Une expérience de conception et de composition de comportements d'agents à l'aide de composants

Jean-Pierre Briot — Thomas Meurisse — Frédéric Peschanski

Laboratoire d'Informatique de Paris 6 (LIP6)
Université Paris 6 - CNRS
4 place Jussieu, F-75252 Paris cedex 05
{Jean-Pierre.Briot,Thomas.Meurisse,Frederic.Peschanski}@lip6.fr

RÉSUMÉ. *Cet article présente une expérience d'utilisation d'un modèle de composant logiciel pour concevoir et construire des agents. Après avoir discuté de diverses approches de décomposition d'une architecture d'agents, nous présentons notre modèle de composants d'agents. Dans ce modèle, nommé MALEVA, les composants encapsulent différents comportements de l'agent (ex. fuir, suivre un gradient...). Parmi les spécificités du modèle, on peut noter une notion de composant composite, encapsulant des comportements construits par assemblage de comportements plus simples, et une notion explicite de flot de contrôle entre les composants (réifiée à travers des bornes, connexions et composants spécifiques), apportant un contrôle fin de l'activation. Plusieurs exemples de conception sont présentés dans l'article.*

ABSTRACT. *This paper relates an experience in using a component model to design and construct agents. After discussing various approaches for decomposing an agent architecture, we describe our component-based model of agents. In this model, named MALEVA, components encapsulate various units of agent behaviors (e.g., flee, follow gradient...). Among its specificities, we may note a notion of composite component, encapsulating behaviors constructed by assembling simpler behaviors, and an explicit notion of control flow between components (reified through specific control ports, connexions and components), for a fine grain control of activation and scheduling. Several design examples are presented in this paper.*

MOTS-CLÉS : *composant, comportement, agent, composition, architecture, contrôle, simulation.*

KEYWORDS: *component, behavior, agent, composition, architecture, control, simulation.*

1. Introduction

Les composants (Bachman *et al.*, 2000) et les systèmes multi-agents (Briot *et al.*, 2001) sont des approches de conception et de développement de logiciel ayant actuellement un grand impact. Toutes deux proposent des abstractions pour organiser le logiciel comme une combinaison d'éléments logiciels, avec pour objectif commun de faciliter son évolution (en premier lieu, remplacement et ajout d'éléments). Nous considérons que les systèmes multi-agents repoussent encore plus loin le niveau d'abstraction et la flexibilité du couplage entre composants (Briot, 2006), notamment à l'aide de leurs capacités d'auto-organisation. Cependant, nous pensons que les concepts et la technologie des composants logiciels peuvent aussi aider à la construction des systèmes multi-agents. Le potentiel de fertilisation croisée est ainsi :

– *apport des agents aux composants* : pour concevoir des applications à base de composants plus autonomes et flexibles, par exemple en utilisant des techniques de mise en correspondance et de négociation pour une assistance à l'assemblage,

– *apport des composants aux agents* :

- *au niveau du système multi-agent* : pour une aide à l'intégration, au « packaging » et au déploiement des systèmes multi-agents (par ex. (Melo *et al.*, 2004)),

- mais également *au niveau d'un seul agent* : en aidant à structurer et réutiliser l'implantation de son architecture.

Cet article se concentre sur ce dernier type d'apport. En effet, nous pensons que la conception et la construction d'un agent individuel peut bénéficier des principes des composants logiciels (encapsulation, interfaces de sortie, connecteurs explicites...). Notre objectif est d'aider à une conception incrémentale des agents comme la composition de comportements et activités plus élémentaires (par ex. suivi de gradient, fuite, reproduction...), encapsulés dans des composants. Une des originalités de notre modèle de composant, nommé MALEVA, est qu'il applique les principes des composants et de composition logicielle à la spécification du contrôle, *via* des notions de bornes, connexions, et composants de contrôle.

Jusqu'à maintenant, notre principal domaine d'application a été la simulation multi-agent de phénomènes (biologique, écologique, social, économique...), et les exemples et les outils présentés ici reflètent cette expérience. Cependant, nous pensons que le modèle de composant d'agent MALEVA possède une portée plus générale.

La section 2 compare différents principes de décomposition d'une architecture d'agent (par cycle, par niveaux, par traitements...). La section 3 introduit le modèle de composant d'agents MALEVA, dans lequel chaque comportement est un composant. La section 4 introduit brièvement le domaine d'application privilégié de la simulation multi-agent. Les sections 5, 6 et 7 décrivent successivement trois différents exemples de conception d'agents à partir de composants. De plus, la section 7 détaille la question importante de l'ordonnancement des différents comportements. La section 8 résume les outils de l'environnement MALEVA et les principes d'implantation. La section 9

complète une comparaison aux travaux du domaine, déjà entamée à la section 2. La section 10 présente quelques perspectives avant de conclure cet article.

2. Critères de décomposition et styles architecturaux d'une architecture d'agent

Une architecture d'agent est la structure logicielle (ou matérielle) qui, à partir d'un ensemble d'entrées, produit un ensemble d'actions sur l'environnement ou sur les autres agents. Sa description est constituée des composants (correspondant aux fonctions) de l'agent et des interactions entre ceux-ci (flux de données et de contrôle). (Boissier, 2001, p. 73).

Le terme *fonctions* (de l'agent), peut correspondre à différents points de vue : types de traitements, niveaux, comportements... Inspirés par les travaux autour du concept d'architecture logicielle de Shaw et Garlan (Shaw *et al.*, 1996), nous proposons une tentative de classification des architectures d'agents, et de leurs principes de décomposition, selon la perspective des *styles architecturaux*.

Notez qu'il ne s'agit pas de la seule typologie possible, et on peut en trouver d'autres dans la littérature (Boissier, 2001) (Müller, 1997) (Wooldridge *et al.*, 1995), par ex. architectures horizontales/verticales, ou encore réactives/cognitives/hybrides. Dans cet article, nous nous focalisons sur les principes ou critères de décomposition en nous intéressant à leur impact sur la réutilisabilité de l'architecture et des composants. Il faut noter que notre typologie ne prétend pas être exhaustive et nous ne détaillons pas ici non plus les différentes architectures servant d'exemples à notre classification, un bon état de l'art sur les architectures d'agents se trouvant par exemple en (Boissier, 2001). Enfin, notez que, comme pour les architectures logicielles (Shaw *et al.*, 1996), une architecture complexe (par ex. InteRRaP, voir la section 2.3) peut juxtaposer et combiner différents styles architecturaux.

2.1. *Décomposition selon le cycle d'activation*

Ce principe de décomposition associé, et le style architectural associé, un des plus simples, suit le cycle de base d'un agent situé dans un environnement : perception (de l'environnement), mise à jour de l'état (mental ou/et données), génération des engagements (d'actions), action. Un premier exemple est une architecture minimale d'agent réactif situé (détaillée à la section 4.2, voir la figure 7). Un autre, cette fois pour un agent cognitif et communicant, est l'architecture agent-oriented programming (AOP), de Yoav Shoham (Shoham, 1993) (voir la figure 1).

2.2. *Décomposition selon les points de vue et traitements associés*

Une autre forme de décomposition, plus stucturelle, repose sur une décomposition de l'architecture de l'agent suivant différents points de vue ou facettes (ex. interac-

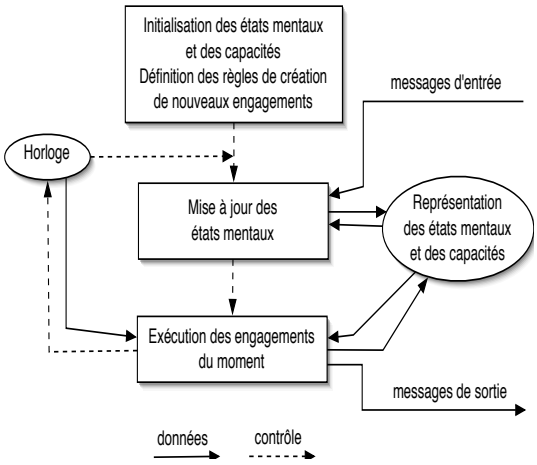

Figure 1. *Architecture AOP*

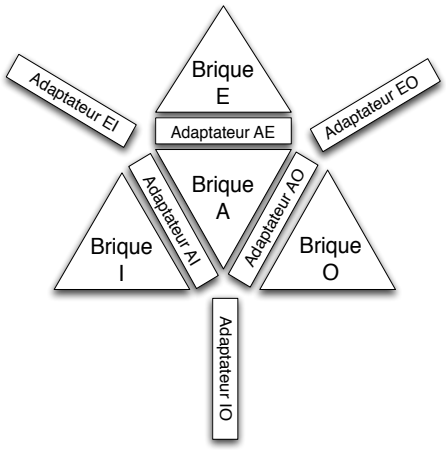

Figure 2. *Architecture VOLCANO*

tion, environnement, organisation…) et différents types de traitements associés (ex. perception, communication, coordination…). Un exemple représentatif est l'architecture VOLCANO (Ricordel *et al.*, 2001), qui suit les principes de décomposition de la méthodologie Voyelles (Demazeau, 1995) en quatre dimensions : A (agent), E (environnement), I (interaction) et O (organisation). Cette architecture est en fait un framework, dont les composants correspondants (appelés briques) sont en conséquence : A, E, I et O. La figure 2 illustre la position centrale de la brique A, entourée des briques E, I et O, et des 6 adaptateurs (*wrappers*) interbriques : AE, AI, AO, EI, EO et IO.

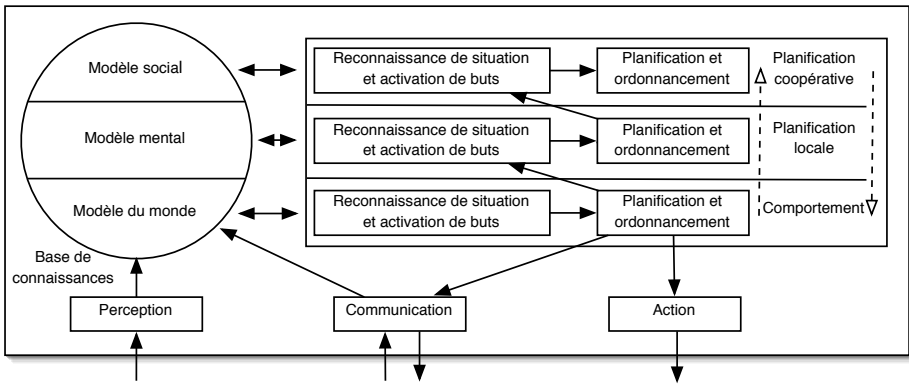

Figure 3. *Architecture InteRRaP*

L'architecture MAST (Vercouter, 2004) prolonge ce travail, en ajoutant la dimension U, interface avec l'utilisateur (*user*), en permettant une redécomposition plus fine à l'intérieur de chaque dimension, et enfin en se fondant sur un vrai mécanisme de diffusion d'événements, qui apporte de l'implicite aux connexions entre les composants.

Un autre exemple est le modèle d'agents et de systèmes multi-agents à base de composants DESIRE (Brazier *et al.*, 1995). Ce projet a en particulier conçu un modèle d'architecture générique (generic agent model : GAM) (Brazier *et al.*, 2001), décomposé en un certain nombre de composants (gestion de l'interaction, gestion de l'information sur le monde...). Ce modèle (là aussi un framework) a été validé par une rétro-conception d'un certain nombre de modèles d'architectures d'agents générales ou appliquées (tels que BDI et ARCHON) (Brazier *et al.*, 2001).

2.3. *Décomposition selon les niveaux et les modèles*

Il est aussi possible de considérer différents niveaux et modèles de connaissances, de raisonnement et d'action, pour structurer l'architecture, notamment par une distinction entre le modèle du monde, le modèle de soi, et enfin le modèle social. Un exemple représentatif est l'architecture InteRRaP (Müller *et al.*, 1993), qui distingue ces trois niveaux (voir la figure 3).[1]

1. On peut remarquer que l'architecture de chaque niveau est similaire. Il est également clair que, comme pour les styles architecturaux (Shaw *et al.*, 1996), une architecture élaborée (telle qu'InteRRaP) peut faire cohabiter et combiner différents principes de décomposition.

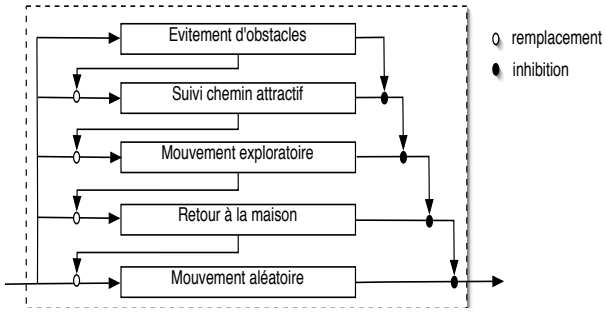

Figure 4. *Architecture de subsomption*

2.4. *Décomposition selon les comportements*

Une décomposition plus radicale, et plus ascendante, considère les comportements de l'agent comme les unités de (dé)composition. Un exemple est l'architecture de subsomption de Rodney Brooks (Brooks, 1986), dans laquelle différents comportements (ex. mouvement aléatoire, évitement d'obstacle...) sont activés simultanément. Ils sont organisés selon une hiérarchie forte et figée et les priorités associées (voir la figure 4). En pratique, un comportement peut en effet remplacer les données d'entrée du comportement immédiatement inférieur et également inhiber ses données de sortie (par exemple, en cas de détection d'obstacle, le comportement d'évitement d'obstacles peut prendre la main).

2.5. *Discussion*

Tentons un premier bilan de ces principes de décomposition des architectures et, c'est ce qui nous intéresse au premier lieu dans l'approche des composants, de leur capacité d'évolution et de réutilisation de composants. Ces architectures sont en général plus ou moins appropriées à un certain type de modèle d'agent (ex. agent cognitif collaboratif pour l'architecture InteRRaP, agent situé ou robot pour l'architecture de subsomption...). Elles ont pour ambition d'être génériques mais peuvent être insuffisamment flexibles. Ainsi, il est en pratique souvent assez difficile, voire impossible, de remplacer ou d'ajouter des composants. D'ailleurs, l'implantation de l'architecture ne repose souvent pas sur des principes de composants logiciels (interfaces de sortie, connecteurs explicites...) ni des modèles de composants classiques (tels que JavaBeans). L'architecture VOLCANO sépare bien les briques, mais remplacer une brique par une autre oblige en général à reprogrammer les adaptateurs liés à cette brique, ce qui déplace en partie la question. L'architecture MAST représente de ce point de vue un net progrès, du fait de son modèle d'implantation à base d'événements et de la décomposition des briques en sous-composants.

De manière plus générale, l'existence même d'une architecture restreint la combinatoire de combinaison de composants, ce qui est d'ailleurs normal et même souhaité. L'architecture de subsomption est plus abstraite, et ne représente en fait qu'un modèle d'architecture, instancié pour un robot et un objectif donnés. Mais une fois instanciée (ex. voir la figure 4), il est ensuite très difficile de la faire évoluer (rajouter un comportement), car la hiérarchie est la clé des rapports entre les composants. Une architecture telle que VOLCANO est un framework avec des composants abstraits à instancier bien identifiés, mais avec également des adaptateurs à implanter.

Une option radicale est alors de ne proposer qu'un modèle de composant, de manière analogue à un modèle de composant logiciel tel que JavaBeans, l'architecture d'agent générale proprement dite disparaissant alors. La conception d'un agent suit alors une approche compositionnelle relativement libre, à partir de composants briques de base.

2.6. *Décomposition selon les comportements revisitée*

Nous pensons que le principe de décomposition selon les comportements, tel qu'utilisé dans l'architecture de subsomption (section 2.4) est assez radical, mais également le plus proche du concept même que l'on veut souvent pouvoir décomposer : le comportement de l'agent. L'architecture de subsomption, en câblant implicitement le contrôle entre composants de comportements, à partir de la hiérarchie, est concise mais assez inflexible et en pratique difficile à mettre au point.

Une alternative est alors de ne proposer qu'un modèle de composant de comportement à assembler librement. Le contrôle hiérarchique figé de l'architecture de subsomption est alors à remplacer par quelque chose de plus flexible. Nous proposons (section 3.1) d'expliciter et de réifier le contrôle, en utilisant les concepts mêmes de composants : bornes, connexions, et composants de contrôle. Comme nous le verrons, ceci permet ainsi de construire un graphe explicite de contrôle, représentant le flot de contrôle entre les composants.

3. Le modèle de composant d'agent MALEVA

Le modèle de composant d'agent MALEVA a été à l'origine conçu au milieu des années 1990 par Marc Lhuillier, dans le cadre de sa thèse (Lhuillier, 1998). Il a ensuite été raffiné et réimplanté (voir la section 8) par Thomas Meurisse, également dans le cadre d'une thèse (Meurisse, 2004).

Comme expliqué précédemment, l'objectif du modèle de composant d'agent MALEVA est de permettre une conception, et une construction, modulaires du comportement d'agent souhaité, comme un assemblage de comportements (encapsulés dans des composants) plus élémentaires.

	Données	*Contrôle*
Borne d'entrée	Consommation de données	Point d'entrée d'activation
Borne de sortie	Production de données	Point de sortie d'activation
Connexion	Transfert de données	Transfert d'activation

Tableau 1. *Bornes de données et de contrôle*

3.1. *Distinction entre flot de données et flot de contrôle*

MALEVA introduit une distinction entre le flot de données et le flot de contrôle entre les composants. Cette caractéristique, et spécificité de notre modèle,[2] permet de concevoir de manière explicite l'architecture de contrôle. Comme nous le verrons à la section 3.2, cette dissociation de l'architecture fonctionnelle des composants du contrôle de leur activation, permet une plus grande généricité des composants, que l'on peut ainsi plonger dans différents contextes d'exécution.

Par voie de conséquence, nous considérons deux types de bornes (*ports*) pour un composant :

– *bornes de données* : elles constituent les points d'entrée ou de sortie des données pour un composant encapsulant un comportement. Notez que les bornes de données sont typées (ce point est détaillé à la section 8.3).

– *bornes de contrôle* : un comportement encapsulé dans un composant est activé à la réception d'un signal d'activation sur sa borne d'entrée du contrôle. Une fois l'exécution du comportement terminée, le signal d'activation est transmis *via* la borne de sortie du contrôle. Ceci permet notamment de spécifier une séquence d'activation de comportements.

En plus de la distinction *sémantique* entre bornes de données et bornes de contrôle, spécifique à MALEVA, nous retrouvons la distinction standard (comme pour tout modèle de composant), et cette fois *structurelle*, entre bornes d'entrée et bornes de sortie (tableau 1).

3.2. *Un exemple introductif*

La figure 5 illustre un premier et très simple exemple d'assemblage/composition de deux composants : en séquence. Le composant `B` est donc activé après que le composant `A` en ait lui-même terminé. En conséquence, cette fois du point de vue des données, le composant `B` consommera les données produites par le composant `A` seule-

2. Bien que dans un contexte sensiblement différent, une distinction entre flot de données et flot de contrôle avait déjà été également introduite dans la méthode SADT (Marca *et al.*, 1987), cependant principalement au niveau de la conception, et pour des applications de gestion de projets.

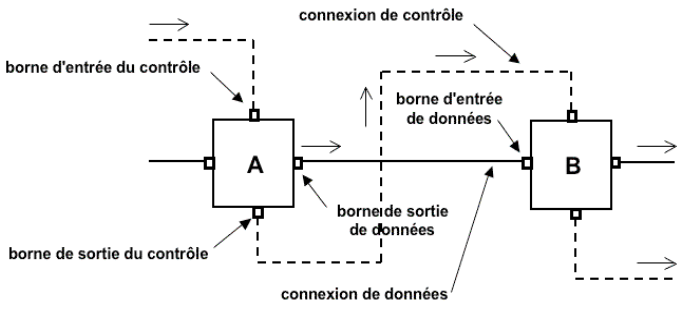

Figure 5. *Activation séquentielle de deux composants*

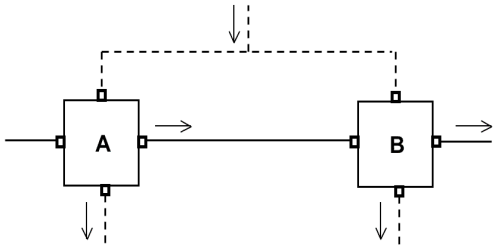

Figure 6. *Activation concurrente de deux composants*

ment après que `A` ait terminé son activation. Pour ce schéma, et pour tous les schémas suivants, les connexions de données sont en trait plein, et les connexions de contrôle en pointillés.

La figure 6 recompose les deux mêmes composants, `A` et `B`, mais cette fois ils sont activés simultanément. Notez que les connexions de contrôle ont été modifiées dans ce but, mais les connexions de données sont restées inchangées. La sémantique de cette activation simultanée (concurrente) est analogue au style architectural *pipes and filters* (Shaw *et al.*, 1996), dans lequel `B` consomme les données produites par `A`, les deux étant simultanément actifs.

Ce simple exemple peut déjà faire entrevoir la relative flexibilité de l'expression du contrôle et de l'activation de notre modèle de composants. Le contrôle étant spécifié à l'extérieur des composants, cela apporte une plus grande généricité dans leur utilisation dans différents contextes d'activation. On peut ainsi aussi bien concevoir des composants autonomes (avec un thread associé), un séquencement explicite ou toute autre forme de combinaison.[3] Ceci permet également un contrôle fin des politiques d'activation et donc d'ordonnancement des composants/comportements d'un même

3. Un aspect complémentaire, le modèle d'activation des composants, qui peut être asynchrone ou synchrone, est abordé succintement à la section 8.2.

agent, indépendamment de leurs fonctionnalités. Cet aspect en pratique fort utile, sera abordé plus avant à la section 7.

3.3. *Concevoir et construire des comportements d'agents*

Concevoir et construire des comportements d'agents pour une application donnée consiste donc en l'assemblage et la définition de composants de comportements. Ceci suppose l'existence de bibliothèques de composants de comportements adaptés au domaine d'application considéré. Un composant peut être primitif (le comportement est alors implanté dans le langage support, en pratique Java), ou bien *composite*,[4] et encapsule et identifie alors une composition de composants.

Comme pour la conception logicielle générale, deux approches alternatives de conception peuvent être suivies :

– *approche ascendante* : construire incrémentalement des composants composites représentant des comportements plus évolués par assemblage de composants représentant des comportements plus élémentaires ;

– *approche descendante* : débuter par une spécification du comportement souhaité, et le décomposer étape par étape jusqu'à des comportements élémentaires.

Si l'approche ascendante est peut-être plus intuitive et plus aisée à mettre en œuvre à l'aide d'une interface graphique de conception, elle ne garantit pas la conformité du composant produit à ses spécifications. De plus elle nécessite une bibliothèque de composants suffisamment riche. De manière symétrique, rien n'assure que la décomposition descendante progressive d'un comportement cible va aboutir 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. En pratique, une conception repose souvent sur des alternances (allers-retours) entre démarches ascendante et descendante.

4. Exemples d'applications en simulation multi-agent

Nous allons maintenant introduire le contexte de la simulation multi-agent, domaine des différentes études de cas présentées ici.

---

4. Nous considérons que la notion de composant composite est importante et souhaitable (Peschanski *et al.*, 2000). Elle correspond à une notion de *composition structurelle* à la différence, ou plutôt en supplément, d'une *composition fonctionnelle* (simple assemblage). L'encapsulation et la hiérarchisation offertes par la notion de composant composite sont utiles pour aider à maîtriser la complexité.

4.1. *Simulation multi-agent*

Le domaine d'application privilégié de notre modèle de composant d'agent est la simulation multi-agent de phénomènes (biologiques, écologiques, sociaux, économiques…), dans laquelle les différents éléments constituant et caractérisant les phénomènes ainsi que leurs interactions sont explicitement modélisés (Moss *et al.*, 2001). Il est utile de noter dès à présent que les concepteurs de modèles et de simulations ne sont pas toujours nécessairement des programmeurs experts (nous verrons notamment à la section 7 comment les composants, ainsi que la spécification distincte de leur contrôle, peuvent aider ces concepteurs). De plus, ils veulent en général pouvoir rapidement prototyper et mettre à jour les caractéristiques comportementales des différents agents qu'ils identifient dans une simulation.

MALEVA a d'ailleurs été utilisé directement ou comme source d'inspiration dans divers projets de simulation multi-agent, notamment appliqués à : la migration urbaine (Vanbergue, 2003), la simulation de trafic automobile (LeCerf *et al.*, 1997), et l'évolution de populations de poissons.

Le premier exemple présenté suit une approche ascendante pour construire des comportements d'animaux de type proie puis de type prédateur, agents situés dans un éco-système. Le deuxième exemple suit une approche descendante pour concevoir des comportements d'agents fourmis, également situés. Enfin, le troisième et dernier exemple, simulant une dynamique de population, nous permet d'illustrer comment MALEVA permet un contrôle fin de l'ordonnancement des différents comportements d'un agent (cette fois, non situé) de manière à mieux contrôler les risques de biais.

4.2. *Le modèle de base d'un agent situé*

Pour une certaine proportion de simulations multi-agent (et d'ailleurs également pour d'autres types d'applications, par exemple en robotique), les agents sont *situés* dans un environnement, qu'ils perçoivent à travers leurs capteurs, et sur lequel ils peuvent agir et produire des effets. Le cycle de calcul (à chaque pas de simulation) est habituellement le suivant : l'agent perçoit *via* ses capteurs son environnement (en particulier, la présence d'autres agents à proximité, d'obstacles, de phéromones…) ; ces données sont traitées par son comportement (interne) ; pour produire, *via* ses actionneurs, des actions sur l'environnement (ex. déplacement, prise de nourriture ou d'objet, dépôt de phéromone…). Le modèle général d'un *agent situé* suit ainsi le cycle suivant :

$$\textit{capteurs} \rightarrow \textit{comportement} \rightarrow \textit{actionneurs}$$

et en conséquence, le modèle d'architecture abstrait illustré à la figure 7.

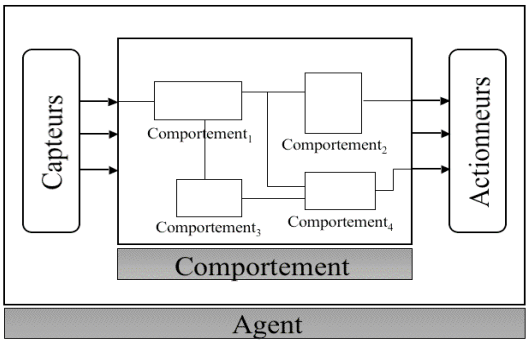

Figure 7. *Architecture générale d'un agent situé*

5. Un premier exemple : conception ascendante d'une proie et d'un prédateur

Ce premier exemple consiste à concevoir des comportements de type proie et prédateur, pour des animaux virtuels, agents situés dans un ecosystème. En adoptant une approche ascendante, nous définissons d'abord un ensemble de comportements élémentaires appropriés, à savoir `Flee` (fuir un prédateur), `Follow` (suivre une proie), et `Exploration` (exploration par déplacement aléatoire). Nous allons composer ces comportements élémentaires de manière à construire les comportements suivants : `Prey` et `Predator`.

5.1. *Comportement de proie*

Un comportement de proie (`Prey`) fuit les prédateurs situés à l'intérieur de son champ de perception. Si aucun prédateur n'est détecté, la proie explore son environnement en se déplaçant aléatoirement (mouvement d'exploration de base). Nous construisons le comportement de proie `Prey` comme la composition des trois comportements (composants) suivants : fuir (`Flee`), exploration (`Exploration`), et un composant de contrôle nommé `Switch`.

5.2. *Composants de contrôle*

Le composant de contrôle `Switch`[5] réifie la structure de contrôle conditionnelle en un composant primitif. La condition est ici la présence ou l'absence d'une donnée d'entrée (dans cet exemple, une donnée perçue, autrement dit un stimulus).

5. Notez que la bibliothèque standard MALEVA inclut plusieurs composants de contrôle, analogues des structures de contrôle standard (ex. boucle *repeat*) et des opérateurs de synchronisation (ex. barrière de synchronisation) (Meurisse, 2004). Ils ne sont pas détaillés dans cet article.

if une donnée a été reçue sur la borne `If` (borne d'entrée de données) **then**
  transférer le contrôle *via* (*i.e.*, activer) la borne `Then` (borne de sortie du contrôle)
  and transmettre la donnée *via* la borne `Then` (borne de sortie de données)
else
  transférer le contrôle *via* la borne `Else` (borne de sortie du contrôle)
end if

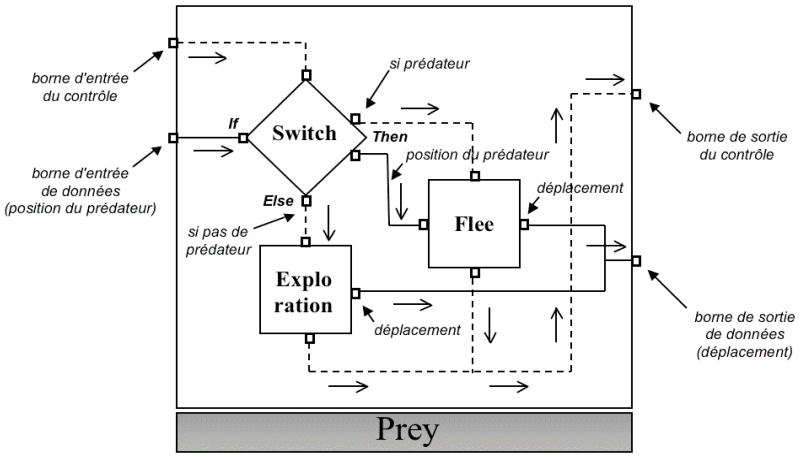

Figure 8. `Prey` *: comportement d'une proie*

L'architecture du comportement de proie suit cette logique (voir la figure 8).[6] Si un prédateur est détecté (dans ce cas, une donnée représentant la position du prédateur a été reçue sur la borne d'entrée de données du comportement `Prey`),[7] le composant de contrôle `Switch` transfère le contrôle *via* sa borne `Then`, ce qui active le comportement de fuite `Flee`. Ce comportement calcule un déplacement à partir de la position du prédateur (reçu *via* sa borne d'entrée de données) et transmet ce déplacement (mouvement de fuite de l'agent) *via* sa borne de sortie de données, et donc au final à la borne de sortie de données du composant `Prey`. Si aucun prédateur n'a été détecté (absence de données sur la borne d'entrée de données de `Prey`), le `Switch` transfère le contrôle *via* sa borne `Else`, ce qui active le comportement `Exploration`, qui calcule un déplacement transmis au final à la borne de sortie de `Prey`. Notez que `Exploration` n'a pas besoin de données d'entrée pour produire un déplacement aléatoire.

6. Pour rappel, les connexions du flot de données sont en trait plein, et les connexions du flot de contrôle en pointillés.
7. On suppose que les bornes d'entrée de données (perception d'un prédateur dans l'environnement) et de sortie de données (déplacement de l'agent) ont été connectées aux bornes correspondantes des capteurs et effecteurs, en suivant l'architecture générale d'un agent situé, illustrée à la figure 7.

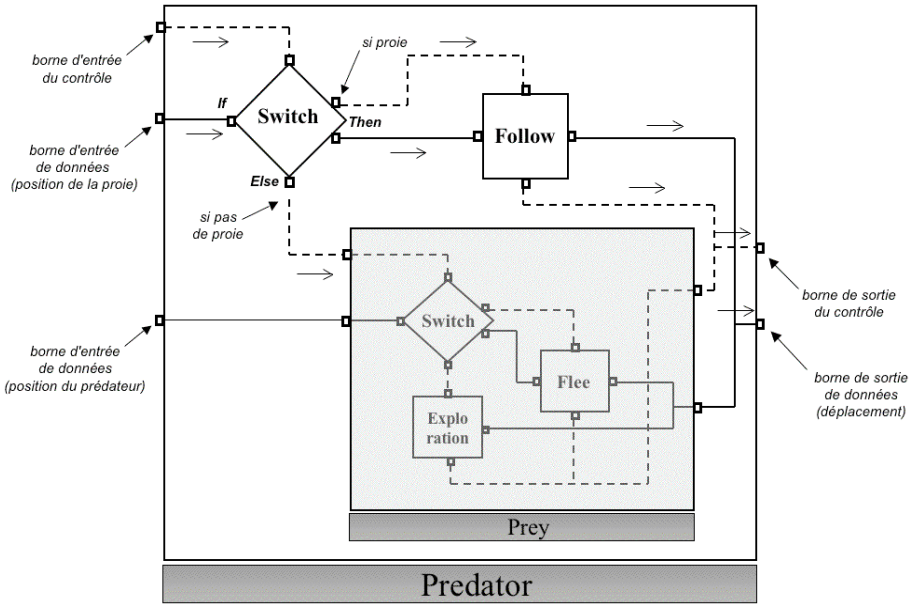

Figure 9. `Predator` *: comportement d'un prédateur (incluant un comportement de proie)*

5.3. *Comportement de prédateur*

Nous pouvons maintenant réutiliser le comportement de proie `Prey` pour concevoir et construire le comportement d'un prédateur `Predator` ainsi défini : il poursuit des proies, tout en fuyant ses congénères prédateurs, et a un comportement d'exploration aléatoire tant qu'il ne perçoit aucun autre agent. Un prédateur se réduit donc à un comportement de proie (il fuit les prédateurs et sinon se déplace aléatoirement) auquel est ajouté un comportement de prédation (poursuivre ses proies). En suivant notre approche compositionnelle, nous allons définir le comportement `Predator` comme un nouveau comportement composant composite encapsulant *tel quel* le comportement existant `Prey` (voir l'architecture résultante à la figure 9). Notez que dans cette conception, la faim (prédation) a priorité sur la peur (fuite), du fait que `Prey` est activé par `Predator`. D'autres combinaisons seraient également possibles.

6. Un deuxième exemple : conception descendante de comportements de fourmis

Ce deuxième exemple illustre une conception cette fois descendante de comportements d'agents. L'application complète a consisté en une reconception (*re-engineering*) en MALEVA (Guillemet *et al.*, 1998) d'un travail de modélisation et

de simulation de colonies de fourmis et de l'étude de leur sociogenèse (le framework MANTA d'Alexis Drogoul (Drogoul, 1993)). Différents types d'agents fourmis sont considérés : oeufs, larves,[8] fourmis ouvrières, reine... Nous nous restreindrons ici à la conception hiérarchique du comportement général de fourmi (fourmi ouvrière).

6.1. *Le patron Living*

La première étape de notre conception va identifier une caractéristique commune à chaque agent vivant, la capacité de vieillir (et ultimement de mourir). Nous allons en conséquence définir un comportement en partie abstrait, nommé `Living`, illustré à la figure 10. Il comprend quatre composants : `CheckAgeLimit`, qui gouverne le principe de vieillissement (il contient une variable `age` qui est incrémentée à chaque pas d'activation et comparée à la limite d'âge) ; `Die` qui a pour charge de faire mourir l'agent ; un comportement abstrait, nommé ici `Behavior` et représenté en grisé dans la figure ; et un composant de contrôle `Switch`. Quand l'agent meurt (parce qu'il a atteint sa limite d'âge), `CheckAgeLimit` émet une donnée `die`. Le `Switch` active alors `Die`, qui émet une donnée `suicide` à destination des actionneurs sur l'environnement (en pratique, retirer l'agent de l'environnement). Sinon (pas de donnée émise par `CheckAgeLimit`, l'agent est encore vivant), le comportement `Behavior` est activé par le `Switch`.

Il s'agit en définitive d'une forme simplifiée de *patron de conception* (*design pattern* (Gamma *et al.*, 1995)).[9] Le comportement abstrait implante ce patron comme une forme de « mini black-box framework », où le composant à spécialiser (*hot spot*) est le comportement abstrait `Behavior`. Pour construire un comportement spécifique d'agent, nous remplacerons (instancierons) ce comportement abstrait `Behavior` par un comportement spécifique, par exemple : fourmi, œuf, ou larve, comme nous allons le voir au paragraphe suivant.

---

8. Cette conception en MALEVA utilise le potentiel de reconfiguration des comportements (dynamicité architecturale) pour modéliser et implanter le processus de transformation menant d'un œuf à une larve, puis à une fourmi (Guillemet *et al.*, 1998). Ceci a été réalisé en considérant un composant de niveau « méta », appelé *métacomposant*, qui se comporte comme un « serveur de profils comportementaux ». En fonction de la requête issue d'un des composants de comportements de l'agent auquel il appartient, il va déterminer le nouveau profil comportemental, récupérer la liste des composants constituant l'agent, confronter ces informations pour établir les listes des composants à conserver et à détruire, et établir les connexions selon le nouveau profil. Pour des raisons de place, cette technique ne sera pas détaillée ici, voir (Guillemet *et al.*, 1998).

9. Nous en avons identifié plusieurs autres, par exemple le patron « exploration sauf si détection », utilisé pour le comportement de proie, à la section 5, et qui resservira pour le comportement interne de fourmi `AntActivity`, à la section 6.2. Une discussion sur les patrons de conception de MALEVA est plus détaillée dans (Guillemet *et al.*, 1999, p. 4-5) et surtout dans (Guillemet *et al.*, 1998). Ces patrons de conception correspondent tous deux à une structure conditionnelle *if-then-else* autour d'un switch, avec certains comportements prédéfinis, ex. `Exploration` ou `Die`, et d'autres à instancier, ex. `Behavior`.

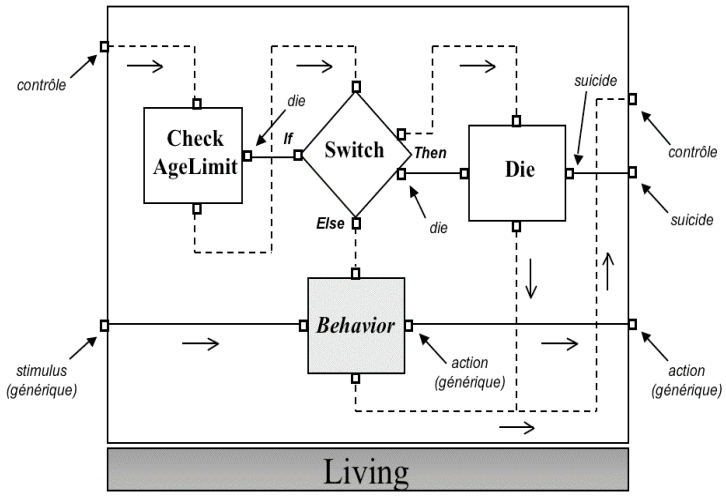

Figure 10. *Le comportement abstrait* `Living`

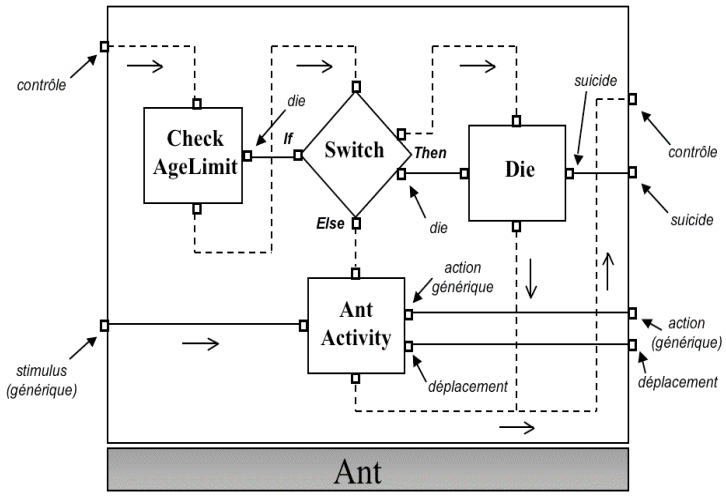

Figure 11. `Ant` *: comportement d'une fourmi*

6.2. *Le comportement de fourmi*

Une fourmi ouvrière possède un comportement relativement complexe du fait de ses multiples fonctions : déplacement, suivi de gradients de phéromones, transport d'œufs, soin d'œufs. Et, pour certaines espèces (la *Ectatomma ruidum* cannibale, étu-

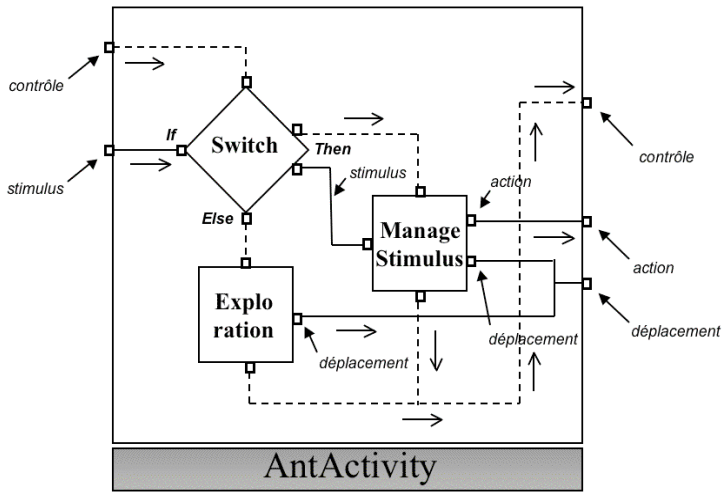

Figure 12. `AntActivity` *: 1^{er} niveau de décomposition du comportement d'une fourmi*

diée dans le projet MANTA (Drogoul, 1993)), elle peut tuer les œufs pour se nourrir. Nous en présentons ici une simplification.

Nous allons tout d'abord instancier le comportement abstrait (patron de conception) `Living` (voir la figure 10) en un comportement concret spécifique aux fourmis, nommé `Ant`. En pratique, nous remplaçons le comportement abstrait `Behavior` par un comportement concret nommé `AntActivity`.[10] Le résultat se trouve à la figure 11.

Nous pouvons maintenant définir le comportement spécifique de la fourmi ouvrière, `AntActivity`, illustré à la figure 12. Il stipule que la fourmi explore son environnement (de manière aléatoire) sauf si elle détecte un stimulus. En conséquence, `AntActivity` réutilise le patron de conception « exploration sauf si détection », déjà utilisé pour les comportements `Prey` et `Predator` (figures 8 et 9). Ceci illustre les capacités de réutilisation de MALEVA, au niveau des comportements, concrets ou abstraits (patrons de conception).

Toujours en suivant l'approche descendante, nous définissons finalement le deuxième niveau de décomposition de la fourmi, nommé `ManageStimulus`, illustré à

10. On peut remarquer que le composant `AntActivity` possède une borne de sortie de données supplémentaire, comme deux données distinctes pourront être produites : action (ex. déposer phéromone) et déplacement. Cette distinction permet de distinguer les différents actionneurs et d'utiliser au mieux le typage. Une simplification, conservant une seule borne de sortie de données, serait de regrouper tous les types de données (action et déplacement) dans une même borne/connexion, et entrée d'actionneur associée, en les définissant comme sous-types d'un type commun.

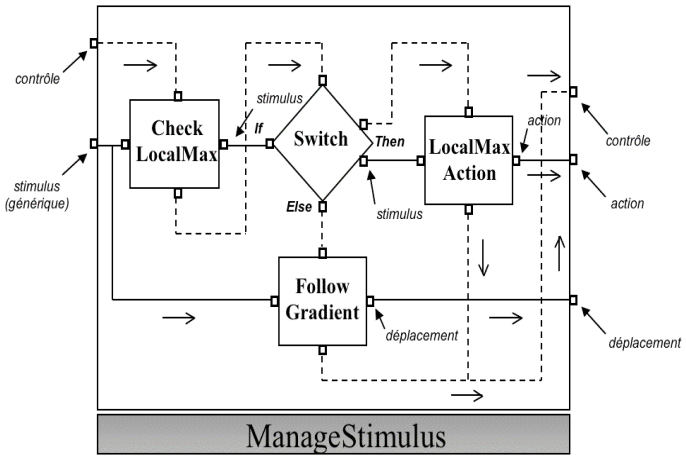

Figure 13. `ManageStimulus` *: 2^e niveau de décomposition du comportement d'une fourmi*

la figure 13. Si la fourmi est placée sur un maximum local à la source d'émission du stimulus (test effectué par le composant `CheckLocalMax`), une action correspondant au type de stimulus (ex. nourriture, phéromone) est alors produite par le comportement `LocalMaxAction`. Sinon, la fourmi suit le gradient associé au stimulus, ceci étant mis en œuvre par le comportement `FollowGradient`.

Au final, la décomposition et l'architecture complètes du comportement d'une fourmi ouvrière, comprenant trois niveaux de décomposition et au total 14 composants, est résumée à la figure 14.

7. Un troisième exemple : contrôle des dépendances temporelles entre les comportements d'un agent

Dans les deux précédents exemples, nous avons montré comment composer ou décomposer des comportements d'agents, à partir de ou jusqu'à des comportements plus simples. Spécifier de manière explicite le contrôle (par les bornes et connexions relatives au flot de contrôle) aide à découpler les fonctionnalités des comportements d'un agent de la logique de leur activation, rendant ainsi les comportements plus génériques. Nous allons maintenant rapidement illustrer un avantage supplémentaire de l'explicitation du contrôle : la capacité de spécifier les dépendances temporelles entre les comportements d'un agent, en vue de mieux contrôler l'ordonnancement, et ainsi de possibles biais de simulation.

Dans ce but, introduisons un exemple simplifié de microsimulation multi-agent (Gilbert *et al.*, 1999), inspiré d'une application d'analyse de l'évolution démographique de population (le modèle Destinie (INSEE, 1999)). Dans notre modèle ici très

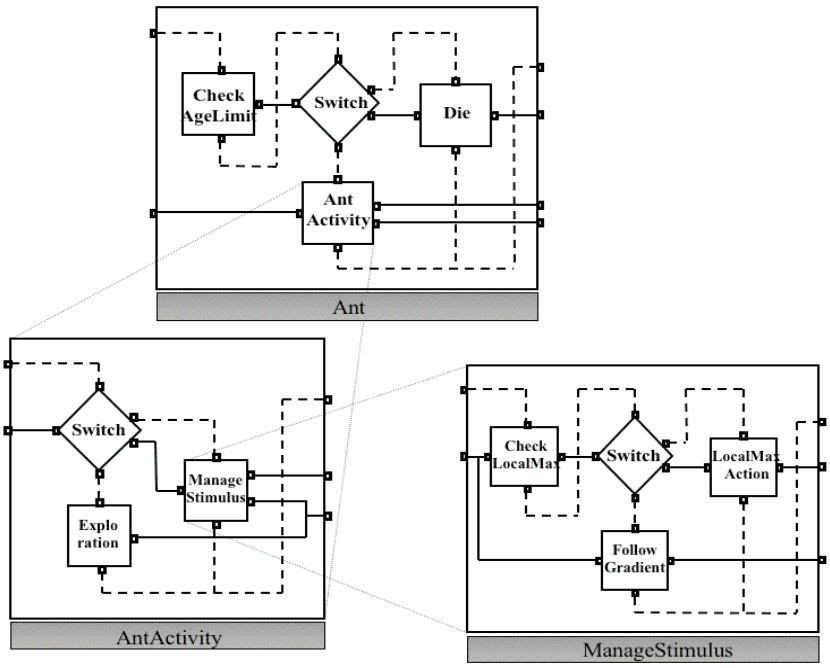

Figure 14. `Ant` *: décomposition complète du comportement d'une fourmi*

simplifié, peuplé d'individus d'une espèce virtuelle, la reproduction nécessite la rencontre de deux individus, mais sans considérer de distinction sexuelle (*i.e.*, les agents sont hermaphrodites). Nous ne considérons ici que 3 comportements élémentaires : mise en couple (`Mate`), séparation du couple (`Separate`), et reproduction (au sein du couple) (`Reproduce`). Les comportements y sont vus comme des changements d'états déterminés par des lois de transition de type probabiliste. Activer le comportement consiste à évaluer s'il y a changement d'état selon la probabilité associée.

Il est évident que ces 3 comportements ne sont pas indépendants, mais ils ne sont pour autant pas forcément liés par une relation séquentielle. En effet il n'y a *a priori* aucun ordre d'activation à garantir entre les comportements, tous les enchevêtrements de séquencement des 3 comportements étant valides. Des contraintes éventuelles de séquencement se font sur l'ordre dans lequel les probabilités seront évaluées, qui peuvent exprimer un ordonnancement spécifique sur les actions, mais pas obligatoirement.

Cependant, en règle générale (Gilbert *et al.*, 1999) et dans la plupart des exemples de simulations abordant ce type de problème (Doran *et al.*, 1994), ces 3 comportements sont instanciés de manière séquentielle, avec l'accord des thématiciens. En effet, il n'est pas évident de réaliser *a priori* l'importance du biais que ce choix peut induire (Meurisse *et al.*, 2001b).

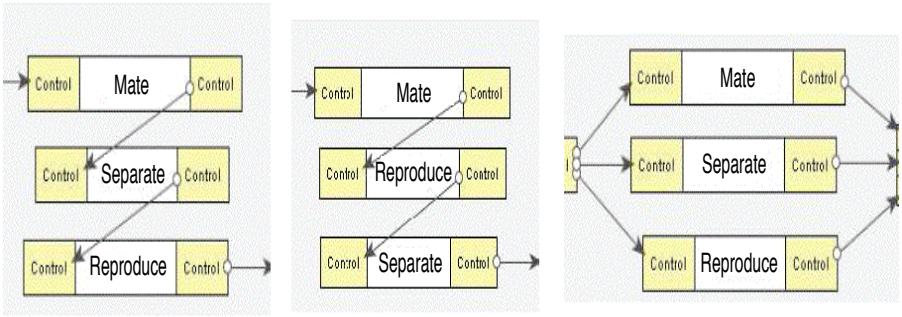

Figure 15. *Les trois types de contrôle des dépendances temporelles*

Considérons différentes dépendances temporelles entre ces 3 comportements :

– cas *(a)* : *séquence* { `Mate ; Separate ; Reproduce` }

– cas *(b)* : *séquence* { `Mate ; Reproduce ; Separate` }

– cas *(c)* : *concurrence* { `Mate || Separate || Reproduce` }

Les deux premières stratégies *(a)* et *(b)* fixent un ordre d'activation des comportements, alors que pour la troisième stratégie *(c)*, aucune contrainte n'est donnée, l'ordonnancement étant laissé libre à l'ordonnanceur (*scheduler*) du moniteur d'exécution (*runtime system*).[11] Il est important de noter que la variation de spécification des dépendances temporelles se fait *via* les connexions de contrôle, et sans aucune modification du code des comportements. De plus, elles se font en général graphiquement, *via* un outil graphique,[12] comme illustré à la figure 15.

Les histogrammes résultants sont illustrés à la figure 16 et affichent le nombre d'individus (coordonnée y) ayant un certain nombre d'enfants (coordonnée x). Les résultats sont relativement triviaux : si le comportement d'avoir un enfant est placé avant de divorcer (stratégie *(b)*), il y a en moyenne plus d'enfants que s'il est placé après (stratégie *(a)*). Un enchevêtrement libre (stratégie *(c)*) produit un nombre moyen.

Notez que notre objectif n'est pas de savoir lequel des 3 séquencements possibles serait *meilleur que les autres*, mais de mettre en évidence l'importance du choix de séquencement. Ainsi, (Lawson *et al.*, 2000) montre que des résultats de simulations peuvent être biaisés dans le cas où l'ordonnancement des actions d'un agent reste déterministe. La facilité de manipulation des dépendances temporelles, indépendamment des fonctionnalités des comportements (qui restent encapsulés), permet ainsi d'aider

11. Pour être plus précis, l'ordonnanceur de MALEVA tente, à chaque pas de simulation, de maximiser les entrelacements possibles entre les activations de comportements (Meurisse, 2004).
12. Cet outil est introduit brièvement à la section 8.1. On peut noter l'absence de bornes de données, les comportements étant ici très simples.

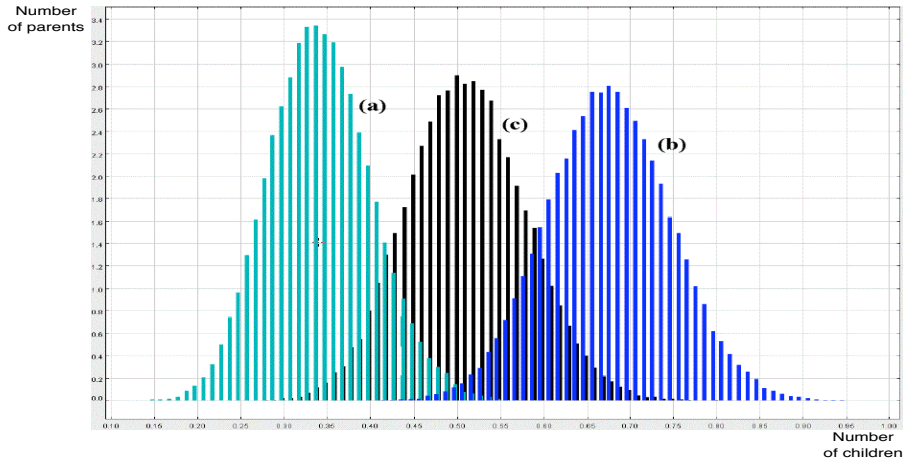

Figure 16. *Histogrammes des résultats : nombres d'enfants*

les experts à faire varier les dépendances - habituellement implicites - entre comportements, à comparer les résultats avec les modèles cibles, et à quantifier l'impact du séquencement et des biais induits (Briot *et al.*, 2006).

Le lecteur intéressé par ces aspects pourra se reporter à (Meurisse, 2004) pour une analyse quantitative du biais et pour une proposition de méthodologie générale d'opérationnalisation de conception de simulations multi-agent.

8. Outils de l'environnement et implantation

L'environnement prototype MALEVA inclut une bibliothèque de composants (composants de comportements d'agents et composants de contrôle) ; un éditeur de graphes de connexions ; un éditeur pour construire des environnements virtuels pour agents situés ; et un moniteur d'exécution (*run time support*) pour ordonnancer et activer les agents.

8.1. *Vers une rétroconception : des méthodes aux composants*

Une caractéristique intéressante de l'outil d'édition de graphes de connexions CGraphGen[13] (voir la figure 17) est l'importation relativement aisée de code Java existant et sa réification en des composants MALEVA. La granularité choisie est la méthode Java. Après avoir spécifié le nom de la classe, le nom de la méthode, et sa signa-

13. CGraphGen signifie *concurrent graph generation*. Il a été implanté à l'aide de la bibliothèque graphique Monarch (Monarch, 2000).

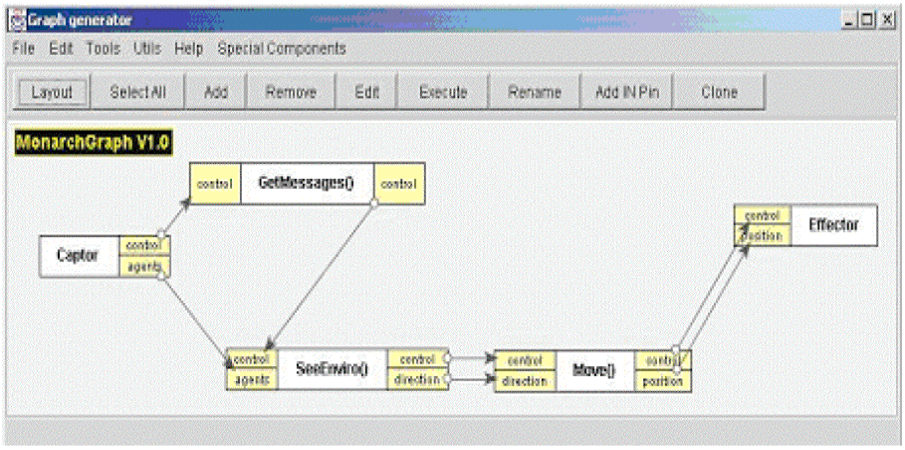

Figure 17. *L'outil CCgraphGen*

ture, CGraphGen génère automatiquement un composant associé dont les bornes de données correspondent à la signature de la méthode : une borne d'entrée de données pour chaque paramètre, et une borne de sortie optionnelle pour le résultat (aucune en cas de méthode `void`). Deux bornes de contrôle (une d'entrée et une de sortie) sont implicitement ajoutées. CGraphGen permet alors de manipuler des connexions, de données et de contrôle, entre les composants, et de définir des composants composites. Notez que des descriptions à base de XML peuvent être utilisées en importation ou en exportation. Ces facilités (réification de code existant dans des composants, manipulation des dépendances temporelles) s'avèrent très utiles pour re-concevoir des applications existantes de simulation, spécialement si l'on considère que les concepteurs des modèles et des simulations ne sont pas toujours des experts en programmation.

8.2. *Modes d'activation et d'ordonnancement*

Au niveau de l'ordonnanceur deux modes d'activation ont été implantés : un mode *asynchrone* et un mode *synchrone*. Le mode asynchrone est plus efficace et permet une expression du contrôle plus précise. Au niveau de chaque agent, la spécification des connexions de contrôle indique la synchronisation intra-agent désirée. Cependant, à moins que le concepteur n'utilise également des connexions de contrôle explicites entre les agents, les différents agents peuvent ne pas être synchronisés (certains peuvent progresser plus vite que d'autres). Dans le mode synchrone, l'ordonnanceur émet la prochaine activation une fois que tous les composants ont terminé, ce qui garantit mais aussi oblige à une synchronisation globale. Le choix parmi les deux modes dépend des besoins de l'application (voir par exemple sur ce sujet : (Lawson *et al.*, 2000) et (Meurisse *et al.*, 2001a)).

8.3. *Evolution de l'implantation*

Le modèle de composant MALEVA et son environnement de développement graphique prototype ont été successivement implantés dans 3 versions et langages : Delphi[14] (Lhuillier, 1998), Java (Meurisse *et al.*, 2001a), et enfin C++ (Meurisse, 2004).

La réimplantation en Java a permis d'apporter un certain nombre d'améliorations. La possibilité de changement dynamique de comportement[15] s'est avérée utile par exemple pour modéliser la métamorphose des fourmis (voir la section 6). Le typage des bornes a été ajouté, permettant ainsi un certain contrôle de conformité lors des connexions. De plus, le sous-typage peut aider à définir des composants plus génériques. Les facilités d'introspection (API et outils) de Java s'avèrent également fort utiles pour aider le concepteur.

L'implantation en Java, fondée sur les JavaBeans, a aussi donné l'occasion de confronter le modèle de composant prototype MALEVA à un modèle de composant industriel tel que les JavaBeans de Sun. Notez à ce sujet que le modèle conceptuel de JavaBeans (Sun, 2006) suit un modèle de communication événementiel de type *publish/subscribe*, mais l'implantation de JavaBeans repose sur l'appel de méthode traditionnel (objet). Dans notre implantation de MALEVA, une boîte aux lettres (queue de messages FIFO) est associée à chaque borne d'entrée de données ainsi qu'à la borne d'entrée du contrôle, de manière à découpler transfert de données et mode d'activation.

Une réimplantation en C++ a été principalement motivée par des besoins d'efficacité pour des expérimentations à relativement grande échelle, notamment avec le modèle Sugarscape (Epstein *et al.*, 1996) (Meurisse, 2004). Enfin, une réimplementation complète en Smalltalk (Squeak), nommée MalevaST, a été indépendamment récemment coordonnée par Noury Bouraqadi à l'École des Mines de Douai.

8.4. *Performances*

D'un point de vue de performance, il est clair qu'une décomposition de grain très fin d'un comportement d'agent induit un coût. Ceci renvoie au dilemne habituel entre généricité et performance. Mais il est important de noter que la granularité de décomposition peut être librement choisie par le concepteur d'une application en fonction de ses priorités. Enfin, rien n'empêche d'envisager l'ajout à MALEVA d'un générateur de code transformant (compactant) un composant composite en un composant d'une seule pièce, avec des optimisations additionnelles (par ex. transformer certaines activations en des appels de méthodes synchrones quand la concurrence intra-agent n'est pas exploitée).

---

14. Delphi est un langage et environnement de développement intégré à base de composants, fondé sur le langage Pascal, proposé par la société Borland, et précurseur de JBuilder.
15. L'implantation en Delphi imposait une recompilation, du fait de l'absence de chargement dynamique de code.

9. Travaux connexes

En complément de notre analyse comparative au début de l'article (section 2), nous signalons quelques travaux connexes supplémentaires, tant au niveau des modèles généraux de composants, qu'au niveau des architectures modulaires d'agents.

9.1. *Modèles de composants*

CCM (corba component model) (OMG, 2006) est un modèle de composant industriel générique, offrant de vrais modèles de communication asynchrone et par événements, mais qui n'inclut pas la notion de composant composite. De plus, son cycle de développement est portable mais relativement complexe.

Le modèle de composant Fractal (Bruneton *et al.*, 2004) est fondé sur des notions d'interfaces requises/fournies et de composant composite. Une notion de contrôleur est proposée mais il s'agit principalement d'interfaces de contrôle simples pour la gestion du cycle de vie ou la reconfiguration structurelle. Le modèle n'offre pas de caractérisation explicite du fil de contrôle.

L'introduction de composants de contrôle dans des modèles à base de composants a été également faite dans (Tarpin-Bernard *et al.*, 1999). De manière plus générale, 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 de plus en plus présente dans les travaux de recherche aujourd'hui, et à notre avis liée aux tentatives de réification des propriétés non fonctionnelles.

9.2. *Architectures modulaires d'agents*

Un certain nombre de plates-formes utilisent des modèles, généraux ou spécifiques, de composants, mais souvent focalisés au niveau du système multi-agent (chaque agent est implanté par un composant), et peu au niveau d'un agent (Shehory, 1998). Des exemples sont notamment : Jack (Brenton *et al.*, 1998) et Retsina (Decker *et al.*, 1996). Le lecteur pourra également consulter (Bordini *et al.*, 2006) pour une étude comparative générale récente sur les langages, architectures et plateformes multi-agents.

L'architecture ABLE (Bigus *et al.*, 2002), proposée dans le cadre du programme *Autonomic Computing* d'IBM, offre un ensemble de composants outils, regroupés par types d'outils et de techniques : par exemple le *Learning bean*, `BackPropagation` implantant la rétropropagation des réseaux de neurones, ou le *Rule bean*, `FuzzyForwardChaining` implantant de l'inférence floue. Tous ces composants sont implantés à partir de JavaBeans. Le concepteur va identifier des enchaînements de traitements pour un objectif donné, par exemple pour assurer un équilibrage de charges automatique pour un serveur.

L'architecture modulaire DIMA (Guessoum, 1996) (Guessoum *et al.*, 1999), bien que ne se fondant pas sur un vrai modèle de composant, offre une certaine ouverture et la possibilité de rajouter des composants. Le concepteur a en effet une certaine latitude de choix sur l'ensemble des composants de traitement et sur leur cohabitation, par exemple un module réactif, avec un module délibératif à base de règles, et un module d'apprentissage à base de classeurs. Le concepteur redéfinit alors le niveau méta de contrôle (décrit par un automate à états) qui est chargé d'ordonnancer et d'activer les modules de traitement.

L'architecture JADE (Bellifemine *et al.*, 2001) offre un certain support au concepteur pour construire un agent comme un ensemble de *comportements*, instances de la classe `Behaviour`. Ses sous-classes, ex. `CompositeBehaviour` et `ParallelBehaviour`, implantent différents types d'hiérarchies de comportements ou/et des structures de contrôle, par exemple `FSMBehaviour` est fondée sur un automate à états finis. Cependant, les comportements de JADE ne sont pas de vrais composants (pas d'interfaces de sortie ni de connecteurs), l'architecture d'un agent restant en conséquence en partie cachée dans le code.

L'architecture JAF (Java agent framework) (Horling, 1998) propose un mécanisme intéressant de mise en correspondance de composants. 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 du composant, JAF recherche la meilleure correspondance entre les descriptions fournies dans la liste et les composants présents lors de l'exécution.

Le modèle MAST (Vercouter, 2004), déjà discuté à la section 2.2, a un objectif similaire de plus grande flexibilité des connexions entre composants, car inférées en grande partie par les rôles et les types d'événements considérés. Ceci est augmenté d'un mécanisme de délégation du traitement des événements, toutefois contrôlé par des priorités explicites.

Pour finir, on peut citer des travaux sur l'utilisation de la programmation par aspects pour aider à l'intégration et la combinaison de fonctionnalités ou propriétés des agents (telles que l'autonomie, l'apprentissage, la mobilité…) (Garcia *et al.*, 2004). Mais ces fonctionnalités ajoutées ne sont pas décrites comme des composants bien délimités (cette transversalité - ou non localité - d'un aspect par rapport au code est d'ailleurs la caractéristique et la motivation de la programmation par aspects).

10. Directions futures

Un premier enjeu, pour notre architecture de composants d'agents MALEVA, est que les concepteurs d'applications multi-agent, et en particulier de simulations multi-agents, doivent élaborer des modèles d'activation de comportements par manipulation de connexions de contrôle, de relativement bas niveau et qui peuvent être fastidieuses. Nous avons montré dans les exemples précédents (notamment aux sections 5 et 6) plusieurs exemples de patrons d'architectures qui peuvent aider à capitaliser les ex-

périences. Cependant il reste à systématiser cette démarche, la bibliothèque actuelle restant encore trop limitée. Plus précisément, nous envisageons deux types de bibliothèques :

– une bibliothèque de composants de comportements, par exemple : `Exploration`, et de composants « système », par exemple : de perception, de migration, d'interaction.

– une bibliothèque de composants abstraits, représentant les micro-architectures récurrentes (patrons de conception), sortes de micro-frameworks « à trous », par exemple `Living` (à la section 6.1), qu'on instanciera ensuite en insérant des composants spécialisés.

De plus, l'assistance d'outils utilisant des informations telles que les informations de typage peut aider le concepteur à la fois à analyser les schémas existants et à en créer de nouveaux. (Voir à ce sujet des travaux sur l'assistance à l'induction de patrons de conception ou encore sur l'assistance à leur utilisation, tels que (Ziane, 2001)).

Un deuxième enjeu provient de notre expérience de spécification du contrôle *via* des connexions explicites. Elle montre que dans le cas de grandes applications, la logique de connexion peut devenir complexe et les graphes complets de grande taille, *bien qu'ils* puissent être hiérarchisés et encapsulés dans des composants composites (voir par exemple la figure 14). Une approche radicale pour réduire ce problème, et pour rendre la spécification du contrôle plus accessible à l'analyse, consiste à l'abstraire en un formalisme adéquat.

Nous pensons qu'une algèbre de processus telle que CCS (Milner, 1982) pourrait permettre une représentation concise de patrons d'activation complexes. Cette idée est proche des langages de coordination, mais pour des composants de grain très fin.

Le principe de départ est de modéliser les données utilisées pour le contrôle (ex. présence de proie, de prédateur, de phéromone) par des canaux et de synchroniser l'activité des comportements sur eux. Il en résulte une expression compacte pour exprimer un graphe de contrôle analogue à l'exemple proie-prédateur (de la section 5):

$$isPrey.Follow \;||\; isPredator.Flee \;||\; (isNoPrey.Exploration + isNoPredator.Exploration)$$

où *isPrey*, *isNoPrey*, *isPredator* et *isNoPredator* sont des canaux, connectés aux capteurs de l'agent, et *Follow*, *Flee*, et *Exploration* sont des processus.

Une telle formalisation devrait aussi permettre certaines formes d'analyse sémantique de tels patrons, par exemple par vérification de modèles (*model checking*).

Un dernier enjeu possible concerne la dynamicité. Du fait de l'évolution possible d'un agent dans le temps - par exemple pour s'adapter à son environnement, ou bien pour modéliser la nature évolutive de son comportement (ex. métamorphose des fourmis, à la section 6) - une telle capacité d'évolution dynamique s'avère indispensable, aussi bien au niveau de son comportement qu'au niveau de son architecture. Notre mé-

canisme actuel, reposant sur un métacomposant dédié (voir la section 6), reste encore un peu trop opérationnel et n'est pas suffisamment bien exprimé au niveau du modèle. Le mécanisme prototype de reconfiguration/réassemblage automatique de composants d'agents MadCar (Grondin *et al.*, 2006), guidé par des notions de plus haut niveau telles que configuration, rôles et politiques, nous semble en ce sens un candidat intéressant. Nous comptons d'ailleurs étudier son applicabilité. Enfin, pour assurer la dynamicité des formalismes de patrons d'activation, des modèles d'algèbres de processus offrant communication et dynamicité, tels que le Pi-calcul, doivent alors être envisagés.

11. Conclusion

Dans cet article, nous avons présenté notre expérience d'utilisation de composants pour concevoir et construire de manière compositionnelle des agents. Ce modèle est relativement original, voire radical, au sens où le critère de décomposition est le comportement de l'agent et aussi par le fait qu'il applique les principes des composants et de composition logicielle à la spécification explicite du contrôle, *via* des notions de bornes, connexions, et composants de contrôle. Ces caractéristiques offrent des possibilités intéressantes de généricité et de combinaison, ainsi qu'un contrôle précis de l'ordonnancement interne à un agent, un enjeu d'importance en particulier pour les applications de simulation multi-agent.

Nous pensons qu'il n'existe pas d'architecture d'agents optimale, comme cela dépend notamment du domaine d'application envisagé et des besoins. Une architecture générale (hybride), telle qu'InteRRaP, qui tente de réconcilier et factoriser à la fois les architectures cognitives et les architectures réactives, est puissante mais aussi complexe. A l'opposé, notre modèle d'architecture est plus simple, il se limite en fait à un modèle de composant, l'architecture effective restant à définir par le concepteur. Notre modèle de composant MALEVA a été plus particulièrement testé sur des applications de simulation. Nous espérons d'ailleurs avoir pu faire entrevoir ses capacités à cette occasion. Son applicabilité à d'autres types d'applications et d'agents reste à expérimenter et évaluer. Mais le modèle de composant MALEVA est en fait assez général, et il s'agit surtout de disposer d'une bibliothèque de composants et d'architectures abstraites appropriée aux types d'applications et d'architectures envisagées. De manière plus générale, nous pensons que certaines des caractéristiques de notre modèle pourraient être transposées, et que rendre disponible le contrôle au niveau de la composition pourrait aider l'utilisation des composants pour des gammes d'applications plus vastes que celles pour lesquelles ils ont été initialement conçus.

Remerciements

Nous remercions pour leur contribution au projet MALEVA : Marc Lhuillier, le concepteur de l'architecture et de la plate-forme initiales (Lhuillier, 1998), Alexandre

Guillemet et Grégory Haïk, pour leur stage de magistère sur la reconception des fourmis MANTA et sur l'identification de patrons de conception (Guillemet *et al.*, 1998). Nous remercions également les relecteurs pour leurs commentaires et suggestions.

12. Bibliographie

Bachman F., Bass L., Buhman C., Comella-Dorda S., Long F., Robert J., Seacord R., Wallnau K., Volume II: Technical Concepts of Component-Based Software Engineering, Technical Report CMU/SEI-2000-TR-008 & ESC-TR-2000-007, Software Engineering Institute, Carnegie Mellon, Pittsburgh, PA, États-Unis, mai 2000.

Bellifemine F., Poggi A., Rimassa G., « Developing Multi-Agent Systems with a FIPA-compliant Agent Framework », *Software Practice and Experience*, No. 31, 2001, p. 103-128.

Bigus J.P., Schlosnagle D.A., Pilgrim J.R., Mills W.N., Diao Y., « ABLE: A Toolkit for Building Multiagent Autonomic Systems », *IBM Systems Journal*, Vol. 41, No. 3, 2002, p. 350-371.

Boissier O., « Modèles et architectures d'agents », (Briot *et al.*, 2001, p. 71-107).

Bordini R., Braubach L., Dastani M., El Fallah Seghrouchni A., Gomez-Sanz J.J., Leite J., O'Hare G., Pokahr A., Ricci A., « A Survey of Programming Languages and Platforms for Multi-Agent Systems », *Informatica*, No. 30, 2006, p. 33-44.

Brazier F., Dunin-Keplicz B., Jennings N., Treur J., « Formal Specification of Multi-Agent Systems: a Real-World Case », *1st International Conference on Multi-Agent Systems*, San Francisco, CA, États-Unis, MIT Press, 1995, p. 25-32.

Brazier F., Jonker C., Treur J., Wijngaards N., « Compositional Design of a Generic Design Agent », *Design Studies Journal*, No. 22, 2001, p. 439-471.

Brenton A., Wood T., JACK Intelligent Agents User Guide, Agent Oriented Software Pty Ltd., Australie, 1998.

Briot J.-P., Demazeau Y. (ed), *Principes et architecture des systèmes multi-agents*, Collection IC2, Hermès, 2001.

Briot J.-P., Meurisse T., « A Component-based Model of Agent Behaviors for Multi-Agent-based Simulations », *7th International Workshop on Multi-Agent-Based Simulation (MABS'06)*, Antunes L., Takadama K. (ed), 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'2006), Hakodate, Japon, mai 2006, p. 183-190.

Briot J.-P., « Composants logiciels et systèmes multi-agents », *Technologies SMA et leur utilisation dans l'industrie*, El Fallah-Seghrouchni A. (ed), Collection IC2, Hermès, 2006. (À paraître).

Brooks R.A., « A Robust Layered Control System for a Mobile Robot », *IEEE Journal of Robotics and Automation*, Vol. 2, No. 1, mars 1986, p. 14-23.

Bruneton E., Coupaye T., Leclerc M., Quema V., Stefani J.-B., « An Open Component Model and its Support in Java », *7th International Symposium on Component-Based Software Engineering*, No. 3054, LNCS, Springer-Verlag, mai 2004, p. 7-22.

Decker K., Sycara K., Designing Reusable Behaviors for Information Agents, Technical Report, The Robotics Institute, CMU, Pittsburgh, PA, États-Unis, janvier 1996.

Demazeau Y., « From Interactions to Collective Behaviour in Agent-Based Systems », *1st European Conference on Cognitive Science*, Saint-Malo, 1995, p. 117-132.

Doran J.E., Gilbert N. (ed), *The Computer Simulation of Social Phenomena*, UCL Press, Royaume-Uni, 1994.

Drogoul A., De la simulation multi-agent à la résolution collective de problèmes, Thèse de doctorat, Université Paris 6, 1993.

Drogoul A., Vanbergue D., Meurisse T., « Multi-Agent Based Simulation: Where are the Agents ? », *3rd International Workshop on Multi-Agent-Based Simulation (MABS'02)*, Sichman J., Bousquet F., Davidsson P. (ed), No. 2581, LNAI, Springer-Verlag, 2003.

Epstein J.M., Axtell R.L., *Growing Artificial Societies: Social Science from the Bottom Up*, MIT Press, 1996.

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns - Elements of Reusable Object Oriented Software*, Professional Computing Series, Addison-Wesley, 1995.

Garcia A., Kulesza U., Lucena C., « Aspectizing Multi-Agent Systems: From Architecture to Implementation », *Software Engineering for Multi-Agent Systems III*, No. 3390, LNCS, Springer-Verlag, décembre 2004, p. 121-143.

Gilbert N., Troitzsch K.G., *Simulation for the Social Scientist*, Open University Press, Royaume-Uni, 1999.

Grondin G., Bouraqadi N., Vercouter L., « Assemblage automatique de composants pour la construction d'agents avec MadCar », *2^{e} Journée Multi-Agent et Composant (JMAC'06)*, LMO'06, Nîmes, mars 2006.

Guessoum Z., Un environnement opérationnel de conception et de réalisation de systèmes multi-agents, Thèse de doctorat, Université Paris 6, mai 1996.

Guessoum Z., Briot J.-P., « From Active Objects to Autonomous Agents », Special Series on Actors and Agents, Kafura D., Briot J.-P. (ed), *IEEE Concurrency*, Vol. 7, No. 3, juillet–septembre 1999, p. 68-76.

Guillemet A., Haïk G., Évaluation d'une architecture componentielle de conception d'agents, Rapport de Stage de 1re année de Magistère d'Informatique et Modélisation, École Normale Supérieure de Lyon, septembre 1998.

Guillemet A., Haïk G., Meurisse T., Briot J.-P., Lhuillier M., « Mise en œuvre d'une approche componentielle pour la conception d'agents », *Journées Francophones sur l'Intelligence Artificielle et les Systèmes Multi-Agents (JFIADSMA'99)*, Gleizes M.-P., Marcenac P. (ed), Hermès, novembre 1999, p. 53-66.

Horling B.C., A Reusable Component Architecture for Agent Construction, Technical Report No. 1998-49, Computer Science Dept., UMASS, MA, États-Unis, octobre 1998.

INSEE, Le modèle de microsimulation dynamique DESTINIE, Technical Report G9913, Division Redistribution et Politiques Sociales, Institut National de la Statistique et des Etudes Economiques (INSEE), 1999.

Lawson B.G., Park S., « Asynchronous Time Evolution in an Artificial Society Mode », *Journal of Artificial Societies and Social Simulation*, Vol. 3, No. 1, 2000.

LeCerf V., Pintado M., « An Adaptive Model of Camera-Driven Urban Intersections Observation », *Workshop on Dynamic Scene Recognition*, Tessier C. (ed), CERT/ONERA, Toulouse, 1997.

Lhuillier M., Une approche à base de composants logiciels pour la conception d'agents. Principes et mise en œuvre à travers la plate-forme MALEVA, Thèse de doctorat, Université Paris 6, février 1998.

Marca D.A., McGowan C.L., *SADT (Structured Analysis and Design Technics)*, McGraw-Hill, 1987.

Melo F., Choren R., Cerqueira R., Lucena C., Blois M., « Deploying Agents with the CORBA Component Model », *Proceedings of the 2nd International Conference on Component Deployment (CD'2004)*, No. 3083, LNCS, Springer-Verlag, mai 2004, p. 234-247.

Meurisse T., Briot J.-P., « Une approche à base de composants pour la conception d'agents », *Technique et Science Informatiques (TSI)*, Vol. 20, No. 4, avril 2001, p. 583-602.

Meurisse T., Vanbergue D., « Et maintenant à qui le tour ? Aperçu de problématiques de conception de simulations multi-agents », *Colloque Agents Logiciels, Coopération, Apprentissage, Activité humaine (ALCAA'01)*, Bayonne, 2001.

Meurisse T., Simulation multi-agent : du modèle à l'opérationnalisation, Thèse de doctorat, Université Paris 6, juillet 2004.

Milner R., *A Calculus for Communicating Systems*, Springer-Verlag, 1982.

Monarch Graph Web Page, 2000.
"`http://singletonlabs.com/mgraph.html`".

Moss S., Davidsson P. (ed), *2nd International Workshop on Multi-Agent-Based Simulation (MABS'2000) - Revised and additional papers*, No. 1979, LNCS, Springer-Verlag, 2001.

Müller J.P., Pischel M., The Agent Architecture InteRRaP: Concept and Application, Technical Report RR-93-26, DFKI, Saarbrucken, Allemagne, 1993.

Müller J.P., « Control Architectures for Autonomous and Interacting Agents: A Survey », *Intelligent Agent Systems: Theoretical and Practical Issues*, No. 1209, LNAI, Springer-Verlag, 1997, p. 1-26.

OMG (Object Management Group), Corba Component Model (CCM), 2006.
"`http://www.omg.org/technology/documents/formal/components.htm`".

Peschanski F., Meurisse T., Briot J.-P., « Les composants logiciels : Évolution technologique ou nouveau paradigme ? », *Conférence Objets, Composants, Modèles (OCM'2000)*, Nantes, France, mai 2000, p. 53-65.

Ricordel P.-G., Demazeau Y., « La plate-forme VOLCANO : modularité et réutilisabilité pour les systèmes multi-agents », *Technique et Science Informatiques (TSI)*, Vol. 20, No. 4, avril 2001.

Shaw M., Garlan D., *Software Architectures - Perspective on an Emerging Discipline*, Prentice Hall, 1996.

Shehory O., Architectural Properties of Multi-Agent Systems, Technical Report No. CMU-RI-TR-98-28, The Robotic Institute, CMU, Pittsburgh, PA, États-Unis, décembre 1998.

Shoham Y., « Agent Oriented Programming », *Artificial Intelligence*, Vol. 60, No. 1, 1993, p. 51-92.

Sun Microsystems Inc., JavaBeans Specification, 2006.
"`http://java.sun.com/products/javabeans/`".

Tarpin-Bernard F., David B., « AMF : un modèle d'architecture multi-agents multi-facettes », *Technique et Science Informatiques (TSI)*, Vol. 18, No. 5, 1999, p. 555-586.

Vanbergue D., Conception de simulation multi-agents : Application à la simulation des migrations intra-urbaines de la ville de Bogota, Thèse de doctorat, Université Paris 6, 2003.

Vercouter L., « MAST : Un modèle de composants pour la conception de SMA », *1res Journées sur les Multi-Agents et les Composants (JMAC'04)*, Paris, novembre 2004. "`http://csl.ensm-douai.fr/MAAC/3`".

Wooldridge M., Jennings N.R., « Agent Theories, Architectures, and Languages: a Survey », *Workshop on Agent Theories, Architectures, and Languages on Intelligent Agents (ATAL'94/ECAI'94)*, Springer-Verlag, 1995, p. 1-39.

Ziane M., « Towards Tool Support for Design Patterns using Program Transformations », Colloque Langages et Modèles à Objets (LMO'2001), *L'Objet*, Vol. 7, No. 1-2, Hermès, janvier 2001, p. 199-214.

ANNEXE POUR LE SERVICE FABRICATION

A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LA REVUE :
   RSTI - L'objet – 12/2006. Composants et systèmes multi-agents

2. AUTEURS :
   Jean-Pierre Briot — Thomas Meurisse — Frédéric Peschanski

3. TITRE DE L'ARTICLE :
   Une expérience de conception
   et de composition de comportements
   d'agents à l'aide de composants

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Composition de comportements d'agents

5. DATE DE CETTE VERSION :
   19 novembre 2006

6. COORDONNÉES DES AUTEURS :

   – adresse postale :

   Laboratoire d'Informatique de Paris 6 (LIP6)
   Université Paris 6 - CNRS
   4 place Jussieu, F-75252 Paris cedex 05

   {Jean-Pierre.Briot,Thomas.Meurisse,Frederic.Peschanski}@lip6.fr

   – téléphone : +55 21 2551-9123

   – télécopie : +55 21 2511-5645

   – e-mail : Olivier Boissier

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes.cls`,
   version 1.2 du 03/03/2005.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com