Une approche componentielle pour la modélisation d'agents mobiles coopérants

Min-Jung Yoo, Jean-Pierre Briot

Une approche componentielle pour la modélisation d'agents mobiles coopérants

Min-Jung YOO*, Jean-Pierre BRIOT

Thème OASIS
LIP6 : Laboratoire d'Informatique de Paris 6
Université Paris 6 – CNRS
Case 169, 4 place Jussieu, 75252 Paris Cedex 05, France
E-mail : Min-Jung.Yoo@hec.unil.ch, Jean-Pierre.Briot@lip6.fr

Résumé

Le développement d'une application sur un environnement ouvert distribué comme l'Internet nécessite l'intégration des utilisateurs et de programmes qui travaillent en coopération. Nous nous intéressons à la réalisation de telles applications en utilisant le paradigme d'agents et de systèmes multi-agents (SMA) et en intégrant la technologie d'agents mobiles. Nous avons choisi l'approche componentielle pour faciliter la conception et la modification du modèle d'agents. Le langage SCD permet de modéliser des agents mobiles en utilisant des composants. Nous avons fourni un générateur de code Java (programme Java) à partir du modèle conçu en SCD. Le programme généré est ensuite intégré dans une plate-forme d'agents mobiles. Nous avons choisi un scénario d'agence de voyage comme première application de notre approche. A partir du modèle conçu en SCD, nous avons dérivé le modèle de validation en réseaux de Petri colorés et procédé, en collaboration avec l'équipe SRC, à la validation de certaines propriétés dans notre modèle.

Mots clés : composants, agents mobiles, protocole de coopération, Java, Voyager, commerce électronique, validation, réseaux de Petri colorés

1. INTRODUCTION

Du fait de la popularisation de l'Internet et de la technologie du Web, la réalisation des applications coopératives dans cet environnement distribué est de plus en plus nécessaire. La tendance actuelle est de donner la capacité à coopérer entre des applications dispersées sur les réseaux pour fournir un service aux utilisateurs. Dans ce contexte, nous nous sommes intéressés à la réalisation d'applications coopératives en appliquant la technologie d'agents et de système multi-agents (SMA).

Comme la plupart des systèmes complexes, les problématiques générales de génie logiciel doivent être prises en compte pendant la modélisation de systèmes multi-agents. Nous considérons particulièrement :

1) La facilité de la conception,
2) L'efficacité de l'implémentation,
3) La validité du modèle.

Des travaux déjà réalisés dans le domaine des systèmes multi-agents n'ont pas beaucoup considéré les problématiques citées (1-3) de façon intégrée dans un cycle de développement d'agents logiciels. La plupart des plates-formes d'agents et de SMA concernent la facilité et l'efficacité de la réalisation des

* Min-Jung Yoo est actuellement Professeur assistante à l'École des Hautes Études Commerciales (HEC) de l'université de Lausanne, Suisse. La recherche décrite dans ce rapport a été effectuée pendant son doctorat au LIP6.

agents et l'aspect validation formelle n'est guère traité. Il y a déjà des travaux existants, dans le domaine des SMA, concernant l'utilisation de modèles formels comme les réseaux de Petri, tantôt pour la validation [vonMartial 92][ElFallah Haddad 94], tantôt pour la modélisation et la simulation [Moldt Wienberg 97], mais la réalisation du modèle est souvent empirique.

Il est nécessaire d'intégrer les trois problématiques de génie logiciel mentionnées ci-dessus dans une plate-forme de développement de SMA en considérant les caractéristiques de notre système cible : des applications coopératives distribuées qui peuvent être développées sur plusieurs sites par différents concepteurs. Par une méthodologie intégrée - de la conception, de la validation et de la réalisation -, le système multi-agent peut être obtenu avec certaines assurances de fonctionnement dans un environnement ouvert.

Notre recherche a donc pour but d'étudier l'intégration des trois différentes étapes (conception, validation, réalisation) dans un cycle de développement d'agents logiciels.

Dans le paragraphe suivant, nous expliquons les caractéristiques de notre système cible et des problématiques à considérer.

1.1 Exemple de l'agence de voyage

Le service de l'agence de voyage est un exemple classique dans le domaine des systèmes de commerce électronique, souvent pris comme étude de cas ([FIPA 97] [Merlat 98] [Linden 96], etc.). Le système multi-agent de l'agence de voyage est constitué d'un ensemble d'agents reliés par l'intermédiaire de l'Internet et qui fournissent divers types de services :

- des agents représentant des compagnies aériennes, Air France, United Airlines, British Airways, etc. Il fournissent des services de transport aérien.
- des agents représentant des compagnies ferroviaires, SNCF par exemple.
- des agents représentant des hôtels ou des compagnies de location de voitures.

L'agent assistant de voyage (agence de voyage) travaille pour son utilisateur afin de superviser l'organisation de voyages : il facilite les tâches d'utilisateur concernant la sélection des serveurs de voyage, la planification d'itinéraire, l'organisation de séjours, etc. Pour ce faire, il doit coopérer avec d'autres agents représentant des services de voyage.

Le voyageur demande à son agent assistant de voyage d'organiser un voyage de Bordeaux à NewYork pour 7 nuits. C'est cet agent qui sélectionne des agents serveurs de voyage et organise l'itinéraire : par exemple, il effectue les réservations des billets de train (de Bordeaux à Paris), le vol (de Paris à New-York) et la réservation de l'hôtel (à New-York pour 7 nuits), etc.

Nous nous intéressons en particulier aux caractéristiques suivantes du système :

1) ouverture de l'environnement : De nouveaux agents assistants ou serveurs peuvent être introduits ou retirés. On ne peut pas savoir où s'arrête ce système, ni le nombre d'agents qui travaillent dans ce système.
2) coopération entre des agents : Les besoins de l'utilisateur nécessitent que son agent « agence de voyage » coopère avec d'autres agents pour fournir un service satisfaisant l'utilisateur : coopération simple comme une simple relation de « client/serveur » - par exemple, la demande des informations sur les services de voyage (horaires, prix, etc.) et la réponse donnant des informations concernées -, ou celle plus complexe réalisée par plusieurs messages, par exemple, pour trouver un vol de prix minimal les agents vont coopérer suivant le protocole d'appel d'offre (figure 1).
3) la mobilité d'agents : la mobilité d'agents aide à économiser les ressources de réseaux. Par exemple, après avoir choisi un agent serveur de voyage, l'agent assistant de voyage migre sur le site du serveur pour le processus de réservation.

Notre objectif est de trouver une méthodologie de conception pour aider à la modélisation d'un tel système multi-agents en tenant compte de l'intégration des différentes étapes de modélisation. Le paragraphe suivant détaille les problématiques à considérer pour cet objectif.

1.2 Problématiques

En considérant les trois problématiques de génie logiciel, notamment : 1) la facilité de conception, 2) l'efficacité de la réalisation, et 3) la validité du modèle, nous soulevons les questions suivantes pour la modélisation et la réalisation des agents dans le scénario :

1) comment réaliser la modularité et la réutilisabilité de certains modules pour la modélisation d'agent et leurs protocoles de coopération ?
2) comment modéliser et intégrer la mobilité d'agent ?
3) comment valider certaines propriétés dans un système multi-agents ?

1.2.1 Modularité dans le modèle d'agent

Bien qu'il existe de nombreux modèles d'agents, qui sont différents les uns des autres, on peut considérer qu'un modèle d'agent peut être simplifié en deux comportements distincts [Ferber 96] : (1) le fonctionnement interne, (2) le comportement externe (l'interface). Le fonctionnement interne d'agent concerne diverses caractéristiques d'agents, par exemple le comportement réactif ou délibératif, la planification ou l'apprentissage, etc. Le comportement externe représente l'aspect communication et coopération, ou la perception de son environnement et l'action dans cet environnement.

Dans notre exemple, le fonctionnement interne d'agents est une tâche (un service) spécifique aux agents, par exemple le service informatique de voyage en vol ou en train, etc., pour des agents serveurs de voyage. Pour l'agent assistant de voyage, un mécanisme de sélection du service de voyage et de planification caractérise le fonctionnement interne. Dans le contexte du scénario, les agents des compagnies de transports « encapsulent » en général les systèmes d'information de ces compagnies, qui sont des « legacy systems » [Wooldridge Jennings 98] : les agents assurent en fait l'interface pour permettre à ces systèmes d'intervenir dans des transactions électroniques.

Il est souhaitable d'avoir un modèle d'agent qui est facile à appliquer pour la réalisation des applications coopératives sous la forme d'agent.

1.2.2 Modèle de coopération entre les applications

En ce qui concerne la coopération des agents dans l'exemple, le concepteur doit résoudre les problématiques suivantes :

- comment faire communiquer les applications hétérogènes ?
- comment modéliser facilement des protocoles de coopération ?
- comment intégrer le protocole de coopération d'agent modélisé dans le modèle (ou architecture) d'agent ?

Des langages de communication inter-agent traitent le premier point ([Petrie 96][Genesereth 97]). L'agent qui est capable d'interpréter un langage ACL ("Agent Communication Language") est qualifié d'agent à messages typés « typed-message agents »). Divers ACL existent : le langage KQML [Finin et al. 97] et celui de FIPA [FIPA 97].

Pour modéliser des agents en coopération, le concepteur d'agents est souvent obligé de définir une suite d'échange des messages, c'est-à-dire une conversation d'agents. Cet échange de message suit une certaine séquence attendue. De telles conversations sont définies par leurs protocoles (appelés

« protocoles de coopération » ou « protocole de conversation »), qui indiquent les séquences autorisées de messages [Von Martial 92] [Burmeister et al. 93] [Demazeau 95].

Une grande variété de protocoles existent : par exemple le protocole d'appel d'offre « Contract net protocol » ([Smith 80]), divers protocoles dans les travaux de [Demazeau 95]. Récemment, FIPA propose quelques protocoles génériques, par exemple « Dutch auction protocol », « iterated contract net protocol », etc. [FIPA 97].

Ces protocoles de coopération ont leur propres caractéristiques, qui les rendent adaptés à certaines situations seulement. C'est pourquoi, il peut être nécessaire de changer ou mettre à jour les protocoles de conversation après modification du système multi-agents, comme par exemple après l'introduction de nouveaux agents ou le changement de mode de coopération.

1.2.3 Modélisation de la mobilité d'agents

La technologie d'agents mobiles est en fort développement, mais son intérêt réel reste encore contesté. Tout d'abord, il y a quelques problèmes posés par la mobilité d'agent, par exemple, le problème de sécurité, de confidentialité (« privacy »), ou le problème de gestion des agents mobiles, etc. Nous tentons d'évaluer l'apport réel de la mobilité dans le cadre du développement d'une architecture d'agents avec coopération.

Plusieurs plates-formes d'agents mobiles sont commercialisées, notamment Aglet, Odyssey, Voyager, etc. Ce sont des études qui s'occupent plutôt du développement d'un environnement d'agents mobiles concernant la transmission du code d'agents, la réception du code et la re-création d'agents, la communication locale entre agents, etc. Aucun modèle d'agent au sens de la définition d'« agent » dans le domaine des SMA [Demazeau Müller 91] n'est fourni. La réalisation d'agents mobiles se réduit donc à la programmation directe sur cet environnement.

Souvent, la programmation d'agents mobiles sur les plates-formes pose quelques difficultés, et plus particulièrement les deux difficultés suivantes :

1) la gestion de la migration d'agent avec de multiples tâches (threads),
2) le problème de reprise d'une activité après la migration.

Pour réaliser des activités internes en parallèle dans un modèle d'agent, on utilise des threads au niveau de la programmation. Mais un agent qui contient plusieurs threads ne peut pas être « sérialisé (‘serialised') » pour migrer vers un autre site. Une gestion de multiple threads, en les détruisant avant la migration et en les récupérant après la migration, est nécessaire.

Comme la plupart des plates-formes d'agents mobiles ne garantit pas la transmission du vecteur d'état qui décrit l'état précis de l'objet (au sens de l'unité d'exécution dans une plate-forme d'agents mobiles) juste avant la migration, la reprise d'activité après la migration doit être réalisée par le concepteur.

1.2.4 Problème de validation

Parce qu'un système ouvert évolue, sa validité est toujours remise en cause. Avant de lancer un agent dans un système multi-agent ou après avoir modifié certains comportements d'agents, le concepteur veut une validation rigoureuse pour que l'agent puisse travailler avec des autres sans erreurs.

Dans le domaine des systèmes multi-agents, le moyen le plus souvent utilisé pour valider certaines propriétés est de le simuler directement dans une plate-forme cible (exécution du modèle opératoire). Mais en considérant les caractéristiques de notre système - un environnement ouvert - il est difficile de valider le modèle de telle façon. Ceci parce que l'environnement d'exécution des agents n'est pas limité à une plate-forme ou à un système local. Cet environnement est réparti sur plusieurs plates-

formes d'agents liées par des réseaux. En conséquence le degré de confiance en une validation empirique est moins crédible. C'est pourquoi nous nous sommes intéressés à une validation formelle.

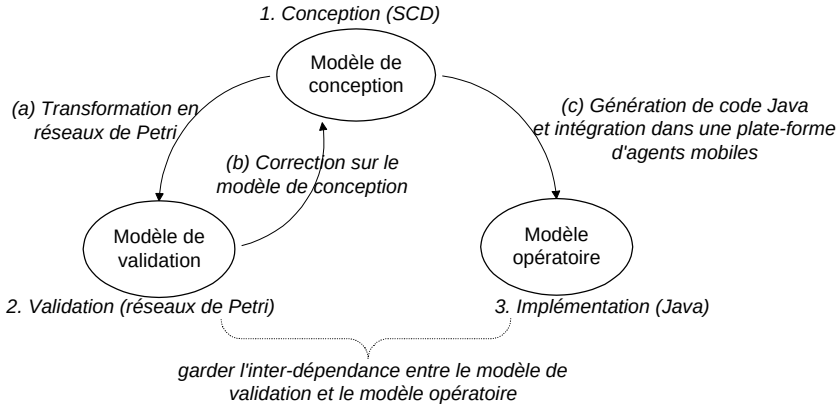
Dans le domaine de l'IA distribuée, la réécriture du modèle d'acteurs en réseaux de Petri a été souvent menée ([Sami Vidal-Naquet 91] [Vernadat et al. 95] [Agha Sami 92]) pour sa capacité de spécification des processus concurrents. Dans le domaine des SMA, le formalisme a été utilisé pour d'une part la simulation du parallélisme dans un modèle de système multi-agents ([Moldt Wienberg 97]), d'autre part la validation de protocoles de conversation d'agents ([von Martial 92]) ou du plan global d'agents ([ElFallah Haddad 95]). Malgré le fait que la possibilité d'une validation formelle est un des avantages principaux des réseaux de Petri, beaucoup de recherches n'ont pas clairement montré l'aspect validation en termes de réseaux de Petri dans leurs travaux. Ceci tend à montrer qu'il est difficile d'obtenir des résultats significatifs du fait de problèmes tels que : taille du réseau, non-dynamicité, énumération nécessaire des valeurs.

2. APPROCHE SUIVIE

Nous avons étudié les problématiques à considérer pendant la modélisation des applications coopératives. Il est donc nécessaire d'utiliser une méthodologie particulière. En effet, en reposant généralement sur un modèle unique, les méthodologies existantes ne parviennent pas à intégrer de manière satisfaisante tous les besoins.

Dans le schéma suivant, nous présentons notre environnement de développement d'agents qui est composé de trois étapes avec trois types de modèles associés :

1) modèle de conception : conception d'agents en utilisant un langage de description des composants qui est à la fois traduisible en un modèle de validation et transformable en un modèle opératoire.
2) modèle de validation : le modèle sur lequel l'étape de validation est réalisée. La transformation du modèle en réseaux de Petri concerne uniquement le modèle de coopération d'agents qui a été conçu sous forme de composants de protocole de coopération.
3) modèle opératoire : implémentation des agents sur une plate-forme d'agent mobile et exécution.



<Figure 1. Etape de modélisation d'agents>

Nous détaillons, dans les paragraphes suivants, chaque étape dans le cycle de développement.

2.1 Modèle d'agent

Notre analyse du paragraphe 1.2 montre que les agents ont besoin, dans leurs comportements, des caractéristiques suivantes :

- le mécanisme de communication suivant un langage ACL,
- la capacité de coopération suivant une convention partagée, pré-définie,

- la tâche spécifique à chaque agent pour le fonctionnement interne.

Ces caractéristiques sont souvent reprises partiellement pour la réalisation des agents. Mais comme cité dans [Kendall 98] ou [Wooldridge Jennings 98], au lieu de réutiliser des modèles ou des systèmes existants, les gens préfèrent développer leurs propres agents. Ceci parce qu'il est difficile de modifier ou d'adapter un ancien modèle par rapport à un nouveau besoin même s'ils partagent certaines caractéristiques. Une des raisons est que l'on ne dissocie pas clairement chaque caractéristique au moment de la réalisation du modèle. Dans certains cas, le modèle de la partie coopération [Barbuceanu Fox 95] est intégré dans le modèle de communication de langage ACL. Nous modélisons les différentes caractéristiques par des sous modules clairement séparés dans une architecture d'agent. Chaque sous module interne d'agent est réalisé par des composants qui peuvent être connectés et remplacés facilement. Nous avons réalisé l'architecture d'agents en utilisant un framework.

2.2 Langage de description,

Nous avons choisi l'approche componentielle pour la modélisation d'agents. Du fait de la caractéristique évolutive, la première exigence d'un système ouvert est sa flexibilité, c'est-à-dire la facilité à se modifier au cours de son cycle de vie. Une partie du service doit pouvoir être modifiée sans remise en cause des autres [Nierstrasz Meijler 94].

Pour la modélisation d'agents en utilisant des composants, nous avons besoin d'un langage de description de composants logiciels (modèle de conception) qui satisfait les caractéristiques suivantes :

- pour passer à l'étape de validation (étape a de la figure 1), la description du modèle de conception doit être traduisible dans le modèle de validation, c'est-à-dire en réseaux de Petri.
- pour passer à l'étape d'implémentation (étape c), il faut pouvoir traduire le modèle de conception en un modèle opératoire (en Java).
- facilité de modélisation des protocoles de coopération.
- capacité de modéliser la mobilité d'agents.
- encapsulation de services existants (« legacy systems »).

Parmi les langages de description existants de composants ou d'agents, il y a peu de descriptions qui satisfont tous les besoins. Certains langages, par exemple OLAN [Bellisard et al. 95], permettent de modéliser des systèmes distribués par une approche componentielle, mais il est difficile d'obtenir un modèle de validation. Certaines descriptions existent pour la modélisation d'agents, par exemple SodaBot [Coen 94], LALO [Gauvin et al. 97]. Mais la facilité de modification du modèle par une approche componentielle n'est pas considérée. Le langage SCD a été conçu pour satisfaire ces objectifs. Le langage permet à la fois de modéliser des agent et de le traduire en un modèle de validation basé sur les réseaux de Petri colorés. Il permet également de modéliser l'aspect mobilité d'agent. Le modèle d'agent décrit en SCD est ensuite traduit en code Java qui peut être intégré sur une plate-forme d'agent mobile.

2.3 Modélisation de la mobilité d'agents et sa réalisation

La réalisation d'agents mobiles est souvent faite par la programmation directe dans une plate-forme d'agents mobiles. De ce fait, il n'est pas facile de transférer un modèle d'agent réalisé sur une plate-forme d'agents mobiles, par exemple Aglet, sur une autre plate-forme, par exemple Voyager, même si des agents partagent certaines caractéristiques au niveau du comportement. Ceci parce que l'on n'a pas dissocié la modélisation de haut niveau d'agents (le niveau conceptuel) et la réalisation de bas niveau des objets mobiles (le niveau technique) dédié à une plate-forme.

Pour résoudre cet inconvénient, nous essayons de combiner la technologie existante des agents mobiles avec la méthodologie de conception d'agents. Nous avons dissocié clairement le *modèle conceptuel* d'agent (modèle 1 de la figure 1) du *modèle structurel physique* (modèle 3) de la réalisation d'agents mobiles. Ensuite nous avons fourni un moyen d'implantation du modèle conceptuel à l'aide du modèle physique (étape c de la figure 1). Nous avons choisi deux plates-formes d'agents mobiles pour l'expérimentation, notamment JNA [Merlat 98] et Voyager, pour montrer la faisabilité de notre approche.

Nous avons soulevé dans le paragraphe 1.2.3 les difficultés de la programmation d'agent sur une plate-forme d'agents mobiles actuelles. Nous avons résolu certaines difficultés en réalisant notre environnement de composants.

2.4 Problème de validation

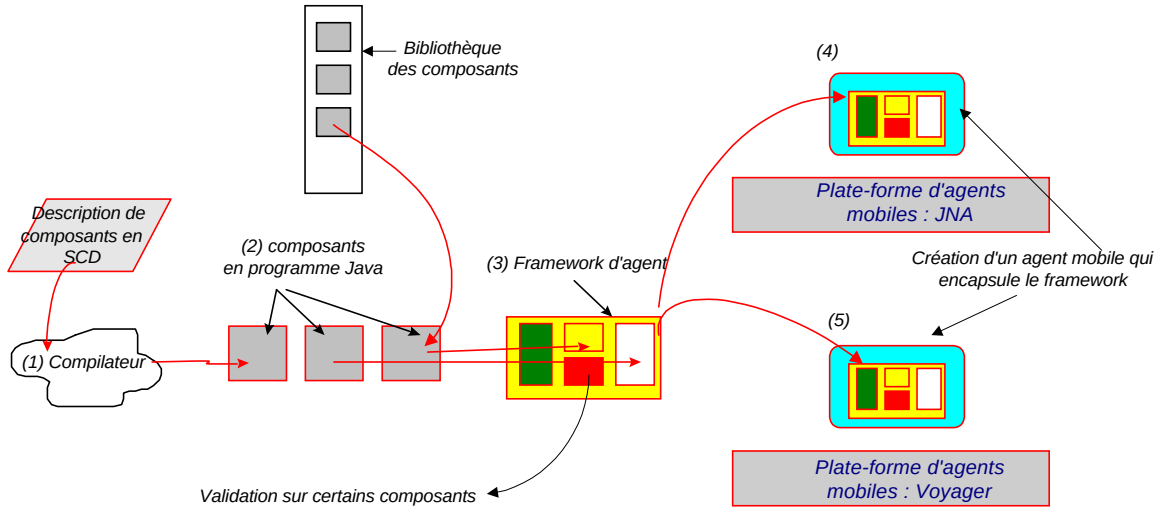
En ce qui concerne la portée de la validation, nous nous somme intéressés à valider le modèle de *coopération* d'agents. Plusieurs travaux ont déjà été menés sur l'utilisation de réseaux de Petri, soit pour valider le modèle de conversation d'agents ([VonMartial 92], [Chauhan 98]), soit pour un formalisme de spécification de protocoles de conversation ([Cost et al. 99]). Le résultat du travail de [Burkhard 93] justifie également notre choix pour le modèle de validation. Les propriétés liées au modèle de coopération des agents sont capables d'être obtenues uniquement par l'intermédiaire d'un modèle global du système.

L'importance de la validation pendant la modélisation du protocole de coopération est aussi soulignée par [von Martial 92]. Des réseaux de Petri à prédicats ont été utilisés pour vérifier le modèle de protocole de coopération conçu. Notre démarche est basée sur le même principe. Nous utilisons des réseaux de Petri colorés pour assurer certaines propriétés dans le modèle de coopération d'agents du SMA qui est exprimé sous la forme de protocoles de coopération.

Nous avons traduit le modèle de protocoles de coopération d'agents décrit en SCD en réseaux de Petri colorés, selon une règle de transformation. Nous avons validé des propriétés, par exemple, la vivacité et le non-blocage.

2.5 Environnement de développement.

La figure 2 montre la chaîne de conception d'agents mobiles dans notre environnement de développement. Le concepteur modélise des agents en utilisant des composants existants ou des composants conçus par le concepteur. Le composant modélisé en utilisant la description (SCD) est ensuite traduit en code Java par le générateur de code (1. le compilateur dans la figure 2). Les composants en code Java (2) sont compilés dans un environnement Java (ex. commande 'javac') et intégrés dans un modèle d'agent composite (3) qui est un framework d'agent. Ce framework est encapsulé dans une structure d'agent mobile (4, 5) fournie par une plate-forme d'agents mobiles.



<Figure 2.Chaîne de conception d'agents mobiles>

3. EXEMPLE SIMPLE : AGENCE DE VOYAGE AVEC PROTOCOLE D'APPEL D'OFFRE

Dans ce paragraphe, nous présentons un exemple de la réalisation d'agents à l'aide de notre approche pour le modèle d'agents qui communiquent suivant le protocole d'appel d'offre. Nous présentons tout d'abord dans le paragraphe 4.1, le langage de description de composant SCD avec lequel nous avons modélisé des composants du modèle d'agents.

3.1 Langage de description des composants : SCD (SoftComponent Description)

Pour utiliser des composants pour la conception d'agents, nous proposons le langage de description SCD (SoftComponent Description). Les caractéristiques du langage sont les suivantes :

1) Il est facile de modéliser les protocoles de coopération entre agents. La syntaxe SCD est basée sur la notion d'état/transition. Ceci permet de décrire facilement des protocoles de coopération d'agents. Egalement, la traduction de SCD vers le formalisme de réseaux de Petri colorés est possible grâce à sa syntaxe.
2) SCD permet de modéliser la migration d'agent comme une action intégrée dans un protocole de coopération. La modélisation d'un agent mobile demande à indiquer le moment où l'agent doit se déplacer. Le concepteur peut modéliser un protocole de coopération en intégrant la mobilité d'agent qui est synchronisée avec d'autres comportements. La mobilité est ensuite traduite par notre générateur en code Java qui déclenche la migration d'agents.
3) SCD permet d'encapsuler un module programmé en Java comme un composant connectable. Ceci facilite le développement d'un service réel en permettant l'utilisation de technologies courantes basées sur Java (e.g., Applet, Java Beans, JDBC, etc.) qui sont souvent essentielles au développement d'un « legacy » système.

La syntaxe SCD est donnée dans l'annexe.

3.2 Exemple : Scénario 'agence de voyage'

3.2.1 Présentation

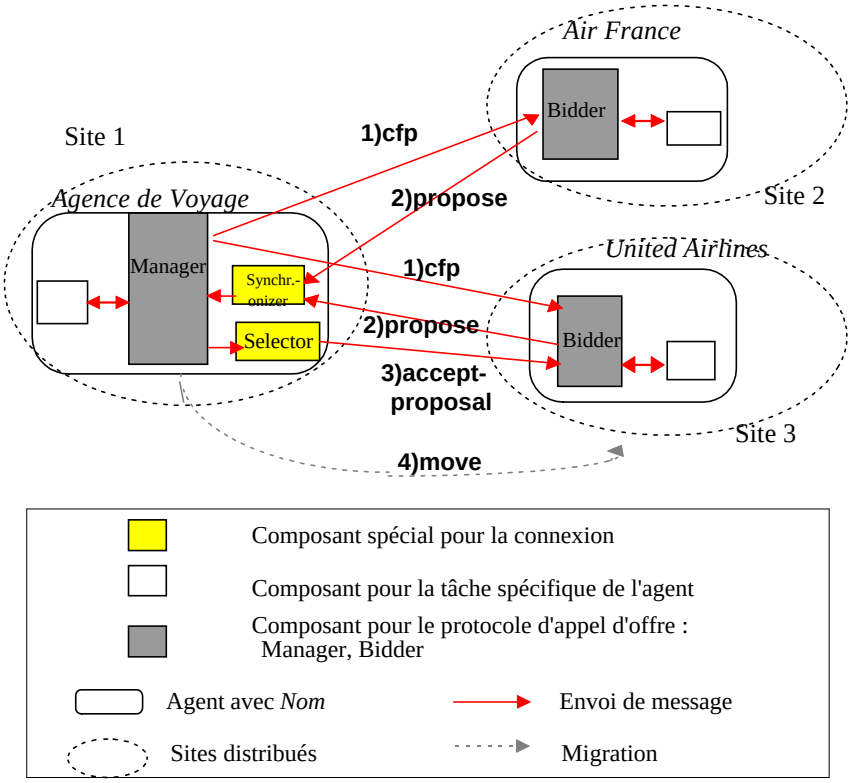
A partir du modèle de système 'agence de voyage', imaginons la situation suivante :

Situation : **trouver un vol de Paris à New York avec un prix minimum**

L'agence de voyage examine le prix du billet d'avion entre Air France et United Airlines pour choisir un bon prix parmi les propositions. Après avoir sélectionné une compagnie aérienne, il migre vers le site du serveur choisi pour finaliser la réservation.

Le protocole d'appel d'offre (« Contract Net » [Smith 80]) est l'exemple classique pour réaliser ce genre de coopération. La figure 3 montre la configuration des agents et leurs composants, avec le protocole d'appel d'offre entre l'agence de voyage et deux agents serveurs de compagnies aériennes, par exemple Air France et United Airlines.

Nous avons conçu quelques protocoles de coopérations en SCD [Yoo 99] : par exemple le protocole d'appel d'offre (« Contract Net »), 'FIPA-Query-référence', 'FIPA-request', 'Subscribe'. Chaque protocole de coopération est modélisé par deux types de composants : (i) l'un qui doit être intégré dans le modèle d'agent qui débute le protocole (« protocol invoker »), et (ii) l'autre qui doit être intégré dans le modèle d'agent participant au protocole (« protocol invokee »). Par exemple, le protocole d'appel d'offre est modélisé par les composants : « Contract_Manager » (M dans la figure 3) et « Contract_Bidder » (B dans la figure 3). Si le concepteur veut modéliser un agent qui joue le rôle de manager du protocole d'appel d'offre, il utilise le composant « Contract_Manager » et réalise la connexion avec d'autres composants. Dans cette article, nous ne prenons que le protocole d'appel d'offre comme un exemple simple et suivons notre méthodologie de conception pour notre environnement de développement d'agents.



<Figure 3. Configuration des agents dans le protocole d'appel d'offre>

3.2.2 Modèle de coopération des agents

Dans les paragraphes suivants, nous illustrons quelques situations possibles dans le scénario en exposant l'utilisation des protocoles de coopération.

Le protocole d'appel d'offre (« Contract Net » [Smith 80]) est l'exemple classique pour réaliser ce genre de coopération.

La figure 3 montre la configuration des agents et leurs composants, avec le protocole d'appel d'offre entre l'agence de voyage et deux agents serveurs de compagnies aériennes, par exemple Air France et United Airlines.

3.2.3 Composants spéciaux

En réalisant le protocole d'appel d'offre, nous considérons quelques composants spéciaux qui complètent la communication entre agents, notamment 'Synchronizer' et 'Selector'. L'implémentation de ces composants est réalisée directement en Java.

Le rôle du composant 'Synchronizer' est de synchroniser des messages d'entrée en attendant jusqu'à ce que tous les messages arrivent aux ports d'entrées. Ce composant a un port de sortie et peut avoir plusieurs ports d'entrées. Le composant 'Selector' a pour but de sélectionner un agent à qui le composant renvoie le message arrivé à l'entrée du composant.

L'utilisation de ces composants spéciaux (primitifs) est limitée à la composition avec d'autres composants SCD dans un composant composite. Voir par exemple la Table 5, les commentaires notés (1) et (2).

3.3 Modélisation des agents en utilisant SCD

Nous avons conçu des composants internes d'agents en utilisant le langage SCD. Par exemple, les deux composants de protocole d'appel d'offre, la partie Manager et la partie coopérant sont modélisés comme suit.

Par exemple, le composant « Contr_Manager » pour le protocole d'appel d'offre est modélisé en SCD comme suit :

```
DEFCOMPO     Contr_Manager {

   SUPER SoftComponent
   VAR {  condt : VoyageDescr ;
          bidList : Vector                                  // List of ServiceDescr ;
          award : ServiceResult ;
          sender, receiver, dest : AgentAddress ;
          receiver_list : AgentAddress;}
   MSGIN { beginContract (receiver_list, condt), propose (sender, bidList)}
   MSGOUT {cfp (receiver_list, condt), accept-proposal (receiver, award)}
   REQUEST {selectService(bidList) RETURNTYPE ServiceResult}
                                                      // invocation de communication synchrone

   STATE {init (INITIAL 1), wait_bid, plDistance}     // état de conversation.

   OPERATION {
          action1 {
                 ONSTATE {init}
                 INPUT {beginContract (receiver_list, condt)}
                 THEN  SEND cfp(receiver_list, condt)}
                                                      // Distribution de message 'cfp' à tous les agents dans la liste
                                                      // 'receiver_list' avec même paramètre (condt)
                        NEXTSTATE {wait_bid}
                 END}

          action2 {
                 ONSTATE {wait_bid}
                 INPUT {propose (sender bidList)}
                 IF (size (bidList) >0)
                 THEN {
                        ASK selectService FOR award ;
                        receiver = award.receiver() ;
                 SEND {accept-proposal (receiver, award)}
                        MOVETO (receiver)
                        NEXTSTATE {plDistance}
                 }
                 NEXTSTATE {init}
}}
```

<Table 1. Composant 'Contr_Manager' en SCD>

Le composant pour le protocole d'appel d'offre pour la partie contractant est modélisé en SCD comme suit.

```
DEFCOMPO Contr_Bidder {

  SUPER SoftComponent
  VAR {   condt : VoyageDescr ;
          bid : ServiceDescr ;
          award : ServiceResult ;
          sender, receiver : AgentAddress ; }

  MSGIN {cfp (sender, condt), accept_proposal(sender, award)}
  MSGOUT {propose (receiver, bid), beginService (award)}
  REQUEST {hasService() RETURNTYPE boolean, makeBid (condt) RETURNTYPE ServiceDescr}

  STATE  {init, BidCondition }

  OPERATION {

          tr1 {       ONSTATE {init}
                      INPUT {cfp (sender, condt)}
                      THEN {
                              receiver = sender ;
                              NEXTSTATE {BidCondition}
                      }

          tr2 {       ONSTATE {BidCondition}
                      IFEXT hasService (condt)
                      THEN
                        ASK  makeBid (condt) FOR bid        // bid = makeBid (condt)
                        SEND {propose (receiver, bid)}
                      END}

          tr3 {       INPUT {accept_proposal (sender, award)}
                      THEN
                        SEND {beginService (award)}                          // by late binding
                      END}

  }       //End Operation
}         //End Definition
```

<Table 2. Le composant 'Contr_Bidder' en SCD>

Le composant de tâche qui est connecté au composant 'Contr_Manager' est programmé en Java comme suit :

```
package agentTest.contractNet;

.....
public class Manager_Task  {

   public Manager_Task () {
          super();
   }

   public Award selectService (Vector bidList) {                    // list of 'Bid's
                  System.out.println (" __________   in Method selectService");
                  System.out.println (" _______    Vector bidList  = "+ bidList);
          Enumeration allBids = bidList.elements();
          Bid compared;
          Bid best = (Bid) allBids.nextElement();
             System.out.println ("Vector for BEST = "+ best.toString());

          while (allBids.hasMoreElements()) {
             compared = (Bid) allBids.nextElement();                    //[t112 paris lille 10:00 .. 300]
                  System.out.println ("Vector for COMPARED = "+ compared);

             if (compared.calculValue() > best.calculValue())
                  best= compared;
          }
          Award response = new Award (best.getAddress(), best.id());
          return (response);
   }

   public AgentAddress getDest (Award award) {
                  //  ........}
   String findValue (Vector bidList) { // last item of the list is the compared value
                  // ........}
   String findKey (Vector bidList) {
```

```
                    //  ........}
   boolean calculVal (Vector compared, Vector best) {        // compared > best testing
                    //  .......}
}
```

<Table 3. Code source en Java pour le comportement interne de la partie 'contract manager'>

Le composant pour la tâche spécifique à l'agent ADV est modélisé en encapsulant cette classe Java comme suit :

```
DEFCOMPO  Manager_Task_Compo {
   SUPER SoftComponent
   VAR {bidList : Vector ;}                    // Liste de ServiceDescr

   REPLY {selectService(bidList)  RETURNTYPE ServiceResult}
                                               // Ce composant est connecté par un message synchrone
   ENCAPSULATE   Manager_Task AS myWork
                                               // Ce composant encapsule une classe Java 'Manager_Task.class '
   BIND { selectService TO  selectService }}
                                               // Le port de communication 'selectService' de ce composant est lié
                                               // à la méthode 'selectService de l'objet
```

<Table 4. Composant pour encapsuler le comportement interne>

Les descriptions suivantes montrent le modèle d'agent « agence de voyage » avec ses sous composants que nous avons modélisé dans [Yoo 99].

```
DEFCOMPO  AgenceDeVoyage  {

          VAR  {.....}
          MSGIN { .....}
          MSGOUT {....}
          SUBCOMPONENT {
                    .....
                    protocole_manager ISA Contr_Manager {          // composant de protocole
                              INPORT {beginContract, propose}
                              OUTPORT {selectService, cfp, accept_proposal}
                    }
                    waitBids  ISA  Synchronizer  INPORT propose          // composant spécial  (1)
                    sendAward  ISA Selector  OUTPORT accept_proposal     // composant spécial  (2)

                    manager_task ISA Manager_Task_Compo {          // composant de tâche
                              INPORT {selectService}
                    }
                                                                                    }
          CONNECTION {
                    .....
                    ta_task  beginContract          TO  protocole_manager
                    protocole_manager  selectService TO manager_task
                    protocole_manager  accept_proposal  TO sendAward
                    waitBids  bid TO  manager_role
                    .....
          }
 }
```

<Table 5. Composition des sous composants pour le comportement d'agent 'Agence de Voyager'>

3.4 Génération de code

Nous avons modélisé des agents en SCD et généré du code Java. La table 6 montre le code généré en Java à partir de la description du composant 'Contr_Manager' de la table 1. Le nom du module en Java porte systématiquement le nom du composant défini dans la description en SCD.

```
 package agentTest.contractNet;                    // inséré à la main après avoir généré du code

import softCompo.softBase.*;
import java.util.*;
import java.lang.*;
import softAgent.*;                                // inséré à la main après avoir généré du code
import agentTest.contractNet.*;                    // inséré à la main après avoir généré du code

     // Primary Component
public class Contr_Manager  extends SoftComponent  {

   Place plDistance = new Place("plDistance");
```

```
  Award award;

  public Contr_Manager() {
     super ();
  }

  public Contr_Manager(String name, String msgin, String msgout) {
     super (name, msgin, msgout);
   }

  void move(VSoftAgent dest)  {
                System.out.println (" ------  BEGIN move  --------");
     moveTo (dest);
  }

  void beginContract(Condition condt)  {
                System.out.println (" ------  BEGIN beginContract  --------");
      sendMessage ("cfp", loadArgument (new Vector(), condt));
  }

  void propose(Vector bidList)  {
                System.out.println (" ------  BEGIN propose  --------");
     if ( bidList.size() >0 ) {
        award = (Award) require ("selectService", loadArgument (new Vector(), bidList));
        receiver = award.eceiver() ;
        sendMessage ("accept-proposal", loadArgument (new Vector(), award));
        move (receiver) ;
     }
}

  void _after_move()  {
                System.out.println (" ------  BEGIN _after_move  --------");
     plDistance.appendToken();
  }

  public void messageDispatch (String msgName, Vector params) {
     int i = registerNo(msgName).intValue();
     switch (i) {
        case 1 :
           VSoftAgent dest = (VSoftAgent) unloadArgument(params);
           move(dest);
           break;
        case 2 :
           Condition condt = (Condition) unloadArgument(params);
           beginContract(condt);
           break;
        case 3 :
           Vector bidList = (Vector) unloadArgument(params);
           propose(bidList);
           break;
        case 4 :
           _after_move();
           break;
     }
  }

}

/* End of Compilation */
```

<Table 6. Code en Java généré par le compilateur>

Ensuite, on compile tous les codes sources générés en bytecode Java (en utilisant par exemple la commande ‘javac’ dans un environnement JDK) comme suit :

> javac Contr_Manager.java

> javac Contr_Bidder.java

> javac Manager_Task.java

> javac AgenceDeVoyager.java

> .....

Après avoir produit des classes Java qui représentent des composants exécutables, le concepteur les encapsule dans une structure d'agent mobile à l'aide de la classe 'SoftAgent' adaptée à chaque plate-forme d'agents mobiles et fournie dans l'environnement de développement. L'encapsulation du composant SCD dans une structure d'agent mobile est réalisée par une définition simple de la classe d'agent.

Nous avons réalisé deux différents types de 'SoftAgent' pour l'implémentation sur deux plates-formes d'agents mobiles : JNA [Merlat 98] et Voyager [Voyager 97]. JNA est une plate-forme d'agents mobiles développée dans le but d'avoir une plate-forme expérimentale dans le cadre de la thèse de [Merlat 98] et avec le concours de Claude Seyrat lors de son stage de DEA au LIP6 [Seyrat 96].

Voici ci-dessous, à titre d'exemple, le code d'encapsulation du composant 'Travel_Agency' :

```
package  DemoAgent.SoftAgent ;

import java.lang.* ;
import jafa.io.* ;
import softCompo.softBase.* ;
import softAgent.* ;

public class Agent_TravelAgency extends SoftAgent  {

        public Agent_TravelAgency () {
                super ("travel-agency-agent",                                   (1)
                        new Travel_Agency ("travel-agency-compo",               (2)
                                " beginQuery  refuse inform-ref agree  inform ",  (3)
                                "query-ref request "),                                  (4)
                        "travel_agency");                                               (5)
        }
}
```

Le code du constructeur de la classe Agent_TravelAgency inclut les informations suivantes :

1) le nom de l'agent,
2) instanciation du composant interne, avec le nom de composant comme premier paramètre d'instanciation,
3) liste des messages en entrée vers ce composant,
4) liste des message en sortie de ce composant,
5) le rôle de l'agent, i.e., le type d'agent : agent serveur de voyage, agent agence de voyage dans notre exemple.

Pour tester l'exécution des agents dans une plate-forme d'agents mobiles, ce code a été rajouté à la main en décrivant la définition de chaque agent dans le scénario pour créer une sous classe d'agents 'SoftAgent' qui encapsule le composant SCD compilé.

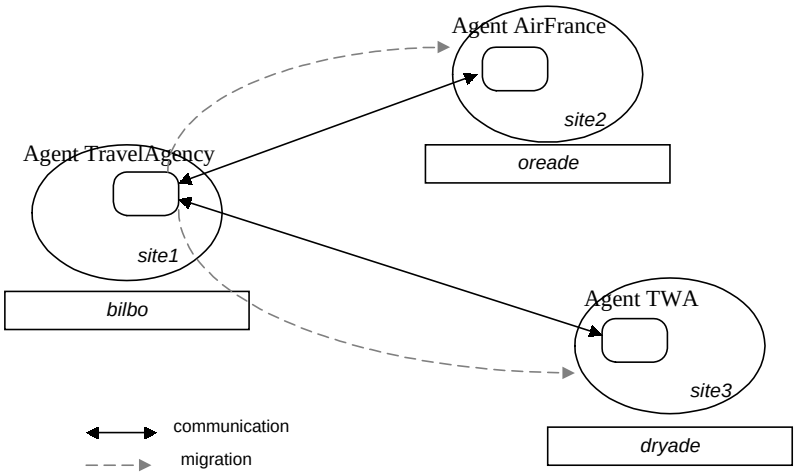
3.5 Exécution du modèle

Nous avons exécuté le même modèle d'agent obtenu sur deux différentes plates-formes [Yoo et al. 98a] [Yoo et al. 98b], sur JNA et Voyager.

3.5.1 Implantation sur JNA

La version 1.0 de JNA a été installée sur Sun SPARCs. Nous avons lancé trois sites différents sur les différentes machines et trois agents (TravelAgency, Air_France et TWA), un par site.

La communication des agents est exécutée suivant la figure 4 : l'agent 'TravelAgency' communique avec les autre agents contractants en utilisant le protocole d'appel d'offre pour trouver un vol de prix minimal. Après avoir choisi un contractant, l'agence de voyage migre sur le site du contractant choisi pour continuer la communication.



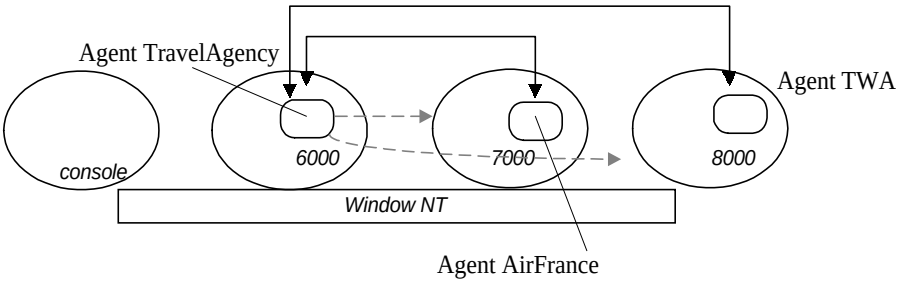
<Figure 4. Configuration sur JNA>

Dans la plate-forme JNA, il y a un environnement d'utilisateur pour exécuter et visualiser des comportement d'agents. Nous avons utilisé cet environnement pour visualiser tous les messages à tracer : les messages entre agents et ceux entre composants et vérifié que les agents communiquent suivant la définition du protocole d'appel d'offre.

3.6 Implantation sur Voyager

La plate-forme Voyager, version 1.0 (la version de lancement), a été installée sur PC sous Windows NT. Pour des raisons matérielles, l'expérimentation sur Voyager a été réalisée entre plusieurs sites exécutés sur une seule machine (Figure 5).

La plate-forme Voyager ne fournit pas un environnement utilisateur qui permet de visualiser le comportement des agents, ou des messages échangés. De ce fait, pour notre expérimentation, nous avons visualisé des messages qui sont échangés entre agents en imprimant la suite des messages sur l'écran.



<Figure 5. Configuration dans Voyager>

Dans la figure 5, il y a quatre environnements d'exécution sur la même machine pour tester le même exemple de protocole d'appel d'offre : un pour la console et trois pour lancer différents sites Voyager numérotés 6000, 7000, 8000. L'agent TravelAgency communique avec les agents serveurs par l'intermédiaire du protocole d'appel d'offre. Nous avons utilisé le même modèle de composants que nous avons modélisé et compilé pour l'expérimentation sur la plate-forme JNA. La différence avec JNA se réduit au package « softCompo.softAgent » et à la définition de « SoftAgent » (une sous classe de la classe 'Agent' de Voyager) qui encapsule le composant SCD pour le comportement d'agent.

Nous avons pu visualiser l'échange des messages et vérifié la migration des agents selon le site du contractant choisi. Le temps de migration a été moins long qu'en JNA.

Le même modèle conçu en SCD a été encapsulé dans différentes structures d'agents mobiles nommées « SoftAgent » puis exécuté. Nous avons obtenu le même résultat concernant des échanges de message entre agents. Ceci montre que notre approche facilite le portage du modèle d'agents mobiles sur certaines plates-formes.

3.7 Validation

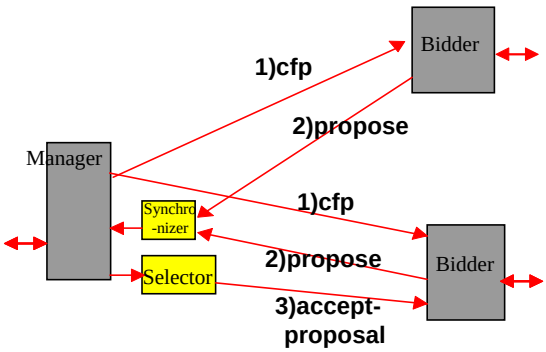
Nous avons appliqué l'approche de validation de réseaux de Petri à la validation d'un modèle de coopération d'agents. Au lieu de valider le modèle de SMA global (c'est-à-dire l'ensemble des comportements complets des agents), nous ne validons que le modèle de coopération d'agents.

La validation sur le modèle de coopération d'agents est effectuée sur le réseau de Petri global qui est obtenu par la composition des réseaux de Petri modulaires. D'abord, en sélectionnant des composants qui sont composés pour modéliser le protocole de coopération, nous pouvons obtenir le modèle de coopération des agents. Certaines règles de transformation sont décrites à la Table 7.

SCD	Réseau de Petri coloré
Un état (STATE) ou une variable (VAR)	Une place en terme de réseau de Petri coloré et des jetons qui y circulent avec leurs définitions de couleurs
Une opération (OPERATION)	Une transition en terme de réseau de Petri
Une entrée/sortie de message asynchrone	Une place
Une entrée/sortie de message synchrone	une transition qui simule l'exécution (le service) en liant une entrée de message asynchrone et une sortie de message asynchrone

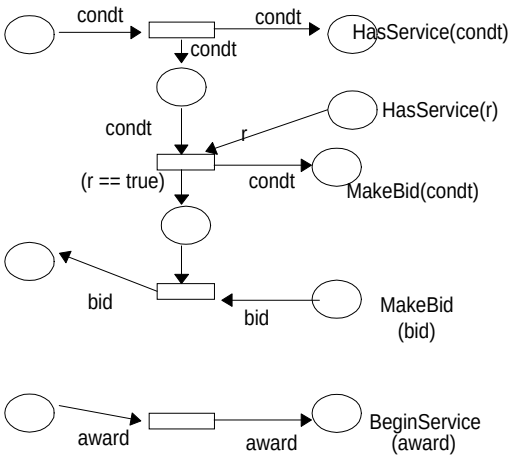
<Table 7. Quelques règles de transition de SCD en réseau de Petri coloré>

Pour ce qui est des règles de transformation complètes de SCD en réseaux de Petri colorés, consulter [Yoo 99]. Nous ne le détaillerons pas plus dans cette article.



<Figure 6. Configuration des composants dans le modèle de validation>

A partir de la configuration dans la figure 3, nous avons sélectionné des composants comme dans la figure 6. Chaque composant concernant le protocole d'appel d'offre (« Manager » et « Bidder » donc les descriptions sont données dans la table 1 (Contr_Manager) et dans la table 2 (Contr_Bidder)) peut être traduit en réseau de Petri modulaire. La figure 7 montre les réseaux de Petri modulaires obtenus à partir de la description pour le contractant du protocole de conversation en SCD dans la Table 2.



<Figure 7. Réseaux de Petri modulaires à partir de la description de Table 2>

La figure dans l'annexe montre le réseau de Petri global qui représente le modèle de coopération des agents de la figure 6. Il n'y a qu'un manager dans le protocole de Contract-Net. Mais pour la partie 'contractant' il y a deux agents contractants concernés. Au lieu de mettre deux réseaux modulaires pour représenter les contractants, nous avons composé un seul réseau modulaire pour représenter les contractants en ajoutant une place « servers » qui accepte des jetons de type « AgentAddress ». Au moment de l'initialisation du réseau, on a mis les deux jetons initiaux qui représentent les contractants. Pour valider le protocole avec dix contractants, on met dix jetons initiaux dans la place « servers ».

Nous avons validé ce réseau en utilisant l'outil CPN/AMI pour des propriétés suivantes :

- Invariants des places,
- Simulation pas-à-pas,
- Génération de graphe d'occurrence sur l'espace des états accessibles.

Nous avons vérifié que le protocole d'appel d'offre conçu a terminé dans l'état final acceptable sans blocage. Voir le paragraphe 4.2 pour une discussion plus précise sur les résultats obtenus.

4. DISCUSSION ET CONCLUSION

Nous avons réalisé un environnement de développement d'agents qui concerne trois étapes : la modélisation, la validation et l'implémentation. Nous avons choisi une approche componentielle pour la facilité de modélisation et de modification. Le langage SCD permet de modéliser le comportement des agents par des composants dans une architecture d'agent et ensuite d'être traduit en un modèle de validation basé sur des réseaux de Petri.

Nous avons choisi le scénario d'agence de voyage (proposé par FIPA) pour une première application de notre approche. Nous avons modélisé en SCD des agents qui coopèrent suivant les protocoles de coopération. Des agents dans le scénario simplifié ont été exécutés dans deux différentes plates-formes d'agents mobiles. Nous avons pu visualiser des échanges de messages entre agents suivant la définition de chaque protocole de coopération.

Nous discutons les résultats obtenus plus spécialement sur i) le passage de la modélisation vers l'implémentation, ii) l'aspect validation, dans les paragraphes suivants.

4.1 Efficacité de la réalisation d'agents mobiles

Notre approche concernant la génération de code et l'implémentation peut résoudre certaines difficultés de la programmation des agents mobiles dans une plate-forme basée sur Java. Ce sont :

1) la gestion de la migration d'agent avec de multiple threads,
2) le problème de reprise d'une activité après la migration,
3) la portabilité du modèle d'agents sur différents types de plates-formes d'agents mobiles.

Quand on implémente des agents mobiles dans une plate-forme basée sur Java, il y a des problèmes de gestion de la migration d'agents avec de multiple threads. Notre environnement de développement fournit des mécanismes de base qui traitent la migration de multiples threads dans le modèle d'un agent.

Le deuxième point nous paraît relativement important concernant le développement d'agents mobiles et la technologie courante. La plupart des plates-formes d'agents mobiles garantit la transmission de données et de procédures (JNA, Aglet). Mais très peu de plates-formes (hormis essentiellement Telescript) garantissent la transmission du vecteur d'état c'est-à-dire que l'on transmet le vecteur d'état décrivant l'état précis de l'agent juste avant la migration. Ce niveau apporte une grande flexibilité : après avoir migré, l'agent reprend l'exécution de son comportement à l'endroit précis où il s'était arrêté. Avec notre approche, le concepteur peut obtenir cette facilité dans une plate-forme qui ne l'implémente pas. Par exemple, si le concepteur modélise le comportement d'un agent en SCD :

```
op1 {STATUS  {état1 == condition1}          // une opération dans le comportement d'un agent : test de condition
        THEN MOVETO (adresse_site1)         // si la condition est satisfaite, alors migrer au site 1
                NEXTSTATE état2             // l'état après la migration est 'état2'
        END                                 // après la migration, l'agent récupère son état comme 'état2'

op2 {STATUS  état2                          // si l'agent est dans l'état 'état2',
        THEN SEND inform                    // alors il fait l'action précisée : envoyer un message 'inform'
        .....
```

Dans ce cas, l'agent peut garder son état avant la migration comme l'état 1. Après la migration (MOVETO) il récupère son état courant comme état 2. Comme l'agent est dans l'état2 juste après la migration, il continue son action d'envoi de message « SEND inform ».

Finalement pour le troisième point, en dissociant clairement le modèle d'agent de haut niveau qui est donné par la description en SCD et la structure physique d'agents mobiles qui est fournie par une plate-forme d'agent mobile, nous avons simplifié le problème de portabilité du modèle d'agents sur deux différents types de plates-formes.

4.2 Difficultés et limites dans la validation

Notre expérimentation montre que le mécanisme de fonctionnement des réseaux de Petri s'adapte bien à représenter le mécanisme de protocole de coopération, c'est-à-dire " si l'agent est dans un *état_x*, et s'il reçoit un *message_i*, alors il envoie un *message_j* et il passe dans un *état_y* ". Nous avons pu transformer la *structure* qui représente la transition des états du protocole de coopération d'agents en un modèle de validation (en réseaux de Petri) sans difficulté. Par contre, la transformation des variables dans le modèle de conception nous a posé quelques problèmes. Du fait des contraintes posées sur le formalisme, la structure des variables est fixée par la définition des couleurs de jetons, d'où la difficulté de mise à jour dynamique des contenus du message.

Le " model checking " est estimé comme un bon moyen de validation/vérification d'un système du fait de la facilité d'automatisation. Mais comme la validation/vérification est effectuée sur l'espace des états accessibles, si la spécification de comportement du système est très complexe, le nombre d'états est très élevé au point qu'une exploration exhaustive n'est pas possible [Etique et al. 95], [Lawry et al. 97], etc. Pour valider un modèle de taille importante, il faut réduire la taille de l'espace par une abstraction du modèle [Lowry et al. 97], soit à la main soit de manière automatique. Dans notre cas, cette abstraction a été faite à la main.

Pour ce qui est des propriétés à valider, nous avons considéré :

1) (propriété générale) Est-ce que le protocole d'appel d'offre se termine dans un état final sans échec ?
2) Est-ce que tous les serveurs de voyage reçoivent l'appel d'offre de l'agence de voyage ?
3) Est-ce qu'il est possible qu'un serveur de voyage n'ait jamais reçu le message 'accept-proposal' depuis le manager (l'agence de voyage dans notre exemple) ?

La propriété 1) est une propriété générale et il suffit de vérifier l'accessibilité de l'état final sur le réseau de Petri obtenu. La propriété 2) est un énoncé spécifique à cette application. Dans notre réseau de Petri global obtenu (modèle de validation), les connexions entre l'agence de voyage et les serveurs de voyages sont toujours établies sans défaillance, ce qui n'est pas garanti dans le cas d'une exécution réelle. Donc par la validation de réseaux de Petri, ce genre de propriété est validante sous l'hypothèse très forte qu'aucune défaillance n'intervient sur le réseau.

La propriété 3) n'a pu être validée en utilisant *notre réseau de Petri* obtenu. Pour valider ce genre de propriété, il faut d'abord exprimer que notre modèle de validation concerne uniquement les interactions entre les agents. Au cours de la transformation, les autres parties du modèle étaient simplifiées. Si on veut vérifier une telle propriété, il faut une transformation en formalisme de réseaux de Petri de la partie du modèle qui traite la sélection des agents contractants pour diffuser le message 'cfp'. Mais d'après notre expérimentation sur la traduction du modèle de coopération, il ne sera pas facile de représenter un algorithme aussi complexe en réseaux de Petri à l'aide de jetons et leur couleurs.

Le résultat de la validation concernant les propriétés générales est intéressant, par exemple pour trouver un état de blocage, une mauvaise valeur dans un champ de variable, etc. Mais en ce qui concerne les propriétés spécifiques au système (SMA), nous n'avons pas pu obtenir de résultat satisfaisant.

4.3 Conclusion générale

Le résultat de ce travail peut être appliqué dans différentes directions. Nous pensons que notre environnement de modélisation d'agents basé sur des composants peut être utilisé dans un contexte différent, par exemple une plate-forme de développement d'agents basée sur Java sans mobilité d'agents.

Notre expérimentation exploite également la réutilisabilité des composants SCD. Un des avantages de l'approche componentielle est la possibilité de réutiliser des composants existants pour la conception d'un autre modèle. Nous avons étendu le modèle de protocole d'appel d'offre, soit pour compléter son comportement, soit pour avoir un protocole plus complexe, en réutilisant des composants utilisés dans le premier modèle conçu en SCD. Pour plus de détails concernant la réutilisabilité des composants, voir [Yoo 99].

Enfin, nous avons fourni des techniques de bases pour l'implémentation d'un système de commerce électronique par agents. Pour que ces premiers résultats soient applicables à des applications réelles, nous souhaitons intégrer une API de traitement d'ACL (KQML ou FIPA) dans notre architecture et élargir le fonctionnement interne d'un agent serveur par un « legacy system » de type JDBC encapsulé dans un composant SCD.

Remerciements *Ce travail a été financé par France Télécom/CNET (CTI No. 95 1B 176).*

Références

[Agha Sami 92] Gul Agha, Miriyala Sami**.. *"Visualizing Actor Program using Predicate Transition Nets*.**"** *Journal of Visual languages and Computation*, 3 (2) June, 1992.

[Aglet 96] available on http://www.trl.ibm.co.jp/aglets/.

[Barbuceanu Fox 95] Mihai Barbuceanu, Mark Fox. "***Cool: A Language for Describing Coordination in Multi Agent Systems***." *ICMAS 95*, 1995.

[Bellisard et al. 95] L. Bellissard, S. Ben Atallah, A. Kerbat, M. Riveil. "***Component-based Programming and Application Management with OLAN***." *Workshop on Object-Based Parallel and Distributed Computation*, Tokyo, 1995.

[Bradshow 97] Jeffrey Bradshaw, ed. ***"Software Agents",*** AAAI Press/MIT Press, 1997.

[Burkhard 93] Hans-Dieter Burkhard. "***Liveness and Fairness Properties in Multi-Agent Systems***." *IJCAI 93, International Joint Conference on Artificial Intelligence*, 1993.

[Burmeister et al. 93] Birgit Burmeister, Afsaneh Haddadi, Kurt Sundermeyer, ***"Generic, Configurable, Cooperation Protocols for Multi-Agent Systems"***, LNAI 957, 5th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW '93.

[Chauhan 98] Deepika Chauhan ***"JAFMAS: A Java-based agent framework for multiagent systems development and implementation",*** Master's thesis, University of Cincinnati, 1998.

[Coen 94] Michael H. Coen**. *"SodaBot: A Software Agent Environment and Construction System"***, A.I.Technical Report Messachusettes Institue of Technology, 1994.

[Cost et al. 99]R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, Yun Peng, ***"Modeling Agent Conversations with Colored Petri Nets"*** , *Workshop on specificating and Implementing Conversation Policies, Autonomous Agents '99*, Seattle, USA, 1999.

[Demazeau 95] Yves Demazeau, ***"From interactions to collective behavior in agent-based systems"***, In Proceedings of the 1st European Conference on Cognitive Science, St.Malo, France, March 1995.

[Demazeau Müller 91] Yves Demazeau and Jean-Pierre Müller, editors, ***"Decentralized A.I. II"***, Elsevier Science Publishers B.V., Amsterdam, NL, 1991.

[ElFallah Haddad 94] Amal El Fallah-Seghrouchni, Serge Haddad. "***Représentation et Manipulation de Plans à l'aide de réseaux de Petri.***" *2ème Joutnées Francophones IAD&SMA*, 1994.

[Etique et al. 95] Pierre-Alain Etique, Jean-Pierre Hubaux, Tuncay Saydam**, *"Verification et validation de services de télécommunications spécifiés par une méthode orientée objets"***, CFIP 95, 1995.

[Ferber 96] Jacques Ferbers, ***"Les Systèmes Multi-Agents"***, InterEditions, 1996.

[Finin et al. 97] T. Finin, Y.Labrou, J.Mayfield, ***"KQML as an Agent Communication Language"***, in [Bradshow 97].

[FIPA 97] Foundation for Intelligent Physical Agents, Specification Version 2, Part1-Part5, 1997, http://drogo.cselt.stet.it/fipa/spec.

[Gauvin et al. 97] Daniel Gauvin, Gervé Marchal, Carlos Saldanha, ***"LALO: un environnement de programmation ouvert pour des systèmes multi-agents"***, acte des 5éme jounées francophones JFIADSMA '97, Hermes 1997.

[Genesereth 97] Michael R. Genesereth, ***"An Agent-Based Framework for Interoperability"***, in [Bradshow 97].

[Kendall et al. 98] Elizabeth A. Kendal, P. V. Murali Krishna, Chirag V. Pathak, C. B. Suresh. "***A Java Application Framework for Agent Based Systems***." *Autonomous Agents '98*, 1998.

[Linden et al. 96] Greg Linden, Steve Hanks, Neal Lesh, ***"Interactive Assessment of User Preference Models: The Automated Travel Assistant"***, 1996.

[Lowry et al. 97] Michael Lowry, Klaus Havelund, John Penix, ***"Verification and Validation of AI Systems that Control Deep-Space Spacecraft"***, Foundations of Intelligent Systems : 10th International Symposium, ISMIS 97, Charlotte, North Carolina, USA, Octobre 1997.

[Merlat 98] Walter Merlat, Claude Seyrat, Jacques Ferber, ***"Mobile-Agents for Dynamic Organizations: The Conversational-Agent Paradigm"***, 8th European Workshop on Modeling Autonomous Agents in a Multi-Agent World (MAAMAW), Ronnevy Sweden, LNAI 1237, Springer-Verlag 1997.

[Moldt Wienberg 97] Daniel Moldt, Frank Wienberg. "***Multi-Agent-Systems based on Colored Petri Nets***." *18th International Conference on Application and Theory of Petri Nets*, Toulouse, France, 1997.

[Nierstrasz Meijler 94] Oscar Nierstrasz, Theo Dirts Meijler, "***Requirements for a Composition Language***", ECOOP 94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Italy, LNCS 924, Springer-Verlag, 1994.

[Petrie 96] Charles J. Petrie. "***Agent-Based engineering, the Web, and Intelligence***." *IEEE Expert*, Decembre, 24-29, 1996.

[Sami Vidal-Naquet 91] Yasmina Sami, Guy Vidal-Naquet**. *"Formalisation of the Behavior of Actors by Coloured Petri Nets and some Applications***." *PARLE 91 Conference on Parallel Architectures and Languages*, 1991.

[Seyrat 96] Claude Seyrat, ***"JavaNetAgents : Un environnement d'exécution d'agents mobiles pour le développement de systèmes multi-agents coopératifs sur Internet"***, Rapport de stage de DEA/IARFA, 1996.

[Smith 80] Reid G. Smith, ***"The Contract Net Protocol: A High Level Negociation Protocol for Distributed Problem Solving"***, IEEE Transactions on Computer (29) 1980.

[Telescript 95] General Magic. ***"Telescript: An Overview"***. Available on http://www.genmagic.com.

[Vernadat et al. 95] F. Vernadat, A. Lanusse, P. Asema, "***Modélisation par réseau de Petri d'un langage acteur : Application à la vérification de système multi-agents****."*, Deuxième Journées Francophones IAD&SMA - 1994.

[Voyager 97] ObjectSpace Voyager, available on http ://www.objectspace.com/.

[VonMartial 92] F. von Martial, ***" Coordinating plans of autonomous agents"***, LNCS 610, Springer-Verlag, 1992.

[Wooldridge Jennings 98] Michael Wooldridge, Nicolas R. Jennings. "***Pitfalls of Agent-Oriented Development***." *Autonomous Agents '98*, 1998.

[Yoo 99] Min-Jung Yoo, ***" Une approche componentielle pour la modélisation d'agents coopératifs et leur validation",*** Doctoral thesis, Université de Paris 6, October 1999, LIP6 Technical report No. 2000/020.

[Yoo et al. 98a] Min-Jung Yoo, Walter Merlat, Jean-Pierre Briot, ***"Modeling and Validation of Mobile Agents on the Web"***, in the Proceedings of the International Conference on the Web-based Modeling & Simulation, San Diego, SCS Simulation Series, Vol. 30, N° 1, January 1998, pages 23-28.

[Yoo et al. 98b] Min-Jung Yoo, Jean-Pierre Briot, Jacques Ferber, ***"Using components for modeling intelligent and collaborative mobile agents"***, *WetIce 98 IEEE International Workshops on Enabling Technology in Collaborative Enterprises*, 1998.

Annexe

CLASS
Condition is [c1, c2, c3];
AgentAddress is [af, twa];
Proposition is [prop0, prop1, prop2, prop3, prop4, prop5];

DOMAIN
Bid is <AgentAddress, Proposition>;
BidList is <AgentAddress, AgentAddress>;
Award is <AgentAddress>;
ACondition is <AgentAddress, Condition>;

VAR
condt in Condition;
a, aa, ab, ac in AgentAddress;
pr, pra, prb, prc in Proposition;