Une approche à base de composants pour la conception d'agents

Thomas Meurisse — Jean-Pierre Briot

Laboratoire d'Informatique de Paris 6
LIP6 - OASIS
8, rue du Capitaine Scott
F-75015 Paris

{Thomas.Meurisse, Jean-Pierre.Briot}@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 via l'introduction du modèle de composant que nous proposons : le modèle de composant Maleva. Plusieurs exemples de conception d'agents sont présentés dans le domaine de la simulation d'écosystèmes. Nous comparons ensuite notre approche à d'autres travaux similaires en matière d'architectures modulaires d'agents avant de conclure sur les travaux futurs.

ABSTRACT. This paper describes a component-based approach for 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 our design principles through the definition of our component model : the Maleva component model. We then introduce some examples of simulation of eco-systems. We compare then our approach to similar work on modular agent architectures before concluding on our future work.

MOTS-CLÉS : composants, agents, conception, architecture, simulation.

KEYWORDS: components, agents, conception, architecture, simulation.

Introduction

Les architectures à base de composants ont pour objectif de permettre une meilleure structuration du développement d'application et de fournir des possibilités accrues de décomposition et de réutilisation. La finalité recherchée étant de pouvoir manipuler facilement un groupe de composants en les connectant les uns aux autres afin d'obtenir une solution au problème posé.

Aujourd'hui dans les différents domaines de recherche utilisant les systèmes multi-agents (*SMA*), de nombreux travaux concernent l'identification de propriétés spécifiques à ces systèmes permettant de rationaliser les étapes de conception d'un SMA [JEN 00]. Toutefois l'architecture des agents, telle qu'elle peut se définir dans ces travaux, comporte encore de nombreuses zones d'ombre, l'agent étant vu comme l'entité canonique de ces systèmes [WOO 00]. Un agent est donc défini dans ces approches comme un acteur, un objet actif de granularité minimale, etc. Nous pensons que l'utilisation de composants pour définir l'architecture interne des agents peut permettre une rationalisation « efficace » des agents.

Ainsi, nous présentons dans cet article un modèle d'architecture d'agents fondé sur la notion de composants. Le but de ce modèle est de permettre une conception incrémentale des agents en exploitant les notions de décomposition architecturale que permettent les composants, mais également de permettre la réutilisation de comportements (réifiés par les composants) en intégrant différentes bibliothèques de composants. Afin d'illustrer ce modèle de conception, nous présentons deux exemples de simulation multi-agents, lesquels ont été réalisés avec la plate-forme de conception d'agents à architecture componentielle MALEVA [LHU 98].

Nous allons tout d'abord nous intéresser dans la première section au modèle de composant proposé (composants *Maleva*), puis présenter dans la section suivante les possibilités que fournissent ces approches à base de composants pour la conception d'agents. La section 3 présente différents exemples de conception et donc d'instanciation du modèle proposé. La section 4 présente les implantations découlant de l'instanciation de ce modèle de composant, et notamment la plate-forme Maleva. Et enfin, la section 5 présente les travaux connexes à cette approche.

1. Notre modèle componentiel de conception d'agents

1.1. *Le composant logiciel*

Les approches à base de composants proposent un mécanisme de communication permettant le découplage entre entités. Pour ce faire, ces composants sont munis de **bornes de communication**.

Les bornes de plusieurs composants vont être connectées entre elles de manière assurer l'assemblage et la communication d'informations entre eux. C'est dans ce sens que les modèles de conception orientée composants peuvent être vus comme un paradigme de conception supérieur à celui des objets classiques. L'intérêt de cette approche est principalement le découplage entre entités, les dépendances n'étant plus explicitées à l'intérieur même de celles-ci, mais définies de manière externe [PES 00]. Chaque composant peut être partiellement décrit en fonction de ses bornes d'entrée et de sortie (notion d'interface). Chacune des bornes de sortie est potentiellement liée, *via* une **connexion**, à une borne d'entrée d'un autre composant. L'idée ici est de fournir une abstraction de communication entre entités indépendante des choix opérationnels. Par exemple la manière dont les composants s'échangent des données peut s'apparenter à un modèle de communication type *publish/subscribe*, semblable à celui des JavaBeans [SUN 97]. Mais l'utilisateur des composants ne doit pas a priori pouvoir faire des hypothèses sur ce modèle.

L'absence de référence à l'entité appelée dans la définition 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 composants. Les principes d'encapsulation ne sont pas violés.

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

Les approches de conception par composants ne sont pas à opposer totalement à celles du monde objet, bien que souvent elles essaient de résoudre les problèmes amenés par le *tout objet*. Le couplage en est un exemple des plus significatifs. Dans le monde objet, lorsqu'un objet communique avec un autre, implicitement via le passage de message, un couplage aussi bien structurel que comportemental apparaît (échange des données mais, et surtout, du contrôle) [PES 00]. Les composants essaient de relâcher ce couplage, autant d'un point de vue statique (un composant ne référence pas directement un autre composant), que dynamique (le passage de données entre composants n'implique pas forcément le passage du contrôle).

1.2. *Des composants pour les comportements*

Nous utilisons les composants afin de réifier les comportements internes des agents. Dans le monde des agents, ces comportements doivent maintenir 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 fonctionnalités sont distribuées parmi ses composants. Le modèle componentiel appliqué aux agents permet donc de définir les agents de manière modulaire plutôt que monolithique.

	Données	**Contrôle**
`Bornes d'entrée`	Consommation de données	Point d'entrée de l'activation
`Bornes de sortie`	Production de données	Point de sortie de l'activation
`Connexion`	Propagation de données	Propagation de l'activation

Tableau 1. *Différences entre données et contrôle dans le modèle de composant Maleva*

1.3. *Flot de contrôle et flot de données*

MALEVA [LHU 98] propose un paradigme de conception des agents fondé sur la séparation du flot de contrôle et du flot de données entre les composants, faisant apparaître deux types de bornes au sein d'un composant :

– **Bornes de données**. Elles ne servent qu'à véhiculer des données entre composants.

– **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.

Le tableau 1 résume les différences données/contrôle au niveau des différents types de bornes des composants et des connecteurs. La figure 1 donne un exemple de schéma utilisant le modèle de composant proposé. Sur l'ensemble des schémas présents dans cet article, nous représentons, par convention, les connexions liées au contrôle en pointillé, et les connexions liées aux données en trait plein. Toujours par convention, les bornes de données sont à lire de gauche à droite (borne d'entrée à gauche du composant, borne de sortie à droite) et les bornes de contrôle de haut en bas (borne d'entrée en haut, borne de sortie en bas). Ici les composants A et B sont activés en séquence : la borne de sortie de contrôle de A est connectée à la borne d'entrée de contrôle de B, ou formulé différemment, B est activé lorsque A termine son traitement.

En terme de données, le composant B ne consommera les données produites par le composant A que lorsque A terminera son traitement. La figure 2 présente les deux mêmes composants, mais connectés différemment : l'activation peut se faire simultanément, les bornes d'entrée de contrôle de chaque composant étant connectées à une même source. Par contre les connexions relatives aux données restent les mêmes, nous avons ici un modèle d'exécution s'approchant du style architectural *Pipes and Filters* [SHA 96] : le composant B peut consommer « en continu » ce que A produit.

Ainsi, la séparation contrôle/données permet plus de liberté dans l'expression du contrôle. Il est possible d'exprimer des composants actifs et indépendants (*thread* associé), des séquencements explicites ou toute autre forme de combinaison. Ce nouveau pouvoir d'expression est fourni au niveau du graphe d'interconnexion et non plus dans le code interne du composant. Ceci peut se traduire en deux principaux apports : d'une part les composants gagnent en généricité (une partie de la description des activations/du graphe de contrôle est extraite du composant), et d'autre part l'expressivité

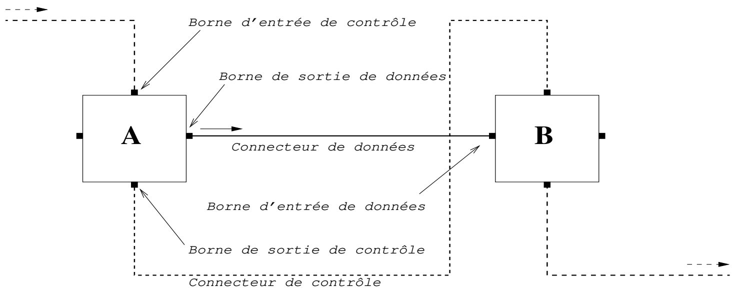

Figure 1. *Activation en séquence de deux composants*

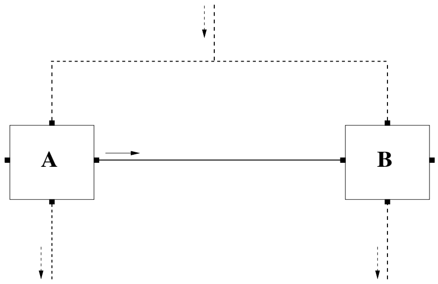

Figure 2. *Activation en parallèle de deux composants*

au niveau du graphe de connexion des composants de contrôle donne à l'agenceur de composants la capacité de modifier les politiques d'exécution intra-composants.

Dans un autre contexte, une distinction entre flots de données et flots de contrôle avait déjà été introduite via SADT [MAR 87], mais propose principalement cette séparation au niveau conception/conduite de projet. Ici nous nous proposons, en plus d'intégrer cette distinction au niveau conception, de la rendre accessible à l'implémentation[1].

Dans le domaine de la conception d'agents, cette distinction permet de rationaliser l'architecture des agents en gardant un modèle de conception relativement simple. Un des problèmes classique auquel se retrouve confronté un concepteur de système multi-agent est celui de la synchronisation des agents lors d'une simulation par exemple. En utilisant la séparation contrôle/données, cette synchronisation devient plus simple à modéliser : à chaque pas de simulation, le simulateur « envoie » aux différents agents un signal d'activation (plus exactement aux bornes d'entrée de contrôle des agents) et se met en attente d'un signal de retour de chacun des agents. Entre ces deux phases les

1. Pratiquement, rien n'empêche lors de l'utilisation de SADT de faire apparaître cette distinction également à l'implémentation. Les idées sont similaires mais utilisées dans un contexte très différent.

agents ont été activés et ont pu enclencher leurs comportements respectifs. De même, [LAW 00] montre que des résultats de simulations peuvent se retrouver biaisés dans des cas où l'ordonnancement des actions au sein d'un agent reste déterministe. Rendre le contrôle explicite dans l'architecture des agents permet de modifier simplement ces ordonnancements de comportements, alors que si les différents comportements des agents sont codés sous forme monolithique selon un modèle objet classique par exemple, ces modifications (bien que possibles) deviennent beaucoup plus difficiles.

2. Concevoir des agents dans un paradigme componentiel

2.1. *Une conception assistée*

Concevoir un système multi-agents dans le paradigme componentiel consiste à 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 *a 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 consiste à proposer une bibliothèque de comportements élémentaires sous forme de composants génériques ou spécialisés ainsi que les outils qui en permettent la maintenance. Ceci en fait un cadre 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 [2] : l'approche ascendante et l'approche descendante.

L'approche ascendante 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 dis-

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

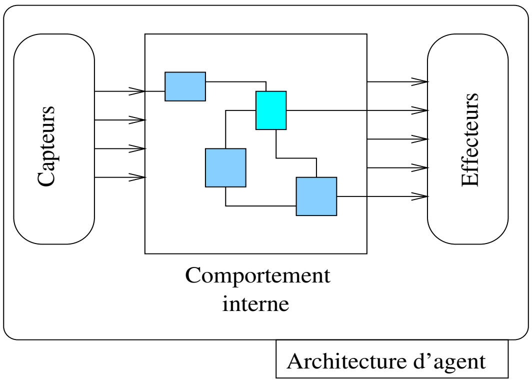

Figure 3. *Architecture des agents*

ponible. 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.

3. Exemples de conception

Dans les exemples présentés ici, nous nous intéressons à la simulation d'écosystèmes. L'architecture « type » d'un agent dans ce cadre est conforme à la figure 3. Un agent est situé dans un environnement, qu'il perçoit par l'intermédiaire de ses capteurs, et sur lequel il peut émettre des actions via ses effecteurs. L'effecteur est un composant technique (non comportemental) présent dans tous les agents, qui traduit dans l'environnement les actions déterminées (ex. se déplacer, prendre de la nourriture, laisser une trace, mourir, ...) par les composants comportementaux. De même, le capteur récupère des informations fournies par l'environnement (position des différents agents présents dans la simulation, présence d'obstacles, présence de traces, ...) et les rend accessibles à l'agent.

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 les 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 aléa-

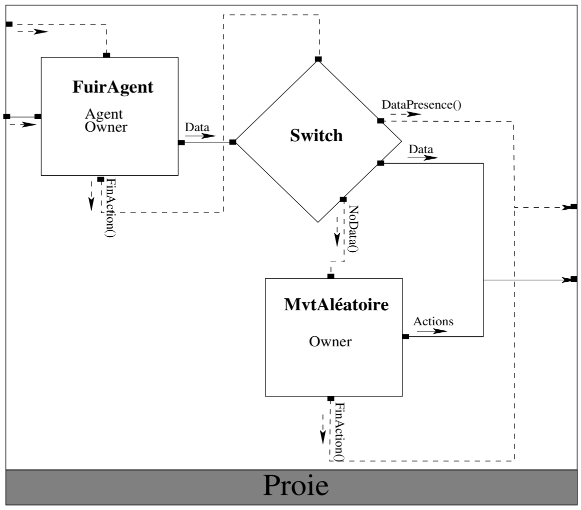

Figure 4. *Architecture d'une proie*

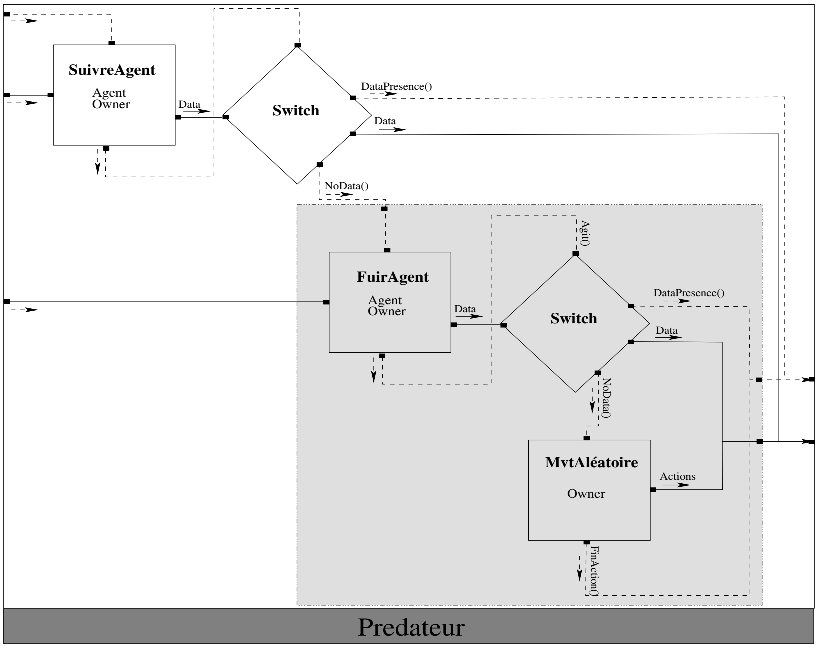

Figure 5. *Architecture d'un prédateur (la zone grise délimite l'architecture de la proie)*

toire. Le comportement de l'agent *proie* est donc défini par trois composants : la fuite (le composant *FuirAgent*), le mouvement aléatoire (le composant *MvtAléatoire*), et un composant de contrôle appelé *Switch* qui permet de réifier la conditionnelle perception/non-perception [3]. Le composant *FuirAgent* ne produit aucune donnée si aucun prédateur n'est détecté, sinon il produit une donnée spécifiant le déplacement à effectuer pour fuir le prédateur (qui sera récupérée par l'effecteur, qui l'appliquera ensuite à l'environnement). Si aucun prédateur n'est perçu par le composant *FuirAgent* (donc si la borne *Data* ne produit pas de données), le *Switch* active le comportement de mouvement aléatoire via l'enclenchement de la borne de contrôle du composant *MvtAléatoire*. Tout comme le composant *FuirAgent*, ce composant produit une donnée spécifiant le déplacement à effectuer.

- Un **prédateur** : il suit les proies tout en fuyant ses congénères, et se déplace aléatoirement s'il ne perçoit aucun autre agent. 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). Il est donc défini comme un nouveau composant englobant le composant *proie* spécifié auparavant.

Les figures 4 et 5 montrent bien la corrélation entre les deux architectures. Les lignes pleines représentent les connexions de flots de données, les pointillés les flots de contrôle. Pour l'élaboration du prédateur (fig. 5), il a suffi d'ajouter deux composants sans modifier le schéma structurel initial de la proie (qui est représenté en zone grisée sur la figure). Ce principe de composition permet à moindre coût de réaliser une proie qui suit ses congénères, un prédateur qui attaque tous les agents, etc.

Cet exemple de conception ne met en jeu qu'un nombre réduit de composants. L'exemple suivant montre que ce paradigme de conception s'avère aussi efficace pour des applications plus conséquentes et cette fois selon une approche descendante.

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),

3. 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).

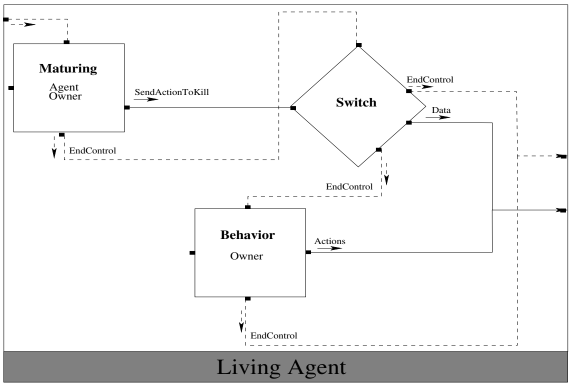

Figure 6. *Motif de base d'un agent vivant*

- les ouvrières qui sont capables de se déplacer, de suivre des gradients de phéromones (stimulus), de porter, nourrir, et soigner d'autres agents,

- 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 et aux œufs. 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 (cf. figure 6) : sa capacité à vivre (et donc vieillir), et 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 *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'autodestruction. Le composant comportemental résultant, à savoir *Maturing* est représenté sur la figure 7.

Maturing comporte trois composants, dont deux composants comportementaux élémentaires (*Aging* et *Dying*) et un composant de contrôle (*Switch*) :

- **Aging**. Une 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

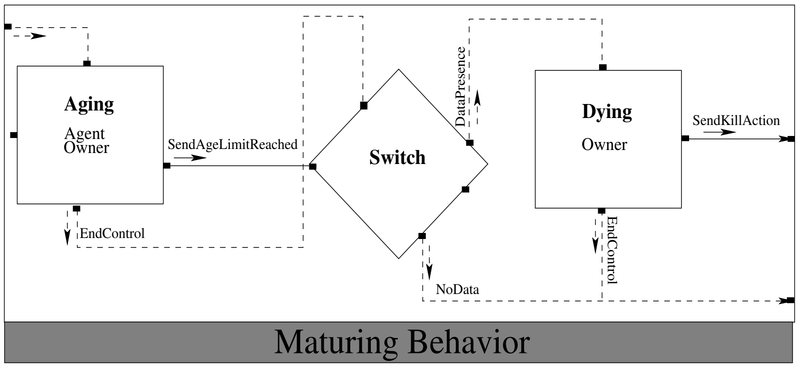

Figure 7. *Décomposition de* Maturing

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 flot 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 *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 *SendKillAction* 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 6.

3.2.2. *Les œufs*

L'œuf est ce que l'on peut considérer comme la phase de maturation d'une fourmi. Il est instancié de façon dynamique par la reine (ponte). Cet agent est soumis aux contraintes temporelles (*Maturing*, figure 7), et exhibe un comportement spécifique. Il faut savoir que les *Ectatomma ruidum*, espèce étudiée dans MANTA, ont un comportement de cannibalisme, c'est-à-dire qu'elles se nourrissent de leurs œufs. Ainsi le comportement spécifique de ces derniers consiste en l'émission de phéromones qui vont permettre aux fourmis adultes de repérer leur source de nourriture. Il est réifié par le composant *EmitStim* de la figure 8 qui représente le comportement de l'œuf (**Maturing+EmitStim**).

La conception de cet agent ne met pas en œuvre explicitement l'approche descendante, dans la mesure où le comportement *EmitStim* est suffisamment simple pour être

4. au sens de la composition structurelle.

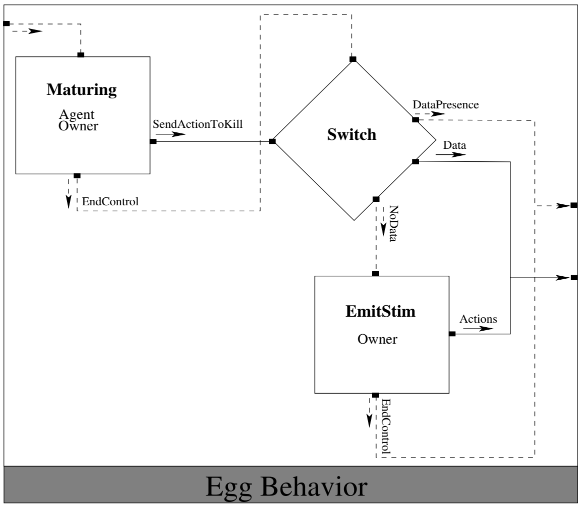

Figure 8. *Le comportement de l'œuf*

réifié par un composant élémentaire. L'émission de stimulus est un comportement fréquent - notamment dans le monde animal - aussi on entrevoit la possibilité de réutiliser le composant *EmitStim* dans le cadre d'autres applications.

3.2.3. *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* vieillissement/comportement interne mis en place précédemment (figure 6). 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 9). 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 9 est développé en figure 10) :

- 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 10).

Il apparaît que le motif mettant en relation deux composants comportementaux et un *Switch* se retrouve dans les figures 7, 8 et 9, et ce à différents niveaux dans l'analyse du

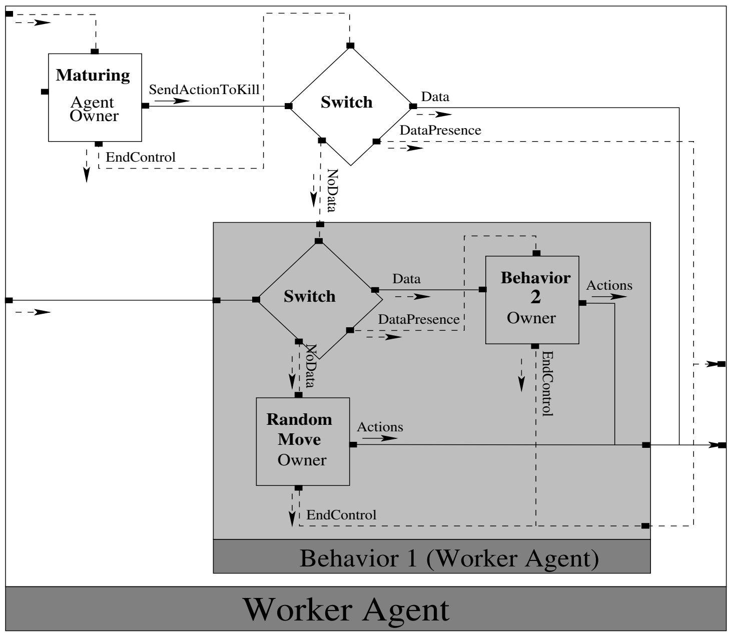

Figure 9. *Comportement de l'ouvrière : premier niveau d'analyse*

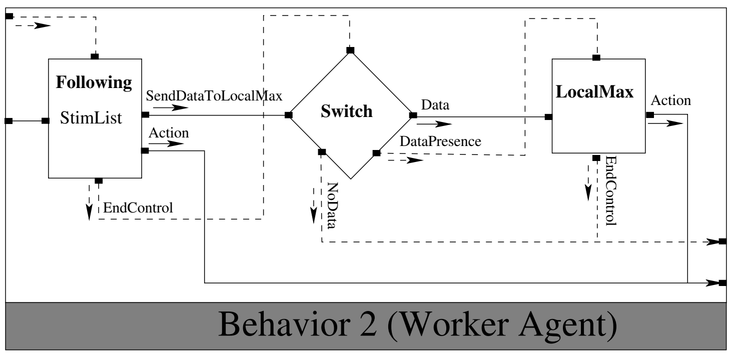

Figure 10. *Comportement de l'ouvrière : second niveau d'analyse*

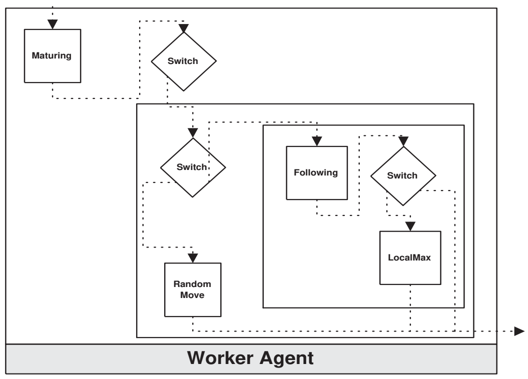

Figure 11. *Le comportement de l'ouvrière, incluant les deux niveaux d'analyse*

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 dans la figure 11, où l'on peut discerner les différents niveaux d'analyse : les cadres regroupant les composants par blocs indiquent les étapes de l'analyse descendante.

3.3. *Synthèse*

Ces deux exemples (proie/prédateur et fourmis) d'utilisation 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.

Il est intéressant de remarquer que tous les agents que nous avons créés ici peuvent s'intégrer très facilement dans un simulateur de système multi-agents. En effet, chaque agent possède au moins une borne d'entrée de contrôle permettant son activation[5], et en retour renvoie un signal sur une borne de contrôle de sortie. La synchronisation entre agents se trouve donc grandement simplifiée, et de manière plus générale, les phases d'implémentations (et de débugage). Il est facile d'identifier un composant mal conçu dès lors qu'il ne retourne pas sur sa borne de sortie un flot de contrôle après une activation.

5. Il s'agit ici d'un abus de langage. Ce n'est pas l'agent qui possède une borne, mais le composant complexe le représentant.

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 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 première validation du modèle de composant et du prototype opérationnel (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. Elle a également été utilisée par d'autres chercheurs pour prototyper des projets de simulation, notamment la simulation de trafic automobile [LEC 97].

Actuellement une deuxième version de cette plate-forme est en cours de réalisation en prenant comme base le modèle de JavaBeans proposé par Sun. La première motivation était de « confronter » le modèle de composant Maleva à un autre modèle de composant, plus souple que celui de Delphi. Le modèle componentiel décrit dans cet article est étendu afin de fournir une plate-forme de conception mais également d'exécution évoluant dans un environnement dynamique. La première version de la plate-forme (en Delphi) permettait la conception de nouveaux composants, mais obligeait la recompilation de l'ensemble (plate-forme + nouveaux composants) à chaque modification du code des composants. Seule la composition (i.e. la conception d'agents et de composants composites) était autorisé à l'exécution. Ces restrictions provenaient principalement du langage Delphi qui n'autorisait pas l'adjonction de code (via le chargement dynamique) à l'exécution. Nous souhaitons étendre l'exemple des fourmis décrit ci-dessus en fournissant à cette nouvelle version de la plate-forme 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).

De plus, la notion de typage des bornes des composants est ajoutée au modèle (la version Delphi ne proposait qu'un typage faible des composants Maleva, la séparation contrôle/donnée n'étant que purement conceptuelle). Par exemple, le composant *Mvt-Aléatoire* décrit dans la section 3.1 produit une donnée de type DÉPLACEMENT (qui est donc le type de la borne de sortie de ce composant). Java permet également de faire remonter beaucoup d'informations sur un composant via les outils d'introspection de composants. Ainsi, le concepteur d'architectures d'agents peut, simplement en interrogeant un composant, obtenir ses informations internes. Le typage des connexions permet également de rationaliser le graphe d'interconnexion et les notions d'héritage de types permettent de définir des composants génériques pouvant être réutilisés dans des contextes cohérents (la garantie de définir un schéma de connexion cohérent entre composants est partiellement assurée par le typage des bornes des composants).

Enfin, nous axons nos travaux sur les possibilités que fournit ce modèle de composant en terme de pouvoir d'expression de politiques d'ordonnancement de tâches. Les Beans s'exécutent principalement de manière séquentielle : ils communiquent selon un modèle événementiel en *publish/subscribe* mais d'un point de vue de l'implantation, se limitent lors de la diffusion d'un événement à un simple appel de méthode. L'extension que nous proposons est d'encapsuler chaque Beans dans une capsule permettant l'exécution concurrente du code de chacun d'eux. Pour cela, nous ajoutons à chaque Beans une boîte aux lettres (*mailbox*) semblable à celles que les modèles d'acteurs utilisent [AGH 86].

Enfin, via ces outils de typage et de composition visuelle, nous espérons pouvoir identifier plus facilement des schémas de conceptions récurrents, qui une fois définis permettent 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 de faciliter grandement la conception d'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 performances) 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 représentée par un composant unique (donc équivalent à du code exécutable), dans ce cas les performances ne souffrent pas du poids de la décomposition.

La plate-forme MALEVA, et plus généralement le modèle componentiel qu'elle fournit peut donc être utilisée de deux manières différentes : soit comme plate-forme de validation afin de tester l'architecture générale des agents et de vérifier la cohérence des différents modules avant de passer à une implantation brute en code exécutable, soit comme plate-forme 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-composants n'est pas exploitée).

5. Travaux connexes

5.1. *Le monde des composants*

De nombreux modèles à base de composants ont été proposés par les domaines de recherche orientés «industriel», tels que les JavaBeans, le COM, les EJB, etc. Toutefois l'ensemble de ces propositions orientent principalement leurs approches sur un axe d'implémentation.

Depuis 1995, Microsoft développe un ensemble de technologies permettant la composition d'applications à partir de briques logicielles développées séparément. Dans ces technologies, COM et dérivés [MIC 97] privilégient l'approche des composants binaires : l'utilisateur n'a aucun accès au code des composants qu'il manipule. Le modèle proposé est donc extrêmement simple, principalement orienté implémentation via des mécanismes précis (par exemple la découverte d'interfaces via *IUnkown*). Très peu de concepts peuvent être extraits du modèle au niveau conception.

Les JavaBeans [SUN 97], développés par Sun Microsystems, représentent une sorte de contrepoint à l'approche Microsoft. Inspiré par les architectures logicielles, ce modèle de composant privilégie l'extraction de la relation de couplage via l'utilisation d'un mode de communication de type implicite, basé sur les événements. Un Beans n'identifie pas directement le destinataire des événements qu'il émet, c'est le concepteur d'application qui explicite ces dépendances. Point important, le modèle de mise en œuvre proposé est clairement séparé du modèle conceptuel.

Il est intéressant de remarquer que le modèle de composant proposé par les JavaBeans nous a inspirés pour concevoir une nouvelle implémentation de la plate-forme Maleva. Le modèle de communication événementiel des JavaBeans a été étendu car trop restrictif. Mais parallèlement, Sun a proposé une nouvelle technologie de communication entre Beans, InfoBus, qui rejoint partiellement nos travaux, notamment en terme d'adapteurs ou de composants de contrôle [SUN 99] [6]. La version initiale des Beans fournissait un premier niveau de découplage entre Beans via les événements, mais le contrôle restait très dépendant entre Beans (lorsqu'un Beans envoyait un événement, il fournissait également son flot de contrôle permettant l'activation du récepteur). L'approche Maleva permet d'éviter ce genre d'inconvénients via la réification (extraction/accessibilité) du contrôle entre composants. De même, Sun commence à réfléchir sur les notions de hiérarchisation de composants (ou composants composite), mais encore d'un point de vue implémentation et ne propose que peu de choses au niveau modèle [SUN 98].

5.2. *Plates-formes de conception d'agents*

Peu de frameworks utilisent un modèle componentiel pour caractériser l'architecture des agents, l'exploitation de la modularité par ces plates-formes s'opérant essentiellement au niveau du système multi-agent et peu au niveau agent [MOO 95]. Pour des descriptions détaillées de telles plates-formes modulaires, le lecteur pourra se référer à [BRE 98], [DEC 96], [BRA 95], et plus particulièrement 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-composants et en ajoutant des mécanismes de contrôle intra-composants. Contrairement à Maleva, ce framework ne sépare pas explicitement flot de contrôle et flot de données. De plus, un composant JAF peut référencer des ser-

6. L'introduction de composants de contrôle dans des modèles à base de composants est également faite dans [TAR 99]. L'idée d'extraire des informations hors des composants (dans notre cas le contrôle) et d'adjoindre au modèle des composants dédiés à la manipulation de ces informations semble donc de plus en plus présente dans les travaux de recherche aujourd'hui.

vices 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 dépendances, 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 fourni 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 a 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 utilisant le modèle de conception proposé par MALEVA. C'est d'ailleurs un de nos projets en cours. Ainsi nous souhaitons fournir au concepteur d'architecture d'agents deux types de bibliothèques :

- Une bibliothèque de composants élémentaires fournissant aussi bien des composants de type *agents* (représentant la partie « comportement commun » à plusieurs familles d'agents) mais également la partie « système » des agents (par exemple des composants de perception, de migration, d'interaction, etc).

- Une bibliothèque de graphes d'interconnexion représentant des schémas d'interconnexion récurrents au sein d'une même famille d'agents, ou plus particulièrement représentant des schémas de concurrence récurrents entre différents composants et pouvant être ré-exploités dans différents contextes. Ceci équivaut à des petits frameworks « à trou » d'agents qu'on instanciera en insérant des composants spécialisés.

6. 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 d'architecture d'agents et du concepteur de composants élémentaires 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.

De manière plus générale, nous présentons ici un modèle de composant pour la conception et l'implémentation d'agents. Au niveau conception, la distinction flot de contrôle/flot de données permet de rationaliser la spécification de composants en séparant les contraintes inhérentes aux fonctionnalités propres de chaque composant de la manière dont ils vont s'activer dans l'architecture de l'agent. Principalement ce modèle est orienté simulation de systèmes multi-agents. La spécification de différentes séquences d'activation est très importante dans ce cadre d'application. Au sein d'un agent les composants doivent pouvoir être synchronisés facilement afin de conserver un état global cohérent. De plus, la hiérarchisation permet de donner de « bonnes propriétés » aux architectures d'agents créées, notamment en fournissant des outils de composition de haut niveau. Par exemple, elle permet de concevoir à moindre coût de nouvelles architectures d'agents (et permet l'encapsulation classique du monde objet). Rendre accessible le contrôle au niveau de la composition devrait autoriser également l'utilisation de composants dans des cadres d'applications beaucoup plus vastes que ceux dans lequel ils avaient été initialement pensés.

7. Bibliographie

[AGH 86] AGHA G., *Actors : a model of concurrent computation in distributed systems*, Series in Artificial Intelligence, MIT Press, 1986.

[BRA 95] BRAZIER F., DUNIN-KEPLICZ B., JENNINGS N., TREUR J., « Formal Specification of Multi-Agent Systems : a Real-World Case », *Proceedings of the First International Conference of Multi-Agent Systems*, 1995, p. 25-32, San Fransisco, CA, MIT Press.

[BRE 98] BRENTON A., WOOD T., « JACK Intelligent Agents User Guide », 1998, ©Agent Oriented Software Pty Ltd.

[DEC 96] DECKER K., SYCARA K., « Designing Reusable Behaviors for Information Agents », rapport, Janvier 1996, The Robotics Institute, Carnegie-Mellon University.

[DRO 93] DROGOUL A., « De la Simulation Multi-Agent la Résolution collective de Problèmes », Thèse de doctorat, Université Paris 6, 1993.

[GUE 96] GUESSOUM Z., « Un environnement opérationnel de conception et de réalisation de systèmes multi-agents », Thèse d'université, Université Paris VI, Mai 1996, LAFORIA TH96/09.

[GUI 98] GUILLEMET A., HAÏK G., « Evaluation d'une Architecture Componentielle de Conception d'Agents », Rapport de Stage de 1ère année de Magistère d'Informatique et Modélisation, Ecole Normale Supérieure de Lyon, Septembre 1998.

[HOR 98] HORLING B. C., « A reusable Component Architecture for Agent Construction », Technical Report n° 1998-49, Octobre 1998, UMASS Computer Science.

[JEN 00] JENNINGS N. R., WOOLDRIDGE M., « Agent-Oriented Software Engineering », BRADSHAW J., Ed., *Handbook of Agent Technology*, AAAI/MIT Press, 2000.

[LAW 00] LAWSON B. G., PARK S., « Asynchronous Time Evolution in an Artificial Society Mode », *Journal of Artificial Societies and Social Simulation*, vol. 3, n° 1, 2000.

[LEC 97] LECERF V., PINTADO M., « An adaptive model of Camera-Driven Urban Intersections Observation », *Workshop on dynamic scene recognition*, 1997.

[LHU 98] LHUILLIER M., « Une Approche à Base de Composants Logiciels pour la Conception d'Agents. Principes et Mise en Oeuvre à Travers la Plate-forme Maleva », Thèse d'université, Université Paris 6, Février 1998.

[MAR 87] MARCA D. A., MCGOWAN C. L., *SADT Structured Analysis and Design Technics*, McGraw-Hill, New York, New York, 1987.

[MIC 97] Microsoft Corp., « Microsoft COM Technologies », 1997, `http ://www.microsoft.com/com`.

[MOO 95] MOORE B., NOSCHANG M., PENIX J., « An Annotated Bibliography of Software Agent Technology », 1995.

[PES 00] PESCHANSKI F., MEURISSE T., BRIOT J.-P., « Les composants logiciels : Evolution technologique ou nouveau paradigme ? », *Actes de la conférence OCM 2000*, 2000, p. 53-65.

[SHA 96] SHAW M., GARLAN D., *Software Architecture : Perspective on an Emerging Discipline*, Prentice Hall, 1996.

[SHE 98] SHEHORY O., « Architectural Properties of Multi-Agent Systems », rapport n° CMU-RI-TR-98-28, Décembre 1998, The Robotic Insitute.

[SUN 97] Sun Microsystems Inc., « JavaBeans Specification v1.01 », 1997, `http ://www.java.sun.com/beans/doc/spec.html`.

[SUN 98] Sun Microsystems Inc., « The JavaBeans Runtime Containment and Services Protocol Specification v1.0 », 1998, `http ://www.java.sun.com/beans/glasgow/beancontext.pdf`.

[SUN 99] Sun Microsystems Inc., « InfoBus 1.2 Specification », 1999, `http ://www.java.sun.com/beans/infobus/infobus1.2.pdf`.

[TAR 99] TARPIN-BERNARD F., DAVID B., « AMF : un modèle d'architecture multi-agents multi-facettes », *Technique et Science Informatique (TSI)*, vol. 18, n° 5, 1999, page 555 to 586.

[WOO 00] WOOLDRIDGE M., JENNINGS N. R., KINNY D., « The Gaia Methodology for Agent-Oriented Analysis and Design », *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 3, n° 3, 2000.

Article reçu le 8 Novembre 1999

Version révisée le 16 Juin 2000

Rédacteur responsable : Christophe Dony

Thomas Meurisse *est étudiant au Laboratoire d'Informatique de Paris 6 (LIP6). Il est actuellement allocataire de recherche et prépare une thèse au sein du thème OASIS. Ses activités de recherche concernent les environnements de développement d'agents, et plus particulièrement les architectures d'agents.*

Jean-Pierre Briot *est directeur de recherche au CNRS, membre du Laboratoire d'Informatique de Paris 6 (LIP6). Il y est responsable du thème de recherche « Objets et Agents pour les Systèmes d'Information et de Simulation » (OASIS), au carrefour du génie logiciel, de l'intelligence artificielle, et de l'informatique répartie.*