JOURNEE MULTI-AGENTS ET COMPOSANTS
JMAC 2004

mardi 23 novembre 2004

à l'Ecole des Mines de Paris

Avec le soutien du GDR Algorithmique, Langage et Programmation du CNRS

JOURNEE MULTI-AGENTS ET COMPOSANTS
JMAC 2004

mardi 23 novembre 2004
à l'Ecole des Mines de Paris

Les systèmes multi-agents proposent un ensemble de concepts, modèles et outils pour modéliser, développer des systèmes coopératifs, complexes, hétérogènes, évolutifs, décentralisés et ouverts pour la simulation, la résolution ou l'ingéniérie des systèmes. Ils ont acquis un impact important dans le domaine de l'intelligence artificielle et des systèmes distribués. Ils ont été utilisés avec succès dans la supervision, le commerce électronique, la domotique, les services Web, etc.

Parallèlement, dans le domaine du logiciel, les composants logiciels font l'objet de propositions variées, qualifiées d'industrielles (EJB, COM, .Net, Fractal) ou plus académiques (ArchJava, ACME, ...). Les applications concernent plus particulièrement le domaine du génie logiciel : la programmation, la vérification ou les infrastructures logicielles. Des applications réussies ont été conduites dans le domaine des lignes de produit multi-media, du commerce électronique, des systèmes d'exploitation, des télécommunications, etc.

Des besoins croisés émergent actuellement comme l'utilisation des composants pour les agents. Il s'agit d'utiliser la notion de composant pour aider à la conception, la construction et le déploiement de systèmes multi-agents, modulaires et réutilisables. Inversement d'autres tentatives essaient de transposer des propriétés des agents vers les composants pour concevoir les applications réparties du futur. Dans ce cas il s'agit de donner plus d'autonomie aux composants de l'application : capacités d'adaptation, de prise de décision, d'auto-assemblage (incluant le match-making), de coordination (avec d'autres composants), et donc de s'inspirer de concepts (autonomie, adaptation, coordination) des systèmes multi-agents. Comme on le voit avec les deux tendances précédentes il existe un champ d'investigation à la frontière des multi-agents et des composants.

L'objectif de la journée JMAC est de faire le point sur l'intersection de ces deux domaines de recherche en informatique en essayant de mettre en évidence les points communs, les divergences, les complémentarités au niveau des concepts, des modèles, des méthodes, des applications cibles.

JOURNEE MULTI-AGENTS ET COMPOSANTS
JMAC 2004

mardi 23 novembre 2004
à l'Ecole des Mines de Paris

COMITE DE PROGRAMME

Jean-Claude Royer, EM Nantes (Président)

Philippe Aniorté	LIUPPA, Pau
Noury Bouraqadi	EM Douai
Jean-Pierre Briot	LIP6, Paris
Christophe Dony	LIRMM, Montpellier
Marie-Pierre Gleize	IRIT, Toulouse
Marc-Philippe Huget	IMAG, Grenoble
Amedeo Napoli	LORIA, Nancy
Jacques Noyé	INRIA-EMN, Nantes
Philippe Mathieu	LIFL, Lille
Philippe Merle	INRIA, Lille
Michel Occello	LCIS - INPG, Valence
Jean-Paul Sansonnet	LIMSI-CNRS, Orsay
Christelle Urtado	EM Alès, Nîmes
Laurent Vercouter	EM St-Etienne

LECTEURS ADDITIONNELS

Djamel Seriai	EM Douai
Olivier Boissier	EM St-Etienne
Sylvain Vauttier	EM Alès, Nîmes

COMITE D'ORGANISATION

Olivier Boissier, Noury Bouraqadi, Jean-Claude Royer, Djamel Seriai, Christelle Urtado, Sylvain Vauttier, Laurent Vercouter

REMERCIEMENTS

Les organisateurs tiennent à remercier :
- les auteurs et les relecteurs,
- le GDR ALP du CNRS pour son soutien,
- l'Ecole des Mines de Paris pour son appui logistique dans l'organisation de la journée,
- les différentes personnes des Ecoles des Mines d'Alès, Douai, Nantes et Saint-Etienne qui ont rendu possible l'organisation de cette journée.

JOURNEE MULTI-AGENTS ET COMPOSANTS
JMAC 2004

mardi 23 novembre 2004
à l'Ecole des Mines de Paris

PROGRAMME DE LA JOURNEE	
10h — 10h30	Accueil — café offert
10h30 — 11h	Interactions entre composants et environnements multi-agents S. KHALFAOUI, W.L. CHAARI, A.-M. PINNA DERY
11h — 11h30	MAST : Un modèle de composant pour la conception de SMA Laurent VERCOUTER
11h30 — 12h	Déploiement et adaptation de systèmes P2P : Une approche à base d'agents mobiles et de composants S. LERICHE, J.P. ARCANGELI
12h — 14h	Pause repas libre
14h — 15h00	Exposé invité Jean-Pierre BRIOT
15h — 15 h30	Pause — café offert
15h30 — 16h	Négociations de contrats : des systèmes multi-agents aux composants logiciels Hervé CHANG - Philippe COLLET
16h — 16h30	Un modèle de spécification et d'implémentation de composants - rôles pour les systèmes multiagents Nabil HAMEURLAIN
16h30 — 17h	Discussion, bilan de la journée et perspectives

Avec le soutien du GDR Algorithmique, Langage et Programmation du CNRS

Interactions entre composants pour environnements multi-agents

Sami Khalfaoui* – Wided Lejouad Chaari – Anne-Marie Pinna Dery*****

**Laboratoire RIADI*
***Pôle GRIFT Laboratoire CRISTAL*
Ecole Nationale des Sciences de l'Informatique
Campus universitaire La Manouba
2010 Manouba Tunisie
sami.khalfaoui@ensi.rnu.tn - wided.chaari@ensi.rnu.tn

****Laboratoire I3S, Université de Nice*
Bat ESSI,
930, route des Colles
06903 Sophia Antipolis Cedex
pinna@essi.fr

RÉSUMÉ. *Actuellement les systèmes informatiques sont de plus en plus complexes, souvent répartis sur plusieurs sites et constitués de logiciels en interaction entre eux ou avec des êtres humains. Le besoin d'utiliser la technologie agents s'est fait ressentir et les évolutions dans ce domaine sont remarquables. Dans ce contexte, les applications à base d'agents doivent aussi bien s'adapter aux modifications de l'environnement et à l'évolution des interactions qu'aux retrait et ajout de nouveaux composants. Dans ce papier, nous décrivons certains travaux sur les interactions logicielles et la programmation par composants et leur utilisation dans un environnement agent dynamique. Nous illustrons notre approche par une application de trafic routier.*

MOTS-CLÉS : *Agents, interactions, adaptation, évolution.*

1. Introduction

Ces dernières années, les applications à base d'agents sont utilisées dans de multiples domaines et sont devenues de plus en plus complexes et distribuées. La complexité des systèmes multi-agents (SMA) fait que ces derniers ont besoin de plus de dynamicité et d'évolutivité entre les composants qui en font partie.

L'évolutivité peut concerner les interactions comme elle peut concerner les agents eux-même. Dans le premier cas, il s'agit d'ajouter, d'enlever et de changer les interactions entre agents. Dans le second cas, il s'agit de faire évoluer un système multi-agents en y ajoutant un nouveau service (matérialisé par un agent), non prévu à l'avance. La dynamicité consiste à faire ces modifications en cours d'exécution.

La dynamicité est un élément indispensable pour répondre aux besoins des nouvelles applications. Grâce aux progrès technologiques autour des communications réseaux (sans fil ; ATM), les besoins applicatifs se recentrent autour d'une prise en compte rapide du contexte d'exécution afin d'adapter les fonctionnalités offertes par les applications à la situation en cours. Si on prend l'exemple d'une application de trafic routier, le contexte d'exécution peut impliquer la mise en place d'interactions spécifiques (délestage, mise en place de secours…), en cas d'accidents, de catastrophes naturelles qui ne devraient pas nécessiter l'arrêt de l'application. Nous retrouvons aussi dans ce même esprit, les applications de simulation qui nécessitent de tester différentes situations d'interactions sans arrêter l'application. Cependant, ce besoins n'a pas fait l'objet de beaucoup de travaux, et les travaux qui se sont intéressés à la dynamicité dans les systèmes multi-agents comme [RIB 98], [JOU 03], [QUE 03] et [MEY 03] n'offrent pas la possibilité de modifier concrètement les interactions entre agents dynamiquement.

Vu la maturité qu'a pu atteindre le monde objet et composant, le besoin de dynamicité et d'évolutivité a été déjà soulevé et plusieurs travaux ont été menés en ce sens à l'instar des travaux de [BLA 02] autour des composants et qui portent sur l'adaptation dynamique des applications à base de composants.

Dans cet article, nous allons montrer comment les résultats obtenus dans le domaine des applications réparties construites à base de composants peuvent être un atout dans la réalisation de SMA dynamiques et hétérogènes. En particulier, nous nous focalisons sur les résultats de l'équipe RAINBOW qui consistent en l'utilisation d'un serveur d'interaction appelé « NOAH » et un langage « ISL » pour la définition, la pose et la destruction des interactions entre composants en cours d'exécution.

L'organisation de l'article se présente comme suit : dans la section 2, nous décrivons les travaux sur les interactions aussi bien dans les SMA que dans les

composants. Dans la section 3, nous présentons le cas pratique qui servira d'illustration tout au long de notre travail. La section 4 présente l'utilisation et l'apport des interactions à travers l'exemple décrit. Et, en conclusion nous reprenons les principales idées et nous présentons les perspectives.

2. Travaux sur les SMA et les Composants vs Evolutivité des applications

Dans cette section, nous décrivons les travaux dans le domaine des agents qui répondent aux besoins d'adaptabilité des SMA actuels. Puis, nous passons en revue les travaux sur les composants qui présentent un intérêt pour notre travail.

2.1. Principaux travaux sur la dynamicité des SMA

[RIB 98] propose un modèle dynamique d'interactions pour les SMA. Le principe est de soulager l'agent de certaines tâches d'interaction et de les transférer à une sorte de *messager*. Ce *messager* peut manipuler, dynamiquement, le message de différentes manières : trouver le bon destinataire dans le cas où l'adresse est incorrecte, donner une meilleure présentation du message, s'assurer que le message sera bien interprété par le destinataire (ajouter des informations supplémentaires au message d'origine). Aussi, l'utilisation du *messager* facilite-t-elle l'interaction entre agents hétérogènes. Ce travail se place dans le même contexte que [LEJ 98] qui propose l'utilisation d'un agent spécialisé pour prendre en charge la communication entre les autres agents.
[JOU 03] résout dynamiquement le problème d'interopérabilité de protocoles en utilisant un agent intermédiaire appelé *proxy* qui s'intercale entre les agents participant à une conversation. Le *proxy* doit implémenter un protocole assez générique pouvant être adapté aux autres protocoles.
En ce qui concerne l'évolutivité des SMA, [MEY 03] propose une plate-forme générique de simulation SIMENV permettant de construire des simulations à base d'agents hétérogènes communicants via des canaux de communications et de faire évoluer dynamiquement le système par l'ajout et la suppression d'agents et de canaux de communication entre agents en cours d'exécution. Ces agents et ces canaux de communication doivent être prévus à l'avance.

Les travaux sur l'hétérogénéité des agents s'intéressent à l'adaptation dynamique des messages à différents protocoles, l'intégration dynamique d'un intermédiaire *proxy*, etc. Cependant, peu de travaux ce sont intéressés à la dynamicité par l'ajout, la modification et la suppression d'interactions en cours d'exécution, ou encore, l'ajout, dynamiquement, d'un agent non prévu à l'avance.

Ainsi, l'idée de la dynamicité dans les SMA mérite d'être creusé davantage. C'est pourquoi nous nous sommes tournés vers le monde objet et composant qui, quant à lui, a su donné des éléments de réponse dans ce domaine. Les travaux sur les composants sont déjà intervenus dans le monde agent, mais principalement pour la conception et la modélisation des SMA.

Yoo M. dans [YOO 00] s'intéresse au problème de modélisation des agents et utilise une approche componentielle pour leur conception. Ici, les composants sont utilisés pour leur caractéristique évolutive. En effet, en utilisant les composants une partie du service peut être modifiée sans remettre en cause les autres. De plus, l'utilisation des composants dans ce contexte a pour objectif de favoriser la réutilisation des agents.
Meurisse et Briot dans [MEU 01] se proposent d'utiliser les composants pour réifier les comportements internes des agents. L'intérêt de cette approche est principalement le découplage entre entités. Les dépendances n'étant plus implicites, mais définies de manière externe. Ce qui autorise la modification du graphe de connexion sans avoir à changer les composants et facilite la réutilisation des comportements.

Dans le paragraphe suivant nous présentons brièvement des travaux sur les interactions entre composants qui pourraient être appliqués aux SMA pour améliorer l'évolutivité des applications.

2.2. Composants – Interactions et dynamicité

Les interactions étant omniprésentes dans le monde réel, différents travaux ont pris en charge leur spécification et leur mise en œuvre dans la communauté des développements d'applications à base de composants. Leur présence permet essentiellement de mieux décrire les assemblages au déploiement d'une application ou dynamiquement à l'exécution lorsque l'on souhaite adapter une application.

Les principaux concepts sous-jacents aux ADLs (Architecture Description Language) sont les composants et les connecteurs assemblés à l'aide de configurations [MED 97]. Une configuration architecturale définit la structure de l'architecture d'une application au travers d'un graphe de composants et de connecteurs. Les langages de configuration visent par exemple à exploiter les descriptions d'architectures pour produire en partie l'implantation des composants ou pour supporter, au moins partiellement, le déploiement des applications. Le dernier-né de ces langages est le Component Assembly Descriptors du CORBA Component Model, qui permet une automatisation du déploiement. Certains de ces langages ont intégré une dimension dynamique permettant de préciser les évolutions possibles d'une application comme, par exemple, l'ajout ou le remplacement de composants et de connexions à l'exécution [SEN 02].

Le développement d'applications sur des plates-formes à composants « standards » tels que EJB [EJB 00], CCM [MAR 02] ou .Net [RIC 02], force à prévoir a priori les interactions au niveau des interfaces et du code métier des composants. Par Le modèle de composants CORBA permet d'intégrer au composant la réactivité aux perturbations du système. Néanmoins, celles-ci

doivent être prévues statiquement, et la connectique (comment réagir à la présence d'un événement dans une source, quel événement produire dans un puits, etc.) reste à la charge du programmeur des composants [BER 02]. Certains services (événements, notifications), l'utilisation de scripts, d'intercepteurs CORBA [OMG 01] ou les conteneurs ouverts [VAD 02] se présentent également comme des technologies permettant d'implémenter les interactions. Aussi, la part de connectique et de contrôle induite par les interactions (synchronisation, conditionnelle, etc.) est-elle toujours à la charge du programmeur et dépend donc des plates-formes cibles. La programmation par aspects ([PAW 01], [RIV 00], [SAR 01]) proposent également d'expliciter les interactions. Néanmoins leur mise en oeuvre en terme de réception et émission de requêtes rend non « naturelle » l'écriture de ces interactions.

Ces techniques sont essentielles à la réalisation d'une adaptation dynamique des composants, mais il est nécessaire de les abstraire pour faciliter leur programmation et aller vers des systèmes plus sûrs. Les travaux de l'équipe RAINBOW (http://rainbow.essi.fr) autour de NOAH [BLA 02], quand à eux, considèrent les interactions comme des entités de première classe. Les interactions sont ainsi décrites dans des schémas d'interactions. Ces schémas d'interaction, écrits en ISL (Interaction Specification Language), sont définis indépendamment des implémentations en se basant sur les interfaces des composants.

Aussi est-ce sur la base de ces travaux que nous allons proposer de rendre les SMA dynamiques et évolutifs. La démarche que nous proposons permet une gestion dynamique des interactions par l'ajout, la modification et la suppression des interactions entre agents sans arrêter l'application. Cette démarche permet aussi d'ajouter des agents, qui n'ont pas été prévus lors de la conception, au système en cours d'exécution.

3. Exemple applicatif

Dans cette section nous présentons l'application qui a servi de base pour étudier notre approche. Il s'agit d'une application de simulation de trafic routier basée sur des agents. Le choix de cette application est fondé sur sa nature dynamique. Cette application fait intervenir au moins les classes d'agents suivantes : l'agent Conducteur qui est l'acteur principal de la simulation. Cet agent cherche à atteindre son point d'arrivée à partir d'un point de départ tout en évitant les situations de conflit. Plusieurs types d'agents Conducteurs peuvent exister, par exemple, Conducteurs non disciplinés, les Conducteurs spéciaux (ambulances, voitures de police,etc.) , etc. Chaque type de ces agents Conducteurs présente un comportement spécifique. L'agent Environnement sert à représenter graphiquement la simulation et à fournir aux agents Conducteurs des informations générales sur l'environnement. L'agent Carrefour est lié à un carrefour donné et contrôle les Conducteurs présents au niveau de ce carrefour. Il

existe plusieurs types d'agent Carrefour selon que le carrefour est contrôlé par des feux, des panneaux de signalisation, ou bien, il s'agit d'un carrefour simple. L'agent Voie concerne une voie donnée. De même, il existe plusieurs types d'agents Voies selon le type de la voie en question : avec feux, avec panneaux de signalisation, ou bien, voie simple. Le rôle de cet agent est de contrôler les Conducteurs se trouvant sur la voie. Et, l'agent Policier qui a une hiérarchie supérieure et qui intervient au niveau des carrefours pour gérer le passage des Conducteurs. Pour réaliser cette simulation nous avons utilisé la plate-forme JADE [BEL 02] dans laquelle les agents interagissent par échange de messages dans le langage FIPA-ACL [FIP 02].

Notre application présente plusieurs besoins de dynamicité. Certains de ces besoins peuvent être pris en charge par la plate-forme agent, d'autres pas.En effet, lors d'une simulation, nous pouvons avoir besoin de rajouter de nouveaux agents à la simulation pour la rendre plus complexe et observer le comportement qui en résulte. Cette opération est prise en charge par les plates-formes agent actuelles, notamment Jade, à condition que le nouvel agent ajouté soit prévu à l'avance et développé avec le SMA. Par exemple, il est possible de rajouter, au cours d'une simulation, des agents Conducteurs pour tester différentes situations de trafic. Ceci est aussi valable pour la suppression.

Cependant, il se peut que lors d'une simulation, nous ayons besoin d'intégrer un nouvel agent qui n'était pas prévu lors du développement du SMA. Par exemple, nous voulons intégrer l'agent Piéton dans la simulation. Ce type d'ajout n'est pas faisable, dynamiquement, avec les plates-formes agents actuelles. Ceci est dû principalement au fait que le code de la communication est intégré dans le code de l'agent. Il faut donc, arrêter le SMA, développer le nouvel agent en tenant compte de ses interactions avec les autres agents et modifier le code des agents qui seront en interaction avec ce nouvel agent.

Aussi, au cours d'une simulation, il serait opportun, de pouvoir rajouter une nouvelle interaction entre des agents du SMA et observer le nouveau comportement du système. Par exemple, nous voulons rajouter une nouvelle interaction entre des agents Conducteurs qui énonce que lorsqu'un Conducteur s'arrête trop longtemps, son poursuivant klaxonne et les deux conducteurs entament une négociation. Cette possibilité de rajouter des interactions à un SMA en cours d'exécution n'est pas supportée par les plates-formes agents actuelles.
Autres fonctionnalités non prises en charges par les SMA, ce sont la modification et la suppression dynamique d'interactions entre agents. En effet, il est intéressant, lors d'une simulation, de pouvoir tester différents protocoles d'interactions entre agents. A travers cette simulation de trafic routier, nous avons essayé de dégager les possibilités qu'offrent les SMA actuels, en terme de dynamicité, ainsi que leurs limites. Dans la section suivante, nous utilisons, pour le développement de notre application, une approche à base d'interactions entre

composants et nous essayons de voir si cette approche permet de surmonter les lacunes des SMA actuels.

4. Une approche basée sur les interactions entre composants pour les SMA

Dans cette partie, nous présentons l'approche à base d'interactions, ses apports et ses limites.

4.1. Présentation de l'approche

Notre approche consiste à dissocier les interactions entre agents du code métier des agents et à déléguer la gestion de ces interactions, désormais exprimées dans le langage ISL, à un serveur d'interaction appelé NOAH. Les agents seront toujours gérés par la plate-forme agents habituelle.

Nous avons repris la même application de simulation de trafic routier en utilisant la nouvelle approche et nous avons essayé de voir si la nouvelle version de l'application, qui utilise ISL et NOAH, permet de surmonter les faiblesses de la première version, à savoir, les possibilités d'ajout, de suppression et de modification dynamique d'interactions entre les agents ainsi que, l'ajout d'agent, non prévu dans le SMA et ce, sans arrêter la simulation. Nous essayons aussi de dégager les limites de cette approche.

Voici deux schémas d'interaction, pour la gestion du trafic routier, tirés de la nouvelle version de l'application. Le premier gère deux véhicules qui se suivent et le second gère le passage d'un carrefour géré par Feux.

```
Interaction suivreVehicule(Conducteur conducteur, Conducteur
poursuivant)
{
        conducteur.arret()→
                conducteur._call[1] ;
                if(conducteur.calculDistance(poursuivant.
                getPosition())<distanceSecurite) then
                        poursuivant.arret()
                end if

        poursuivant.avancer()→
                if(poursuivant.calculDistance(conducteur.
                getPosition())<distanceSecurite) then
                        poursuivant.arret()
                else
                        poursuivant._call
                end if

        conducteur.avancer()→
                conducteur._call;
                if(poursuivant.stop()) then /* à l'arrêt
```

[1] _call signifie que nous exécutons la méthode qui a déclenché la règle.

```
                        poursuivant.avancer()
                end if
}
```

Ce schéma d'interaction relie deux agents conducteurs et comporte trois règles d'interaction. La première indique que si un agent Conducteur s'arrête et si la distance qui le sépare de son poursuivant est inférieure à la distance de sécurité, le poursuivant s'arrête à son tour. Suivant la seconde règle, un Conducteur n'avance que si la distance qui le sépare de son précédent est supérieure à la distance de sécurité. Et, la troisième énonce que si un Conducteur avance son poursuivant avance aussi. Une interaction à partir de ce schéma d'interaction est posée dynamiquement entre deux agents Conducteurs lorsque qu'un agent Voie détecte que ces deux Conducteurs se poursuivent. Et, elle est dynamiquement supprimée dès que la Voie détecte que les conducteurs ne se suivent plus.
On pourrait envisager la définition d'un autre schéma d'interaction pour qu'un poursuivant impatient klaxonne lorsque son prédécesseur est à l'arrêt. Il est possible de basculer dynamiquement d'un type d'interaction à une autre entre les deux mêmes conducteurs très facilement grâce à la pose et le retrait d'interaction dynamique.

```
interaction rencontrerFeux(Conducteur conducteur, Feux feu)
{
        conducteur.avancer()→
                if(feu.getCouleur.equals("rouge")) then
                        conducteur.arret()
                else
                        conducteur._call
                end if


        feu.changeCouleur()→
                if(feu.getCouleur().equals("vert")&&
                conducteur.stop()) then
                        conducteur.avancer()
                end if
}
```

Selon ce schéma d'interaction, un Conducteur ne peut avancer que si le Feu est au vert. Une interaction à partir de ce schéma d'interaction est posée dynamiquement lorsqu'un Conducteur entre dans un carrefour géré par des Feux. Cette interaction est supprimée dynamiquement lorsque ce Conducteur quitte le carrefour.

Comme le montre ces exemples, le comportement n'est plus imbriqué en totalité dans le code des agents mais réparti dans différentes règles d'interactions. Ceci réduit la complexité et facilite la mise à jour du code en localisant la modification. Un autre apport de NOAH consiste en la fusion des règles d'interactions qui permet de retrouver automatiquement un comportement global équivalent. Quelque que soit l'ordre dans lequel les règles sont fusionnées, le comportement résultant est toujours le même. Ceci est du au fait que la fusion est

associative et commutative. Les principes de la fusion sont détaillés dans [BER 01].

Par exemple, soit les agents Conducteur « conducteur_1 » et « conducteur_2 » qui circulent sur une voie contrôlée par un agent voie « voi_1 ». Lorsque « voi_1 » détecte que « conducteur_2 » poursuit « conducteur_1 », il pose une interaction instanciée à partir du schéma «suivreVehicule » qui prend comme arguments « conducteur_1 » et « conducteur_2 ». Puis lorsque « conducteur_1 » arrive au niveau du carrefour « carrefour_1 » contrôlé par l'agent Feux « feu_1 », l'agent « carrefour_1 » pose l'interaction « recontrerFeux » entre « conducteur_1 » et « feu_1 ». Nous remarquons que la méthode « avancer () » du Conducteur « conducteur_1 » se trouve impliquée dans deux règles d'interaction. Ces deux règles sont dynamiquement fusionnées pour générer une seule règle avec un comportement équivalent. La règle résultante se présente comme suit :

```
Conducteur_1.avancer()→
if(feu_1.getCouleur().equals("rouge") then
        conducteur_1.arret()
else
        conducteur_1._call;
        if(conducteur_2.stop()) then
                conducteur_2.avancer()
        end if
end if
```

L'architecture interne d'un agent illustrée par la figure 1 peut être représentée par :

- Ses rôles : que fait un agent.

- Ses compétences : celles-ci peuvent être internes, c'est à dire, comment l'agent assure ses rôles par son propre savoir-faire. Comme elles peuvent être le fruit d'une coopération et interactions avec les autres agents.

- Ses interactions : permettant à un agent de communiquer.

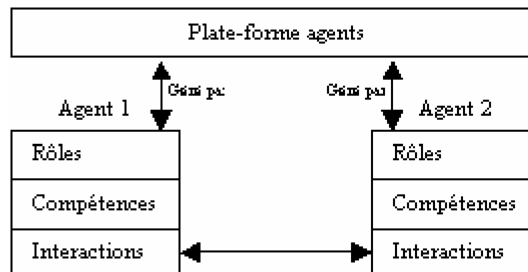


Figure 1. Architecture interne d'agents

L'approche par interactions que nous proposons (voir figure 2) dissocie les interactions de l'architecture interne de l'agent. Ces interactions, désormais

gérées par le serveur NOAH, définissent aussi le comportement social des agents. En plus, une politique d'arbitrage et de prise de décision peut être définie dans l'agent pour la gestion des comportements.

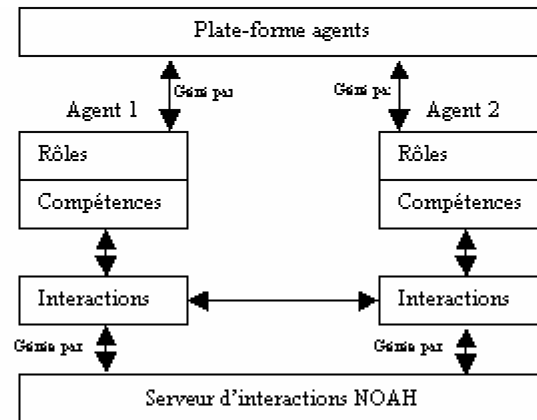


Figure 2. Nouvelle architecture d'agents

4.2. Apports et limites de l'approche à base d'interactions

Dans l'approche à base d'interactions, nous remarquons qu'il est désormais possible d'ajouter, au cours d'une simulation, une interaction entre des agents. Pour cela, nous demandons au serveur NOAH de poser une interaction définie dans un schéma d'interaction donné entre ces agents (un conducteur devient le poursuivant d'un autre). Il est également possible d'ajouter une nouvelle forme d'interaction dynamiquement en enregistrant un nouveau schéma d'interaction auprès du serveur (introduction de poursuivants impatients, par exemple). Aussi, est-il désormais possible de supprimer des interactions, en cours d'exécution. Ces possibilités d'ajout et de suppression d'interaction en cours d'exécution rendent possible la modification dynamique d'interactions. Ainsi, pour modifier une interaction entre agents sans arrêter le SMA, il faut la supprimer, enregistrer éventuellement un nouveau schéma d'interactions, puis poser dynamiquement une autre interaction entre les agents.

La possibilité d'ajouter une interaction en cours d'exécution rend possible l'ajout dynamique d'agents non prévus dans le SMA. En effet, pour intégrer un nouvel agent dans un SMA, il faut développer la classe du nouvel agent puis la lancer via la plate-forme agent. Par la suite, il suffit de lier cet agent aux autres agents du SMA à partir des schémas d'interactions qui auraient été décrits pour exprimer sa communication avec les agents du système.

Malgré ses apports, cette approche présente une limite due aux différences entre le monde agents et le monde composants. Il s'agit de la perte de l'asynchronisme

au niveau des interactions entre agents. En effet, dans un modèle agent classique, lorsqu'un agent reçoit un message, il peut ne pas le traiter immédiatement (mode asynchrone). Alors que dans la nouvelle approche, la partie droite d'une règle d'interaction est immédiatement exécutée lors du déclenchement de cette règle (mode synchrone). Cette limitation est due au Serveur Noah qui prend pour base de la communication entre composants l'appel de méthodes synchrone.

Nous réfléchissons à une projection des interactions sur un mode communication asynchrone. Une telle solution consiste à poser des interactions ISL au niveau des méthodes « send() »[2] et « receive() » d'un agent. Ces méthodes servent, respectivement, à envoyer des messages en FIPA-ACL et à les recevoir. L'intégration des interactions doit respecter les mêmes règles que pour la communication synchrone : si au niveau du « send() » (ou « reveive() ») des interactions sont posées, alors, les interactions sont exécutées sinon l'envoi de message simple est conservé. Lorsqu'une interaction est posée au niveau du « receive() » (resp. send), elle n'est déclenchée que si la méthode « receive() » (resp. send) est invoquée. Un nouveau mécanisme de fusion doit être mis en place pour prendre en considération le message envoyé ou reçu dans la fusion des interactions. Les interactions ne concernant pas le même message ne devant pas être fusionnées.

Dans le même contexte, nous envisageons de concevoir un agent qui prendra en charge la gestion de l'asynchronisme. La spécification de cet agent fera l'objet d'une étude ultérieure.

5. Conclusion

Dans ce papier, nous avons souligné un besoin de rendre les systèmes muti-agents plus dynamiques et plus évolutifs. Nous avons proposé une approche permettant de poser, modifier et détruire les interactions entre agents en cours d'exécution, chose qui nécessitait l'arrêt de l'application et la modification du code des agents. Notre approche offre aussi la possibilité d'intégrer dynamiquement un agent non prévu à l'avance dans un SMA. Pour réaliser ce travail nous avons utilisé le langage ISL pour la définition des interactions et le serveur d'interaction NOAH, tous deux issus des travaux de l'équipe RAINBOW sur les interactions entre les composants. Nous avons validé notre approche et vérifié sa faisabilité à travers un exemple d'une simulation de trafic routier. Toutefois, cette approche présente une limitation au niveau du mode d'interaction des agents qui devient synchrone. Plusieurs solutions ont été étudiées pour contourner ce problème. Il s'agit d'intervenir au niveau de la plate-forme agent et précisément au niveau du mécanisme de gestion des interactions

[2] Les méthodes « send » et « receive » sont spécifiques aux agents développées en JADE. Cependant, les agents développés sur des plates-formes différentes possèdent des méthodes similaires pour pouvoir échanger des messages.

pour prendre en compte les spécificités des agents, l'expression et le déclenchement des interactions logicielles.

Ainsi, une des principales perspectives de notre travail est de mettre en œuvre une solution pour résoudre le problème de l'asynchronisme. Nous projetons aussi d'utiliser ISL et NOAH pour des interactions plus complexes, comme les négociations, basés sur des protocoles, et vérifier si la dynamicité est aussi bien assurée. Nous travaillons aussi sur une meilleure manière d'intégrer ce nouveau type d'interactions dans la plate-forme agent que nous utilisons (JADE), puis d'essayer de rendre cette intégration assez générique pour l'appliquer à d'autres plates-formes agent. Aussi, utiliserons-nous notre nouvelle approche pour définir une bibliothèque générique d'interactions pour le trafic routier ou toute autre application du même type (par exemple, contrôle du trafic d'un réseau informatique).

Bibliographie

[BEL 02] Bellifemine, F., Caire, G., Trucco, T., Rimassa, G., « Jade Programmer's Guide », JADE 2.5,2002.

[BER 01] BERGER L., « Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés: le modèle MICADO », PhD thesis, Université de Nice, 2001.

[BER 02] Berger L., « Support des interactions dans les systèmes objets et componentiels », Numéro spécial de L'OBJET : Coopération dans les systèmes à objets, vol. 7, 2001-2002.

[BLA 02] Blay-Fornarino M., Ensellem D., Occello A., Pinna-Dery A., Riveill M., Fierstone J., Nano O., Chabert G., « Un service d'interactions : principes et implémentation », Journée des composants, 2002.

[CAR 03] Carabelea C., Boissier O., Florea A., « Autonomie dans les systèmes multi-agents », JFSMA '03, 2003.

[EJB 00] «Enterprise Javabeans Specification Version 1.1», janvier 2000, Sun Microsystem Inc. http://java.sun.com/poducts/ejb/docs.html

[FIP 02] « The FIPA ACL Message Structure Specifications », 6 décembre 2002.

[HOF 01] Hofmann Thomas F., « OPENSPACES, An Object-Oriented Framework for Configurable Coordination of Heterogeneous Agents », 2001.

[JOU 03] Jouvin D., Hassas S., « Architectures dynamiques de systèmes multi-agents conversationnels », Journée francophone des systèmes multi-agents, 2003.

[LEJ 98] Lejouad-Chaari W., Mouria-Beji F., «High level commununication protocol in a distributed multiagent system », 11th Intenationa Conference on Indistrial and Engineering Applications of Artificial Intelligence end Expert Systems, Spain, June 1998.

[MAR 02] Marvie R., Pellignini M.-C., «Modèles de composants, un état de l'art», Numéro spécial de L'Objet, vol. 8, no 3, 2002, Hermès Sciences, Coopération dans les systèmes à objets.

[MED 97] Medvidovic N., Taylor R., «A Framework for Classifying and Comparing Architecture Description Languages», M. Jazayeri et H. Schauer, editeurs, Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), pages 60_76. Springer_Verlag, 1997.

[MEU 01] Meuriosse T., Briot J.P., « Une approche à base de composants pour la conception d'agents », Technique et science informatiques Volume 20, pages 583_602, 2001.

[MEY 03] Meyer D., Buchta C., « SIMENV: A Dynamic Simulation Environment for Heterogeneous Agents », 2003.

[OMG 01] OMG, CORBA 2.4.2 Interceptors chapter, février 2001, OMG Document formal/01-02-57, http://www.omg.org.

[PAW 01] Pawlak R., Seinturier L., Duchien L., Florin G., «JAC: A Flexible Framework for AOP in Java», Reflection'01, Kyoto, Japan.

[QUE 03] Quenum J., Slodzian A., Aknine S., « Configuration automatique de modèles d'interaction d'agents », Journée francophone des systèmes multi-agents, 2003.

[RIB 98] Ribeiro Alexandre M., Demazeau Y., «A Dynamic Interaction Model for Multi-Agent Systems », ,1998.

[RIC 02] Jeffrey Richter «Applied Microsoft .Net Framework Programming», MS Press, ISBN 0735614229, 2002

[RIV 00] Riveill M., Bruneton E., JavaPod: «an Adaptable and Extensible Component Platform», RM'2000, Workshop on Reflective Middleware, New York, USA, April 2000.

[SEN 02] Senart A, Riveill M. «Aspects dynamiques des langages de description d'architecture logicielle», Numéro spécial de la revue L'OBJET : Coopération dans les systèmes à objets, 8(3):109-129, 2002.

[SAR 01] Sarradin F., Ledoux T., «Adaptabilité dynamique de la sémantique de communication dans Jonathan», Colloque Langages et Modèles à Objets (LMO'01), Hermès Science, L'objet-7/2001, Le Croisic, France, janvier 2001.

[SHA 95] Shaw M., DeLine R., Klein D., Ross T., Toung D., Zelesnik G, «Abstraction for Software Architecture and Tools to Support Them», IEEE Trans. Software Engineering, SE-21(4):314_335, Avril 1995.

[VAD 01] Vadet M., Merle P., « Les conteneurs ouverts dans les plates-formes à composants », Journées Composants 2001, Besançon, 25-26 octobre.

[YOO 01] Yoo M, Briot J.P., « Une approche componentielle pour la modélisation d'agents mobiles coopérants », 2001.

MAST : Un modèle de composant pour la conception de SMA

Laurent Vercouter

Département SMA, Centre G2I
École NS des Mines de Saint-Étienne
158 cours Fauriel, 42023 Saint-Étienne, Cedex 2, France
Laurent.Vercouter@emse.fr

RÉSUMÉ. *Cet article présente un modèle de composant, implémenté dans l'environnement de développement MAST (Multi-Agent System Toolkit), utilisé pour la conception de systèmes multi-agents. Ce modèle de composant a été défini spécifiquement pour la conception d'applications multi-agents et permet d'aborder à la fois le niveau global (par des composants orientés système) et le niveau local (par des composants orientés agent). Un concepteur construit alors son application en sélectionnant les composants nécessaires, en les assemblant autour d'une architecture d'intégration, puis en développant les composants non génériques. Cette approche à base de composants accroît la modularité et la généricité des développements favorisant ainsi leur ré-utilisabilité lors de développements futurs. Cette propriété démarque MAST des environnements existants par le fait qu'il n'est contraint par aucun modèle multi-agent particulier tout en fournissant une bibliothèque de modèles génériques.*

ABSTRACT. *This article presents a component model, implemented in the MAST (Multi-Agent System Toolkit) environment, used for the design of multi-agent systems. This component model has been defined especially for the design of multi-agent applications and makes it possible to tackle both the global level (with system oriented components) and the local level (with agent oriented components). A designer builds its application by selecting the relevant components, assembling them around an integration architecture and by developping some application specific components. This componential approach increases the modularity and the genericity of the coding, and facilitates component re-use for future applications. This property is original as MAST is not constrained by any multi-agent model and provides a library of generic models.*

MOTS-CLÉS : *Environnements de développement multi-agent, Plates-formes agents, composants*

KEYWORDS: *Multi-Agent IDE, Components*

1. Introduction

La conception et le développement d'un système multi-agent (SMA) sont des problèmes complexes car ils nécessitent la prise en compte de plusieurs parties du système qui peuvent souvent être abordées sous différents angles. Le concepteur doit identifier l'ensemble des problèmes à résoudre, trouver des modèles multi-agents pour leur résolution, les implémenter puis les intégrer en un système cohérent. Ces tâches justifient l'utilisation d'*environnements de développement* qui assistent le concepteur en lui fournissant des outils et des modèles déjà développés sur lesquels il peut s'appuyer.

Il existe un grand nombre de modèles multi-agents qui peuvent être indépendants, concurrents, complémentaires ou incompatibles. La combinaison de plusieurs modèles pour construire une application complexe est un problème non trivial. La plupart des environnements existants [BEL 03, GUT 00, FIP, NWA 99] sont fondés sur un modèle principal, ce qui évite le problème de cohérence entre modèles mais restreint les catégories d'applications développables à celles ciblées par le modèle. Quelques environnements récents [MEU 01, OCC 02, RIC 02] ont introduit l'idée d'utiliser un modèle de composant pour exprimer les dépendances entre modèles puis les combiner.

Cet article présente le modèle de composant utilisé dans l'environnement MAST (*Multi-Agent System Toolkit*). Il a été défini pour la conception de SMA et à ce titre répond à des besoins spécifiques. L'approche à base de composants adoptée est présentée dans la section 2 puis le modèle de composant utilisé pour la construction d'agent est détaillé en section 3. La section 4 donne un exemple d'application réalisable avec ce modèle. Enfin, la section 5 compare notre proposition aux modèles existants.

2. Approche à base de composants pour les SMA

Les SMA se sont souvent révélés être une solution adaptée dans le cas d'applications complexes [BOI 04] inabordables sous un angle global. Dans ce type d'applications, l'approche multi-agent permet de décomposer le problème général de manière à analyser séparément chacune de ses parties, à concevoir et développer des réponses à chacune d'entre elles, puis à intégrer l'ensemble en un système cohérent.

2.1. *Les décompositions Voyelles*

L'approche Voyelles [DEM 01] (également notée `AEIOU`) considère qu'un SMA est composé des quatre dimensions `Agents`, `Environnements`, `Interactions` et `Organisations` auxquelles il convient également d'ajouter une prise en compte des `Utilisateurs`. Elle est généralement employée pendant la phase d'analyse d'un problème pour identifier les sous-problèmes propres à chaque dimension, puis pour leur appliquer des modèles appropriés de résolution.

Cependant, dans des systèmes de taille importante, le sous-problème attaché à une dimension peut rester un problème complexe, lui-même composé de plusieurs facettes.

Il est alors intéressant d'utiliser à nouveau l'approche Voyelles pour décomposer une dimension en facettes `AEIOU` [BOI 03]. L'exemple le plus évident concerne la dimensions `Agents`. Même si un choix de langages et de protocoles est fait au niveau global lors de la conception des `Interactions`, il est nécessaire d'aborder, lors de la conception d'un agent, la manière dont il va utiliser et interpréter ces éléments d'interaction. Il en va de même pour les dimensions `EOU` qui ont aussi un sens à ce niveau local.

2.2. *Niveaux système et agent*

Cette double utilisation de l'approche Voyelles souligne les deux niveaux qu'on peut aborder dans un SMA. On peut considérer le SMA dans son ensemble (niveau système) ou se focaliser sur une dimension particulière. Dans le cas de la conception logicielle d'un SMA, ce second niveau concerne principalement la conception des entités logicielles actives du système - les agents - et nous ne considérons dans cet article que le niveau agent. Le modèle de composant proposé ici tient compte de ces niveaux en étant constitué de deux types distincts de composants : (i) les composants orientés système (COS) utilisés lors de la conception globale du SMA ; (ii) les composants orientés agent (COA) utilisés pour la conception d'un agent.

2.2.1. *Composants orientés système*

Un composant orienté système implémente une partie du système global, rattachée à l'une des dimensions Voyelles. Étant donné la grande diversité de COS envisageable (y compris au sein d'une même dimension), il est très difficile de définir un modèle général de composant ainsi qu'un mode d'interaction entre COS. Nous n'avons donc pour le moment pas spécifié de modèle spécifique aux COS. Quelques exemples de COS sont donnés ci-dessous selon la dimension concernée :

– **Agents :** tous les agents du SMA
– **Environnements :** environnement situé (ex. simulateur Robocup), ...
– **Interactions :** intergiciel de communication (ex. plate-forme FIPA [FIP]), ...
– **Organisations :** support organisationnel (kernel MadKit [GUT 00]), ...
– **Utilisateurs :** interfaces graphiques de l'application, outils de supervision, ...

2.2.2. *Composants orientés agent*

Au niveau global, un agent est considéré comme un composant orienté système. Mais si l'on se place à un niveau local à l'agent, celui-ci est un sous-système souvent complexe qui, dans le modèle présenté ici, est également formé par un assemblage de composants. Le niveau agent ne présente pas les mêmes besoins que le niveau système et il est nécessaire d'utiliser un modèle de composants différent pour la conception d'un agent. Ces composants orientés agent (COA) implémentent une fonctionnalité spécifique d'un agent et toute l'implémentation de l'agent est le résultat de leur assemblage. De plus, une dimension Voyelles donnée d'un agent est implémentée par un sous-ensemble de COA de cet agent.

2.2.3. *Architecture générale du système*

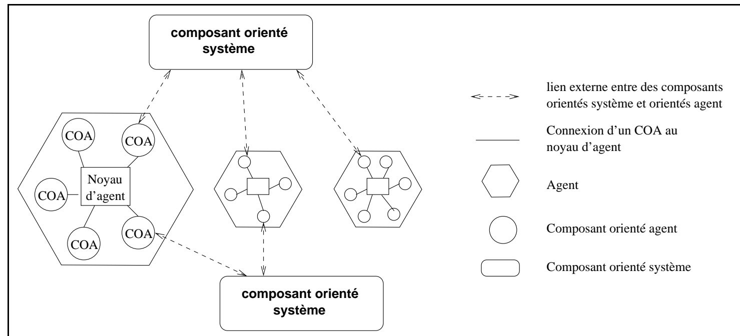


Figure 1. *composants orientés systèmes et orientés agents*

La figure 1 illustre un exemple de SMA constitué de cinq COS, dont trois agents eux-mêmes composés de COA. La base d'un agent est son *noyau*. Tous les COA d'un même agent sont connectés à son noyau qui gère les interactions entre composants orientés agent (le mode d'interaction entre COA est détaillé plus loin). Cet exemple montre aussi qu'un agent peut être relié à un composant orienté système par le biais d'un composant orienté agent spécifique. La manière dont ce COA, qui joue un rôle d'interface avec le COS, interagit avec celui-ci dépend fortement de la nature des composants concernés. Par exemple, si le COS est un environnement partagé par des agents, le COA correspondant apporte à l'agent des fonctionnalités de perception et d'action sur celui-ci. Si c'est un intergiciel de communication, le COA gère les envois et réceptions de messages, etc. La section suivante détaille le modèle de composant utilisé pour les COA ainsi que leur mode d'interaction.

3. Un modèle de composant pour la conception d'agents

Dans le modèle utilisé pour les COA, deux structures précises sont définies : (i) l'**interface d'un composant**, c'est-à-dire ce que le composant *fournit* aux autres composants et ce qu'il *requiert* d'eux ; (ii) les **événements échangés** entre composants.

3.1. *Besoins spécifiques à la conception d'agent*

Parmi les propriétés revendiquées par les systèmes multi-agents, on retrouve fréquemment l'ouverture, l'adaptativité et l'autonomie. L'ouverture signifie qu'il est possible d'ajouter ou de retirer de nouvelles fonctions au système après sa conception. La propriété d'adaptativité nécessite que le système modifie son fonctionnement de manière à prendre en compte les changements dans son contexte d'exécution. L'adaptativité peut être considérée comme une utilisation intelligente de l'ouverture car la

modification de fonctionnement du système peut être réalisée par l'ajout et/ou le retrait automatiques des fonctions concernées par le changement de contexte. Enfin, l'autonomie du système et de ses agents vise à réduire la place de l'utilisateur dans les processus de prise de décision. Un système est autonome pour une catégorie donnée de décisions s'il peut automatiquement (c'est-à-dire indépendamment de toute intervention extérieure) aboutir à une décision.

Ces propriétés sont généralement évoquées de manière globale pour indiquer que le système est ouvert à l'ajout et au retrait de COS qui sont adaptatifs et autonomes. Dès lors que nous proposons une approche à base de composants à la fois au niveau système et au niveau agent, il est essentiel que notre modèle de COA exhibe également ces propriétés. Un ajout automatique d'une fonction suite à un changement de contexte peut en effet nécessiter un ajout similaire de fonction au sein même des agents. Il en découle les besoins suivants pour le modèle de COA :

– **Ouverture :** Le modèle doit autoriser l'ajout ou le retrait de COA pendant l'exécution de l'agent. Ces ajouts/retraits ne doivent pas perturber la cohérence et la stabilité du fonctionnement de l'agent.

– **Adaptativité :** Les COA d'un même agent doivent être capables de s'adapter à des changements dans leur contexte d'exécution, c'est-à-dire dans notre cas à la présence de nouveaux COA ou à leur disparition. Cette adaptativité revient à la faculté de remettre en cause les connexions reliant les composants.

– **Autonomie :** Un COA est autonome dans l'accomplissement d'une fonction s'il ne nécessite pas d'intervention extérieure. Il est souhaitable dans notre modèle que les COA soient autonomes par rapport à l'utilisateur lorsqu'ils s'adaptent à un changement dans l'agent. Concrètement, la remise en cause de connexions doit être réalisée automatiquement et judicieusement par les COA.

Ces besoins sont originaux car spécifiques à la construction de systèmes multiagents. A notre connaissance, aucun modèle de composant existant ne les satisfait. Nous avons donc défini un nouveau modèle de composant détaillé dans la suite de cette section.

3.2. *Interface d'un composant*

Un composant est décrit par une interface (table 1) contenant les champs suivants :

– `name` : un nom symbolique associé au composant.

– `dimension` : l'ensemble des dimensions Voyelles auxquelles appartient ce composant. Ce champ n'est pas interprété par le noyau et a pour principal objectif de rendre plus lisible une bibliothèque de composant (en la structurant par dimension).

– `priority` : une valeur numérique qui représente la priorité *par défaut* du composant. Ce champ est interprété lors de la transmission d'un événement (cf section 3.4).

<table>
<tr><th><interface></th><th><event></th><th><evt_mask></th></tr>
<tr><td>NAME <string></td><td>NAME <string></td><td>NAME <string></td></tr>
<tr><td>DIMENSIONS <string>*</td><td>TYPE <string></td><td>TYPE <string></td></tr>
<tr><td>PRIORITY <float></td><td>FROM <string></td><td>PRIORITY <float> | null</td></tr>
<tr><td>PROVIDED_ROLES <string>*</td><td>TO <string>*</td><td>FROM <string>* | null</td></tr>
<tr><td>REQUIRED_ROLES <string>*</td><td>REPLY_WITH <evt_mask>*</td><td>TO <string>* | null</td></tr>
<tr><td>SENT_EVTS <event mask>*</td><td>CONTENT <Object></td><td>REPLY_WITH <evt_mask>* | null</td></tr>
<tr><td>HANDLED_EVTS <event mask>*</td><td></td><td>CONT_CLASS <Java class> | null</td></tr>
<tr><td></td><td></td><td>CONT_VALUE <Object> | null</td></tr>
</table>

Tableau 1. *Structure des interfaces et événements*

– `provided roles` : l'ensemble des noms des rôles joués par ce composant. Le concept de rôle correspond à celui de composant abstrait. Un rôle est également décrit par une interface et tout composant jouant ce rôle doit satisfaire aux contraintes exprimées par son interface.

– `required roles` : l'ensemble des noms des rôles qui doivent être remplis par au moins un autre composant de l'assemblage. L'assemblage ne fonctionnera correctement que si tous les rôles requis par tous ses composants sont remplis par au moins un composant de l'assemblage.

– `sent events` : l'ensemble des événements qui peuvent être émis par le composant. Chaque élément de cet ensemble est une description partielle d'un événement.

– `handled events` : l'ensemble des événements qui peuvent être traités par le composant. Chacun de ses éléments est une description partielle d'un événement.

3.3. *Structure d'un événement*

Un événement (cf table 1) contient les champs suivants :

– `name` : un nom symbolique associé à l'événement.

– `type` : le nom du type de l'événement. Ce nom fait référence à une liste finie de type d'événement (contenant par exemple les types `request`, `inform`, `send message`, `message received`, ...).

– `from` : le nom du composant qui a émis l'événement.

– `to` : l'ensemble des noms de rôles à qui s'adresse cet événement.

– `reply with` : l'ensemble des événements utilisables en réponse.

– `content` : Le contenu de l'événement. Sa nature est libre (comme l'implémentation existante est écrite en Java, le type `Object` lui est ici assigné).

Dans les structures d'interface et d'événement, il est fait référence à d'autres événements en utilisant le type `evt_mask`. Ce type est une description partielle d'événement pour contraindre les événements qui peuvent être utilisés dans une réponse (champ `reply with`), ou ceux qui peuvent être émis ou traités (champs `sent events`

et `handled events` d'une interface). Si aucune contrainte ne porte sur un champ, celui-ci a la valeur `null`. La signification de ces champs est très semblable à celle des champs d'un événement excepté celui portant sur la priorité. Cette priorité, uniquement interprétée si le `event mask` sert à décrire les événements traités par un composant (champ `handled events`), peut être utilisée de préférence à la priorité globale du composant (cf. section 3.4).

3.4. *Noyau d'agent*

L'assemblage se fait en connectant chaque COA à un élément central : le *noyau* de l'agent. La principale fonction de ce noyau est de déterminer automatiquement à quels composants transmettre un événement émis. L'algorithme de transmission d'événements est le suivant :

1) Un composant émet un événement en le transmettant au noyau auquel il est connecté ;

2) Le noyau sélectionne un ensemble de composants destinataires et les classe selon un ordre de priorité ;

3) Le noyau transmet l'événement au composant affichant la plus haute priorité ;

4) Ce composant traite l'événement. Après l'avoir traité, le composant peut décider de *consommer* ou non l'événement. S'il le consomme l'événement ne sera pas transmis aux autres destinataires potentiels ;

5) Si l'événement n'a pas été consommé, le composant de plus haute priorité est retiré de la liste des destinataires et l'événement est transmis à un autre composant en recommençant l'étape 3 jusqu'à ce que cette liste soit vide ou que l'événement soit consommé ;

6) Si l'événement nécessite une réponse (son champ `REPLY_WITH` n'est pas vide), le noyau recueille une réponse par destinataire retenu puis transmet l'ensemble au composant ayant émis l'événement initial. Si aucune réponse n'est attendue le processus est terminé.

La sélection des destinataires potentiels d'un événement se fait en comparant la structure de l'événement à l'interface de chaque composant. Un composant est destinataire d'un événement si l'un des deux critères suivant est rempli :

– Le composant remplit un des rôles indiqué dans le champ `TO` de l'événement.

– L'événement satisfait toutes les contraintes d'un des événements acceptés par le composant (dans la liste `handled event` de son interface).

Si le second critère est rempli, la priorité accordée au composant pour la réception de l'événement est celle attachée (s'il y en a une) au champ `handled event` correspondant. Sinon, le noyau considère sa priorité par défaut.

3.5. *Propriétés du modèle*

A partir de descriptions détaillées des interfaces et des événements, le noyau d'agent établit dynamiquement des connexions entre les composants de manière à ce que l'assemblage résultant soit ouvert et adaptatif. Aucune connexion n'est concrètement créée : le noyau interprète la description d'un événement pour établir une liste de destinataires potentiels. L'assemblage est ouvert à l'ajout ou au retrait de composants car la liste des interfaces qui sont comparées à chaque événement n'est pas statique.

Ce modèle de composant est adaptatif si les interfaces et événements sont décrits en faisant référence à des concepts partagés par l'ensemble des composants. Cela est principalement nécessaire lorsqu'il est fait référence à des rôles de composant et à des types d'événements. Si un nouveau composant est décrit selon des rôles et des types d'événements également utilisés par d'autres composants, le noyau est en mesure d'adapter l'assemblage pour intégrer ce nouveau composant.

Enfin, l'adaptation de l'assemblage est réalisée automatiquement sous réserve que les composants soient correctement décrits. L'affectation de priorités aux composants permet au noyau de gérer les situations où plusieurs composants sont concurrents pour la réception d'un événement. Cet événement est alors transmis successivement à chacun de ces composants jusqu'à ce que l'un d'entre eux le consomme, empêchant ainsi les composants de plus basse priorité de le recevoir. L'autonomie, par rapport aux utilisateurs, dont fait preuve le noyau pour adapter l'assemblage est intéressante si les priorités affectées aux composants sont judicieuses. Or, la valeur d'une priorité n'a de sens que relativement aux priorités des autres composants. C'est une des contraintes induites par le modèle de composant présenté ici : le concepteur d'un composant doit connaître la bibliothèque de composant disponible pour le décrire correctement, notamment en terme de priorités. Le gain provoqué par cette exigence est que l'utilisateur qui créé un assemblage n'a plus qu'à sélectionner plusieurs composants sans les connecter explicitement pour qu'ils fonctionnent ensemble. Il peut, s'il le souhaite, modifier les propriétés d'un composant pour gérer son assemblage plus finement ou alors laisser le noyau le faire de manière autonome.

4. Exemple d'utilisation

Ce modèle de composant a été utilisé pour créer des agents dans une application test décrite ci-dessous.

4.1. *Application test*

L'application utilisée pour tester notre modèle est une version simplifiée de celle introduite dans [BOU 00]. Elle prend la forme d'une grille sur laquelle sont positionnés plusieurs agents qui accomplissent des tâches pour lesquelles ils reçoivent une récompense. Il existe deux types de tâches qui sont également placées sur la grille :

des tâches simples (ST) et des tâches coopératives (CT). Une ST peut être réalisée par un seul agent. Une CT peut être réalisée par plusieurs agents qui doivent se coordonner pour l'accomplir ensemble. Chaque tâche a une durée d'exécution, une échéance et une récompense. Dans le cas de CT, la récompense est équitablement partagée par les agents qui l'ont réalisée.

Les agents se déplacent, à chaque pas de temps, d'une case ou ne bougent pas. Tous les agents connaissent la position des ST. Un agent ne connaît l'emplacement d'une CT que si sa position sur la grille est proche de la sienne ou si un autre agent l'a informé de son existence. Pour exécuter une tâche, un agent doit se déplacer sur la case qui la contient. Une CT ne peut être exécuté que si le nombre suffisant d'agents pour la réaliser, sont présents sur sa case.

Si un agent souhaite exécuter une CT, il doit trouver des partenaires parmi les autres agents. Pour cela un agent communique avec d'autres agents de deux manières : soit il les informe simplement de l'existence de la CT, soit il leur demande s'ils acceptent de réaliser une partie de cette CT. Les agents qui reçoivent une telle demande peuvent l'accepter (et se déplacer vers la case de la CT) ou la refuser.

Enfin, le comportement des agents est régulé par une norme : si un agent accepte une demande de participation, il est obligé de respecter son engagement. Le respect de cette norme n'est cependant pas assuré et il est possible que certains agents la violent.

4.2. *Compositions des agents*

A partir du modèle de composant décrit dans cet article, nous avons défini trois types d'agents :

– **L'agent social :** Il est capable de communiquer avec d'autres agents. Il accepte toute demande de participation si cela est possible compte-tenu de l'échéance de la CT et de ses engagements en cours. Son mode de fonctionnement fait qu'il respecte implicitement la norme.

– **L'agent social autonome :** C'est un agent social qui choisit d'accepter ou non de participer à une CT selon son intérêt propre (c'est-à-dire la décision qui maximise les récompenses reçues). Il peut ne pas respecter un engagement si cela augmente son bénéfice.

– **L'agent normatif autonome :** C'est un agent social autonome qui tente de maximiser ses gains mais sans violer la norme.

Chacun de ces types d'agents est conçu par un assemblage de composants pris parmi la bibliothèque de COA disponibles. Tous ces agents doivent contenir un composant **Communication** qui fournit des fonctions d'échanges de messages avec d'autres agents et un composant **Connexion à la grille** qui fournit des fonctions d'actions et de perception sur la grille. Un composant **Gestionnaire de tâches** extrait une tâche d'un message reçu ou d'une perception de la grille et tente de l'ajouter à celles déjà prévues. Si la réponse à cette tentative est positive, l'agent accepte de réaliser la tâche,

sinon il refuse. Le composant **Ordonnanceur** gère les demandes d'ajout ou d'annulation de tâches. L'ajout d'une tâche est acceptée par ce composant si l'agent peut la réaliser avant son échéance sans remettre en cause les autres tâches. En plus de ces composants, les agents autonomes contiennent un composant **Sélection utilitaire** qui reçoit les demandes d'ajout de tâches puis cherche l'ensemble de tâches qu'il est possible de réaliser et qui maximise ses bénéfices. Le dernier composant qui est utilisé pour cet exemple est le **Filtre normatif** qui vérifie que les demandes d'annulation de tâches ne violent pas la norme.

	`Ordonnanceur`	`Sélection utilitaire`	`Filtre normatif`
`PRIORITY`	4	6	5
`PROVIDED_ROLES`	Ordonnanceur		
`REQUIRED_ROLES`		Ordonnanceur	Ordonnanceur
`HANDLED_EVTS`	name : add task type : add task reply_with : {confirm, reject} cont_class : Task	name : add task ... name : remove task ...	name : remove task type : remove task reply_with : {confirm, reject} cont_class : Task
	name : remove task type : remove task reply_with : {confirm, reject} cont_class : Task	name : confirm type : confirm name : reject type : reject	
`SENT_EVTS`	name : confirm type : confirm	name : add task ...	name : reject type : reject
	name : reject type : reject	name : remove task ...	
		name : confirm type : confirm	
		name : reject type : reject	

Tableau 2. *Interfaces des composants Ordonnanceur, Sélection utilitaire et Filtre normatif*

Le tableau 2 décrit quelques parties des interfaces des trois derniers composants. L'interface du composant `Ordonnanceur` montre qu'il accepte des événements de types `add task` ou `remove task` auxquels il peut répondre par un accord (`confirm`) ou un refus (`reject`). Le composant `Sélection utilitaire` accepte en entrée ces quatre types d'événements et peut aussi les émettre. Grâce à une plus grande priorité, ce composant filtre toutes les entrées/sorties du composant `Ordonnanceur`. Enfin le composant `Filtre normatif` ne réceptionne que les événements de type `remove task`. Si l'évenement viole la norme, il le consomme et répond par un refus `reject`. Sinon il laisse le composant de priorité inférieure recevoir cette demande d'annulation. On peut également noter que le rôle `Ordonnanceur` fourni par le composant de même nom est requis par les deux autres. De cette manière, le concepteur des deux autres

composants définit une dépendance dans le sens où tout assemblage qui contient l'un d'entre eux sans aucun composant fournissant le rôle `Ordonnanceur`, est incohérent.

Le noyau d'agent s'appuie sur ces interfaces pour gérer le routage d'évenements entre composants comme s'ils étaient connectés. La figure 2 illustre les échanges d'événements calculés par le noyau dans le cas d'un agent normatif autonome.

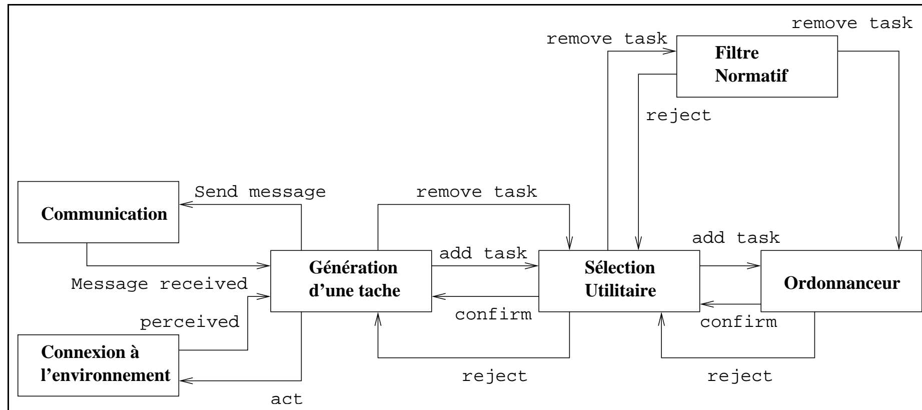


Figure 2. *Échange d'événements pour un agent normatif autonome*

Cette figure ne montre les échanges d'événements qu'à un instant donné. L'intérêt du modèle de composant présenté ici est que le noyau d'agent gère dynamiquement les modifications du contenu de l'assemblage. Par exemple, si l'on souhaite que l'agent ne respecte plus la norme, il suffit de supprimer le composant `Filtre normatif`. Le noyau enverrait alors directement les événements `add task` issus de la `Sélection utilitaire` à l'`Ordonnanceur`. On peut même transformer l'agent en simple agent social en supprimant également le composant `Sélection utilitaire`. Les transitions inverses (passer d'un agent social à un agent social autonome puis à un agent normatif autonome) sont réalisables par le simple ajout des composants nécessaires.

5. Travaux apparentés

De nombreux travaux se sont intéressés au développement d'applications multi-agents ainsi qu'à la programmation par composants. Il existe déjà un grand nombre d'environnements et de modèles de composants différents et même certaines propositions qui abordent en même temps ces deux domaines. Nous pensons cependant que le modèle de composant décrit ici et implémenté dans MAST est original et se démarque des outils existants.

5.1. *Encore un autre environnement de développement ?*

Un des objectifs avoués de la plupart des environnements de développement est la généricité [GUE 01]. Mais il n'existe pas encore de modèle multi-agent standard qui couvre tous les types d'application et les environnements existants (Jack [HOW 01], MadKit [GUT 00], Zeus [NWA 99]) implémentent tous un modèle multi-agent spécifique. Ils restent donc dépendants de l'adéquation du modèle sous-jacent avec l'application ciblée. Même les plates-formes multi-agents de plus bas niveau (Jade [BEL 03], SACI [HUB 02]) sont dédiées à un langage d'interaction particulier.

L'approche à base de composants utilisée positionne MAST comme un environnement "multi-modèles". C'est-à-dire que le noyau minimal de MAST n'implémente aucun modèle multi-agent particulier et reste très général. Ce noyau est principalement un support sur lequel peuvent se greffer des composants implémentant des modèles multi-agents spécifiques. MAST permet ainsi d'utiliser des modèles différents sous réserve qu'ils soient implémentés dans sa bibliothèque de composant. Cette bibliothèque se remplit incrémentalement au fur et à mesure des besoins des utilisateurs actuels de MAST car le même modèle de composant est utilisé pour les composants génériques et ceux plus spécifiques à une application facilitant ainsi le changement de statut d'un composant (rendre générique une partie d'un composant spécifique).

Quelques environnements utilisent une programmation par composants, mais soit ces composants sont de taille importante (un unique composant par dimension Voyelles pour MASK [OCC 02] et Volcano [RIC 02]) les rendant peu génériques, soit ils sont attachés à un modèle multi-agent (une architecture à base de comportements pour DIMA [GUE 99]). L'approche à base de composants présentée ici est proche de celle adoptée dans MALEVA [MEU 01]. Les modèles de composant MALEVA et MAST ont le même objectif - la conception d'un agent - et manipulent des composants de taille similaire. MAST se distingue principalement de MALEVA par le modèle de composant mis en oeuvre. En effet MALEVA utilise un modèle "classique" pour lequel il est nécessaire que les composants soient connectés à l'assemblage alors que le modèle MAST permet d'établir une connexion automatique et flexible. On peut enfin citer le projet MIMOSA [MUL] qui utilise également un modèle de composant pour un environnement de développement mais orientée vers les applications de simulation.

5.2. *Encore un autre modèle de composant ?*

Dans le domaine de la programmation par composants, il existe également plusieurs modèles (Open CCM [MER 03], EJB [EJB], ArchJava [ALD 02], ...) et une initiative récente du consortium ObjectWeb a pour objectif d'unifier ces approches en un standard : Fractal [BRU 02]. Mais aucun de ces modèles ne satisfait tous les besoins spécifiques à la construction d'agents telle qu'elle est présentée dans cet article.

Le principal obstacle réside dans les interactions entre composants. La grande majorité des modèles attribuent à chaque composant des ports en entrée et en sortie. C'est

au moment de l'assemblage que le concepteur doit connecter précisément des ports de sortie de certains composants aux ports d'entrée d'autres composants. Il faut donc que le concepteur ait une connaissance précise des entrées/sorties des composants sélectionnés et les connexions ainsi établies sont souvent rigides. Ce mode d'interaction n'est pas assez flexible dans notre cas pour deux raisons :

– Le concepteur de l'application (qui effectue l'assemblage) ne doit avoir besoin que d'une connaissance superficielle des entrées/sorties des composants sélectionnés. C'est le concepteur du composant qui est le mieux placé pour définir la manière dont il interagit avec d'autres composants.

– Une fois l'assemblage établi, les connexions doivent pouvoir être automatiquement définies et remises en cause dans certains cas.

6. Conclusion

Cet article décrit un modèle de composant adapté pour concevoir des agents par assemblage de composants. Ce modèle est suffisamment expressif pour qu'une partie de l'assemblage (les connexions entre composants) soit réalisée automatiquement, ce qui permet de distinguer deux niveaux de concepteurs : le concepteur de composants qui implémente des modèles multi-agents sous la forme de composants et le concepteur de l'application qui n'a besoin que d'une connaissance superficielle des concepts manipulés pour construire son application. Ce modèle de composant orienté agent s'intègre dans un modèle de composant orienté système (où chaque agent est un composant du système) évoqué de manière plus générale et schématique. Une implémentation de ce modèle de composant a été réalisée au sein de l'environnement de développement MAST librement téléchargeable à l'URL : http ://www.emse.fr/~vercoute/mast.

7. Bibliographie

[ALD 02] ALDRICH J., CHAMBERS C., NOTKIN D., « ArchJava : Connecting Software Architecture to Implementation », *Proceedings of the International Conference on Software Engineering*, Orlando, Florida, May 2002.

[BEL 03] BELLIFEMINE F., CAIRE G., POGGI A., RIMASSA G., « JADE, A white paper », September 2003, http ://exp.telecomitalialab.com/upload/articoli/V03N03Art01.pdf.

[BOI 03] BOISSIER O., Contrôle et Coordination Orientés Multi-Agents, Saint-Étienne, France, Février 2003, HDR de l'Univ. Jean Monnet et de l'École des Mines de St-Étienne.

[BOI 04] BOISSIER O., GITTON S., GLIZE P., « Caractéristiques des systèmes et des applications », *Systèmes Multi-agents, Observatoire Français des Techniques Avancées (OFTA)*, chapitre 1, p. 25-54, Éditions TEC & DOC, Février 2004.

[BOU 00] BOURNE R., EXCELENTE-TOLEDO C., JENNINGS N., « Run-time selection of coordination mechanisms in multi-agent systems », *14th European Conf. on Artificial Intelligence (ECAI-2000)*, Berlin , Germany, 2000, p. 348-352.

[BRU 02] BRUNETON E., COUPAYE T., STEFANI J.-B., « Recursive and Dynamic Software Composition with Sharing », *WCOP02*, June 2002.

[DEM 01] DEMAZEAU Y., VOYELLES, Grenoble, France, Avril 2001, Habilitation à Diriger les Recherches de l'INP Grenoble, Laboratoire LEIBNIZ.

[EJB] « Enterprise JavaBeans », http ://java.sun.com/products/ejb/index.jsp.

[FIP] « Foundation for Intelligent Physical Agents », http ://www.fi pa.org.

[GUE 99] GUESSOUM Z., BRIOT J.-P., « From Active Objects to Autonomous Agents », *IEEE Concurrency 7,(3)*, , 1999, p. 68-76.

[GUE 01] GUESSOUM Z., OCCELLO M., « Environnements de Développement », BRIOT J.-P., DEMAZEAU Y., Eds., *Principes et architectures des systèmes multi-agents*, p. 177-206, Hermès Sciences Publications, Paris, France, Décembre 2001.

[GUT 00] GUTKNECHT O., FERBER J., MICHEL F., « MadKit : Une expérience d'architecture de plate-forme multi-agent générique », PESTY S., SAYETTAT C., Eds., *JFIAD-SMA'00*, St Jean-la-Vêtre, Loire, France, Octobre 2000, Hermès, p. 223-236.

[HOW 01] HOWDEN N., RÖNNQUIST R., HODGSON A., LUCAS A., « JACK- Summary of an Agent Infrastructure », *5th Int. Conf. on Autonomous Agents*, Montreal, Canada, 2001.

[HUB 02] HUBNER J. F., SICHMAN J. S., BOISSIER O., « Spécifi cation structurelle, fonctionnelle et déontique d'organisations dans les Systèmes Multi-Agents », MATHIEU P., MULLER J.-P., Eds., *JFIADSMA'02*, Lille, France, 2002, p. 205-216.

[MER 03] MERLE P., « OpenCCM : The Open Source CORBA Components Platform », *ObjectWeb Conference*, Rocquencourt, France, November 2003.

[MEU 01] MEURISSE T., BRIOT J.-P., « Une approche à base de composants pour la conception d'agents », *TSI, Numéro spécial Réutilisabilité*, vol. 20, n° 4, 2001, p. 583-602.

[MUL] MULLER J.-P., « Projet MIMOSA », http ://lil.univ-littoral.fr/Mimosa/.

[NWA 99] NWANA H., NDUMU D., LEE L., COLLIS J., « Zeus ; A Tool-Kit for Building Distributed Multi-Agent Systems », *Applied Artificial Intelligence Journal*, vol. 13, n° 1, 1999, p. 129-186.

[OCC 02] OCCELLO M., BAEIJS C., DEMAZEAU Y., KONING J.-L., « MASK : An AEIO Toolbox to Develop Multi-Agent Systems », *Knowledge Engineering and Agent Technology, IOS Series on Frontiers in AI and Applications*, Amsterdam, The Netherlands, 2002.

[RIC 02] RICORDEL P.-M., DEMAZEAU Y., « La plate-forme VOLCANO : modularité et réutilisabilité pour les systèmes multi-agents », *Numéro spécial sur les plates-formes de développement SMA. Revue Technique et Science Informatiques (TSI)*, , 2002.

Déploiement et adaptation de systèmes P2P : une approche à base d'agents mobiles et de composants

Sébastien LERICHE — Jean-Paul ARCANGELI

IRIT-UPS, 118 route de Narbonne 31062 Toulouse Cedex 4

{leriche, arcangel}@irit.fr

RÉSUMÉ. *Les systèmes « pair à pair » (P2P) sont des systèmes complexes décentralisés et ouverts. Leur construction est difficile car elle doit prendre en compte en particulier les évolutions et la volatilité des ressources et des services ainsi que leur localisation. Dans ce papier, nous proposons une approche à base de composants logiciels dont le déploiement, l'adaptation dynamique et la localisation sont assurés par des agents mobiles adaptables. A des fins de validation, un prototype de système pair à pair a été développé sur le modèle proposé en s'appuyant sur le middleware JavAct.*

MOTS-CLÉS : *agents mobiles, composants, adaptation, architecture logicielle, systèmes pair à pair, répartition à grande échelle, grille*

1. Introduction

Popularisés par différents logiciels de partage et d'échange de fichiers, les systèmes « pair à pair » (*peer-to-peer* ou P2P) [MIL 02] reposent sur le principe de mutualisation de ressources (par exemple des données, des programmes, des services, des capacités de stockage ou de calcul) à l'échelle de l'Internet. Le modèle P2P est une alternative au modèle client-serveur : les pairs, qui sont tous au même niveau, peuvent à la fois offrir (rôle serveur) et demander (rôle client) des ressources. Le modèle P2P semble particulièrement intéressant pour les systèmes d'information répartis à grande échelle qui, par nature, sont hétérogènes et instables.

En mode P2P pur, il n'existe pas de point de centralisation ni d'administration globale ; au contraire, les systèmes sont complètement décentralisés et d'une certaine manière auto-organisés. Les pairs peuvent librement intégrer et quitter le système (ou tomber en panne), et fournir les ressources de manière intermittente et dans une forme

susceptible d'évoluer au cours du temps. L'instabilité des systèmes et la volatilité des ressources sont donc fortes et incontrôlées et, du fait de l'échelle, la communauté de pairs ne peut pas être informée des changements. Ainsi, lorsqu'un pair veut accéder à une ressource, il n'a *a priori* qu'une connaissance partielle (qui peut être en partie obsolète) de l'état du système P2P (de la disponibilité des serveurs et de la qualité des services).

Pour être viable, un tel système doit être flexible et capable de s'adapter dynamiquement, de manière transparente pour l'utilisateur. En outre, il doit permettre à un pair non seulement de localiser les ressources qu'il recherche mais aussi de découvrir des méta-informations sur le système (sur les pairs et les ressources) afin de mettre à jour ses connaissances sur le réseau P2P. Enfin, il doit offrir la possibilité d'exécuter des traitements personnalisés (expertise et besoins spécifiques du client), ceci sur les sites serveurs afin de limiter le volume des données échangées sur le réseau.

Nous nous intéressons au développement, au déploiement et à la maintenance des systèmes P2P purs et nous étudions des solutions permettant d'en limiter la complexité (et les coûts). Outre les problèmes d'hétérogénéité matérielle et logicielle, de sécurité, de performance, la construction de systèmes P2P doit prendre en compte les besoins d'adaptation et de personnalisation. Pour cela, nous proposons ici une approche à base d'agents mobiles et de composants logiciels.

Dans la section 2, nous examinons les caractéristiques des systèmes P2P et nous exhibons leurs principaux composants. Dans la section 3, nous présentons les grandes lignes de notre architecture de système P2P dans laquelle les agents mobiles sont les vecteurs du déploiement et de l'adaptation des composants P2P. Dans la section 4, nous rendons compte du développement d'un prototype de système pair à pair sur le modèle proposé, au moyen du *middleware* JAVACT à base d'agents mobiles adaptables. Enfin, en section 5, nous concluons et nous discutons les perspectives de ce travail.

2. Architecture de composants pour les systèmes P2P

2.1. *Localisation des ressources*

Même si les échanges de données et plus généralement l'exploitation des ressources se font directement entre pairs sans intermédiaire, on peut distinguer trois types d'architecture de systèmes P2P, selon le mode de mise en relation entre demandeur et fournisseur.

Les systèmes les plus simples, à la limite du P2P et du client-serveur, sont dits *centralisés* (Napster[1]...) et se basent sur un serveur unique pour indexer les données. A chaque connexion d'un pair, ce dernier annonce à l'index la liste des ressources qu'il met à disposition de la communauté.

1. http://www.napster.com

Les systèmes *décentralisés* sont apparus, pour augmenter la robustesse du serveur (panne, goulot d'étranglement). Certains (Freenet[2], Chord...) conservent un index qui est réparti sur l'ensemble des pairs (table de hashage distribuée). Mais si l'un d'eux devient indisponible, une partie de l'information ne peut plus être retrouvée. D'autres (Gnutella[3], PeerCast...) préfèrent se passer d'index en découvrant dynamiquement les ressources de leurs voisins (de façon récursive), ce qui leur permet un fonctionnement dans des réseaux désorganisés ou en constante évolution.

Enfin, les systèmes *hybrides* (FastTrack/Kazaa, Bearshare, eDonkey...) utilisent des *superpeers*, des pairs spécialement choisis pour leur capacité de calcul et de bande passante, dont le rôle est d'indexer les ressources d'un sous-ensemble des pairs du réseau.

Nous nous intéressons ici plus particulièrement aux systèmes P2P décentralisés, parce qu'ils sont les mieux adaptés à la mise à disposition de ressources volatiles ou évolutives et aux systèmes à grande échelle et à topologie instable.

2.2. *Composants de systèmes pair à pair*

Nous proposons d'identifier un ensemble de constituants élémentaires dont chaque pair doit disposer, afin de définir une architecture générique de système P2P.

En considérant les besoins minimaux, chaque pair doit pouvoir accéder à d'autres pour y découvrir les ressources accessibles, afin de les utiliser. La localisation dans les systèmes P2P purs ne passe pas par l'utilisation de serveurs ; il y a donc au départ un besoin de découverte et de localisation des pairs serveurs, puis ensuite sur ces pairs un besoin d'extraction des ressources et services. Pour cela nous distinguons deux composants de recherche. L'un de **localisation** pour la découverte et l'accès aux pairs (déplacement de l'agent selon différents algorithmes), l'autre de **recherche locale** sur le pair serveur qui correspond à divers algorithmes de fouille locale[4].

La localisation des ressources s'appuie un élément d'**information** qui répertorie les connaissances des pairs sur eux-mêmes et sur le réseau P2P (méta-informations).

Après avoir découvert une ressource, il faut pouvoir l'exploiter. Il peut s'agir par exemple de communiquer à l'utilisateur la description de la ressource trouvée, de télécharger un fichier, d'exécuter un service puis de transmettre les résultats, de lancer une nouvelle recherche si la ressource trouvée dépend d'une autre, ou plus généralement de superviser l'exploitation d'une ressource. Ces opérations sont à la charge d'un composant dédié à **l'exploitation des ressources**.

De plus, dans un contexte de ressources volatiles et évolutives, un composant supplémentaire fournissant un moyen d'accès uniforme aux ressources (accès à une base

---

2. http://freenet.sourceforge.net
3. http://rfc-gnutella.sourceforge.net/developer/stable
4. Ces composants jouent un rôle équivalent au composant d'allocation de ressources dans les systèmes centralisés

de données, à un système de fichiers, à un web service. . .) peut être ajouté. De cette façon par exemple, si la forme d'un web service change (nouvelle version) seul le composant d'**accès aux ressources** doit être modifié. Des systèmes comme Spitfire[5], utilisé dans le projet DATAGRID pour l'accès uniforme aux bases de données, existent déjà.

Cet ensemble de constituants génériques peut être implanté sous forme de **composants logiciels**, afin de simplifier la construction de différentes applications sur le modèle P2P, du partage de fichier à la recherche et l'exécution de services.

En complément, la personnalisation des composants est nécessaire pour pouvoir faire d'une part des recherches évoluées propres à chaque client, d'autre part des traitements spécifiques (expertise du client) côté serveur. Pour cela, les composants doivent pouvoir être déployés depuis le site du pair client sur le site du pair serveur.

2.3. *Plateformes de développement P2P existantes*

Nous avons étudié quelques environnements de développement susceptibles de permettre la construction d'applications P2P, dont les deux derniers utilisant le paradigme agent, afin d'étudier leur adéquation à ce contexte et de situer notre approche.

1) **XtremWeb**[6] est une plateforme pour la distribution de tâches en mode P2P sur des grilles de calcul, utilisée dans le cadre de l'ACI GRID. Elle a été conçue pour la distribution de calculs sur des pairs qui peuvent être demandeurs ou fournisseurs de ressources de calcul. La distribution est par nature centralisée même si une hiérarchie de serveurs permet de réduire la charge sur le serveur central.

2) **Globus** est une alliance pour le développement de technologies autour de la grille de calcul ouverte. Le Globus Toolkit[7] est une implémentation en logiciel libre du standard OGSI (Open Grid Services Infrastructure), un modèle de Web Services adaptés à la grille. Cette boîte à outils offre les services et les outils de base requis pour construire une grille de calcul (sécurité, localisation des ressources, gestion des ressources, transmissions). La lourdeur du framework et les difficultés de déploiement le rendent moins adapté aux systèmes P2P les plus dynamiques.

3) **JXTA**[8] est une initiative de Sun Microsystems, proposant un ensemble de protocoles ouverts pour interconnecter des dispositifs allant du téléphone cellulaire jusqu'aux serveurs. Le fonctionnement repose sur le découpage de parties du réseau réel en réseaux virtuels, dans lesquels chaque pair peut accéder aux ressources des autres sans se préoccuper de leur localisation ou de leur environnement d'exécution. JXTA propose une base de six protocoles (services de découverte des pairs, de rendez-vous, d'informations sur les pairs, de communication, de routage et de résolution des pairs) dans lesquels la communication se fait par des messages XML. Plusieurs implémen-

5. http://edg-wp2.web.cern.ch/edg-wp2/spitfire
6. http://www.lri.fr/~fedak/XtremWeb
7. http://www-unix.globus.org/toolkit
8. http://www.jxta.org

tations existent, par exemple en Java avec JXME[9]. Cette approche semble intéressante dans le contexte visé, néanmoins elle nous semble plutôt réservée à l'exécution de services distants, sans possibilité de personnalisation.

4) **Jade**[10] (Java Agent DEvelopment Framework) est un framework Java pour l'implémentation de systèmes multi-agents au dessus d'un *middleware* conforme aux standards FIPA[11], notamment pour permettre l'interopérabilité entre agents. Elle permet de réaliser des systèmes P2P dans lesquels chaque pair peut fonctionner de manière *proactive*, communiquer sans se soucier de la localisation, et se coordonner pour résoudre un problème complexe grâce à l'utilisation d'agents. Un agent spécial unique (Agent Managing System ou AMS) sert de superviseur pour la plateforme, et fournit les services d'annuaire, de cycle de vie [BEL 04]. Tout agent créé doit s'enregistrer auprès de l'AMS pour obtenir son identifiant, indispensable à la communication. Ce point de centralisation est un frein à l'adoption de Jade dans des systèmes répartis à grande échelle, et restreint son usage à des clusters de machines ou des réseaux de petite taille.

5) Un système **Anthill**[12] est un ensemble de pairs sur lesquels est déployé un système multi-agents, dont l'interaction permet de résoudre des problèmes complexes (grâce à des comportements émergents et des algorithme génétiques) comme par exemple le routage des messages. Anthill s'inspire des colonies de fourmis, en proposant des agents aux comportements simples, autonomes, et pourvus d'un environnement sur lequel ils basent leurs actions [BAB 02]. Chaque pair est un *nid*, qui fournit à l'application des services spécifiques au P2P : gestion de ressources, communication, gestion de la topologie, planification des actions. La dernière implémentation (version 1.1 de 2002) repose sur JXTA pour tout ce qui concerne la répartition. Les auteurs proposent également un logiciel pour le partage de fichiers (Gnutant), compatible avec Gnutella et Freenet. Ils montrent que le développement d'une telle application est simplifié par l'usage de leur système.

Hormis JXTA, aucune de ces plateformes n'est vraiment adéquate pour la construction de systèmes P2P purs. En particulier, la plupart des *middleware* pour la grille reposent sur des services centralisés (éventuellement hiérarchisés) d'allocation de ressources à des tâches indépendantes. Nous proposons une approche qui se démarque des technologies citées ci-dessus par la décentralisation, l'utilisation de composants génériques et la personnalisation des services au moyen d'agents mobiles.

9. http://platform.jxta.org
10. http://jade.tilab.com
11. http://www.fipa.org
12. http://www.cs.unibo.it/projects/anthill

3. Déploiement et adaptation par les agents mobiles

3.1. *Définitions*

Un *agent logiciel* est une entité autonome capable de communiquer, disposant de connaissances et d'un comportement privés ainsi que d'une capacité d'exécution propre. Un agent agit pour le compte d'un tiers (un autre agent, un utilisateur) qu'il représente sans être obligatoirement connecté à lui.

Un *agent mobile* [FUG 98, BER 02] est un agent logiciel qui peut se déplacer d'un site à un autre en cours d'exécution pour se rapprocher de données ou de ressources. Il se déplace avec son code et ses données propres, mais aussi avec son état d'exécution. L'agent décide lui-même de manière autonome de ses mouvements. La mobilité ne se substitue pas aux capacités de communication des agents mais les complète (la communication distante, moins coûteuse dans certains cas, reste possible). Afin de satisfaire aux contraintes des réseaux de grande taille ou sans fil (latence, non permanence des liens de communication), les agents communiquent par messages asynchrones. Le principal apport des agents mobiles se situe sur un plan du génie logiciel [CHE 94] : ils unifient en un modèle unique différentes technologies de traitement réparti (client-serveur, appel de procédure à distance, code mobile...). Nous considérons que, par nature, ils constituent un outil privilégié pour le déploiement.

En complément, un agent logiciel est dit *adaptable* si certains de ses mécanismes internes, opérationnels (envoi de messages, déplacement...) ou fonctionnels (comportement), sont modifiables en cours d'exécution. Conformément à la propriété d'autonomie, l'agent contrôle lui-même ses propres évolutions.

Un *composant* [SZY 02, MAR 02] est un morceau de logiciel « assez petit pour que l'on puisse le créer et le maintenir, et assez grand pour qu'on puisse l'installer et le réutiliser ». Il possède des interfaces qui spécifient la façon de le composer (services offerts et requis). Chaque modèle de composant (EJB, CCM, Fractal...) rafine cette définition et précise les règles d'assemblages des composants. Un environnement d'exécution (ou plateforme à composants) fournit un ensemble de services permettant leur instanciation et leur exécution.

3.2. *Des composants aux agents mobiles*

Il est possible de faire une analogie entre les modèles de composants et ceux d'agents mobiles, particulièrement si l'on considère la séparation des préoccupations. Tout d'abord, les composants comme les comportements des agents contiennent le code fonctionnel (code métier). Par ailleurs, le conteneur permet au composant d'instancier, d'activer et d'accéder aux services non fonctionnels fournis par l'environnement d'exécution (par exemple la gestion du cycle de vie, de la sécurité ou des communications).

Nous avons proposé une architecture d'agent mobile adaptable [LER 04a] dans laquelle la réification sous forme de micro-composants d'un certain nombre de mécanismes internes et élémentaires des agents les rend interchangeables. Ils correspondent chacun à un mécanisme d'exécution non fonctionnel activé par délégation (boîte aux lettres, déplacement...). Dans notre modèle, l'agent joue le rôle de conteneur : ses micro-composants enveloppent le comportement et implantent les mécanismes non-fonctionnels (envoi des messages, réception, cycle de vie...) ou éventuellement les délèguent au système d'accueil.

De fait, dans ce qui suit nous proposons d'utiliser des composants (sans supposer de modèle particulier) sous la forme de comportements d'agents, afin de bénéficier des apports des agents pour le déploiement et l'adaptation.

3.3. *Déploiement des composants*

Notre idée est de permettre le déploiement des composants proposés au 2.2 depuis les sites clients vers les sites serveurs, en utilisant la mobilité de code et d'agent. Le principe est schématisé dans la figure 1 (pour une meilleure lisibilité, on a distingué le pair jouant le rôle de client de celui jouant le rôle de serveur mais ils sont bien sûr complètement symétriques).

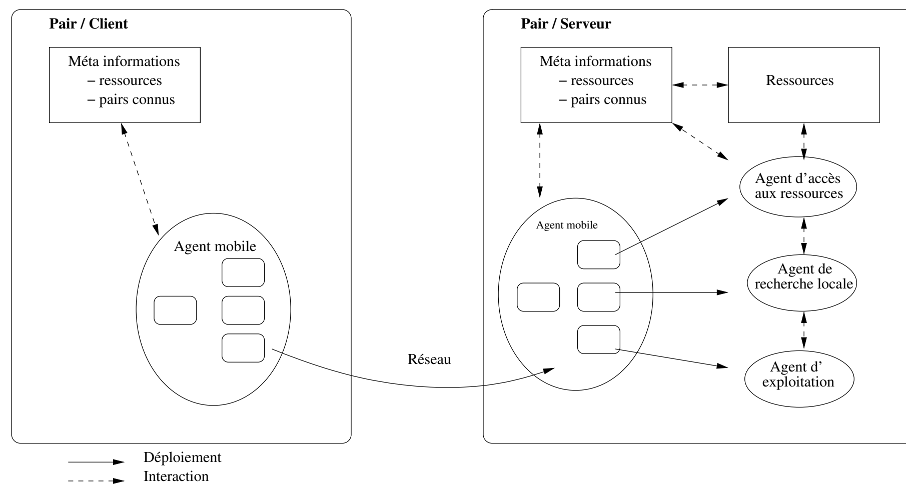


Figure 1. *Déploiement des composants P2P*

Pour chaque requête d'un client, un agent mobile (ou plusieurs pour exploiter le parallélisme) transporte les composants sur le réseau. Le composant de localisation constitue l'essentiel de son comportement. Cet agent mobile se déplace de pair en pair en effectuant dynamiquement la localisation, ce qui lui permet d'adapter la recherche

à l'état courant du système P2P. Ainsi, l'agent mobile est le vecteur du **déploiement adaptatif** des composants.

Lorsqu'il arrive à destination, l'agent mobile installe un mini système multi-agent déployé à partir des composants, pour traiter la requête localement. Là, chaque composant est encapsulé dans un agent dont il constitue le comportement et à travers lequel il offre ses services. Inversement, l'agent sert de conteneur et d'activateur au composant et lui fournit les mécanismes non fonctionnels nécessaires à son exécution.

De manière générale, les composants peuvent être fournis par le pair client, par le pair serveur ou par un tiers. Ainsi, le composant d'accès aux ressources peut ne pas être systématiquement fourni par le serveur. Mais en cas d'*upgrade* côté serveur (évolution du mécanisme d'accès aux ressources), il faut prévoir un protocole permettant l'acquisition *in situ* et à la volée du nouveau composant d'accès. Celui-ci doit pouvoir être authentifié (signature du fournisseur) pour éviter les problèmes de sécurité. Ceci contribue à la robustesse de la recherche, permet l'adaptation dynamique aux évolutions des serveurs et limite les opérations de maintenance des serveurs à un cadre local.

3.4. *Adaptation individuelle des composants*

Du fait de leur mobilité, les agents sont amenés à se déplacer sur des systèmes hétérogènes, sur lesquels il est souvent impossible de faire d'hypothèses sur la qualité de service du réseau, de la disponibilité, etc. Pour permettre aux composants embarqués de fonctionner de façon optimale, les agents doivent donc adapter leurs mécanismes internes (envoi de messages, réception de messages, cycle de vie. . .) de manière statique ou dynamique.

Notre architecture d'agent mobile adaptable permet d'obtenir la flexibilité et l'extensibilité nécessaire à l'adaptation : adapter un agent revient à changer un ou plusieurs de ses micro-composants. L'agent est ainsi le vecteur de l'**adaptation individuelle** des composants. Cette forme d'adaptation est complètement séparée de la programmation des composants, ce qui contribue à simplifier leur développement.

3.5. *Exploitation des méta-informations*

En parcourant le réseau et en explorant des zones primitivement inconnues du client, il est possible de glaner des connaissances sur les sites visités lors de la phase de recherche, en accédant à leurs méta-informations (nature des ressources, connaissance d'autres pairs). Ces éléments peuvent alimenter la politique de déplacement de l'agent, afin d'obtenir un itinéraire de recherche dynamique.

D'autre part, l'un des rôles du composant de recherche globale (éventuellement par l'intermédiaire d'un composant de glanage) est de transmettre l'information glanée pour mettre à jour du côté du client l'élément d'information (base de donnée qui

répertorie les propriétés des serveurs distants connus et alimente les futures sélections de serveur). Plus le client dispose d'informations sur d'autres sites, plus les chances de trouver l'information recherchée augmente.

4. Le prototype JAVANE

JAVANE est un prototype de logiciel P2P basé sur l'architecture proposée, que nous conçu pour en vérifier la validité. Dans sa première version, JAVANE permet le partage, la recherche et l'échange de ressources réparties sur un réseau hétérogène de grande taille. Sur chaque site du réseau, JAVANE permet la publication de fichiers quelconques (son, image, texte, composant logiciel, paquetage...) rendus publics par leurs propriétaires. En complément, JAVANE permet à un utilisateur de rechercher des fichiers sur le réseau, puis par exemple de télécharger ceux qu'il aura retenus. Chaque site possède des connaissances propres sur l'existence d'autres sites serveurs, sur lesquels pourront s'effectuer des traitements. Les liens de connaissance entre sites peuvent évoluer au cours du temps.

4.1. *JAVACT*

La mise en œuvre repose sur JAVACT[13] [ARC 01, ARC 04], un *middleware* Java standard développé dans notre équipe. Outre un haut niveau d'abstraction par rapport à la répartition et aux opérations distantes, JAVACT offre un modèle de programmation à base d'agents mobiles ainsi que des mécanismes d'adaptation individuelle fins.

Une application JAVACT s'exécute sur un ensemble de places (des machines virtuelles Java) connectées en réseau, *via* un intergiciel de plus bas niveau (Java RMI dans la version standard actuelle) dont l'utilisation est cachée au programmeur de l'application. Sur les places, un ensemble d'agents coopèrent pour résoudre un problème commun. Pendant l'exécution, un agent peut s'interrompre, changer de place, puis reprendre son exécution sur une nouvelle place. Les agents JAVACT sont des acteurs [AGH 86], entités logicielles autonomes qui communiquent par messages asynchrones, traitent en série les messages reçus, et dont le comportement peut évoluer en cours d'exécution.

4.2. *Les composants dans JAVANE*

JAVANE a été entièrement implanté, testé et validé expérimentalement dans des configurations variées (réseaux internet et intranet, wifi, PC, stations Sun...). Au niveau de l'application, différents composants ont été écrits (voir [LER 04b] pour plus de détails). En pratique, il s'agit de comportement d'agents JAVACT, implémentant

13. JAVACT (cf. http://www.irit.fr/recherches/ISPR/IAM/JavAct.html) est distribué sous forme de logiciel libre sous licence LGPL

différents protocoles. Par exemple :
- composant de localisation : recherche par inondation (type Gnutella), par cheminement d'agent mobile et mixte
- composant d'exploitation : téléchargement de fichier
- composant de recherche locale : filtre de fichiers multicritères (taille, type, nom. . .)
- composant d'accès aux ressources : accès à une base de donnée au format texte

Au niveau du *middleware*, un ensemble de micro-composants (communication cryptée, tolérance aux déconnections, authentification. . .) ont également été écrits.

4.3. *L'expérience de développement*

L'expérience montre bien les avantages des agents mobiles dans ce contexte où, par nature, il n'y a pas de centralisation possible de l'information et des méta-données qui sont fortement instables et volatiles. Ici, la recherche s'adapte dynamiquement à l'état du système (liens entre les places, contenu) et de la recherche (résultats trouvés) sans interaction avec le client, et par le biais d'agents aux droits limités (donc *a priori* inoffensifs). En complément, la sélection personnalisée sur le site serveur limite le volume du trafic sur le réseau aux seules données sélectionnées.

Les versions successives de JAVANE ont été développées par des étudiants de niveau Master 1$^{\text{ère}}$ année (maîtrise d'informatique, 2$^{\text{ème}}$ année ENSEEIHT) dans le cadre de projets après quelques heures de formation au concept d'agent mobile, au modèle d'acteur et à la bibliothèque JAVACT. Deux avantages majeurs ont été constatés :

– Le modèle de programmation est relativement simple à appréhender (les étudiants l'ont confirmé) et la conception des protocoles à base d'agents mobiles est naturelle.

– En cachant les problèmes de répartition, de mobilité, de synchronisation et de communication, le *middleware* JAVACT simplifie le développement et le déploiement. Le passage au codage n'apporte pas de difficulté, ce qui augmente la fiabilité du logiciel. De plus, le volume de code écrit est relativement faible.

5. Conclusion et perspectives

Notre contribution repose sur l'utilisation d'agents mobiles adaptables pour permettre le déploiement et l'adaptation de composants dans le cadre de systèmes pair à pairs purs. D'une part l'encapsulation des composants par des agents mobiles simplifie le déploiement par délégation de la gestion de la mobilité et de l'installation au *middleware*. D'autre part les propriétés d'adaptation des agents et leur nature proactive donne la flexibilité nécessaire à la construction d'applications sur le modèle P2P pur. En particulier, les agents mobiles permettent la découverte personnalisée et adaptative d'informations et de pairs tout en limitant le trafic sur le réseau. Ainsi, ils permettent

de développer simplement un système réparti à grande échelle ouvert et robuste (face aux évolutions des serveurs et aux besoins des clients).

Nous avons expérimenté dans la première version de JAVANE l'architecture de composants sous la forme d'un prototype de partage de fichiers en P2P. Celui-ci peut être enrichi de diverses façons, via l'écriture de nouveaux composants : exécution de services distants (appels de procédures ou web services), mise à jour de composants distants (un fournisseur de composants peut mettre à jour les composants d'un logiciel chez les clients), ou encore téléchargement de composants pour réaliser un auto-déploiement. Ainsi, la dernière version de JAVANE qui en résulte est devenue un outil générique de construction d'applications réparties sur le modèle P2P, grâce à une architecture configurable à base de composants.

Ces caractéristiques semblent intéressantes dans des domaines autres que le P2P. Dans celui de la recherche d'information distribuée, notre approche semble complémentaire aux solutions à base d'agents et de systèmes multi-agent plus classiques [FAN 99, CAZ 02] et compatible avec des stratégies de recherche sophistiquées. Dans le domaine de la grille de calcul, la possibilité de rechercher des composants, de vérifier leurs dépendances et éventuellement de télécharger récursivement les composants requis semble intéressante. Plus généralement, nous considérons les agents mobiles comme un outil privilégié pour la recherche et le déploiement de services et de logiciels sur les grilles ; nous allons mener des travaux dans cette voie.

Certains problèmes restent ouverts. La sécurité est une préoccupation majeure (protection des serveurs et des agents), et il reste à étudier comment et à quels endroits intégrer les différentes solutions possibles (à base de cryptage, authentification, limitation de droits, etc.). A un autre niveau, pour améliorer la sûreté des systèmes, il est nécessaire d'introduire des mécanismes de validation de l'assemblage (dynamique) des composants de l'application comme du *middleware*.

6. Bibliographie

[AGH 86] AGHA G., *Actors : a model of concurrent computation in distributed systems*, M.I.T. Press, Cambridge, Ma., 1986.

[ARC 01] ARCANGELI J.-P., MAUREL C., MIGEON F., « An API for high-level software engineering of distributed and mobile applications », *8th IEEE Workshop on Future Trends of Distributed Computing Systems , Bologna (It.)*, Los Alamitos, Ca., U.S.A., 2001, IEEE-CS Press, p. 155-161.

[ARC 04] ARCANGELI J.-P., HENNEBERT V., LERICHE S., MIGEON F., PANTEL M., « JAVACT 0.5.0 : principes, installation, utilisation et développement d'applications », rapport n° IRIT/2004-5-R, 2004, IRIT.

[BAB 02] BABAOGLU O., MELING H., MONTRESOR A., « Anthill : A Framework for the Development of Agent-Based Peer-to-Peer Systems », rapport n° UBLCS-2001-09, 2002, Dept. of Computer Science, Univ. of Bologna, Italy.

[BEL 04] BELLEFEMINE F., CAIRE G., TRUCCO T., RIMASSA G., « JADE programmer's guide (JADE3.2) », rapport, 2004, TILab S.p.A.

[BER 02] BERNARD G., ISMAIL L., « Apport des agents mobiles à l'exécution répartie », *Revue des sciences et technologies de l'information, série Techniques et science informatiques*, vol. 21, n° 6, 2002, p. 771-796, Hermès Science Publications, Lavoisier.

[CAZ 02] CAZALENS S., DESMONTILS E., JACQUIN C., LAMARRE P., « De la gestion locale à la recherche distribuée dans des sources d'informations et de connaissances », *Revue des sciences et technologies de l'information, série L'Objet*, vol. 8, n° 4, 2002, p. 47-69, Hermès Science Publications, Lavoisier.

[CHE 94] CHESS D., HARRISON C., KERSHENBAUM A., « Mobile Agents : Are They a Good Idea ? », rapport, 1994, IBM Research Division, New York.

[FAN 99] FAN Y., GAUCH S., « Adaptive Agents for Information Gathering from Multiple Sources », *AAAI Symposium on Agents in Cyberspace*, 1999, p. 40-46.

[FUG 98] FUGGETTA A., PICCO G., VIGNA G., « Understanding Code Mobility », *IEEE Transactions on Software Engineering*, vol. 24, n° 5, 1998, p. 342-361.

[LER 04a] LERICHE S., ARCANGELI J.-P., « Une architecture pour les agents mobiles adaptables », *Actes des Journées Composants JC'2004*, 2004.

[LER 04b] LERICHE S., ARCANGELI J.-P., PANTEL M., « Agents mobiles adaptables pour les systèmes d'information pair à pair hétérogènes et répartis », *Actes des NOuvelles TEchnologies de la REpartition*, 2004, p. 29-43.

[MAR 02] MARVIE R., PELLEGRINI M.-C., « Modèles de composants, un état de l'art », *Revue des sciences et technologies de l'information, série L'Objet*, vol. 8, n° 3, 2002, p. 61-89, Hermès Science Publications, Lavoisier.

[MIL 02] MILOJICIC D., KALOGERAKI V., LUKOSE R., NAGARAJA K., PRUYNE J., RICHARD B., ROLLINS S., XU Z., « Peer-to-Peer Computing », rapport n° HPL-2002-57, 2002, HP Laboratories Palo Alto.

[SZY 02] SZYPERSKI C., *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley / ACM Press, 2002.

Négociation de contrats – des systèmes multi-agents aux composants logiciels[1]

Hervé Chang — Philippe Collet

Équipe OCL, Objets et Composants Logiciels

I3S – CNRS – Université de Nice - Sophia Antipolis
Les Algorithmes, Bât. Euclide, 2000 route des Lucioles
BP 121, F-06390 Sophia Antipolis Cedex

{Herve.Chang,Philippe.Collet}@unice.fr

RÉSUMÉ. *L'approche contractuelle se justifie pleinement dans le développement logiciel à base de composants. Afin d'adapter cette approche à des modèles de composants hiérarchiques comme* `Fractal`*, nous avons conçu* `ConFract`*, un système qui gère des contrats, à la fois sur les connexions entre composants et sur les composants composites. Comme les reconfigurations dynamiques et la fluctuation des aspects non fonctionnels entraînent des remises en cause fréquentes des contrats, des mécanismes de négociation deviennent alors nécessaires. Dans cet article, nous présentons notre approche qui s'inspire des processus de négociation dans les systèmes multi-agents. Certains aspects sont ainsi repris et adaptés afin d'élaborer un modèle pour négocier des contrats logiciels. Nous décrivons aussi un mécanisme de négociation par relâchement, qui convient bien à des contrats comportementaux exprimés par des assertions exécutables.*

ABSTRACT. *The use of contracts justifies itself in component-based software development. In order to adapt contracts to hierarchical component models such as* `Fractal`*, we have proposed the* `ConFract` *system, which manages contracts both on bindings between components and on composite components. As dynamic re-configurations and fluctuations of nonfunctional aspects lead to frequent challenges of the contracts, some negotiation mechanisms are needed. This paper presents an approach inspired from negotiation processes in multi-agent systems. Some aspects are thus taken up and adapted in order to define a model for the negotiation of software contracts. We also describe a concession-based negotiation policy, which is well-suited to behavioral contracts based on executable assertions.*

MOTS-CLÉS : *génie logiciel des composants, négociation, contrat, composants hiérarchiques, systèmes multi-agents, Contract-Net Protocol, ConFract, Fractal.*

KEYWORDS: *Component-Based Software Engineering, Negotiation, Contract, Hierarchical Components, Multi-Agent Systems, Contract-Net Protocol, ConFract, Fractal.*

1. Cette recherche a été en partie financée par le contrat de recherche externe n° 422721832-I3S avec France Télécom R&D.

1. Introduction

Dans le domaine du génie logiciel, l'approche par composants est actuellement l'objet d'un intérêt croissant de la part de la communauté scientifique et des industriels. Cette approche présente notamment les avantages de mieux séparer interface et implémentation et de rendre l'architecture des applications explicite. Ainsi, un composant logiciel ne montre que ses interfaces requises et fournies, et des *contrats* basiques s'établissent lors des assemblages de composants. Il est maintenant reconnu que des contrats doivent prendre en compte des propriétés fonctionnelles plus finement décrites, ainsi que des propriétés non fonctionnelles (synchronisation, qualité de services, etc.) [BAC 00, SZY 02]. Cette notion de contrat devient primordiale lorsque les composants sont hiérarchiques [ALD 02, BRU 04]. En effet, comme un composant peut être créé par assemblage d'autres composants, des propriétés de ces assemblages et du composite résultant doivent aussi être exprimées et contractualisées.

Pour répondre à ce besoin, nous avons conçu `ConFract`, un système de contractualisation pour composants logiciels [COL 04b, COL 04a]. Son objectif est d'établir et de vérifier des propriétés fonctionnelles et non fonctionnelles sur des composants hiérarchiques en s'appuyant sur des *contrats d'interface*, établis entre chaque connexion, et des *contrats de composition*, qui supervisent le contenu des composants. La mise en œuvre de `ConFract` est basée sur la plate-forme `Fractal` [BRU 03], qui fournit notamment des possibilités de reconfiguration dynamique. Comme `ConFract` se doit aussi de prendre en compte des aspects non fonctionnels qui peuvent être très fluctuants, des modifications fréquentes et importantes interviennent sur les contrats. Des mécanismes de négociation sont alors nécessaires pour automatiser le rétablissement de contrats valides.

L'étude des travaux connexes nous a amené à nous inspirer des mécanismes de négociation dans les systèmes multi-agents. Nous définissons ainsi un modèle de négociation, dans lequel une négociation atomique est associée à chaque disposition de contrat en échec. Cette négociation atomique se base sur une version adaptée du *Contract-Net Protocol* [SMI 80, COM 02b] et différentes politiques de négociation sont potentiellement applicables. Nous décrivons ici une politique par relâchement, qui est bien adaptée à des contrats exprimés par des assertions exécutables.

L'article s'organise de la manière suivante. La section 2 décrit brièvement le système de contractualisation `ConFract` et définit notre problématique sur un exemple de lecteur multimédia. Dans la section 3, nous étudions la proximité des mécanismes de négociation dans les systèmes multi-agents et dans `ConFract`. La section 4 décrit le modèle de négociation résultant. Enfin, la section 5 conclut cet article.

2. Le système ConFract

`ConFract` se présente comme un système pour organiser sous forme de contrats la spécification et la vérification de propriétés fonctionnelles et non fonctionnelles sur des composants logiciels `Fractal`. À partir de spécifications, `ConFract` construit des contrats lors des assemblages de composants. Ces contrats sont alors des objets de

première classe pendant les phases de configuration et d'exécution des composants. Le système *ConFract* a été conçu pour séparer clairement les mécanismes contractuels de l'expression des spécifications. Ainsi, différents formalismes peuvent être pris en compte, mais actuellement, seul un langage d'assertions exécutables, nommé *CCL-J* (*Component Constraint Language for Java*), est intégré au système. Ce langage, partiellement inspiré d'OCL [OBJ 97], est dédié à Java, car l'implémentation actuelle de *ConFract* repose sur l'implémentation en Java de *Fractal* [BRU 04]. Les exemples de spécification présentés par la suite utilisent d'ailleurs le langage *CCL-J*. *ConFract* introduit différentes formes de contrat pour tenir compte des spécificités du modèle *Fractal*. En effet, un composant *Fractal* peut exposer plusieurs interfaces, requises ou fournies. Ces interfaces sont composées d'un nom et d'une signature[1]. Des interfaces requises et fournies compatibles peuvent ainsi être connectées pour établir des communications entre composants. De plus, certains composants peuvent être composites et contenir d'autres composants. En *Fractal*, des contrôleurs permettent de gérer les aspects techniques des composants tout en appliquant une approche par séparation des préoccupations. Les plus utilisés gèrent le cycle de vie (*LifecycleController*), les connexions (*BindingController*) et le contenu (*ContentController*), respectivement LC, BC et CC sur la figure 1.

Tout au long de cet article, nous nous appuierons sur un exemple de lecteur multimédia simplifié (cf. figure 1). Le composant de type FractalPlayer contient trois sous-composants : Player qui fournit exclusivement le service de lecture vidéo (méthode start) et gère ses paramètres par des attributs, GuiLauncher qui gère l'interface graphique et VideoConfigurator qui fournit des services pour optimiser la lecture vidéo (la méthode canPlay détermine par exemple si une vidéo peut être visualisée entièrement avec une dimension donnée, en tenant compte des ressources disponibles comme la batterie et la mémoire).

2.1. *Types de contrat*

Dans le système *ConFract*, deux grands types de contrat sont distingués. Un *contrat d'interface* est établi sur chaque connexion entre une interface requise et une interface fournie. Il regroupe les spécifications des deux interfaces, sachant que ces spécifications ne peuvent référencer que les méthodes de l'interface. Par exemple, une spécification *CCL-J* peut être associée à l'interface MultimediaPlayer (haut de la figure 1), pour décrire la précondition de la méthode start comme on le fait dans un système à objets. Un *contrat de composition*, quant à lui, est associé à la membrane d'un composant et regroupe toutes les spécifications définies sur cette membrane. On distingue d'une part un contrat de composition *externe*, dont les spécifications ne référencent que des interfaces visibles à l'extérieur de la membrane, et d'autre part, un contrat de composition *interne*, qui contient des spécifications qui référencent des interfaces dans le contenu du composant — soit des interfaces internes du composant, soit des interfaces externes de ses sous-composants —. Par exemple, la spécification

1. Cette signature peut être assimilée à une interface Java.

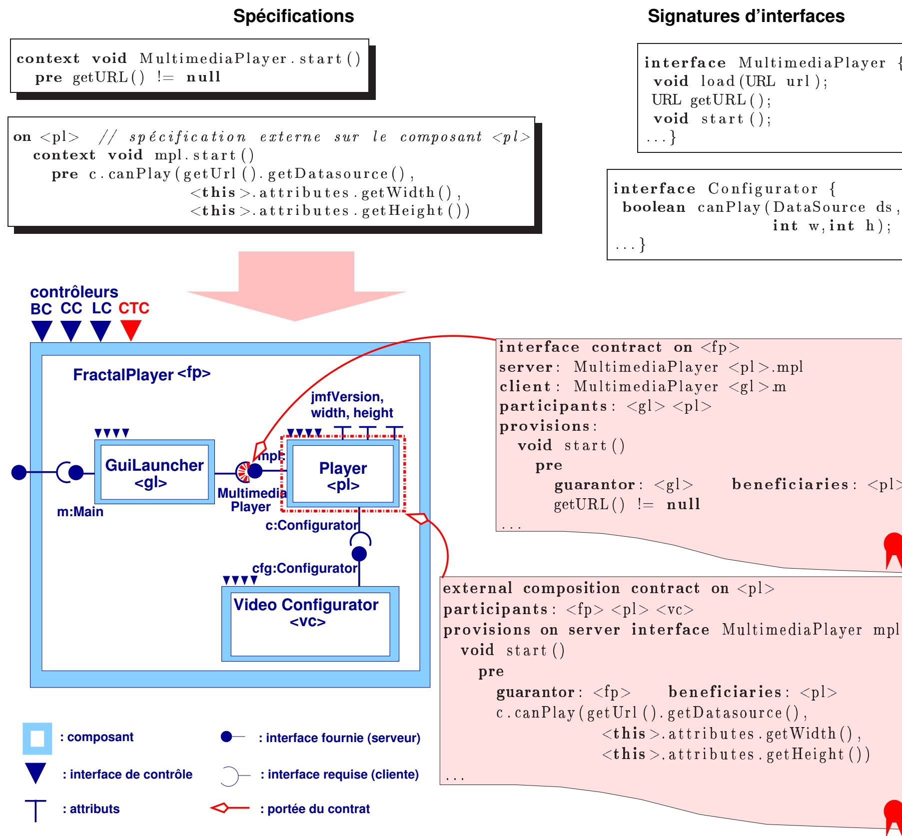


Figure 1 – Contractualisation du lecteur multimédia.

concernant le composant `<pl>` donnée dans la figure 1 fera partie du contrat de composition externe de ce composant. Elle exprime en effet une précondition sur la méthode `start` de l'interface *Fractal* nommée `mpl` ; et cette précondition référence une autre interface externe de `<pl>` — `c`, interface requise de type `Configurator` — pour exprimer la condition « *la source vidéo peut-elle être entièrement jouée dans les dimensions configurées ?* » C'est bien cette référence à plusieurs interfaces sur le même composant qui définit le contrat de composition externe.

Les contrats construits par le système *ConFract* regroupent les spécifications suivant leur catégorie et chaque clause de spécification devient alors une *disposition* du contrat. Ces contrats contiennent aussi les références aux composants participants à l'évaluation de chacune des dispositions. Un métamodèle décrit en effet chaque type de spécification, par exemple les préconditions d'une méthode dans une spécification externe de composant en *CCL-J*, avec les responsabilités attribués aux composants impliqués.

Ainsi, les responsabilités du contrat de composition externe du composant <pl> (fig. 1) sont données par la table suivante[2] :

Type d'interface	Construction	Garant	Bénéficiaires
serveur (mpl)	pre	<fp>	<pl>
serveur (mpl)	post	<pl>	<fp>, <gl>
cliente (c)	pre	<pl>	<fp>, <vc>
cliente (c)	post	<fp>	<vc>

Il est à noter que le garant de certaines conditions, comme la précondition sur une méthode serveur (interface mpl), est le composant englobant, ici <fp>, car il est le seul à voir cette précondition. En effet, le composant <gl>, utilisateur de l'interface mpl, n'est garant que du contrat d'interface sur la connexion. C'est bien le composant englobant qui est responsable de la bonne utilisation générale de ses sous-composants.

2.2. *Gestion des contrats*

Dans *ConFract*, les contrats sont gérés par des contrôleurs de contrats (CTC sur la figure 1) placés sur la membrane de chaque composant composite. Chaque contrôleur de contrats prend en charge le cycle de vie et l'évaluation *(i)* du contrat de composition interne du composite sur lequel il est placé, *(ii)* du contrat de composition externe de chacun des sous-composants et *(iii)* du contrat d'interface de chaque connexion dans son contenu. Le contrôleur de contrat utilise le mécanisme de mixin[3] de l'implémentation en Java de *Fractal* pour effectuer des intercessions sur les contrôleurs de cycle de vie, de connexion et de contenu. Il construit et met à jour les contrats en fonction des événements d'assemblage qui surviennent sur les composants (insertion d'un sous-composant, connexion de deux interfaces, etc.).

Les dispositions des contrats de configuration qui définissent des invariants sur les composants sont vérifiées à la fin de la phase de configuration. Dans le cas du langage *CCL-J*, toutes les autres dispositions sont vérifiées dynamiquement. Lorsqu'une méthode est appelée sur une interface *Fractal*, les préconditions du contrat d'interface sont d'abord évaluées, puis celles du contrat de composition externe du composant recevant l'appel, et enfin celles du contrat de composition interne. Des vérifications, dans l'ordre inverse, sont effectuées au retour de la méthode pour les postconditions et les invariants.

---

2. Nous ne montrons que les responsabilités de ce type de contrat pour des pré et postconditions.
3. Un mixin est une classe dont la super-classe est définie de manière abstraite par l'ensemble des primitives uniquement nécessaires ; les différents mixins d'un contrôleur sont ainsi fusionnés en une seule classe au chargement.

2.3. *Problématique*

Le système `ConFract` a pour objectif de prendre en compte des aspects fonctionnels et non fonctionnels. Par exemple, dans la figure 1, la précondition du contrat de composition externe établit les bonnes conditions de lecture de la source vidéo. Le composant `VideoConfigurator` va ainsi établir le résultat de la méthode `canPlay` en fonction de différents paramètres du système (niveau de la batterie) et de la source vidéo (complexité du décodage). La fluctuation de certaines propriétés non fonctionnelles peut ainsi couramment entraîner la violation de bon nombre de contrats. Actuellement, le système `ConFract` gère ces violations en notifiant le composant garant de la disposition en échec et en lui fournissant toutes les informations sur le contexte de la violation. Ceci implique la multiplication de codes de rattrapage pour rétablir des contrats valides alors qu'intuitivement, un grand nombre d'entre eux semblent négociables. Dans notre exemple, on pourrait consommer moins d'énergie en diminuant la taille de l'affichage, voire retirer la condition tout en sachant que la lecture pourra s'interrompre. La proximité des notions de contrat et de négociation, dans les domaines du génie logiciel et des systèmes multi-agents, nous a ainsi amené à étudier ce dernier domaine afin de concevoir des mécanismes de négociation.

3. Négociation dans les Systèmes Multi-Agents

3.1. *Principes de la négociation*

Dans les systèmes multi-agents (SMA), les agents interagissent pour réaliser fidèlement des actions commandées par une entité externe. Ainsi, dans le cas général, les agents sont amenés à coordonner leurs actions et parfois à coopérer afin d'atteindre leur but alors qu'ils peuvent avoir des motivations différentes. La négociation permet ainsi aux agents de se coordonner, de se partager des ressources limitées ou de résoudre un conflit en s'accordant sur une solution dans laquelle leurs intérêts respectifs sont au mieux satisfaits. Dans notre modèle, il s'agit plutôt de négocier pour rétablir un contrat valide en traitant chacune de ses dispositions en échec.

Les modèles de négociation mis en œuvre dans les SMA spécifient généralement un langage de communication et un protocole de négociation afin de concevoir le schéma des interactions entre agents, ainsi que les capacités de raisonnement des agents, qui modélisent leur fonctionnement interne pour mener leurs stratégies. Les langages de communication définissent un ensemble de règles pour réaliser l'échange d'information entre agents. Ces règles portent sur des aspects situés au bas niveau de la communication entre agents et peuvent, par exemple, spécifier la structure des messages ou les actions de communication [COM 02a]. Comme le modèle `Fractal` facilite la communication directe entre composants, nous ne nous y intéressons pas.

3.2. *Protocole de négociation*

Dans la majorité des cas d'application de la négociation, le protocole utilisé se base sur le *Contract-Net Protocol* (CNP) [SMI 80] ou une version étendue [COM 02b]. Ce protocole définit un processus général de négociation en faisant intervenir un initiateur et des participants et en se basant sur des itérations de primitives simples (proposer, refuser, accepter) organisées en trois étapes : l'appel d'offres, la réception des offres et leur analyse et enfin la contractualisation. Ces caractéristiques en font un protocole intéressant pour notre problématique. De plus, et à la différence d'un CNP standard, nous connaissons les responsabilités de chaque disposition et notre protocole peut ainsi être plus fin en orientant la négociation en fonction des rôles des parties dans la négociation.

3.3. *Domaines d'application et raisonnements*

Les capacités de raisonnement des agents déterminent fortement l'issue de la négociation et sa vitesse de convergence. Elles sont souvent adaptées en fonction des applications cibles.

Dans les systèmes de négociation qui modélisent des places de marché virtuelles [SPR 96], le raisonnement des agents se base fréquemment sur des stratégies prédéfinies (fonctions de type linéaire, quadratique ou additive pondérée) qui ne prennent en compte qu'un seul critère. Ce raisonnement convient bien aux systèmes de ventes aux enchères. Pour notre application, les propriétés qui interviennent dans les dispositions ne sont pas négociables avec de telles fonctions.

D'autres travaux [FAR 99] détaillent des stratégies qui distinguent clairement l'évaluation des propositions de leur formulation ; elles ont été appliquées par Jennings et al. dans [FAR 00] pour la négociation de ressources dans les réseaux. L'évaluation des propositions se fait par le biais d'une fonction additive multi-critère pondérée, ce qui permet d'apprécier, pour chaque critère, la valeur reçue et sa pondération en fonction de son importance. La formulation des propositions se base sur les stratégies de donnant-donnant, de rajout-suppression ou de concession. La stratégie du donnant-donnant consiste à formuler une proposition différente de celle reçue, mais qui reste équivalente par une fonction de similarité. La stratégie de rajout-suppression de critères se base sur la modification de la liste des critères sur lesquels porte la négociation, en retirant les critères sur lesquels bute la négociation ou en rajoutant de nouveaux critères susceptibles de relancer la négociation. Mais ces deux stratégies ne nous conviennent pas car elles sont beaucoup trop sophistiquées et présentent notamment des défauts de complexité calculatoire (recherche d'une proposition équivalente) et de sémantique de la négociation (recherche des critères à ajouter ou supprimer). En ce qui concerne la stratégie de concession, son principe repose sur l'établissement de fonctions décroissantes (fonctions polynomiales ou exponentielles) dont la vitesse de décroissance peut dépendre du temps, des ressources restantes ou de l'observation de la manière dont les autres font leurs concessions. Cette stratégie peut convenir à

notre modèle de négociation. Toutefois, au lieu de définir des fonctions qui guident les concessions, nous adopterons une déclinaison établie par Balogh et al. [BAL 00], qui se base en effet sur une liste d'alternatives successives. Pour notre application, ces alternatives seraient des modifications des dispositions ou des reconfigurations simples des composants.

Enfin, le projet de recherche GeNCA [MAT 03] définit une modèle générique de négociation pour mettre en œuvre des systèmes de négociation. Plusieurs exemples d'application sont donnés (enchère, prise de rendez-vous, choix d'un restaurant) et mettent en œuvre des stratégies par défaut basées sur des fonctions linéaires ou additives qui prennent en compte des préférences. Ce schéma de négociation, basé sur des demandes/propositions de modification, correspond bien à nos besoins, et comme le modèle reste ouvert à la définition d'autres stratégies, son utilisation reste envisageable.

4. Modèle de négociation

De manière générale, les contraintes exprimées dans les contrats portent sur des aspects fonctionnels et non fonctionnels. La négociation des propriétés fonctionnelles reste surtout valable lors des tests. En revanche, comme les aspects non fonctionnels spécifient la qualité de fonctionnement des composants (qualité des services) et leurs relations avec l'environnement (contraintes de déploiement), nous proposons de négocier des dispositions qui spécifient de tels aspects. Pour cela, comme les contrats sont découpés en dispositions, nous définissons une négociation atomique, qui est déclenchée pour chaque disposition en échec, en vue de la rétablir. Cette négociation peut intervenir lors d'une phase de (re-)configuration pour une disposition vérifiable statiquement, ou à l'exécution pour une disposition vérifiable dynamiquement. De plus, comme le métamodèle de `ConFract` permet d'identifier précisément les responsabilités et les participants associés à chaque disposition, notre modèle de négociation s'appuie sur la connaissance des garants et des bénéficiaires pour modifier les dispositions. Il est ainsi possible, d'après ces responsabilités, d'adopter différents types de *politique* qui définissent le raisonnement des participants et orientent la négociation atomique.

4.1. *Négociation atomique*

La négociation atomique se déroule selon un protocole inspiré du CNP étendu [COM 02b] et fait intervenir un initiateur et des participants en s'appuyant sur trois étapes. Ainsi, l'initiateur *demande des modifications* aux parties sur la disposition en échec (étape 1 du CNP) et les parties *soumettent des modifications*. Par la suite, l'initiateur re-vérifie la disposition modifiée en vue de la rétablir (étape 2 du CNP). Ces étapes sont réitérées et la négociation se termine, soit lorsqu'une solution satisfaisante est trouvée et la négociation finalisée (étape 3 du CNP), soit par la rupture du contrat après échec des modifications ou dépassement de la durée limite de la négociation.

Quant aux parties négociantes, ce sont *(i)* le contrôleur de contrat dans le rôle de l'initiateur, car c'est lui qui gère le cycle de vie du contrat et réalise les vérifications, *(ii)* les participants de la disposition en échec, et enfin *(iii)* un *participant externe* qui permet de représenter les intérêts d'une entité possédant une capacité de décision supérieure. Par exemple, cette partie peut représenter l'administrateur de l'application et ainsi permettre d'intégrer des contraintes globales sur le système (contraintes de déploiement) et des données pour paramétrer le processus de négociation (négociabilité d'une disposition étant donné le contexte de déploiement, durée limite de la négociation, diffusion d'informations sur des négociations à d'autres niveaux de la hiérarchie des composants). Dans la précondition du contrat de composition de notre exemple, les parties qui négocient sont le contrôleur de contrat de `<fp>` en tant qu'initiateur, `<fp>` lui même au titre de garant, et `<pl>` en tant que bénéficiaire (cf. figure 2).

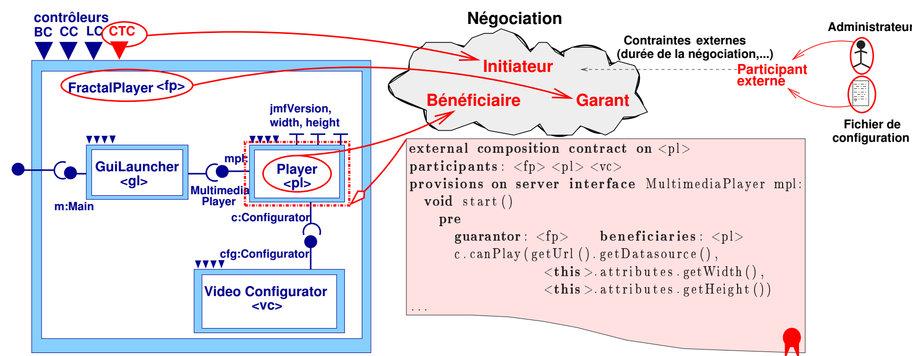


Figure 2 – Illustration des parties négociantes pour le contrat de composition externe.

4.2. *Politique de relâchement*

Le principe de cette politique consiste à s'orienter vers les bénéficiaires de la disposition pour qu'ils procèdent au *relâchement des contraintes*. Ainsi, elle peut conduire au changement de la disposition en échec en conservant le contexte d'exécution du système ou à la modification des attributs des bénéficiaires à disposition constante. C'est une politique suffisamment générale pour être applicable du moment que le formalisme de spécification utilisé permette le découpage des expressions en dispositions.

Dans la mise en œuvre de la politique de relâchement, la notion de bénéficiaire est affinée en distinguant les bénéficiaires principaux des bénéficiaires auxiliaires. Les bénéficiaires principaux sont directement concernés par la disposition et ont la capacité de modifier la disposition. En revanche, les bénéficiaires auxiliaires ont un rôle plus passif et ne peuvent pas faire avancer la négociation. Ainsi, si une postcondition avait été définie sur la méthode `start` de l'interface `mpl`, `<fp>` serait le bénéficiaire princi-

pal car responsable de l'utilisation correcte du contrat et `<gl>` bénéficiaire auxiliaire, car simple client du service et plus indirectement concerné.

En cas d'échec d'une vérification, le processus de relâchement se décompose en trois phases décrites à la figure 3. Tout d'abord (phase 1), l'initiateur demande la négociabilité de la disposition en échec à tous les bénéficiaires et détermine le résultat par une fonction additive pondérée. Si la disposition n'est pas négociable, la négociation atomique échoue. Dans le cas contraire (phase 2), l'initiateur demande aux bénéficiaires principaux de *faire des concessions* en leur décrivant la disposition en échec, et les bénéficiaires principaux *proposent des concessions*. Les bénéficiaires principaux peuvent alors renvoyer soit une nouvelle disposition, soit une alternative qui décrit le relâchement à effectuer sur leurs attributs. Par la suite, l'initiateur effectue les relâchements proposés et réévalue la disposition. Si l'évaluation de la disposition est fausse alors l'initiateur annule les modifications pour rétablir le contexte initial et redemande des concessions. Au contraire, si l'évaluation est vraie alors la négociation atomique est réussie, et l'initiateur finalise la négociation en demandant aux bénéficiaires d'effectuer les relâchements correspondants. Autrement (phase 3), si l'initiateur ne reçoit que des demandes de retrait venant des bénéficiaires principaux, il procède à une dernière consultation pour demander aux bénéficiaires principaux et auxiliaires l'autorisation de retirer la disposition.

Les bénéficiaires principaux peuvent raisonner en s'appuyant sur un ensemble d'alternatives qui définit les relâchements. Un composant C associe ainsi à la disposition $\#id$ un ensemble d'alternatives (formules ou valeurs) ordonnées $\mathcal{A}_{\#id,C} := \{A^1_{\#id,C}, A^2_{\#id,C}, ..., A^n_{\#id,C}, \text{STOP ou RELEASE}\}$. Ainsi, dans le protocole de négociation, à chaque fois que l'initiateur émet une demande de concession au composant, celui-ci renvoie un relâchement, correspondant à une nouvelle disposition ou une modification des attributs, nouvellement construit à partir de cet ensemble d'alternatives. La dernière alternative STOP (resp. RELEASE) permet de notifier la fin du relâchement en conservant la disposition (resp. en autorisant le retrait de la disposition).

Dans l'exemple portant sur la capacité du lecteur à jouer une vidéo dans sa totalité, la négociation par relâchement de la précondition fait intervenir le contrôleur de contrat de `<fp>`, et `<pl>` en tant que bénéficiaire principal. Il n'y a, dans ce cas, pas de bénéficiaire auxiliaire. Ainsi, le déroulement de la négociation par relâchement conduirait à consommer moins de ressources batterie, par exemple, en modifiant successivement les paramètres d'affichage de la vidéo pour pouvoir la visualiser dans sa totalité. Si cela reste insuffisant, il est alors possible d'imaginer le retrait complet de la contrainte, et la vidéo pourra être interrompue en cours de la lecture si la batterie s'avère être insuffisante. Dans un tel scénario, `<pl>` définit un ensemble d'alternatives $\mathcal{A}_{pre_1,<pl>} := \{(width := \frac{width}{\sqrt{2}}, height := \frac{height}{\sqrt{2}}), \text{RELEASE}\}$ qui représente les relâchements à effectuer sur les attributs d'affichage de la vidéo. De cette manière, à la première demande de concession, `<pl>` renvoie une alternative qui décrit la modification à réaliser sur les attributs `width` et `height`. L'initiateur modifie alors les attributs d'affichage de `<pl>` et si la disposition est satisfaite, la négociation atomique s'achève par un succès. Sinon, le contrôleur de contrat annule la modification faite

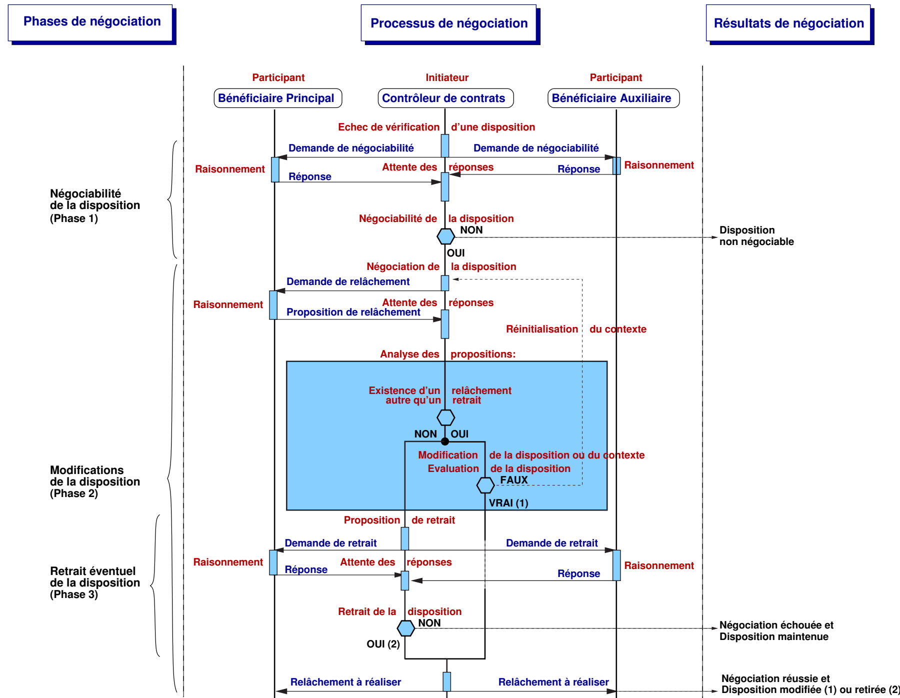


Figure 3 – Processus de négociation atomique par relâchement.

et redemande une concession à laquelle `<pl>` répond par RELEASE pour proposer le retrait de la disposition. Comme il est l'unique bénéficiaire, la disposition est retirée et la négociation réussit.

5. Conclusion

Dans cet article, nous avons proposé des mécanismes de négociation adaptés à des contrats non fonctionnels, placés sur des composants logiciels hiérarchiques, comme ceux de la plate-forme *Fractal*. Ces mécanismes ont été développés pour le système *ConFract*, qui permet de poser des contrats à la fois sur les connexions entre composants et sur les composants eux-mêmes. Notre modèle de négociation s'inspire en partie des protocoles de négociation des systèmes multi-agents. Il reprend le principe du *Contract-Net Protocol* en l'appliquant à chaque niveau de hiérarchie : le contrôleur de contrats est alors l'initiateur de la négociation et les participants du contrat sont consultés selon une politique donnée. Nous avons décrit une politique de négociation par relâchement qui est bien adaptée au langage d'assertions exécutables *CCL-J*, actuellement utilisé dans *ConFract*. Dans cette politique, l'initiateur consulte les bénéficiaires des dispositions négociées, pour essayer de réduire les contraintes posées.

L'intégration de ces mécanismes de négociation dans le système `ConFract` a débuté. A la suite des premières expérimentations, nous envisageons d'utiliser une bibliothèque Java existante pour la négociation d'agents, comme GeNCA [MAT 03] ou JADE [BEL 03], afin de bien caractériser notre modèle de négociation par rapport à ceux utilisés dans les systèmes multi-agents. L'utilisation des mécanismes présentés dans plusieurs applications de référence, dont un lecteur multimédia sur terminal mobile, devrait nous apporter les retours nécessaires pour valider et améliorer notre modèle.

D'autres politiques sont envisageables pour améliorer nos mécanismes de négociation. Une politique par effort consisterait à orienter la négociation vers les garants des dispositions. Un composant garant pourrait typiquement *faire un effort* sur une propriété non fonctionnelle si celle-ci était déterminée de manière compositionnelle et que ce composant puisse *reconfigurer* son contenu. Par la suite, une politique de donnant-donnant pourrait ainsi être appliquée sur plusieurs dispositions en même temps. Dans un futur proche, nous comptons donc concevoir un modèle compositionnel pour des propriétés non fonctionnelles, ainsi que les politiques de négociation adaptées afin de développer nos mécanismes de négociation, pour le moment confinés à un niveau de hiérarchie, en les diffusant au niveaux inférieurs. Dès lors, il sera aussi nécessaire d'élaborer des solutions aux problèmes de dépendances cycliques afin d'assurer la terminaison de l'ensemble des négociations.

6. Bibliographie

[ALD 02] ALDRICH J., CHAMBERS C., NOTKIN D., « Architectural reasoning with ArchJava », *ECOOP'2002*, Malaga, Spain, juin 2002.

[BAC 00] BACHMAN F., BASS L., BUHMAN C., COMELLA-DORDA S., LONG F., ROBERT J., SEACORD R., WALLNAU K., « Technical Concepts of Component-Based Software Engineering », rapport n° CMU/SEI-2000-TR-008, mai 2000, Carnegie Mellon Software Engineering Institute, Volume 2.

[BAL 00] BALOGH Z., LACLAVÍK M., HLUCHÝ L., « Model of Negotiation and Decision Support for Goods and Services », *Proceedings of XXIInd International Colloquium ASIS 2000 - Advanced Simulation of Systems*, Ostrava, Czech Republic, 2000.

[BEL 03] BELLIFEMINE F., CAIRE G., POGGI A., RIMASSA G., « JADE, a white paper », *TILAB journal*, vol. 3, n° 3, 2003, p. 6-19.

[BRU 03] BRUNETON E., COUPAYE T., STEFANI J.-B., « The Fractal Component Model », Specification, Technical Report n° v1, v2, 2002,2003, The ObjectWeb Consortium, http ://-fractal.objectweb.org.

[BRU 04] BRUNETON E., COUPAYE T., LECLERCQ M., QUÉMA V., STEFANI J.-B., « An Open Component Model and Its Support in Java », *ICSE 2004 - CBSE7*, vol. 3054 de *LNCS*, Springer Verlag, mai 2004.

[COL 04a] COLLET P., DEVEAUX D., ROUSSEAU R., TRAON Y. L., « Contract-based Testing : from Objects to Components », *International Workshop on Testability Assessment (IWoTA'04)*, novembre 2004.

[COL 04b] COLLET P., ROUSSEAU R., « Contracting Hierarchical Components », rapport, mars 2004, I3S Laboratory - Sophia Antipolis.

[COM 02a] COMMUNICATION F. T., « FIPA ACL Message Structure Specification », rapport, 2002, FIPA Organization.

[COM 02b] COMMUNICATION F. T., « FIPA Contract Net Interaction Protocol Specification », rapport, 2002, FIPA Organization.

[FAR 99] FARATIN P., SIERRA C., JENNINGS N., BUCKLE P., « Designing Flexible Automated Negotiators : Concessions, Trade-Offs and Issue Changes », rapport n° RP-99-03, 1999, Institut d'Investigacio en Intel.ligencia Artificial Technical Report.

[FAR 00] FARATIN P., JENNINGS N., BUCKLE P., SIERRA C., « Automated Negotiation for Provisionning Virtual Private Networks using FIPA-Compliant Agents », *Proc. 5th Int. Conf. on the Practical Application of Intelligent Agents and Multi- Agent Systems (PAAM-2000)*, Manchester, UK, 2000, p. 185-202.

[MAT 03] MATHIEU P., VERRONS M.-H., « ANTS : an API for creating negotiation applications », *Proceedings of the CE 2003 Madeira Island - Portugal*, July 26-30 2003.

[OBJ 97] OBJECT MANAGEMENT GROUP I., « Object Constraint Language Specification », rapport n° version 1.1, ad/97-08-08, septembre 1997, IBM www.software.ibm.com/ad/ocl

[SMI 80] SMITH R. G., « The contract net protocol : high-level communication and control in a distributed problem solver », *IEEE Transactions on Computers*, vol. 29, 1980, p. 1104-1113.

[SPR 96] SPRINKLE J., BUSKIRK C. V., KARSAI G., « Kasbah : An Agent Marketplace for Buying and Selling Goods », *Proceedings of the First International Conference on the Practical Appication of Intelligent Agents and Multi-Agent Technology*, London, UK, 1996.

[SZY 02] SZYPERSKI C., *Component Software — Beyond Object-Oriented Programming*, Addison-Wesley Publishing Co. (Reading, MA), 2002.

Un modèle de spécification et d’implémentation de composants-rôles pour les systèmes multi-agents

Nabil Hameurlain

LIUPPA-Université de Pau
Avenue de l'Université BP 1155
64012 Pau, France
nabil.hameurlain@univ-pau.fr

RÉSUMÉ. *Les rôles sont l’un des concepts fondamentaux utilisés en tant que blocs de base, pour la modélisation de la structure organisationnelle des systèmes multi-agents (SMA), pour la spécification des protocoles d’interaction, et enfin pour la spécification fonctionnelle du comportement des agents. La modélisation des interactions à base de rôles offre plusieurs avantages ; le plus important est celui de la séparation des aspects, en dissociant le niveau agent (aspects algorithmiques) et le niveau système en ce qui concerne l’interaction. Cependant, l’ouverture et l’extensibilité des applications dans les SMA peut nécessiter la composition de rôles, développés le plus souvent de manière indépendante, et peut donc mener à l’émergence d’un comportement inattendu des agents. Cet article présente une spécification des rôles pour des interactions complexes au sein des SMA. L’objectif de cette approche est d’intégrer, dans le développement basé-composants (DBC) des rôles, des aspects liés à la spécification et à la vérification des rôles. Un exemple d'application de protocole d'interaction est donné pour illustrer notre approche.*

MOTS-CLÉS : *rôles, composants, spécification, réseaux de Petri, protocoles d’interaction.*

ABSTRACT. *Roles are an important concept used for different purposes like the modelling of organisational structure of MAS, the modelling of protocols, and as basic building clocks for defining the behaviour of agents. Modeling interactions by roles gives several advantages, the most important of which is the separation of concerns by distinguishing the agent-level and system-level with regard to interaction. However, in open MAS, the composition of independently developed roles may lead to unexpected emergent behaviour of the agents. This paper presents a specification model of roles for complex interactions. Our approach aims to integrate specification and verification into the Component Based Development (CBD) of roles. An application example of interaction protocols is given to illustrate our formal framework.*

KEY WORDS: *roles, components, specification, Petri nets, interaction protocols.*

1. Motivations

Le paradigme des Systèmes Multi-Agents (SMA) est l'une des approches les plus prometteuses pour concevoir des systèmes ouverts et dynamiques, où des composants hétérogènes sont naturellement représentés par des agents autonomes. Ces agents sont capables d'interagir avec l'environnement, de rejoindre et de quitter le système à volonté. L'interaction entre agents autonomes est fondamentale pour la dynamique des SMA ouverts [FEG 98] ; en effet, les agents appartenant à la même application doivent interagir pour coordonner leur activité afin d'atteindre leur but global et commun, alors que les agents appartenant à des applications différentes, peuvent également avoir besoin d'interagir, par exemple, dans le cas d'une compétition pour une ressource.

Les rôles sont un concept fondamental utilisés, en tant que blocs de base, pour la modélisation de la structure organisationnelle des systèmes multi-agents, pour la spécification des protocoles d'interaction, et enfin pour la spécification fonctionnelle du comportement des agents. La modélisation des interactions par des rôles permet une séparation des aspects en dissociant le niveau algorithmique des agents et le niveau système concernant les interactions. Elle permet également la réutilisation des rôles précédemment définis pour les applications semblables ; ainsi, les rôles doivent être spécifiés et modélisés de manière appropriée, sachant que la composition des rôles développés indépendamment les uns des autres peut mener à l'émergence d'un comportement inattendu des agents.

Le Développement Basé Composants (DBC) [SZY 01] est l'une des techniques les plus utilisées à l'heure actuelle en génie logiciel, permettant de faciliter la construction des applications à grande échelle en supportant la composition de modules simples et élémentaires dans des applications complexes. La spécification et la vérification sont nécessaires pour permettre une composition sûre des systèmes à partir de composants. Le DBC et la spécification -vérification sont synergiques : le DBC introduit des structures compositionnelles et les règles de composition dans le système à construire, alors que la spécifications et la vérification permettent un développement fiable de logiciels basé-composants.

Bien que le concept de rôle ait été exploité dans différentes approches pour le développement d'applications à base d'agents [CLZ 03, FEG 95, ZJW 03], aucun consensus n'a été atteint sur ce qu'est un rôle et comment il devrait être spécifié. Dans cet article nous utilisons une définition spécifique des rôles, qui n'est pas en opposition avec celles déjà proposées dans les différentes approches citées ci-dessus, mais qui est simple, et qui peut être exploitée dans la spécification et l'implémentation. Un rôle inclut un ensemble d'éléments d'interface (soit des attributs ou des opérations, requis et fournis, nécessaires pour accomplir les tâches du rôle), un comportement (donnant une sémantique aux éléments de l'interface), et des propriétés (prouvées satisfaites par le comportement). Un agent qui veut assumer un rôle, crée une nouvelle instance du composant-rôle et ce composant-rôle est lié à cet agent. Le but est d'intégrer des aspects liés à la spécification et à la vérification dans le développement basé-composants des rôles.

La structure de l'article est la suivante. La section 2 présente le modèle de spécification de composants de rôles RICO (Role-based Interactions CompOnents) [HSB 04] et ses caractéristiques. La section 3 présente une implémentation de RICO par les Objets Coopératifs [SIB 01], un langage de spécification formelle orienté

objets ; pour illustrer cette implémentation, nous présentons un exemple de spécification de protocoles d'interaction, le protocole d'enchère au poisson, et étudions les propriétés de sûreté et de vivacité des composants-rôles de ce protocole. La vérification de certaines propriétés de ce protocole par graphe d'accessibilité ainsi que par modèle est également présentée dans cette section. Dans la section 4, nous présentons un état de l'art en situant notre approche par rapport aux travaux existants, avant de conclure dans la section 5.

2. Spécification des rôles par composants : le modèle RICO

RICO (Role-based Interactions CompOnents) [HSB 04] est un modèle générique pour spécifier les rôles dans les SMA par approche componentielle. La principale motivation de modéliser les rôles comme des composants est de capturer des schémas d'interaction qui :

- ont des fonctionnalités bien définies et des propriétés prouvées,
- peuvent être combinés entre eux pour réaliser un but,
- peuvent être dynamiquement liés et dissociés de l'agent, si nécessaire.

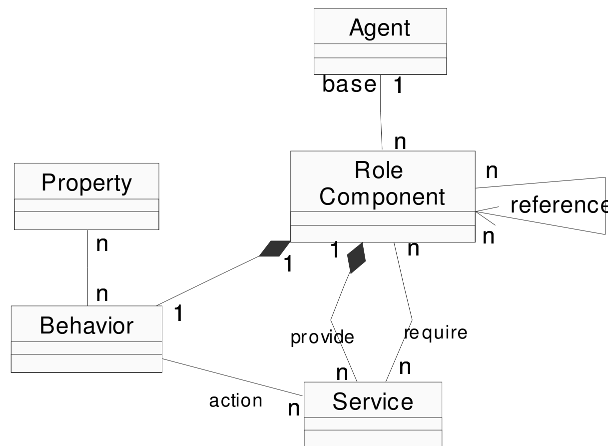


Figure 1. méta-modèle UML des différents concepts utilisés dans le modèle de spécification RICO.

Dans RICO, un rôle est considéré comme un composant qui offre un ensemble d'éléments d'interface (soit des attributs ou des opérations, requis et fournis, nécessaires pour accomplir les tâches du rôle), un comportement (donnant une sémantique aux éléments de l'interface), et des propriétés (prouvées a être satisfaites par le comportement) ; la figure 1 présente un diagramme de classe UML montrant les différents éléments utilisés par RICO et la relations entre eux. Dans ce méta-modèle, un composant-rôle fournit et requiert des services, qui doivent être implémentés par d'autres composants, et un service fait partie d'un et d'un seul composant-rôle. Cette indépendance de l'implémentation des services du composant-rôle est une caractéristique essentielle du modèle RICO respectant ainsi l'approche

componentielle. Ce modèle a pour but de représenter la relation entre ces éléments de manière générique, puisqu'en pratique, cette relation varie suivant les techniques de spécification, en distinguant par exemple les spécifications orientées objets et les spécifications procédurales.

Dans RICO, un agent peut prendre un ou plusieurs rôles en même temps, et un agent peut assumer le même rôle plusieurs fois, dans la même conversation ou dans des conversations distinctes. Pour la simplicité, et pour éviter la gestion des conflits d'accès aux ressources (de l'agent), nous considérons qu'à tout moment, au plus un seul rôle est activé, et ceci sous la décision de l'agent [DRH 04]. Dans ce qui suit, nous donnons une définition explicite des éléments utilisés dans ce modèle.

Définition 2.1
Un Composant-Rôle RC, pour un rôle R est un 5-uplet, *(Ag, Ref, Serv, Behav, Prop)* où,

- *Ag* est l'identité de l'agent auquel RC est lié : *Ag* crée et intancié RC, et RC joue le rôle R sous son propre contrôle.
- *Ref* est la liste des identificateurs des composants-rôles, utilisée et connue par RC pour interagir avec.
- *Serv* est l'interface de RC, l'ensemble des éléments publics à travers laquelle RC interagit avec d'autres composants ; ces éléments sont des attributs ou des opérations, des méthodes selon la dénomination orientée objets. De plus les éléments de Serv sont des éléments fournis ou requis par le composant-rôle. Les éléments fournis sont gérés par RC, et qui sont à la disposition des autres composants-rôles, et les éléments requis sont fournis par d'autres composants-rôles et utilisés par RC. Notons que cette interface constitue le moyen de base pour interagir avec l'agent assumant ce rôle R et les autres composants-rôle.
- *Behav* définit le comportement de RC vis à vis des autres composants-rôles du système. Il décrit le cycle de vie de RC et les séquences d'actions observables supportées par RC, comme des opérations fournies ou requises. *Serv* et *Behav* peuvent être considérés comme la spécification fonctionnelle de RC ; *Behav* étant le langage défini sur les opérations de Serv, et concerne les éléments de *Serv* en capturant de manière précise leur sémantique et leur comportement. Par exemple, la sémantique peut être spécifiée en utilisant les pre- et post-conditions, décrivant le cycle de vie du composant-rôle : séquentialité, synchronisation, et concurrence des opérations.
- *P* est un ensemble de propriétés, prouvées satisfaites par Behav, permettant aux composants-rôle, utilisant les services de RC, d'avoir confiance à l'accomplissement de ces services ; nous intéressons plus particulièrement aux propriétés de sûreté et de vivacités [MPN 95]. Ces propriétés sont des spécifications fonctionnelles plus abstraites et plus déclaratives que *Behav*, et qui sont vérifiées par RC. Il s'agit d'invariants, de propriétés comportementales, ou bien de propriétés temporelles exprimées par exemple par des formules de logique temporelle CTL [MPN 95]. P peut

être étendu de manière incrémentale: une propriété n'est rajoutée dans P que lorsqu'il s'avère qu'elle est satisfaite.

Les composants-rôles interagissent entre eux par appels aux opérations fournies ; des messages en entrée ou en sortie de RC sont consommées ou générés par Behav. Ces composants-rôles permettent de spécifier, fonctionnellement le comportement des agents, et les protocoles d'interaction puisque :
- les composants-rôles sont réactifs, proactifs, et autonomes: à travers leurs interfaces, la réactivité (resp. proactivité) est assurée par les opérations et les services fournis (resp. requis), et l'autonomie est assurée par le comportement. Ainsi, les composants-rôles peuvent être utilisés comme constituants de base de la structure interne des agents.

- les composants-rôles peuvent être considérés comme composants (membres) d'un protocole d'interaction ; ceci permet de modéliser des protocoles d'interaction complexes, ouverts et concurrents par approche componentielle. Ainsi, un agent peut jouer un ou plusieurs rôles en même temps, dans différentes conversations (protocoles), et chaque participation est gérée par une instance d'un composant-rôle.

Du point de vue spécification et conception des rôles, nous avons une vue semblable à la méthodologie Gaia [ZJW 03]. Dans ce papier, nous nous focalisons sur la spécification, la vérification des composants-rôles, et leur implémentation par un langage de spécification formelle orienté objets, les Objets CoOpératifs (OCO) [SIB 01].

3. Une implémentation des composants RICO par les Objets CoOpératifs

Dans [HSB 04] nous avons montré comment instancier le modèle de composants-rôles RICO par le formalisme des Objets CoOpératifs (OCO) [SIB 01], un langage de spécification formelle orienté objets, permettant de modéliser un système par une collection d'objets actifs coopérant de manière asynchrone selon le protocole client/serveur. Dans ce formalisme, chaque objet est une instance de sa classe d'OCO, ayant une référence, et peut être crée et détruit de manière dynamique. La structure d'une classe d'OCO inclut un ensemble *d'attributs*, un ensemble *d'opérations*, une *structure de contrôle* appelée OBCS (OBject Control Structure), et un ensemble de services (requis et fournis) supporté par l'OBCS. La définition d'une classe d'OCO est divisée en deux parties (voir l'exemple ci dessous) : la partie spécification concerne les attributs publics composant l'interface, alors que la partie implémentation inclut les attributs privés, notamment l'OBCS, un réseaux de Petri à Objets définissant la structure de contrôle. Les services sont traités selon l état de l'OBCS. Ce sont des méthodes publics avec des paramètres d'entrée et de sortie typés : les *services fournis* sont graphiquement distingués par un arc pendant entrant à chaque transition acceptant ce service, alors que les *services requis* sont distingués par un arc sortant de chaque transition demandant ce service. Les services sont traités selon l'état de l'OBCS. Pour illustrer l'implémentation du modèle de spécification RICO, nous étudierons un exemple de spécification de protocoles d'interaction, le protocole d'enchère au poisson. Dans la suite nous montrerons dans un premier temps comment spécifier et implémenter les deux composants rôles de ce protocole, les composants Vendeur et Acheteur (ou Preneur),

dans l'environnement SYROCO [SIB 95], un environnement C++ supportant les OCO, ensuite nous explicitons leurs propriétés de sûreté et de vivacité en exploitant des outils d'analyse, comme l'outil INA [ROS 99].

Rappelons que dans n'importe quelle conversation respectant les règles de ce protocole, nous avons un seul vendeur, et un certain nombre d'acheteurs potentiels. Le vendeur a un seau de poisson à vendre pour un prix initial. Un acheteur peut faire une offre en signalant son intérêt. Si aucun (ou resp. plus d'un) acheteur est intéressé, le vendeur annonce un prix inférieur (ou resp. plus élevé). Quand un et seulement un acheteur est intéressé, le vendeur attribue le poisson à ce preneur. Une fois que le seau de poisson est attribué, le vendeur donne le poisson et reçoit le paiement, alors que le preneur paie le prix et reçoit le poisson.

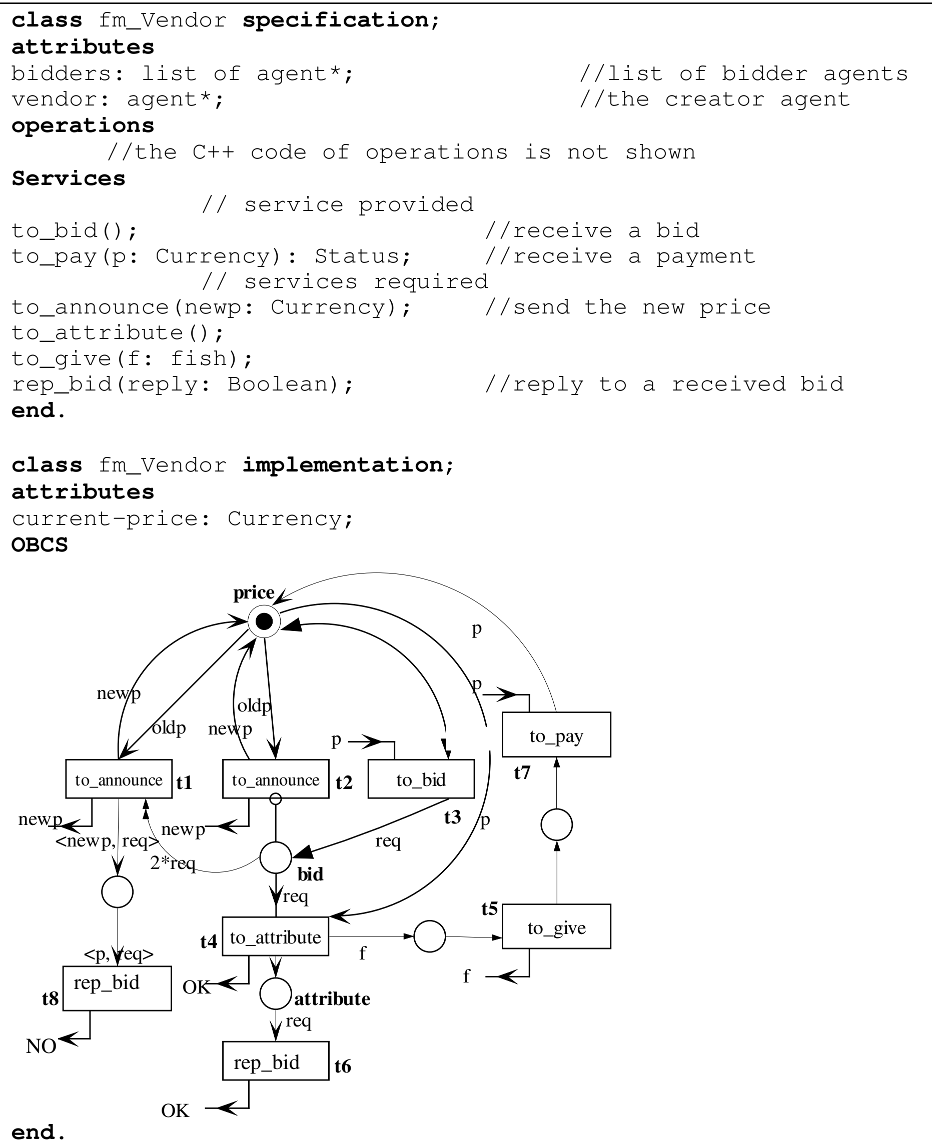
3.1. Spécification des Composants

Tout d'abord, nous considérons la classe fm_Vendor (figure 2). Le comportement de cette classe est comme suit : depuis l'état initial (un jeton dans la place d'entrée `price`), seulement les transitions t2 est et t3 sont franchissables, et le vendeur ne peut exécuter que ces deux transitions ; t2 est une transition de demande du service `to_announce`, et t3 est une transition d'acceptation du service `to_bid`. L'occurrence du service `to_bid` (transition t3) produit un jeton dans la place `bid`. Cette transition reste franchissable aussi longtemps qu'il y a un jeton dans la place `announce`, c'est à dire jusqu'à l'occurrence de la transition t4, qui est une transition de demande du service `to_attribute`[1]. Ce dernier service est accepté s'il y a exactement un jeton dans la place `bid`, c'est à dire seulement s'il y a exactement un seul preneur. Autrement, le vendeur annonce un nouveau prix par l'occurrence de la transition t1 de demande du service `to_announce` et retourne `NO` à l'acheteur par l'occurrence de la transition t8. L'occurrence du service `to_attribute` (transition t4) retourne `OK` à l'acheteur et sensibilise t5 et t6 qui sont respectivement des transitions de demande du service `rep_bid` (qui retourne `OK`) et `to_give`. L'occurrence du service `to_give` permet de sensibiliser la transition t7 d'acceptation du service `to_pay`. L'état final de la conversation est donc atteint puisque le marquage atteint est le marquage initial comprend un jeton dans la place `price`.

Dans la figure 3, on donne l'implémentation du composant-rôle Preneur par la classe fm_Buyer. Le comportement de ce composant est comme suit : à partir de l'état initial (un jeton dans la place d'entrée `portfolio`), seulement la transition t1 est franchissable ; c'est une transition d'acceptation du service `to_announce`. L'occurrence de `to_announce` sensibilise le service `to_bid` ; Le franchissement de la transition t6 produit un jeton dans la place `bid`, et sensibilise la transition t2 d'acceptation du service `rep_bid` ; l'occurrence de ce dernier service produit, soit un jeton dans la place `announce`, soit un jeton dans la place `attribute`, sensibilisant ainsi la transition t3 d'acceptation du service `to_attribute`. La transition t1 reste franchissable (dans le cas d'une nouvelle annonce depuis le vendeur) jusqu'à l'occurrence de la transition d'acceptation du service `to_attribute`. L'occurrence du service `to_attribute` permet de sensibiliser

1 Les transitions t1 et t2 sont considérées comme transitions plus prioritaire que t 3, et la transition t3 est plus prioritaire que t4.

la transition t4 de demande du service `to_pay` ; l'occurrence de ce dernier service sensibilise la transition t5 d'acceptation du service `to_give`. L'état final de la conversation est atteint puisque le marquage initial contient un jeton dans la place `portfolio`.



```
class fm_Vendor specification;
attributes
bidders: list of agent*;              //list of bidder agents
vendor: agent*;                       //the creator agent
operations
       //the C++ code of operations is not shown
Services
             // service provided
to_bid();                             //receive a bid
to_pay(p: Currency): Status;          //receive a payment
             // services required
to_announce(newp: Currency);          //send the new price
to_attribute();
to_give(f: fish);
rep_bid(reply: Boolean);              //reply to a received bid
end.

class fm_Vendor implementation;
attributes
current-price: Currency;
OBCS
```



```
end.
```

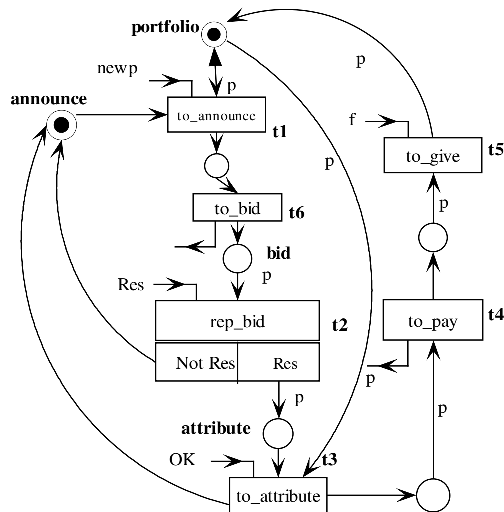
Figure 2. Vendeur dans le protocole Enchère au poisson comme classe d'OCO.

```
class fm_Buyer specification;
attributes
bidder: agent*;                         //the creator agent
vendor: agent*;                         //the vendor agent
other_bidders: list of agent*;

operations
       //C++ code of operations not shown

services
              // service required
to_bid();
to_pay(p: Currency): Status;
              // services provided
to_announce(newp: Currency);
to_attribute();
to_give(f: fish);
rep_bid(reply: Boolean);
end.

class fm_Buyer implementation;
attributes
portfolio: Currency;
current-price: Currency;
OBCS
```




```
end.
```

Figure 3. Preneur dans le protocole Enchère au poisson comme classe d'OCO.

3.2. Vérification des propriétés de sûreté et de vivacité

Nous nous focalisons sur les propriétés, de sûreté et de vivacité, comportementales et qualitatives de l'OBCS, et leur interprétation dans le contexte des SMA. Rappelons que pour analyser des propriétés comportementales d'un réseau de Petri comme, le sans blocage, la réinitialisation, le caractère borné, l'accessibilité, il est possible de générer le graphe des marquages accessibles de ce réseau en utilisant des outils d'analyses, comme l'outil INA [ROS 99]. Ce graphe montre tous les marquages accessibles et toutes les séquences de transitions ou d'actions qui peuvent être exécutés ; ainsi, la vérification par modèles de propriétés exprimées en CTL [MPN 95] est possible. Si le graphe est infini, en raison du fait que le réseau est non borné, le graphe de couverture peut être construit ; il est fini, et permet une analyse comportementales de certaines propriétés [MUR 88].

Propriétés de sûreté. Les propriétés de sûreté d'un composant-rôle expriment le fait qu'un événement indésirable ne soit pas produit durant l'exécution du (de l'instance) composant. Ces propriétés sont liées au passé, et sont vérifiées sur une exécution (trace) finie. Par exemple, un rôle exige qu'une variable dans ce composant (instance) soit toujours initialisée quand un agent prend ce rôle. Dans le protocole d'enchère par exemple, l'identité (de l'agent) et le prix initial à annoncer doivent être fixés par l'agent prenant le rôle Vendeur (c'est à dire l'agent initialisant le protocole). Ce type de propriétés peut être simplement exprimé par des invariants définis à travers les variables (publics) listées dans le composant-rôle. Les propriétés de sûreté d'un rôle peuvent non seulement exprimer des conditions sur les l'état (valeurs) de ses attributs (variables) mais aussi des conditions portant sur sa trace d'exécution (historique) et son état interne. L'analyse comportementale de l'OBCS du vendeur et du preneur révèle que le comportement associé à leurs composants-rôle est sans blocage. Concernant le caractère borné, l'OBCS du Preneur présenté à la figure 3 est borné, par contre l'analyse de l'OBCS du vendeur présenté à la figure 2 est non borné. Néanmoins, en exploitant le graphe de couverture de ce dernier, on peut vérifier des propriétés temporelles comme:

- le service `to_announce` peut être réalisé depuis l'état initial, ou bien, si depuis sa dernière occurrence, aucune ou plusieurs occurrences du service `to_bid` n'ont été réalisées.
- le service `to_attribute` est réalisé exactement une fois, lorsque, depuis la précédente occurrence du service to_`announce`, le service `to_bid` n'a été réalisé q'une seule fois.
- `to_give` et `to_pay` peuvent être réalisés seulement une seule fois, après l'occurrence de `to_attribute` .

Propriétés de vivacité. La vérification des propriétés de sûreté est un moyen très puissant pour garantir l'exactitude du comportement de l'agent, en vérifiant qu'un état erroné n'est jamais atteint. Parfois, ceci n'est pas suffisant, et nous voulions qu'un événement souhaité se produise dans le futur. C'est le but des propriétés de vivacité, et parfois même le respect des propriétés de sûreté peut exiger d'un composant « d'accorder » ses propriétés de vivacité en conséquences. Ces propriétés subtiles, nécessitent la vérification de cycles dans le graphe de marquages (comportement) du composant-rôle. Ainsi, une propriété de vivacité est violée s'il

existe une exécution infinie ne garantissant pas une progression de cette exécution. L'analyse comportementale de l'OBCS du vendeur--preneur, le réseau de Petri à Objets obtenu par la composition des OBCS des classes fm_Vendor et fm_Buyer selon le protocole client/serveur, révèle que l'état initial : un jeton dans la place price (resp. dans la place portfolio) peut être reproduit. Les comportement du vendeur et des acheteurs sont donc réversibles, et par conséquent l'enchère (protocole) peut se terminer avec succès, et toute conversation peut éventuellement être complète et finie. Par ailleurs, en exploitant le graphe des marquages accessibles de l'acheteur, on peut vérifier des spécifications comme :

- après l'intervention de `to_attribute`, le service `to_pay` doit être réalisé;
- après l'intervention de `to_pay`, le service to_`give` doit être réalisé.

4. Travaux voisins et discussion

Plusieurs approches et méthodologies pour la spécification des interactions dans les SMA ont été proposées. Dans [FEG 98], l'auteur propose un méta-modèle pour la définition des modèles organisationnels, basé sur les trois concepts : agent, groupe, et rôle. Les agents appartiennent à des groupes en tenant des rôles, et les interactions peuvent avoir lieu seulement entre agents d'un même groupe et selon leurs rôles respectifs. Notre approche est dans la même ligne, puisqu'elle est basée sur les composants-rôles, et les agents tenant un rôle dans une conversation (protocole) constitueront un groupe. En outre, notre approche donne une sémantique formelle et une définition plus précise des composants-rôles comme patterns d'interaction -protocoles et rôles - et les groupes sont définis sur la base des conversations, c'est à dire des instances des rôles et des protocoles.

La Programmation par Aspects (PA) a été exploitée pour implémenter le concept des rôles dans [KEN 02]. L'auteur décrit l'intérêt de modéliser les rôles pour les systèmes à base d'agents. Dans notre approche, les rôles des protocoles d'interaction peuvent être considérés comme des composants de protocoles pour l'interaction entre agents. Ainsi, cette approche permet de sélectionner et de réutiliser les protocoles en considérant non seulement leurs fonctionnalités mais également leurs propriétés. Dans [CLZ O3], les auteurs suivent cette approche de programmation par aspects, et proposent un modèle d'interactions basé sur la notion de XRole (Rôle décrit en XML), où les notations basées sur XML sont adoptées pour supporter la définition et l'exploitation des rôles à différentes phases du développement d'applications. Cette approche est proche de la notre puisqu'elle est basée sur le principe de séparation des préoccupations. Les XRole se focalisent sur la flexibilité, l'ouverture et l'adaptation des rôles, et non pas sur leur sémantique formelle, par conséquent, il n'y a aucune possibilité de valider et de vérifier le comportement des rôles. Dans [ZJW 03], la méthodologie Gaia adopte une description abstraite et semi-formelle pour exprimer le comportement (les « capabilities») prévu des rôles impliqués dans un protocole. Cette méthodologie est proche de la notre puisqu'elle est basée sur des abstractions organisationnelles pour l'analyse et la conception des interactions ouvertes et complexes ; néanmoins, elle a pour limitation la spécification formelle et la vérification, et notamment l'implémentation et l'exécution des rôles. En effet, ceci est dû au fait que le cycle de vie des rôles dans Gaia est seulement

exprimé par des propriétés de sûreté et de vivacité, et cette méthodologie ne traite pas directement les aspects formels, d'analyse et d'implémentation.

En considérant l'aspect spécification et validation des SMA complexes et ouverts, [KHW 01] propose une extension d'AUML (Agent UML) pour la conception modulaire des protocoles d'interaction en composant des micro-protocoles ; la contribution principale de cette approche est de réduire l'espace existant entre la spécification informelle et semi-formelle des interactions, en utilisant les diagrammes de protocoles (diagrammes de séquences AUML), et un langage graphique pour la conception des protocoles. Néanmoins, la spécification et la vérification des protocoles d'interaction ouverts restent un processus non trivial, puisque, comme l'auteur l'indique, le concepteur est contraint d'utiliser des passerelles vers d'autres formalismes pour la phase de validation. Dans [BDJ 97] la spécification d'un SMA est basée sur une hiérarchie de composants, définis en termes de contraintes temporelles. L'architecture proposée est dédiée à la spécification et à la simulation des SMA ; néanmoins elle présente une réelle difficulté pour passer de la spécification à l'implémentation, par raffinement; de plus, la vérification est limitée seulement à la vérification par modèles. [GHK 02] utilise une approche multi-formalismes de spécification incluant les statecharts et le formalisme Object-Z. l'auteur propose une approche formelle pour le prototypage et la simulation des SMA. Bien que cette approche permette de spécifier et de simuler, contrairement à d'autres [LIN 95], elle a quelques limites. En effet, le formalisme Object-Z n'est toujours pas exécutable, et donc seulement une analyse par simulations de la spécification décrite par le statecharts est possible.

5. Conclusions

L'objectif de cet article est d'intégrer des méthodes de spécification et de vérification dans le développement basé composants des rôles dans les SMA. Tout d'abord nous avons proposé le modèle RICO, un modèle de spécification et de validation de rôles par approche componentielle ; l'instanciation de ce modèle par le formalisme des Objets CoOpératifs a permis une spécification formelle, une analyse et une validation des interactions ouvertes et complexes basées sur le concept de rôles. Le modèle proposé est générique, et peut être bénéfiquement instancié par d'autres formalismes (cf. section 5, travaux proches). Ensuite, nous avons défini et explicité des propriétés de sûreté et de vivacités de ces composants-rôles. Finalement, nous avons montré comment vérifier ces propriétés comportementales en exploitant la théorie des réseaux de Petri : graphe des marquages accessibles pour l'analyse comportementale, la vérification par modèle ; pour ce faire, nous avons utilisé les fonctionnalités offertes par des outils d'analyses de réseaux de Petri comme l'outil INA.

Remerciements. Ce travail a été supporté par le département STIC-CNRS, SUB/2003/076/DR16, dans le contexte de l'action C2ECL (Coordination et Contrôle de l'Exécution de Composants Logiciels).

6. Bibliographie

[BDJ 97] F. M. T. Brazier, B. Dunin Keplicz, N. Jennings, J. Treur, "Desire: Modelling Multi-agent Systems in a Compositional Formal Framework", IJCIS, 6:67-94, 1997.

[CLZ 03] G. Cabri, L. Leonardi, F. Zambonelli "BRAIN: a Framework for Flexible Role-based Interactions in Multi-agent Systems", Proceedings of CoopIS 2003, 2003.

[CES 86] E. M. Clarke, E.A. Emerson, A. P. Sistla, "Automatic Verification of finite-State Concurrent Systems using Temporal Logic Specifications", ACM Transactions on Programming Languages and Systems, Vol 8, N° 2, 1986, pp244-263.

[DRH 04] M. Dastani, M. B. van Riemsdijk, J.Huslstijn, F. Dignum, J-J. Meyer, "Enacting and Deacting Roles in Agent Programming", AOSE'04.

[FEG 98] J. Ferber, O. Gutknecht, "Aalaadin: A Meta-model for the Analysis and Design of Organizations in Multi-agent System", ICMAS'98, 1998.

[GHK 02] P. Gruer, V. Hilaire, A. Koukam, "Formal Specification and Verification of Multi-agent Systems", ICMAS'2000, IEEE.

[HSB 04] N. Hameurlain, C. Sibertin-Blanc "Specification of Role-based Interactions Components in MAS", Software Engineering for Multi-Agent Systems III: Research Issues and Applications, to appear, LNAI/LNCS, December 2004.

[KEN 02] E. A. Kendall, "Role Modelling for Agent Systems Analysis, Design and Implementation", IEEE Concurrency, 8(2): 34-41, April-June 2000.

[KHW 01] J-L. Koning, M-P. Huget, J. Wei, X. Wang. Extended Modeling Languages for Interaction Protocol Design. AOSE'2001, Springer-Verlag, pp 93-100, 2001.

[MPN 95] Z. Manna, A. Pnueli, "Temporal Verification of Reactive Systems-Safety", Springer-Verlag, 1995.

[MUR 88] T. Murata. Petri Nets: Properties, Analysis and Applications; Proc. of the IEEE, vol. 77, N° 4, pp. 541-580, 1988.

[LIN 95] M. Luck, M. d'Inverno, "A Formal Framework for Agency and Autonomy", ICMAS'95, AAAI Press/MIT Press, editor.

[ROS 99] S. Roch, P. H. Starke, "INA: Integrated Net Analyzer, Version 2.2", Humboldt-Universitat of Berlin, April 1999.

[SIB 01] C. Sibertin-Blanc, "CoOperative Objects : Principles, Use and Implementation", In Petri Nets and Object Orientation, LNCS 2001, Springer-Verlag. 2001.

[SIB 95] C. Sibertin-Blanc et Al., "SYROCO : Reference Manual V7", University Toulouse1, Oct 1996, (C) 1995, 97, CNET and University Toulouse 1.

[SZY 02] C. Szyperski, "Component Software-Beyond Object-Oriented Programming", Addison-Wesley, 2002.

[ZJW 03] F. Zambonelli, N. Jennings, M. Wooldridge, "Developing Multiagent Systems: The Gaia Methodology", ACM TSEM, Vol 12, N° 3, July 2003, pp317-370.