Mise en œuvre d'une approche componentielle pour la conception d'agents

Alexandre Guillemet — Grégory Haïk
Ecole Normale Supérieure de Lyon, {Alexandre.Guillemet, Gregory.Haik}@ens-lyon.fr
Thomas Meurisse
Université Pierre et Marie Curie Paris 6, Thomas.Meurisse@poleia.lip6.fr

Jean-Pierre Briot — Marc Lhuillier
Laboratoire d'Informatique de Paris 6, {Jean-Pierre.Briot, Marc.Lhuillier}@lip6.fr

RÉSUMÉ. *Cet article décrit une approche de conception et de construction d'agents fondée sur la notion de composant. Elle permet une conception ascendante d'un agent par composition de composants (représentant des comportements élémentaires d'agents) ou descendante par décomposition progressive. Après avoir motivé notre travail, nous décrivons les principes de notre approche. Plusieurs exemples de conception d'agents sont présentés dans le domaine de la simulation d'éco-système. Ils font apparaître plusieurs patterns de conception. Nous comparons notre approche à d'autres travaux similaires en matière d'architectures modulaires d'agents avant de conclure sur les travaux futurs.*

MOTS-CLÉS : *Composants, Agents, Conception, Architecture, Simulation.*

ABSTRACT. *This paper describes a component-based approach for both designing and building agents. This approach allows both a bottom-up design through composition (of basic behavior components) and a top-down design through progressive decomposition. After motivating our work, we describe ou design principles. We then introduce some examples of simulation of eco-systems. This leads to several emerging design patterns. We compare our approach to similar work on modular agent architectures before concluding on our future work.*

KEY WORDS : *Components, Agents, Conception, Architecture, Simulation.*

Introduction

Les modèles et architectures d'agents proposés dans la communauté ont pour objectif de structurer et faciliter la conception et la programmation d'agents. Cependant on s'aperçoit que ces architectures restent encore souvent relativement monolithiques dans l'expression du comportement d'un agent. Un modèle d'architecture modulaire est donc intéressant car il aide à une expression modulaire de comportements et donc à une décomposition de la complexité. Il permet également une certaine réutilisabilité par l'identification de comportements de base utilisables dans de nombreux contextes et combinables.

Dans cet article nous présentons un modèle d'architecture d'agents fondé sur une notion de composants issue du génie logiciel. Ainsi, la propriété de réutilisabilité des composants réduit l'ampleur du développement, et la manipulation des comportements sous forme componentielle ouvre la voie à la conception assistée via la reconnaissance de schémas de conception (design patterns). De plus, le modèle componentiel autorise à modifier dynamiquement la structure de chacun des agents - notamment l'ajout et la suppression de comportements. Afin d'illustrer ce modèle de conception, nous présentons deux exemples de simulation multi-agent, lesquelles ont été réalisées sur la plate-forme de conception d'agents à architecture componentielle MALEVA [LHU 98].

1. Le modèle componentiel de conception d'agents

1.1. *Le composant logiciel*

Le paradigme componentiel propose un mécanisme d'appel de méthode sans référencement explicite du composant-récepteur. Pour ce faire, les composants sont munis de **bornes de communication** protégeant leurs méthodes. Chaque composant détient des bornes d'entrée et de sortie. Chacune des bornes de sortie est potentiellement liée, via une **connexion**, à une borne d'entrée d'un autre composant. Pour envoyer un message, le composant écrit les paramètres du message sur une borne de sortie. Le message est alors pris en charge par le **gestionnaire de communication** du composant appelant (voir figure 1). Ce dernier maintient d'une part la liste des connexions impliquant le composant propriétaire et d'autre part la liste des bornes. Le principal service fourni par le gestionnaire est d'appeler une borne/méthode se situant à l'extrémité de sortie d'une connexion, dans un autre composant. L'absence de référence au composant appelé dans les méthodes de l'appelant comporte deux avantages essentiels :

— Ce mécanisme de communication autorise la modification (éventuellement dynamique) du graphe des connexions sans avoir à changer les méthodes des composants.

— Les composants deviennent des entités potentiellement réutilisables, dans la mesure où leur définition est indépendante de leur environnement logiciel.

Ce modèle autorise d'autre part l'instanciation dynamique des composants et des connexions.

1.2. *Des composants pour les comportements*

Nous utilisons les composants afin de réifier les comportements d'agents, qui sont assimilables en définitive à du code exécutable. En outre, les comportements main-

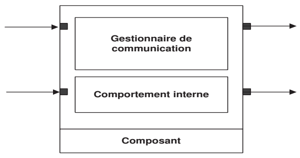

Figure 1. *Le composant et son gestionnaire de communication.*

tiennent un ensemble de données (la vitesse et direction courantes pour un comportement de déplacement d'un agent en environnement simulé par exemple) sans lesquelles ils n'ont aucun sens : pour manipuler les comportements de façon atomique, il est donc nécessaire d'encapsuler dans les composants d'une part le code des comportements qu'ils représentent, d'autre part les données qu'ils maintiennent. Un agent se résume alors à un réseau interconnecté de composants. Sa structure statique et ses méthodes sont distribuées parmi ses composants : le modèle componentiel rompt avec les architectures décrivant les agents sous forme monolithique plutôt que modulaire.

1.3. *Le composant MALEVA*

MALEVA [LHU 98] propose un paradigme de conception des agents fondé sur la séparation du flux de contrôle et de données entre les composants, faisant apparaître des **bornes de données** et des **bornes de contrôle**.

— **Bornes de données**. Les méthodes associées aux bornes d'entrée de données doivent se contenter d'affecter les variables d'instances du composant aux valeurs transmises dans le paramètre via la connexion. Symétriquement, les bornes de sorties de données transmettent les valeurs pour les variables d'instance à modifier.

— **Bornes de contrôle**. Un comportement encapsulé dans un composant n'est activé que lorsque ce dernier en reçoit explicitement l'ordre via une borne d'entrée de contrôle. Lorsque l'exécution du comportement est terminée, le composant peut rendre le contrôle par une borne de sortie de contrôle.

Ainsi, la différenciation contrôle/données permet plus de liberté dans l'expression du contrôle. On peut donc exprimer des composants actifs et indépendants (thread associé), des séquencements explicites ou toute autre forme de combinaison. La séparation contrôle/données est le principal apport en terme de réutilisabilité, dans la mesure où elle permet de réutiliser les composants dans des contextes d'activation différents.

2. Concevoir des agents dans le paradigme componentiel

2.1. *Une conception assistée*

Concevoir un système multi-agent dans le paradigme componentiel, c'est assembler des **comportements élémentaires** (sous forme de composants). Le concepteur doit donc disposer d'une bibliothèque de composants relatifs à son "métier". Cette approche permet de dégager en partie le concepteur de la tâche de programmation, prise en charge par le programmeur de composants élémentaires. La tâche du concepteur est de construire une application multi-agents à partir des composants fournis par le programmeur. Il est spécialiste du domaine d'application et n'a à priori pas de compétence spécifique en informatique : il assemble visuellement les différents composants dans un environnement graphique. Le programmeur de composants élémentaires veille à produire des composants suffisamment génériques pour être réutilisables dans un grand nombre d'applications. L'approche de MALEVA consistant à proposer une bibliothèque de comportements élémentaires sous forme de composants spécifiques ainsi que les outils en permettant la maintenance en fait un framework de conception et simulation de systèmes multi-agent.

2.2. *Analyse ascendante vs. descendante*

Nous distinguons deux techniques d'analyse pour la conception de composants composites [1] : l'approche ascendante et l'approche descendante. La première consiste à construire de façon incrémentale le composant composite en assemblant des composants élémentaires. A contrario, dans une approche descendante, le concepteur se donne comme base de travail les spécifications globales du comportement de l'agent, pour le décomposer étape par étape jusqu'aux composants élémentaires. Si l'approche ascendante est aisée à mettre en œuvre dans une interface graphique de conception, elle n'assure pas la conformité du composant produit à ses spécifications. Symétriquement rien n'assure que la décomposition descendante d'un comportement mène à un jeu de composants élémentaires inclus dans la bibliothèque disponible. Autrement dit l'approche descendante n'exclut pas d'avoir à programmer de nouveaux composants élémentaires, mais c'est la seule méthode qui assure de manière générale une certaine conformité vis-à-vis des spécifications.

2.3. *Les patterns de conception*

L'écriture du comportement d'un agent sous forme componentielle permet d'une part de réutiliser les composants, mais d'autre part elle met en évidence des similitudes (patterns) dans l'agencement des-dits composants, c'est-à-dire dans le graphe des connexions. Dans le domaine de la technologie du logiciel à objets, un design pattern[GAM 95] est la description abstraite d'un "bon" schéma de conception par objets qui peut être appliqué dans la conception d'un système quelconque. C'est par conséquent un moyen de maîtriser le sentiment de déjà vu souvent éprouvé par les concepteurs. L'écriture componentielle a l'intérêt de fournir des schémas de conception abstraits reflétant de façon complète le comportement des agents qu'ils repré-

1. Un composant composite est définit comme étant l'agrégation de plusieurs composants, les sous-composants devenant inaccessibles pour l'extérieur.

sentent. En effet, le comportement est déterminé par la nature des composants et le graphe des connexions. Ainsi rechercher des schémas de conception d'agent, c'est ici identifier "visuellement" des agencements récurrents. Nous pensons que les patterns représentent un outil de conception, dans la mesure où ils aident le concepteur à structurer le graphe des connexions. Les patterns fournissent un ensemble de connexions préétablies sémantiquement cohérentes autour desquelles le concepteur construit son architecture.

3. Exemples de conception

Nous présentons deux simulations que nous avons réalisées, illustrant les différentes techniques d'analyse. Dans un premier temps, nous mettons en œuvre l'approche ascendante pour concevoir les agents d'une simulation proie-prédateur. Nous détaillons par la suite les étapes de conception descendante de deux agents issus d'une application plus complexe relative à la simulation d'une colonie de fourmis.

3.1. *Conception ascendante : Exemple des proies-prédateurs*

Adoptant une stratégie ascendante, nous définissons un ensemble de composants élémentaires, représentant les comportements de base des proies et des prédateurs, à savoir la fuite, l'attraction, et le mouvement aléatoire, pour ensuite les composer, de manière à former des agents suivants:

— Une **proie** : cet agent fuit les prédateurs se situant dans son champ de perception. Si aucun prédateur n'est à proximité, la proie effectue un mouvement aléatoire. Le comportement de l'agent proie comprend trois composants : la fuite, le mouvement aléatoire, et un composant de contrôle appelé Switch qui permet de réifier la conditionnelle perception/non-perception[2]. Si aucun prédateur n'est perçu par le composant FuirAgent (donc si la borne Data n'envoie pas de données), le Switch active le comportement de mouvements aléatoire via l'enclenchement de la borne de contrôle du composant MvtAléatoire.

— Un **prédateur** : il suit les proies tout en fuyant ses congénères, et se déplace aléatoirement s'il ne perçoit rien. Un prédateur est donc vu comme une proie (il fuit les prédateurs et effectue un mouvement aléatoire) auquel a été ajouté un comportement de prédation (il suit les proies). En plus de réutiliser les composants de la proie, nous en réutilisons la structure (au sens des connexions).

La figure 2 montre bien la corrélation entre les deux architectures. Pour l'élaboration du prédateur, il a suffit d'ajouter deux composants sans modifier le schéma structurel initial de la proie. Ce principe de composition permet à moindre coût de réaliser une proie qui suit ses congénères, un prédateur qui attaque tous les agents, etc. . .

2. Le composant Switch permet la séparation d'un flot de contrôle en deux selon une conditionnelle de la forme :
Si Data(IN) Alors SendControl(OUT1) Sinon SendControl(OUT2).

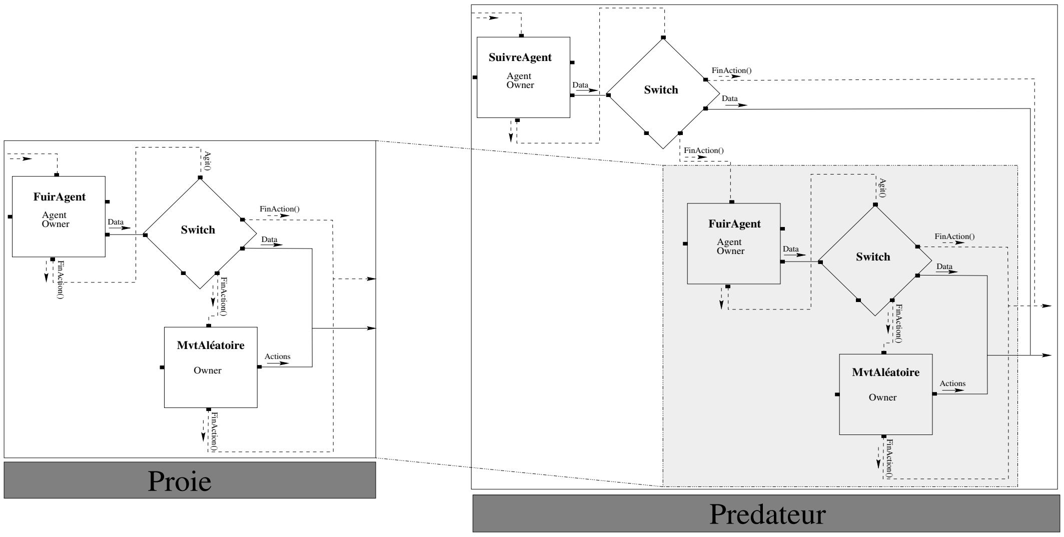

Figure 2. *Architecture d'une proie et d'un prédateur. Les lignes pleines représentent les connexions de flots de données, les pointillés les flots de contrôle.*

3.2. *Conception descendante : Exemple de la fourmilière*

Afin d'illustrer la méthode descendante de conception d'agents, nous avons réalisé une simulation de fourmilière, en nous inspirant des travaux effectués par Alexis Drogoul dans le cadre du projet MANTA [DRO 93]. Ce dernier consiste en la simulation d'une colonie de fourmis et l'étude de la sociogénèse qui en résulte. Dans notre application nous avons légèrement simplifié le comportement de ces fourmis, si bien que l'organisation globale de ces agents ne correspond pas au modèle biologique. Aussi nous nous sommes restreints à la conception de quatre types d'agents : les œufs qui émettent un stimulus pour indiquer leur présence aux autres agents, les larves qui ont besoin d'être soignées et nourries (par envoi de stimuli spécifiques), les ouvrières qui sont capables de se déplacer, de suivre des gradients de phéromones (stimulus), de porter, nourrir, et soigner d'autres agents, et enfin la reine qui possède la capacité de pondre des œufs pour perpétuer l'espèce. Nous nous limitons dans cet article à la présentation des étapes de conception relatives aux ouvrières. Pour plus de détails sur l'implantation complète de cette simulation, le lecteur pourra se référer à [GUI 98].

3.2.1. *Les comportements propres aux êtres vivants*

Chacun des agents du système est, en tant qu'être vivant, soumis aux contraintes temporelles, c'est-à-dire que tous les agents que nous mettrons en place auront la "faculté" de vieillir et de mourir. Ainsi, l'architecture de base de tout être vivant peut être décomposée en deux parties (c.f. figure 3): sa capacité à vivre (et donc vieillir), et

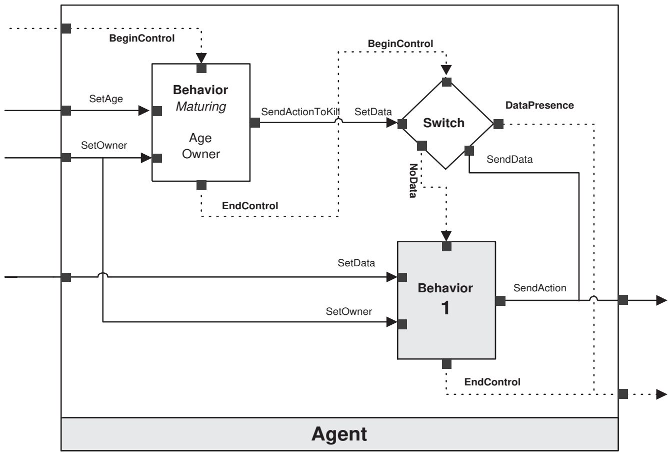

Figure 3. *Motif de base d'un agent vivant.*

son comportement propre. Nous construisons un composant encore abstrait (il sera décomposé par la suite selon une approche descendante), $Maturing$ pour représenter ce comportement de vieillissement. Lorsque l'agent doit mourir, une action est envoyée à son effecteur[3] via la borne SendKillAction pour faire disparaître l'agent de l'environnement. Maturing nécessite la connaissance de l'agent auquel il appartient, aussi il est muni d'une borne SetOwner. Ce comportement est si simple que nous aurions pu en faire un comportement élémentaire, mais afin d'illustrer l'analyse descendante, nous le décomposons en deux comportements élémentaires : l'un est chargé du principe de vieillissement, l'autre est spécialisé dans l'envoi du signal d'auto-destruction. Le composant comportemental résultant, à savoir Maturing est représenté sur la figure 4. Maturing comporte trois composants, dont deux composants comportementaux élémentaires (Aging et Dying) et un composant de contrôle (Switch).

— Aging. Ses bornes d'entrée sont SetOwner et SetAge. Seule SetOwner est connectée (au composant représentant l'agent englobant pour la connaissance de soi), SetAge ayant pour but de fournir un accès en écriture à la donnée Age encapsulée dans le composant. Si cette borne n'est pas utilisée dans cette application, il est toutefois intéressant de la conserver pour donner au composant $Aging$ une généricité maximale. La variable Age est initialisée à zéro lors de la création du composant. Cette valeur est incrémentée à chaque fois que le composant reçoit le contrôle sur sa borne BeginControl. Si l'âge maximum est atteint (constante AGE_MAX), sa valeur est envoyée par la borne de sortie SendAge au Switch.

— Switch. Lorsqu'il reçoit le flux de contrôle depuis Aging, ce composant scrute sa borne d'entrée de données SetData. Si aucune information n'a été reçue depuis la dernière activation, le contrôle est envoyé via la borne NoData à la borne de sortie de

3. L'effecteur est un composant technique (non comportemental) présent dans tous les agents, qui traduit dans l'environnement les actions déterminées par les composants comportementaux.

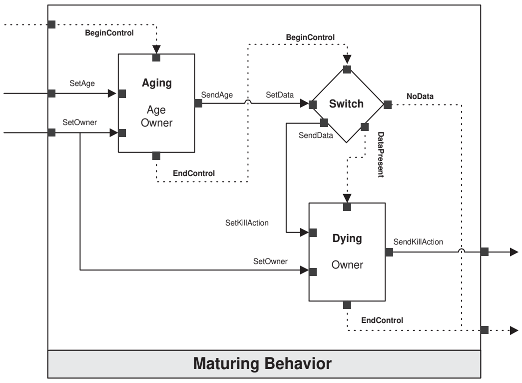

Figure 4. *Décomposition de Maturing.*

Maturing[4], et dans le cas contraire (i.e. si l'âge maximum est atteint), le Switch active le composant Dying via sa borne DataPresence.

— **Dying**. Le plus élémentaire de tous les composants. . . Dying se contente de transformer un ordre reçu sur sa borne SetKillAction en une structure permettant de véhiculer une action, laquelle est au format adéquat pour être comprise par l'effecteur. Ce comportement va être réutilisé dans tous les agents que nous allons créer, ainsi la base de départ de nos agents va correspondre au motif de la figure 3.

3.2.2. *Les ouvrières*

Les ouvrières ont un comportement relativement complexe puisque les fonctionnalités de celles-ci sont multiples : en particulier celles-ci sont capables de se déplacer, de suivre des gradients de phéromones, de porter des agents, de soigner des agents et de tuer les œufs pour nourrir d'autres agents.

Pour commencer, comme tous nos agents les ouvrières rentrent dans le cadre du pattern mis en place précédemment (figure 3). Reste à réaliser en terme de composants élémentaires le comportement de l'ouvrière. L'approche descendante nous conduit à distinguer les cas où l'ouvrière capte un stimulus et les cas où elle n'en capte pas (figure 5). Dans le cas où l'agent ne perçoit pas de stimulus, celui-ci effectue des mouvements aléatoires. Lorsque au moins un stimulus est capté l'ouvrière va réagir en fonction de l'intensité du stimulus. Là encore la délibération vis à vis d'un stimulus peut s'opérer de deux façons différentes (le composant *behavior 2* de la figure 5 est développé en figure 6):

— si l'agent se situe sur un maximum local, c'est-à-dire à l'endroit où se situe la source d'émission du stimulus, selon le type de ce dernier une action est renvoyée.

— sinon il suit le gradient associé au stimulus (composant Following de la figure 6).

Il apparaît que le motif mettant en relation deux composants comportementaux et un

4. au sens de la composition structurelle

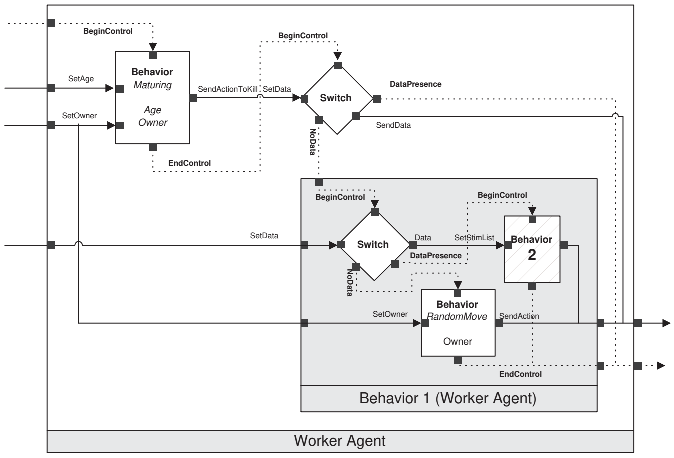

Figure 5. *Comportement de l'ouvrière : premier niveau d'analyse.*

Switch se retrouve dans les figures 4, et 5, et ce à différents niveaux dans l'analyse du comportement de l'ouvrière. Ceci pourra être exploité dans le cadre de la conception assistée.

A ce stade de l'analyse, les comportements sont suffisamment simples pour être réifiés par des composants élémentaires. Le résultat de l'analyse est représenté sur la figure 7, où l'on peut discerner les différents niveaux d'analyse : les cadres regroupant les composants par blocs indiquent les étapes de l'analyse descendante.

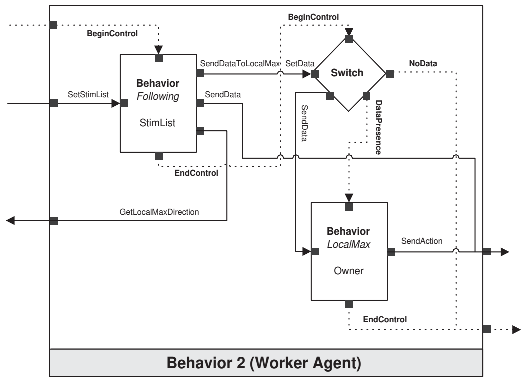

Figure 6. *Comportement de l'ouvrière : second niveau d'analyse.*

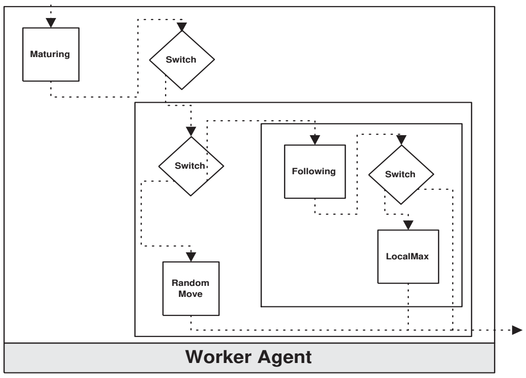

Figure 7. *Le comportement de l'ouvrière, incluant les deux niveaux d'analyse.*

3.3. *Synthèse*

Ces deux exemples (proie/prédateur et fourmis) d'instanciation du modèle componentiel proposé montrent qu'aussi bien une approche ascendante que descendante sont possibles avec notre modèle. Les avantages de la décomposition componentielle sont pleinement exploités par ces deux approches, aussi bien au niveau de la réutilisation de composants (utilisation de l'architecture de la proie pour générer le prédateur), que de la gestion de la complexité.

4. Implémentations

Le modèle que nous proposons a donné lieu à l'implémentation de la plate-forme Maleva [LHU 98]. Elle fournit un ensemble d'outils permettant l'évolution d'agents en environnement simulé. Ces outils permettent : de concevoir des composants, de composer les composants précédemment conçus, de concevoir des environnements topologiques, et enfin de simuler les agents dans un environnement.

Les exemples d'instanciation du modèle décrit ci-dessus (proie/prédateur et fourmis) ont été implantés sur cette plate-forme et ont permis une validation du modèle et du prototype (pour des descriptions détaillées de ces implantations se référer à [LHU 98] et [GUI 98]). Cette plate-forme a été développée en Delphi, langage de développement orienté composants de Borland. Actuellement une deuxième version de cette plate-forme est en cours de réalisation en prenant comme base le modèle de JavaBeans proposé par Sun. Le modèle componentiel décrit dans cet article est étendu afin de fournir une plate-forme de conception mais également d'exécution complètement dynamique. Nous souhaitons étendre l'exemple des fourmis décrit ci-dessus en fournissant à cette nouvelle version du framework les outils permettant une gestion automatique de cette dynamicité architecturale en cours d'exécution (passage de l'œuf à la la larve, de la larve à la fourmi). Ainsi, la notion de typage fort des connexions entre composants est ajoutée au modèle, et Java fournit des capacités d'introspection et de réflexion qui permettent à cette nouvelle plate-forme d'instancier de manière complètement dynamique de nouveaux composants et d'en déduire les schémas d'interconnexion intercomposants. Enfin, via ces outils de typage et de composition visuelle, nous espérons pouvoir identifier plus facilement des schémas de conceptions (Design Patterns) capables de guider le concepteur d'agents (au sens agenceur de composants) pour la génération d'architectures d'agents de tout type (réactif, délibératif, hybride, ...). Cette identification de dénominateurs communs entre ces différentes architectures pourrait permettre une convergence de vision de l'architecture des agents.

D'un point de vue des performances, il est évident que le surcoût engendré par la granularité potentiellement très fine de la décomposition de l'architecture de chaque agent est non négligeable. Ce surcoût représente le poids (en performance) de la généricité et de la modularité comparé aux performances d'un codage brut de l'architecture de chaque agent, voire du système multi-agent. Plus la granularité de décomposition sera fine, plus les performances baisseront, mais potentiellement la réutilisabilité de chaque composant sera possible. Rien n'empêche de définir via MALEVA un agent dont l'architecture est représenté par un composant unique (donc équivalent à du code exécutable), dans ce cas les performances ne souffrent pas du poids de la décomposition. La plateforme MALEVA, et plus généralement le modèle componentiel qu'elle fournit peut donc être utilisé de deux manières différentes : soit comme plateforme de validation afin de tester l'architecture générale des agents et de verifier la cohérence des différents modules avant de passer à une implantation brute en code exécutable, soit comme plateforme d'implantation et dans ce cas l'impact de la granularité sur les performances ne doit pas être négligé. Rien n'empêche toutefois d'ajouter à MALEVA un générateur de code chargé d'optimiser certaines parties de codes lorsque les architectures générées n'utilisent pas tous les atouts de l'approche componentielle (par exemple rendre certains appels de méthodes synchrones lorsque la concurrence intra-composant n'est pas exploitée).

5. Travaux connexes

Peu de frameworks utilisent un modèle componentiel pour caractériser l'architecture des agents, l'exploitation de la modularité par ces plateformes s'opérant essentiellement au niveau du système multi-agent et peu au niveau agent [MOO 95]. Pour des descriptions détaillées de telles plateformes modulaires, le lecteur pourra se référer à [BRE 98], [DEC 96], [BRA 95], et plus généralement à [SHE 98] pour une discussion générale sur les architectures de différents systèmes multi-agents. Nous pouvons notamment citer JAF (Java Agent Framework [HOR 98]) comme framework exploitant un modèle componentiel. JAF étend le modèle des Java Beans en renforçant les relations inter-composant et en ajoutant des mécanismes de contrôle intra-composant. Contrairement à Maleva, ce framework ne sépare pas explicitement flot de contrôle et flot de données. De plus, un composant JAF peut référencer des services fournis par un autre composant, mais sans en avoir une représentation explicite. Ainsi, chaque composant définit de manière statique une liste de dépendances décrivant les composants dont il a besoin pour pouvoir s'exécuter. Lors de l'instanciation de ce composant en phase de conception, il enregistre sa liste de dépendance, et JAF cherche la meilleure correspondance entre les descriptions fournies dans la liste et les composants présents lors de l'exécution.

La principale différence entre JAF et Maleva se situe au niveau de décomposition des comportements au sein des composants. JAF opère cette décomposition à un niveau relativement élevé, alors que le modèle fournit par Maleva voit les décompositions à un grain beaucoup plus fin, ceci étant principalement dû à la représentation explicite des flots de contrôle. Toutefois, cette différence reste très conceptuelle : rien n'empêche de concevoir des composants avec JAF à un grain extrêmement fin, et de même, Maleva peut accueillir des composants dits monolithiques.

De manière générale, cet article présente un modèle componentiel d'architecture d'agents basé sur une décomposition comportementale (un comportement est modélisé par un composant). Toutefois, l'architecture componentielle MALEVA ne fixe pas à priori le mode de décomposition. Il sera donc intéressant d'étudier comment d'autres formes de décomposition (fonctionnelles, modulaires, . . .), par exemple les modules de délibération, communication, et perception de DIMA [GUE 96], peuvent être décrites en MALEVA. C'est d'ailleurs un de nos projets en cours.

Conclusion

La manipulation des comportements sous forme de composants ouvre la voie à la conception assistée d'agents : en effet, ce paradigme incite le concepteur à effectuer une analyse descendante des comportements. De plus, la dissociation des tâches du concepteur et du programmeur permet de recentrer l'activité de chacun sur sa spécialité. L'approche componentielle semble ainsi être un bon guide pour l'établissement et la maintenance d'une bibliothèque de comportements élémentaires relativement génériques. Mais pour que cette approche permette effectivement d'aider le concepteur, il est impératif de mettre à sa disposition un grand nombre de patterns, ou schémas de conception, qui permettent de guider la décomposition des comportements en proposant des structures d'assemblage de composants. Or la reconnaissance de patterns nécessite de passer en revue un nombre important d'agents sous forme componentielle, dans le but d'en extraire les-dits schémas récurrents. L'exemple de la fourmilière que nous présentons dans cet article nous a permis de dégager des patterns de conception centrés sur le gel des connexions autour des composants de contrôle tels que le Switch. Si ces patterns sont rudimentaires, ils permettent néanmoins d'aider le concepteur à structurer le flux de contrôle au sein de l'agent qu'il développe. Le travail de reconnaissance de patterns, essentiellement expérimental, nécessite la conception de diverses applications pour étoffer l'ensemble des motifs proposés au concepteur dans un framework de construction componentielle de comportements d'agents.

Références

[BRA 95] F. BRAZIER, B. DUNIN-KEPLICZ, N. R. JENNINGS ET J. TREUR. Formal specification of multi-agent systems : a real-world case. Proceedings of the First International Conference of Multi-Agent Systems, pages 25–32, 1995. San Fransisco, CA, MIT Press.

[BRE 98] A. BRENTON ET T. WOOD. JACK intelligent agents user guide, 1998. ©Agent Oriented Software Pty Ltd.

[DEC 96] K. DECKER ET K. SYCARA. Designing reusable behaviors for information agents. January 1996. The Robotics Institute, Carnegie-Mellon University.

[DRO 93] A. DROGOUL. De la Simulation Multi-Agent la Résolution collective de Problèmes. PhD thesis, Université Paris 6, 1993.

[GAM 95] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON ET JOHN VLISSIDES. Design Patterns - Elements of Reusable Object Oriented Software. Addison-Wesley Professional Computing Series, 1995.

[GUE 96] ZAHIA GUESSOUM. Un environnement opérationnel de conception et de réalisation de systèmes multi-agents. PhD thesis, Université Paris VI, May 1996. LAFORIA TH96/09.

[GUI 98] A. GUILLEMET ET G. HAÏK. Evaluation d'une architecture componentielle de conception d'agents. Rapport de stage de 1ère année de magistére d'informatique et modélisation, Ecole Normale Supérieur de Lyon, September 1998.

[HOR 98] B. C. HORLING. A reusable component architecture for agent construction. Technical Report 1998-49, UMASS Computer Science, October 1998.

[LHU 98] M. LHUILLIER. Une Approche à Base de Composants Logiciels pour la Conception d'Agents. Principes et Mise en Oeuvre à Travers la Plate-forme Maleva. PhD thesis, Université Paris VI, February 1998.

[MOO 95] B. MOORE, M. NOSCHANG ET J. PENIX. An annotated bibliography of software agent technology. 1995.

[SHE 98] ONN SHEHORY. Architectural properties of multi-agent systems. Technical Report CMU-RI-TR-98-28, The Robotic Insitute, December 1998.