(new AuctionneerSkill(this));
  }
}
```

Passons maintenant à la réalisation du service d'achat. Chaque acheteur au départ est initialisé avec une stratégie d'achat plus ou moins agressive. Il possède une certaine somme d'argent et choisit quelques articles à acheter au catalogue. L'implémentation de ce service est résumée ici :

```
public class BidderSkill
       extends DefaultSkill {
  private Vector MyCatalog;

  // Constructeur qui calcule la liste
  // des produits à acheter en fonction
  // du catalogue
  public Bidder(){...}

  // Choisit un nombre fixé d'articles
  // du catalogue pour l'achat
  public Catalog chooseArticles(Catalog c)
  {...}

  // Accesseurs pour le catalogue, la
  // somme d'agent disponible par l'agent
  // et la stratégie employée par cet agent.
```

```
  public void setCatalog(Catalog c) {...}
  public void setMoney(Integer i) {...}
  public void setStrategy(Integer i ) {...}

  // Appelée par le vendeur à chaque enchère.
  // Permet à l'acheteur de renchérir.
  public void buyArticle(Auction a) {...}

  // Appelée par le vendeur à la fin du
  // Timer. Signifier la fin de la vente
  // à l'acheteur.
  public void endArticle(Auction a) {...}
}
```

La réalisation d'un agent possédant ce service est maintenant très classique. Tout se passe comme pour l'agent `Vendeur` mais cette fois-ci il faut penser à connecter l'acheteur au vendeur. Le vendeur est donc un superviseur au sens Magique, on connecte l'acheteur à celui-ci par la méthode `connectToBoss`. :

```
public class AcheteurImp {
    public static void main() {
        Platform p = new Platform();
        Agent a = p.createAgent("acheteur");
        a.addSkill(new BidderSkill(this));
        a.perform("setMoney",500);
        ....
        a.connectToBoss("vendeur");
    }
}
```

La méthode `connectToBoss` permet de connecter n'importe quel agent à n'importe quel autre à travers un réseau IP. Le paramètre de cette méthode indique l'adresse IP de l'agent sous la forme `nom@numpIP :port` permettant ainsi une distribution très aisée des agents à travers le réseau. Quand les agents sont sur la même machine il peuvent bien sûr être identifiés par leur nom court comme dans l'exemple précédent.

2.3 Conclusion

Magique est une architecture de SMA écrite sous forme d'une API Java. Cette architecture permet de construire des agents à partir d'une coquille vide dans laquelle on greffe des services. L'écriture d'un service ressemble à l'écriture d'une classe dans laquelle on a remplacé l'appel de méthodes externes par un méthode générique de type `perform`, `ask`, ou `askNow`. L'API s'occupe entièrement de l'aspect distribué et du multi-threading des communications entre agents. L'agent peut bien sûr implémenter plusieurs services et dans le cas de notre exemple devra aussi implémenter le service "interface graphique" aussi bien

pour l'acheteur que pour le vendeur. Magique offre de nombreuses autres possibilités que celles présentées ici, notamment en ce qui concerne la mise en concurrence des agents, la gestion des accointances et l'échange dynamique de compétences.

3 La plateforme SMAS

3.1 Description générale de la plate-forme

SMAS est une plate-forme instrumentée de développement et de mise au point d'agents ayant des comportements complexes, qui seront ensuite implantés dans l'environnement OSACA [Sca96, SVdAB96, BS97]. Les agents sont totalement indépendants, mais forment un groupe. Les communications se font à l'aide de plusieurs protocoles ("basic" et plusieurs versions de Contract-Net) qui peuvent être spécifiés pour chaque message, la mise en œuvre du Contract-Net étant entièrement transparente ; les messages sont typés (à la KQML) ; les messages peuvent être envoyés en "point-to-point", "broadcast", ou "multi-cast" ; les agents sont clonés à partir d'un agent générique qui possède une structure assez complète ; les agents peuvent dynamiquement entrer et sortir du groupe ; la simulation se fait à l'aide d'un scénario. SMAS a été entièrement développé en CommonLisp.

3.2 Résolution du problème

Les agents sont définis à l'aide de la commande defagent. Chaque agent doit avoir une identité unique, ex :`(defagent AUCTIONEER)`. La vente fait appel à plusieurs compétences. Tout d'abord une compétence permettant de distribuer le catalogue. L'agent Auctioneer réalise cette action en utilisant la compétence advertise-catalog définie comme suit :

```
(defskill advertise-catalog AUCTIONEER
  :static-fcn advertise-catalog)
```

L'action advertise-catalog est réalisée par une fonction :

```
(defun advertise-catalog (agent env)
  (declare (ignore env))
  ;; envoi du message d'info en broadcast
  (send-inform agent :to 'ALL
      :args (list (cons 'CATALOG
                    (get-product-list
              (symbol-value agent)))))
  ;; fin de la compétence
  (static-exit agent :done))
```

send-inform et static-exit sont des fonctions de la bibliothèque SMAS. L'agent AUTIONEER utilisera un message d'information à nouveau pour prévenir l'agent qui aura gagné l'enchère. Les agents de type BIDDER devront donc avoir une compétence leur permettant de traiter les messages d'information.

Les compétences nécessaires pour la vente sont un peu plus compliquées et se décomposent pour l'agent AUCTIONEER en deux parties : une partie d'initialisation de l'échange et une partie qui sera activée lors des réponses des acheteurs. On notera également une fonction qui sera appelée en fin de vente (déclenchée lorsque la limite de temps sera atteinte) :

```
(defskill auction AUCTIONEER
  :static-fcn auction-start
  :dynamic-fcn auction-step
  :time-limit-fcn auction-time-limit)
```

Le début de l'enchère se fait de la façon suivante :

```
(defun auction-start (agent environment)
   ;; choix d'un produit tiré au sort
   ...
   ;; envoi d'un message FOR-SALE en broadcast
   ;; avec indication du prix désiré et
   ;; de la date de fin d'enchères.
   ...)
```

Sur un retour d'enchère le comportement est différent :

```
(defun auction-step (agent environment answer)
   ;; annulation de la tâche correpondant
   ;; au niveau d'enchère précédent
   ...
   ;; acceptation de l'offre si supérieure
   ;; à la précédente et enregistrement
   ...
   ;; envoi d'une nouvelle tâche (offre à battre)
   (send-subtask agent :to 'ALL
                 :action 'FOR-SALE ...)
   ...)
```

Sur le traitement de fin d'enchères le comportement est le suivant :

```
(defun auction-time-limit (agent env mess)
  ;; annuler la tâche en cours
  (cancel-all-subtasks agent)
  ;; vérifier si quelqu'un a enchéri
   ...
  ;; si oui l'informer qu'il a gagné
  (when buyer
    (send-inform agent :to buyer ...)
  ;; enregistrer le résultat
  ...
  ;; nettoyer les données temporaires
  (update-environment agent nil)
  ;; et terminer l'enchère pour ce produit
  (dynamic-exit agent ...))  )
```

Au niveau des agents acheteurs, la compétence est très simple :

```
(defskill FOR-SALE BUYER-1
  :static-fcn buyer-sale)
```

Les arguments de la fonction sont transmis par le système :

```
(defun buyer-sale
       (agent environment buyer product
        last-price-offered time-limit)
  ;; d'abord est-on intéressé à acheter
  ;;le produit ?
  (unless (member product products-to-buy)
      ;; si pas, on ne répond pas
      (return-from buyer-sale :abort))
  ;; sinon, vérification de l'auteur de
  ;; l'offre précédente ... si c'était
  ;; nous, on n'enchérit plus.
  (when (equal agent buyer)
      (return-from buyer-sale :abort))
  ;; sinon, on calcule le temps qu'il faut
  ;; pour réfléchir
   ...
  ;; si trop cher, on passe
   ...
  ;; sinon, on fait une offre
  (static-exit agent
          (list agent product bid))))
```

Pour pouvoir simuler il faut ensuite définir un scénario. La durée de la vente est fixée par le temps limite accordé à la tâche. La figure ci-dessous montre une trace d'exécution pour une seule vente de durée 30, correspondant au scénario suivant :

```
(defscenario SC-1
  (1 VENDOR :request ADVERTISE-CATALOG ())
  (3 VENDOR :request AUCTION () 30))
```

Cette figure (normalement en couleur) montre une trace d'exécution du simulateur pour une population de 6 agents acheteurs et un agent vendeur. Dans le scénario exécuté, l'agent vendeur vend 1 des 8 produits (tiré au hasard) et 4 des agents acheteurs se montrent intéressés. La vente se termine au bout d'un temps fixé à l'avance (30 unités de temps). La vente se déroule comme suit :

- Dans un premier temps l'agent vendeur informe (broadcast) tous les agents du système (qu'il ne connaît pas), qu'il a 8 produits à vendre. Chaque agent choisit 3 des produits du catalogue (au hasard).

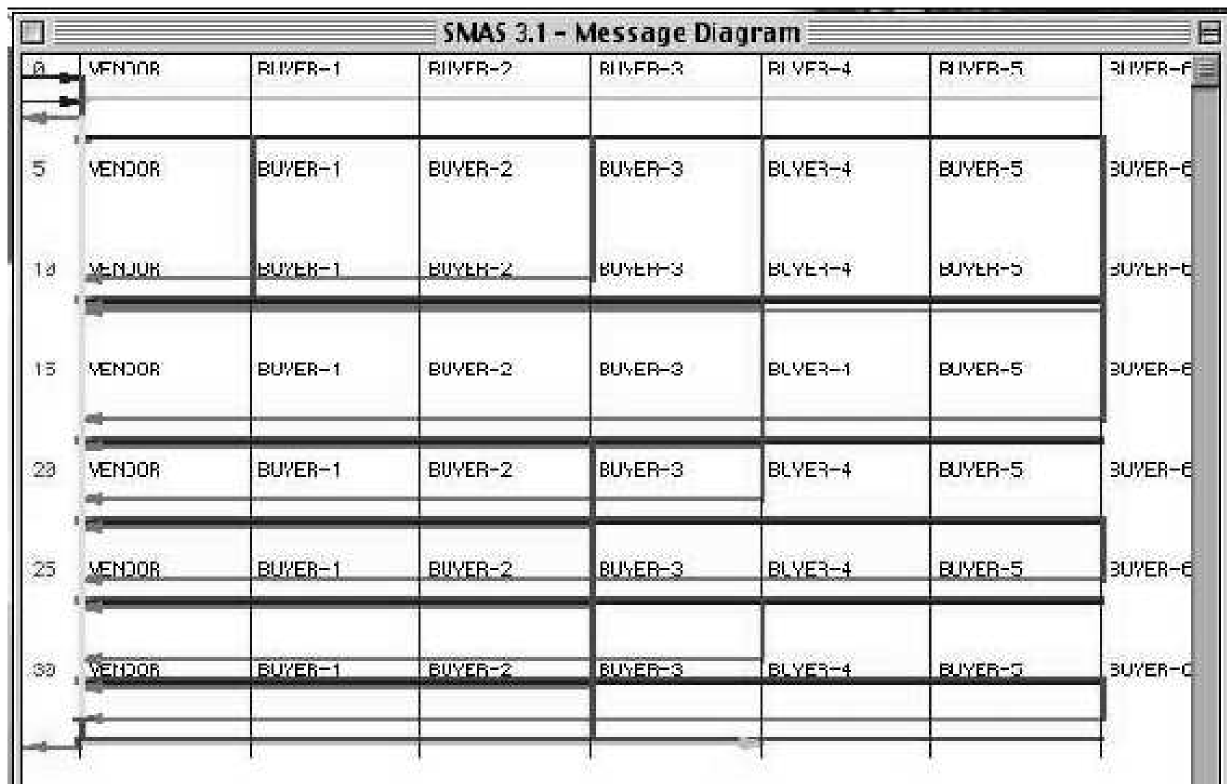

Fig. 3 – Réalisation avec SMAS

- Dans un deuxième temps l'agent vendeur envoie un appel d'offres (broadcast) sur l'un des produits tiré au hasard. Quatre agents sont intéressés. L'agent acheteur 3 fait l'offre la plus rapide. A réception de l'offre l'agent vendeur annule l'appel d'offres (broadcast rouge) et envoie un nouvel appel d'offres (broadcast bleu) avec le prix de référence et le nom de l'acheteur. Les autres acheteurs intéressés réfléchissent et l'acheteur 6 fait l'offre la plus rapide. Etc.
- La vente se termine lorsque la limite de temps interrompt le vendeur qui annule alors le dernier appel d'offres et indique à l'acheteur 4 qu'il a remporté les enchères (inform orange). On remarque sur la trace graphique que les agents deviennent de plus en plus fébriles à mesure que la fin de l'enchère approche, faisant des offres de plus en plus rapprochées. Ce comportement est déterminé au niveau de la compétence FOR-SALE de chaque agent acheteur.

3.3 Description générale de la plate-forme

SMAS est une plate-forme offrant de nombreuses possibilités pour le développement et la mise au point de systèmes multi-agents complexes. Le modèle d'agent de base est relativement complet, permettant d'implanter des mécanismes d'apprentissage, ou des modèles de type BDI, ou encore de tester des environnements où l'utilisateur fait partie du système. On peut également simuler des pannes en tuant certains agents puis en les faisant ressusciter, jouer sur les différentes constantes de temps du système (délais d'exécution des compétences, de temps de transmission, de durée globale de tâches, de timeouts, etc). Le développement continu de SMAS permet d'améliorer les mécanismes pour la mise au point qui dans le domaine SMA pose un véritable problème.

4 La Plateforme MASK

En cours de développement depuis 1993, la plateforme MASK (Multi-Agent System Kernel) de l'équipe MAGMA du Laboratoire LEIBNIZ de Grenoble est le support logiciel associé à la méthode Voyelle AEIO [Dem95].

4.1 Présentation Générale

L'objectif est de fournir des bibliothèques d'agents (A), de manipulation d'environnement (E), d'interaction (I) et d'organisation (O) ainsi que les outils d'aide à la programmation.

On dispose pour l'instant des modèles d'agents hybrides (ASTRO) et réactifs (PACORG) , des modèles d'interactions comme le langage à protocoles (IL) ou les forces (PACO), du modèle d'organisation statique (RESO).

La plate-forme MASK est composée de boîtes à outils couvrant les différents aspects du paradigme multi-agents, ces boîtes contiennent des outils de deux types : des bibliothèques de primitives et des éditeurs.

La boîte à outils Agent Elle fournit à l'utilisateur des éditeurs d'agents instances de modèles d'agents prédéfinis ou permet au concepteur de modèles d'ajouter de nouveaux modèles avec leurs éditeurs associés.

Pour les besoins du problème on utilisera le modèle d'agent ASTRO qui repose sur un tableau noir parallèle générique implanté en C++. L'éditeur interactif ASTRO de MASK permet de construire les noyaux des agents et d'intégrer les aspects externes par la construction des comportements et l'aide à l'écriture des modules.

La boîte à outils Environnement Elle fournit à l'utilisateur des bibliothèques de fonctions pour manipuler les environnements (simulés ou réels) où évoluent les agents. Le concepteur peut définir de nouveaux environnement et associer les bibliothèques correspondantes.

La boîte à outils Interaction Elle est chargée de la présentation des fonctions d'interaction. On peut utiliser les modèles IL/AGIP (avec les agents de types cognitif) et PACO (avec les agents réactifs). Pour le langage IL, on dispose des bibliothèques IL-API, un ensemble de primitives (C++) qui réalisent le langage d'interaction.

La boîte à outils Organisation La boite à outils Organisation se décompose comme la boite interaction en deux parties :

- un ensemble d'outils de modélisation de l'organisation fournissant des primitives d'exploitation,
- des éditeurs permettant d'instancier des organisations sur notre SMA.

On dispose du modèle "RESO" qui permet de représenter une organisation. Des relations peuvent être construites par une analyse des dépendances entre agent en terme de tâches, de position sociale ou d'influence. L'expression d'une organisation doit être externe. Les agents ont la capacité d'exploiter cette connaissance organisationnelle. La modélisation de l'organisation se fait naturellement avec des relations de subordination qui expriment la priorité d'un d'agent sur un autre. Ce modèle propose une interface graphique de construction de la représentation d'une organisation de SMA.

4.2 Résolution du problème

Nous présentons les principales étape de la construction du SMA de vente aux enchères selon la méthode présentée dans [OK00].

Phase de situation. Consiste à définir les limites de l'application, à caractériser les agents et l'environnement. Nous distinguerons un agent vendeur (AgV) et des agents acheteurs (AgA).

Phase Individuelle. La décomposition du processus de développement des agents est centrée sur la distinction entre les aspects externes et internes des agents. L'aspect externe concerne la définition des média mettant en rapport l'agent avec le monde extérieur, c'est-à-dire ce qu'il peut percevoir et comment il le perçoit, ce qu'il peut communiquer (le type des interactions), comment il peut les exploiter. L'aspect interne consiste à définir ce qui est propre à l'agent, c'est-à-dire ce qu'il sait faire (une liste d'actions) et ce qu'il connaît (sa représentation des A - E - I- 0). Les actions peuvent consister en des modifications de l'environnement, des initialisations d'interactions ou la mise en place de buts internes. Elles sont codées dans les modules du système blackboard. Elles peuvent éventuellement contenir des interactions avec l'utilisateur.

Voici les plans (séquences d'actions) obtenues ainsi que les buts auxquels ils répondent.

$$Lancer_Enchere \longrightarrow [Choisir_Article(),$$
$$Diffuser_Enchere(), Demarrer_Timer()]$$
$$Nouvelle_Offre \longrightarrow [Enregistrer_Offre(),$$
$$Diffuser_Offre(), Demarrer_Timer()]$$
$$Nouveau_Part. \longrightarrow [Envoi_Catalogue()]$$
$$Cloture_Offre \longrightarrow [Diffuser_Gagnant]$$

Dans la plupart des cas, les actions travaillent à partir des données disponibles dans la représentation de l'environnement détenue par l'agent. Il s'agit donc ensuite de spécifier cette représentation à partir des besoins exprimés lors de la spécification des actions.

Voici le détail des Connaissances d'un AgV, placées sur le blackboard :

Environnement	Catalogue [Prod., Prix, Vendu, AgA]
	Produit_courant [Produit]
	Offre_Courante [Produit, Prix,AgA]
Autres	Liste_AgA
	Protocoles_encours[AgA, NomProt, Etat]
Moi	Plans Disponibles
	...

Phase Sociale. Les interactions entre les agents sont réalisées par échange de messages. Les modes d'échanges sont formalisés de façon externe et commune aux agents à l' aide de protocoles d'interaction qui mettent en oeuvre des actes de langages (Langage IL).

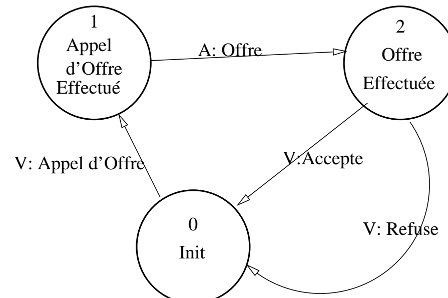

FIG. 4 – Le protocole de gestion d'une offre

Phase de socialisation des individus. A ce point, les influences possibles sur les plans initiaux doivent être analysées. Ces influences sont intégrées dans les agents par l'intermédiaire des capacités d'évaluation de la communication et de la perception. C'est par l'intégration des influences sociales dans les évaluateurs des agents que l'on va donner une dynamique au SMA, que lon gère la réactivité.

Voici le détail du module évaluateur de communication d'un AgA.

```
#include <stdio.h>
#include <errno.h>
#include "comm.h"
#include "EvalMessage.h"
long q_pid;

main(int argc, char *argv[])
{
  // Enregistrement du module
  // Système tableau noir
  initModuleBB(argv[1], argv[2], argv[3]);
  // Initialisation Communication IL
  ilapi::type_com modecom;
  // En réception
  modecom=ilapi::RECV;
  clio.connect("AgA",modecom);
  clio.pierror("Connect Status");
  ilapi::il_query mess;
  ....
  //Boucle d'attente de message
  while (true) {
     //Attente reception
     //Format IL: <communication><application>
     status=reception(...);
     ....
     //Test des champs communication
     if (strcmp(mess.from,''AgV'')==0)
     {
       {
         // Détermination du comportement
         Eval(applic[0],applic[1]);
       }
      };
```

```
    ......
  }
  // Déconnexion IL
  clio.disconnect();
  clio.pierror("Disconnect Status");

  //Info systeme BB
  envoyer(EMETTEUR,...,TERMINAISON,...);

}
```

L'évaluation peut créer de nouveaux buts. Le mécanisme d'exécution des tâches fourni par defaut est décrit dans [ODB98].

4.3 Conclusion

MASK a été conçue comme un outil support à la méthode "Voyelles-AEIO". Une des principales caractéristiques de MASK est d'adopter une approche multi-modèles. Nous restons persuadé que la multiplicité des problèmes et des domaines impose une telle approche alors que la plupart des plate-formes existantes sont construites autour de modèles uniques. Au niveau de l'outil lui-même, le principal avantage est la flexibilité de l'architecture elle-même qui garantit une évolution possible de l'outil. Au niveau de la plateforme d'exécution, MASK permet une distribution effective des agents sur un réseau de stations de travail. Comparé aux produits industriels MASK propose une interface homme-machine assez pauvre, mais une programmation visuelle modulaire comme dans ABE peut être envisagée afin de rendre l'outil plus attractif.

5 La plateforme DIMA

5.1 Description générale

DIMA [Gue00] [GB99] [GD96] est un environnement de développement de systèmes multi-agents dont le développement a débuté en 99 dans le thème Objets et Agents pour Systèmes d'Information et de Simulation (OASIS) du LIP. La première version de DIMA a été implémentée en Smalltalk-80 et a été ensuite portée en JAVA.

Le modèle d'architecture d'agents DIMA est fondé sur la conclusion suivante : les travaux sur les architectures d'agents ont engendré plusieurs résultats intéressants. Il est très difficile de comparer ces architectures dans le but de trouver la meilleure architecture. Chacune est bien appropriée à une catégorie d'agents. Une bonne plate-forme intègre les différentes architectures d'agents existantes ainsi que d'éventuelles nouvelles architectures. Ainsi, le modèle d'architecture d'agents proposé par DIMA peut être vu comme un modèle 'ouvert', une proposition d'agent minimal est faite permettant, par raffinements successifs, d'ajouter des fonctionnalités fournies par les différentes bibliothèques de la plate-forme. La force de DIMA réside donc à la fois dans sa proposition de modèle d'architecture d'agents modulaire, mais également dans les différentes bibliothèques disponibles. Chaque agent est un composant simple ou composant composite qui gère l'interaction de l'agent avec son environnement et qui représente son comportement interne.

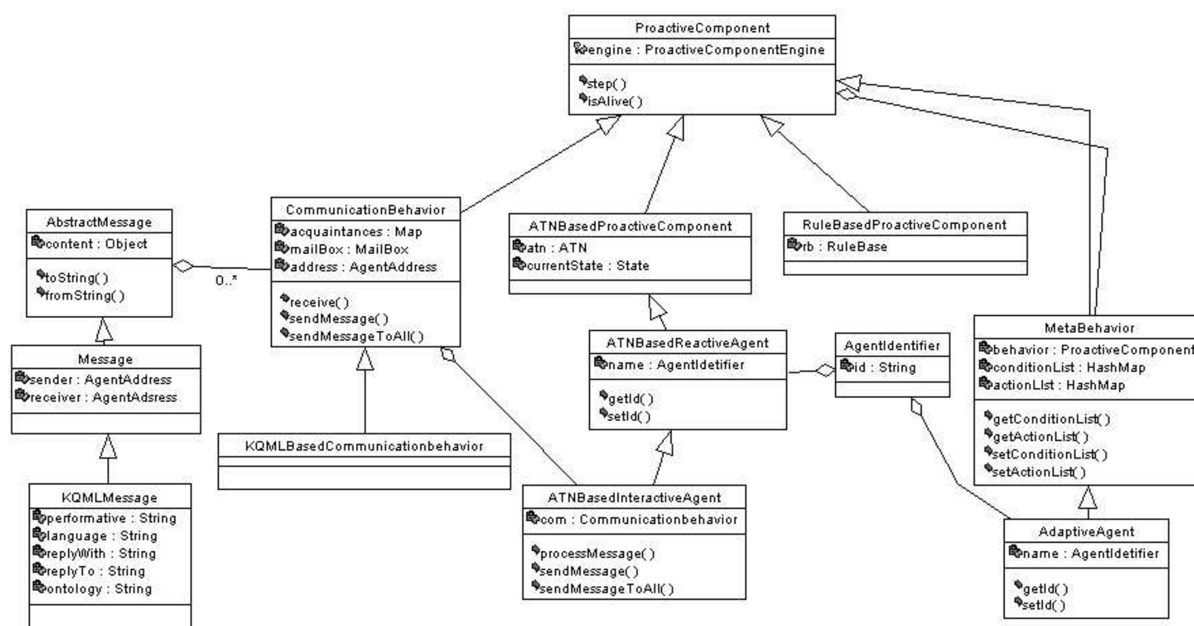

FIG. 5 – Exemple de bibliothèque d'agents DIMA

La brique de base de cette architecture est le composant proactif. Ce dernier est une entité autonome et proactive. Le noyau de base de DIMA est un framework minimal de

composants proactifs. Il est composé de plusieurs classes et de leurs méthodes. Ce noyau minimal est principalement composé de la classe ProactiveComponent (voir Figure 5). Une instance de cette classe décrit :) les compétences de base du composant proactif,) le but du composant proactif (implicitement ou explicitement décrit par la méthode isAlive()) et) le comportement (définit la manière dont les compétences sont sélectionnées, séquencées et activées en fonction du but du composant proactif).

Différents paradigmes (Automate, règles de production, etc.) sont réutilisés pour définir de nouvelles classes de composants proactifs. Ces dernières sont ensuite réutilisées pour définir des classes d'agents réactifs et des classes d'agents adaptatifs.

5.2 Résolution du problème

Dans DIMA, un SMA est défini par un ensemble d'agents et éventuellement un ensemble d'objets représentant leur environnement (Catalogue et Product dans le cas de la vente aux enchères).

Les principales étapes pour implémenter les agents sont :
- Détermination du type d'agents (réactif ou adaptatif),
- Choix des différents composants de l'agents (décision et communication pour l'exemple ventes aux enchères)
- Implémentation de ces composants en utilisant ou en sous-classant les classes de la librairie,
- Création et activation de l'agent.

Pour cet exemple,
- les agents vendeurs sont réactifs et les agents acheteurs peuvent être réactifs ou adaptatifs,
- le seul composant à écrire est la décision qui représente l'ensemble des compétences de base de l'agent. Le composant de communication est obtenu en instanciant une classe de la bibliothèque.

Les classes Auctionner et Bidder sont sous classes de AtnBasedIntertactiveAgent. Elles sont donc définies par un ensemble de compétences décrites par des méthodes (lancer une vente, mettre à jour le prix d'un produit, etc.) et un automate permettant de contrôler l'activation de ces différentes compétences.

```
public class Bidder
extends AtnBasedIntertactiveAgent{

private Map receivedProds;
private Map prodsInfo;
private Map selectedProds;

public Bidder();
public void addProduct(Product p);
public void buyProduct(Product p);
public boolean isActive();
```

```
public Boolean noProduct();
/* teste si la liste des produits proposés
est vide */
public Product[] selectProducts();
/* Sélectionne un des produits proposés */
public void subscribe();
/*s'inscrit dans les listes des différents
vendeurs */
 ...
}
```

```
public class Auctionner
extends AtnBasedIntertactiveAgent{

private Map Prods;
private Product selectedProd;

public Auctionner(AgentId id, Map Catalg
public void sendCatalog();
public Product chooseProd();
/* Sélectionne un des produits proposés */
 public boolean isActive();
/*test si tous les produits n'ont pas été
tous vendus*/
public Boolean CatalgueNotEm();
public void updatePrice();
...
 }
```

Un ATN décrit le comportement de l'agent (voir figure 6 qui décrit le comportement d'un vendeur).

L'interpréteur de cet automate est déclenché après l'activation de l'agent.

La création d'un agent consiste à instancier les classes Bidder et Auctioner. Elle est définie dans la classe AgentManager.

```
public static void main(String[] args) {
Catalog c;
/*initilisation du catalogue avec
10 produits */
c.init(0);
Bidder b1 = new Bidder();
Bidder b2 = new Bidder();
Auctioner a= new Auctioner();
b1.addAddress(a); b2.addAddress(a);
this.add(a); this.add(b1); this.add(b2);
this.startAll();
}
```

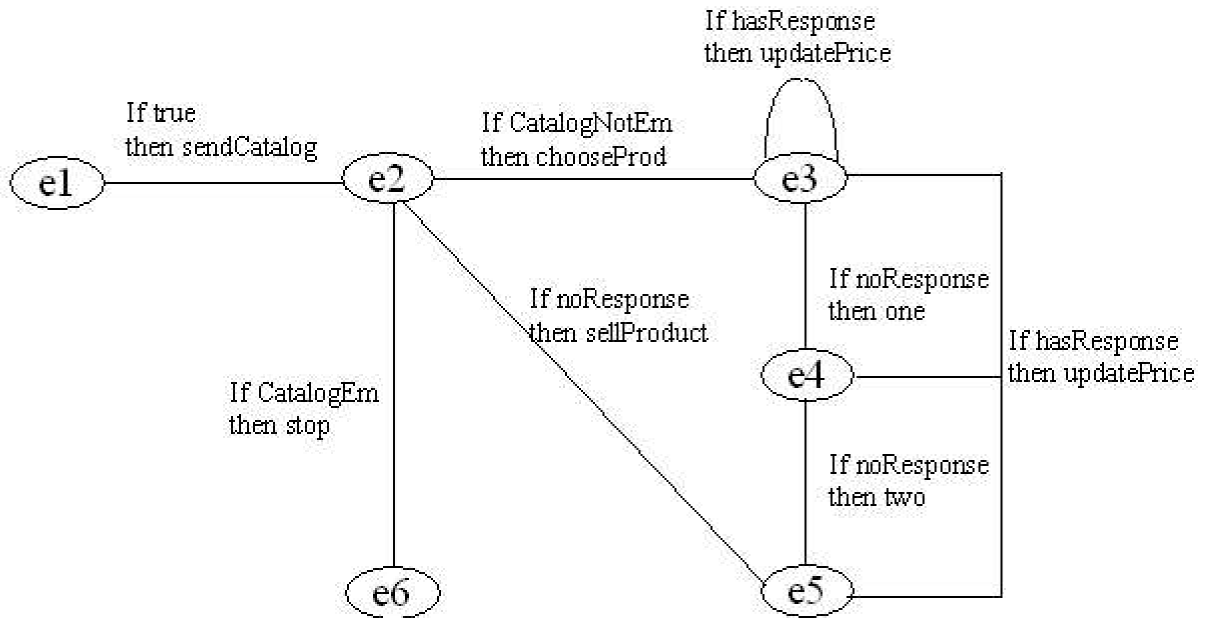

FIG. 6 – Exemple d'automate décrivant le comportement d'un vendeur

Le seul lien qui existe entre les agents est le réseau d'accointances.

5.3 Conclusion

DIMA est un environnement de développement multi-agents écrit en JAVA et en Smalltalk. La principale caractéristique de DIMA est son architecture modulaire et des librairies offrant les briques de base pour construire des modèles d'agents divers. Ces différentes briques ont pour but d'offrir à l'utilisateur une grande variété de paradigmes (par exemple automate, règle, etc.) d'une part, et d'autre part, une implémentation des différentes propositions conceptuelles introduites par la communauté multi-agents (BDI, KQML, ACL, etc.). Les librairies de DIMA regroupent des classes qui peuvent être réutilisées et/ou adaptées pour construire facilement des agents. Ces classes peuvent être instanciées ou sous-classées pour implémenter un agent ou un composant de l'agent. Par exemple, dans l'exemple décrit dans cet article, nous avons réutilisé le composant de communication et nous avons spécialisé le composant de décision.

La réutilisabilité offre les avantages suivants : réduire le temps d'implémentation, faciliter le développement pour des non-spécialistes des SMA, etc.

Plusieurs extensions de la version JAVA de DIMA sont en cours de réalisation : introduction de modèles organisationnels (statiques et dynamiques), tolérance aux pannes, etc.

6 La plateforme Geamas

6.1 Description générale

Geamas [Mar97, SMCC97] est un environnement de développement générique de systèmes multi-agents basé sur le langage Java et développé par l'équipe MAS2 (MultiAgent Systems, Modelling And Simulation)[3] de l'Université de la Réunion.

L'application *«Enchère»* est implantée comme une couche *Application* de *Geamas*, à partir de laquelle les simulations sont réalisées. Il s'agit donc d'une spécialisation du noyau de *Geamas* qui implémente le modèle générique d'agent proposant trois éléments de structuration fondamentaux : le *micro-agent*, le *medium-agent* et le *macro-agent* [Mar97]. Cette structure est complétée par le concept de *rôle* [VC98]. Les rôles peuvent être associés aux agents composant une application. A chaque rôle est associé une notion de *protocole* permettant aux agents de communiquer et de définir dynamiquement des comportements propres en fonction des messages reçus. L'environnement du SMA est quant à lui modèlisé par la notion de *GeamasObject* permettant la représentation d'objets participant à l'environnement spatio-temporel du système considéré.

La plate-Forme Geamas comprend un noyau agent minimal indépendant et fournit un ensemble d'outils complémentaires permettant de l'utiliser comme laboratoire virtuel de simulation. Ainsi, *Geamas* est accompagné d'un éditeur permettant la création et le contrôle de simulations multi-agents.

6.2 Résolution du problème

Afin de modéliser ce problème les trois couches d'agents de *Geamas* sont utilisées de la manière suivante : un *macro-agent* est créé pour représenter l'application de vente aux enchères. Il peut également imposer des contraintes à l'ensemble des agents du système. De plus, cet agent possède la maîtrise de l'ordonnancement des agents ainsi que la gestion du temps global. Dans le cas de cette application le *macro-agent «Enchère»* utilise un traitement par défaut en invoquant une primitive de macro-contôle de chaque agent du SMA.

Nous allons utiliser la notion de *rôle* de *Geamas* de la manière suivante : Un agent de commerce, désigné par *AgDeCommerce* (concrêtement une entreprise ou une personne) peut posséder le rôle de vendeur, qui propose un catalogue de produits, ou celui d'acheteur et peut ainsi acquérir une liste de biens. Ces agents sont définis comme des entités collectives ou *medium-agents* s'appuyant sur des *micro-agents* auxquels ils délèguent la gestion de chaque enchère ; achat ou vente d'un seul bien. C'est au niveau de cet agent de commerce que s'effectue la gestion des différentes enchères. Les propositions des vendeurs sont donc coordonnées à ce niveau, où s'effectue la stratégie d'achat en fonction de paramètres de vente, dont la somme d'argent disponible. Les agents *«AgDeCommerce»*, centraux dans la modélisation, sont ceux dont la fonction est définie par la composition des rôles qu'ils jouent dans la vente aux enchères : Vendeur de Produits (*VdP*), Acheteur de Produits (*AdP*). Un agent *AgDeCommerce* pourra jouer alternativement ou simultanément chacun de ces rôles (cf Fig. 4).

A ces entités viennent donc s'ajouter les agents du niveau le plus bas de la hiérarchie, les *micro-agents gestionAchat* et *gestionVente* capables de gérer une opération d'achat ou de vente

[3] `http ://www.univ-reunion.fr/~mas2`

sur un seul bien. Ces *micro-agents* sont alors subordonnés aux agents *AgDeCommerce* qui les ont créés.

Les biens à vendre ou à acheter qui n'ont aucune autonomie relative sont représentés par l'entité *«Produit»*, entités dérivant du type *GeamasObject*. Ils sont alors considérés comme des ressources pouvant être associées à des agents de type *AgDeCommerce*. Un tel agent pourra alors exercer les rôles *AdP* ou *VdP* en fonction de l'état des entités *GeamasObject* qu'il possède.

Agents	***Rôles***		***GeamasObjects***
« AgDeCommerce»	*VdP*	*AdP*	*Soit P ={Produits associés à l'agent}*
UnVendeur	■		$\exists\ p_i \in P$ tel que Etat p_i = "vendre"
UnAcheteur		■	$\exists\ p_i \in P$ tel que Etat p_i = "acheter"
UnAcheteurEtVendeur	■	■	$\exists\ p_i, p_j \in P$ tel que Etat p_i = "vendre" et Etat p_j = "acheter"
UnSpectateur			Nul

FIG. 7 – Correspondance entre agents, rôles et GeamasObjects

Les agents de commerce rassemblent des agents de gestion d'une vente ou d'un achat. Ils sont susceptibles d'imposer certaines contraintes à leurs membres (par exemple : ne pas enchérir au-delà d'une certaine somme). Un tel *AgDeCommerce* est associé à ces agents de gestion d'une vente ou d'un achat par des relations d'*accointance* définissant des réseaux d'échanges privilégiés. Un exemple simple de vue organisationelle et structurelle du modèle *Enchère* est donné figure 5.

Les mécanismes d'interactions entre agents : Messages et Protocoles

Tout agent est doté d'un *protocole* lui permettant d'émettre et de percevoir des messages prédéfinis. Certains protocoles sont définis par rapport à des types d'agents, ces protocoles seront alors reconnus par de tels agents pendant tout leur cycle de vie, d'autres protocoles sont définis dans le cadre des rôles tenus par les agents à un instant donné au sein du système. Ainsi, un même agent peut voir ces possibilités d'interactions évoluer dynamiquement en fonction des différents rôles qu'il tient au cours du temps. Cette propriété est utilisée dans l'application *«Enchère»* de façon à gérer les possibilités d'échanges d'un agent de commerce au travers des rôles Acheteur de Produits (*AdP*) et Vendeur de Produits (*VdP*). Trois protocoles sont définis dans l'application *«Enchère»* (cf Fig. 6). Ces protocoles définissent les possibilités de liens entre agents. Ainsi, deux agents qui n'ont pas de protocoles applicables entre eux ne peuvent entretenir une relation d'accointance. La modélisation que nous avons retenue ne met jamais directement en relation les agents de commerce entre eux, les interactions sont établies au niveau des *micro-agents* subordonnés de gestion d'achat et de vente au travers du protocole P1. Les deux protocoles P2 et P3 permettent par ailleurs aux *medium-agents* d'imposer leurs contraintes aux *micro-agents* membre du groupe (par exemple : déterminer la durée maximum allouée à l'enchère d'un produit).

Chaque agent possède un niveau d'autonomie qui lui permet de paramétrer dynamiquement les comportements internes qu'il souhaite associer à tout message du protocole. Par exemple le code ci-dessous illustre comment un agent de gestion d'achat peut modifier sa façon de réagir au message *"annonce prix"* du protocole *P1* afin que son agent de commerce lui donne des directives.

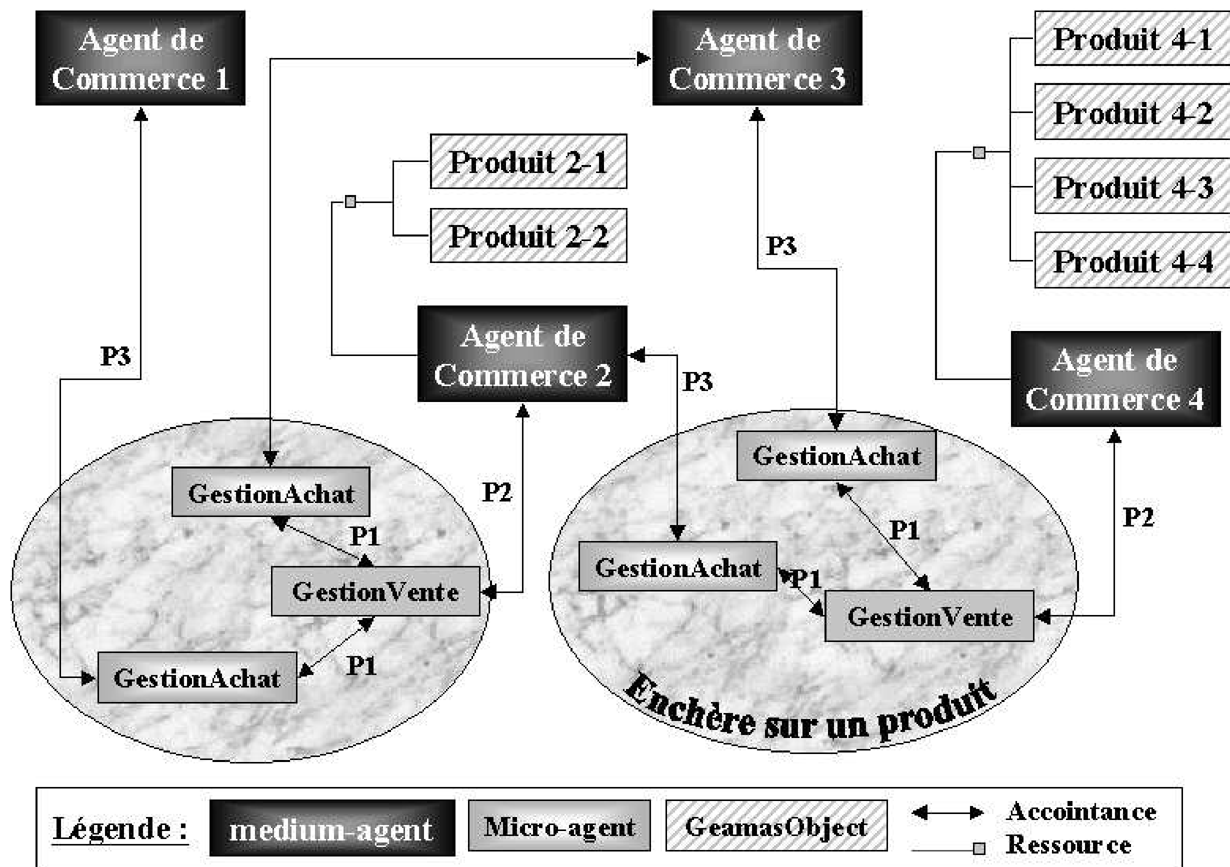

Fig. 8 – Modèle structurel de l'application

Protocole P1 : gestionVente ↔ gestionAchat	**Protocole P2 :** Rôle VdP ↔ Agent gestionVente	**Protocole P3 :** Rôle AdP ↔ Agent gestionAchat
<annonce prix> <accepte> <refuse> <offre prix >	<DuréeMax durée> <initInforme> <stopInforme> <informe prix>	<SeuilMax prix> <FixePrix prix> <Abandon> <initInforme> <stopInforme> <informe prix>

Fig. 9 – Messages échangés entre agents en fonction de leur type et de leur rôle

```
public class GestionAchat extends MicroAgent
{
 ...
 // comportement par défaut invoqué à la
 // réception du message "annonce" de P1.
 public void traiteAnnonceDePrix()
 {
  // Tant que le prix est sous un certain
  // seuil, l'argent de gestion d'achat
  // gère par lui-meme le comportement,
  // sinon il informe son agent de commerce
  if (condition de dépassement de seuil) {
   // informe l'agent de commerce
   Message m = new Message("informe", prix);
   sendMessage(monAgDeCommerce, m);

   // modifie le protocole pour que la mé-
   // thode toBoss soit dorénavant invoquée.
   setProtocole("annonce", toBoss());
  }
  else {...}
 }
}
```

6.3 Conclusion

Au travers de cette application, nous avons succinctement introduit les principaux concepts de modélisation et de développement d'une application avec *Geamas* : structuration en 3 niveaux d'agents, *Environnement* et *GeamasObject*, les rôles, les messages et les protocoles d'interactions dynamiques. D'autres aspects touchant à la représentation de l'environnement, l'auto-adaptation individuelle et collective sont en cours d'étude. *Geamas* est développé en Java et supporte actuellement plusieurs applications opérationnelles issues de collaborations avec des organismes de recherche nationaux (CIRAD, IFREMER).

7 L'environnement oRis

7.1 Description générale

oRis [4] est un langage de programmation conçu pour le développement d'applications composées d'entités autonomes telles que des agents. Il a été développé par Fabrice Harrouet [Har00] au Laboratoire d'Informatique Industrielle de l'ENIB.

oRis [5] est un langage à objets actifs à typage fort dont le vocabulaire et la grammaire sont volontairement très proches de ceux de C++ ou Java. À la différence de ces derniers, les propriétés des objets peuvent être définies avec une granularité instance. Le langage est interprété dynamiquement et *oRis* fournit aussi un environnement de développement et d'exécution. Ces deux mondes ne font d'ailleurs qu'un puisque *oRis* a été conçu comme un outil de modélisation en ligne : de nouvelles portions de code peuvent être introduites lors d'une simulation. *oRis* permet de manipuler des objets classiques et des objets actifs ayant leurs propres flots d'exécution ; ces objets peuvent communiquer par appel de méthodes ou par envoi de messages.

oRis est une plate-forme généraliste, donc aucune architecture d'agent n'est imposée, destiné à la réalisation de systèmes à agents. En ce sens, il utilise la programmation par objets actifs, la possibilité de s'interfacer avec d'autres langages (C++, Java, Prolog), et offre un contrôle fin de l'ordonnancement, des mécanismes de communication par réseau, des possibilités d'introspection du code...

7.2 Résolution du problème

La solution proposée ici a une vocation pédagogique et ne prétend donc pas constituer une solution optimale[6].

Les classes d'agents et d'objets. On trouve les classes d'agent Auctioneer et Buyer, une classe d'agent Timer pour la gestion du temps, la classe Product pour le représentation des objets à vendre et à acheter et les classes de messages échangés (elles dérivent toutes de AuctionOntologyMsg). L'interface de la classe Auctioneer – qui dérive de BAgent (*Basic Agent*) – est doonée en exemple ci-dessous : :

```
class Auctioneer: BAgent {
 Product[] _catalog;    // Produits en vente
 Timer _timer;          // Agent chronometre
 void new(void);        // Comportement initial
 void delete(void);
 // Traitement messsages
 void performMessage(Message msg);
 int getTime(void);     // Acces au temps
 // Competences ‘‘vendeur’’
 void startAuction(void);
 void endAuction(void);
 void evaluateBid(BidMsg bid);
}
```

[4] Un manuel de référence, des transparents de présentation et des exemples sont disponibles sur la page proposant le téléchargement d'*oRis* : `http://www.enib.fr/~harrouet/oris.html`

[5] Le suffixe latin *oris* signifie *"celui qui agit"* ; exemple : *cantoris* → chanteur, *"celui qui chante"*.

[6] L'intégralité du code peut être obtenue à l'adresse `http://www.enib.fr/~chevaill/auction.html`

La communication entre agents : Auctioneer et Buyer communiquent selon le mode défini par la classe BAgent. Le traitement des messages se fait de manière asynchrone dans un flot d'exécution qui lui est dédié ; ces classes doivent redéfinir les traitements associés aux messages (performMessage()). L'agent chronomètre Timer utilise un des mécanismes de base d'*oRis* : l'arrivée d'un message déclenche la méthode onMessage(). Il y a donc indépendance entre l'émisssion et le traitement des messages. L'appel à : :yield() assure un fonctionnement similaire quelque soit le mode de multi-tâches *oRis* choisi (coopératif, préemptif ou changement de contexte tous les n micro-instructions).

```
void BAgent::new(void) {
 start { while (true) {
          readMessage(); ::yield(); }
 }
}
void BAgent::performMessage(Message msg) {
 // A definir dans les classes derivees...
}
void BAgent::readMessage(void) {
 Message msg = getNextMessage();
 if (msg != (Message)NONE) performMessage(msg);
}
```

Lors des enchères, les agents communiquent en point à point (1 Buyer $\mapsto$ 1 Auctioneer) et par diffusion (1 Auctioneer $\mapsto$ * Buyer : le vendeur ne connaît pas les acheteurs). Afin que les Buyer reçoivent les messages relatifs aux enchères, ils sont déclarés être sensibles à leur classe de base (setSensitivity()) et les traiter en mode asynchrone :

```
void Buyer::new(void) { //...
 setSensitivity("AuctionOntologyMsg",
                "asynchronousReception");
}
```

Le mode de gestion du temps. Chaque Auctioneer utilise un chronomètre (agent Timer) servant à décider quand le temps imparti à une surenchère est écoulé. Le temps (ici un temps simulé) est donné par getTime(), déclarée dans BAgent et redéfinie ici. Le comportement du chronomètre consiste à décrémenter le délai d'expiration. Le test à zéro est assuré par un lien réflexe comme cela est présenté plus loin (une solution plus simple existe évidemment !).

```
int Auctioneer::getTime(void) {
  return ::getClock(); }
void Timer::main(void) { _deadline--; }
```

Les protocoles d'interactions. Il convient de spécifier les actions que les agents effectuent pour chaque classe de messages et les messages qu'ils renvoient. Le démarrage d'une vente consiste à déclencher le chronomètre (envoi en point à point d'un message InitTimerMsg au _timer) et à diffuser un message PutOnSaleMsg.

```
void Auctioneer::performMessage(Message msg) {
 if (msg->isA("AuctionSubscribeMsg"))
  new AuctionCatalogMsg(_catalog)
   ->sendTo(msg->getEmitter());
 else if (msg->isA("BidMsg"))
  evaluateBid((BidMsg)msg);
 else if (msg->isA("ExpiredTimeMsg")) {
  endAuction(); startAuction(); }
 BAgent::performMessage(msg);
}
void Auctioneer::startAuction(void) {
 if (_catalog) {
  int duration = 1000 + ?(1000);
  new InitTimerMsg(duration)->sendTo(_timer);
  new PutOnSaleMsg(_catalog[0])->broadcast();
 }
}
```

Le comportement autonomes des agents est supporté par différents flots d'exécution : traitement des messages, méthode main() et flots dédiés à des comportements spécifiques.

À tout moment, un agent peut redéfinir son comportement (Cf. Timer : :new() qui fait appel à différentes fontionnalités d'*oRis* telles que le contrôle du contexte d'exécution). Le mot-clé that indique à quel objet appartient le contexte appelant, un agent peut ainsi adapter son comportement en fonction de cette information. Un agent peut suspendre ses lots d'exécution et les relancer (suspend() / resume()).

Cet exemple montre la possibilité de modifier dynamiquement le comportement d'un agent : Timer : :new() définit ici un lien réflexe (AttrCB) sur son attribut _deadline et le comportement associé à sa méthode after() ; ceci illustre la granularité instance des propriétés des objets *oRis*. Ici, le lien réflexe permet de déclencher un traitement lorsque l'attribut _deadline change.

```
void Timer::new(void) {
 _deadline = 0;
 _client = (Agent)that; // createur de l'agent
 AttrCB reflex = NEW AttrCB(this, "_deadline");
 string  newBehavior =
   format("void ", reflex, "::after(void) {
    if (", this, "->_deadline == 0) {
     new ExpiredTimeMsg()->sendTo(",
     this, "->_client);", this, "->suspend();
    } }");
 parse(newBehavior); // modif. en ligne
 suspend();
}
void Timer::onMessage(void) {
 Message msg = getNextMessage();
 if (msg->isA("InitTimerMsg")
     && (Agent)msg->getEmitter() == _client) {
  _deadline = ((InitTimerMsg)msg)->_duration;
  resume();}
}
```

L'instanciation des agents consiste à lancer la simulation, ce qui passe par la définition du mode d'ordonnancement. *oRis* étant dynamiquement interprété, il est possible d'instancier de nouveaux agents ou de modifier leur comportement à tout instant. Le code suivant peut donc être introduit plusieurs fois au cours d'une même session, ce qui permet à plusieurs ventes d'avoir lieu parallèlement et à un agent humain d'y participer par le biais d'un avatar.

```
include "Auction.pkg"
execute {
 NEW Auctioneer;  // Agent persistant
 for (int i = 0; i < 10; i++) NEW Buyer;
}
execute { Agent humanAvatar = NEW Agent;
 humanAvatar->setSensitivity("AuctionOntologyMsg",
                "asynchronousReception");
Agent.1::execute {
 new AuctionSubscribeMsg()->sendTo(Auctioneer.1);
}
```

7.3 Conclusion

La solution présentée n'utilise que des fonctionnalités de base d'*oRis*. Cependant, de nombreux packages permettent de concevoir des comportements d'agents plus évolués. Les packages object2d.pkg et object3d.pkg permettent de définir des agents situés et mobiles dans un environnement en deux ou trois dimensions et de le percevoir. Des packages ont été développés pour la distribution des agents, et l'utilisation de KQML, de contrôleurs flous, de cartes cognitives flous, de la logique temporelle de Allen [dLC00]... Enfin *oRis* est intégré à la plate-forme de réalité virtuelle *ARéVi* [HRT99].

8 La plate-forme MadKit

8.1 Description générale

MadKit[7] est une plate-forme générique d'exécution de systèmes multi-agents, conçue pour fédérer différents modèles. Les services du système (communication distribuée, surveillance, mobilité) et interfaces graphiques sont découplés et agentifiés pour une flexibilité maximale. Cette infrastructure est conçue pour supporter des architectures d'agent hétérogènes (agents cognitifs ou réactifs), des modèles de communications distincts (types de messages et protocoles d'interaction) et faire fonctionner simultanément plusieurs applications [FG98].

8.2 Architecture

L'architecture de la plate-forme est basée sur un noyau minimal qui n'assure que les fonctions primitives de communication locale, gestion du cycle de vie d'un agent, et maintien des groupes et rôles locaux. Les services de plus haut niveau (communication distante, gestion des groupes distribués, migration des agents, visualisation et contrôle de la plate-forme, etc.) sont fournis sous la forme de meta-agents : il est possible d'ajouter et/ou de modifier ces services sans avoir à intervenir sur la plate-forme elle-même. Par exemple, il est possible de passer d'une communication "socket" à une communication "CORBA" par un simple remplacement d'un agent. L'instrumentation d'un SMA peut également se faire par cette voie.

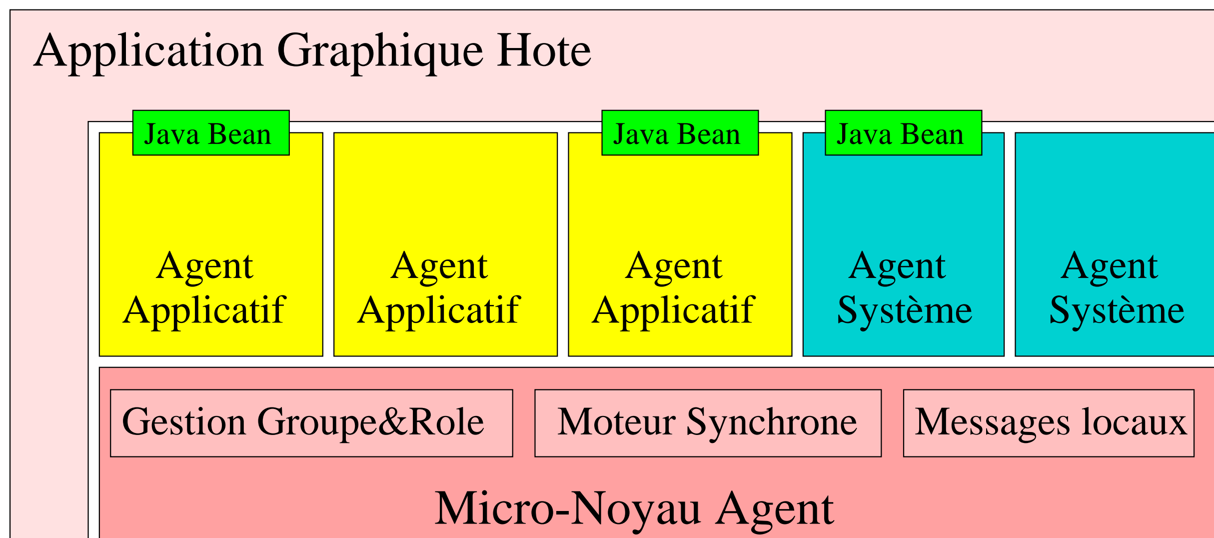

Fig. 10 – Architecture de la plate-forme

MadKit n'impose aucune architecture spécifique aux agents : le modèle des agents est volontairement resté simple et générique pour faciliter l'intégration de différents modèles d'agents. Plusieurs bibliothèques d'agents ont donc été ainsi développées à partir de cette architecture :

[7] http ://www.madkit.org

frameworks objet (Java), Scheme, comportement représenté en règles CLIPS, modèles à base de tâches, systèmes d'agents réactifs. Un trait néanmoins partagé par l'ensemble des agents de la plate-forme est un modèle opérationnel [FG99] de structures sociales qui permet d'exprimer les rôles des agents et leurs groupes.

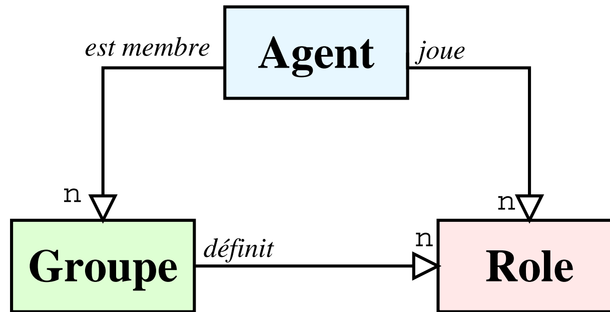

Fig. 11 – Modèle agent-groupe-rôle

Ce modèle est une définition minimale de structures sociales, à base de groupes (définis comme le lieu d'action des agents) et de rôles (permettant identification fonctionnelle et expression des accointances).

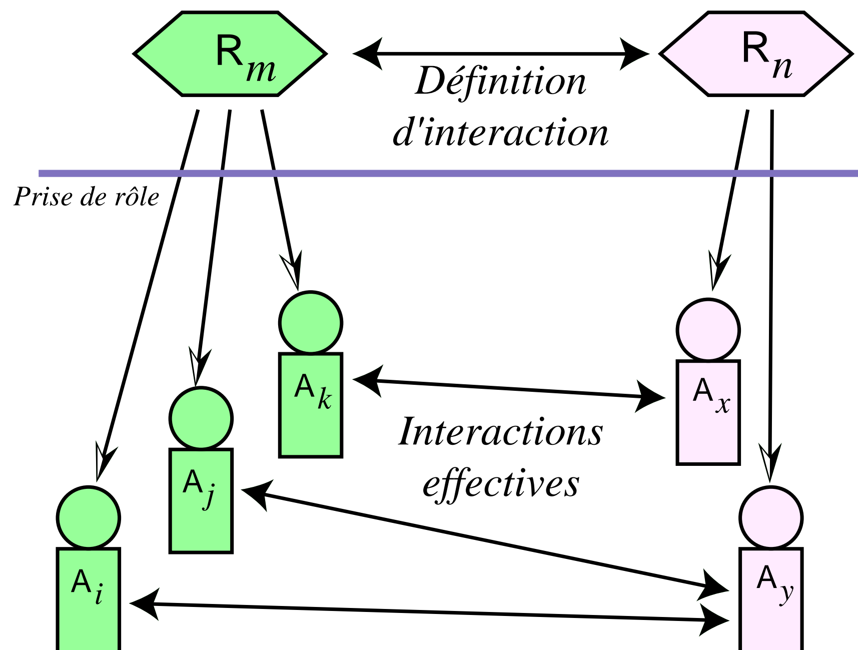

Fig. 12 – Relation rôles / agents

8.3 Construction des agents

Construire un système dans MadKit conjugue deux choix importants : la définition du modèle d'interaction (groupes, rôles, protocoles), et le choix d'une architecture particulière pour l'agent en fonction du domaine applicatif.

Dans le cas présent, nous avons choisi de construire l'application dans un seul groupe représentant la salle d'enchère et d'identifier trois rôles : fournisseur de *catalogue*, *acheteur* et *commissaire-priseur*, qui vont être utilisé pour définir les diagrammes d'interactions, de rôle à rôle.

Pour le premier agent, on utilisera simplement le modèle de base de MadKit, qui correspond à un développement sous forme d'objet Java. On choisit ici de faire jouer au même agent concret le rôle de fournisseur de catalogue et de commissaire-priseur.

```
public class MonVendeur extends Agent
{ ...
 public activate()
 {
   joinGroup("salle");
   requestRole("salle","catalogue");
   broadcastMessage("salle","acheteurs",
              new InformMessage(catalog));
   requestRole("salle","commissaire-priseur");
 }
  public void live() {
    Message m = waitNextMessage();
    processAuction(m);
 }
```

Ce très bref extrait montre d'une part l'action sur la structure sociale (à l'initiative de l'agent), et la communication qui peut être directement rattachée à l'expression en groupe et rôle, et donc très proche du schéma d'interaction.

Pour mettre en évidence la versatilité de la plate-forme, nous avons choisi de développer chaque agent dans un modèle de base différent. L'agent acheteur a été mis en place dans le modèle d'agent Scheme de MadKit, qui fait la liaison entre l'infrastructure et un interprète Scheme :

```
(define (answer-auction produit prix)
  .. on se détermine sur ce produit
 (if (> prix seuil)
  (send-message
    (get-agent-with-role "salle" "vendeur")
    (new-message 'refusal))) ...
```

La structure sociale et les mécanismes de messages étant définis dans l'infrastructure, ils sont facilement rendu visibles dans n'importe quel modèle d'agent, quel que soit sa structure interne ou même son langage d'implémentation.

Remarquons également que l'extension à plusieurs ventes simultanées serait triviale dans MadKit. En effet, on est alors dans le cas où plusieurs activités simultanées se déroulent et doivent être cloisonnées, tout en observant les mêmes protocoles d'interaction et les mêmes jeux entre rôles. Cela correspond exactement à la définition de groupe : un même agent pouvant faire partie de plusieurs groupes simultanés. Cela se fait alors avec le même rôle ou non : un vendeur dans une situation peut très bien être dans le rôle d'acheteur pour une autre.

De même, l'introduction d'agents humains se fait très facilement. On construit un agent "représentant" que l'on place dans les mêmes groupes et rôles, mais dont l'architecture n'est qu'une simple liaison vers l'interface homme-machine de son utilisateur. L'important est là encore que l'interaction (les messages et interactions) et la structuration applicative (les groupes et rôles) soient respectés, le modèle interne exact important alors peu.

8.4 Conclusion et évolutions

MadKit est utilisé actuellement dans des plusieurs applications allant de la simulation de robotique collective à la gestion de mémoire d'entreprise en passant par la surveillance de réseaux ou le développement de jeux, et ce dans des contextes universitaires ou industriels. De plus, sa modularité et la taille réduite de son noyau lui permet d'être déployé à différentes échelles (par exemple, dans des assistants numériques de type Palm).

Une évolution de la plate-forme [GF99] est l'ajout d'un outil de conception graphique : SEdit, un éditeur générique de "modèles". Chaque modèle étant une description d'un formalisme à base de graphe qui permettra de définir un éditeur spécialisé. L'intérêt vis à vis de la plate-forme et des questions de conception est que chaque éditeur est en fait un agent MadKit à part entière.

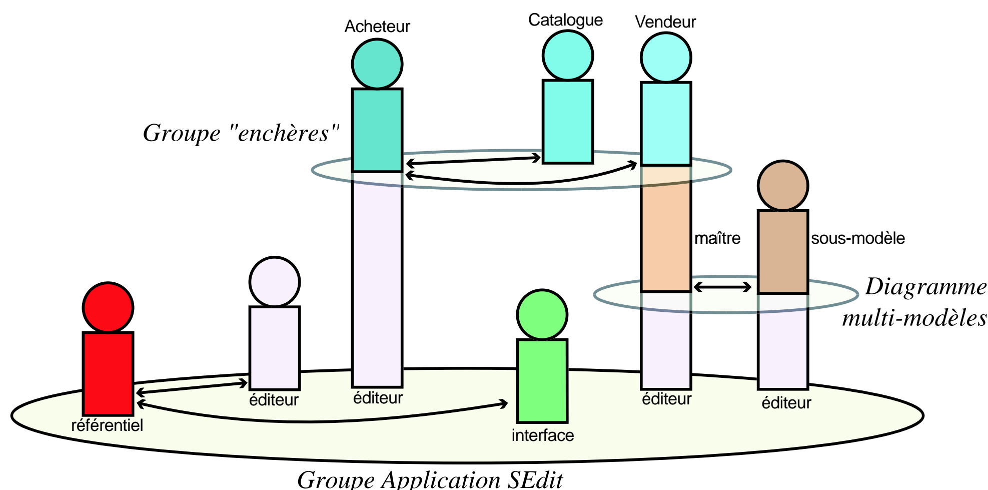

Fig. 13 – Intégration SEdit/MadKit/SMA

Certains formalismes associés (Pétri, règles d'actions) permettent alors de définir des schémas d'organisations ou des comportements d'agents et de les simuler. Plutôt que de générer le code des agents et les réintégrer dans MadKit, on peut tester les agents *en place*, via certains éléments du formalisme capturant les messages arrivant à l'éditeur et les réintégrant, par exemple sous

forme de jetons. L'agent SEdit joue alors à la fois le rôle d'éditeur dans le SMA SEdit et le rôle de l'agent applicatif en construction plongé dans son groupe final, sans que ses accointances ne s'aperçoivent qu'il s'agit d'un agent en édition.

9 Conclusion

La vente aux enchères que nous avons décrite dans cet article présente de nombreux avantages pour la comparaison de plateformes. Elle nécessite la mise en place de plusieurs mécanismes dont les agents ont fréquemment besoin comme l'autonomie, la distribution, ou le raisonnement propre à chacun. Ce problème est donc bien adapté à la comparaison de plateformes que nous venons d'effectuer. Chacun peut maintenant, après avoir vu ces plates-formes à l'œuvre, se faire une idée plus précise des avantages et inconvénients de chacune, ainsi que leurs facilités de mise en œuvre.

Références

[BM95] Nourredine Bensaid and Philippe Mathieu. Un modèle d'architecture multi-agents entièrement écrit en Prolog. In *IV Journées Francophones de Programmation Logique, JFPL'95*, pages 381–385, Dijon-France, 1995. teknea, Toulouse-France.

[BM97] Nourredine Bensaid and Philippe Mathieu. A Hybrid and Hierarchical Multi-Agent Architecture Model. In *Proceedings of the Second International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology, PAAM97*, pages 145–155, London-United-Kingdom, 21st-23rd April 1997. The Practical Application Company Ltd.

[BS97] J-P Barthès and E.E. Scalabrin. An environment for building cognitive agents for cooperative work. In P. Siriruchapatong, Z. lin, and J-P. Barthès, editors, *CSCW'97. Bangkok. Nov 26-28. Thailande. Proceedings of the Second International Workshop on CSCW in Design.*, 1997. ISBN 7-80003-412-7/TP-19.

[CMC98] R. Courdier, P. Marcenac, and S. Calderoni. Zooming on a multiagent simulation system : from the conceptual architecture to the interaction protocol (poster). In *Proceedings of 3rd International Conference on Multi-Agent Systems (ICMAS 98)*, pages 411–412. IEEE Computer Society Press, 1998.

[Dem95] Y. Demazeau. From cognitive interactions to collective behaviour in agent-based systems. In *Proceedings of 1st European Conference on Cognitive Science*, Saint Malo, France, april 1995.

[dLC00] P. de Loor and P. Chevaillier. Generation of agent interactions from temporal logic specifications. In M. Deville and R. Owens, editors, *16th IMACS World Congress 2000*, Lausanne, Suisse, 21-25 August 2000.

[FG98] Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Third International Conference on Multi-Agent Systems (ICMAS '98) Proceedings*, pages 128–135. IEEE Computer Society, 1998.

[FG99] Jacques Ferber and Olivier Gutknecht. Operational semantics of multi-agent organizations. In *Workshop on Agent Theories, Architectures and Languages (ATAL'99)*, 1999.

[GB99] Z. Guessoum and J.-P. Briot. From active objects to autonomous agents. *IEEE Concurrency*, 7(3) :68–76, 1999.

[GD96] Z. Guessoum and M. Dojat. A real-time agent model in an asynchronous object environment. In W. Van de Velde and J. Perram, editors, *Agent Breaking Away*, number 1038 in LNAI, pages 190–203, Eindhoven, The Netherlands, 1996. Springer Verlag.

[GF99] Olivier Gutknecht and Jacques Ferber. Vers une méthodologie organisationnelle pour les systémes multi-agents. In Marie-Pierre Gleizes, editor, *Actes des Journèes Francophones en Intelligence Artificielle Distribuèe et Systémes Multi-Agents 1998*. Hermés, Nov 1999.

[Gue00] Z. Guessoum. A multi-agent simulation framework. *Transactions of Computer Simulation*, 17(1) :2–11, 2000.

[Har00] F. Harrouet. *oRis : s'immerger par le langage pour le prototypage d'univers virtuels à base d'entités autonomes*. thèse de doctorat, Université de Bretagne Occidentale, 8 December 2000.

[HRT99] F. Harrouet, P. Reignier, and J. Tisseau. Multi-agent systems and virtual reality for interactive prototyping. In M. Torres, B. Sanchez, J. Corchado, and D. Andina, editors, *Virtual Engineering and Emergent Computing*, volume 3 of *Proceeedings of the 3rd World Multiconference on Systemics, Cybernetics and Informatics (SCI'99)*, pages 50–57, Orlando, USA, August 1999.

[KFD98] Jean-Luc Koning, Guillaume François, and Yves Demazeau. An approach for designing negotiation protocols in a multiagent system. In José Cuena, editor, *15th IFIP World Computer Congress, IT&KNOWS Conference*, Vienna (Austria), Budapest (Hungaria), September 1998. Austrian Computer Society.

[Mar97] P. Marcenac. Modélisation de systèmes complexes par agents. In *Technique et Science Informatiques, vol. 16, n° 8*, pages 1013–1037. Hermes, Paris, 1997.

[MRS00] P. Mathieu, JC Routier, and Y Secq. Dynamic skills learning. Technical report, LIFL, 2000.

[MT00] P. Mathieu and A Taquet. Une forme de négociation pour les Systèmes Multi-Agents. In *Journées francophones IAD-SMA*, St Jean La vêtre, France, Novembre 2000. Hermes.

[OD96] M. Occello and Y Demazeau. Une approche du temps réel dans la conception d'agents. In *4 èmes Journées Francophones IAD-SMA*, Port Camargue, France, Avril 1996. Hermès.

[OD97] M. Occello and Y. Demazeau. Vers une approche de description et de conception récursive en univers multi-agents. In *Journées francophones IAD-SMA*, La Colle sur Loup, France, Avril 1997. HERMES.

[OD98] M. Occello and Y. Demazeau. Modelling decision making systems using agents satisfying real time constraints. In *3rd IFAC Symposium on Intelligent Autonomous Vehicles*, volume 1, pages 51–56, Madrid, Spain, march 1998.

[ODB98] M. Occello, Y. Demazeau, and C. Baeijs. Designing Organized Agents for Cooperation in a Real Time Context. In , editor, *Collective Robotics*, volume LNAI 1456, pages 25–37. Springer-Verlag, 1998.

[OK00] M. Occello and J.L. Koning. Multi-agent based software engineering : an approach based on model and software reuse. In *EMCSR 2000 Symposium, "From Agent Theory to Agent Implementation 2"*, pages 645–653, Vienna, april 2000.

[Sca96] E.E Scalabrin. *Conception et réalisation d'environnment de développement de systèmes d'agents cognitifs.* PhD thesis, Université de Technologie de Compiègne. Spécialité Contrôle des Systèmes., 1996.

[SMCC97] J.-C. Soulié, P. Marcenac, S. Calderoni, and R. Courdier. Geamas v2.0 : An object-oriented platform for complex systems simulations. In *Proc. 26th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 230–242. IEEE Computer Society Press, 1997.

[SVdAB96] E.E. Scalabrin, L. Vandenberghe, H. de Azevedo, and J-P. A. Barthès. *Advances in Artificial Intelligence. Series Lecture Notes in Artificial Intelligence.*, chapter A Generic Model of Cognitive Agent to Develop Open Systems., pages 61–70. Springer-Verlag, 1996. ISBN 3-540-61859-7.

[VC98] J.D. Vally and R. Courdier. A conceptual 'role-centered' model for design of multi-agent systems. In *Lecture notes in artificial intelligence subseries of lecture notes in computer science, vol .1599.* Springer-Verlag, 1998.