Actes de la 2eme Journee Multi-Agent et Composant

Manifestation connexe a LMO 2006

Ecole des Mines d'Ales

Nîmes, 21 Mars 2006

http://csl.ensm-douai.fr/MAAC/JMAC2006

Comité de programme

- Noury Bouraqadi, EM Douai (Président)

- Philippe Aniorté, LIUPPA, Pau
- Olivier Boissier, EM St Etienne
- Jean-Pierre Briot, LIP6, Paris
- Christophe Dony, LIRMM, Montpellier
- Jacques Ferber, LIRMM, Montpellier
- Marie-Pierre Gleizes, IRIT, Toulouse
- Marc-Philippe Huget, LISTIC, Annecy
- Philippe Mathieu, LIFL, Lille
- Philippe Merle, INRIA, Lille
- Amedeo Napoli, Loria, Nancy
- Michel Occello, LCIS - INPG, Valence
- Jean-Paul Sansonnet, LIMSI-CNRS, Orsay
- Djamel Seriai, EM Douai
- Serge Stinckwich, GREYC, Caen
- Christelle Urtado, EM d'Alès
- Sylvain Vauttier, EM d'Alès
- Laurent Vercouter, EM St Etienne

Organisateurs

- Olivier Boissier (Ecole des Mines de Saint Etienne)
- Noury Bouraqadi (Ecole des Mines de Douai)
- Jean-Claude Royer (Ecole des Mines de Nantes)
- Djamel Seriai (Ecole des Mines de Douai)
- Christelle Urtado (Ecole des Mines d'Alès)
- Sylvain Vauttier (Ecole des Mines d'Alès)
- Laurent Vercouter (Ecole des Mines de Saint Etienne)

Table des matières

Programme

- 9h30-10h00 : *Accueil*
- 10h00-11h00 : **Exposé invité : Jacques Ferber, LIRMM**
- 11h00-11h30 : *Pause café*
- 11h30-11h50 : Présentation de l'équipe AOC (Agent-Objet-Composant), LIUPPA
  - Nabil Hameurlain
- 11h55-12h15 : **Papier** "vers l'adaptation dynamique de services : des composants monitorés par des agents"
  - Jérôme Lacouture — Philippe Aniorté
- 12h20-12h40 : **Papier** "Modélisation de systèmes complexes distribués : l'ingénierie des modèles pour l'intégration des paradigmes agent et composant"
  - Philippe Aniorté — Eric Gouardères — Eric Cariou
- 12h45-13h05 : Présentation de l'équipe SMAC, LIFL
  - Philippe Mathieu
- 13h05-14h25 : *Déjeuner*
- 14h30-14h50 : **Papier** "Lisaac agent Petit modèle agent pour langage à prototype"
  - Pierre-Alexandre Voye — Benoît Sonntag
- 14h55-15h15 : Présentation de l'équipe SMA, Centre G2I, ENSM Saint-Etienne
  - Olivier Boissier
- 15h20-15h40 : Présentation de l'équipe informatique de l'Ecole des Mines de Douai
  - Noury Bouraqadi
- 15h45-16h05 : **Papier** "Assemblage Automatique de Composants pour la Construction d'Agents avec MaDcAr"
  - Guillaume Grondin — Noury Bouraqadi — Laurent Vercouter
- 16h10-16h40 : *Pause café*
- 16h40-17h00 : Présentation de l'équipe LYRE, IRIT
  - Jean-Paul Arcangeli
- 17h05 - 17h25: **Papier** "Vers un modèle d'agent flexible"
  - Sébastien Leriche — Jean-Paul Arcangeli

Vers l'adaptation dynamique de services : des composants monitorés par des agents

Jérôme Lacouture – Philippe Aniorté

Laboratoire Informatique de L'Université de Pau et des Pays de l'Adour (LIUPPA)
IUT de Bayonne - Département Informatique
Place Paul Bert - 64100 Bayonne
{lacoutur,aniorte}iutbayonne.univ-pau.fr

RÉSUMÉ. *La grille informatique, contexte de notre travail, laisse transparaître une problématique inhérente aux systèmes distribués actuels et futur, à savoir la nécessité d'adapter et d'intégrer dynamiquement des services. Nous nous appuyons sur de précédents travaux menés au sein du laboratoire autour du modèle de composant Ugatze conçu pour l'intégration de composants assemblés de manière statique. Dans cet article, nous étendons ce modèle pour répondre au problème de l'adaptation dynamique de services. Pour cela nous proposons de nouveaux points de spécifications pour l'interface des composants et nous introduisons une solution basée sur les paradigmes composants et agents pour la mise en œuvre de ce modèle étendu.*

ABSTRACT. *Grid computing, context of our research activities, underlines a central concern of current and future distributed systems, and more precisely, the need to dynamically adapt and integrate services. We base our works on a component model, named Ugatze, developed in our laboratory, which is approved for a static component integration. In this paper, we extend the model to deal with the dynamic adaptation of services. So, we propose new specification points for component interfaces and we introduce a solution based on components and agents paradigms to execute this extended model.*

MOTS-CLÉS : *Composants Logiciels, Agents, Adaptation Dynamique, Grille.*

KEYWORDS: *Software Components, Agents, Dynamic Adaptation, Grid.*

1. Introduction

Initialement conçue pour une exploitation des ressources la plus efficace possible, axée vers le « supercomputing », la grille est le support où des organisations virtuelles évoluent dans le but de partager des services, appelés services de grille. Les organisations virtuelles sont des groupes virtuels émergeant à travers des échanges ou conversations mettant en jeu un certain niveau de satisfaction entre ses participants (Rheingold, 1993). Plus précisément, les organisations virtuelles sont des groupes d'individus qui sont formés par affinités, but communs ou activités communes.

L'intérêt porté à la grille a conduit à l'émergence de nouveaux standards tant au niveau architectural, avec OGSA (Foster *et al.*, 2002), qu'au niveau de la spécification des services de grille, avec OGSI (Tuecke *et al.*, 2003) puis WSRF (Czajkowski *et al.*, 2004). Ces efforts visent la production d'une architecture de base de haut niveau pour assurer une certaine qualité de services.

Pour résumer, on peut dire qu'une grille coordonne un ensemble de ressources au travers d'organisations virtuelles avec un contrôle totalement décentralisé, utilise des standards et des protocoles ouverts et assure une certaine qualité de services à ses usagers.

La grille donne la voie de l'autonomie, de l'adaptation et de la coordination de ressources. Pour nous, elle illustre une des caractéristiques inhérentes aux systèmes distribués actuels et futurs, c'est à dire, la capacité d'adapter, d'intégrer et de coordonner « à la volée » les services disponibles. En effet, la composition dynamique de services et leur adaptation deviennent un souci central. Pour cela, la prise en compte du contexte, de l'existant, et par conséquent la problématique de la réutilisation, demeurent, bien évidemment, un aspect non négligeables.

Or, l'idée de recourir au paradigme composant se justifie par les résultats obtenus, d'une part dans le domaine de la réutilisation, et d'autre part en matière de modélisation et de déploiement d'application distribuées.

Par conséquent, cet article a pour objet de proposer une voie à l'adaptation de la dynamicité grâce à la production de composants de plus en plus autonomes. L'article s'organise comme suit. En section 2, nous repositionnons l'intérêt du paradigme composant dans le domaine de la réutilisation. Puis en section 3, nous revenons sur le modèle Ugatze qui constitue une solution aboutie pour une intégration statique. Nous proposons, section 4, une extension de ce modèle vers l'adaptation de composants logiciels en intégrant le principe de variabilité, puis nous conférons à cette phase d'adaptation un caractère dynamique par une approche mêlant agents et composants.

2. Le Paradigme Composant

On assiste à l'émergence de besoins en ingénierie de systèmes qui s'expriment essentiellement en terme d'évolution. Il s'agit de faire évoluer les systèmes existants vers de nouveaux systèmes caractérisés par l'hétérogénéité des éléments hérités, et leur distribution via un réseau au travers duquel ils pourront interagir. Cette évolution se décline à la fois en termes de « réutilisation » et « d'interopérabilité ».

La réutilisation, qui s'oppose aux approches usuelles dans lesquelles le développement d'un nouveau système part de rien (« from scratch ») et nécessite de tout réinventer à chaque fois, est l'objet de beaucoup d'intérêt de la part des organisations. En effet, présentée comme une nouvelle approche d'ingénierie de systèmes selon laquelle il est possible de construire un système à partir d'éléments existants, elle constitue un élément de réponse fondamental aux besoins exprimés par ces organisations. Ces dernières y voient un moyen de faire l'économie de développements longs et coûteux, grâce à la réutilisation des éléments hérités.

L'interopérabilité peut se définir comme la faculté de permettre l'interaction entre éléments hétérogènes pouvant revêtir différentes formes. S'opposant à une vision monolithique des systèmes (McCarthy *et al.*, 1989), elle apparaît, auprès des organisations, comme un autre élément essentiel à la réussite de l'évolution, constituant le support aux interactions entre éléments hétérogènes hérités.

Dans la littérature, le terme « composant » est largement associé à la réutilisation. Les composants réutilisables sont représentés à l'aide d'un modèle de composants. Deux principes essentiels doivent idéalement sous-tendre ces modèles : le principe d'abstraction (cf Figure 1). et le principe de variabilité (cf Figure 2).

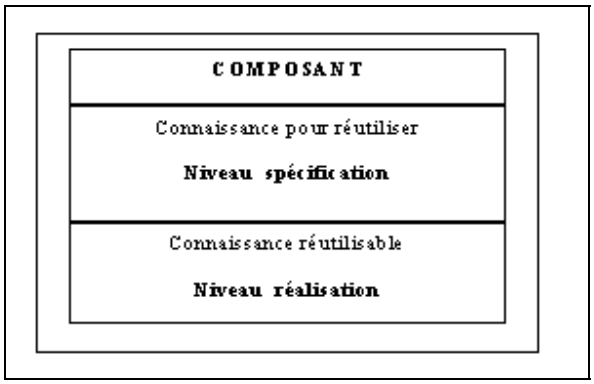


Figure 1. *Représentation du principe d'abstraction (Cauvet et al., 1999)*

L'abstraction est un mécanisme qui est depuis longtemps reconnu en matière de réutilisation (Parnas, 1976) (Neighbors, 1984) (Krueger, 1992). Il consiste à distinguer explicitement dans un composant la connaissance effectivement

réutilisable (la réalisation du composant) et la connaissance utile à sa réalisation (la spécification ou le descripteur du composant). La réalisation est la partie « cachée ». La spécification est la partie « visible » du composant.

Ce principe est nécessaire pour construire des infrastructures de réutilisation efficaces. En effet, la partie spécification d'un composant peut jouer un rôle essentiel dans l'exploitation de l'architecture de réutilisation, en particulier lors de la recherche et de l'intégration de composants.

Le principe de variabilité (cf figure 2) conduit à distinguer dans la spécification d'un composant une partie fixe et une partie variable. Ces deux parties sont visibles dans la spécification du composant ; elles introduisent le caractère générique du composant. Au moment de la réutilisation, c'est en fixant la partie variable que l'on choisit une réalisation particulière et que l'on adapte le composant en fonction des spécificités du système en cours de développement.

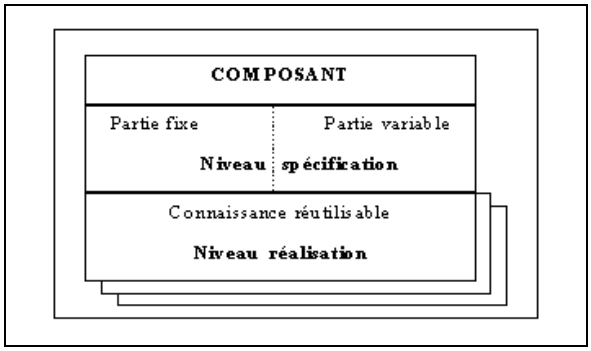


Figure 2. *Représentation du principe de variabilité (Cauvet et al.*, 1999*)*

3. Le Modèle Ugatze

Le contexte des travaux passés nous à conduit à manipuler des composants définis comme abstractions d'entités logicielles héritées de systèmes existants (Aniorté, 2004). Concrètement, il peut s'agir de programmes, voire d'applications, entités dont la « granularité » est importante. Dans le cas présent, les composants sont des abstractions des services de grille. Nous qualifions ces composants de « haut niveau d'abstraction » caractérisés par la propriété d'*autonomie*, autonomie de conception, et autonomie à l'exécution. L'autonomie implique que le couplage entre composants soit nécessairement lâche, à tel point que le principe régissant l'interaction entre composants est appelé « principe de découplage ».

Le (méta)modèle de composants dénommé « Ugatze » (Seyler, 2004), adapté à la réutilisation de composants logiciels autonomes, hétérogènes et distribués,

s'inscrit dans le domaine de l'Ingénierie Dirigée par les Modèles (IDM) (Aniorté, 2004). Ce métamodèle repose sur deux notions essentielles :

– l'interface du composant. C'est le résultat de la (re)spécification du composant, activité propre à la réutilisation (Cauvet *et al.*, 1999). La représentation de tous les composants à réutiliser est « unifiée » (au sens des besoins exprimés pour l'interopérabilité) via le métamodèle.

– l'interaction entre composants. Elle permet de gérer l'interopérabilité entre composants.

3.1. L'interface du composant

La spécification des composants réutilisables telle que nous la pratiquons repose sur le concept « d'interface » constituée de « points d'interaction ». Ils sont dédiés à l'intégration du composant dans le système à développer, rendant effective sa réutilisation.

Nous distinguons différents types de points d'interaction. L'autonomie des composants et le principe de découplage associé, reposant entre autres sur la séparation du flot d'information et du flot de contrôle (Lhuillier, 1998), nous conduisent à recenser :

– des points d'interaction de donnée,

– des points d'interaction de contrôle.

Considérant l'aspect flux, nous distinguons :

– des points d'interaction d'entrée, flux entrant dans le composant,

– des points d'interaction de sortie, flux sortant du composant.

En croisant les deux critères, donnée / contrôle et entrée / sortie, nous obtenons :

– des points d'interaction d'entrée de donnée ou Data Input (DI),

– des points d'interaction de sortie de donnée ou Data Output (DO),

– des points d'interaction d'entrée de contrôle ou Control Input (CI),

– des points d'interaction de sortie de contrôle ou Control Output (CO).

3.2. L'interaction entre composants

Le métamodèle proposé est basé sur le principe de découplage entre composants. L'interaction s'appuie sur ce principe. Elle permet de gérer l'interopérabilité conceptuelle entre composants.

Les composants réutilisables, modélisés à l'aide des points d'interaction, sont destinés à être intégrés afin de développer le nouveau système. Au sein de ce système, ils doivent pouvoir interopérer. L'interopérabilité est gérée à l'aide du

concept d'interaction. La mise en œuvre des interactions est basée sur les points d'interaction présentés précédemment. Nous distinguons :

– interactions « prédéfinies ». Elles constituent pour le concepteur « l'offre de base » en matière d'interactions. Cette offre se caractérise par des dispositifs généraux, dont le nombre est limité, utilisés de façon récurrente. A l'instar des points d'interaction, les interactions prédéfinies se déclinent en deux catégories : interactions d'information et interactions de contrôle.

– interactions « à façon ». Elles constituent pour le concepteur une alternative à l'offre de base. L'idée est de donner au concepteur la possibilité de définir une interaction idoine, comme solution à un problème spécifique. Cette possibilité contribue grandement à la gestion de composants *autonomes*, en autorisant un haut niveau de découplage, ce qui dans le contexte de nos travaux est fondamental. En conséquence, la prise en compte de l'hétérogénéité s'en trouve facilitée. In fine, la gestion de l'intéropérabilité est favorisée.

Outre les notions clés déjà présentées (interface et interaction), le métamodèle « Ugatze » se caractérise par une représentation graphique et des règles de bonnes utilisations. Nous renvoyons le lecteur intéressé à (Seyler, 2004).

4. Une Extension du Modèle Ugatze

4.1. De nouveaux points de spécification : les points d'ajustement

Le modèle Ugatze offre une solution au problème de la réutilisation de composants dont l'intégration est statique, c'est à dire fixée au niveau conceptuel, mais il ne permet pas en l'état de gérer l'intégration dynamique de composants, c'est à dire décidée lors de l'exécution. Il devient alors difficile de gérer les changements imprévus une fois le système déployé (apparition de nouveaux services, ancien service remplacé par un nouveaux …). Les composants devraient pouvoir s'adapter automatiquement aux changements du système.

Dans Ugatze, l'interface du composant est décrite à l'aide de points de spécifications baptisés point d'interaction qui répondent au soucis de réutilisation. Nous proposons d'ajouter une autre catégorie de points de spécification baptisés «points d'ajustements » afin d'avoir un modèle de composant gérant l'adaptation, en plus de l'intégration.

Ces nouveaux points de spécification reposent sur le principe de variabilité (cf. section 2). La prise en compte de ce principe nous amène à considérer la réalisation comme un noyau (partie fixe), auquel sont associés des services requis et fournis (partie variable). Un point d'ajustement sera ainsi associé à chaque service, permettant d'adapter le composant à des besoins spécifiques. La représentation graphique d'un composant est synthétisé par la figure 3.

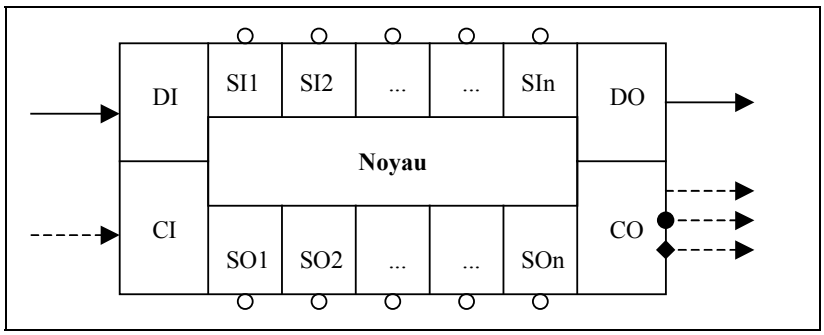


Figure 3. *Représentation graphique d'un composant d'origine*

Les parties latérales représentent les quatre types de points d'interaction (DI, CI, DO, CO) précédemment cités.

La partie centrale du schéma représente la réalisation du composant vue comme un noyau (partie fixe) et un ensemble de services fournis SO1 … SOn ou requis SI1 … SIn (partie variable). A chaque service est associé un point d'ajustement qui permet de sélectionner (ou non) le service. Pour éviter toutes confusions dans la suite de l'article avec les services de grille que nous voulons abstraire, nous parlerons d'interfaces fournies et d'interfaces requises à la place de services (SO et SI). A chacune de ses interfaces est associées une description sous fore de méta-informations mettant en exergue les propriétés fonctionnelles (relatives au tu type de service fourni) et non-fonctionnelles (QoS, transactions, performance …).

L'adaptation du composant se fait donc par sélection d'un certain nombre d'interfaces du composant. A chaque composant d'origine est ainsi associé une « image » par adaptation du composant. Chacune de ces images correspond à un cas de réutilisation. A chaque composant d'origine peuvent donc être associés plusieurs cas de réutilisation relatif aux besoins contextuels au système. Nous allons voir dans la prochaine section comment cette adaptation se traduit dans un exemple de service de grille.

4.2. Exemple d'adaptation d'un service de grille

Notre approche reposant, comme nous l'avons vu, sur le principe d'abstraction, nous abstrayons un service de grille et nous nous préoccupons des interfaces qu'il fournit et qu'il requiert de manière à décrire un composant d'origine, base de l'adaptation.

Pour illustrer notre approche, nous décrivons un service de grille baptisé e-Portfolio (Razmerita, 2005) (cf figure 4). Ce service a pour objet d'être le support d'un apprentissage informel. Construit à partir d'une ontologie commune, garant d'une compréhension entre apprenants, il est plus qu'un simple curriculum vitae

retraçant l'activité d'un étudiant. Il est aussi le support d'échange et de partage d'informations. C'est un espace de travail collaboratif mis à disposition d'une communauté d'apprenant. Dans la figure 4, le e-Portfolio devient actif après acquisition des concepts présent dans l'ontologie (informations sur l'apprenant), attributions d'instruction et d'objectifs à atteindre. Le e-Portfolio stocke aussi les résultats et la classification de l'apprenant en niveaux. Il délivre en contrepartie, une expertise, un savoir relatif aux connaissances acquises (ici, connaissance sur des activités de décollage, d'atterrissage, de maintient de cap, de gestion d'instructions).

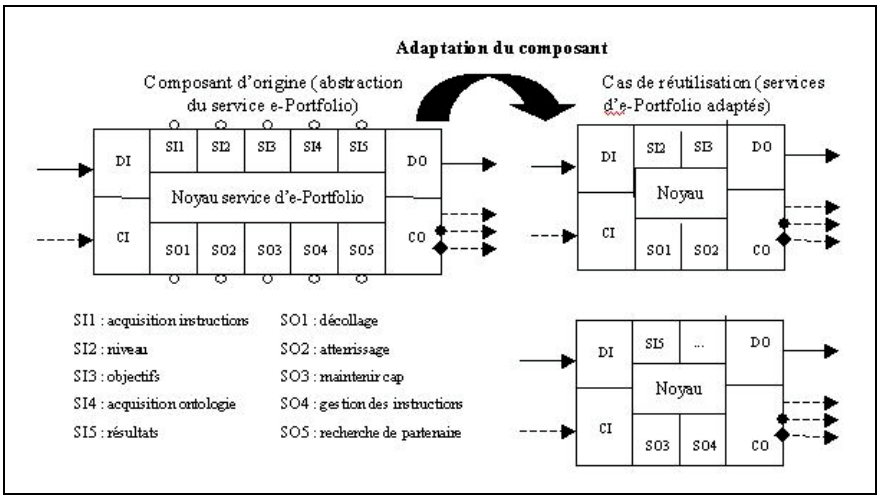


Figure 4. Adaptation d'un service de grille (e-Portfolio). Différents cas d'utilisation sont dérivés du composant d'origine.

Après avoir décrit un composant source encapsulant les interfaces fournies et requises de l'e-Portfolio, nous souhaitons adapter ce composant pour différentes activités. Une sélection des points d'ajustement est alors effectuée permettant de décliner des cas de réutilisation. A noter que cette sélection sera basée sur des métadonnées associées à chaque point d'ajustement. Ainsi une description (sous forme de méta-informations) à la fois décrivant l'aspect fonctionnel et non-fonctionnel des interfaces pourra guider la sélection. Dans l'exemple illustré, un premier service de grille (e-Portfolio) est destiné à l'apprentissage de phases précises que sont décollage et atterrissage. Un deuxième service sera lui destiné à faire un bilan des résultats et établir de nouvelles instructions personnelles.

Les points d'ajustement sont donc les outils de l'adaptation des services de grille, dits d'origine. La gestion de ces points, notamment leur sélection pour participer à un cas de réutilisation reste à être considérée. L'intégrer au niveau conceptuel nous enfermerait dans une problématique d'adaptation statique de service. En effet, beaucoup d'approches essaient de voir la composition à priori, de

distinguer au préalables les différentes évolutions possibles, soit en proposant directement des composants sur l'étagère, soit en représentant le comportement des composants par des machines à états, et par conséquent , spécifier des états cibles. Nous proposons donc de reléguer l'adaptation au niveau de l'exécution. C'est à ce niveau là que nous jugeons judicieux d'appréhender cette phase d'adaptation par une approche combinant paradigmes agents et composants.

4.3. Adaptation dynamique

De nombreuses recherches essaient d'allier les avantages des paradigmes agents et composants (Briot, 2005). D'un côté, les agents bénéficient d'une autonomie naturelle, autonomie en terme d'évolution, de comportement et d'adaptation. D'un autre côté, les composants bénéficient d'une forte maturité et d'une certaine validation pour la représentation de systèmes, avec les approches guidées par les modèles. Le rapprochement souvent fait amène à penser que les agents peuvent amener aux composants un gain en autonomie et adaptativité au niveau des interactions et de l'organisation, et les composants amènent leur potentiel en formalisation grâce aux approches orientées modèles.

Ainsi nous proposons de faire intervenir des agents pour :

- exprimer les besoins (interfaces requises), en s'appuyant sur une description sous forme de méta-informations.
- Parcourir les cas de réutilisation existants : recherche de cas de réutilisation correspondant aux interfaces requises exprimées par d'autres composants.
- Négocier l'adaptation d'un composant d'origine. Lorsque le système nécessite l'adaptation d'un composant, les agents permettent de sélectionner les points d'ajustements adéquats.

Afin de permettre aux agents de travailler ainsi, nous introduisons :

- le concept de rôle (Hameurlain, 2004). Terme souvent employé dans la terminologie du monde des agents, le rôle se résume, pour nous, à un cas de réutilisation d'un composant (c'est à dire à la mise en jeu de points d'ajustements). Ainsi, un composant s'engage à jouer un rôle au travers d'un cas de réutilisation. Chaque composant est capable de jouer un ou plusieurs rôles (rôles fournis). Et, suite à de nouveaux besoins ou changements du système, de nouveaux rôles sont requis (rôles requis).
- des règles. Elles permettent de définir, d'une part, le(s) rôle(s) requis par un composant, et d'autre part, les rôles qui peuvent être joués. Par rapport à un rôle donné, basé sur des objectifs ou des besoins, les règles définissent quels points d'ajustement sont disponibles. Elles sont amenées à tenir compte aussi bien des propriétés non-fonctionnelles (QoS, sécurité, …) que des propriétés fonctionnelles décrites dans les méta-informations associées aux points d'ajustements.

Exemple de rôles fournis par un service e-Portfolio : *tuteur pour les procédures d'atterrissage, apprenant de niveau 2 pour les procédures de décollage…*

Exemple de règles : *rôle de tuteur : {SO2, SO3} pour apprenant de niveau 2 maximum, fiabilité 4. Rôle d'apprenant : {SO1} pour tuteur seulement.*

Pour chaque composant du système nous attribuons un agent responsable de son adaptation. Raisonnant sur les possibilités offertes par les règles, il crée les nouveaux rôles joués ou publie les rôles requis. Ainsi des agents négociant des rôles entre composants permettent une adaptation dynamique du système en intégrant le caractère « à la demande » des besoins créés contextuellement.

5. Conclusion

Basés sur de précédents travaux menés au sein de l'équipe autour de la définition du modèle Ugatze, nous proposons d'étendre ce modèle destiné à la réutilisation dans un environnement statique vers une adaptation dynamique. Dans un premier temps nous avons intégré le principe de variabilité au modèle existant avec l'ajout de points d'ajustements qui permettent de décliner un composant source en cas de réutilisation. Nous avons ensuite proposé une approche introduisant des agents pour la mise en œuvre du modèle étendu. Nos travaux sont cependant en cours de développement et plusieurs aspects restent à approfondir. Ces travaux trouveront leur champ d'application et une implémentation au cœur du projet européen EleGI (Gaeta *et al.*, 2004) auquel nous participons.

Nous pensons de plus que les mécanismes (agents, rôles et règles) introduits, pour rendre l'adaptation de composants dynamique, sont une piste intéressante pour étendre le modèle à une intégration dynamique aussi.

6. Bibliographie

Aniorté P., Vers des systèmes distribués et hétérogènes : Une approche basée composants guidée par les modèles, Habilitation à Diriger des Recherches, Université de Pau et des Pays de l'Adour, 21 octobre 2004

Briot J.P. – Foreword - In Ricardo Choren, Alessandro Garcia, Carlos Lucena, and Alexander Romanovsky, editors, *Software Engineering for Multi-Agent Systems III*, Number 3390 in Lecture Notes in Computer Science, Springer Verlag, 2005, pages V-VI. (Invited).

Cauvet C., Semmak F., « La réutilisation dans l'ingénierie des systèmes d'information », *Dans Génie objet - Analyse et conception de l'évolution,* Sous la direction de OUSSALAH C., pp. 25-55, Hermès, 1999

Czajkowski K., Ferguson D., Foster I., & al., From Open Grid Services Infrastructure to WS, Resource Framework: Refactoring & Evolution , Version 1.1, Global Grid Forum, May 2004.

Foster,I., Kesselman,C., Nick,J. and Tuecke,S., « The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration », June 2002.

Gaeta M., Ritrovato P., The European Learning Grid Infrastructure Integrated Project, ERCIM News n. 59, October 2004.

Hameurlain, N. « Un Modèle de Spécification et d'Implémentation de Composants-rôles pour les Systèmes Multi-Agents. »Journées Multi-Agents et Composants, JMAC'04, Ecole des Mines de Paris, novembre 2004

Krueger C.W., « Software reuse », *in ACM Computing Surveys*, vol. 24, n°2, June 1992

Lhuillier M., Une approche à base de composants logiciels pour la conception d'agents, Principes et mise en œuvre à travers la plateforme MALEVA, Thèse de l'Université Paris VI, Février 1998.

McCarthy D.R., Dayal U., « The Architecture of an Active Data Base Management System » *Proceeding ACM-SIGMOD Conference*, Portland, May 1989, pp 215-224.

Neighbors J.M., « The DRACCO approach to constructing software from reusable components », *IEEE transactions on software engineering SE-10(5)*, p.564-574 , 1984

Parnas D., « On the design and development of program families », *IEEE Trans. Software engineering*, March 1976.

Razmerita, L., Gouarderes, G., Comte, E., « Ontology-based User Modeling and e-Portfolio Grid Learning Services», *in Applied Artificial Intelligence Journal,* Francis & Taylor (eds.), 2005.

Rheingold, H., *The Virtual Community: Homesteading on the Electronic Frontier*, Reading, MA: Addison-Wesley Pub. Co, 1993.

Seyler F., Ugatze : métamodélisation pour la réutilisation de composants hétérogènes distribués, Thèse de l'Université de Pau et des Pays de l'Adour, 16 décembre 2004.

Tuecke S., Czajkowski K., Foster I., Frey J., Graham S., Kesselman C., Maguire T., Sandholm T., Vanderbilt P., and Snelling D., Open Grid Services Infrastructure (OGSI) Version 1.0, Global Grid Forum Draft Recommendation, June 2003.

Modélisation de systèmes complexes distribués : l'ingénierie des modèles pour l'intégration des paradigmes « agent » et « composant »

Philippe Aniorté[1], **Eric Cariou**[2] **et Eric Gouardères**[2]

[1] *Laboratoire Informatique de l'Université de Pau et des Pays de l'Adour (LIUPPA)*
IUT de Bayonne – Département Informatique
Place Paul Bert – 64100 Bayonne
aniorte@iutbayonne.univ-pau.fr

[2] *Laboratoire Informatique de l'Université de Pau et des Pays de l'Adour (LIUPPA)*
Université de Pau – Département Informatique – B.P. 1155
64013 Pau CEDEX
Eric.Gouarderes@univ-pau.fr, Eric.Cariou@univ-pau.fr

RÉSUMÉ. *Cet article s'intéresse dans un premier temps à la modélisation et à la simulation des systèmes complexes qui constituent notre champ d'investigation. Après avoir mis en évidence les besoins de cette communauté, des éléments de solution basés sur le paradigme « agent » sont listés. Dans un second temps nous présentons les grandes lignes d'Ugatze, un modèle de composants logiciels développé au sein de l'équipe, dédié à l'ingénierie de systèmes complexes distribués, basé sur le concept d'interaction. Enfin, il décrit l'état de nos réflexions concernant le recours à l'ingénierie des modèles (IDM ou MDE) pour intégrer les paradigmes « agent » et « composant » et définir une architecture de haut niveau pour la simulation distribuée.*

MOTS-CLÉS : MODELISATION, SIMULATION, AGENT, COMPOSANT, INGENIERIE DES MODELES (IDM, MDE)

Introduction

Jusqu'il y a peu, les communautés « composant » et « agent » étaient relativement cloisonnées, dans le sens ou peu d'équipes travaillaient à l'intersection de ces deux paradigmes. Le domaine des composants, que nous restreignons dans cet article aux « composants logiciels », est le creuset de nombreuses propositions, en matière de modèles (« industriels » et « académiques »), de démarches, et d'outils, issus essentiellement de la communauté « Génie Logiciel » mais également de la communauté « Systèmes d'Information ».

Dans le même temps, les Systèmes Multi-Agents et les propositions associées, issus de la communauté « Intelligence Artificielle », ont prouvé leur bien-fondé dans différents champs d'application, en particulier les systèmes complexes distribués, et plus spécialement la simulation. Récemment, une réflexion menée par des chercheurs issus des communautés citées précédemment a conduit à mettre en évidence des besoins croisés, et à s'interroger, entre autres, sur :

– des applications utilisant conjointement les deux approches,

– des méthodes d'analyse et de conception intégrant ces deux approches.

Le projet présenté dans cet article s'inscrit dans cette tendance. Il s'agit d'intégrer les résultats obtenus dans le domaine de la modélisation de structures d'organisations multi-agents, et ceux obtenus en matière de modélisation de composants et d'interactions (entre composants). Plus précisément, l'objectif consiste, en s'appuyant sur les principes de l'Ingénierie Dirigée par les Modèles (IDM ou MDE), à élaborer des propositions visant à construire un modèle d'architecture de haut niveau pour la simulation distribuée permettant l'intégration de composants de simulation, de systèmes opérationnels et d'opérateurs humains (superviseurs, décideurs, utilisateurs). Outre ce modèle d'architecture, nous envisageons des outils d'édition et de transformation de modèles pour permettre la projection vers des plates-formes de mise en œuvre. En particulier, des outils formels devront garantir la validité des transformations. Le champ d'application de ces travaux est la formation aéronautique (pilotage, maintenance…).

Dans un premier temps, nous présentons la problématique de la communauté « Modélisation et Simulation » (M&S) ainsi que les avantages du paradigme « agent » en la matière. La deuxième partie vise à présenter les grandes lignes d'un modèle de composants logiciels autonomes conçu au sein de l'équipe et basé sur le concept d'interaction. Puis, nous décrivons dans la troisième partie l'état de nos réflexions concernant l'apport de l'approche IDM, support à l'intégration des paradigmes « agent » et « composant » pour la conception de systèmes de simulation distribués.

1. Modélisation et simulation des systèmes complexes

La communauté Modélisation et Simulation (M&S) est née des besoins induits par l'étude et la modélisation des systèmes complexes mais aussi par leur exploitation et leur maintenance. Un système complexe est défini comme un ensemble d'éléments en interaction entre eux et avec l'extérieur. Par essence il s'agit de systèmes ouverts, hétérogènes dans lesquels les interactions sont non linéaires et dont le comportement global ne peut être obtenu par simple composition des comportements individuels.

1.1. Problématique

Depuis une dizaine d'années, de nouvelles architectures de systèmes d'information ont émergé. Elles sont caractérisées par l'absence d'autorité centrale, des composants faiblement couplés et la capacité à répondre – réactivité – et à s'adapter – flexibilité – rapidement aux nouveaux besoins et aux nouvelles contraintes selon des principes d'auto-organisation [MASS99]. Le World Wide Web en est l'exemple le plus connu, mais on peut trouver des exemples dans le domaine de la production (chaînes logistiques, e-manufacturing), des transactions financières (e-business) de la formation (e-learning) et l'entraînement (e-training). Par contre ces nouvelles architectures ont eu peu d'impact dans le domaine de la simulation jusqu'au milieu des années 90 où le concept de "Simulation basée Web" (Web Based Simulation, WEBSIM http://www.websim.net/) a été proposé, et est notamment supporté depuis par l'OMG. L'objectif est d'encourager le développement et l'application de standards ouverts et de démarches conceptuelles qui permettent à la communauté M&S d'exploiter ces nouvelles architectures afin d'intégrer leurs produits dans des systèmes à plus grande échelle. La vision est celle d'un système distribué basé sur une intégration horizontale de composants M&S, de systèmes opérationnels et d'opérateurs humains (superviseurs, décideurs, utilisateurs). Chaque partie de ce système étant autonome, interfaçable avec les autres parties pour fournir ou consommer des données en temps réel.

Parallèlement, le Département de la Défense américain (DoD) a investi de façon significative pour soutenir des projets dans le domaine de la simulation distribuée. Il a encouragé l'évolution de standards pour supporter la réutilisation et l'interopérabilité de simulations ce qui a abouti à HLA (High Level Architecture [DAHM97]) standardisé par l'OMG (HLA 1.3) et l'IEEE (HLA 1516) et peut être reconnu aujourd'hui comme un standard de facto pour résoudre les problèmes liés à l'intégration, la réutilisation et l'interopérabilité dans le domaine de la simulation distribuée. Si c'est effectivement le cas dans le domaine militaire, la diffusion dans l'industrie non militaire des technologies de simulation distribuée reste confidentielle. Les raisons sont multiples et analysées dans un article de synthèse par un ensemble de chercheurs et de professionnels du domaine [TAYL02]. On peut citer :

- Le manque de méthodologies et d'outils pour aborder la modélisation d'un système complexe à différents niveaux de granularité.
- Le manque de standards pour décrire les interactions (échanges de données) entre composants.
- Le manque de flexibilité : capacité d'adaptation et de configuration dynamique du système. La flexibilité doit permettre de simuler des scénarios complexes fondés sur les points de vue multiples de chaque acteur (simulateur, opérateur humain) où les règles, les contrats et la structure du réseau peuvent changer en cours de simulation.

1.2. Un élément de solution : le paradigme « agent »

Les systèmes multi-agents peuvent apporter des contributions significatives aux limites enoncées ci-dessus en fournissant des modèles et des méthodologies de conception de haut niveau basés sur une approche centrée organisation (OCMAS, Organization Centered Multi-Agent Systems, [FERB03]), des protocoles standards d'interaction adaptés (ex : FIPA Interaction protocols[1]) et des plates-formes d'exécution distribuées (ex: MADKIT[2], JADE[3]). En terme d'architecture de simulation distribuée, une approche consiste à construire des applications de simulation multi-agents au-dessus de HLA. ACES [ARON03] est un exemple d'architecture flexible et modulaire exploitant le paradigme agent et HLA, pour l'analyse des systèmes d'échange dans l'aviation civile. Actuellement, le projet Federation Grid[4] a pour objectif le développement d'un environnement de simulation flexible proposé comme un ensemble de web (grille) services pour les jeux de stratégies. Dans ces deux exemples, HLA fournit un support pour l'interopérabilité des différents composants et les agents sont utilisés pour la flexibilité et la modularité du système global. C'est pourquoi un des enjeux de ce qui est appelé aujourd'hui « Agent Directed Simulation » est de fournir des services d'agents « plug and play » pour les applications futures sur la grille.

Agent-Directed Simulation [OREN00] est proposé comme un nouveau paradigme qui couvre les différentes approches et relations entre agents et simulation. Il s'agit de promouvoir, au sein de la communauté modélisation et simulation, l'utilisation des agents logiciels comme assistants pour le développement de systèmes et la conduite d'expérimentations en simulation. Trois types de synergies ont été identifiées: (1) au niveau conceptuel, les agents sont utilisés comme métaphores pour la modélisation de systèmes (agent simulation), (2) au niveau développement, les agents sont utilisés comme métaphore de programmation pour faciliter la réutilisation de modèles et de services de simulation (agent supported

---

[1] FIPA, Foundation for Intelligent Physical Agents, FIPA Interaction Protocols Technical Commitee, http://www.fipa.org/activities/interaction_protocols.html, 2003

[2] Multi Agent Development KIT. http://www.madkit.org

[3] http://sharon.cselt.it/projects/jade/

[4] http://www.federationgrid.org/, http://www.magnetar.org/sdk/

simulation), (3) les agents utilisent la simulation comme méthode de décision ou la simulation est utilisée comme technique expérimentale pour évaluer le comportement de sociétés d'agents (agent based simulation).

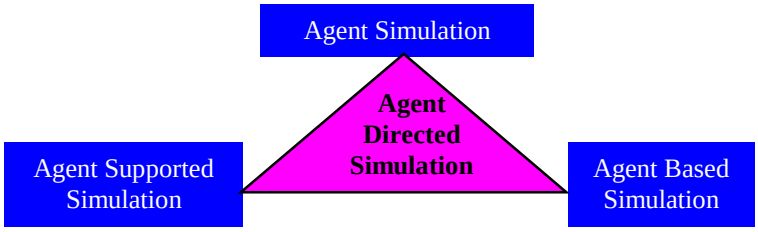


Figure 1 : Agent Directed Simulation

Nous avons exploité en particulier les synergies mises en évidence ci-dessus, pour proposer un cadre de développement d'environnement de simulation de systèmes de pilotage de la production [GOUA05]. La contribution principale est la modélisation de structures de contrôles distribuées. La structure organisationnelle régissant les interactions s'appuie sur le modèle AALAADIN [GUTK99], aujourd'hui AGR (Agent Groupe Rôle). Les groupes sont créés pour prendre en compte des objectifs (respect des délais, des quantités, minimisation des coûts…). Une structure abstraite précise les relations structurelles entre rôles au sein des groupes, ce qui permet une description simple de structures d'organisations et d'interactions quel que soit le niveau de granularité considéré (par exemple cellule, Atelier, Entreprise) [TCHI04].

Nous avons choisi AGR car c'est un méta-modèle qui respecte notamment trois principes le rendant générique pour décrire différents modèles organisationnels [FERB03]: (1) le niveau organisationnel décrit le "quoi" et non le "comment"; Il borne le domaine d'activité des agents mais ne décrit pas le comportement de ceux-ci, (2) aucune contrainte ne doit être posée sur la dimension cognitive des agents, qui peuvent être intentionnels ou réactifs et (3) une organisation est un moyen de partitionner un système. Chaque partition ou groupe définit le contexte d'interaction des agents de du groupe. En particulier des agents de groupes différents ne peuvent pas communiquer directement. D'autres modèles organisationnels tels que STEAM [TAMB97] ou MOISE [HANN00] auraient pu être envisagés mais ils sont plus spécifiques et concernent des agents intentionnels (notion d'engagement à partir de buts et de plans pour les réaliser). Nous avons privilégié plutôt une approche émergente.

2. Un modèle de composants autonomes

Au cours de ces dernières années, l'équipe a développé et expérimenté (dans le cadre du projet européen ASIMIL) un (méta)modèle de composants dénommé « Ugatze », adapté à la réutilisation de composants logiciels autonomes, hétérogènes

et distribués. Ce modèle de composants est défini précisément via un méta-modèle, ce qui permet de manipuler des modélisations Ugatze via des outils dans le cadre d'un processus logiciel de type « Ingénierie Dirigée par les Modèles » (IDM ou MDE pour Model-Driven Engineering – voir section 3). Ces aspects n'étant pas développés dans cet article, nous renvoyons le lecteur intéressé à [ANIO04]. Ce méta-modèle repose sur deux notions essentielles :

– l'interface du composant. C'est le résultat de la (re)spécification du composant, activité propre à la réutilisation [CAUV99]. La représentation de tous les composants à réutiliser est « unifiée » (au sens des besoins exprimés pour l'interopérabilité) via le méta-modèle.

– l'interaction entre composants. Elle permet de gérer l'interopérabilité entre composants.

2.1. L'interface du composant

La spécification des composants réutilisables telle que nous la pratiquons repose sur le concept « d'interface » constituée de « points d'interaction ». Ils sont dédiés à l'intégration du composant dans le système à développer, rendant effective sa réutilisation.

Nous distinguons différents types de points d'interaction. L'autonomie des composants et le principe de découplage associé, reposant entre autres sur la séparation du flot d'information et du flot de contrôle [LHUI98], nous conduisent à recenser :

– des points d'interaction de donnée,

– des points d'interaction de contrôle.

Considérant l'aspect flux, nous distinguons :

– des points d'interaction d'entrée, flux entrant dans le composant,

– des points d'interaction de sortie, flux sortant du composant.

En croisant les deux critères, donnée / contrôle et entrée / sortie, nous obtenons :

– des points d'interaction d'entrée de donnée ou Data Input (DI),

– des points d'interaction de sortie de donnée ou Data Output (DO),

– des points d'interaction d'entrée de contrôle ou Control Input (CI),

– des points d'interaction de sortie de contrôle ou Control Output (CO).

2.2. L'interaction entre composants

Le méta-modèle proposé est basé sur le principe de découplage entre composants. L'interaction s'appuie sur ce principe. Elle permet de gérer l'interopérabilité conceptuelle entre composants.

Les composants réutilisables, modélisés à l'aide des points d'interaction, sont destinés à être intégrés afin de développer le nouveau système. Au sein de ce

système, ils doivent pouvoir interopérer. L'interopérabilité est gérée à l'aide du concept d'interaction. La mise en œuvre des interactions est basée sur les points d'interaction présentés précédemment. Nous distinguons :

– interactions « prédéfinies ». Elles constituent pour le concepteur « l'offre de base » en matière d'interactions. Cette offre se caractérise par des dispositifs généraux, dont le nombre est limité, utilisés de façon récurrente. A l'instar des points d'interaction, les interactions prédéfinies se déclinent en deux catégories : interactions d'information et interactions de contrôle.

– interactions « à façon ». Elles constituent pour le concepteur une alternative à l'offre de base. L'idée est de donner au concepteur la possibilité de définir une interaction idoine, comme solution à un problème spécifique.

De façon théorique, ce dernier type d'interactions offre une solution au problème classique du nombre fini de dispositifs, nécessairement non suffisant pour traiter la variété des problèmes pouvant se poser. De ce point de vue, l'analogie avec les types dans les langages de programmation nous semble pertinente. La seule limite tient à l'expressivité de la forme algorithmique, car le concepteur est invité à définir ces interactions à l'aide d'algorithmes. Plus concrètement, cette possibilité contribue grandement à la gestion de composants autonomes, en autorisant un haut niveau de découplage, ce qui dans le contexte de nos travaux est fondamental. En conséquence, la prise en compte de l'hétérogénéité s'en trouve facilitée. In fine, la gestion de l'intéropérabilité est favorisée.

2.3. Représentation graphique et règles

Outre les notions clés déjà présentées (interface et interaction), le méta-modèle « Ugatze » se caractérise par :

– une représentation graphique. Elle permet d'instrumenter son utilisation en offrant la possibilité de manipuler les points de spécification de l'interface des composants et les interactions entre composants basées sur ces points.

– des règles de bonne utilisation. Elles permettent de vérifier les modèles instanciés à partir du méta-modèle.

Nous ne détaillons pas la représentation graphique des interactions. Nous renvoyons le lecteur intéressé à [SEYL04]. Cette représentation graphique est complétée par des règles. De telles règles ont pour objet de spécifier un ensemble de contraintes relatives aux différents éléments constitutifs du méta-modèle, interface (points de spécification) et interactions. Elles sont destinées à aider à sa bonne utilisation.

Ces règles correspondent typiquement aux règles de bonne construction (« well-formedness rules ») que l'on retrouve dans la définition de méta-modèles. Elles sont décrites en langage naturel et à l'aide du langage OCL (Object Constraint Language).

3. L'ingénierie des modèles appliquée à la simulation distribuée

L'ingénierie des modèles – ou MDE (pour Model-Driven Engineering) – est une discipline émergente qui vise à définir un nouveau mode de développement des logiciels [AS-MDA04]. Son but est de déplacer la complexité de réalisation des applications logicielles de l'écriture du code du logiciel (la programmation) vers la spécification de l'application (la modélisation). L'objectif principal du MDE est de se baser sur la spécification abstraite (une modélisation indépendante de toute mise en œuvre, de toute technologie) d'une application comme support pour générer automatiquement le code de cette application pour une plate-forme technologique donnée. Pour passer de cette spécification abstraite au code, les modèles sont transformés via une série de transformations correspondant à des raffinements ou des projections afin d'aller vers une spécification détaillant l'implémentation et enfin générer le code. Le MDE est donc une nouvelle « philosophie » de développement où les modèles sont de plus en plus considérés avec une vision productive (dans le sens où l'implémentation sera directement dérivée des modèles de haut niveau) au lieu de contemplative (dans le sens où le développeur consulte les modèles pour écrire son code).

Les technologies et thématiques de recherche autour du MDE concernent un nombre important de points comme, entre autres, des techniques et outils de définition et de gestion de méta-modèles (afin de pouvoir manipuler formellement des modèles) ainsi que des techniques et outils de transformation de modèles (pour réaliser des projections, des raffinements ou bien encore des fusions de modèles). Ces technologies sont centrales dans le cadre de notre projet et elle seront utilisées pour intégrer les approches composants et agents dans le contexte de la simulation distribuée.

Les travaux à mener pour aboutir à cette intégration peuvent être classés en deux catégories : modélisation et transformation. En ce qui concerne la partie modélisation, il s'agit de définir un ensemble de méta-modèles qui permettront de manipuler et gérer tous les types de modèles requis. Cela impliquera notamment de définir (ou spécifier plus formellement) les méta-modèles d'organisation multi-agents (comme le méta-modèle AGR) et les méta-modèles de simulation (comme HLA), puis en déduire un méta-modèle abstrait commun. A ces méta-modèles s'ajoute aussi le méta-modèle de composants Ugatze déjà existant. Pour la partie transformation, il s'agira de définir un ensemble de transformations permettant de projeter un modèle d'agents respectant notre méta-modèle abstrait vers un modèle technologique de mise en œuvre comme HLA ou bien encore le modèle de composants Ugatze. Les transformations seront de deux types : (1) transformation d'une spécification abstraite en une spécification concrète par rapport à une plate-forme technologique et (2) génération de code pour une plate-forme donnée à partir d'une spécification concrète. Dans ce cadre de transformation, une attention particulière sera portée à la définition d'outils permettant de valider les transformations de modèles [CMSD04]. Les relations entre les modèles, méta-

modèles et les diverses transformations sont représentées sur la figure 2. Une application est spécifiée à un niveau abstrait, via les concepts de notre méta-modèle abstrait. Ensuite, cette spécification abstraite est par exemple projetée en une spécification concrète par rapport à la plate-forme HLA. Enfin, à partir de cette spécification concrète, le squelette du code de l'application implémentée sur HLA est généré.

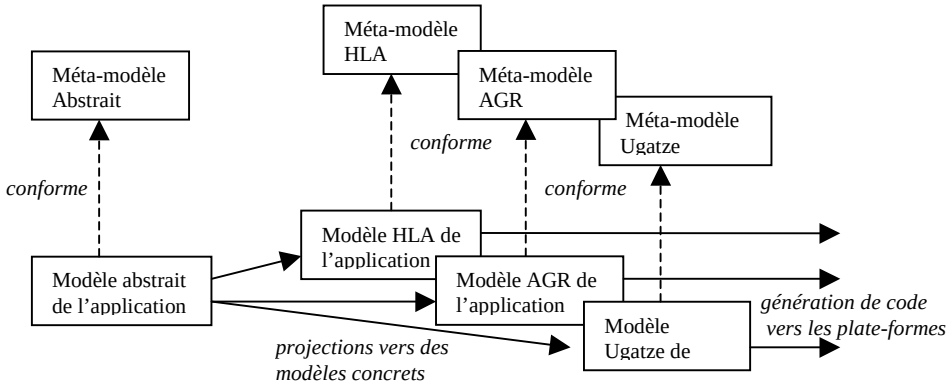


Figure 2 : Modèles et méta-modèles

Notre objectif est donc d'être capable tout d'abord de spécifier une application de simulation distribuée à un niveau abstrait, indépendamment de la plate-forme de mise en œuvre. Ensuite, des transformations de modèles permettront de projeter une telle spécification vers plusieurs plates-formes, soit de type multi-agents, soit de type composants soit de type dédiée comme HLA. Il est également possible d'envisager des projections « mixtes » comme par exemple une mise en œuvre utilisant des composants Ugatze par-dessus une plate-forme multi-agents ou bien encore la mise en œuvre d'une structure d'organisation de type AGR au dessus d'une plate-forme HLA.

La définition du méta-modèle abstrait et des concepts qu'il intègre est évidemment un point crucial. Deux approches sont possibles. La première consiste à prendre le plus grand ensemble des concepts communs ou proches à tous les méta-modèles concrets considérés (on pourra notamment y trouver un concept mixte entre agent et composant qui correspondrait directement au concept de composant dans une plate-forme composants et au concept d'agent dans un approche multi-agents). La seconde approche consiste à définir un ensemble plus large de concepts intégrant des concepts qui ne sont pas présents dans tous les méta-modèles. Par exemple, on intégrerait les notions de groupe et de rôles qui ne sont pas transposables directement en concepts d'une plate-forme composants. Il faudrait alors définir une règle de projection traduisant un concept abstrait en un ensemble d'éléments concrets pour une plate-forme donnée (au lieu d'une correspondance directe entre un concept abstrait et un concept concret comme pour la première approche). Cette seconde approche est évidemment plus complexe à mettre en œuvre, mais elle

présente l'avantage de définir un méta-modèle plus fourni et donc des modèles abstraits plus évolués.

4. Conclusion

Cet article est une contribution à porter au crédit des travaux se situant à l'intersection des paradigmes composant et agent. D'une part, il concerne des applications pouvant bénéficier conjointement des deux approches. En l'occurrence, il s'agit des nombreuses applications issues du domaine de la simulation distribuée, dont nous avons dressé les besoins au cours de la première partie.

D'autre part, cet article concerne aussi les méthodes d'analyse et de conception intégrant ces deux approches. Plus précisément, il s'agit d'intégrer en la matière les avantages du paradigme « composant » et du paradigme « agent ». Le premier a démontré ses atouts pour la réutilisation et la modélisation. Le modèle Ugatze développé au sein de l'équipe et ébauché dans la deuxième partie en est un exemple. Par ailleurs, le paradigme « agent » a donné lieu à un certain nombre de propositions qui constituent des éléments de réponse aux problèmes de la communauté M&S.

Notre contribution vise à recourir à l'approche MDE, basée entre autre sur la méta-modélisation et les transformations de modèles, pour intégrer les avantages de ces deux paradigmes, afin de définir des architectures de haut niveau pour la simulation distribuée. Nos propositions, présentées dans la troisième partie, nécessitent cependant d'être approfondies, ce qui constitue la perspective majeure des travaux présentés dans cet article.

Bibliographie

[ANIO04] Aniorté P. - Vers des systèmes distribués et hétérogènes : Une approche basée composants guidée par les modèles - Habilitation à Diriger des Recherches, Université de Pau et des Pays de l'Adour, 21 octobre 2004

[ARON03] Aronson J., Manikonda V., Peng W., Levy R., and Roth K. 2003. "An hla compliant agent-based fast-time simulation architecture for analysis of civil aviation concepts". In proc. of Spring SISO Simulation Interoperability Workshop, Orlando.

[AS-MDA04] Rapport de Synthèse de l'Action Spécifique (AS) CNRS sur l'Ingénierie Dirigée par les Modèles (IDM), Jean Bézivin, Mireille Blay, Mokrane Bouzeghoub, Jacky Estublier, Jean Marie Favre, Sébastien Gérard, Jean-Marc-Jézéquel - 2004, http://www-adele.imag.fr/mda/as/

[CAUV99] Cauvet C., Semmak F. - La réutilisation dans l'ingénierie des systèmes d'information - Dans Génie objet - Analyse et conception de l'évolution, Sous la direction de Oussalah C., pp. 25-55, Hermès, 1999

[CMSD04] Eric Cariou, Raphaël Marvie, Lionel Seinturier, and Laurence Duchien. OCL for the Specification of Model Transformation Contracts. Workshop OCL and Model Driven

Engineering of the Seventh International Conference on UML Modeling Languages and Applications (UML 2004), October 2004.

[DAHM97] Dahmann, J.S., R.M. Fujimoto, and R.M. Weatherly. . The Department of Defense High Level Architecture. In Proceedings of the 1997 Winter Simulation Conference, ed. S. Andrad´ottir, K.J. Healy, D.H. Withers, and B.L. Nelson, 142-149, Association of Computing Machinery, New York, NY, 1997.

[FERB03] Ferber J., Gutknecht O., Michel F. 2003. From Agents to Organizations: an Organizational View of Multi-Agent Systems. In proceedings of Fourth International Workshop on Agent Oriented Software Engineering (AOSE), 2003.

[GOUA05] Gouardères E., Tchikou M., Lamarque N., Agent-based framework for simulation in manufacturing control. Proceedings of the 2005 European Simulation and Modelling Conference, EUROSIS-ETI Publication, pp 95-100, Porto, Portugal, 2005.

[GUTK99] Gutknecht, O. , Ferber, J. (1999). Vers une méthodologie organisationnelle de conception de systèmes multi-agents. Rapport de recherche de LIRMM, n°99073, 1999.

[HANN00] Hannoun M., Boissier O., Sichman J., Sayettat C. MOISE: An Organizational Model for Multi-agent Systems. Monard, Sichman Eds, IBERAMIA/SBIA, Springer LNAI 1952, 2000.

[LHUI98] Lhuillier M. - Une approche à base de composants logiciels pour la conception d'agents - Principes et mise en œuvre à travers la plateforme MALEVA, Thèse de l'Université Paris VI, Février 1998

[MASS99] Massote P. Auto-organisation dans les structures et les systèmes, Actes de la deuxième conférence francophone de Modélisation et SIMulation, p. 21-29, 1999.

[OREN00] Ören T.I., Numrich S.K., Wilson L.F., Uhrmacher A. M., Gelenbe E. "Agent-directed simulation - challenges to meet defense and civilian requirements". In Proc. of the 2000 Winter Simulation Conference. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds., 2000.

[SEYL04] Seyler F. - Ugatze : métamodélisation pour la réutilisation de composants hétérogènes distribués - Thèse de l'Université de Pau et des Pays de l'Adour, 16 décembre 2004.

[TAMB97] Tambe M. Towards flexible teamwork. Journal of Artificial Intelligence research (JAIR), n°7, pp. 83-124, 1997.

[TAYL02] Taylor S. J E., Fujimoto R., Straßburger S., Bruzzone A., Boon P. G., Ray J.P. Distributed Simulation and Industry : Potentials and pitffalls. In Proc. of the 2002 Winter Simulation Conference, E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds., 2002.

[TCHI04] Tchikou M., Pilotage temps réel basé sur les concepts d'organisation et d'interaction SMA . Xèmes Journées Francophones sur les Systèmes Multi-Agents, Hermès Science publications, ISBN 2-7462-1021-5, pp. 209-222, Paris, France, 24-26 Novembre 2004.

Lisaac agent

Petit modèle agent pour langage à prototype

Benoît Sonntag* — **Pierre-Alexandre Voye****

* *INRIA-Lorraine / LORIA, 615 Rue du Jardin Botanique, BP 101, 54602 VILLERS-LES-NANCY Cedex, FRANCE*
bsonntag@loria.fr
** *La roche, 44360 Vigneux de Bretagne*
pierreavoye@oceanetpro.net

RÉSUMÉ. *Nous proposons dans cet article les bases d'un modèle simple de langage orienté agents réactifs pour le langage Lisaac, langage objet à prototype compilé. Nous présentons les axiomes de bases à implémenter dans le compilateur afin d'intégrer des mécanismes agents, en s'adaptant aux spécificités d'un langage objet à prototype. Nous proposons l'intégration de mécanismes comme la gestion de messages multicast, la définition du comportement réactif de l'agent, un mécanisme de structuration de SMA, et un mécanisme d'extraction structuré de données.*

ABSTRACT. *In this article, we suggest, the basis of a simple agent oriented language for the compiled and prototype-based language Lisaac. We show the basis axioma that we have to implement in the compiler in order to implement agent mecanism and adapt the model to a prototype-based and compiled language. We suggest a mecanism such as multicast messages, a reactive behaviour of the agent, a MAS structuration mecanism, and a structured data extracting system.*

MOTS-CLÉS : *Programmation Orienté Agent, modèle agent, langage à prototype, Lisaac*

KEYWORDS: *Agent Oriented Programming, agent model, prototype-based language, Lisaac*

1. Introduction

Bon nombre de logiciels sont modélisables en termes de petites unités logicielles collaborant ensemble afin de résoudre les problèmes qu'on lui soumet. Les paradigmes de programmation objets ont considérablement amélioré la tâche du concepteur logiciel en lui apportant un mode de découpage plus proche d'un mode de pensé humain. On peut d'ailleurs remarquer que la montée en niveau des langages de développement logiciel est concomittente à une proximité toujours plus accrue des modes de pensés et de représentations humains. Le paradigme objet trouve néanmoins ses limites dans la réalisation de logiciels excédents plusieurs dizaines de millions de lignes en imposant des architectures très complexes, peu souples et peu fiables (Livshitz, 2004). Cette constatation nous a amené à imaginer un modèle entièrement dédié à simplifier la tâche du développeur et surtout à lui permettre d'exprimer au mieux son intuition.

Nous pouvons synthétiser plusieurs caractéristiques de la représentation humaine :

• un être humain conceptualise le monde en terme d'objets animés ou non d'intentionnalités ;

• les objets ont tendance à se comporter en fonction de stimuli extérieur ;

• ces objets se transmettent des informations entre eux, et ces informations peuvent donner lieu à une modification d'un comportement ;

• une cause implique un effet et vice-versa ;

• les informations manipulées dans les programmes représentent des perceptions humaines, et possèdent donc une structure ontologique

Notre modèle a pour objectif d'intégrer ces concepts tout en restant dans le cadre d'un langage objet à prototype classique. Si la modélisation d'une fourmillière nécessite un langage orienté agent, des calculs matriciels se satisfont parfaitement d'un langage orienté objet classique. Les deux approches sont complémentaires. Par conséquent, l'une des contraintes que nous avons imposées à notre modèle est de concevoir une extension élégante dans son uniformité au langage Lisaac. Ainsi, notre modèle respecte les grandes lignes de ce langage et en particulier son caractère typé.

Les extensions ainsi prévues sont principalement orientées vers la conception de SMA réactifs , l'approche par but étant particulièrement problématique à compiler d'une part et d'autre part s'intègre difficilement à un langage objet à prototype classique. Néanmoins, le langage Lisaac étant un langage objet à prototype classique, on peut facilement doter un agent d'une représentation du monde. de plus le système de message présenté permet d'aller plus loin que la définition d'agent réactif purs, bien que l'on ai pas cherché à fournir des outils permettant de réaliser des systèmes multi-agents cognitifs.

Nous identifions plusieurs mécanismes permettant d'atteindre ce but :

1) Chaque agent doit pouvoir se comporter et réagir en fonction de son environnement direct. On défini donc des clauses environnementales déclenchant un comportement spécifique à la satisfaction de ses clauses.

2) Il est nécessaire, pour aider le concepteur d'un SMA, de récupérer facilement des données sur un ensemble d'objets ou d'agents. Nous proposons un système d'extraction de données, inspiré de SQL et adapté au modèle objet. Ici, un identifiant ou une référence d'un objet n'est plus nécessaire à son accès.

3) Les agents doivent pouvoir s'envoyer des messages en multicast en conditionnant la réception de ceux-ci à des critères définissables.

4) Le concepteur de SMA doit pouvoir posséder un outil lui permettant de représenter plusieurs niveaux de systèmes multi-agent, un SMA pouvant être un agent d'un autre SMA.

1.0.1. *Comparaison de notre approche avec d'autres contributions*

De nombreux langages ont été créés pour implémenter aisément des systèmes multi-agents. Ces langages tentent la plupart du temps de permettre de définir des SMA dotés d'agents cognitifs en offrant des fonctionnalités plus ou moins poussées en fonction de l'objectif initial.

Citons

1) AgentSpeak (Weerasooriya *et al.*, 1995) est un langage dans lesquels les agents sont autonomes et dotés d'états mentaux comme des croyances, objectifs, plans et intentions. AgentSpeak implémente un système de *trigger* permettant de créer des SMA constitués d'agents réactifs. Ce langage est exclusivement conçu pour créer des SMA. AgentSpeak n'est pas, à la base, un langage objet à prototype et ne permet pas d'utiliser les propriétés objets classiques.

2) Viva (G., 1996) est un langage combinant la programmation orienté aspect et des concepts issu de PROLOG. De même que AgentSpeak, il permet de définir croyances, objectifs, intentions et tâches. Il permet de définir des prédicats à la manière de SQL ainsi que le comportement de l'agent en fonctions des états mentaux et des évènements. VIVA est un langage orienté agent, suivant les précepts défini par Shoham3 (**?**) Cette approche très intéressante s'intègre difficilement avec un langage objet à prototype, constituant en elle même un paradigme fouillé de programmation et surtout trop spécifique. Viva n'est pas un langage envisageable dans un contexte industriel.

3) DIMA (Guessoum *et al.*, 1997) est l'extension de la plate-forme Actalk constituée afin de pouvoir programmer des SMA cognitifs dans lesquels chaque agent est doté d'un module de raisonnement. La réponse du module est calculé en fonction des ses perceptions, connaissances (grâce à une base de règles) ou de messages. Actalk étant basé sur SmallTalk (un langage objet à classe non typé), notre approche, moins ambitieuse et à base de prototype, reste orthogonal à ces travaux.

2. Le langage Lisaac - un langage objet à prototype

Le choix du langage objet pour l'application de notre modèle agent se justifie essentiellement sur deux points cruciaux. Tout d'abord, les langages à classe reflètent

d'une manière moins intuitive la vision que l'on peut donner à un agent. Le paradigme *classe/instance* des langages à classe compilés implique que la description statique d'un classe n'est pas directement présent et vivant dans l'univers de notre application. Une instance est alors nécessaire pour qu'un premier représentant d'une classe soit présent et vivant en mémoire. Une classe ne permet pas de visualiser celle-ci directement comme un agent, mais plutôt comme un schéma de construction d'un agent. En effet, une classe est un moule qui, dans les langages objets à classes compilés, ne sont vivant qu'après instanciation. Ce qui donne lieu à des différence entre le modèle et l'instanciation. Une classe B héritant de A, est à l'instanciation un objet unique intégrant A et B. A *contrario*, la description d'un prototype constitue directement un objet présent en mémoire, utilisable tel quel ou clonable si nécessaire. Ainsi, un prototype est directement en adéquation à l'image d'un premier "individu agent", comme Adam et Ève vivant dans notre SMA. Un objet B héritant de A donne lieu à deux objets A et B indépendants. Le second point est d'ordre plus pragmatique. Si nous voulons démocratiser la programmation agent, de nombreux programmeurs sont sensible à un point essentiel : les performances à l'exécution. Le langage Lisaac est actuellement l'unique langage à prototype possèdant un compilateur ayant de bonnes performances à l'exécution (Sonntag, 2005).

Dans le cadre de cet article, nous ne pouvons pas réaliser une présentation détaillé du langage Lisaac. Le manuel de référence (Sonntag *et al.*, 2003) est disponible sur le site du projet Isaac (`http ://IsaacOS.loria.fr`).

Pour donner une vue globale de Lisaac, notons simplement qu'il est syntaxiquement et sémantiquement proche du langage objet à prototypes Self (Ungar *et al.*, 1987), ou relativement proche du langage SmallTalk (Goldberg *et al.*, 1983), (Goldberg, 1984) (ce dernier étant néanmoins un langage à classe). En revanche, contrairement à Self, Lisaac se distingue par un système de type proche du langage Eiffel (Meyer, 1992), avec entre autre la généricité (type paramétrique).

D'un point de vue pratique, le langage Lisaac se manipule d'une façon assez similaire à un langage comme Eiffel, voire Java, avec les particularités de l'objet prototype. C'est un langage qui, au delà de ses particularités est très rapidement maîtrisable pour un développeur habitué à un langage comme Java.

3. Caractéristiques de notre petit modèle agent

3.1. *Régles et déclaration d'un agent*

Un ensemble de règles doivent être respectées pour définir un agent :

Règle n˚1 : Un prototype représentant un agent se distingue par l'affectation de la mention AGENT dans le slot `category` de la section HEADER du Lisaac (voir fig. 1 ligne (1)).

Règle n˚2 : Un agent doit obligatoirement hériter du prototype AGENT directement ou indirectement via l'arbre d'héritage (voir fig. 1 ligne (2)). Notons que le prototype

```
section HEADER
  + name := FOURMI;
  - category := AGENT;   // (1)
section INHERIT
  * parent_fourmi:AGENT; // (2)
section PUBLIC
  - is_in:AGENT := FOURMILIERE; // (3)
```

Figure 1. *Exemple de déclaration de l'agent* FOURMI

AGENT n'est pas de catégorie agent. Le prototype AGENT implante un ensemble de fonctionnalités génériques et nécessaires aux agents. Ce point fait l'objet de la section 3.6. De plus, cela nous permet de respecter les règles de typage du langage Lisaac lors de la manipulation des agents.

Règle n˚3 : Un prototype de catégorie agent est obligatoirement un prototype feuille et ne peut donc pas être parent d'un autre prototype. Cela implique l'absence d'un prototype de catégorie agent dans l'arbre d'héritage d'un agent. Cette règle permet une cohérence et une simplification du modèle d'exécution parallèle des agents que nous développons en section 3.3.

Règle n˚4 : La déclaration du slot `is_in` de type AGENT est obligatoire dans la description du prototype AGENT (voir fig. 1 ligne (3)). Ce slot est utilisé au niveau de la hierarchisation récursive des SMA et dans la communication inter-agent. Nous autorisons toutefois son absence pour un seul agent dans le système, la section 3.2 développe la présence de ce slot.

Rappelons la significations des symboles '+','-','*' précédent les déclarations de variables.

- Le symbole '+' signifie que le slot peut être cloné.
- '-' implique que nous partageons la valeur de ce slot avec tous les clones de cet objet.
- '*' signifie que le slot contient directement une représentation de de l'objet (il ne vaut pas '`null`', il contient déjà un objet du type du slot).

3.2. *Définition récursive d'un agent*

Une des bases de la définition d'un système multi-agentsrepose sur la récursivité de la notion : un SMA peut être lui-même un agent d'un autre SMA. Dans notre exemple, chaque organe d'une fourmi est un agent appartenant au SMA définissant l'entité d'une fourmi. De même, chaque fourmi peut être considérée comme un agent du SMA représentant la fourmillière. Dans le même ordre d'idée, plusieurs

fourmillières peuvent définir un SMA englobant l'univers d'application de notre simulation.

Le slot `is_in` imposé par la règle 4 permet d'architecturer les niveaux entre eux. Dans l'exemple de la figure 1, l'agent FOURMI appartient à l'agent ou méta-agent, ou encore au SMA FOURMILLIERE. Cette appartenance étant réalisée par un slot classique, elle reste dynamique.

Un SMA (ou méta-agent) peut alors diriger et interagir en donnant des directives à ces agents de deux manières :

- via un message classique entre agents (voir section 3.4) ;
- via un message en *multicast* en utilisant comme sélection la valeur du slot `is_in`.

L'agent représentant le SMA de l'univers d'application est l'unique agent n'ayant pas de slot `is_in`. Dans notre exemple, l'ensemble des agents FOURMILIERE appartiennent au SMA englobé dans l'agent WORLD. Celui-ci étant l'univers d'application, il ne possède pas de slot `is_in` décrivant son appartenance.

3.3. *Un agent : un prototype toujours actif*

À la différence d'un objet qui n'est actif que lors d'un envoi de message, un agent doit pouvoir être toujours actif et avoir son propre comportement indépendemment des autres.

Ainsi, chaque agent a son propre contexte d'exécution. Le clonage d'un agent a pour conséquence d'obtenir un nouveau contexte d'exécution pour le fonctionnement de celui-ci. La définition de ce clonage particulier est décrite dans le prototype AGENT ; parent imposé par la régle 2 de la déclaration d'un agent. Aussi, pour éviter tout problème d'incohérence des données communes entre deux agents clonés, nous réalisons une duplication récursive de ces données (*deepclone*). Si le programmeur est désireux de partager des données à plusieurs agents, cela est toujours rendu possibles par la construction d'un agent contenant ces données. Cette encapsulation permet de gérer de manière fiable la cohérence de ces données lors d'accès multiples à un instant T.

3.4. *Communication inter-agent*

La communication est réalisée par un envoi de message classique de type `agent.slot`.

Si le slot contient du code (méthode ou procédure), deux cas sont possible :

1) Si l'appelant réclame une réponse, par exemple `a := agent.slot`, le flot d'exécution de l'appelant est stoppé jusqu'à obtention de cette valeur de retour.

2) Si l'appelant ne réclame pas de réponse, le message est envoyé, stocké dans une FIFO de l'agent appelé, et l'appelant continu son flot d'exécution. Le message sera traité ultérieurement selon le choix et le comportement de l'agent visé.

Au niveau du prototype AGENT, nous avons deux slots particuliers permettant de gérer et de maîtriser la FIFO interne des messages asynchrones :

• `has_new_message` de type BOOLEAN permet de savoir si un message est en attente dans la FIFO.

• `pop_message` qui a pour objectif de déclencher l'exécution d'un message en attente dans la FIFO.

Si le slot appelé contient une donnée, nous renvoyons directement celle-ci sans passer par la FIFO et sans couper le flot d'exécution de l'agent appelé et l'agent appelant.

Les objets passés entre agents (arguments ou valeurs de retour) sont traités selon leurs types de la manière suivante :

• Si l'objet est de type `Expanded`, objet de petite taille, comme un entier, un booléen, un caractère..., l'objet est directement passé d'un agent à l'autre.

• Si l'objet est d'un type plus complexe (chaîne de caractère, structure, ...), nous réalisons automatiquement un clonage récursif (*deepclone*) pour éviter les incohérences possible causé par le partage de cet objet durant une exécution parallèle. Si des objets complexes sont à faire passer on les stockent dans un agent virtuel.

Une gestion plus complexe d'envoi de message de type *mail* est alors rendu possible par l'implantation d'une librairie basée sur ces primitives de bases.

3.5. *Comportement*

Un agent, intrinsèquement actif, doit décrire son comportement réactif. Celui-ci exprime comment l'agent se comporte à chaque instant et plus exactement quel comportement adopte t-il en fonction de son environnement.

Chaque comportement est lié à des conditions de son environnement ou à un état interne. C'est pour cela que nous décrivons le déclenchement d'un comportement à l'aide de clauses satisfaites lors de l'application de celui-ci. Chaque clause implémente le concept de cause à effet : La satisfaction d'une clause, autrement dit d'une cause directement liée à l'agent donne lieu à un effet (le comportement proprement dit).

Les clauses se posent sous forme d'une expression booléenne comportant un ensemble de termes et de connecteurs logiques.

Dans un langage objet classique, un message ayant un identifiant est à la base du déclenchement d'une action. Ici, nous remplaçons l'identifiant de message par la clause en question. De ce fait, le déclenchement de l'action (effet) est directement lié à la satisfaction de sa clause.

Exemple :

```
section PUBLIC
  - (is_hungry) <- to_eat_action;
```

Si la clause (`is_hungry`) est vrai, l'agent réalise l'action de manger.

Tant que la clause est satisfaite, l'action se répète. Il peut être nécessaire de réaliser une action préalable au comportement répétitif (`Preface`) et une action de finalisation de ce comportement (`Postface`). Exemple :

```
section PUBLIC
  - (is_hungry) <-
  Preface { to_make_food; }
  ( to_eat_action; )
  Postface { to_clear_table; };
```

Ici, si la clause (`is_hungry`) est vrai, l'agent commence par faire à manger (`to_make_food`), puis fait l'action de manger (`to_eat_action`) tant que la clause est vrai. Avant de démarrer un autre comportement, l'agent débarrassera la table (`to_clear_table`).

Comme l'action répétée peut être interrompue à tout moment, le code présent dans la partie `Postface` permet de rendre la cohérence des données causée par la rupture brutal de l'action.

Il est possible que plusieurs clauses soit satisfaites à un même instant. Le choix du comportement prioritaire sera réalisé par une "force" de priorité définie après une clause.

Exemple :

```
section PUBLIC
  - (is_hungry) Priority 5 <- to_eat_action;

  - (has_enemy) Priority 10 <- to_beat;
```

Si l'agent a un ennemi, il fait l'action de se battre, même s'il a faim.

Notons que la priorité par défaut est de 0. En cas d'égalité des priorités, l'ordre de déclaration des comportements rentre en jeu.

3.6. *Le prototype* AGENT

Nous avons décrit dans les sections précédente un certain nombre de fonctionnalités présentes dans ce prototype. Ici, nous les rappelons rapidement en ajoutant certaines notions qui nous paraissent importantes.

```
section HEADER
  + name := AGENT;
  - category := MICRO;

section PUBLIC
  - clone:SELF <- /* code system */
  // Nouvelle défi nition du slot 'clone' (deep-clone)
  // Avec création d'un nouveau contexte d'exécution

  - has_new_message:BOOLEAN <- /* code system */
  // Vrai, si un message est en attente dans la FIFO

  - pop_message <- /* code system */
  // Retire et exécute un message de la FIFO

  - (has_new_message) Priority 0 <- pop_message;
  // Comportement par défaut: Exécution au plus tôt d'un message en attente

  - time:UINTEGER <- /* code system */
  // Renvoi l'âge de l'agent en milliseconde

  - timein:UINTEGER;
  // Variable contenant un temps, nécessaire au calcul d'une durée.

  - timeout limit_time:UINTEGER :BOOLEAN <-
  // Calcul la durée écoulé entre 'time' et 'timein'.
  // Renvoi Vrai, si cette durée est inférieur ou égal à la
  // valeur 'limit_time' ou si 'timein' est égal à 0.
  (timein = 0) || {(time - timein) <= limit_time};
```

3.7. *Exemples de comportements attendus*

Si nous voulons qu'un comportement soit interrompu pour répondre à un message, nous pouvons utiliser la forme suivante :

```
  - (state = 0) <- ... // current_behaviour
```

Par défaut, la priorité d'un comportement est 0, le comportement par défaut défini dans le prototype AGENT sera prioritaire (car déclaré de priorité 0, mais avant notre comportement). Notre comportement reprendra automatiquement son exécution après le traitement du message (au passage de `has_new_message` = FALSE).

Si nous ne désirons pas interrompre un comportement par un message, nous pouvons utiliser la forme suivante :

```
- (state = 0) Priority 1 <- ... // current_behaviour
```

La priorité est supérieur au traitement d'un message.

Si nous voulons qu'un comportement de priorité supérieur à 0 soit néanmoins interrompu pour répondre à un message, nous pouvons utiliser la forme suivante :

```
- ((state = 0) & (! has_new_message)) Priority 1 <-
( ... /* Current behaviour */ )
Postface { pop_message; };
```

Un système multi-agents doit être réactif, et se conceptualise en temps réel. Le programmeur doit avoir la possibilité de conditionner l'exécution d'un comportement avec une limite de temps. Pour limiter l'exécution d'un comportement A à 30 millisecondes pour ensuite passer à un comportement B, nous pouvons utiliser la forme suivante :

```
- ((state = 0) & (timeout 30)) <-
Preface { timein := time; }
( ... /* Current behaviour A */ )
Postface { state  := 1; timein := 0; };

- (state = 1) <- ... // Code of behaviour B
```

Si nous voulons changer de comportement de manière synchrone à un nouvelle état d'un autre agent :

```
- ((state = 0) & (other_agent.state != 1)) <-
( ... /* Current behaviour */ )
Postface { state := 1; };

- (state = 1) <- ... // New behaviour
```

3.7.1. *Extraction de données*

Les systèmes multi agents supposent une communication accrue, en particulier les agents complexes et cognitifs. Une modélisation complexe nécessite une bonne connaissance de l'environnement extérieur, en particulier des connaissances complexes sur les propriétés d'agent et objets extérieur. Dans les langages classiques, le programmeur est obligé de parcourir inlassablement des collections pour y trouver les données qu'il cherche. Dans un langage agent, c'est la communication qui pallie ce manque.

Nous avons cherché à définir un mécanisme utilisable par les agents et par les objets. C'est pourquoi le modèle *Lisaac agent* intègre un mécanisme d'extraction des données inspiré entre autre de SQL, il s'agit d'envoyer des requêtes permettant de

filtrer des données à partir d'une ou plusieurs collections d'objets, ainsi que de déterminer une liste d'objets/agents répondant à certains critères.

Dans le cadre de cet article court, nous ne pouvons pas développer ces aspects de notre modèle. Le principe étant très proche des mécanismes décrits dans (Seriai, 2003), nous avons préféré mettre l'accent sur précédents points de notre modèle.

4. Conclusion

Le modèle *Lisaac Agent* se veut une synthèse pragmatique entre un langage à objet à prototype classique et un langage permettant d' implémenter des systèmes multi-agentsréactifs. Nous proposons une extension d'un langage de haut niveau qui se veut simple, intuitive tout en restant puissant et polyvalent.

Concevoir des agent cognitifs avec notre approche est beaucoup plus difficile, cette extension se positionne clairement dans le domaine d'agent réactifs. Nous le percevons comme le prolongement naturel de l'approche objet. Néanmoins, le programmeur pourra développer des systèmes multi-agents capables de manifester des comportements cognitifs grâce à l'émergence de comportement intelligents au sein de SMA réactifs. Cela dit, la possibilité d'envoyer des messages en multicast permettront au programmeur de concevoir des systèmes multi-agents faisant montre de fonctionnalités s'approchant de systèmes multi-agents cognitifs.

Une première implentation naïve de notre modèle est fonctionnelle. Si elle ne permet pas de réaliser des mesures d'efficacité sérieuses en terme de performance d'exécution, elle a le mérite de valider la faisabilité et de tester la pertinence de notre approche décrite dans cet article.

5. Bibliographie

G. W., « VIVA knowledge-based agent programming », april, 1996.

Goldberg A., *Smalltalk-80, The Interactive Programming Environment*, Addisson-Wesley, Reading, Massachusetts, 1984.

Goldberg A., Robson D., *Smalltalk-80, the Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

Guessoum Z., Briot J.-P., Dojat M., « Des objets concurrents aux agents autonomes », *Cinquièmes Journées Francophones sur l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents (JFIADSMA'97)*, vol. 33, number 5, Hermès Science Publications, p. 93-107, april, 1997.

Livshitz V., « `http ://java.sun.com/developer/technicalArticles/Interviews/livschitz_qa.html` », , The Next Move in Programming : A Conversation with Sun's Victoria Livschitz, February, 2004.

Meyer B., *Eiffel, The Language*, Prentice Hall, Englewood Cliffs, 1992. ISBN 0-13-247925-7.

Seriai A., « QUEROM : An Object-Oriented Model to rewriting query using views », *International Conference on Enterprise Information Systems*, 2003.

Sonntag B., « Le projet Isaac : une alternative objet de haut niveau pour la programmation système », *4ième Conférence Française sur les systèmes d'Exploitation, (CFSE'4)*, ACM Press, Mars, 2005.

Sonntag B., Colnet D., Boutet J., Lisaac - Programmer's Reference Manual, Technical report, LORIA, 2003. LORIA.

Ungar D. M., Smith R. B., « Self : The Power of Simplicity », *2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, ACM Press, p. 227-241, 1987.

Weerasooriya, Rao, Ramamohanarao, « Design of a concurrent agent-oriented language », *Intelligent Agents : Theories, Architectures, and Languages , no. 890 in Lecture Notes in Artificial Intelligence*, Springer-Verlag, p. 386-402, april, 1995.

Assemblage Automatique de Composants pour la Construction d'Agents avec MADCAR

G. Grondin[*,] — N. Bouraqadi[*] — L. Vercouter[**]**

[*] *Dépt. GIP, Ecole des Mines de Douai*
941, rue Charles Bourseul – B.P. 10838, 59508 Douai Cedex, France
{grondin,bouraqadi}@ensm-douai.fr

[**] *Dépt. G2I, Ecole des Mines de Saint-Étienne*
158 cours Fauriel, 42023 Saint-Étienne cedex 02, France
vercouter@emse.fr

RÉSUMÉ. *Dans cet article, nous proposons d'utiliser le modèle* MADCAR *pour faciliter la conception d'agents à base de composants.* MADCAR *est un modèle abstrait pour construire des moteurs d'assemblage dynamique et automatique d'applications à base de composants. Ici, l'application à modéliser est un agent capable de se ré-assembler à la suite d'un changement contextuel ou à la demande du concepteur. Dans cet agent, nous séparons son comportement normal (matérialisé par un assemblage de composants) de son comportement d'adaptation. Pour ce faire, nous intégrons à cet agent un moteur permettant de déclencher, décider et réaliser des ré-assemblages de composants. Le processus d'assemblage est piloté par une politique d'assemblage explicite, précise et ouverte, qui doit être spécifiée par le concepteur de l'agent.*

ABSTRACT. *In this article, we suggest using the* MADCAR *model to facilitate the design of component-based agents.* MADCAR *is an abstract model for building engines that dynamically and automatically (re-)assemble component-based applications. We use such an engine to define a model of agent able to be re-assembled following contextual changes or the designer's requests. One of the main advantages of* MADCAR *lies in the specification of the assemblies with both flexibility and precision. The assembling process is controlled by an explicit policy of assembling. Last, by using* MADCAR*, our agent model aims at being open so as to be usable in various application domains.*

MOTS-CLÉS : *Conception d'agents, Adaptation d'agents, Assemblage automatique.*

KEYWORDS: *Agent design, Agent adaptation, Automatic assembling.*

1. Introduction

Dans les travaux sur la conception d'agents, l'approche à base de composants est avant tout utilisée comme un cadre pour la spécification du comportement de l'agent. En effet, les composants permettent d'implémenter des capacités qui sont souvent requises chez les agents (communication, perception, planification,...). Mais, trop souvent, l'assemblage des composants doit se faire manuellement et les possibilités d'évolution de cet assemblage ne sont pas prévues.

Un modèle d'agents à base de composants et des mécanismes de ré-assemblage doivent être définis pour permettre la conception d'agents susceptibles d'évoluer régulièrement. Trois propriétés sont notamment recherchées pour ce mécanisme : l'ouverture, l'adaptabilité et l'autonomie. L'**ouverture** fait référence à la possibilité d'ajouter des composants même s'il n'était pas initialement prévu de les ajouter (i.e. après la phase de conception). L'**adaptabilité** fait référence à la capacité de modifier le fonctionnement d'un agent pour prendre en compte certains changements notables dans son contexte. Enfin, la propriété d'**autonomie** fait référence à la faculté des agents à fonctionner sans intervention humaine, en particulier en ce qui concerne les ré-assemblages.

Dans cet article, nous proposons de construire des agents en utilisant le modèle MADCAR[1]. MADCAR est un modèle abstrait qui a pour objectif la reconfiguration dynamique et automatique d'applications à base de composants. Nous voulons utiliser MADCAR pour automatiser la construction d'agents à base de composants grâce à ses mécanismes de (ré-)assemblage. C'est pourquoi, nous proposons un modèle d'agents où le comportement d'adaptation des agents est séparé de leur comportement normal (implémenté par un assemblage de composants). Nous intégrons à chaque agent un moteur conçu avec MADCAR, permettant de déclencher, décider et réaliser des ré-assemblages de composants. Le processus d'assemblage de ce moteur est piloté par une politique d'assemblage, faiblement couplée aux composants de l'agent, et qui doit être spécifiée par le concepteur de l'agent. En résumé, nous proposons un modèle d'agents auto-adaptables, et dont le comportement d'adaptation peut être spécifié de façon à la fois flexible et précise.

La section 2 décrit les besoins pour l'aide à la conception d'agents en se basant sur un environnement de développement existant : MAST. La section 3 présente de manière succincte le modèle MADCAR, et son utilisation pour l'assemblage automatique d'agents. La section 4 illustre le processus de ré-assemblage de MADCAR dans un exemple visant à adapter un agent. Enfin, la section 5 récapitule les principaux apports de MADCAR pour l'aide à la conception d'agents.

1. M*odel for* A*utomatic and* D*ynamic* C*omponent* A*ssembly* R*econfiguration.*

2. Le point sur la construction d'agents par assemblage de composants

Des travaux relativement récents se sont intéressés à la construction d'agents par assemblage de composants. La programmation par composants est en effet intéressante pour ce type de développement car les agents présentent souvent des capacités proches d'une application à une autre (communication, perception, planification,...). Dans ce cas, la construction d'un agent consiste à *assembler des composants pré-existants qui sont chacun chargé d'implémenter une partie bien spécifique du comportement de l'agent*. En d'autres termes, un agent est une entité logicielle contenant un assemblage de composants qui matérialise son comportement[2]. Le développement d'applications multi-agents bénéficierait grandement de l'utilisation d'une bibliothèque de composants implantant ces capacités partagées. Plusieurs environnements de développement ont ainsi adopté une approche par composants (Brazier *et al.*, 2002; Occello *et al.*, 2002; Ricordel *et al.*, 2002; Vercouter, 2004).

Nous allons mettre en évidence des possibilités d'amélioration de l'approche par composants pour concevoir des agents en discutant des avantages et des limites d'un de ces travaux. L'environnement de développement MAST (*Multi-Agent System Toolkit*) intègre un modèle de composant spécifique permettant la modification automatique d'un assemblage (Vercouter, 2004).

Un des principaux avantages de MAST est de permettre la connexion automatique et flexible des composants qui constituent un agent. En effet, les composants d'un agent ne sont pas directement connectés entre eux mais à un noyau commun qui joue un rôle de bus logiciel pour assurer la communication entre les composants. Le noyau reçoit des événements émis par un composant et se base sur une description sémantique de chaque composant pour déterminer le ou les composant(s) à qui il transmet l'événement. Cette flexibilité du mode d'interaction facilite le rôle du concepteur de l'agent lorsque des composants sont ajoutés ou supprimés, car il n'est pas nécessaire de connecter explicitement les entrées/sorties des composants. Ces mécanismes montrent l'importance de l'**automatisation** du processus d'assemblage.

Cependant, ces spécificités qui facilitent la conception d'agents dans MAST ne sont pas sans poser un certain nombre d'inconvénients. En effet, les difficultés qui sont épargnées au concepteur de l'agent se trouvent reportées chez le concepteur des composants. Ce dernier doit fournir aux composants une riche description de leurs capacités d'assemblage notamment sur la base des événements qu'ils prennent en charge. Une partie des informations d'assemblage est ainsi déportée dans la description du composant, ce qui dénote un manque d'**abstraction** alors que les mécanismes d'assemblage doivent pouvoir être spécifiés sans que cela affecte directement les composants.

Par ailleurs, il est important de souligner le manque de contrôle de l'assemblage de composants dans MAST. Les connexions entre composants n'étant pas concrètement construites mais déduites des interfaces des composants, il n'est pas évident de prévoir

2. Remarquons qu'un agent contenant des composants ne doit pas forcément être un composant.

l'impact de chaque modification de l'assemblage. Ainsi, la flexibilité de l'assemblage ne doit pas se faire au détriment du fonctionnement de l'agent. La problématique d'assemblage est transversale aux composants de l'agent car elle peut concerner l'ensemble des composants. Cela montre que le processus d'assemblage doit être géré de manière **explicite** afin de pouvoir spécifier les conditions d'assemblage avec précision, quel que soit le sous-ensemble de composants concernés.

3. Construction et adaptation d'agents avec MADCAR

MADCAR est un modèle abstrait de moteurs d'assemblage dynamique et automatique d'applications à base de composants. Nous montrons dans cette section comment il peut être exploité pour la construction et l'adaptation d'agents. Les seules hypothèses que nous faisons sur les *modèles de composants utilisables dans* MADCAR sont la spécification explicite, pour chaque composant, de leurs interfaces fournies et requises ainsi que de leurs attributs [3].

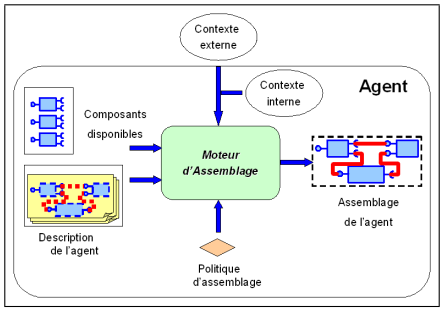


Figure 1. *Structure d'un agent avec* MADCAR

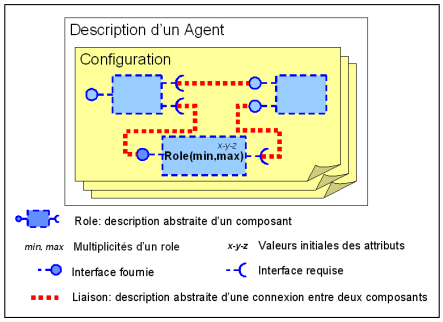


Figure 2. *Description d'un agent avec* MADCAR

3.1. *Structure d'un agent avec* MADCAR

Un moteur d'assemblage basé sur MADCAR permet de construire ou d'adapter un logiciel à partir de quatre entrées : un *ensemble de composants* à assembler, une *description du logiciel* qui représente la spécification de l'assemblage, une *politique d'assemblage* qui dirige les décisions d'assemblage et un *contexte* dont les changements déclenchent le ré-assemblage. La figure 1 représente un tel moteur avec ses différentes entrées/sortie dans le cas où le logiciel à (ré-)assembler est un agent. Un tel agent est constitué du moteur d'assemblage, de ses différentes entrées, ainsi que de l'assemblage résultat. Notez que le contexte est scindé en deux parties : l'une est in-

3. Le mariage entre notre modèle d'agents et le concept de « composant composite » est en cours d'étude.

terne et représente l'état de l'agent et l'autre est externe et représente l'état du monde dans lequel évolue l'agent.

Le choix de placer le moteur d'assemblage et ses différentes entrées à l'intérieur de l'agent est motivé par la préservation de la propriété d'autonomie. En effet, l'agent reste « maître » de sa propre structure et de ses évolutions.

3.2. *Description d'un agent avec* MADCAR

La figure 2 représente une description d'un agent avec MADCAR. Cette description regroupe un ensemble de **configurations** alternatives. Une configuration représente un assemblage de rôles. Un **rôle** est une description abstraite de composants, constituée d'un ensemble de *contrats* (Meyer, 1992; Beugnard *et al.*, 1999). Ces contrats doivent au moins spécifier (1) un ensemble d'interfaces (fournies ou requises) symbolisant ses éventuelles interactions avec les autres rôles, (2) un ensemble d'attributs dont les valeurs permettent d'initialiser les composants, et (3) deux multiplicités. Les *multiplicités* d'un rôle représentent le nombre minimal (min où $min \geq 0$) et le nombre maximal (max où $max \geq min$) de composants qui peuvent remplir ce rôle simultanément.

Les configurations de MADCAR diffèrent du concept de configuration utilisé dans les ADLs[4] (Fuxman, 2000; Medvidovic *et al.*, 1997). En effet, chacune de nos configurations décrit un *ensemble d'assemblages possibles* structurellement équivalents puisqu'ils sont basés sur le même graphe de rôles. Le concept de rôle dans MADCAR permet le découplage total entre l'architecture de l'agent (décrite par la configuration en cours) et les composants qui la composent. De plus, la multiplicité des rôles offre la possibilité d'avoir un nombre plus ou moins grand de composants utilisés dans des assemblages issus de la même configuration.

3.3. *Processus de (re-)assemblage de* MADCAR

Un moteur basé sur MADCAR peut aussi bien être utilisé pour créer automatiquement un assemblage à partir de composants déconnectés, que pour adapter dynamiquement un assemblage existant. En effet, le même processus est réalisé dans les deux cas. Ce processus se décompose en cinq étapes successives.

1) Le (ré-)assemblage est déclenché lorsque des événements pertinents sont détectés. C'est le cas par exemple lors de l'ajout de configurations et/ou de composants.

2) Le moteur identifie les compatibilités entre les composants présents et les rôles contenus dans les configurations fournies : il s'agit de déterminer pour chaque composant s'il peut satisfaire les contrats définis dans les rôles.

4. *Architecture Description Languages.*

3) Le moteur sélectionne une configuration selon la politique d'assemblage et les compatibilités identifiées dans l'étape précédente.

4) Le moteur sélectionne les composants à connecter selon la configuration choisie et sur la base de la politique d'assemblage.

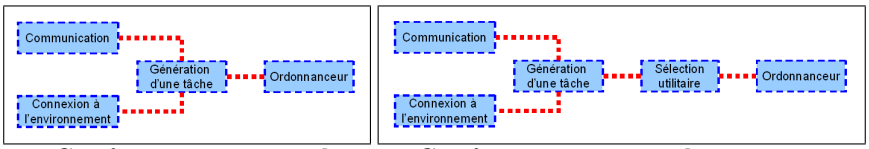
5) Enfin, le moteur réalise l'assemblage des composants sélectionnés selon la configuration choisie.

4. Exemple de ré-assemblage d'agent

Dans cette section, nous illustrons le processus d'assemblage décrit dans la section 3.3 par un scénario de ré-assemblage d'agents. Ce scénario est basé sur une application inspirée de celle décrite dans (Bourne *et al.*, 2000) et déjà utilisée dans (Vercouter, 2004) pour la conception d'agents par assemblage de composants.

4.1. *Cadre du scénario*

L'application est constituée d'une grille sur laquelle sont positionnés plusieurs agents qui accomplissent des tâches qui sont réparties sur la grille. Chaque tâche est caractérisée par une durée d'exécution, une échéance et une récompense. Certaines tâches, dites coopératives, ne peuvent être exécutées que lorsqu'un nombre d'agents suffisant pour la réaliser sont présents sur sa case. Dans ce cas, ces agents se partagent équitablement la récompense. Les agents qui souhaitent exécuter une tâche coopérative peuvent communiquer avec d'autres agents pour les inviter à les rejoindre.



$Config_1$: « Agent Social » $Config_2$: « Agent Social Autonome »

Figure 3. *Description d'un agent comportant deux configurations*

Dans ce scénario, nous voulons concevoir un agent qui a pour objectif de maximiser la somme des récompenses qu'il obtient en exécutant les tâches. Considérons l'agent qui est décrit par les *deux configurations* illustrées sur la figure 3. Chaque configuration représente un comportement possible de l'agent. En l'occurrence, ces deux configurations traduisent deux stratégies pour obtenir des récompenses. La configuration **Agent Social** désigne un agent qui interagit librement avec les autres agents et accepte toutes leurs demandes tant qu'il peut les assumer. Dans cette configuration, le rôle `Communication` permet à l'agent de communiquer avec les autres agents. Le rôle `Connexion à l'environnement` permet à l'agent d'agir sur son environnement et de percevoir les tâches et les autres agents. Le rôle `Génération d'une tâche` permet à l'agent de réifier une tâche à partir des messages ou des percepts

qu'il reçoit. Cette tâche est envoyée vers l'ordonnanceur. Le rôle `Ordonnanceur` gère les demandes d'ajout de tâches : la tâche est acceptée si et seulement si l'agent peut la réaliser avant son échéance sans remettre en cause les autres tâches. Par ailleurs la configuration **Agent Social Autonome** désigne un agent qui choisit les demandes à accepter selon son intérêt propre. Ce choix peut amener l'agent à ne pas accomplir une tâche qu'il s'était précédemment engagé à effectuer, au profit d'une nouvelle tâche qui améliore son bénéfice. Dans la configuration, cette différence de comportement se traduit par le rôle *Sélection Utilitaire* qui reçoit les demandes d'ajout de tâches et modifie l'ordonnanceur pour maximiser les bénéfices de l'agent.

Nous supposons qu'il y a un mécanisme qui calcule régulièrement le taux de satisfaction de l'agent en fonction des récompenses qu'il a obtenues jusqu'à présent. Par hypothèse, la valeur de ce taux de satisfaction fait partie de l'*état de l'agent* et va être utilisée pour déclencher les ré-assemblages. Ainsi, nous allons considérer cinq paliers de satisfaction : 0%, 20%, 40%, 60% et 80%. Dans cet exemple illustratif, nous définissons une *politique d'assemblage* assez simpliste qui se résume à changer alternativement de configuration lorsque le taux de satisfaction de l'agent passe au palier inférieur. Même si la configuration « Agent Social Autonome » est a priori plus rentable que la configuration « Agent Social » en terme de récompense, l'agent a intérêt à ne pas être constamment « Agent Social Autonome ». En effet, pour maximiser ses gains, un agent social autonome peut rompre ses engagements vis-à-vis des autres agents, en abandonnant une tâche coopérative avant qu'elle ne soit réalisée, au profit d'une tâche plus rentable. C'est la raison pour laquelle les autres agents peuvent devenir méfiants envers lui et ne plus lui proposer de tâches coopératives s'il les abandonne trop souvent. Par contre, un agent social ne rompt jamais ses engagements. Il gagne facilement la confiance des autre agents, mais ses gains sont aléatoires. C'est pourquoi il est utile de changer de configuration de temps en temps.

4.2. *Ré-assemblage de l'agent*

Ce scénario décrit l'adaptation automatique d'un agent conçu avec MADCAR, i.e un agent contenant un moteur d'assemblage. Plus particulièrement, nous allons détailler les étapes d'un ré-assemblage d'agent *avec changement de configuration*[5]. Nous supposons que la configuration courante de notre agent est « Agent Social Autonome ». De plus, l'agent contient un ensemble de composants qui sont assemblés conformément à cette configuration : *messenger*, *perceptor*, *task manager*, *scheduler* et *maximizer*.

5. Par opposition à un ré-assemblage *sans changement de configuration* comme lors d'un remplacement du composant de communication de l'agent, afin de se conformer à un nouveau protocole de communication.

4.2.1. *Déclenchement du ré-assemblage*

L'agent perçoit que son taux de satisfaction vient de passer en dessous du palier de 60%. Donc, le processus d'assemblage est déclenché automatiquement conformément à la politique d'assemblage.

4.2.2. *Identification des compatibilités*

Le moteur d'assemblage de l'agent compare deux-à-deux les composants présents dans l'agent avec les rôles des deux configurations fournies. Il est résulte la matrice de compatibilités du tableau 1. Cette matrice est simpliste pour cet exemple mais MADCAR considère qu'il pourrait exister plusieurs composants pouvant jouer un même rôle. De même, il est possible qu'un composant soit compatible avec plusieurs rôles.

	Roles / Components	messenger	perceptor	task manager	scheduler	maximizer
Config 1	Communication (1,1)	X				
	Connexion à l'environnement (1,1)		X			
	Génération de tâche (1,1)			X		
	Ordonnanceur (1,1)				X	
Config 2	Communication (1,1)	X				
	Connexion à l'environnement (1,1)		X			
	Génération de tâche (1,1)			X		
	Ordonnanceur (1,1)				X	
	Sélection utilitaire (1,1)					X

Tableau 1. *Exemple de matrice de compatibilités entre des rôles et des composants*

4.2.3. *Sélection d'une configuration*

D'après la matrice de compatibilités, chacun des rôles spécifiés peut être rempli par un composant différent. En d'autres termes, les deux configurations de l'agent sont éligibles. Le moteur d'assemblage de l'agent doit choisir l'une d'entre elles en fonction de la politique d'assemblage. Celle-ci stipule que l'agent doit forcément changer de configuration. C'est donc la configuration « Agent Social » qui est choisie.

4.2.4. *Sélection d'un ensemble de composants*

La politique d'assemblage de cet agent ne spécifie aucune contrainte sur la façon de choisir les composants. Donc, n'importe quel ensemble de composants capable de remplir la configuration choisie est valide. Les composants sélectionnés sont : *messenger, perceptor, task manager* et *scheduler.*

4.2.5. *Assemblage d'un ensemble de composants*

Les composants sélectionnés sont finalement assemblés selon la configuration « Agent Social ». Parmi les mécanismes qui sont mis en œuvre pendant cette étape de réalisation de l'assemblage, nous pouvons citer la sauvegarde et la restauration de l'état des composants à remplacer, ainsi que la connexion des composants entre eux selon ce qui est décrit dans la nouvelle configuration. Notamment, l'ensemble des

tâches pour lesquelles l'agent s'est engagé doivent être gardées en mémoire lorsque le composant qui remplit le rôle `Ordonnanceur` doit être remplacé. Dans cet exemple, la plupart des composants du précédent assemblage sont conservés sans être réinitialisés. Il reste juste à connecter directement le composant *task manager* au composant *scheduler* : le composant *maximizer* ne sera pas utilisé jusqu'au prochain ré-assemblage.

Dans ce scénario, nous n'avons pas expliqué l'ensemble des mécanismes permettant de garantir la cohérence de l'agent pendant la réalisation de l'assemblage, notamment lorsque les ré-assemblages doivent être procédés dynamiquement. Ces mécanismes doivent faire l'objet d'une étude plus approfondie.

5. Conclusion et perspectives

Dans cet article, nous avons proposé d'utiliser le modèle MADCAR pour réaliser des agents *ouverts*, *adaptables* et *autonomes*. MADCAR est un modèle de *moteurs d'assemblage automatique et dynamique de composants*. Notre proposition consiste à intégrer à chaque agent un moteur d'assemblage basé sur MADCAR, une politique d'assemblage ainsi que l'ensemble des composants à assembler et la description de l'agent. La description de l'agent correspond à un ensemble de configurations valides. Chaque configuration est un assemblage de rôles, où un rôle est une description abstraite de composants constituée d'un ensemble de *contrats*.

Une des principales caractéristiques de MADCAR est l'utilisation des mêmes mécanismes d'assemblage que ce soit pour construire un agent à base de composants ou pour l'adapter. L'adaptation d'un agent consiste à ré-assembler les composants de l'agent, en particulier après un changement de configuration. Notons que notre modèle peut prendre en charge des adaptations imprévues lors de la conception de l'agent puisque l'ensemble des configurations, l'ensemble de composants et la politique d'assemblage peuvent être changés dynamiquement. C'est pourquoi les agents conçus avec MADCAR sont ouverts. L'autonomie de l'agent est basée sur d'une part, un moteur d'assemblage général et d'autre part, une politique d'assemblage particulière. La politique d'assemblage permet de piloter le processus d'assemblage mis en œuvre par le moteur, depuis le déclenchement d'un assemblage jusqu'à sa réalisation. Par ailleurs, la principale tâche du concepteur d'agent est de spécifier des configurations et une politique d'assemblage pour son agent. Les concepts de rôle et de configuration que nous avons définis permettent de spécifier des assemblages en gardant un couplage faible avec les composants de l'agent. Cela présente un avantage pour la réutilisabilité des configurations. Par exemple, un agent pourrait récupérer une configuration depuis un autre agent et l'utiliser à son compte lorsqu'il contient un ensemble de composants qui peut remplir cette configuration.

En perspective, nous comptons fournir un formalisme de haut-niveau pour exprimer facilement des configurations et des politiques d'assemblages dans MADCAR, et aussi pour modéliser le contexte d'une application. Ce formalisme doit nous permettre de spécifier le comportement d'agents auto-adaptables. Actuellement, nous travaillons

sur une projection de MADCAR sur le modèle de composants Fractal (Bruneton *et al.*, 2002). Nous avons choisi de travailler avec ce modèle de composants hiérarchiques car nous envisageons d'étendre l'utilisation de MADCAR au ré-assemblage de composants composites. Enfin, nous devons étudier de manière spécifique la prise en compte de la dynamicité lors de l'étape de réalisation du processus d'assemblage défini dans MADCAR. En effet, les adaptations dynamiques d'applications posent des problèmes bien connus mais difficiles à résoudre de manière automatique (Segal *et al.*, 1993; Hicks, 2001).

6. Bibliographie

Beugnard A., Jezequel J.-M., Plouzeau N., Watkins D., « Making Components Contract Aware », *Computer*, vol. 32, n° 7, p. 38-45, 1999.

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, p. 348-352, 2000.

Brazier F. M. T., Jonker C. M., Treur J., « Principles of component-based design of intelligent agents », *Data Knowl. Eng.*, vol. 41, n° 1, p. 1-27, 2002.

Bruneton E., Coupaye T., Stefani J., « Recursive and dynamic software composition with sharing », *WCOP'02–Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming*, Malaga, Spain, Jun, 2002.

Fuxman A. D., A Survey of Architecture Description Languages, Technical Report n° CSRG-407, Department of Computer Science, University of Toronto, Canada, 2000.

Hicks M., Dynamic Software Updating, PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August, 2001.

Medvidovic N., Taylor R. N., « A framework for classifying and comparing architecture description languages », *ESEC '97/FSE-5 : Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, Springer-Verlag New York, Inc., New York, NY, USA, p. 60-76, 1997.

Meyer B., « Applying "Design by Contract" », *Computer*, vol. 25, n° 10, p. 40-51, 1992.

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.

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.

Segal M. E., Frieder O., « On-the-Fly Program Modification : Systems for Dynamic Updating », *IEEE Softw.*, vol. 10, n° 2, p. 53-65, 1993.

Vercouter L., « MAST : Un modèle de composants pour la conception de SMA », *Journées Multi-Agents et Composants, JMAC 2004*, Paris, France, 23-23 novembre, 2004.

Vers un modèle d'agent flexible

Sébastien Leriche — Jean-Paul Arcangeli

IRIT-UPS, Université Paul Sabatier
118 Route de Narbonne, F-31062 TOULOUSE CEDEX 9
{leriche, arcangel}@irit.fr

RÉSUMÉ. *Dans le cadre de nos travaux sur l'aide au développement d'applications réparties ouvertes, nous avons proposé un modèle d'agent mobile auto-adaptable ainsi que son implémentation au sein d'un middleware Java. Nous étudions actuellement une évolution de ce modèle afin de lui donner davantage de flexibilité. Ce papier présente le travail en cours. Nous discutons le besoin de flexibilité et la possibilité de définir différents modèles d'agents par le biais d'un méta-niveau configurable. Puis, nous présentons les éléments d'une architecture à base de micro-composants permettant l'adaptation dynamique.*

MOTS-CLÉS : *agents, composants, architecture logicielle, flexibilité, ouverture*

1. Introduction

Les évolutions du matériel conduisent de plus en plus à l'exécution d'applications dans des contextes répartis et hétérogènes (en termes de logiciel et de matériel -machines et réseau-) : par exemple sur l'Internet ou sur des grilles qui donnent accès à des quantités importantes de ressources, ou encore sur des équipements mobiles ou enfouis (PDA, PC portables, calculateurs embarqués…). Les ressources et les services y sont plus ou moins volatiles ; les conditions d'exécution varient (bande passante, disponibilité et localité des ressources) et sont partiellement ou mal connues au moment du développement. Aussi, pour offrir des services robustes et performants, ou pour en améliorer la qualité, les applications doivent pouvoir s'adapter dynamiquement. Ce besoin doit être pris en compte dès la conception et par conséquent il est nécessaire de disposer de technologies adéquates.

Le concept d'agent (voir par exemple la définition de J. Ferber (Ferber, 1995)) est particulièrement intéressant pour son niveau d'abstraction. Nous appelons *agent*

logiciel une entité autonome capable de communiquer, disposant de connaissances et d'un comportement privé, ainsi que d'une capacité d'exécution propre. Un agent (logiciel) agit pour le compte d'un tiers (un autre agent, un utilisateur) qu'il représente sans être obligatoirement connecté à lui. Comme les composants logiciels (Szyperski, 2002; Oussalah, 2005), les agents logiciels sont issus du concept d'objet et constituent des briques de base pour la construction des applications. Un *agent mobile* (Fuggetta *et al.*, 1998; Bernard *et al.*, 2002) est un agent logiciel qui peut se déplacer sur un réseau avec son code et ses données propres, mais aussi avec son état d'exécution. Le but peut être de localiser une ressource ou de rapprocher un traitement des données pour réduire la consommation de bande passante ou encore d'adapter un service distant aux besoins et aux capacités du client.

L'utilisation d'agents, éventuellement mobiles, contribue à la flexibilité des applications. Nous avons étudié et proposé des technologies de niveau modèle de programmation et *middleware* pour aider au développement d'applications réparties auto-adaptables : au niveau des agents, l'auto-adaptation repose sur la mobilité proactive ainsi que sur l'évolution fonctionnelle et opératoire. Dans cette communication, nous présentons un travail en cours dont l'objectif est d'augmenter le niveau de flexibilité intra-agent : nous cherchons à définir un modèle et une architecture d'agent flexible (ceci à partir d'un modèle d'agent que nous avons précédemment défini et qui est décrit en section 2.1).

En section 2, nous discutons les besoins en termes d'évolution d'un modèle existant et de définition de différents modèles d'agents. En section 3, nous décrivons les principes architecturaux et nous proposons quelques élements pour la mise en oeuvre. En section 4, nous discutons les limites en termes de flexibilité intra-agent et nous situons notre approche par rapport d'autres (principalement MAST de L. Vercouter). Enfin, nous concluons en section 5.

2. Evaluation des besoins

2.1. *Un premier modèle d'agent adaptable*

Les agents mobiles sont, par nature, des outils d'adaptation en contexte réparti. Afin d'accroître les possibilités d'adaptation, nous avons proposé un niveau de flexibilité intra-agent contrôlé par l'agent lui-même et nous avons défini un modèle d'agent mobile auto-adaptable dynamiquement (Leriche *et al.*, 2004a) : conformément au principe de séparation des préoccupations, le modèle repose sur la réification d'un ensemble de mécanismes d'exécution internes non fonctionnels (envoi des messages, réception, déplacement, cycle de vie…) sous la forme de micro-composants dynamiquement interchangeables. Ces micro-composants constituent un méta-niveau (cf. 3.1.1) de l'agent ; ils opèrent par délégation de son code fonctionnel (niveau de base). Ce modèle d'agent a été mis en œuvre dans le *middleware* JAVACT [1] que nous dé-

1. http://www.irit.fr/free.html

veloppons. Il a été validé expérimentalement dans le cadre de plusieurs applications : prototypes de logiciels pair à pair pour le partage et l'échange de fichiers (Leriche *et al.*, 2004b), distribution de correctifs de sécurité, messagerie instantanée mobile, jointures en base de données réparties... Différents micro-composants ont été développés et réutilisés : cryptage des communications, tolérance aux déconnexions[2], localisation d'agents mobiles...

La séparation des préoccupations en différents modules de code indépendants est une propriété essentielle qui doit être préservée : elle simplifie le développement des différents composants (indépendamment les uns des autres) et leur réutilisation.

2.2. *Vers davantage de flexibilité*

Prenons l'exemple de notre prototype de logiciel pair à pair cité précédemment. Il est implanté par un système d'agents répartis. Certains agents sont des agents d'interface (avec les pairs clients ou serveurs). Certains agents sont mobiles sur le réseau et peuvent être amenés à percevoir leur environnement physique (charge du réseau, niveau de sécurité...) ; pour s'y adapter, ils peuvent redéfinir leurs mécanismes de communication ou de déplacement. Certains agents peuvent développer une algorithmique complexe (recherche de ressource, exploitation d'une ressource...). En termes d'évolution, de décision, de perception, de réactivité et de mobilité, les différents agents qui constituent le système ont des caractéristiques et des besoins variables. Or, dans notre implantation, tous ont du être développés à partir du modèle d'agent unique offert par le *middleware*.

Plus précisément, dans la proposition décrite en 2.1 et mise en œuvre dans le *middleware* JAVACT , l'adaptation se réduit au choix initial des micro-composants et au remplacement d'un micro-composant par un autre qui offre le même service (spécifié par une interface Java). L'évolution des services opératoires est limitée par une interface fixe, et il n'est pas possible de décider de quels micro-composants sont remplaçables et de quels ne le sont pas. L'architecture de l'agent est également figée : les agents sont des *acteurs*[3] (Hewitt, 1977; Agha, 1986) répartis et mobiles et il n'est possible ni d'introduire un nouveau micro-composant ni de retirer un micro-composant inutile (par exemple, le gestionnaire de déplacement pour un agent immobile).

D'autre part, on peut aussi noter que notre expérience du développement du *middleware* JAVACT nous a conduit à réviser plusieurs fois l'architecture des agents et à modifier le *middleware*. Par exemple :

– lors de l'introduction de la mobilité, il a fallu ajouter au méta-niveau un micro-composant dédié (il interagit avec d'autres composants de méta-niveau) et donc modifier l'architecture des agents ;

2. Pour cela, il a fallu développer un composant de perception de l'environnement réseau.
3. L'acteur est un modèle concret d'agent logiciel : communication par messages asynchrones, traitement des messages en série, changement de comportement...

– pour mettre en œuvre un mécanisme de communication synchrone, nous avons dû modifier les interfaces existantes, en l'occurrence celle des micro-composants de communication ;

– nos agents n'ont pu être utilisés tels quels pour implanter des agents AMAS (Capera *et al.*, 2003) et une nouvelle architecture d'agent a été définie (Déjean, 2003).

Si les expérimentations effectuées ont permis de vérifier l'intérêt de l'architecture proposée, celle-ci reste donc limitée en terme de flexibilité : elle n'est pas minimale (si un agent n'effectue pas certaines opérations, par exemple changement de comportement ou déplacement, tous les micro-composants lui sont quand même associés) et ne permet pas de modeler et de personnaliser la structure des agents. Par exemple, il n'est pas possible de définir un agent immobile ou un type quelconque d'agent doté de mécanismes particuliers.

2.3. *Différents types d'agents*

Dans une démarche de classification, on peut être amené à distinguer les agents suivant différentes propriétés (ou composantes) indépendantes, à caractère individuel ou social :

– l'autonomie, déclinée diversement dans différents modèles d'agent et leur implantation, qui induit les notions de cycle de vie et d'activité ;

– le savoir et le savoir-faire (éventuellement offert sous forme de services) ;

– une capacité de décision ;

– l'évolutivité (apprentissage...) ;

– la possibilité d'engendrer de nouveaux agents ;

– la communication ;

– l'interaction avec l'environnement (perception...) dans lequel l'agent est situé (par exemple, un agent mobile est situé géographiquement sur un réseau de machines et interagit avec lui *via* un ensemble des services) ;

– la réactivité[4] ;

– ...

La notion de cycle de vie se retrouve dans tous les modèles sous des formes différentes, et la plupart des agents ont besoin de moyens de communication respectant certains standards (invocations de services distribués, KQML, FIPA-ACL...). Les différentes propriétés d'un agent définissent son type : par exemple, les agents BDI (Rao

4. Un agent est réactif s'il répond de manière opportune aux changements de son environnement issus de stimuli externes. Il n'a pas besoin d'une représentation symbolique élevée de la perception son environnement.

et al., 1995), les agents AMAS, les agents de déploiement (Hall *et al.*, 1997), les agents conteneurs[5], etc.

	Com. asynchrone	Création	Com. synchrone	Mobilité	Perception	Décision	Cycle de vie	...
Acteur	*	*					*	
Agent logiciel	*	*	*				*	
Agent mobile	*	*		*			*	
Agent déploiement	*			*			*	
Agent conteneur	*	*	*	*			*	
Agent réactif		*			*	*	*	
Agent BDI	*	*	*		*	*	*	
Agent AMAS	*	*	*		*	*	*	
...								

Figure 1. *Quelques agents et leurs composantes*

Au sein d'une application, on peut trouver des agents de types différents voire certaines formes d'agents hybrides possédant des composantes issues de différents types. On peut s'interroger sur l'intérêt qui y aurait (pour faciliter le développement) à pouvoir définir des types d'agents « à la carte ». Nous pensons que certains agents pourraient bénéficier de composantes issus d'autres modèles, pour obtenir par exemple des agents AMAS mobiles ou des acteurs sans changement de comportement. La réutilisabilité des composantes est un de nos objectifs.

3. Architecture flexible d'agent logiciel

Nous pensons que la flexibilité de l'architecture doit permettre non seulement le remplacement dynamique de composants, mais aussi la définition de structures d'agent personnalisées (éventuellement l'ajout de nouveaux composants à une structure existante et symétriquement le retrait). Notre idée est de :

5. L'agent mobile adaptable peut être vu comme un support de déploiement au sens de (Carzaniga *et al.*, 1998) pour les comportements (composants fonctionnels) : d'une part la mobilité d'agent permet un transport adaptatif des composants fonctionnels sur le réseau, d'autre part le méta-niveau joue le rôle de conteneur pour le composant fonctionnel : il lui fournit les services nécessaires à son exécution et permet sa configuration (choix des micro-composants) et sa reconfiguration dynamique (Arcangeli *et al.*, 2006).

– permettre la définition de différents modèles (ou types) d'agents sous forme d'un ensemble de mécanismes primitifs (des types de micro-composants) conférés à l'agent ;

– puis de configurer les agents par assemblage de composants sur ce modèle ;

– et enfin de produire une architecture minimale à partir des besoins réels exprimés au niveau de base.

3.1. *Principes architecturaux*

3.1.1. *Réflexivité*

Un système logiciel est dit réflexif lorsqu'il est capable d'inspecter et de modifier sa structure et sa sémantique, de la même manière qu'il opère pour son domaine applicatif. C'est une technique générale pour implanter des mécanismes d'adaptation dynamique (Boyer *et al.*, 2001). En effet, pour faire les bons choix de configuration pour s'adapter au contexte d'exécution, il faut connaître d'abord l'état de l'environnement, mais aussi son propre état et sa structure. On appelle *niveau de base* le niveau habituel de programmation d'une entité, et *méta-niveau* celui qui concerne sa description et dans lequel on réalisera des transformations. Cette séparation entre niveaux permet de rendre transparent au niveau de base un changement structurel ou opératoire dans le méta-niveau.

3.1.2. *Micro-composants*

Pour spécialiser le méta-niveau d'un agent et le doter de nouveaux mécanismes opératoires, une première approche consisterait à procéder par héritage. Mais, dans ce cas, il serait impossible d'adapter dynamiquement les mécanismes internes des agents. L'approche par délégation, bien plus flexible, s'impose. Le paradigme composant (modularité et mécanismes d'assemblage) nous semble un bon candidat pour supporter le codage des différentes composantes des agents identifiées plus haut.

Les agents sont alors construits par assemblage de composants (ce sont donc des composites). Nous avons proposé (Leriche *et al.*, 2004a) de placer les composants fonctionnels au niveau de base, tandis que les composants non-fonctionnels sont regroupés dans le méta-niveau. Ces derniers sont des réifications de mécanismes opératoires spécifiques de l'agent (communication, mobilité...) ; ils sont placés au méta-niveau pour faciliter leur adaptation dynamique via la réflexivité de l'architecture. Pour une identification précise des rôles et donc pour simplifier la validation des assemblages, nous proposons l'utilisation de micro-composants de granularité minimale c'est-à-dire n'offrant qu'un service opératoire unique.

3.1.3. *Architecture composant/connecteur*

Le point central de l'architecture d'agent, le *contrôleur*, est une forme de connecteur. Il permet de faire le lien entre le niveau de base et le méta-niveau ainsi qu'entre les micro-composants du méta-niveau. Il supporte également l'adaptation dynamique

par la modification des liens avec les composants. Pour obtenir la flexibilité souhaitée et que chaque micro-composant possède une interface différente, le contrôleur devrait contenir autant de connexions (ou de ports) que de types de composants possibles. Pour un ensemble donné de types de composants, il faut donc un contrôleur *ad hoc*.

3.1.4. *Minimalité*

Pour répondre au problème précédent, nous proposons de dériver du code fonctionnel de l'agent un type appelé *type réduit* (ou type d'implantation), qui correspond à un ensemble de types de composants effectivement utiles (ou encore aux mécanismes strictement nécessaires à l'exécution). A partir du type réduit identifié, nous proposons de générer automatiquement le contrôleur *ad hoc* associé.

Dans le cas le plus complexe d'un agent qui pourrait changer de comportement fonctionnel, le type réduit pourrait être représenté par un diagramme états/transitions (cf. figure 2) : les transitions correspondraient aux changements de comportement et chaque état représenterait le type réduit pour un comportement.

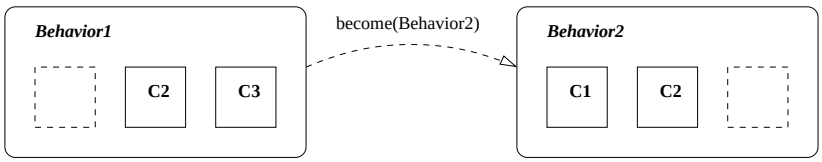


Figure 2. *Type réduit d'agent, représenté par un diagramme état/transition*

Cette minimalité de l'architecture a des avantages en terme d'efficacité à l'exécution (d'autant plus si les agents se déplacent sur le réseau), mais aussi en termes de sûreté et de sécurité car la vérification et la validation des assemblages (dépendances...) est facilitée. Par ailleurs, il n'est pas nécessaire d'expliciter les services de méta-niveau (mécanismes d'exécution) requis par le code fonctionnel puisqu'ils sont directement déduits du code (de même, dans une étape ultérieure, pour les services requis entre micro-composants).

3.2. *Mise en œuvre (en Java)*

Dans un premier temps, pour un *pool* de composants donnés, un contrôleur et une classe abstraite seront engendrés. Dans la classe abstraite, sera implantée une méthode par service de composant ; le corps de cette méthode sera en réalité une indirection vers le contrôleur qui redirigera le traitement vers le composant adéquat. En héritant de la classe abstraite, le code fonctionnel aura accès de manière transparente à tous les services du méta-niveau, sans construction syntaxique particulière.

Après compilation des parties fonctionnelles, un outil en extraira (par rétro-ingénierie) l'ensemble des services non-fonctionnels réellement nécessaires à l'exé-

cution de l'agent, donc par extension le type des micro-composants à instancier dans l'architecture (ou encore le type réduit de l'agent). Il générera ensuite le code du contrôleur minimal *ad hoc*, et pourra effectuer un ensemble de vérifications de cohérence.

Pour réaliser ces étapes, nous utiliserons le framework de manipulation de bytecode Java ASM[6] ainsi que les outils de réflexivité de Java.

4. Discussion

4.1. *Limitations*

Le problème de la « mutation » des agents, *i.e.* le changement dynamique de type, reste ouvert. Nous pensons qu'il serait préférable de ne pas autoriser la mutation si on veut obtenir des systèmes fiables et vérifiables a priori (on pourrait cependant autoriser les mutations prévues lors de la conception). L'adoption dynamique par un agent d'un micro-composant non prévu ne nous semble pas justifiée (un agent pourrait découvrir par lui-même la possibilité de « se bonifier » en changeant ses mécanismes opératoires mais, à notre sens, cela ne pourrait résulter que de capacités d'introspection beaucoup plus élévées que celles que nous envisageons aujourd'hui).

Néanmoins, il est envisageable qu'un agent puisse acquérir dynamiquement un comportement fonctionnel nouveau (dans le cas d'une adaptation fonctionnelle) qu'il aura, par exemple, reçu par message (évolution par changement de comportement dans le cas des acteurs de K. Hewitt et G. Agha). Pour que ce comportement soit effectivement exécutable, il faut que les micro-composants opératoires nécessaires soient disponibles au méta-niveau ; s'ils ne le sont pas, il faut reconfigurer le méta-niveau à partir de nouveaux micro-composants et redéfinir ainsi dynamiquement le type de l'agent.

4.2. *Travaux connexes*

Il existe de très nombreux environnements de développement orientés agents (Jade[7], Actor Foundry[8], MadKit[9], Jack[10]...), mais la majorité est spécifique à une forme particulière d'agent. Par ailleurs, les modèles de composants utilisables aujourd'hui (CCM[11], EJB[12], Fractal[13]...) sont plus adaptés à la conception de gros composants métiers qu'à la conception d'agents à base de composants de petite taille. Nous

6. http://asm.objectweb.org
7. http://jade.tilab.com
8. http://yangtze.cs.uiuc.edu/foundry
9. http://www.madkit.org
10. http://www.agent-software.com/shared/products/index.html
11. http://www.omg.org/technology/documents/formal/components.htm
12. http://java.sun.com/products/ejb
13. http://fractal.objectweb.org

comparons notre approche avec celle de MAST, qui semble très proche de notre proposition.

MAST[14] est un environnement pour le développement et le déploiement d'applications multi-agent. Il fournit des outils pour la conception d'agents par assemblage de composants génériques. MAST propose un modèle de composants spécifique (Vercouter, 2004), avec pour objectif de permettre l'ajout et le retrait de composants sans perturber le fonctionnement de l'agent, et sans intervention extérieure (autonomie de l'agent). Les composants sont assemblés sur un noyau d'agent, qui leur retransmet des évènements auxquels ils sont abonnés. Chaque composant spécifie des ensembles de rôles abstraits fournis et requis, permettant d'en faire un assemblage correct. La méthodologie pour le déploiement des agents est issue de celle des composants répartis : schémas de déploiement spécifié dans un fichier XML et outil de déploiement spécifique.

Notre approche diffère de MAST sur plusieurs points.

– Il est possible d'enlever ou d'ajouter n'importe quel composant à l'exécution dans MAST, sans perturber le fonctionnement des autres composants, mais il n'est pas possible de vérifier si les assemblages sont corrects : par exemple, on peut enlever le composant de communication mais l'agent risque de fonctionner en étant isolé... Nous souhaitons empêcher ce type de faute à l'exécution.

– Les composants de MAST doivent être identifiables dans une méthodologie spécifiques aux SMA (approche *Voyelles*), nous ne souhaitons pas limiter nos composants à un tel ensemble, par exemple nous souhaitons pouvoir modifier le composant de cycle de vie, ou introduire de nouveaux composants qui n'ont rien de spécifique aux SMA (mobilité...).

– Dans MAST, tous les composants de l'agent sont au même niveau, alors que dans notre modèle les composants non-fonctionnels sont placés dans un méta-niveau pour permettre l'introspection et l'intercession sur la structure de l'agent afin de permettre l'adaptation dynamique de l'architecture de l'agent.

– Dans notre proposition, les composants sont des micro-composants, de très petite granularité (service unique), afin de faciliter les validations sur les assemblages.

– L'un des objectifs de notre modèle d'agent est de pouvoir simplifier le développement d'applications réparties. Notre solution permet aux agents de manipuler la répartition et la mobilité, au moyen d'abstractions simples disponibles au niveau de base. Ainsi, il n'est pas nécessaire de prévoir des outils et des schémas de déploiement des agents. De même, les opérations de communication sont réalisées dans des composants de méta-niveau et il suffit d'un appel de méthode au niveau de base pour envoyer un message. La même opération est plus compliquée et moins lisible dans MAST, malgré l'utilisation d'outils.

Pour ces raisons, nous rejoignons les préoccupations de flexibilité de MAST ainsi que certains principes architecturaux (composants...), mais en proposant un modèle

14. http://www.emse.fr/~vercouter/mast

d'architecture différent, plus propice à la conception d'agents logiciels distribués de natures distinctes, plus simple pour le concepteur de l'application à base d'agents et pour lequel il sera possible de vérifier des propriétés de sûreté, notamment à propos de l'assemblage des composants.

5. Conclusion

Dans ce papier, nous présentons un travail en cours dont l'objectif est de simplifier le développement d'applications réparties ouvertes à base d'agents en fournissant un modèle de programmation adapté (basé sur la séparation des préoccupations) ainsi que les outils de niveau *middleware* correspondants. Sur le plan de la méthodologie de développement, notre proposition doit permettre au concepteur de :

– définir un modèle d'agent adapté au besoin applicatif,

– configurer un agent sur ce modèle, *via* la définition ou la réutilisation de composants logiciels (*plug-in*) de grain fin, et valider la composition,

– reconfigurer l'agent en cours d'exécution en changeant les composants.

Dans un travail précédent, nous avons déjà validé un certain nombre d'éléments. Il nous faut maintenant mettre en oeuvre la solution proposée dans sa totalité. Le développement d'un prototype d'agent flexible permettra d'évaluer expérimentalement l'apport au développement ainsi que les limites. En outre, il nous faudra proposer des solutions pour la validation des architectures d'agent, à la fois pour des raisons de sûreté et de sécurité.

6. Bibliographie

Agha G., *Actors : a model of concurrent computation in distributed systems*, M.I.T. Press, Cambridge, Ma., 1986.

Arcangeli J.-P., Leriche S., Pantel M., « Construction et déploiement de systèmes d'information répartis ouverts et adaptables au moyen d'agents mobiles et de composants », *Revue des sciences et technologies de l'information, série L'Objet*, 2006. à paraître.

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, p. 771-796, 2002.

Boyer F., Charra O., « Utilisation de la réflexivité dans les plate-formes adaptables pour applications réparties », *Revue électronique sur les Réseaux et l'Informatique Répartie*, 2001.

Capera D., George J.-P., Gleizes M.-P., Glize P., « The AMAS theory for complex problem solving based on self-organizing cooperative agents », *Twelfth International Workshop on Enabling Technologies : Infrastructure for Collaborative Enterprises*, 2003.

Carzaniga A., Fuggetta A., Hall R. S., van der Hoek A., Heimbigner D., Wolf A. L., A Characterization Framework for Software Deployment Technologies, Technical Report n° CU-CS-857-98, Dept. of Computer Science, University of Colorado, April, 1998.

Déjean T., Développement d'applications multi-agents adaptatifs avec l'API JavAct, Technical report, Institut de Recherche en Informatique de Toulouse, 2003.

Ferber J., *Les systèmes multi-agents : vers une intelligence collective*, Interéditions, 1995.

Fuggetta A., Picco G., Vigna G., « Understanding Code Mobility », *IEEE Transactions on Software Engineering*, vol. 24, n° 5, p. 342-361, 1998.

Hall R., Heimbigner D., van der Hoek A., Wolf A., The Software Dock : A Distributed, Agent-based Software Deployment System, Technical Report n° CU-CS-832-97, Department of Computer Science, University of Colorado, 1997.

Hewitt C. E., « Viewing control structure as patterns of passing messages », *Journal of Artificial Intelligence*, vol. 8, n° 3, p. 323-364, 1977.

Leriche S., Arcangeli J.-P., « Une architecture pour les agents mobiles adaptables », *Actes des Journées Composants JC'2004*, 2004a.

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*, p. 29-43, 2004b.

Oussalah M., *Ingenierie des Composants : Concepts, techniques et outils*, Vuibert Informatique, 2005. Ouvrage collectif.

Rao A. S., Georgeff M. P., BDI agents : From Theory to Practice, Technical Report n° 56, Australian Artificial Intelligence Institute, Melbourne, Australia, 1995.

Szyperski C., *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley / ACM Press, 2002.

Vercouter L., « MAST : Un modèle de composant pour la conception de SMA », *Actes des Journées Systèmes Multi-Agents et Composants*, 2004.