Chapitre 5

Composants logiciels et systèmes multi-agents

5.1. Introduction

Les *composants logiciels* [BAC 00] et les *systèmes multi-agents* (souvent abrégés en *SMA*) [BRI 01, FER 95] 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, notamment à l'aide de leurs capacités d'auto-organisation et d'utilisation de connaissances. Cependant, nous pensons que les concepts et la technologie des composants logiciels peuvent aussi aider à la construction des systèmes multi-agents. L'objectif de ce chapitre est d'analyser ces différents points.

5.1.1. *Approche et positionnement*

Dans un premier temps, nous proposons une analyse comparative entre les composants logiciels et les systèmes multi-agents. De manière à mieux pouvoir les comparer, il nous semble utile de les replacer dans une perspective générale d'évolution de la programmation.

Chapitre rédigé par Jean-Pierre Briot.

Dans un deuxième temps, nous étudierons le potentiel de fertilisation croisée entre composants et systèmes multi-agents. Nous aborderons tout d'abord l'apport potentiel 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 pour une assistance à l'assemblage. Puis nous développerons l'apport potentiel des composants vers les agents comme structure de construction, d'intégration et de déploiement, non seulement à l'échelle du système multi-agent, mais comme nous le verrons, aussi à l'échelle d'un agent.

Il existe un certain nombre d'études comparatives entre les agents et les systèmes multi-agents et : les objets [ODE 02], les objets concurrents [GAS 92] et les acteurs [KAF 98]. Ce chapitre reprend certaines de ces analyses et tâche de les compléter par des aspects propres aux composants, ce qui, à notre connaissance, a encore peu fait l'objet d'études comparatives. Une initiative récente à signaler cependant, sur le thème des croisements entre composants et systèmes multi-agents, est l'organisation de deux éditions successives d'une Journée multi-agents et composants (JMAC), en 2004 puis en 2006. Un numéro spécial de la revue *L'Objet* vient également d'être édité sur ce thème [BOI 06], à la suite de la première journée. Signalons enfin l'ouvrage récent [BOR 05], qui présente sous une même grille d'analyse plusieurs plates-formes et langages multi-agents, fondés suivant sur les cas sur des modèles à objets, logiques ou à base de composants. L'article [BOR 06] présente une analyse analogue et plus concise.

5.2. Analyse

De manière à mieux pouvoir comparer les concepts d'agent et de système multi-agent avec le concept de composant logiciel, nous avons choisi de les replacer dans une perspective générale d'évolution de la programmation, en nous inspirant ainsi notamment de [GAS 98]. Nous avons choisi un référentiel commun à trois dimensions (figure 5.1), qui nous paraissent des enjeux importants de la programmation et du logiciel :

– sélection de l'action. Elle indique quand et comment sera décidée par l'entité logicielle (ou physique, dans le cas d'un robot) l'action et l'activation du code correspondant. L'évolution de la programmation montre un besoin de repousser toujours plus loin et plus tard cette décision (*ever late time binding* : *liaison toujours plus tardive*). Pour un agent, une telle décision se fonde, non seulement sur la nature de l'invocation, comme pour les langages de programmation classiques, mais également sur le contexte et les connaissances propres (par exemple, les buts) de l'agent ;

– flexibilité du couplage. Elle représente la capacité de mettre en relation plusieurs entités logicielles. L'évolution de la programmation montre le besoin croissant de représenter et manipuler ces relations de manière indépendante de la description des

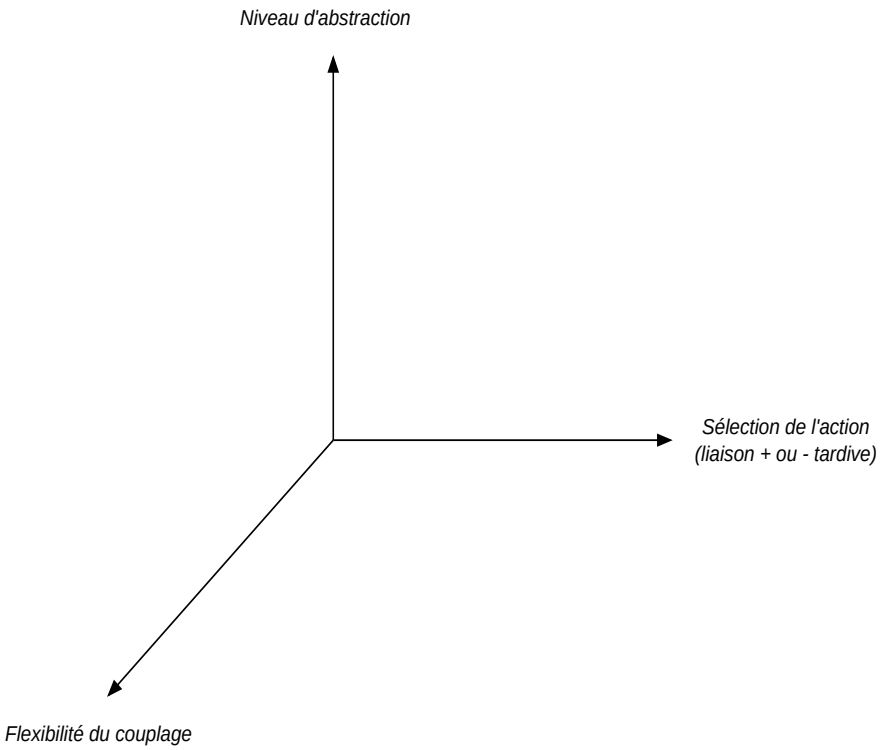

Figure 5.1. *Evolution de la programmation*

entités elles-mêmes, pour offrir une vision architecturale indépendante de l'implantation. Le concept d'architecture logicielle, assemblage de composants *via* des connecteurs bien explicites, représente de ce fait une avancée majeure. Le concept de service apporte une dynamicité (découverte de services) et un contrôle par l'entité elle-même (sélection et invocation de service). Les systèmes multi-agents poussent encore plus loin cette organisation du couplage et de la collaboration (coordination, décomposition, négociation...) à l'aide de connaissances (organisations, tâches, plans, protocoles...) ;

– niveau d'abstraction. Il représente le choix de niveau d'expression des concepts offerts au concepteur et au programmeur. On constate, sans trop de surprise, à partir des premiers concepts de bas niveau liés à l'exécution, tels le concept d'instruction, un souci d'abstraction progressive croissante (procédures, structures de données, objets, modèles... jusqu'aux connaissances).

Il faut noter que ces dimensions ne sont pas complètement indépendantes : la flexibilité de la sélection de l'action peut avoir un certain impact sur la flexibilité du couplage, et les choix des abstractions et mécanismes pour la sélection de l'action et pour le couplage ont un lien clair avec le niveau d'abstraction en général.

De plus, il est possible de considérer la sélection de l'action et le couplage de manière uniforme, tous deux fondés sur un mécanisme unique, celui de liaison (*binding*), voir par exemple [GHE 02]. Nous avons cependant préféré les distinguer car les niveaux et échelles respectifs sont conceptuellement bien distincts (vision micro *versus* vision macro), les professions également (programmeur versus architecture de système) de même que les abstractions et mécanismes (par exemple : architecture d'agents *versus* connecteur).

5.3. Sélection de l'action

Les premiers langages de programmation, et en premier lieu la première version de Fortran, considèrent l'espace du comportement du programme (code) et de son état (données) globalement. Les différentes instructions sont identifiées par l'intermédiaire de leur numéro de ligne. La sélection de l'action est par conséquent exprimée globalement et statiquement.

Les langages de programmation structurée ou modulaire, tels Pascal, puis Modula, apportent une modularisation et encapsulation du code, exprimé sous la forme de *procédures*. La sélection de l'action gagne ainsi en abstraction, l'indication du code à exécuter étant exprimée *via* un nom symbolique et non plus par un numéro de ligne. Elle reste cependant déterminée statiquement. En parallèle, les données gagnent progressivement en structuration et en généralité, au travers de *structures abstraites de données*, tout en restant cependant indépendantes et externes aux procédures.

Les langages de programmation par objets, avec les pionniers Simula67 puis Smalltalk, apportent alors une innovation majeure, avec la réunion de comportements et de données dans une capsule autonome appelée *objet*. Les données deviennent ainsi internes et privées à l'objet et ses comportements. Une autre avancée déterminante est l'apparition avec Smalltalk de la liaison tardive, c'est-à-dire que la procédure (appelée *méthode*) à invoquer sera déterminée en fonction de la classe de l'objet effectif invoqué, et non pas en fonction de la déclaration du type de la variable qui le référence (choix que fera par exemple C++). De ce fait, la liaison de la procédure, et donc la sélection de l'action, est repoussée à l'exécution et non pas résolue statiquement à la compilation. On parle de ce fait de *liaison tardive* (*late binding*), par exemple en

Smalltalk ou en Java, par opposition à une *liaison statique* ou *anticipée* (*early binding*), telle qu'en C++[1].

Du point de vue de la sélection de l'action, les composants logiciels n'innovent pas véritablement car, comme nous le verrons à la section 5.4, ils se concentrent plutôt sur l'enveloppe et donc le couplage plutôt que sur la mise en œuvre du comportement interne. Ils apportent de plus la notion de « *prêt à porter* », à déployer, et à utiliser. Un composant possède des attributs qui peuvent être configurés, l'analogue de l'affectation des attributs d'un objet instance d'une classe. Mais, à l'opposé des objets qui sont orphelins et potentiellement inopérants sans leur classe, voire leur sur-classe (ou classe mère)[2], un composant est « *auto-contenu* », avec tout son code, mais également sa documentation [OUS 05]. En ce sens, il y a ainsi tout de même un léger impact sur la sélection de l'action, qui, à l'opposé d'un objet, ne nécessite donc pas d'informations externes au composant.

Le concept d'agent introduit quand à lui une autonomie interne à la sélection de l'action. En effet, elle n'est plus nécessairement seulement gouvernée de manière externe par la nature de la requête, comme pour les appels de procédure ou de méthode, mais également par l'état interne de l'agent, ou plus exactement par ses connaissances, puisqu'il peut s'agir d'informations cognitives telles que ses *buts* propres. En ce sens un agent n'est plus seulement *réactif* (à des invocations) comme les objets, mais également *proactif* [ODE 02]. La notion de *sélection de l'action* prend alors tout son sens, comme pour un robot ou un être humain, qui arbitre lui-même sa ou ses actions à un moment donné, en fonction à la fois de ses objectifs propres (objectifs internes ou issus de requêtes passées) et des informations recueillies du moment (messages provenant d'autres agents, perceptions de l'environnement). L'arbitrage peut être effectué à un niveau symbolique et cognitif, par exemple en fonction des intentions, dans une architecture telle que BDI [GEO 99]. Les agents *réactifs*, par opposition aux agents *cognitifs*, ont des mécanismes de sélection d'action beaucoup plus simples, fondés sur une réponse assez directe à un type de stimulus. En cela, ils se rapprochent beaucoup plus du mécanisme de sélection de méthode de la programmation par objets, en réponse à la réception d'un message. Il existe en fait un continuum entre les deux catégories, et des architectures hybrides essaient de concilier et combiner les deux approches (voir par exemple l'architecture hybride InteRRaP au paragraphe 5.7.4). Enfin des mécanismes sub-symboliques de régulation, souvent inspirés de la biologie (métabolisme, émotions, adaptation… voir par exemple [NOL 06]) peuvent également intervenir.

---

1. Nous n'abordons volontairement pas ici les relations entre liaison et typage (et sous-typage), du fait qu'elles sont subtiles et qu'il existe plusieurs écoles, en conséquence difficiles à résumer.

2. Par exemple, en cas de mobilité d'un objet vers un autre site qui n'aurait pas déjà chargé la hiérarchie de classes associée.

Programmation	*Monolithique ex : Fortran*	*Modulaire ex : Pascal*	*Par objets ex : Java*	*Par agents ex : AgentSpeak*
Comportement	Global	Modulaire	Modulaire	Modulaire
État	Global	Modulaire et externe	Modulaire et interne	Modulaire et interne
Invocation (et sélection)	Globale et statique *(goto)*	Externe et statique *(appel de procédure)*	Externe et dynamique *(appel de méthode)*	Interne et dynamique *(ex : dirigée par les buts)*

Tableau 5.1. *Structure des entités et sélection de l'action*

Les Gasser a proposé comme un des concepts fondamentaux des agents la notion d'*action persistante structurée* (*structured persistent action*), dans laquelle l'agent tente de manière persistante d'accomplir quelque chose, de manière indépendante de la manière de le programmer [GAS 98]. En programmation procédurale classique, le programmeur contrôle explicitement les tentatives, alors que le concept d'action persistante structurée abstrait et encapsule un tel mécanisme. Le concepteur fournit la description de l'objectif ou des critères de son succès, ainsi qu'en général une collection de méthodes et de recettes. Certains de ces mécanismes ont déjà été abordés par exemple par la programmation logique, tels que Prolog (*programmation déclarative*, *retour arrière* (*backtrack*)), ou par le concept général de recherche (*search*). Mais, à notre avis, le concept d'action persistante structurée représente de manière intéressante l'encapsulation de la notion de choix, d'informations[3], de structure de contrôle itérative (de type *repeat until*), et de ressources propres (*processus* ou *thread* propre). Il faut également considérer une interaction de l'agent avec son environnement pour assurer le *feedback* de ses actions et choix. Il faut enfin noter que la sélection (et donc le choix) de l'action a lieu au moment de l'action et non pas de la programmation de l'agent. En ce sens, l'agent se situe bien dans la poursuite de la *liaison toujours plus tardive* (*ever late time binding*).

Le tableau 5.1, inspiré de [ODE 02], résume cette comparaison et la figure 5.2 la replace dans notre référentiel.

5.4. Flexibilité du couplage

La question de l'expression du couplage entre entités logicielles est un aspect très important de la structuration du logiciel. Elle recouvre en fait plusieurs facettes :

3. Des informations sur les choix possibles d'actions par rapport au domaine dans lequel agit l'agent. Ces informations peuvent être de nature symbolique (croyances, modèles, plans...) ou non, suivant les choix d'architectures et de la représentation du monde dans lequel agit l'agent.

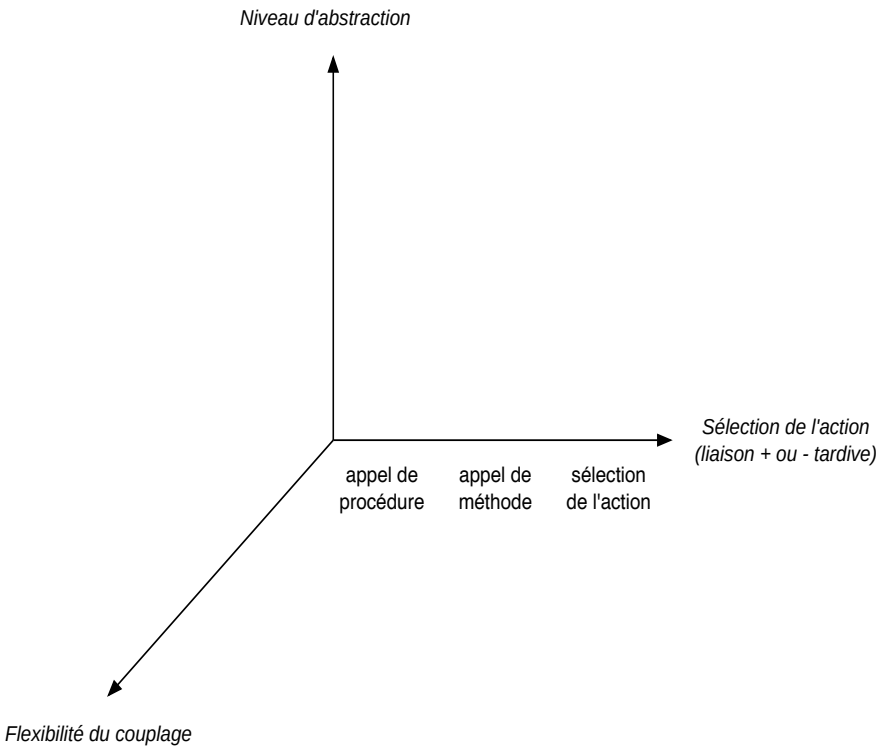

Figure 5.2. *Evolution de la sélection de l'action*

– *structure* : les concepts architecturaux (par exemple, références, connexions...) de mise en relation (référence) structurelle des entités ;

– *communication* : les modes de communication entre entités, caractérisés principalement par : le mode de désignation du receveur, le mode de transfert des données, et le le mode de couplage temporel.

5.4.1. *Couplage structurel*

La question du couplage structurel entre entités logicielles est au départ traitée par la notion de *référence* d'une entité par le biais d'un moyen de l'identifier (*identificateur*). Elle permet ainsi de désigner un objet informatique (dans les premiers langages, des données simples, puis avec Lisp des fonctions, puis des objets) et de le manipuler et le transmettre. Ce modèle, simple mais efficace et général, subsiste jusqu'aux langages de programmation par objets. Ainsi, un objet `A` référence un objet `B` auquel il va

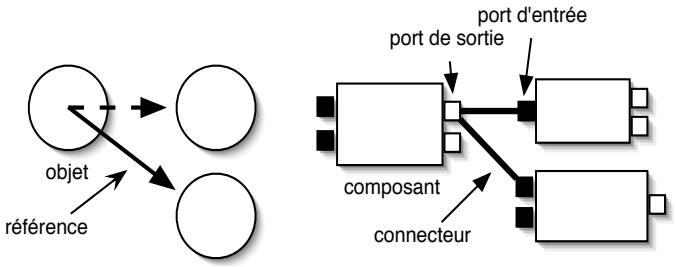

Figure 5.3. *Couplage entre objets versus couplage entre composants*

pouvoir ainsi envoyer des requêtes. En pratique la représentation interne (implémentation) de `A` inclut une variable dont la valeur est l'identifiant de l'objet `B`. Changer une référence par une autre est aisé, il suffit de changer la valeur de la variable, par exemple pour un troisième objet `C`. Cependant, on peut d'ores et déjà observer que cette modification ne peut être opérée que de manière interne à l'objet `A`, seul habilité à accéder à ses données privées (principe d'*encapsulation*).

Une limitation sérieuse survient dès que l'on veut pouvoir *étendre* une référence, par exemple pour que `A` référence *à la fois* `B` et `C` (voir la partie gauche de la figure 5.3). Une variable n'ayant qu'une valeur, ceci n'est pas exprimable directement. Il faut donc introduire une structure de données (une *collection*, par exemple : une liste) contenant `B` et `C`. Mais il faut également modifier l'instruction d'envoi de message en introduisant un *itérateur* sur la collection. Tout ceci nécessite de modifier la représentation interne de l'objet `A` (autrement dit de le réimplémenter), alors qu'il s'agit uniquement d'étendre la référence et le couplage initialement « de A vers `B` » en « de `A` vers `B` *et* `C` ».

La notion de composant logiciel apporte une amélioration notable à ce problème en externalisant les références, et en les décrivant explicitement sous la forme d'*interfaces de sortie*, par opposition, ou plutôt en complément[4], des *interfaces d'entrée* traditionnelles des objets et des procédures. Une terminologie équivalente et souvent employée est celle des *interfaces requises* (de sortie) et *interfaces fournies* (d'entrée).

La manipulation du couplage devient ainsi *explicite* et *externe* aux entités logicielles. Le couplage est explicité par des *connecteurs*, qui vont relier les interfaces des composants. Les identifiants de ces interfaces sont en général appelés des *bornes*,

4. En ce sens, on peut ainsi dire qu'un composant retrouve une symétrie, au niveau des interfaces, d'entrée et de sortie.

ou encore des « *ports* », d'entrées ou de sorties. La reconfiguration souhaitée s'opère ainsi par un simple ajout de connecteur, ceci étant illustré dans la partie droite de la figure 5.3.

Un composant peut posséder plusieurs bornes (interfaces) d'entrée, et de même pour les bornes de sortie. C'est une différence importante avec un objet qui n'a lui qu'un identifiant et point d'entrée. Une conséquence intéressante est que les composants sont *compositionnels*. C'est-à-dire qu'une composition de plusieurs composants est équivalente[5] à un composant qui posséderait la même union d'ensembles de bornes d'entrée et de bornes de sorties. Les objets eux ne sont pas directement compositionnels : une composition de plusieurs objets n'est pas immédiatement équivalente à un objet, car elle a plusieurs points d'entrée.

Les composants apportent une vision *architecturale* (*architecture logicielle* [SHA 96]) explicite de l'application, où l'on s'intéresse à la logique du couplage entre les composants, indépendamment de leur implantation interne. Les *langages de description d'architecture* (« *architecture description languages (ADL)* ») [SHA 96] sont dédiés à la spécification de l'architecture d'une application et ils sont de fait très différents des langages de programmation. Les informations de typage des interfaces des composants sont utilisées pour vérifier la correction de l'assemblage, c'est-à-dire la conformité entre les interfaces mises en relation. Différents types de connecteurs sont habituellement proposés, et sont relatifs à différents styles architecturaux (par exemple, *par couches*, *pipes and filters*, *par diffusion d'événements*... [SHA 96]) et protocoles de communication associés. Les connecteurs peuvent également représenter des caractéristiques non fonctionnelles (telles que la répartition, la qualité de service...) et possèdent donc une sémantique propre [ALL 94].

De manière à exprimer non seulement des indications sur les types de données (typage) mais également sur les comportements des composants, des notions plus générales de *contrats* ont été également proposés. [BEU 99] considère quatre niveaux successifs de contrats : syntaxiques, comportementaux, synchronisation, et qualité de service. Suivant les cas, ils peuvent être *garantis*, *vérifiés* ou *négociés*. Le niveau syntaxique repose sur un système de *types*. Le niveau comportemental repose en général sur des *assertions*. Les trois types principaux sont : les *préconditions*, les *postconditions*, et les *invariants*. L'idée générale des contrats, par rapport à l'utilisation originale

5. Il faut distinguer entre *composition fonctionnelle*, simple assemblage de composants, et *composition structurelle*, qui encapsule une composition fonctionnelle et l'identifie en un nouveau composant, souvent appelé *composant composite*. Nous pensons que ce concept de composition structurelle est important [PES 00], car il offre l'encapsulation et la hiérarchisation qui s'avèrent fort utiles pour aider à maîtriser la complexité. Mais curieusement, il est en pratique seulement supporté par une minorité de modèles de composants (par exemple par Fractal [BRU 04] et MALEVA, voir à la section 5.8, mais pas par JavaBeans [SUN 06] ni par Corba Component Model (CCM) [OMG 06b]).

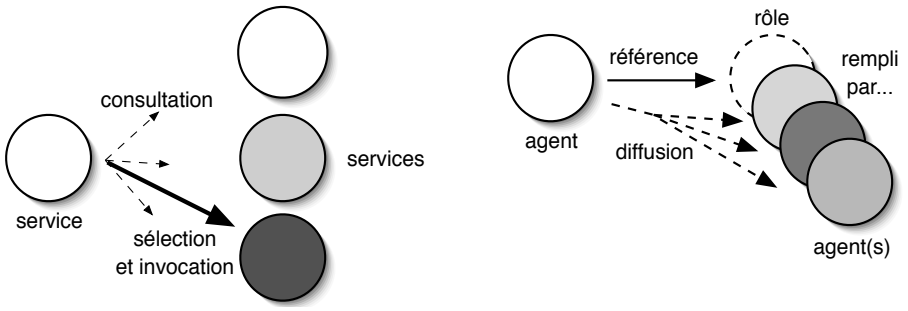

Figure 5.4. *Couplage entre services et couplage entre agents*

des assertions, à l'intérieur d'un programme, est de les rendre visibles à l'extérieur du composant, par l'intermédiaire des interfaces et d'énoncer ainsi des propriétés pouvant engager plus d'une entité logicielle [MEY 92].

Le concept de *service* des *architectures à base de services* (*service oriented architectures (SOA*), dont en particulier les services web [CHA 02], apporte une dynamicité *via* un mécanisme de découverte et de sélection dynamiques de services (illustré dans la partie gauche de la figure 5.4). Les services font l'objet de descriptions plus ou moins élaborées, qui sont mises à disposition (« *published* »), par exemple par des services d'*annuaires* analogues aux « pages jaunes » des annuaires téléphoniques. Pour les services web, les normes UDDI (pour *Universal Description, Discovery and Integration*) et WSDL (pour *Web Services Description Language*) spécifient respectivement les annuaires et les descriptions des services. Un service donné (par exemple, un service d'agence de voyage électronique), à la recherche de services pour assurer certaines sous-tâches (réservation de vols, d'hôtels...) va ainsi pouvoir identifier puis choisir, en général selon différents critères (disponibilité, prix, flexibilité...) et contracter des sous-services. Le couplage entre entités n'est ainsi plus uniquement pris en charge par le concepteur de l'application, mais par les entités elles-mêmes.

Les systèmes multi-agents poussent encore plus loin cette organisation du couplage et de la collaboration entre entités (coordination, décomposition, négociation...) à l'aide de connaissances (organisations, tâches, plans, protocoles...). En effet, plutôt qu'un couplage au départ principalement *syntaxique* (discipline des types) entre les composants, les systèmes multi-agents proposent un couplage *sémantique* guidé par les *connaissances* et par une *organisation sociale du travail*. Un concept fondamental, et d'un niveau plus axé connaissances que le concept d'architecture logicielle, est le concept d'*organisation*. Une organisation liste les différents *rôles* la constituant (par exemple, rôles de producteur, de consommateur, d'intermédiaire...), et leurs *relations* (dépendance, hiérarchie...). Un rôle donné pourra être joué par un ou plusieurs agents et un même agent pourra aussi éventuellement jouer simultanément plus d'un rôle.

Des exemples de modèles d'organisations sont AGR [FER 98] et MOISE+ [HUB 02]. Cassiopée est une des premières tentatives de méthodologies d'analyse de systèmes multi-agents centrée sur la notion d'organisation [DRO 98].

L'abstraction de l'agent vers le rôle permet une description plus générique de l'architecture et également des rapports et interactions entre les agents. Un agent référençant un rôle induit ainsi une référence indirecte et implicite[6] vers l'ensemble des agents remplissant (au moment de l'interaction) ce rôle (voir la partie droite de la figure 5.4). En ce sens, le couplage structurel est externe, comme pour les composants, mais gagne un degré d'implicite, à l'inverse des connexions explicites entre composants[7]. De plus, comme pour les services, les systèmes multi-agents utilisent aussi souvent divers mécanismes de mise en relation dynamiques et indirects, par exemple, par l'intermédiaire de la consultation d'agents *intermédiaires*, d'agents *annuaires*, ou encore d'agents « *facilitators* » (par exemple en KQML [FIN 97]) guidés par le contenu du message. La mise en relation, notamment pour la sélection et la contractualisation de partenaires en vue de traiter des tâches, peut également s'effectuer par des mécanismes d'appels d'offre (avec en particulier le classique *contract net protocol* [SMI 80], illustré à la figure 5.6). Les relations entre les agents sont donc très dynamiques et gérées en partie par eux-mêmes ou *via* les organisations.

Deux capacités importantes des organisations sont en effet la *dynamicité* et l'*autonomie* (*auto-organisation*) de l'organisation, qui peut évoluer en fonction des besoins (soit guidée par le haut, soit à l'initiative des agents eux-mêmes). Un exemple ludique, dans le domaine du football (humains ou robots), est la réorganisation d'une équipe de footballeurs selon une stratégie plus défensive, par exemple venant de l'entraîneur ou du capitaine d'équipe [HUB 02]. Un autre est la formation (puis la dissolution) dynamique d'une mini-organisation d'attaque de type « *une-deux* », cette fois à l'initiative d'un joueur [DRO 98].

La communauté des architectures logicielles et des composants aborde également ces enjeux de dynamicité et certaines capacités de reconfiguration automatique de l'architecture, notamment pour des applications nomades [DUB 06]. Cependant l'approche connaissances et sociale, caractéristique des systèmes multi-agents, reste (pour le moment du moins, voir le paragraphe 5.6.1.1) l'apanage des systèmes multi-agents. L'approche multi-agent est plus ambitieuse, mais elle est de ce fait également plus difficile à vérifier. On retrouve ainsi un dilemne classique entre, besoins croissants de

6. Ce mécanisme de désignation implicite du receveur sera analysé au paragraphe 5.4.2.1.

7. Cependant, d'un point de vue architectural, les agents ne possèdent en général qu'un seul point d'entrée, comme pour les objets, qui servent d'ailleurs souvent de support à leur implantation. De ce fait ils ne sont pas compositionnels, à la différence des composants. Par contre les organisations sont compositionnelles.

flexibilité et donc de délégation de l'initiative, et besoins d'assurer certaines garanties sur le fonctionnement du système.

5.4.2. *Couplage de communication*

L'expression du mode de communication entre entités logicielles comprend plusieurs caractéristiques importantes. Nous considérons ici les trois principales :

– la manière de désigner le ou les receveur(s), par exemple : point à point, multi-point, indexé par le contenu, *via* l'environnement. . . ;

– le mode de transfert des données, par exemple : unidirectionel, ou bidirectionnel avec retour de valeur, *via* un espace partagé. . . ;

– et le couplage temporel (autrement dit la synchronisation des communications), par exemple : synchrone, asynchrone, avec réponse anticipée, coordonné par un protocole. . .

5.4.2.1. *Désignation du receveur*

Le mode de communication entre les objets est fondamentalement *point à point*, c'est-à-dire un à un et en désignant explicitement le receveur du message. Les composants introduisent une communication à la base *multi-point*, du fait qu'une sortie d'un composant peut être connectée à plus d'un composant. Un type de connecteur intéressant est le connecteur de diffusion d'événements, correspondant au style architectural *publish-subscribe* [SHA 96]. Il offre une gestion indirecte et dynamique des connexions par les composants eux-mêmes, par l'intermédiaire d'un *abonnement* d'un composant au diffuseur d'événement. Ce type de mécanisme[8] s'est beaucoup répandu en dehors même du monde des composants, par exemple dans les applications à base d'objets traditionnels, mais il est selon nous une incarnation du concept et de la manipulation du connecteur, de manière externe aux entités. Enfin, le style architectural des espaces partagés (*repositories*), incarné par exemple par les tableaux noirs et les *tuple-spaces* (par exemple le modèle LINDA [GEL 92]), introduit un mode de désignation du receveur totalement implicite, puisqu'il sera indexé par le contenu même du message. Dans ce modèle, des entités actives (processus, agents) peuvent insérer des données structurées et indexées dans l'espace partagé. Elles seront consommées de manière opportuniste par des entités actives en attente du patron correspondant de données.

Les services, et plus encore les systèmes multi-agents, généralisent le mécanisme de désignation indirecte et dynamique, comme nous l'avons vu au paragraphe 5.4.1,

8. Le critère d'abonnement et le mode de diffusion peuvent varier, voir la classification de [EUG 03].

par l'intermédiaire de la consultation d'agents intermédiaires ou annuaires. Un service ou agent peut alors ensuite sélectionner dynamiquement lui-même son interlocuteur. La sélection/redirection peut également être automatique et implicite, Un premier exemple est le mécanisme de *facilitators*, guidés par le contenu du message [KAF 98] (par exemple en KQML [FIN 97], voir les langages de communication d'agents à la section 5.5). Un autre est l'*indexation* des communications selon un *rôle*. Ainsi par exemple, dans le modèle organisationnel AGR (agent groupe rôle) [FER 98], un agent peut s'adresser à un rôle, et l'ensemble des agents remplissant à cet instant ce rôle recevront alors la communication (voir la partie droite de la figure 5.3).

Enfin, dans certains types de systèmes multi-agents, dans lesquels l'environnement (physique ou non)[9] est modélisé explicitement, les agents peuvent communiquer *via* l'*environnement*, en laissant des données spécifiques, par exemple des phéromones pour des fourmis. Il existe d'ailleurs un courant actuel visant à promouvoir l'environnement d'un système multi-agent comme une abstraction privilégiée [WEY 05].

5.4.2.2. *Transfert de données*

Le mode de transfert des données de la programmation par objets est *bidirectionnel*, avec un *retour de valeur*[10]. Il est hérité de l'appel procédural ou fonctionnel. Il correspond (comme nous le verrons au paragraphe 5.4.2.3) à un appel *synchrone*, c'est-à-dire où l'émetteur suspend son activité en attendant le retour de la fin du traitement de la requête par le receveur.

Le modèle des acteurs [AGH 86] introduit un appel *unidirectionnel* (et *asynchrone*, voir au paragraphe 5.4.2.3). Cette fois, le transfert souhaité ne s'effectue que de l'émetteur vers le receveur. Si le receveur veut renvoyer une valeur, elle doit s'effectuer explicitement par un autre envoi de message.

Les modèles de composants, tels par exemple le *Corba component model* (CCM) [OMG 06b], proposent souvent simultanément ces deux types de transferts : *bidirectionnel* par appel de procédure (*via* des interfaces d'entrée et de sortie, appelées par CCM *facettes* et *réceptacles*), et *unidirectionnel* par diffusion d'*événements* (*via* des *puits* et *sources* d'événements), voir la figure 5.5.

Le style architectural des espaces partagés (voir le paragraphe 5.4.2.1) introduit un transfert de données, cette fois indirect, *via* une structure intermédiaire et la distinction entre production et consommation.

---

9. Des algorithmes à base de fourmis et leurs phéromones peuvent être utilisés comme méthode générale d'optimisation (voir à ce sujet le chapitre 4 « Agents situés : une nouvelle voie pour le développement d'applications industrielles »), l'environnement n'ayant alors plus de lien avec une réalité physique.

10. Sauf si le programmeur signale explicitement qu'il n'y a pas de valeur retournée, par exemple en Java à l'aide du type de données spécial `void` qui représente l'absence de données.

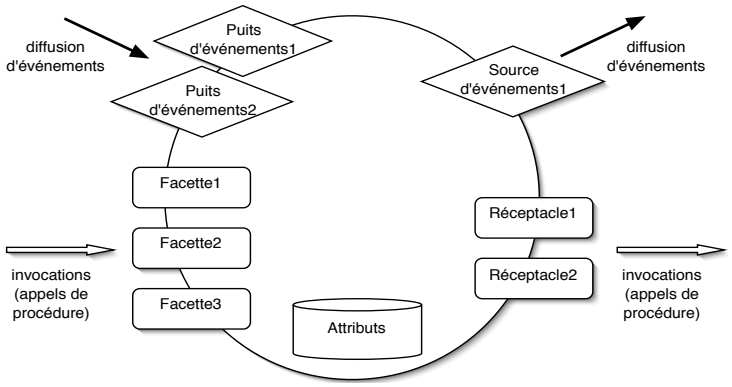

Figure 5.5. *Le modèle de composant CCM*

Les services reposent en général sur des modes et protocoles d'invocation assez simples, en particulier pour les services web, avec par exemple le protocole SOAP [CHA 02]. Une des raisons du succès des services web repose d'ailleurs probablement sur leur mise en œuvre facile au dessus d'une technologie éprouvée et répandue sur le web telle que HTTP. SOAP (*simple object access protocol*) inclut à la fois un mode bidirectionnel et un mode directionnel.

Les agents reprennent en général le mode unidirectionnel (et asynchrone) des acteurs, comme nous le verrons à la section 5.5, par le biais de langages de communication d'agents très élaborés. Ils peuvent de plus spécifier de manière très fine les données, ou plutôt les connaissances, communiquées. Enfin, la communication éventuelle *via* un environnement (par ajout, enlèvement, ou consommation de données) représente un mode indirect de transfert de données.

5.4.2.3. *Synchronisation*

Le modèle de communication original entre entités logicielles (à l'origine, dans un monde séquentiel et centralisé) est du type *appel procédural* ou *fonctionnel* avec *retour de valeur*. L'émetteur est suspendu pendant le traitement de la requête par le receveur. La transposition directe dans un monde concurrent conserve ces principes, l'émetteur attendant la complétion de l'appel. On parle alors de transmission (ou d'envoi de message) *synchrone*. Le passage à un monde distribué ne remet pas non plus *a priori* en cause ces principes, comme en témoigne le RPC (*remote procedure call*), également synchrone.

Le modèle des acteurs [AGH 86] introduit un mode de communication *asynchrone*, c'est-à-dire sans attente du traitement du message, ni même de la réception du message, par le receveur. L'envoi asynchrone s'applique donc mieux au modèle

de calcul concurrent (chaque acteur étant actif, cela évite l'attente de la conclusion du traitement par le receveur) et distribué (du fait de la relative latence du réseau de communication, cela évite l'attente de la livraison du message au receveur). Le modèle des acteurs suppose l'existence d'une *boîte aux lettres* pour chaque acteur, qui stockera les messages selon l'ordre d'arrivée (discipline de type FIFO). Le modèle des acteurs introduit donc un *découplage temporel* fin entre *envoi*, *réception*, *début du traitement*, et *complétion du traitement* du message.

La communication entre agents hérite en général de l'envoi de message asynchrone (et unidirectionnel) des acteurs. Mais, les langages de communication d'agents, en particulier FIPA ACL [FIP 02], permettent souvent de spécifier un protocole associé à une communication. Le protocole systématise ainsi la spécification des échanges de messages valides et la synchronisation entre les agents. La spécification du couplage temporel est ainsi beaucoup plus générale et concerne un nombre arbitraire de messages et d'agents. Un exemple classique de protocole multi-agent est le protocole *contract net protocol* (appel d'offre), spécifié par la FIPA [FIP 02] (voir la figure 5.6). Il existe diverses familles de protocoles multi-agents, par exemple : d'interaction, de coordination, de négociation, d'enchères. Il existe d'ailleurs actuellement des travaux visant à doter les services web de tels mécanismes de coordination (on parle alors d'orchestration de services).

Le tableau 5.2 résume l'évolution du couplage selon les 3 facettes, et la figure 5.7 la replace dans notre référentiel.

5.5. Abstraction

L'histoire de la programmation débute avec des concepts très proches de la machine (instruction, nombre entier...), puis identifie un certain nombre d'abstractions de plus en plus haut niveau (procédure, fonction, structure de donnée, sémaphore, processus, objet, message, composant, modèle...)[11]. Les concepts d'agent et d'organisation s'inscrivent dans cette évolution vers plus d'abstraction, et plus de connaissances, mais aussi vers plus d'explicite. Considérons l'évolution des données manipulées et en particulier échangées.

Le passage de types de données primitives à des structures de données arbitraires permet de représenter et nommer explicitement des données, se rapprochant ainsi des besoins de l'application. La programmation par objets représente une étape majeure de cette évolution, avec des objets informatiques représentations conceptuelles des objets physiques ou conceptuels du monde applicatif envisagé [PER 98][12]. On est donc

11. Voir également à ce sujet le chapitre 6 « Des systèmes multitâches aux systèmes multi-agents », comparant les concepts de tâches et d'agents.

12. Un bilan récent sur leurs réussites, échecs et perspectives, est proposé dans [PER 04].

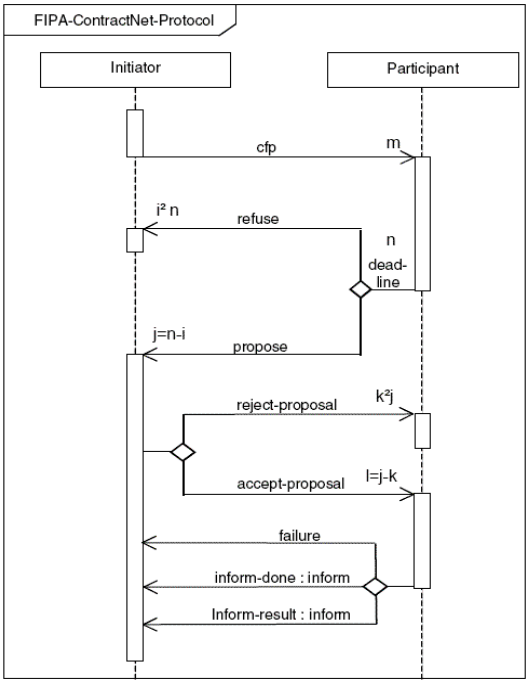

Figure 5.6. *Le* contract net protocol

passé des *données* aux *concepts*. Les agents prolongeront cette évolution avec l'explicitation de *connaissances*. Les agents cognitifs introduisent en effet la notion d'*état mental* (héritée de l'intelligence artificielle symbolique) [SHO 93], avec une représentation symbolique de concepts cognitifs, tels : *croyance*, *but*, *désir*, *intention*... Ces connaissances internes peuvent être communiquées (communication de croyances, de plans, d'intentions...) pour permettre par exemple aux agents d'apprendre ou de se coordonner.

La programmation par objets et l'envoi de message apportent plus d'auto-documentation, en indiquant de manière explicite l'objet et le type de requête. Les langages de communication d'agents apportent encore plus d'explicite et d'abstraction. Ainsi, des informations qui restaient *implicites* dans des applications à objets ou composants (intention des communications, logique de coordination, plans...) deviennent *explicites*, et ainsi documentent mieux le programme. De plus, ces informations pourront également être éventuellement utilisées par les agents eux-mêmes (par exemple pour se coordonner, raisonner sur des échecs de communication, replanifier collectivement, se réorganiser...).

Couplage	*Objets*	*Acteurs*	*Composants*	*Services*	*Agents*
Structure	Implicite interne *(références)*	Implicite interne *(références)*	Explicite externe *(connecteurs)*	Implicite volatil (invocations)	Implicite externe *(rôles)*
Communication					
- Désignation du receveur(s)	Point à point explicite	Point à point explicite	Multi-point explicite ou implicite *(publish-subscribe)*	Multi-point dynamique (découverte et sélection)	Multi-point explicite ou implicite *(indexé par les rôles)*
- Transfert de données	Bidirectionnel *(retour valeur)* direct	Unidirectionnel direct	Bi- ou uni-directionnel *(événements)* direct	Bi- ou uni-directionnel	Unidirectionnel direct ou indirect *(environnement)*
- Synchronisation	Synchrone	Asynchrone	Synchrone ou asynchrone	Synchrone ou asynchrone	Asynchrone protocole

Tableau 5.2. *Nature du couplage*

Comparons les *intergiciels* (en anglais, *middleware*) d'interoperabilité, qui spécifient et standardisent les échanges d'information. L'intergiciel à objets CORBA de l'OMG [OMG 06a] standardise, par l'intermédiaire d'un IDL : *interface description language*, les types de données échangées. L'équivalent pour les agents, spécifie de manière encore plus fine et précise les échanges d'information. Le simple IDL de CORBA est remplacé[13] par un véritable *langage de communication d'agents* (*agent communication language* : ACL). Le premier historiquement est KQML [FIN 97], suivi par l'ACL de la FIPA [FIP 02]. Outre le contenu propre du message, une communication en ACL peut spécifier :

– *performatif* : une désignation symbolique de l'intention de la communication (par exemple : `inform`, `deny`, `recruit`...) ;

13. Il nous faut préciser que CORBA et ACL ne jouent en fait pas exactement les mêmes rôles [VAN 00]. CORBA, à travers l'IDL, offre un standard de description des interfaces (signatures) des objets et des composants. Il existe des mises en correspondances (appelées *projections*) de cet IDL dans différents langages de programmation (par exemple, Java, Smalltalk, C++...). CORBA peut générer des squelettes d'implantation pour le code de l'appelant et le code de l'appelé, et ainsi assurer la transformation et le transfert des données. ACL lui ne propose pas un standard d'interfaces des agents, mais un standard de protocole de communication, ce qui est différent. ACL standardise les types d'invocations (performatifs), les ontologies et les protocoles d'interaction, voir ci-dessous.

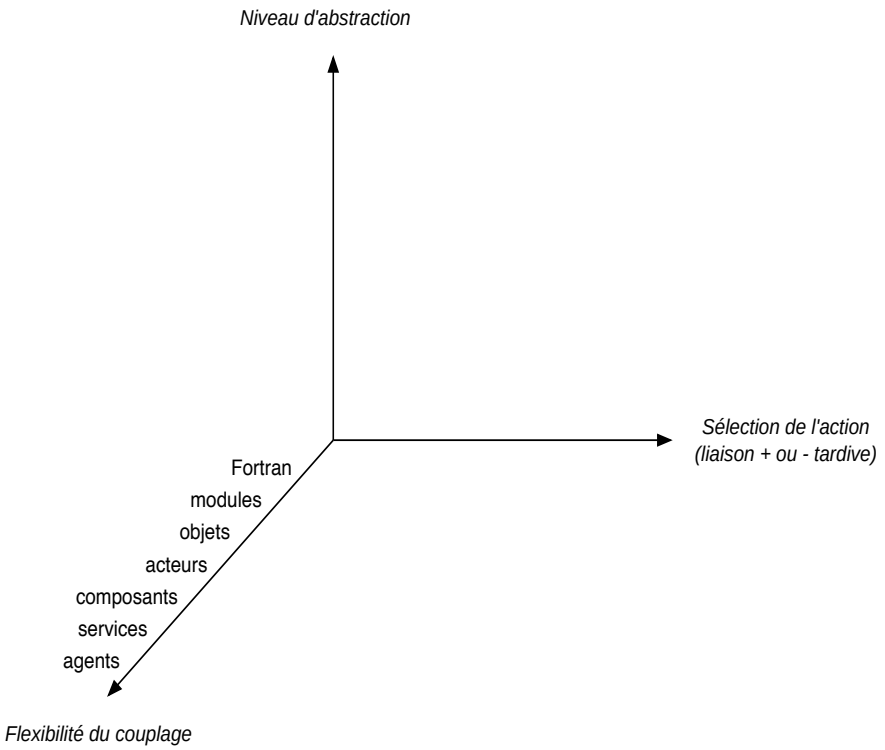

Figure 5.7. *Evolution de la la flexibilité du couplage*

– *langage de description du contenu* : le langage utilisé pour décrire le contenu. Cela peut être un langage de programmation (par exemple Java) ou un langage de représentation de connaissances adapté (par exemple KIF, ou SL [FIP 02]) ;

– *ontologie* : l'*ontologie* des concepts du message (par exemple une ontologie standard sur les transports et services touristiques, pour une application d'agence de voyage électronique) ;

– *protocole* : le protocole utilisé pour la communication (par exemple, le classique protocole d'appel d'offre, appelé `FIPA-Contract-Net`, voir la figure 5.6).

Il faut également souligner le rôle prépondérant de la *conception* pour les systèmes multi-agents. Elles est guidée par l'*organisation du travail* (les concepts d'organisation, de rôle et de dépendance) et par les *connaissances* (états mentaux tels que les intentions), plutôt que par les moyens de réaliser au final ce travail, ce qui correspond à l'approche procédurale traditionnelle de la programmation (données et procédures).

Des propositions de méthodologies multi-agents (en particulier le précurseur Cassiopée [DRO 98]) se basent ainsi souvent sur une analyse des organisations, des rôles et de leurs dépendances, et considèrent les questions de mise en œuvre (quels agents vont remplir les rôles, selon quel découpage, puis quelle implantation) de manière distincte et plus tardive. Une conception sous forme d'agents peut ensuite être réalisée (implantée) sous la forme d'agents, avec des architectures adaptées, ou bien sous la forme d'objets, d'acteurs, ou/et de composants, le niveau agent n'apparaissant ainsi pas toujours nécessairement complètement au niveau de l'implantation[14].

Enfin, dans l'évolution du niveau d'abstraction de la programmation, résumée par la figure 5.8, nous souhaitons également signaler le renouveau actuel de la modélisation comme guide de la construction d'applications (l'*ingénierie des modèles*, en anglais : *model driven engineering*, telle que la *model driven architecture* (MDA) proposée par l'OMG [OMG 06c]). Ce courant est distinct et non spécifique aux systèmes multi-agents, mais il existe des efforts croissants visant à coupler conception multi-agent et ingénierie des modèles (voir par exemple [SIL 05]).

5.6. Apports mutuels

A la suite de cette analyse comparative entre les concept de composant logiciel et le concept d'agent et de système multi-agent, on peut alors légitimement se demander comment on peut, sinon combiner les deux concepts, du moins envisager des apports mutuels.

On peut considérer de manière duale :

– *un 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 ;

– *un apport des composants aux agents* : que l'on peut décomposer en deux niveaux :

- *au niveau du système multi-agent* : pour une aide à l'intégration, au « packaging » et au déploiement des systèmes multi-agents,

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

14. Le maintien d'abstractions telles que les agents, mais également les organisations, en tant qu'entités représentées explicitement au niveau de l'exécution, offre bien entendu des possibilités de manipulation dynamique par le programmeur, mais surtout par les entités elles-mêmes, et ainsi par voie de conséquence des capacités d'adaptation et d'auto-organisation.

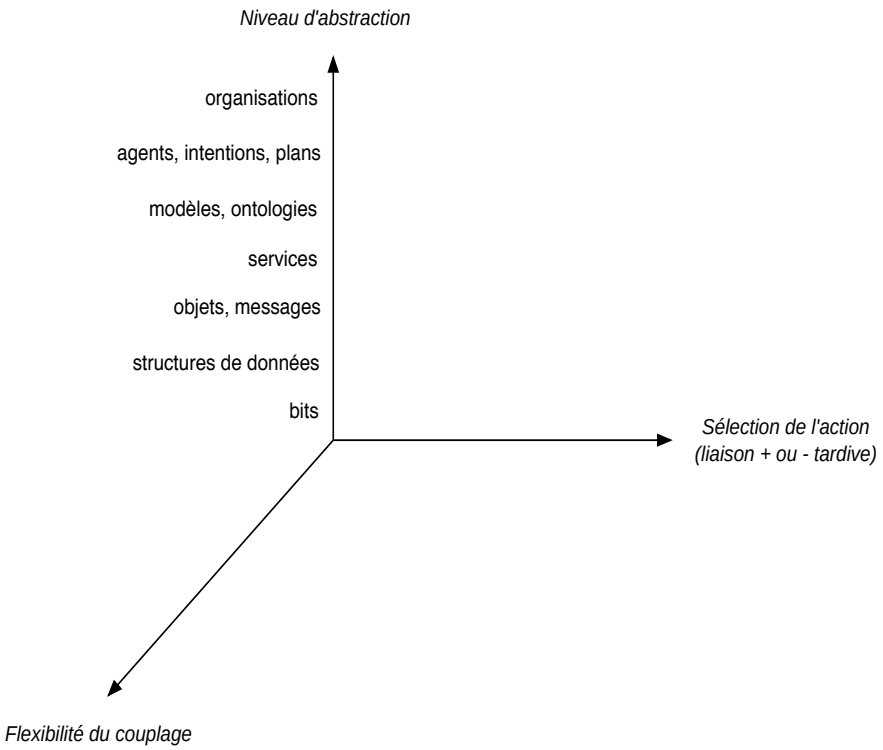

Figure 5.8. *Evolution du niveau d'abstraction*

5.6.1. *Vers des composants plus autonomes et adaptatifs*

Nous avons vu au à la section 5.2 que les systèmes multi-agents représentent une vision plus orientée connaissances des composants logiciels. Une voie extrême est alors de chercher tout bonnement à remplacer les composants logiciels par des systèmes multi-agents. Ceci n'étant pas toujours envisageable (coût de reconception et surcoût d'efficacité), on peut aussi envisager des inspirations et intégrations plus partielles.

5.6.1.1. *Assistance à l'assemblage*

Le choix des composants logiciels pour constituer une architecture est *a priori* manuel. Le concepteur doit donc identifier les composants souhaités, souvent à partir de bibliothèques, internes ou externes (« *sur étagères* »), et les assembler en fonction des objectifs souhaités. L'approche peut être plutôt ascendante, par construction

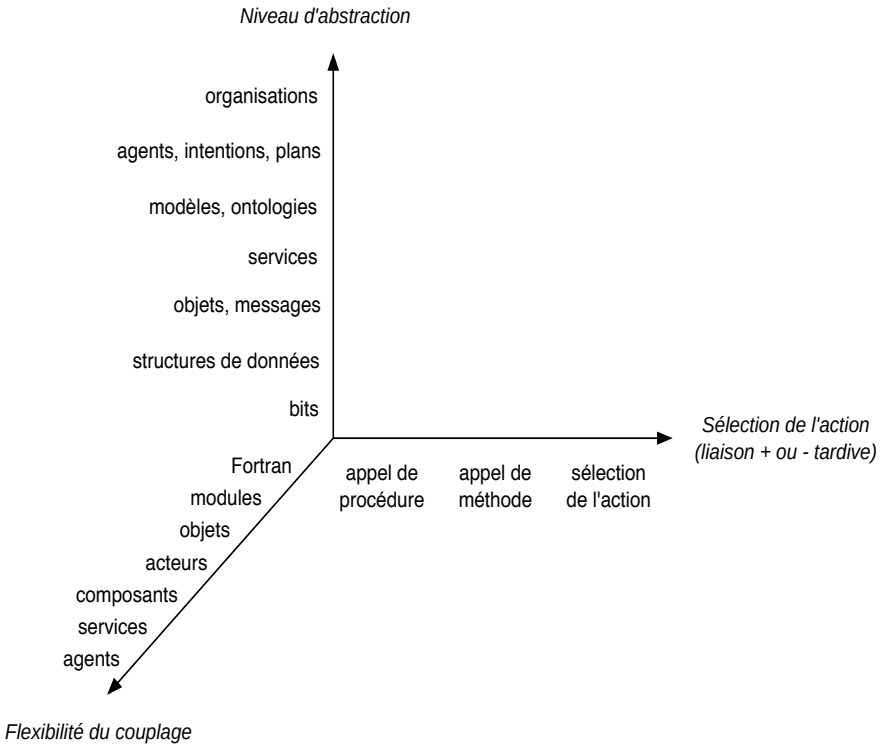

Figure 5.9. *Evolution de la programmation*

progressive d'assemblage de composants, ou plutôt[15] descendante, à partir d'une description abstraite de l'architecture souhaitée, en général spécifiée dans un langage de description d'architecture (ADL, voir au paragraphe 5.4.1).

Le typage des interfaces offre certaines possibilités de vérification de la compatibilité des compositions, mais n'est pas en général un guide très pertinent, les types de données restant souvent de bas niveau. D'une part, une compatibilité des types ne garantit pas pour autant la compatibilité sémantique (« components that plug but just don't play » [NOR 01]). D'autre part, les types peuvent être incompatibles mais en pratique aisément traduisibles (par exemple pour différents formats numériques) *via* l'insertion d'adaptateurs traducteurs. Les contrats comportementaux *via* des assertions

15. En général une combinaison (« *aller-retour* ») entre les deux.

(voir au paragraphe 5.4.1) sont d'un niveau plus sémantique, mais restent plus focalisés sur les hypothèses de fonctionnement correct que sur une véritable description des services offerts ou requis par un composant.

Une direction est en conséquence d'offrir des services d'assistance à l'assemblage (identification, mise en correspondance, assemblage effectif) à partir d'informations sur « *ce que font* » les composants ou/et à partir d'objectifs plus globaux de l'application. Une approche sur des *ontologies*, c'est-à dire de classification de concepts, est une voie relativement pragmatique. En effet, chaque approche de description d'un composant (syntaxique, comportementale, ontologique, formelle) a ses avantages et limitations, en matière de précision, de pouvoir d'expression, et de facilité d'utilisation.

Un exemple d'une telle approche est le modèle de *mise en correspondance* (*match-making*) d'agents LARKS [SYC 02]. Il est relativement élaboré et est fondé sur différents niveaux de descriptions (signatures-types, comportementaux-contraintes, ontologiques-concepts...), différents types de mesures de similarité, et différents types de politiques de mise en correspondance (exact, inclusion, relâché). LARKS a été conçu pour la recherche et la mise en correspondance d'agents logiciels sur Internet, mais l'approche suivie est en fait plus générale et peut aussi s'appliquer à des composants logiciels ou encore à des services (par exemple des services web, en étendant leur modèle de description, qui repose actuellement sur le standard WSDL [CHA 02]). Le projet et modèle de composant COGents est un autre exemple d'assistance à l'assemblage de composants logiciels [BRA 04]. Son domaine d'application est la conception assistée et la simulation de processus pétrochimiques. Le projet illustre bien la progression d'une approche au départ objet, vers une approche composant, puis vers une approche agent. Un premier projet, nommé CAPE-OPEN, a consisté en la rationalisation sous forme de bibliothèques de composants de divers types de logiciels (modèles physiques, solveurs, etc.) à travers des standards d'interfaces et des intergiciels d'interoperabilité tels que CORBA. Le projet suivant, nommé COGents, a ensuite défini une ontologie standard, nommée ONTO-CAPE, et un mécanisme de mise en correspondance, dans la lignée de LARKS [BRA 04].

5.6.1.2. *Auto-reconfiguration de l'architecture*

La conception et la construction d'une architecture logicielle reste encore en général réalisée statiquement à la conception. Or les nouvelles applications dynamiques et ouvertes, telles que l'informatique nomade ou ubiquitaire, engendrent des besoins d'*adaptation* et de *reconfiguration dynamiques* [RIV 04], au niveau de l'architecture, comme au niveau d'un composant. Ceci amène une évolution actuelle de mécanismes d'adaptation, au départ essentiellement réactifs à partir d'évènements [DUB 06], vers des mécanismes de plus haut niveau, à partir de contraintes et de politiques, voire de buts et de plans. Une illustration d'un tel objectif est le modèle abstrait MaDcAr d'assemblage et réassemblage dynamique et automatique d'applications à base de composants [GRO 06]. A partir de configurations existantes de composants, de contrats

de compatibilité et de politiques d'assemblage, MaDcAr gère la reconfiguration d'assemblages. Il faut noter que ce modèle peut *a priori* être appliqué aussi bien au niveau de l'architecture d'une application qu'à un niveau de granularité très fin, comme celui du réassemblage de composants d'un seul agent. Enfin, les besoins d'adaptation concertée de différents niveaux et endroits (composants) d'adaptation, pour minimiser les risques d'incohérence, nous semblent militer à terme pour une approche des mécanismes de reconfiguration de plus en plus cognitive (planification) et coopérative (coordination), autrement dit, selon une approche multi-agent.

5.6.2. *Structuration et déploiement des agents par des composants*

Les concepts d'agent et de système multi-agent se présentant comme des avancées conceptuelles et technologiques par rapport aux composants, on pourrait *a priori* penser que les composants logiciels n'ont de ce fait plus grand-chose à leur apporter. Nous pensons le contraire et qu'une des raisons en est la maturité plus grande des composants en matière de structuration et de déploiement du logiciel [PES 05]. Nous développons plus particulièrement ce point ici.

La notion de composant peut en effet aider :

– *au niveau du système multi-agent*, à la construction, l'intégration, le *packaging* et le déploiement des systèmes multi-agents ;

– mais également *au niveau d'un seul agent*, à la structuration et la réutilisation de l'implantation de son architecture (ceci sera développé à la section 5.7).

5.6.2.1. *Déploiement d'agents*

Tout d'abord les composants logiciels offrent une base de structuration de l'implantation des agents[16], notamment en vue de leur déploiement et de leur documentation. Un exemple de proposition est [MEL 04] qui propose une implantation des agents sous la forme de composants CCM [OMG 06b]. Un des intérêts immédiats d'une telle approche est de bénéficier quasi gratuitement de nombreux services de base de répartition offerts par l'intergiciel associé Corba, tels l'identification, la découverte, plutôt que de les réimplanter « *from scratch* », ce qui avait été par exemple fait dans une plate-forme telle que RETSINA [SYC 03][17].

---

16. L'implantation interne d'un agent pouvant être effectuée à l'aide d'une architecture d'agent ou à l'aide d'objets, d'acteurs [GUE 99], ou de composants (voir la section 5.7).

17. *A contrario*, le modèle de mise en correspondance d'agents LARKS (introduit au paragraphe 5.6.1.1 [SYC 02]), conçu pour RETSINA [SYC 03], suit bien les principes des composants logiciels, en explicitant les interfaces d'entrée mais également de sortie d'un agent logiciel. Ceci n'est pas le cas de beaucoup d'architectures d'agents, qui restent encore souvent fondées sur une implantation objet classique.

De plus, des mécanismes d'aide au déploiement de composants peuvent être utilisés ou étendus. Un exemple est le mécanisme de déploiement associé au langage de description de composants OLAN [BEL 06], dans lequel différentes contraintes de déploiement pour chaque composant (taille mémoire minimale d'une machine, charge maximale, co-localisation avec un autre composant…) sont utilisées pour guider un déploiement de l'application. Le successeur d'OLAN utilise d'ailleurs une architecture à base d'agents réactifs (proches des acteurs), pour encapsuler et contrôler le déploiement des composants [QUE 03]. Ceci témoigne des influences croisées croissantes entre composants et agents.

5.6.2.2. *Conformité d'un agent à un rôle*

Un problème encore peu abordé, mais qui nous semble pourtant un enjeu réel, est de garantir, ou du moins *estimer*, qu'un agent qui va être amené à jouer un rôle sera effectivement en mesure de l'accomplir. Ce problème a été identifié dans [FER 00] comme le problème de l'*admission* d'un agent dans (et par) un groupe, et formulé comme un problème normatif et organisationnel. Les auteurs esquissent comme piste l'utilisation de sanctions *a posteriori* si l'agent ne remplissait pas ses engagements. [FER 98] esquisse également plusieurs pistes de contrôle cette fois *a priori*, conditionnée par les compétences, par l'implantation, et par le dialogue ou la similarité avec les autres agents présents dans le groupe.

Nous pensons que, du point de vue des composants logiciels, ceci peut être reformulé à la base en partie comme le problème assez général de la *conformité* d'une entité logicielle (un agent) à une spécification quelconque (un rôle d'une organisation). Des vérifications classiques fondées par exemple sur des contrats syntaxiques ou comportementaux (voir au paragraphe 5.4.1), sont un possible point de départ. Cependant, du fait de la dynamicité et du possible indéterminisme du comportement d'un agent, elles ne sont pas suffisantes pour garantir la conformité. Il apparaît de toute manière difficile d'envisager une garantie complète, mais plutôt, sans se restreindre à de seules sanctions *a posteriori*, d'éliminer les incompatibilités faciles à vérifier, et d'estimer les capacités de l'agent à accomplir le rôle de manière satisfaisante. Une voie indirecte mais effective est proposée par le modèle d'organisation MOISE+ [HUB 02]. Il vérifie si l'agent ne joue pas déjà d'autres rôles et dans ce cas que le nouveau rôle n'est pas incompatible avec les actuels. L'aspect vérification dynamique peut être abordé par une approche de test d'intégration de composants logiciels, et par monitoring des actions de l'agent et vérification que ses actions sont conformes aux lois sociales régissant un rôle à l'intérieur d'une organisation. Une architecture de monitoring automatiquement générée à partir de normes spécifiées dans une logique déontique [CHO 06] nous semble de ce point de vue un point de départ intéressant.

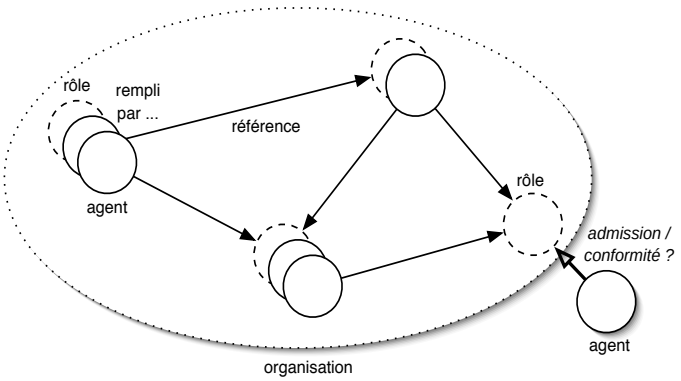

Figure 5.10. *Conformité d'un agent à un rôle*

5.7. Architectures d'agents à base de composants

La notion de composant logiciel (et d'architecture logicielle) nous semble utile pour éclairer et aider la construction modulaire non seulement d'un système d'agents mais également d'un agent individuel. En effet, nous pensons que la conception et la construction d'un agent peut bénéficier des principes des composants logiciels (encapsulation, interfaces de sortie, connecteurs explicites...). Pour mieux situer ce débat, nous proposons tout d'abord une analyse des critères et approches de décomposition de l'architecture d'un agent.

5.7.1. *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). » [BOI 01, page 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 [SHA 96], nous proposons ici 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 [BOI 01, MUL 97, WOO 95], par ex : architectures horizontales/verticales, ou encore réactives/cognitives/hybrides. Nous nous focalisons ici

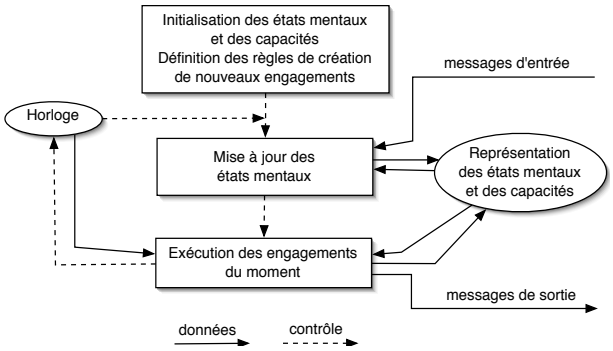

Figure 5.11. *Architecture AOP*

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 détaillerons 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 [BOI 01]. Enfin, notez que, comme pour les architectures logicielles [SHA 96], une architecture complexe (par exemple : InteRRaP, voir au paragraphe 5.7.4) peut juxtaposer et combiner différents styles architecturaux.

5.7.2. *Décomposition selon le cycle d'activation*

Ce principe de décomposition, et le style architectural associé, un des plus simples, suit le *cycle d'activation* de base d'un agent : *perception* (des messages ou/et de l'environnement), *mise à jour* de l'état (données ou connaissances-état mental), génération des *engagements* (d'actions), *action(s)*. Un premier exemple est une architecture minimale d'agents réactifs situés. Un autre exemple, pour un agent cognitif et communicant, est l'architecture *agent-oriented programming* (AOP), de Yoav Shoham [SHO 93] (voir la figure 5.11, adaptée de [SHO 93]).

5.7.3. *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 (exemples : interaction, environnement, organisation…) et différents types de traitements associés (perception, communication, coordination…). Un exemple représentatif est l'architecture Volcano [RIC 01], qui suit les principes de décomposition de la méthodologie

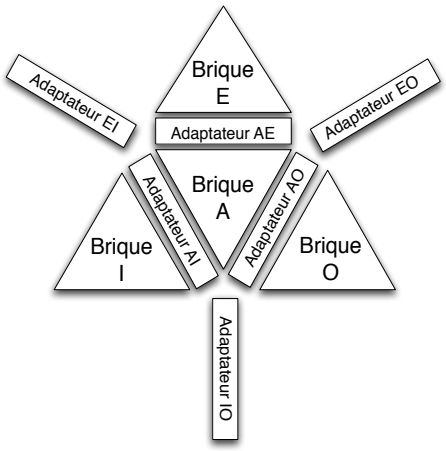

Figure 5.12. *Architecture Volcano*

Voyelles [DEM 95] en quatre dimensions : A (agent), E (environnement), I (interaction) et O (organisation). Cette architecture est en fait un *framework*, dont les composants correspondants (appelées *briques*) sont en conséquence : A, E, I et O. Mais il faut y ajouter les 6 adaptateurs entre briques (d'autres types de briques) : AE, AI, AO, EI, EO et IO (voir la figure 5.12, adaptée de [RIC 01]).

L'architecture MAST [VER 04] prolonge et améliore ce travail, en ajoutant la dimension U interface avec l'utilisateur, en permettant une redécomposition plus fine à l'intérieur de chaque dimension, et surtout 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 [BRA 95]. Ce projet a en particulier conçu un modèle d'architecture générique (*generic agent model* : GAM) [BRA 01], 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 rétroconception d'un certain nombre de modèles d'architectures d'agents générales ou appliquées (tels que BDI et ARCHON) [BRA 01].

5.7.4. *Décomposition selon les niveaux et 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

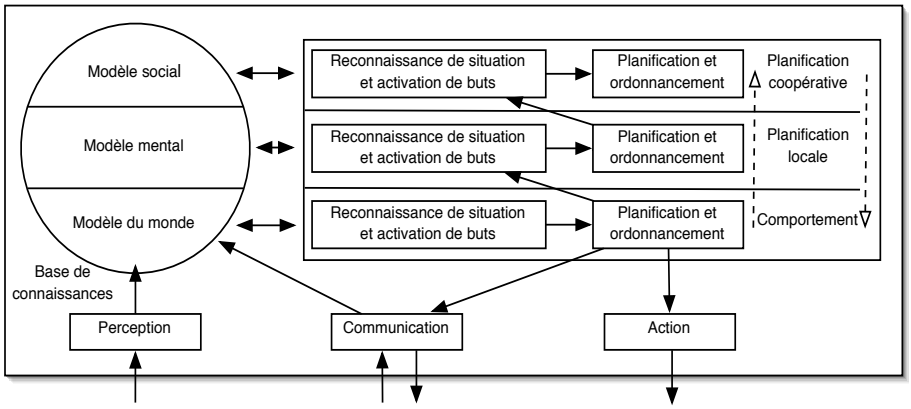

Figure 5.13. *Architecture InteRRaP*

exemple représentatif est l'architecture hybride InteRRaP [MUL 93], qui distingue ces trois niveaux de modèles (voir la figure 5.13, inspirée de [BOI 01]). InterRRaP est structurée selon une hiérarchie de trois niveaux[18], simultanément actifs : planification coopérative (modèle social), planification locale (modèle mental), et comportement réactif (modèle du monde). Deux mécanismes de contrôle et de communication entre les niveaux sont : requête d'activation ascendante, par lequel un niveau active le niveau supérieur quand il ne peut traiter la situation courante, et signal d'engagement descendant, par lequel un niveau exécute ses décisions en déléguant ses engagements au niveau inférieur [BOI 01].

5.7.5. *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*, dans laquelle différents comportements (exemples : mouvement aléatoire, évitement d'obstacle...) sont activés simultanément. Ils sont organisés selon une hiérarchie forte et figée et les priorités associées (voir la figure 5.14, inspirée de [BRO 86]). En pratique, un comportement peut en effet remplacer les données d'entrée du comportement immédiatement inférieur et également inhiber ses données de sortie

18. On peut remarquer que l'architecture de chaque niveau est similaire et est elle-même structurée en deux modules généraux : reconnaissance de situation et planification. Il est également clair que, comme pour les styles architecturaux [SHA 96], une architecture élaborée (telle qu'InteRRaP) peut faire cohabiter et combiner différents principes de décomposition.

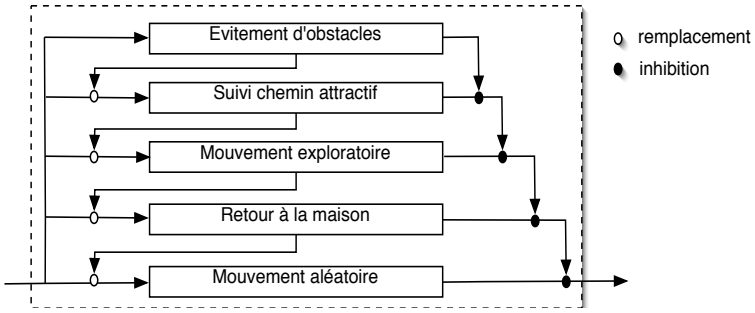

Figure 5.14. *Architecture de subsomption*

(par exemple, en cas de détection d'obstacle, le comportement d'évitement d'obstacles peut prendre la main).

5.7.6. *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 (agent cognitif collaboratif pour l'architecture InteRRaP, agent situé ou robot pour l'architecture de subsomption...). Elles s'affichent comme relativement génériques et flexibles. Il est pourtant 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 les 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 progrès, du fait de son modèle d'implantation à base d'événements et de la décomposition des briques en sous-composants. L'architecture de subsomption est plus abstraite, et ne représente en fait qu'un modèle d'architecture, instancié pour un robot et objectif donné. Mais une fois instanciée (voir la figure 5.14), il est ensuite délicat de la faire évoluer (par exemple de rajouter un comportement), car la hiérarchie est la clé implicite des rapports entre les composants.

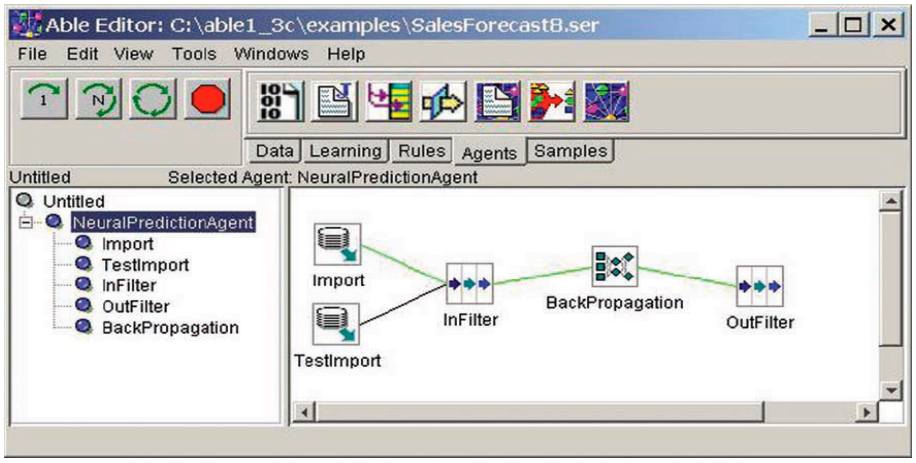

Figure 5.15. *Architecture ABLE*

5.7.7. *Architecture* versus *modèle de composant*

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. Une alternative radicale consiste alors à ne proposer qu'un *modèle de composant*, de manière analogue à un modèle de composant logiciel tel que celui des 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. Des exemples sont les modèles de composants Desire [BRA 95] (voir également au paragraphe 5.7.3), CBAF (*component-based agent framework*) [GOR 04] et ABLE [BIG 02].

L'architecture ABLE [BIG 02], proposée dans le cadre du programme *Autonomic Computing* d'IBM, offre un ensemble de composants outils, regroupés par types d'outils et de techiques : les *data beans*, comme `TimesSeriesFilter` pour des statistiques ; les *learning beans*, comme `BackPropagation` implémentant la rétropropagation des réseaux de neurones ; les *rule beans*, comme `FuzzyForwardChaining` pour de l'inférence floue, et les *specific beans*, comme `GenericSearch` pour un algorithme de recherche général. Tous ces composants sont implémentés à base 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 (voir la figure 5.15, provenant de [BIG 02]).

D'autre modèles de composants, tels qu'AgentTalk [KUW 95] et SCD [YOO 98], ont également été proposés pour la conception de protocoles multi-agents à partir

de spécialisation et de composition de protocoles ou de briques de protocoles plus simples.

Un autre exemple d'architecture modulaire, minimaliste, est l'architecture Magique [MAT 01]. Dans celle-ci, des compétences, formes de modules de comportements, peuvent être ajoutées dynamiquement aux agents. Egalement, un mécanisme de délégation des requêtes permet leur décentralisation au niveau du système.

Un exemple qui nous semble représentatif de l'application relativement systématique des concepts de composants à la conception d'un agent, est le modèle Maleva, car il applique les principes mêmes des composants et de composition logicielle, non seulement à la conception des comportements d'un agent, mais également à la spécification de leur contrôle, ainsi réifié *via* des bornes, des connexions, et également des composants spécifiques de contrôle. De plus, il suit une approche de décomposition selon les comportements (voir au paragraphe 5.7.5).

5.8. Le modèle de composant d'agent Maleva

L'objectif du modèle de composant d'agent Maleva [LHU 98] 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. Il a été plus particulièrement appliqué à la conception de comportements d'agents pour des simulations multi-agents [MEU 04], mais a en fait une portée plus générale [BRI 06].

5.8.1. *Flot de données et flot de contrôle*

Le modèle de composants d'agents Maleva [BRI 06, LHU 98, MEU 04] introduit une distinction entre le *flot de données* et le *flot de contrôle* entre les composants. Cette caractéristique, et une des spécificités de Maleva, permet de concevoir de manière explicite l'architecture de contrôle. 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, Maleva considère deux types de bornes 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) ;

– *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

	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 5.3. *Bornes de données et de contrôle*

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 *structurelle* standard (comme pour tout modèle de composant, voir au paragraphe 5.4.1) entre bornes d'entrée et bornes de sortie. Le tableau 5.3 résume le tout.

Une des originalités du modèle de composant Maleva est donc qu'il applique les principes mêmes des composants et de composition logicielle à la spécification du contrôle, ainsi réifié *via* des bornes, des connexions, et également des composants spécifiques de contrôle.

Le découplage entre communication (données) et activation (contrôle) permet une plus grande flexibilité de recomposition (éventuellement dynamique) des composants, en changeant tout ou partie des graphes de connexion, et en respectant l'encapsulation des composants. Les composants gagnent ainsi en généricité d'utilisation, leur définition étant plus indépendante de leur contexte d'activation (des exemples, de parallélisation ou de contrôle de l'ordonnancement, sont discutés dans [BRI 06]).

5.8.2. *Exemple*

Nous présentons ici brièvement un exemple de conception de comportements d'agents, de type proie et prédateur, pour des animaux virtuels, agents situés dans un éco-système.

Le 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 ses alentours en se déplaçant aléatoirement. En suivant une approche ascendante, nous construisons le comportement de proie `Prey` comme la composition de trois comportements

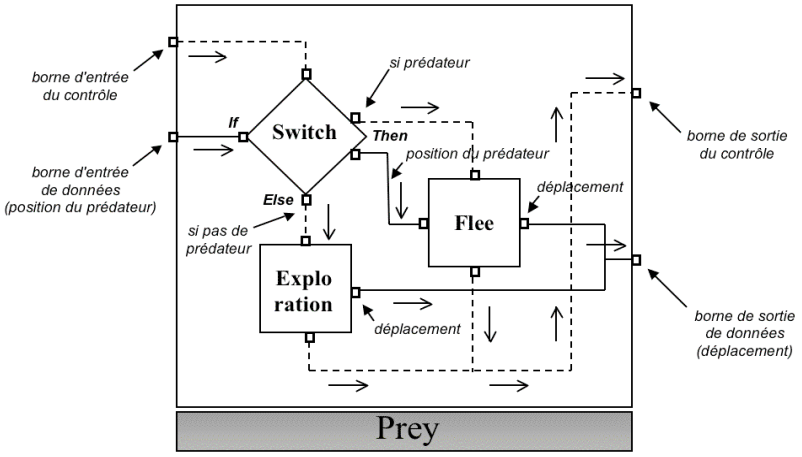

Figure 5.16. *Comportement de proie*

(composants) élémentaires suivants : fuir (`Flee`), exploration (`Exploration`), et un composant de contrôle nommé `Switch`[19].

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`)[20], 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 aléatoire transmis au final à la borne de sortie de `Prey`. L'architecture résultante est illustrée à la figure 5.16 (par convention, les connexions du flot de données sont en trait plein, et les connexions du flot de contrôle en pointillés).

Nous pouvons maintenant réutiliser le comportement de proie `Prey` pour concevoir et construire le comportement d'un prédateur `Predator` ainsi défini : il poursuit des

19. Le *composant de contrôle* `Switch` réifie la structure de contrôle conditionnelle en un composant primitif. La condition est ici la présence ou l'absence d'une donnée d'entrée.

20. 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, d'une architecture générale d'un agent situé [BRI 06].

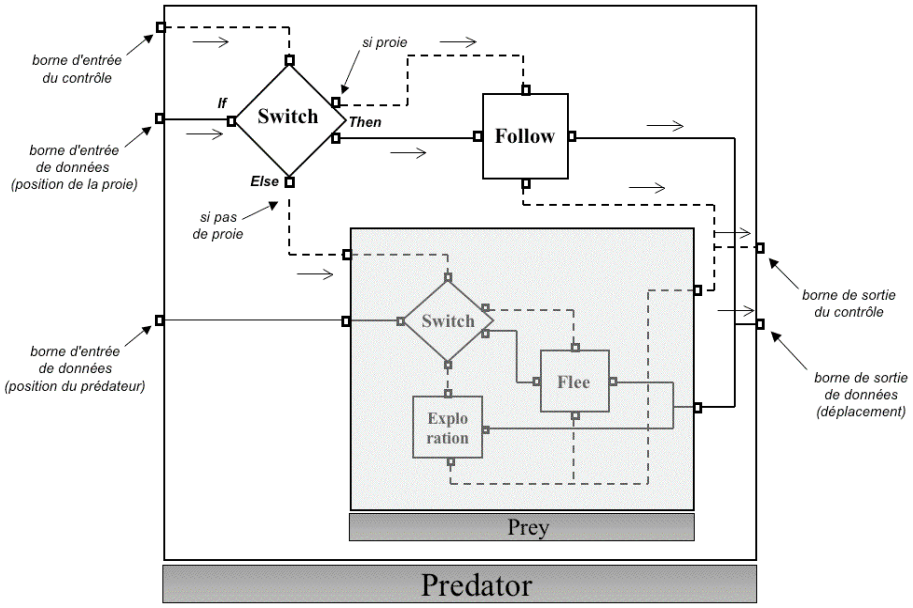

Figure 5.17. *Comportement de prédateur (incluant un comportement de proie)*

proies, tout en fuyant ses congénères prédateurs, et a un comportement d'exploration aléatoire tant qu'il ne perçoit aucun autre agent. En suivant une 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 5.17).

Le lecteur intéressé trouvera une description plus détaillée du modèle de composant d'agent Maleva ainsi que plusieurs exemples en [BRI 06] et dans les thèses de Marc Lhuillier [LHU 98] et de Thomas Meurisse [MEU 04].

5.9. Conclusion

Nous assistons actuellement à un mouvement dual de deux communautés. Du fait des besoins croissants de dynamicité et d'automaticité des nouvelles applications réparties dynamiques et ouvertes (telles que l'informatique ubiquitaire), les modèles de composants logiciels et d'architectures logicielles gagnent progressivement en niveau d'abstraction et en capacités d'adaptation et de reconfiguration (et d'auto-adaptation et auto-reconfiguration). Simultanément, du fait de leur succès, et de ce fait d'exigences de maturité industrielle, les modèles et plates-formes multi-agents gagnent en méthodes (méthodologies) et en outils de développement et de déploiement. La frontière

entre les approches devient ainsi progressivement plus ténue. Cependant, les spécificités culturelles font qu'il existe encore parfois, de part et d'autre, une certaine méconnaissance des travaux respectifs, et donc un risque réel de « *réinventer la roue* ». Ce chapitre a pour objectif de contribuer bien humblement à articuler et mettre en perspective les concepts et enjeux de ces communautés, de manière à favoriser différentes sortes et niveaux de fertilisations croisées.

Pour terminer, il nous faut rappeler l'arrivée récente d'un « *troisième larron* » dans le jeu déjà existant entre composants et agents, les *services web*[21]. Leur technologie est plus simple et plus légère à mettre en œuvre que certains modèles de composants répartis, puisqu'il suffit de l'infrastructure actuelle du web. Par contre, d'étendre les modèles existants assez simplistes[22], notamment vers les besoins de conformité de services et de coordination de services (souvent également appelée *orchestration de services*), restent des enjeux ouverts et dans lesquels il faudra veiller là aussi à intégrer et réutiliser au mieux les savoirs respectifs de ces différentes communautés[23].

Remerciements

Nous tenons à remercier David Hill pour ses commentaires détaillés et ses nombreuses suggestions sur ce texte. Ce chapitre a pour origine un exposé invité à la première Journée multi-agents et composants (JMAC) en 2004 à Paris [ROY 04]. Nous remercions ses organisateurs, et en particulier le président du comité de programme

21. Bien que discutés au cours de notre analyse comparative, nous avons volontairement restreint la description des concepts et travaux en matière d'*architectures à base de services* – et en particulier les services web. Par ailleurs, nous n'avons qu'effleuré le courant pourtant important, et relativement transversal, de l'*ingénierie guidée par les modèles* (*model driven engineering*, telle que proposée par l'OMG [OMG 06c]). Ce chapitre se focalise en effet sur les rapports entre les composants logiciels et les systèmes multi-agents, ce qui donne un chapitre déjà suffisamment étendu.

22. [PAY 08] propose une analyse comparative des services web et des agents.

23. En cela, nous partageons l'analyse suivante : « Automated component-based configuration of, e.g., software agents, entails a thorough understanding of both configuration processes, and the components to be configured. The agent community has made the most progress in automation of the configuration process. The Web service community has made the most progress in development and annotation and in the discovery and retrieval of these components. The software composition community has made the most progress in structuring and modelling components, and on supporting the human designer. Interdisciplinary research may be most fruitful, when (1) combining configuration-expertise with annotation-expertise, (2) generalising and standardising reusable, configurable, components, and (3) when the current structuring and modelling practices of SE are used. Once automated component-based configuration has proven to be feasible, research can focus on automated component-based adaptation as a new challenge. » [VAN 00].

Jean-Claude Royer, pour cette opportunité. Nous remercions Amal El Fallah Seghrouchni, pour son invitation à participer à cet ouvrage, ce qui nous a donné l'opportunité de développer plus avant cette réflexion. Ce travail représente aussi la continuation d'un travail avec Les Gasser sur les rapports entre les objets concurrents (et les acteurs) et les systèmes multi-agents [GAS 92, GAS 98] et nous le remercions pour ses nombreux apports à la présente analyse. Nous remercions également nos collègues ou/et anciens étudiants : Jacques Ferber, Zahia Guessoum, Alexandre Guillemet, Gregory Haïk, Marc Lhuillier, Thomas Meurisse, Frédéric Peschanski, et Min Jung Yoo, pour leurs contributions et nos discussions sur le thème des agents modulaires et des composants. Enfin, nous remercions Jean-François Perrot pour d'innombrables discussions et enseignements toujours lumineux sur les concepts fondamentaux et la pratique de la programmation (voir par exemple [PER 98] et [PER 04]). Nous lui dédions ce chapitre.

5.10. Bibliographie

[AGH 86] G. Agha, *Actors : a Model of Concurrent Computation in Distributed Systems*, Series in Artificial Intelligence, MIT Press, 1986.

[ALL 94] R. Allen et D. Garlan, Formal Connectors, *Reseach Report*, CMU-CS-94-115, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, États-Unis, mars 1994.

[BAC 00] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, et K. Wallnau, 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.

[BEL 06] L. Bellissard, S. Ben Atallah, A. Kerbrat, et M. Riveill, Component-based Programming and Application Management with OLAN, chapitre de *Object-Based Parallel and Distributed Computation*, édité par J.-P. Briot, J.-M. Geib, et A. Yonezawa, LNCS, No 1107, Springer, 1996, pages 290–309.

[BEL 01] F. Bellifemine, A. Poggi, et G. Rimassa, Developing Multi-Agent Systems with a FIPA-compliant Agent Framework, *Software Practice and Experience*, (31) :103–128, 2001.

[BEU 99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, et D. Watkins, Making Components Contract Aware, *IEEE Computer*, juin 1999, pages 38–45.

[BIG 02] J.P. Bigus, D.A. Schlosnagle, J.R. Pilgrim, W.N. Mills, et Y. Diao, ABLE : A Toolkit for Building Multiagent Autonomic Systems, *IBM Systems Journal*, 41(3) :350–371, 2002.

[BOI 01] O. Boissier, Modèles et architectures d'agents, [BRI 01, pages 71–107].

[BOI 06] O. Boissier (éditeur), Composants et systèmes multi-agents, Numéro spécial, *L'Objet*, 12(4), Hermès/Lavoisier, 2006.

[BOR 05] R. Bordini, M. Dastani, J. Dix, et A. El Fallah Seghrouchni (éditeurs), *Multi-Agent Programming : Languages, Platforms and Applications*, International Book Series on Multiagent Systems, Artificial Societies, and Simulated Organizations, Springer-Verlag, 2005.

[BOR 06] R. Bordini, L. Braubach, M. Dastani, A. El Fallah Seghrouchni, J.J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, et A. Ricci, A Survey of Programming Languages and Platforms for Multi-Agent Systems, *Informatica*, 30 :33–44, 2006.

[BRA 04] B. Braunschweig, E. Fraga, Z. Guessoum, W. Marquardt, O. Nadjemi, D. Paen, D. Piñol, P. Roux, S. Sama, M. Serra, I. Stalker, et A. Yang, CAPE Web Services : The COGents Way, Actes du *European Symposium on Computer-Aided Process Engineering*, édités par A.P. Barbosa-Póvoa et H. Matos, Elsevier, 2004.

[BRA 95] F. Brazier, B. Dunin-Keplicz, N. Jennings, et J. Treur, Formal Specification of Multi-Agent Systems : a Real-World Case, Actes de *1st International Conference on Multi-Agent Systems*, San Francisco, CA, États-Unis, MIT Press, 1995, pages 25–32.

[BRA 01] F. Brazier, C. Jonker, J. Treur, et N. Wijngaards, Compositional Design of a Generic Design Agent, *Design Studies Journal*, Vol. 22, 2001, pages 439–471.

[BRE 98] A. Brenton et T. Wood, JACK Intelligent Agents User Guide, Agent Oriented Software Pty Ltd., Australie, 1998.

[BRI 01] J.-P. Briot et Y. Demazeau (éditeurs), *Principes et architecture des systèmes multi-agents*, Collection IC2, Hermès/Lavoisier, 2001.

[BRI 06] J.-P. Briot, T. Meurisse, et F. Peschanski, Une expérience de conception et de composition de comportements d'agents à l'aide de composants, [BOI 06, pages 11–41].

[BRO 86] R.A. Brooks, A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation*, 2(1) :14–23, mars 1986.

[BRU 04] E. Bruneton, T. Coupaye, M. Leclerc, V. Quéma, et J.-B. Stefani, An Open Component Model and its Support in Java, Actes du *7th International Symposium on Component-Based Software Engineering*, No 3054, LNCS, Springer-Verlag, mai 2004, pages 7–22.

[CHA 02] J.-M. Chauvet, Services Web avec SOAP, WSDL, UDDI, et XML, Eyrolles, 2002.

[CHO 06] C. Chopinaud, A. El Fallah-Seghrouchni, et P. Taillibert, Prevention of Harmful Behaviors within Cognitive and Autonomous Agents, *European Conference on Artificial Intelligence (ECAI'06)*, Riva del Garda, Italie, août-septembre 2006.

[DRO 98] A. Drogoul et A. Collinot, Applying an Agent-Oriented Methodology to the Design of Artificial Organisations : a Case Study in Robotic Soccer, *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, (1)1 :113–129, Springer-Verlag, 1998.

[DEC 96] K. Decker et K. Sycara, Designing Reusable Behaviors for Information Agents, *Technical Report*, The Robotics Institute, CMU, Pittsburgh, PA, États-Unis, janvier 1996.

[DEM 95] Y. Demazeau, From Interactions to Collective Behaviour in Agent-Based Systems, Actes de *1st European Conference on Cognitive Science*, Saint-Malo, 1995, pages 117–132.

[DUB 06] J. Dubus et P. Merle, Vers l'auto-adaptabilité des architectures logicielles dans les environnements ouverts distribués, Actes de la *1ère Conférence francophone sur les Architectures Logicielles (CAL'2006)*, Nantes, édités par M. Oussalah et F. Oquendo, Hermès/Lavoisier, septembre 2006.

[EUG 03] P. Eugster, P. Felber, R. Guerraoui, et A.-M. Kermarrec, The Many Faces of Publish/Subscribe, *ACM Computing Surveys*, 35(2) :114–131, juin 2003.

[FER 95] J. Ferber, *Les Systèmes Multi-Agent - Vers une Intelligence Collective*, InterEditions, 1995.

[FER 98] J. Ferber et O. Gutknecht, A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems, Actes de la *3rd International Conference on Multi-Agent Systems (ICMAS'98)*, Paris, juillet 1998, IEEE, pages 128–135.

[FER 00] J. Ferber et O. Gutknecht, Admission of Agents in Groups as a Normative and Organizational Problem, Actes du *Workshop on Norms and Institutions in Multi-Agent Systems*, ACM Press, 2000.

[FIN 97] T. Finin, Y. Labrou, et J. Mayfield, KQML as an Agent Communication Language, *Software Agents*, édité par J. Bradshaw, MIT-Press, 1997, pages 291–316.

[FIP 02] "`http ://www.fipa.org/repository/aclspecs.html`", 2002.

[GAM 95] E. Gamma, R. Helm, R. Johnson, et J. Vlissides, *Design Patterns - Elements of Reusable Object Oriented Software*, Professional Computing Series, Addison-Wesley, 1995.

[GAR 04] A. Garcia, U. Kulesza, et C. Lucena, Aspectizing Multi-Agent Systems : From Architecture to Implementation, *Software Engineering for Multi-Agent Systems III*, No 3390, LNCS, Springer-Verlag, décembre 2004, pages 121–143.

[GAS 92] L. Gasser et J.-P. Briot. Object-based Concurrent Programming and Distributed Artificial Intelligence, *Distributed Artificial Intelligence : Theory and Praxis*, édité par N.M. Avouris et Gasser, pages 81–107, Kluwer, 1992.

[GAS 98] L. Gasser, Agents and Concurrent Objects, Interview par J.-P. Briot, Special series on Actors and Agents, éditée par D. Kafura and J.-P. Briot, *IEEE Concurrency*, 6(4) :74–81, octobre-décembre 1998.

[GEL 92] D. Gelernter et D. Carrierro, Coordination Languages and their Significance, *Communications of the ACM*, 35(2), 1992.

[GEO 99] M. Georgeff, B. Pell, M. Pollack, M. Tambe, et M. Wooldridge, The Belief-Desire-Intention Model of Agency, Actes du *5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL'98)*, No 1555, LNCS, Springer-Verlag, 1999, pages 1–10.

[GHE 02] C. Ghezzi et G.P. Picco, An Outlook on Software Engineering for Modern Distributed Systems, Actes du *Monterey Workshop on Radical Approaches to Software Engineering*, Venise, Italie, octobre 2002.

[GOR 04] H.J. Goradia et J.M. Vidal, Building Blocks for Agent Design," *Agent-Oriented Software Engineering IV*, No 2935, LNCS, Springer Verlag, 2003, pages 153–166.

[GRO 06] G. Grondin, N. Bouraqadi, et L. Vercouter, Assemblage automatique de composants pour la construction d'agents avec MaDcAr, Actes de *2ème Journée Multi-Agents et Composants (JMAC'06)*, LMO'06, Nîmes, mars 2006.

[GUE 99] Z. Guessoum et J.-P. Briot, From Active Objects to Autonomous Agents. Special Series on Actors and Agents, édité par D. Kafura et J.-P. Briot, *IEEE Concurrency*, 7(3) :68–76, juillet–septembre 1999.

[HOR 98] B.C. Horling, A Reusable Component Architecture for Agent Construction, *Technical Report* No 1998-49, Computer Science Dept., UMASS, MA, États-Unis, octobre 1998.

[HUB 02] J.F. Hübner, J.S. Sichman, et O. Boissier, Using the Moise+ Model for a Cooperative Framework of MAS Reorganisation, Actes du *17th Brazilian Symposium on Artificial Intelligence (SBIA'04)*, No 3171, LNAI, Springer-Verlag, pages 506–515.

[JEN 00] N.R. Jennings, On Agent-based Software Engineering, *Artificial Intelligence*, 117 :277–296, 2000.

[KAF 98] D. Kafura et J.-P. Briot. Introduction to Actors and Agents, Special Series, *IEEE Concurrency*, 6(2) :24–29, avril–juin 1998.

[KUW 95] K. Kuwabara, T. Ishida, et N. Osato, AgenTalk : Coordination Protocol Description for Multiagent Systems, Actes de *1st International Conference on MultiAgent Systems (ICMAS'95)*, AAAI Press, juin 1995 (poster).

[LHU 98] M. Lhuillier, 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.

[MAT 01] P. Mathieu, J.-C. Routier, et Y. Secq, Dynamic Skills Learning : a Support to Agent Evolution, Actes du *AISB'01 Symposium on Adaptive Agents and Multi-agent Systems*, The Society for the Study of Artificial Intelligence and the Simulation of Behaviour (SSAISB), York, Royaume-Uni, mars 2001, pages 25–32.

[MEL 04] F. Melo, R. Choren, R. Cerqueira, C. Lucena, et M. Blois, Deploying Agents with the CORBA Component Model, Actes de la *2nd International Conference on Component Deployment (CD'2004)*, No 3083, LNCS, Springer-Verlag, mai 2004, pages 234–247.

[MEU 04] T. Meurisse, Simulation multi-agent : du modèle à l'opérationnalisation, *Thèse de doctorat*, Université Paris 6, juillet 2004.

[MEY 92] B. Meyer, Applying 'Design by Contract', *IEEE Computer*, octobre 1992, pages 40–52.

[MIL 82] R. Milner, *A Calculus for Communicating Systems*, Springer-Verlag, 1982.

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

[MUL 97] J.P. Müller, Control Architectures for Autonomous and Interacting Agents : A Survey, chapitre de *Intelligent Agent Systems : Theoretical and Practical Issues*, No 1209, LNAI, Springer-Verlag, 1997, pages 1–26.

[NOL 06] S. Nolfi, G. Baldassarre, R. Calabretta, J. Hallam, D. Marocco, O. Miglino, J.-A. Meyer, et D. Parisi, *From Animals to Animats 9 : Proceedings of the 9th International*

Conference on Simulation of Adaptive Behaviour, No 4095, LNAI, Springer-Verlag, 2006.

[NOR 01] L. Northrop, Reuse That Pays, Keynote, 23rd International Conference on Software Engineering (ICSE'2001), Toronto, Canada, mai 2001.

[ODE 02] J. Odell, Objects and Agents Compared, *Journal of Object Technology (JOT)*, 1(1), mai 2002.

[OMG 06a] Object Management Group (OMG), Common Object Request Broker Architecture (CORBA), "`http ://www.omg.org/corba/`", 2006.

[OMG 06b] Object Management Group (OMG), Corba Component Model (CCM), "`http ://www.omg.org/technology/documents/formal/components.htm`", 2006.

[OMG 06c] Object Management Group (OMG), Model Driven Architecture (MDA), "`http ://www.omg.org/mda/`", 2006.

[OUS 05] M. Oussalah (éditeur), *Ingénierie des composants - Concepts, techniques et outils*, Vuibert, 2005.

[PAY 08] T.R. Payne, Web Services from an Agent Perspective, *IEEE Intelligent Systems*, 23(2) :12–14, mars avril 2008.

[PER 98] J.-F. Perrot, Objets, classes et héritage : Définitions, chapitre de *Langages et modèles à objets - État des recherches et perspectives*, édité par R. Ducournau, J. Euzenat, G. Masini, et A. Napoli, Collection Didactique, INRIA, 1998, pages 3–31.

[PER 04] J.-F. Perrot et J.-P. Briot, Introduction, Numéro spécial : Des octets aux modèles - Vingt ans après, où en sont les objets ? Revue *L'Objet*, 10(4) :11–16, décembre 2004.

[PES 00] F. Peschanski, T. Meurisse, et J.-P. Briot, Les composants logiciels : évolution technologique ou nouveau paradigme ?, Actes de la *Conférence Objets, Composants, Modèles (OCM'2000)*, Nantes, France, mai 2000, pages 53–65.

[PES 05] F. Peschanski et J.-P. Briot, Architectures de composants répartis, [OUS 05, pages 247–279].

[QUE 03] V. Quéma, R. Balter, Luc Bellissard, D. Féliot, A. Freyssinet, et S. Lacourte, Déploiement asynchrone et hiérarchique d'applications réparties à composants, Actes de *RENPAR'15/CFSE'3/SympAAA'2003*, La Colle sur Loup, octobre 2003.

[RIC 01] P.-G. Ricordel et Y. Demazeau, La plate-forme VOLCANO : modularité et réutilisabilité pour les systèmes multi-agents, *Technique et Science Informatiques (TSI)*, Hermès/Lavoisier, 20(4), avril 2001.

[RIV 04] M. Riveill (éditeur), Systèmes à composants adaptables et extensibles, Numéro spécial, *Technique et Science Informatiques (TSI)*, Hermès/Lavoisier, 23(2), 2004.

[ROY 04] J.-C. Royer (éditeur), *Actes de la Journée Multi-Agents et Composants (JMAC'04)*, École des Mines de Paris, novembre 2004. "`http ://csl.ensm-douai.fr/MAAC/3`".

[SHA 96] M. Shaw et D. Garlan, *Software Architectures - Perspective on an Emerging Discipline*, Prentice Hall, 1996.

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

1998.

[SHO 93] Y. Shoham, Agent Oriented Programming, *Artificial Intelligence*, 60(1) :51–92, 1993.

[SIL 05] V. Silva, R. Choren, et C. Lucena, Using UML 2.0 Activity Diagram to Model Agent Plans and Actions, Actes de *International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2005)*, Utrecht, Pays-Bas, juillet 2005.

[SMI 80] R.G. Smith, The Contract Net Protocol : High-Level Communication and Control in a Distributed Problem Solver, *IEEE Transactions on Computers*, C29(12) :1104–1113, 1980.

[SUN 06] Sun Microsystems Inc., JavaBeans Specification, "`http ://java.sun.com/products/javabeans/`", 2006.

[SYC 02] K. Sycara, S. Widoff, M. Klusch, et J. Lu, LARKS : Matchmaking among Heterogeneous Agents in Cyberspace, *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 5(2) :173–203, 2002.

[SYC 03] K. Sycara, M. Paolucci, M. van Velsen, et J.A. Giampapa, The RETSINA MAS Infrastructure, *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 7(1&2), juillet 2003.

[VER 04] L. Vercouter, MAST : Un modèle de composants pour la conception de SMA, article de [ROY 04].

[VAN 00] S. van Splunter, N. Wijngaards, F. Brazier, et D. Richards, Automated Component-Based Configuration : Promises and Fallacies, Actes du *Adaptive Agents and Multi-Agent Systems Workshop*, AISB'2004 Symposium, Leeds, Royaume-Uni, 2004, pages 130–145.

[WEY 05] D. Weyns, H. Van Dyke Parunak, F. Michel, T. Holvoet, et J. Ferber, Environments for Multiagent Systems - State-of-the-Art and Research Challenges, *Environments for Multi-Agent Systems (E4MAS'2004)*, édité par D. Weyns et al., LNAI, No 3374, Springer-Verlag, 2005, pages 1–47.

[WOO 95] M. Wooldridge et N.R. Jennings, Agent Theories, Architectures, and Languages : a Survey, Actes de *Workshop on Agent Theories, Architectures, and Languages on Intelligent Agents (ATAL'94/ECAI'94)*, Springer-Verlag, 1995, pages 1–39.

[YOO 98] M.J. Yoo, W. Merlat, et J.-P. Briot, Modeling and Validation of Mobile Agents on the Web, *Simulation Series*, The Society for Computer Simulation (SCS), 30(1) :23–28, janvier 1998.