Composants et agents : évolution de la programmation et analyse comparative

Jean-Pierre Briot [1,2]

1. Sorbonne Universités, UPMC Univ Paris 06, UMR 7606
Laboratoire d'Informatique de Paris 6 (LIP6), F-75005, Paris, France

2. CNRS, UMR 7606, Laboratoire d'Informatique de Paris 6 (LIP6)
F-75005, Paris, France

Jean-Pierre.Briot@lip6.fr

RÉSUMÉ. *L'objectif de cet article est de situer et comparer les concepts de composant logiciel et d'agent (et de système multi-agent), en les replaçant dans une perspective générale d'évolution de la programmation (des procédures aux objets, acteurs, composants, services, agents…). Un référentiel commun à trois axes/dimensions est proposé : sélection de l'action à exécuter au niveau d'une entité, flexibilité du couplage entre entités, niveau d'abstraction. On peut en effet observer une recherche continue de plus grande flexibilité (par des notions telles que la liaison tardive, ou la réification des connexions) et de plus grand niveau d'abstraction (des données aux connaissances). Les concepts de composant et d'agent ont ainsi en partie des objectifs communs (flexibilité du logiciel), les systèmes multi-agents repoussant encore plus loin le niveau d'abstraction et la flexibilité du couplage entre entités, notamment à l'aide de capacités d'auto-organisation et l'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. Nous évoquerons ainsi dans la deuxième partie de l'article quelques pistes de fertilisation croisée entre composants et systèmes multi-agents.*

ABSTRACT. *The objective of this article is to compare concepts of software component and of agent (and multi-agent system), placing them within a general perspective of the evolution of programming (from procedures to objects, actors, components, services, agents…). Some common referential with three axes/dimensions is proposed: action selection at the level of one entity, flexibility of coupling between entities, level of abstraction. We indeed may observe a continuous search for a higher flexibility (through notions such as late binding, or reification of connections) and higher level of abstraction. Concepts of components and agents have some common objectives (software flexibility), multi-agent systems pushing further abstraction level and coupling flexibility between entities, notably through the notion of auto-organization and the use of knowledge. Meanwhile, we believe that the concepts and technology of software components may help at the construction of multi-agent systems. We will present in the second part*

of the article some prospects for cross-fertilization between software components and multi-agent systems.

MOTS-CLÉS : *analyse, comparaison, historique, programmation, composant logiciel, agent, système multi-agent, objet, acteur, service, abstraction, évolution, invocation, liaison, couplage.*

KEYWORDS: *analysis, comparison, history, programming, software component, agent, multi-agent system, object, actor, service, abstraction, evolution, invocation, binding, coupling.*

DOI:10.3166/TSI.33.85-115 © 2014 Lavoisier

1. Introduction

Les *composants logiciels* (Bachman *et al.*, 2000) et les *systèmes multi-agents* (souvent abrégés en *SMA*) (Briot, Demazeau, 2001 ; Ferber, 1995) sont des approches de conception et de développement de logiciel ayant un impact certain. 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.

Dans un premier temps, et la partie principale de l'article, nous proposons une analyse comparative entre les composants logiciels [1] 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 (en nous inspirant de (Gasser, Briot, 1998)).

Dans un deuxième temps, nous évoquerons 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 évoquerons 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 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 (Odell, 2002), les objets concurrents (Gasser, Briot, 1992 ; 1998), les acteurs (Kafura, Briot, 1998)... Cet article reprend et intègre certaines de ces analyses mais les complète par des aspects propres aux composants, ce qui, à notre connaissance, a encore peu fait l'objet d'études comparatives. Une initiative à signaler cependant, sur le thème des croisements entre composants et systèmes multi-agents,

1. Par la suite, nous utiliserons le plus souvent le seul terme *composants*.

a été l'organisation en France de deux éditions successives de Journées multi-agents et composants (JMAC), en 2004 puis en 2006. Un numéro spécial de la revue *L'Objet* a d'ailleurs été édité sur ce thème (Boissier, 2006), à la suite de la première journée. Signalons également ici pour information deux analyses comparatives de différents modèles de composants (Crnković *et al.*, 2011 ; Lau, Wang, 2007) et deux analyses comparatives de diverses plates-formes et langages multi-agents, fondés sur des modèles à objets, logiques ou à base de composants (Bordini *et al.*, 2005 ; 2006).

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 (Gasser, Briot, 1998). Nous avons choisi un référentiel commun à trois dimensions qui nous paraissent des enjeux importants de la programmation et du logiciel :

– sélection de l'action à exécuter. Elle indique quand et comment sera décidée par l'entité logicielle (ou physique, dans le cas d'un robot, par exemple tourner) 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 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 entités elles-mêmes, pour offrir une vision architecturale indépendante de l'implantation [2]. 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 – qui pourront être manipulés indépendamment d'une plateforme d'im-

2. Nous avons préféré utiliser les termes implantation et implanter, plutôt que les anglicismes implémentation et implémenter.

plantation, jusqu'aux connaissances – sur lesquelles des mécanismes de traitement automatisé (raisonnement) pourront être appliqués.

Il faut souligner que ces trois 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 clairement un lien 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 les deux fondés sur un mécanisme unique, celui de liaison (*binding*, voir par exemple (Ghezzi, Picco, 2002)) : liaison de l'appel au code effectif dans le cas de la sélection de l'action, et liaison d'une entité à une autre entité dans le cas du couplage. 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* architecte de système) de même que les abstractions et mécanismes par exemple : architecture d'agents *versus* connecteur).

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 (à exécuter) 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, exprimée 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 des *structures abstraites de données*, tout en restant cependant indépendantes et externes aux procédures.

Les langages de programmation par objets, avec les pionniers Simula 67 puis Smalltalk, apportent alors une innovation majeure, avec la réunion de procédures et des données associées dans une capsule autonome appelée *objet*. Les données deviennent ainsi internes et privées à l'objet et ses procédures (appelées *méthodes*) et l'*envoi de message* est l'unique moyen d'invoquer un objet, en activant une de ses procédures. Une avancée déterminante est l'apparition avec Smalltalk de la liaison tardive, c'est-à-dire que la procédure à invoquer sera déterminée en fonction de la classe [3] 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

3. Une *classe* est la définition d'une famille d'objets similaires. C'est la classe qui définit les méthodes (procédures) et les variables (modèle de données) communes aux objets qui seront ses instances.

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++ [4].

Du point de vue de la sélection de l'action, les composants logiciels n'innovent pas véritablement car, comme nous le verrons section 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) [5], un composant est *auto-contenu*, avec tout son code, mais également sa documentation (Oussalah, 2005). 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 quant 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* (Odell, 2002). 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 (Georgeff *et al.*, 1999). 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 (Müller, Pischel, 1993)).

Enfin des mécanismes sub-symboliques (sans représentation explicite du monde) de régulation, souvent inspirés de la biologie (métabolisme, émotions, adaptation… voir par exemple (Ziemke *et al.*, 2012)) peuvent également intervenir.

4. 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.

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

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 (Gasser, Briot, 1998). 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 [6], 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 binding*).

Le tableau 1, inspiré de (Odell, 2002), résume cette comparaison et la figure 1 la replace dans notre référentiel.

Tableau 1. Structure des entités et sélection de l'action

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 externe et dynamique *(ex : dirigée par les buts)*

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 :

– *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 mode de couplage temporel.

6. 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.

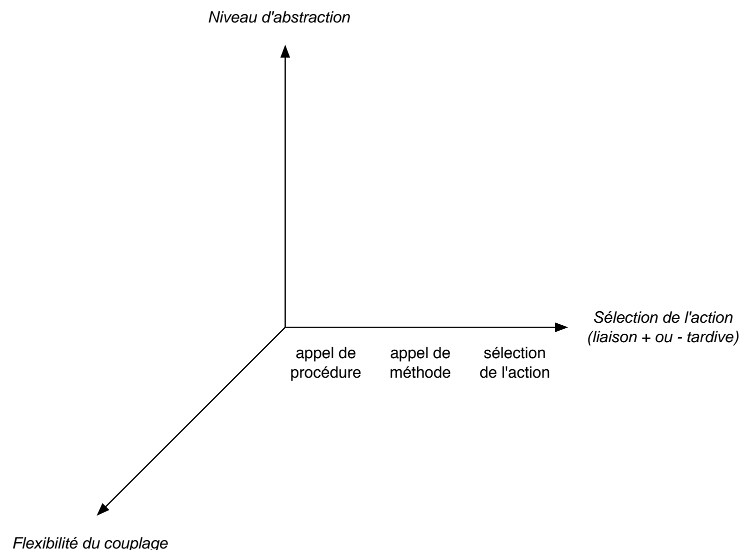

Figure 1. Evolution de la sélection de l'action

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 (*identifiant*). 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 pouvoir ainsi envoyer des requêtes. En pratique la représentation interne (implantation) 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 2). 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 réimplanter), 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` ».

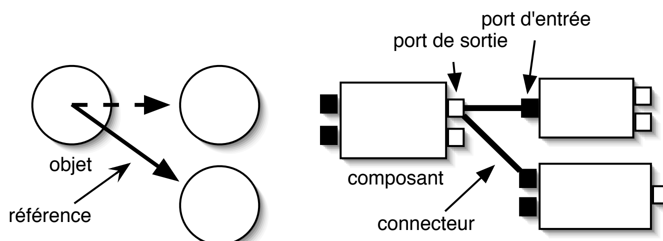

Figure 2. Couplage entre objets versus couplage entre composants

La notion de composant logiciel [7] 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 [8], 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) (Oussalah, 2005 ; Lau, Wang, 2007).

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*, 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 2.

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 [9] 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.

7. Pour une analyse plus complète sur les caractéristiques des composants logiciels et une comparaison entre différents modèles de composants, nous renvoyons à (Crnković *et al*., 2011) et (Lau, Wang, 2007).

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

9. 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 (Peschanski *et al*., 2000), car il offre l'encapsulation et la hiérarchisation qui s'avèrent fort utiles pour aider à maîtriser la complexité. Mais, il est en pratique seulement supporté par une minorité de modèles de composants (par exemple par Fractal (Bruneton *et al*., 2004) et MALEVA, (Briot *et al*., 2006), mais pas par JavaBeans (Sun, 2006) ni par CORBA Component Model (CCM) (OMG, 2006)). Voir l'analyse des techniques de liaison (binding) associées (appelées liaison horizontale pour la composition fonctionnelle et liaison verticale pour la composition structurelle) dans (Crnković *et al*., 2011).

Les composants apportent une vision *architecturale* (*architecture logicielle* (Shaw, Garlan, 1996)) 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)* ») (Shaw, Garlan, 1996) 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*... (Shaw, Garlan, 1996)) 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 (Allen, Garlan, 1994).

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 proposées. (Beugnard *et al.*, 1999) 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 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 (Meyer, 1992).

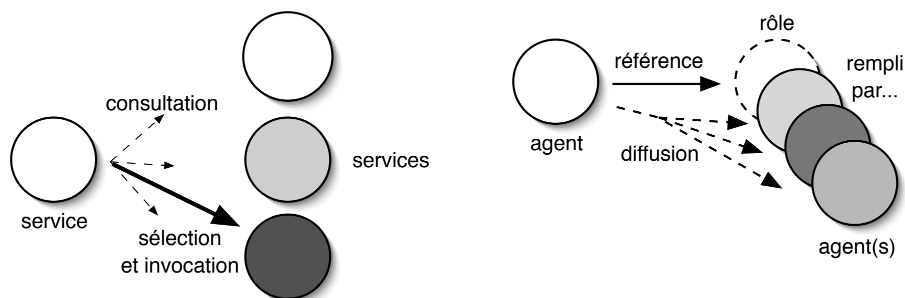

Figure 3. Couplage entre services et couplage entre agents

Le concept de *service* des *architectures à base de services* (*service oriented architectures (SOA*), dont en particulier les services web (Chauvet, 2002), apporte une dynamicité *via* un mécanisme de découverte puis de sélection dynamiques de services (illustré dans la partie gauche de la figure 3). 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* (Chauvet, 2002)) 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...). 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 (Ferber, Gutknecht, 1998) et MOISE+ (Hübner *et al.*, 2007).

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 [10] vers l'ensemble des agents remplissant (au moment de l'interaction) ce rôle (voir la partie droite de la figure 3). 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 [11]. 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 (Finin *et al.*, 1997)) 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* (Smith, 1980), qui sera abordé section 4.2.3 et illustré figure 5). Les relations entre les agents sont donc très dynamiques et gérées en partie par eux-mêmes ou *via* les organisations.

---

10. Ce mécanisme de désignation implicite du receveur sera analysé section 4.2.1.

11. 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 en général pas compositionnels, à la différence des composants.

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 (Hübner *et al.*, 2007). 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 (Drogoul, Collinot, 1998).

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 (Dubus, Merle, 2006). Cependant l'approche connaissances et sociale, caractéristique des systèmes multi-agents, reste (pour le moment du moins, voir plus loin section 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 flexibilité et donc de délégation de l'initiative, et besoins d'assurer certaines garanties sur le fonctionnement du système.

4.2. Couplage de communication

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

– la manière de désigner le ou les receveur(s), par exemple : point à point, multipoint, 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…

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 de manière naturelle *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* (Shaw, Garlan, 1996). 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 [12] 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 (Gelernter, Carrierro, 1992)), 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 section 4.1, 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 (Kafura, Briot, 1998) (par exemple en KQML (Finin *et al.*, 1997), voir les langages de communication d'agents section 5). Un autre est l'*indexation* des communications selon un *rôle*. Ainsi par exemple, dans le modèle organisationnel AGR (agent groupe rôle) (Ferber, Gutknecht, 1998), 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 2).

Enfin, dans certains types de systèmes multi-agents, dans lesquels l'environnement (physique ou non) [13] 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 visant à promouvoir l'environnement d'un système multi-agent comme une abstraction privilégiée (Weyns *et al.*, 2005).

4.2.2. Transfert de données

Le mode de transfert des données de la programmation par objets est *bidirectionnel*, avec un *retour de valeur* [14]. Il est hérité de l'appel procédural ou fonctionnel. Il correspond (comme nous le verrons section 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.

---

12. Le critère d'abonnement et le mode de diffusion peuvent varier, voir la classification de (Eugster *et al.*, 2003).

13. Des algorithmes à base de fourmis et leurs phéromones peuvent être utilisés comme méthode générale d'optimisation (voir par exemple (Albert *et al.*, 2009)), l'environnement n'ayant alors plus de lien avec une réalité physique.

14. 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.

Le modèle des acteurs (Agha, 1986) introduit un appel *unidirectionnel* (et *asynchrone*, voir section 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. Des langages basés sur les acteurs, tel par exemple ABCL (Actor-Based Concurrent Language) (Yonezawa *et al.*, 1986), peuvent offrir au programmeur le choix entre un appel unidirectionnel et asynchrone ou un appel bidirectionnel et synchrone.

Les modèles de composants, tels par exemple le *CORBA component model* (CCM) (OMG, 2006)[15], proposent eux aussi souvent 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 figure 4.

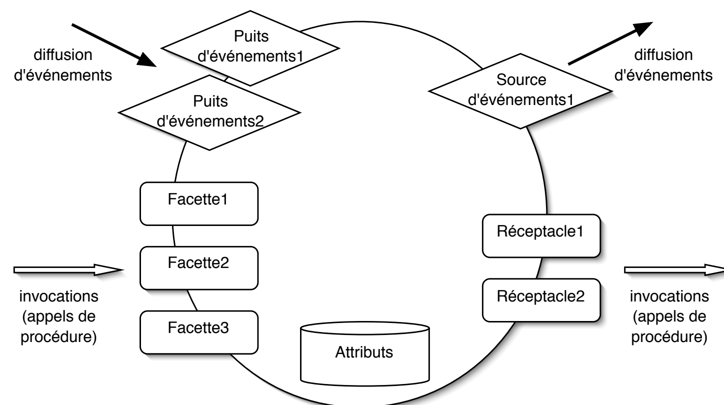

Figure 4. Le modèle de composant CCM

Le style architectural des espaces partagés (voir section 4.2.1) introduit un transfert de données, cette fois indirect, *via* une structure intermédiaire et la distinction entre production et consommation.

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 (Chauvet, 2002). 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 (terme qui était à l'origine l'acronyme de *simple object access protocol*) inclut à la fois un mode bidirectionnel et un mode unidirectionnel.

Les agents reprennent en général le mode unidirectionnel (et asynchrone) des acteurs, comme nous le verrons section 5, par le biais de langages de communication

15. Notez qu'un exemple de modèle (également un standard industriel) de composant plus récent, intégré dans un modèlle d'architecture à base de services, est SCA (OASIS, 2013a). Nous avons cependant choisi ici d'illustrer notre analyse à travers CCM, pour sa valeur historique et pédagogique.

d'agents très élaborés. Ils peuvent de plus spécifier de manière très fine les informations communiquées. Enfin, la communication éventuelle *via* un environnement (par ajout, enlèvement, ou consommation de données) représente un mode indirect de transfert de données.

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 (Agha, 1986) 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.

Comme indiqué section 4.2.2, des langages basés sur les acteurs, tel ABCL, peuvent offrir à la fois un appel unidirectionnel et asynchrone et un appel bidirectionnel et synchrone (et même d'autres modes, tel avec promesse de réponse anticipée – souvent appelé *future* –, que nous ne développerons pas ici, voir (Yonezawa *et al.*, 1986)). Scala est lui un exemple de langage de programmation qui intègre programmation par objets, fonctionnelle et programmation par acteurs (Odersky *et al.*, 2010). Enfin, pour une analyse plus spécifique de l'intégration du modèle de programmation par objets avec les dimensions de programmation concurrente et répartie, nous renvoyons à (Briot *et al.*, 1998).

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 (FIPA, 2002), 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. Il existe diverses familles de protocoles multi-agents : d'interaction (communication d'information, requête. . .), de coordination (tel l'appel d'offre simple ou itéré – voir ci-dessous), de négociation, d'enchères (à l'anglaise ou à la hollandaise, avec un prix proposé augmentant ou décroissant). Un exemple classique de protocole

multi-agent est le protocole d'*appel d'offre* (également appelé *contract net protocol*), spécifié par la FIPA (FIPA, 2002). La figure 5 représente le diagramme d'interactions correspondant. On y voit la phase de l'appel initial (`cfp` pour "call for proposals"), les propositions éventuelles (ou refus) faites par les participants – tout cela gouverné par une date limite pour répondre, l'acceptation ou le refus d'une proposition d'un participant, et enfin le participant qui aura été choisi informant de la finalisation (ou échec) de l'exécution de sa proposition.

Les services web proposent eux aussi des mécanismes de coordination, appelée plutôt *chorégraphie*. Le langage *Web Services Choreography Description Language* (WS-CDL) avait été initialement défini dans ce but par le standard W3C et a été ensuite remplacé par les standards BPEL (*Business Process Execution Language*) et BPNM (*Business Process Model and Notation*) (OASIS, 2013b). [16]

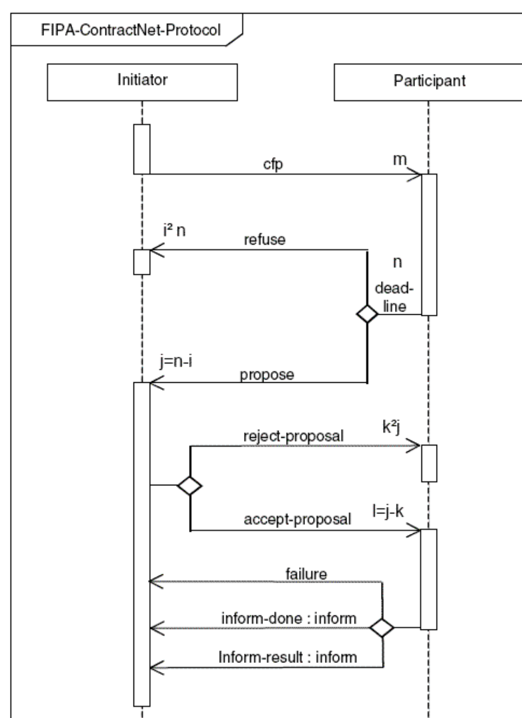

Figure 5. Le contract net protocol

Le tableau 2 résume l'évolution du couplage selon les 2 facettes principales (structure et communication) et les 3 sous-facettes de cette dernière (désignation du (ou des)

16. Nous ne détaillerons pas ici les caractéristiques des services et des services Web, qui font l'objet de standards et de nombreuses spécifications techniques (voir par exemple les ouvrages (Chauvet, 2002) et (Papazoglou, 2012), ainsi que (Payne, 2008) pour une perspective agent des services Web) car cela serait l'objet d'un autre article, celui-ci s'intéressant en priorité aux rapports entre composants et agents.

receveur(s), mode de transfert des données, couplage temporel – synchronisation). La figure 6 la replace dans notre référentiel.

Tableau 2. Nature du couplage

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	Bi-directionnel *(retour valeur)* direct	Uni-directionnel direct	Bi- ou uni-directionnel *(événements)* direct	Bi- ou uni-directionnel	Uni-directionnel direct ou indirect *(environnement)*
Synchronisation	Synchrone	Asynchrone	Synchrone ou asynchrone	Synchrone ou asynchrone	Asynchrone ou protocole

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 données, sémaphore, processus, objet, message, composant, modèle. . .). 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 (Perrot, 1998) [17]. On est donc 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) (Shoham, 1993), avec une représentation symbolique de concepts cognitifs, tels : *croyance*, *but*, *désir*, *intention*. . . Ces connaissances internes peuvent être communiquées (communication

17. Un bilan sur leurs réussites, échecs et perspectives, est proposé dans (Perrot, Briot, 2004).

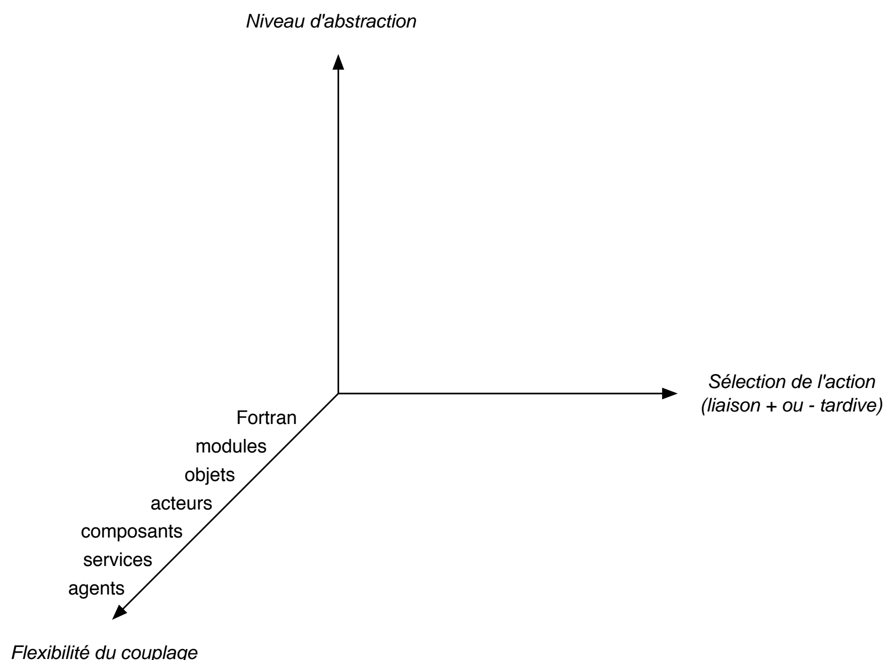

Figure 6. Evolution de la la flexibilité du couplage

de croyances, de plans, d'intentions. . .) pour permettre par exemple aux agents d'apprendre ou de se coordonner entre eux.

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 (et surtout), 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. . .).

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, 1997) standardise, par l'intermédiaire d'un IDL : *interface description language*, les types de données échangées. L'analogue pour les agents spécifie de manière encore plus fine et précise les échanges d'information. Le simple IDL de CORBA est remplacé [18] par un véritable *langage de communication d'agents* (*agent*

18. Il nous faut préciser que CORBA et ACL ne jouent en fait pas exactement les mêmes rôles (Splunter *et al.*, 2004). CORBA, à travers l'IDL, offre un standard de description des interfaces (signatures) des objets

communication language : ACL). Le premier historiquement est KQML (Finin *et al.*, 1997), suivi par l'ACL de la FIPA (FIPA, 2002). 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...`) ;

– *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 (FIPA, 2002)) ;

– *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 figure 5).

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 (Drogoul, Collinot, 1998)) 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 [19].

Enfin, dans l'évolution du niveau d'abstraction de la programmation, résumée par la figure 7, nous souhaitons également signaler le courant 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, 2010)). Ce courant est distinct et non spécifique aux systèmes multi-

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.

19. 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 (voir par exemple les travaux autour du modèle d'organisation MOISE+ (Hübner *et al.*, 2007)).

agents, mais il existe des efforts croissants visant à coupler conception multi-agent et ingénierie des modèles (voir par exemple (Silva *et al.*, 2005)).

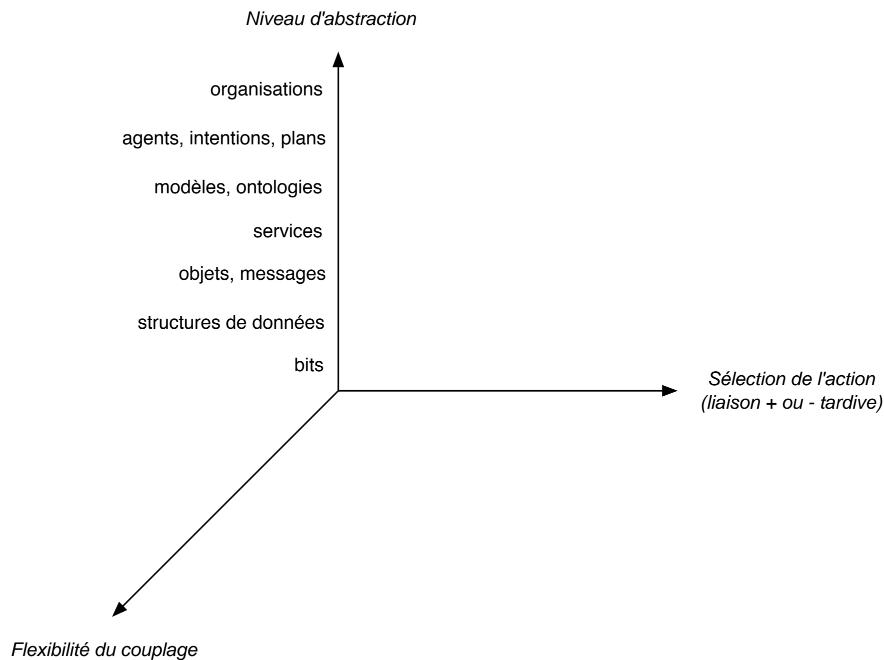

Figure 7. Evolution du niveau d'abstraction

6. Apports mutuels

A la suite de cette analyse comparative entre le 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.

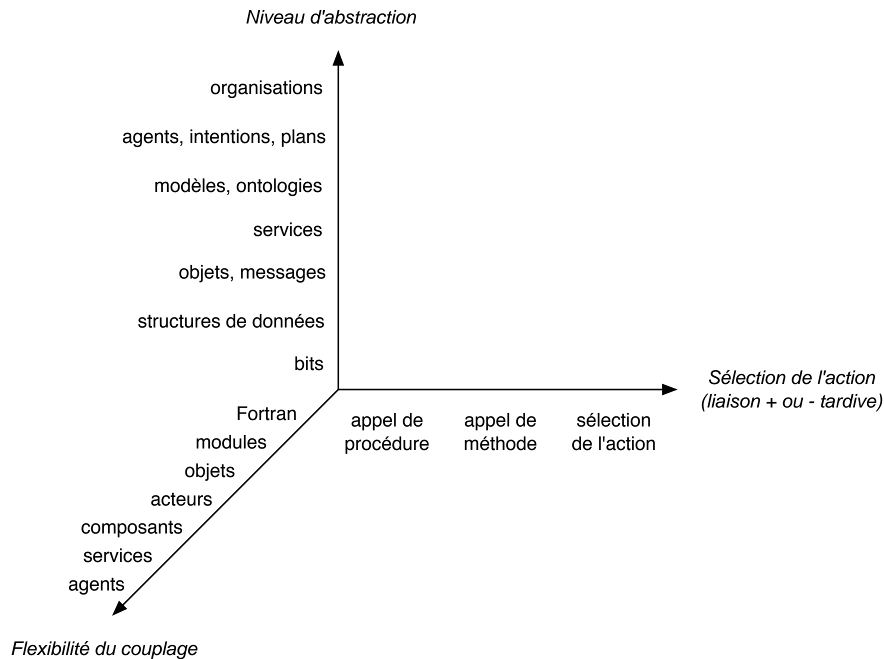

Figure 8. Evolution de la programmation

6.1. Vers des composants plus autonomes et adaptatifs

Nous avons vu section 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.

6.1.1. Assistance à l'assemblage

Le choix des composants logiciels pour constituer une architecture constitue une question pratique. Le concepteur doit identifier les composants souhaités, souvent à partir de bibliothèques, internes ou externes (« *sur étagères* »), et les assembler en fonction des objectifs souhaités. Une problématique 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.

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 » (Northrop, 2001)). 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 (voir section 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. Ainsi, chaque approche possible 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.

La communauté du génie logiciel et des composants logiciels a proposé diverses méthodes d'aide à la sélection de composants, telles qu'OTSO (Kontio *et al.*, 1995) ou CODER (Cortellessa *et al.*, 2006). Elles sont en général pragmatiques et fondées sur la minimisation du coût et la prise en compte de la robustesse et du temps de livraison. L'utilisateur peut être amené à comparer le coût de l'utilisation d'un composant préexistant au coût de développement d'un composant spécifique.

La communauté des agents a de manière relativement indépendante développé des méthodes analogues, avec une approche moins orientée coût mais parfois plus orientée représentation de connaissances, en utilisant par exemple des ontologies de concepts. Un exemple est le modèle de mise en correspondance (*matchmaking*) d'agents LARKS (Sycara *et al.*, 2002). Ce modèle est assez élaboré et est fondé sur différents niveaux de descriptions (signatures-types, comportementaux via des contraintes, ontologiques via des 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 pourrait aussi être transposée 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 (W3C, 2007)). Il sera intéressant de voir de quelle manière ces différentes approches peuvent donner naissance à des méthodes de sélection et assistance à l'assemblage de composants logiciels élaborées et pratiques à la fois, des prototypes tels que CoBaSA les préfigurant (Manolios *et al.*, 2007) [20].

6.1.2. Auto-reconfiguration de l'architecture

La conception et la construction d'une architecture logicielle restent encore en général réalisées statiquement lors de 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* (Riveill, 2004) (au niveau de l'architecture, (Dubus, Merle, 2006) comme au niveau d'un composant (Malenfant *et al.*, 2001)). Ceci amène une évolution actuelle de mécanismes d'adaptation, au départ essentiellement réactifs à partir d'évènements (Dubus, Merle, 2006), vers des méca-

20. Nous ne détaillerons pas plus ici les techniques de sélection de composants, car cela serait l'objet d'un autre article.

nismes 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 (Grondin *et al.*, 2006). Le niveau de contrôle (*meta*) de l'architecture MaDcAr est en charge du processus d'adaptation (déclenchement, décisions et exécution). Il inclut les modules suivants : un gestionnaire de contexte de l'agent, qui déclenche les adaptations et fournit les données contextuelles nécessaires aux décisions d'adaptation ; un gestionnaire d'assemblage qui dirige les adaptations ; un gestionnaire du contenu, qui gère l'ajout ou la suppression de composants dans le niveau de base. Le gestionnaire d'assemblage va déterminer à l'aide d'un solveur de contraintes une nouvelle configuration, où une configuration se compose d'un graphe de rôles qui seront remplis par des composants et d'un ensemble de fonctions de caractérisation définissant la pertinence de cette configuration par rapport au contexte (ceci incluant l'adéquation aux ressources matérielles disponibles). Pendant la phase d'adaptation, les communications entre composants et avec l'extérieur sont temporairement stockées jusqu'à la terminaison de l'assemblage. Il faut noter que le modèle MaDcAr 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.

Ces travaux peuvent être mis en regard de travaux sur l'assemblage automatique de composants, capables d'un certain degré d'évolution dynamique (remplacement, ajout ou suppression d'un composant lors de l'exécution, mais non substitution complète par un autre assemblage). (Pelliccione *et al.*, 2008) propose une approche d'assemblage automatique de composants combinant deux approches : une analyse et une vérification des propriétés au niveau de l'architecture logicielle, et une génération du code d'assemblage. En cas de changement dynamique d'un composant le contrôle de l'adaptation suspend l'activité de ce composant pendant son remplacement et effectue un transfert d'état du composant vers le nouveau composant.

Notons que les besoins de coordonner des adaptations possiblement à différents endroits et niveaux d'une architecture tout en maintenant la cohérence, 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.

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 (Peschanski, Briot, 2005). 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 évoqué section 6.4).

6.2.1. Déploiement d'agents

Tout d'abord les composants logiciels offrent une base de structuration de l'implantation des agents [21], notamment en vue de leur déploiement et de leur documentation. Un exemple de proposition est (Melo *et al.*, 2004) qui propose une implantation des agents sous la forme de composants CCM (OMG, 2006). 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 (Sycara *et al.*, 2003) [22]. [23]

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 (Bellissard *et al.*, 1996), 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 (Quéma *et al.*, 2003). Ceci témoigne des influences croisées croissantes entre composants et agents.

6.3. 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 (Ferber, Gutknecht, 2000) 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. (Ferber, Gutknecht, 1998) esquisse également plusieurs pistes

---

21. 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 (Guessoum, Briot, 1999), ou de composants (voir section 6.4).

22. *A contrario*, le modèle de mise en correspondance d'agents LARKS (introduit section 6.1.1 (Sycara *et al.*, 2002)), conçu pour RETSINA (Sycara *et al.*, 2003), 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.

23. Une alternative à l'utilisation de CORBA est l'utilisation de services web comme support de déploiement, d'interoperabilité et de communication (voir par exemple (Melliti *et al.*, 2005)).

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, voir figure 9). Des vérifications classiques fondées par exemple sur des contrats syntaxiques ou comportementaux (voir section 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+ (Hübner *et al.*, 2007). 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 (Chopinaud *et al.*, 2006) nous semble de ce point de vue un point de départ intéressant.

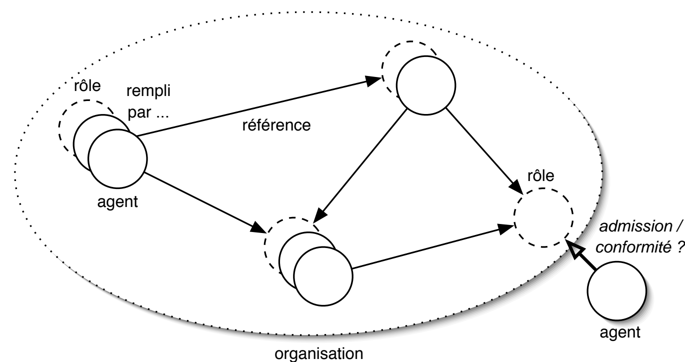

Figure 9. Conformité d'un agent à un rôle

6.4. 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.

Un exemple est l'architecture ABLE (Bigus *et al.*, 2002), proposée dans le cadre du programme *Autonomic Computing* d'IBM. Elle offre un ensemble de composants outils techniques (statistiques, inférence floue, recherche. . .), implantés à base de Ja-

vaBeans, qui peuvent être composés en une chaîne de traitements pour un objectif donné, par exemple un équilibrage de charges automatique pour un serveur (voir figure 10, provenant de (Bigus *et al.*, 2002)).

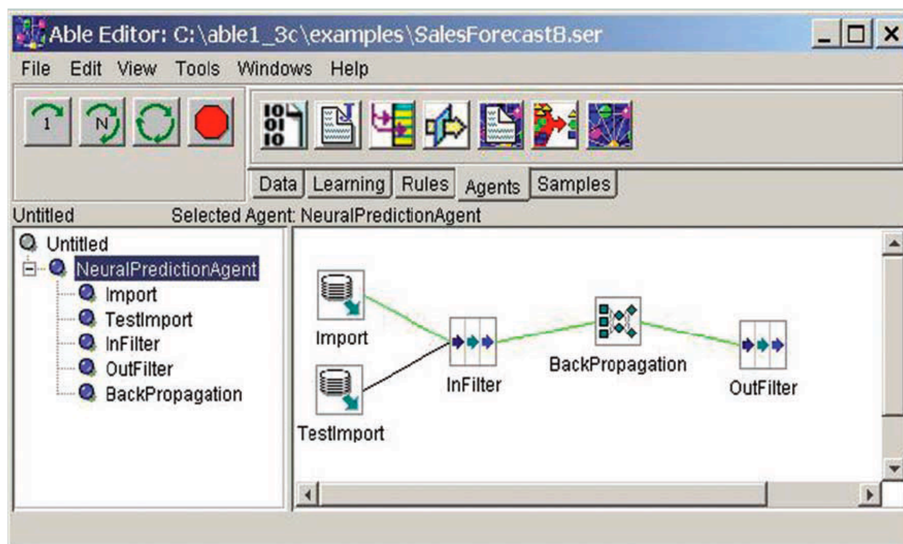

Figure 10. Architecture ABLE

D'autres architectures d'agents à base de composants existent. Dans une autre étude, nous en avons recensé un certain nombre et proposé une classification de différents critères de décomposition d'une architecture d'agent en composants (selon : cycle d'activation, points de vue, modèles, comportements. . .). Cette étude a été présentée dans un article paru dans un numéro spécial de TSI sur composants et systèmes multi-agents (Briot *et al.*, 2006). Pour éviter des redites et du fait des limitations de place de cet article, nous proposons au lecteur intéressé de s'y référer. Soulignons que cet article introduit de plus un modèle de composant d'agents spécifique, nommé MALEVA. Une de ses originalités est qu'il applique les principes des composants et de composition logicielle, non seulement à la conception des comportements d'un agent, mais également à la spécification du contrôle, ainsi réifié *via* des notions de bornes, connexions, et même des composants de contrôle.

7. Conclusion

Nous assistons 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 autoreconfiguration). 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. Cet article a parmi ses objectifs 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 la présence plus récente d'un *troisième larron* dans le jeu déjà existant entre composants et agents, il s'agit des architectures à base de services et d'une incarnation naturelle sous forme de *services web* [24]. 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. Ils offrent des spécifications de la coordination entre services (appelée chorégraphie) mais qui n'atteint pas encore le degré de sophistication des systèmes multi-agents dans le domaine (voir à ce sujet une analyse comparative des services web et des agents (Payne, 2008)). Un enjeu important sera donc de pouvoir intégrer et réutiliser au mieux les savoirs respectifs de ces différentes communautés (voir, par exemple, à ce sujet (Splunter *et al.*, 2004)).

Remerciements

Les prémisses de cette étude remontent à un entretien que nous avions conduit avec Les Gasser sur les rapports entre les objets concurrents et les agents (Gasser, Briot, 1998), publié dans une série spéciale sur les acteurs et agents (Kafura, Briot, 1998). Nous le remercions pour sa contribution pionnière et fondamentale à cette réflexion. Nous remercions également Jean-François Perrot pour ses discussions et enseignements toujours lumineux sur les concepts fondamentaux et la pratique de la programmation (voir par exemple (Perrot, 1998) et (Perrot, Briot, 2004)). Nous leur dédions cet article. Nous tenons également à remercier Dominique Longin pour son aide efficace et bénévole pour maintenir le style LaTeX de la revue.

Bibliographie

Agha G. (1986). *Actors: a model of concurrent computation in distributed systems*. MIT Press.

Albert P., Armetta F., Hassas S. (2009). Agents situés : une nouvelle voie pour le développement d'applications industrielles. In A. E. Fallah-Seghrouchni, J.-P. Briot (Eds.), *Technologies des systèmes multi-agents et applications industrielles*, p. 101–146. Hermès/Lavoisier.

Allen R., Garlan D. (1994, mars). *Formal connectors*. Research Report n° CMU-CS-94-115. Pittsburgh, PA, États-Unis, Department of Computer Science, Carnegie Mellon University.

24. 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 également 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, 2010)). Cet article se focalise en effet sur les rapports entre les composants logiciels et les systèmes multi-agents, ce qui donne un article déjà suffisamment étendu.

Bachman F., Bass L., Buhman C., Comella-Dorda S., Long F., Robert J. *et al.* (2000, mai). *Volume II: Technical concepts of component-based software engineering*. Technical Report nº CMU/SEI-2000-TR-008 & ESC-TR-2000-007. Pittsburgh, PA, États-Unis, Software Engineering Institute, Carnegie Mellon.

Bellissard L., Atallah S. B., Kerbrat A., Riveill M. (1996). Component-based programming and application management with Olan. In J.-P. Briot, J.-M. Geib, A. Yonezawa (Eds.), *Object-based parallel and distributed computation*, p. 290–309. Springer.

Beugnard A., Jézéquel J.-M., Plouzeau N., Watkins D. (1999, juin). Making components contract aware. *IEEE Computer*, vol. 32, nº 7, p. 38–45.

Bigus J., Schlosnagle D., Pilgrim J., Mills W., Diao Y. (2002). Able: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, vol. 41, nº 3, p. 350–371.

Boissier O. (2006). Composants et systèmes multi-agents. *L'Objet*, vol. 12, nº 4.

Bordini R., Dastani M., Dix J., Seghrouchni A. E. F. (2005). *Multi-agent programming: Languages, platforms and applications*. Springer.

Bordini R., Dastani M., Dix J., Seghrouchni A. E. F., Gomez-Sanz J., Leite J. *et al.* (2006). A survey of programming languages and platforms for multi-agent systems. *Informatica*, vol. 30, nº 1, p. 33–44.

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

Briot J.-P., Guerraoui R., Löhr K.-P. (1998, septembre). Concurrency and distribution in object-oriented programming. *Computing Surveys*, vol. 30, nº 3, p. 291–329.

Briot J.-P., Meurisse T., Peschanski F. (2006). Une expérience de conception et de composition de comportements d'agents à l'aide de composants. *L'Objet*, vol. 12, nº 4, p. 11–41. (Numéro spécial Composants et systèmes multi-agents)

Bruneton E., Coupaye T., Leclerc M., Quéma V., Stefani J.-B. (2004, mai). An open component model and its support in Java. In *7th international symposium on component-based software engineering*, p. 7–22. Springer.

Chauvet J.-M. (2002). *Services Web avec SOAP, WSDL, UDDI, et XML*. Eyrolles.

Chopinaud C., Fallah-Seghrouchni A. E., Taillibert P. (2006, août–septembre). Prevention of harmful behaviors within cognitive and autonomous agents. In *European conference on artificial intelligence (ECAI'06)*, p. 205–209. Riva del Garda, Italie, IOS Press.

Cortellessa V., Marinelli F., Potena P. (2006). Automated selection of software components based on cost/reliability tradeoff. In *Software architecture – third european workshop, EWSA 2006, Nantes, France, september 4-5, 2006, revised selected papers*, p. 66–81. Springer Verlag.

Crnković I., Sentilles S., Vulgarakis A., Chaudron M. R. V. (2011, septembre). A classification framework for software component models. *IEEE Transactions on Software Engineering*, vol. 37, nº 5, p. 593–615.

Drogoul A., Collinot A. (1998). 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)*, vol. 1, nº 1, p. 113–129.

Dubus J., Merle P. (2006, septembre). Vers l'auto-adaptabilité des architectures logicielles dans les environnements ouverts distribués. In M. Oussalah, F. Oquendo (Eds.), *1ère conférence francophone sur les architectures logicielles (CAL'2006).* Nantes, Hermès/Lavoisier.

Eugster P., Felber P., Guerraoui R., Kermarrec A.-M. (2003, juin). The many faces of publish/subscribe. *ACM Computing Surveys*, vol. 35, n° 2, p. 114–131.

Ferber J. (1995). *Les systèmes multi-agent – vers une intelligence collective*. InterEditions.

Ferber J., Gutknecht O. (1998, juillet). A meta-model for the analysis and design of organizations in multi-agent systems. In *3rd international conference on multi-agent systems (ICMAS'98)*, p. 128–135. Paris.

Ferber J., Gutknecht O. (2000, juin). Admission of agents in groups as a normative and organizational problem. In *4th international conference on autonomous agents (Agents'2000) workshop on norms and institutions in multi-agent systems.* Barcelona, Espagne, ACM Press.

Finin T., Labrou Y., Mayfield J. (1997). KQML as an agent communication language. In J. Bradshaw (Ed.), *Software agents*, p. 291–316. MIT-Press.

FIPA. (2002). *http://www.fipa.org/repository/aclspecs.html*. Rapport technique. Auteur.

Gasser L., Briot J.-P. (1992). Object-based concurrent programming and distributed artificial intelligence. In N. Avouris, L. Gasser (Eds.), *Distributed artificial intelligence: Theory and praxis*, p. 81–107. Kluwer.

Gasser L., Briot J.-P. (1998, octobre–décembre). Agents and concurrent objects. *IEEE Concurrency*, vol. 6, n° 4, p. 74–81. (Interview of Les Gasser by J.-P. Briot)

Gelernter D., Carrierro D. (1992). Coordination languages and their significance. *Communications of the ACM*, vol. 35, n° 2.

Georgeff M., Pell B., Pollack M., Tambe M., Wooldridge M. (1999). The belief-desire-intention model of agency. In *5th international workshop on intelligent agents V: Agent theories, architectures, and languages (ATAL'98)*, p. 1–10. Springer.

Ghezzi C., Picco G. (2002, octobre). An outlook on software engineering for modern distributed systems. In *Monterey workshop on radical approaches to software engineering.* Venezia, Italie.

Grondin G., Bouraqadi N., Vercouter L. (2006). MaDcAr: an abstract model for dynamic and automatic (re-)assembling of component-based applications. In *9th int. sigsoft symposium on component-based software engineering (CBSE'2006)*, p. 360–367. springer.

Guessoum Z., Briot J.-P. (1999, juillet–septembre). From active objects to autonomous agents. *IEEE Concurrency*, vol. 7, n° 3, p. 68–76.

Hübner J. F., Sichman J. S., Boissier O. (2007). Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering (IJAOSE)*, vol. 1, n° 3–4, p. 370–395.

Kafura D., Briot J.-P. (1998, avril–juin). Introduction to actors and agents. *IEEE Concurrency*, vol. 6, n° 2, p. 24–29.

Kontio J., Chen S., Limperos K., Tesoriero R., Caldiera G., Deutsch M. S. (1995). A COTS selection method and experiences of its use. In *20th annual software engineering workshop.* NASA, Greenbelt, MA, États-Unis.

Lau K.-K., Wang Z. (2007, octobre). Software component models. *IEEE Transactions on Software Engineering*, vol. 33, n° 10, p. 709–724.

Malenfant J., Segarra M.-T., André F. (2001). Reflection for dynamic adaptability: Lessons learned from the molène experiment. In *Metalevel architectures and separation of crosscutting concerns (Reflection'2001)*, p. 110–117. Springer.

Manolios P., Subramanian G., Vroon D. (2007, juillet). Automating component-based system assembly. In *Sigsoft international symposium on software testing and analysis (ISSTA'07)*, p. 61–71. London, Royaume-Uni.

Melliti T., Haddad S., Suna A., El Fallah Seghrouchni A. (2005). Web-MASI: Multi-agent systems interoperability using a web services based approach. In *IEEE/WIC/ACM international conference on intelligent agent technology*, p. 739-742. IEEE Computer Society.

Melo F., Choren R., Cerqueira R., Lucena C., Blois M. (2004, mai). Deploying agents with the corba component model. In *2nd international conference on component deployment (CD'2004)*, p. 234–247. Springer.

Meyer B. (1992, octobre). Applying design by contract. *IEEE Computer*, vol. 25, n° 10, p. 40–51.

Müller J. P., Pischel M. (1993). *The agent architecture InteRRaP: Concept and application*. Technical Report n° RR-93-26. Saarbrücken, Allemagne, DFKI.

Northrop L. (2001, mai). Reuse that pays. In *23rd international conference on software engineering (ICSE'2001).* Toronto, ON, Canada. (Keynote)

OASIS. (2013a). *Open composite services architecture (CSA)*. Rapport technique. http://www.oasis-opencsa.org, OASIS (Organization for the Advancement of Structured Information Standards).

OASIS. (2013b). *Web services business process execution language (BPEL)*. Rapport technique. http://bpel.xml.org, OASIS (Organization for the Advancement of Structured Information Standards).

Odell J. (2002, mai). Objects and agents compared. *Journal of Object Technology (JOT)*, vol. 1, n° 1.

Odersky M., Spoon L., Venners B. (2010). *Programming in Scala*. Artima.

OMG. (1997). *Common object request broker architecture (CORBA)*. Rapport technique. http://www.omg.org/corba/, Object Management Group (OMG).

OMG. (2006). *Corba component model (CCM)*. Rapport technique. http://www.omg.org/technology/documents/formal/components.htm, Object Management Group (OMG).

OMG. (2010). *Model driven architecture (MDA)*. Rapport technique. http://www.omg.org/mda/, Object Management Group (OMG).

Oussalah M. (2005). *Ingénierie des composants – concepts, techniques et outils*. Vuibert.

Papazoglou M. (2012). *Web services & SOA, principles and technology, second edition*. Pearson.

Payne T. (2008, mars–avril). Web services from an agent perspective. *IEEE Intelligent Systems*, vol. 23, n° 2, p. 12–14.

Pelliccione P., Tivoli M., Bucchiarone A., Polini A. (2008). An architectural approach to the correct and automatic assembly of evolving component-based systems. *The Journal of Systems and Software*, n° 81, p. 2237–2251.

Perrot J.-F. (1998). Objets, classes et héritage : Définitions. In R. Ducournau, J. Euzenat, G. Masini, A. Napoli (Eds.), *Langages et modèles à objets – état des recherches et perspectives*, p. 3–31. INRIA.

Perrot J.-F., Briot J.-P. (2004, décembre). Introduction. *L'Objet*, vol. 10, n° 4, p. 11–16. (Numéro spécial : Des octets aux modèles – Vingt ans après, où en sont les objets ?)

Peschanski F., Briot J.-P. (2005). Architectures de composants répartis. In M. Oussalah (Ed.), *Ingénierie des composants – concepts, techniques et outils*, p. 247–279. Vuibert.

Peschanski F., Meurisse T., Briot J.-P. (2000, mai). Les composants logiciels : évolution technologique ou nouveau paradigme ? In *Conférence objets, composants, modèles (OCM'2000)*, p. 53–65. Nantes.

Quéma V., Balter R., Bellissard L., Féliot D., Freyssinet A., Lacourte S. (2003, octobre). Déploiement asynchrone et hiérarchique d'applications réparties à composants. In *RENPAR'15/CFSE'3/SympAAA'2003*. La Colle sur Loup.

Riveill M. (2004). Systèmes à composants adaptables et extensibles. *Technique et Science Informatiques (TSI)*, vol. 23, n° 2. (Numéro spécial)

Shaw M., Garlan D. (1996). *Software architectures – perspective on an emerging discipline*. Prentice Hall.

Shoham Y. (1993). Agent oriented programming. *Artificial Intelligence*, vol. 60, n° 1, p. 51–92.

Silva V., Choren R., Lucena C. (2005, juillet). Using UML 2.0 activity diagram to model agent plans and actions. In *International conference on autonomous agents and multi-agent systems (AAMAS'2005)*. Utrecht, Pays-Bas.

Smith R. (1980). The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, vol. 29, n° 12, p. 1104–1113.

Splunter S. van, Wijngaards N., Brazier F., Richards D. (2004). Automated component-based configuration: Promises and fallacies. In *AISB'2004 convention 4th symposium on adaptive agents and multi-agent systems (AAMAS-4)*, p. 130–145. Leeds, Royaume-Uni.

Sun. (2006). *Javabeans specification*. Rapport technique. http://java.sun.com/products/javabeans/, Sun Microsystems Inc.

Sycara K., Paolucci M., Velsen M. van, Giampapa J. (2003, juillet). The retsina mas infrastructure. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, vol. 7, n° 1&2.

Sycara K., Widoff S., Klusch M., Lu J. (2002). Larks: Matchmaking among heterogeneous agents in cyberspace. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, vol. 5, n° 2, p. 173–203.

W3C. (2007). *Web service description language (WSDL)*. Rapport technique. http://www.w3.org/standards/webofservices/description, World Wide Web Consortium (W3C).

Weyns D., Parunak H. V. D., Michel F., Holvoet T., Ferber J. (2005). Environments for multiagent systems – state-of-the-art and research challenges. In D. Weyns, H. V. D. Paru-

nak, F. Michel (Eds.), *Environments for multi-agent systems – first international workshop, E4MAS 2004, new york, ny, july 19, 2004, revised selected papers*, p. 1–47. Springer.

Yonezawa A., Briot J.-P., Shibayama E. (1986, novembre). Object-oriented concurrent programming in ABCL/1. *Sigplan Notices*, vol. 21, nº 11, p. 258–268. (Special Issue. Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86), Portland OR, États-Unis)

Ziemke T., Balkenius C., Hallam J. (Eds.). (2012). *From animals to animats 12 – 12th international conference on simulation of adaptive behavior, SAB 2012, odense, denmark, august 2012, proceedings* nº 7426. Springer.

Reçu le 2 janvier 2013
Accepté le 24 octobre 2013

Jean-Pierre Briot. *Il est Directeur de recherche au CNRS, actuellement Directeur du Bureau du CNRS au Brésil, qu'il a créé en 2010. Il est membre du Laboratoire d'Informatique de Paris 6 (LIP6), UMR UPMC-CNRS, et Professeur invité permanent à l'Université PUC-Rio au Brésil. Il a soutenu son Doctorat en 1984 et son HdR en 1989, tous deux en informatique à l'UPMC (Université Pierre et Marie Curie, Paris). Ses intérêts généraux de recherche portent sur la conception et l'auto-adaptation de logiciels décentralisés coopératifs, à la croisée des langages de programmation, du génie logiciel, de l'intelligence artificielle et des systèmes répartis. Il a également participé à différents projets interdisciplinaires, tels qu'en informatique musicale ou en appui informatique à la gestion participative de l'environnement.*