Composition opportuniste et ascendante à base d'agents coopératifs

G. Denis, J.-P. Arcangeli, V. Noël, C. Triboulot, S. Trouilhet
IRIT - Université de Toulouse, UPS, 118, route de Narbonne, 31062 Toulouse Cedex, France
{Prenom.Nom}@irit.fr

Résumé : *Les systèmes ambiants sont composés d'entités physiques ou humaines, mobiles pour certaines d'entre elles, en interaction. Du fait de la dynamique (principalement la mobilité) et de l'ouverture, des systèmes peuvent se former de manière opportuniste de par la présence simultanée en un même espace d'appareils communicants. De nouvelles applications peuvent alors émerger simplement parce que les éléments qui les composent sont en situation d'interaction, c'est-à-dire susceptibles d'être composées ou de se composer. Nous présentons ici un premier travail qui vise la construction d'applications ambiantes émergentes au moyen de systèmes multi-agents coopératifs capables de s'auto-organiser. La coopération entre agents composants et agents instances supporte la composition ascendante et autonome, la configuration, le déploiement et le maintien en production (adaptation dynamique).*

Mots-clés : *intelligence ambiante, composant, composition ascendante, système multi-agent adaptatif, adaptation dynamique.*

1 Introduction

Les systèmes ambiants sont des systèmes complexes composés d'entités physiques (mobiles pour certaines d'entre elles) et d'humains en interaction ; ils sont répartis, ouverts, décentralisés, hétérogènes, dynamiques… L'intelligence ambiante représente la capacité de l'environnement ambiant à fournir des services pertinents et adaptés aux humains qui y sont plongés. Comme l'avait imaginé M. Weiser il y a une vingtaine d'années [1], nous sommes aujourd'hui face à un environnement informatique nouveau qui demande de nouvelles techniques pour la construction, le déploiement et le maintien en production des applications.

Dans ce contexte, du fait de la mobilité et de l'ouverture principalement, on peut imaginer que des systèmes puissent se former de manière opportuniste de par la présence simultanée en un même espace d'appareils communicants. De nouvelles applications (systèmes logiciels) sont en mesure d'émerger simplement parce que les entités qui les composent sont en situation d'interaction, c'est-à-dire d'être composées ou de se composer. Par exemple, une application multimédia comme la lecture d'un film peut résulter de la composition entre une télécommande, un media center, un écran et des haut-parleurs. On peut aussi penser à un utilisateur mobile équipé d'un *smartphone* dont l'un des composants est défaillant qui pourrait trouver, sur un *smartphone* à proximité, un composant disponible et exploitable à distance en remplacement de son composant défaillant. La seule présence de ces dispositifs matériels, à un instant et dans un environnement donnés, peut permettre au système de se former sans nécessiter d'intervention humaine. On peut ainsi imaginer qu'une application se compose de façon autonome, à l'initiative des éléments eux-mêmes, sans que la composition n'ait été exprimée par le concepteur ou l'utilisateur, sans que les composants n'aient été conçus pour composer ce système.

L'objectif de notre travail est d'explorer la problématique de la composition autonome et d'en évaluer la faisabilité. Ce travail a été initié en 2011 [2] ; nous présentons ici quelques premiers éléments.

1.1 Composition, composants et architecture

La composition est au cœur du développement logiciel. C'est le moyen le plus élémentaire pour construire des programmes : en programmation impérative, le programmeur compose des instructions au moyen de structures de contrôles ; en programmation fonctionnelle, il s'appuie sur la composition de fonctions ; en programmation logique, il compose des prédicats au moyen de règles logiques. Le concept d'objet permet de développer des applications en composant des classes par héritage ou association. Les *design patterns* [3] apportent, à des problèmes de conception

récurrents, des solutions de composition réutilisables qui limitent le couplage entre les classes et améliorent la flexibilité du logiciel.

Le concept de « composant logiciel » [4] est une évolution du concept d'objet dans laquelle l'interface requise (l'ensemble des services requis -dépendances- sur d'autres composants) est exposée au même niveau que l'interface fournie. La figure 1 représente un composant « *Video Player* » qui offre une interface de commande et requiert un service d'affichage et un service de diffusion du son. Alors que la composition entre objets est directe (elle se matérialise par des références d'un objet vers un autre), la composition entre composants est exprimée de manière externe sous forme d'une configuration. Elle est réalisée par le développeur dans une activité d'assemblage. Habituellement, cette activité s'effectue hors ligne (statiquement) en phase de conception ou dans le cadre d'opérations de maintenance. Il existe différents « modèles de composants » qui définissent la nature des composants et de leur composition. Certains modèles autorisent la reconfiguration dynamique, c'est-à-dire le changement de configuration sans arrêter l'exécution. Par ailleurs, certains modèles de composant introduisent le concept de « conteneur » : un conteneur est une infrastructure dans laquelle un ou plusieurs composants peuvent être déployés, et qui « gère » les composants à l'exécution et leur fournit les mécanismes et les ressources nécessaires.

FIG. 1 - *Représentation du composant « Video Player » en UML2*

Lorsque l'assemblage concerne des briques logicielles d'une certaine granularité, on entre dans le domaine de l'architecture logicielle. Pour L. Bass *et al.*, « l'architecture logicielle est la structure ou les structures d'un système, comprenant les éléments logiciels, les propriétés visibles de ces éléments et les relations entre eux » [5]. L'architecte logiciel conçoit une architecture dans le but de répondre aux différentes exigences techniques (fonctionnelles et non fonctionnelles) et non techniques exprimées par les parties prenantes au projet. Les langages de description d'architectures ou ADL [6] permettent de représenter des architectures logicielles au moyen de composants, de connecteurs (supports pour la liaison entre composants) et de configurations, et dans certains cas de vérifier plus ou moins formellement des propriétés sur les assemblages.

Il existe différents styles d'architecture parmi lesquels les « architectures orientées service » ou SOA. Dans ce cadre, les briques logicielles sont disponibles sous forme de services qui peuvent être localisés, puis invoqués et coordonnés à l'exécution via un courtier de service. La construction des applications (fabrication de processus de niveau « métier ») se fait ici en composant des invocations de services existants plutôt qu'en développant de nouveaux composants.

Enfin, dans les systèmes à base d'agents, les composants sont des agents, entités logicielles autonomes capables d'interaction [7]. De par la décentralisation intrinsèque de ces systèmes, ce sont les agents eux-mêmes qui sont à l'origine des interactions donc de la composition, qu'il s'agisse d'interaction directe entre agents ou indirecte à travers l'environnement (qui joue un rôle de médiateur).

1.2 La composition en contexte « ambiant »

En architecture logicielle, les facteurs de complexité sont multiples. Dans le contexte ambiant, la mobilité des dispositifs matériels ou les pannes, les actions imprévues des utilisateurs humains, les limites en termes de ressources (par exemple, en matière d'énergie) rendent les environnements d'exécution instables et les ressources volatiles. Ainsi, des éléments logiciels peuvent apparaître ou disparaître dynamiquement (ou être mis à jour), rendant possibles certains assemblages ou

inopérants des assemblages existants. L'adaptation dynamique est donc un besoin essentiel que la programmation des systèmes ambiants doit prendre en compte.

Un autre point clé est la décentralisation propre à ces systèmes qui sont composés d'éléments répartis et enfouis, administrés indépendamment et reliés par des réseaux de communication dont la qualité de service ne permet pas un contrôle centralisé du système. L'autonomie des composants est donc une propriété naturelle dans un tel contexte dynamique et décentralisé. Plus largement, on peut penser que les systèmes informatiques du futur ou leurs composants devront de plus en plus être dotés de capacités d'auto-organisation, d'auto-administration, d'auto-réparation, d'auto-adaptation, etc.

Telle que nous l'imaginons, la composition autonome (à l'initiative des composants eux-mêmes) pose des problèmes originaux par rapport aux contextes de programmation plus classiques introduits précédemment :

- Il n'y pas de client qui exprime des besoins, ni de réalisation dans l'intention de les satisfaire. Il n'y a donc pas de validation possible au sens traditionnel (montrer que le produit répond au besoin initial). Il n'y a pas non plus de spécification du produit, donc pas de vérification de conformité à ces spécifications.
- Il n'y a pas de concepteur du système à l'initiative des compositions et qui les contrôle.
- Il existe des utilisateurs humains qui font eux-mêmes partie du système et qui peuvent jouer un rôle dans l'acceptation des applications qui apparaissent dynamiquement.

La question qui nous intéresse ici est relative à la composition opportuniste dans un contexte « ambiant » dans le but de produire des applications que l'on peut qualifier « d'émergentes », à son automatisation, à son contrôle, à sa mise en œuvre, etc. En particulier, nous prenons le parti de privilégier une approche ascendante (*bottom-up*). Dans [8], le lecteur trouvera une analyse de la problématique de la conception des applications ambiantes qui conclut à la nécessité d'une approche ascendante et décentralisée. Au-delà de la composition, nous nous intéressons à l'adaptation dynamique de ces applications et à leur maintien en production. Pour supporter la composition et l'adaptation, nous proposons une approche à base de système multi-agent coopératif (nous faisons abstraction ici des problèmes d'interopérabilité des composants).

Remarquons que le problème d'automatisation et de contrôle de la composition peut se retrouver sous une autre forme dans le cadre de l'architecture logicielle et du développement de logiciels à base de composants. Dans ce cadre, le rôle de l'ingénieur-développeur-architecte est d'assembler des composants logiciels extraits d'une bibliothèque (qui peut être de grande taille) afin de satisfaire les différentes exigences des parties prenantes au projet. Pour faciliter la réutilisation, pour lever d'éventuels conflits et face à la complexité et à la combinatoire, on pourrait imaginer l'assister dans sa démarche en lui proposant des assemblages pertinents auto-construits.

1.3 Plan

Ce papier est organisé comme suit. Dans la section 2, nous développons un exemple afin d'illustrer notre problème dans un cadre concret. Dans la section 3, nous proposons un cadre pour analyser les travaux dans le domaine de la composition automatique, puis nous étudions un certain nombre d'entre eux. Dans la section 4, nous décrivons les principes de notre solution à base d'agents coopératifs ainsi que les grandes lignes du prototype que nous avons développé.

Enfin, nous concluons dans la section 5 et nous listons quelques problèmes ouverts.

2 Illustration

2.1 Cas d'étude : Plop à l'université.

Le cas d'étude que nous présentons ici a pour objectif d'illustrer notre propos et de mettre en évidence des compositions opportunistes possibles dans un environnement ambiant donné.

Plop sort du métro, arrive à l'université et découvre les lieux. Comme à son habitude quand il accoste en terre inconnue, il sort son *smartphone* et consulte les services qui lui sont proposés. Le relais de l'université y a automatiquement chargé différentes informations qui pourraient l'intéresser, en fonction des données relatives à son dossier, et lui propose notamment une carte du campus, avec certains bâtiments en surbrillance. L'écran du *smartphone* de Plop n'est malheureusement pas très pratique pour afficher une carte.

Heureusement, un panneau publicitaire propose à Plop d'afficher la carte sur son écran. Mais le panneau publicitaire n'ayant pas d'interface d'interaction, l'écran tactile du *smartphone* de Plop fait office de télécommande pour se déplacer sur la carte affichée, zoomer et sélectionner des lieux.

Après avoir accompli les démarches administratives, (facilitées par le système ambiant) Plop acquiert enfin son nouveau statut d'étudiant. Sa carte étudiant est directement intégrée à son *smartphone*, ce qui provoque l'apparition de nouveaux services et applications. En plus d'appliquer le statut d'étudiant à toutes ses cartes de réduction, le *smartphone* de Plop lui propose une nouvelle application : un agenda personnalisé avec ses horaires de cours, ses salles, ses matières et même les menus proposés chaque jour au restaurant universitaire.

Plop se décide enfin à rentrer chez lui et saute dans la première rame de métro. Mais au moment où il pénètre dans les tunnels, son *smartphone*, peu moderne, perd sa connexion au réseau et Plop se retrouve dans l'impossibilité de recevoir ou d'émettre un appel. Heureusement, la rame de métro dispose d'un relais téléphonique et la localisation du *smartphone* de Plop est transférée automatiquement sur celui-ci. C'est justement à ce moment là que Plip, la petite amie de Plop, choisit de l'appeler. L'appel est alors reçu par la rame qui le transmet au *smartphone* de Plop, qui sonne... La discussion continue et Plop doit descendre. Il sort de la rame qui s'éloigne. Toujours en sous-sol et sans accès au réseau, le *smartphone* détecte alors un autre *smartphone* (dans la poche d'un autre voyageur) bien plus puissant, qui peut capter les signaux à plusieurs mètres de profondeur. Il se compose alors avec celui de Plop afin de prendre en charge l'appel et le lui transférer avant que la rame ne soit hors de portée. La conversation de Plip et Plop continue, sans que ni l'un ni l'autre ne se soit rendu compte de rien.

2.2 Analyse

Considérons la première partie de ce cas d'étude et examinons quels composants entrent en jeu et leur composition.

Avant d'arriver sur le campus, le composant *Navig* du *smartphone* de Plop est « dormant » : pour fonctionner, il requiert un service *map* qui permet d'accéder et de contrôler une carte. Au moment où Plop sort du métro, l'environnement met à sa disposition un composant *Map* qui fournit la carte des lieux (le service *map*). *Map* se compose automatiquement avec *Navig*. Les autres composants entrent alors en action et se composent avec l'ensemble afin de faire émerger l'application permettant l'affichage et le contrôle de la carte sur le *smartphone*. Le composant *Display* permet d'afficher la carte sur l'écran et le composant *Tact* fournit une interface d'interaction grâce à l'écran tactile. La figure 2 schématise la composition obtenue.

Alors que Plop se déplace, son *smartphone* détecte le panneau publicitaire, qui lui-même possède un composant *Display*. Ce dernier ayant une meilleure qualité de service que celui actuellement utilisé, le système se recompose pour inclure ce nouveau composant *Display*, tout en conservant l'interface d'interaction sur le *smartphone*. La carte s'affiche alors sur le panneau publicitaire, et

Plop peut continuer à interagir sur celle-ci grâce à son *smartphone*. La figure 3 schématise la composition obtenue après adaptation (le trait discontinu représente la nouvelle liaison.

Les autres parties du cas d'étude suivent la même logique, et les applications proposées sont toutes issues de compositions opportunistes et ascendantes entre composants hébergés dans le *smartphone* ou dans son environnement.

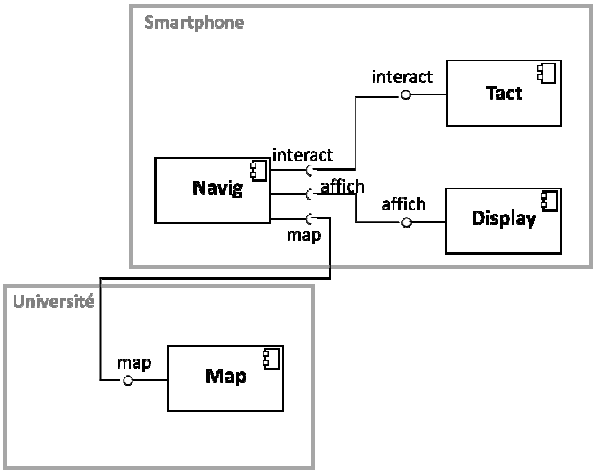

FIG. 2 - *Composition de composants*

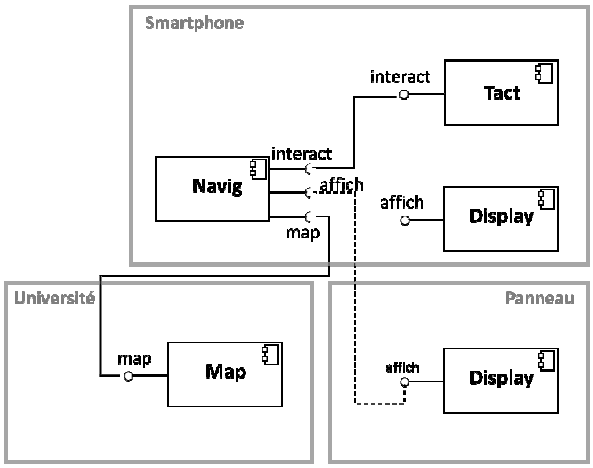

FIG. 3 - *Adaptation de la composition*

3 Travaux en composition automatique

Dans cette section, nous présentons brièvement quelques travaux sur la composition automatique dans le cadre des systèmes ambiants et des architectures à composants.

3.1 Grille d'analyse

Afin de pouvoir étudier différents travaux dans le domaine de la composition automatique et les comparer, nous proposons une grille d'analyse qui regroupe les principales caractéristiques en trois catégories concernant respectivement :

- Les composants et le système produit par composition (ce qu'on compose et ce qu'on obtient) : nature des composants (composants logiciels, services…), support de composition (interfaces, préconditions ou postconditions…), distribution ou pas des composants, nature du système produit (composant composite, *workflow*…).
- Le système de composition (par quel(s) moyen(s) on compose) : technologie supportant la composition (résolution de contraintes, planification…), technique d'assemblage (unification d'interfaces…), contrôle de la composition (centralisé, décentralisé…), contrôle de la combinatoire, ouverture (ajout ou suppression dynamique de composant).
- L'expression des besoins et des buts : qui exprime (développeur, utilisateur…), ce qui est exprimé (application, contraintes…), quand (statiquement ou dynamiquement) et comment.

Les systèmes qui suivent sont étudiés selon ce cadre. La liste n'étant pas exhaustive, le lecteur trouvera des compléments sur la composition de composants dans [9] et sur celle de services dans [10].

3.2 Composition de *workflow*

Le système décrit dans [11] vise la construction et l'exécution d'un *workflow* à partir d'un ensemble de fragments de *workflow* distribués dans un environnement mobile. Un fragment de *workflow* est un ensemble de tâches reliées entre elles par des conditions (les conditions sont

représentées par des noms). Chaque tâche possède des préconditions à sa réalisation et des postconditions résultantes de son exécution. Un fragment représente un savoir-faire, embarqué dans un dispositif mobile, et que la personne qui utilise le dispositif est capable de réaliser. L'ensemble des fragments représente le savoir-faire d'une communauté, c'est-à-dire des personnes présentes dans l'environnement, à un moment donné. Deux fragments sont composables si une précondition de l'un est identique à une postcondition de l'autre.

L'objectif est de construire le *workflow* le plus adapté pour répondre au besoin de l'utilisateur. Pour cela, dans un premier temps le système construit un supergraphe de tâches qui représente toutes les compositions possibles. La phase d'exploration consiste ensuite à parcourir et à colorer l'ensemble des conditions et des tâches en partant des conditions initiales jusqu'à arriver aux conditions finales. La dernière phase consiste à faire le chemin à l'envers dans le but de supprimer dans le graphe d'éventuelles branches inutiles. Les auteurs simplifient le problème de composition en considérant d'une part que chaque fragment de *workflow* consiste en un graphe non cyclique, et d'autre part que les conditions ont un nom unique dans un même fragment.

Les besoins sont exprimés par l'utilisateur au lancement de l'application. La technologie utilisée est basée sur une planification avec *matching* de préconditions et de postconditions et le système composé est donc de type *workflow*. Le système de composition présente l'avantage d'être distribué et ouvert. En revanche le contrôle de la composition est centralisé.

3.3 Composition d'applications ambiantes

L'architecture FCAP permet la composition dynamique d'une application en fonction des ressources et des fonctionnalités disponibles dans l'environnement [12]. Les besoins de l'utilisateur sont spécifiés par des applications abstraites (une application abstraite étant la description abstraite de fonctionnalités et des interactions requises entre ces fonctionnalités) ; le rôle du système est alors de réaliser ces applications abstraites c'est-à-dire de trouver les fonctionnalités concrètes et de les assembler. Pour cela, trois couches distinctes sont mises en place: la couche de bas niveau « infrastructures de services » pour d'accéder aux services proposés, la couche « gestion de descriptions » et enfin la couche « gestion de composition ».

La couche « gestion de descriptions » repose sur les infrastructures de services et contient un ensemble de documents décrivant sémantiquement les applications abstraites, les fonctionnalités et les configurations. La couche « gestion de la composition » s'appuie sur la couche gestion des descriptions et est implémentée sous la forme d'un système d'agents. Pour chaque application, il existe une équipe composée d'agents composeurs, d'agents superviseurs et d'agents d'interprétation. Un agent de résolution est dédié à une application abstraite particulière. Chaque fonctionnalité abstraite de l'application est prise en charge par des agents superviseurs. Leur but est de rechercher la meilleure fonctionnalité réelle correspondant à l'abstraite. Les superviseurs collectent des informations d'interprétation auprès des agents d'interprétation puis proposent la meilleure fonctionnalité à l'agent de résolution. Par résolution de contraintes, ce dernier établit la meilleure composition possible de fonctionnalités réelles pour produire l'application. Pour terminer, il met en place la configuration associée.

Dans cette approche, les besoins sont exprimés par le concepteur lors du déploiement. La technologie agent est utilisée avec un *matching* par résolution de contraintes. Les éléments à composer sont des fonctionnalités ; le système composé est donc de type application.

Le système de composition présente l'avantage d'être distribué et ouvert. Il est tout à fait possible d'introduire ou de retirer de nouvelles fonctionnalités dans l'environnement sans pour autant perturber le système Le contrôle est là encore centralisé par l'agent composeur.

3.4 Système auto-assemblé adaptatif

Dans [14] et [15], les concepteurs de l'ADL à base de composant – Darwin – proposent un modèle d'architecture à trois couches permettant la reconfiguration dynamique d'une application. L'objectif est de permettre aux composants de reconfigurer automatiquement leurs interactions pour prendre en compte les évolutions environnementales, tout en conservant les objectifs et exigences donnés lors de la phase de spécification.

Les trois couches sont hiérarchiques : la couche de bas-niveau « *Component Control* » est constituée des composants interconnectés. Si ces derniers détectent une nouvelle situation qu'ils ne peuvent prendre en charge, ils remontent leurs états à la couche intermédiaire « *Change Management* » qui est chargée d'effectuer la reconfiguration. Ce niveau donne un ensemble de plans, chacun énumérant la séquence d'actions permettant de traiter la nouvelle situation. Une action indique quel composant est requis ; cette étape peut connecter un nouveau composant, le réparer ou bien encore modifier des interconnexions. Pour cela le modèle propose un algorithme qui minimise les perturbations dans le système lors du changement et qui s'assure de l'intégrité des propriétés de sécurité du système : par exemple, un composant ne peut pas être supprimé s'il est lié à d'autres ou s'il est actif. Enfin, ce niveau dispose de plans pré-calculés ; si aucun de ses plans ne convient, le niveau « *Goal management* » est sollicité. Ce dernier est chargé de fournir les plans permettant de gérer la réorganisation. L'enjeu pour ce niveau consiste à avoir une spécification du domaine assez précise mais qui permette quand même de prendre en compte les contraintes de l'environnement.

Les éléments composés dans ce modèle sont donc des composants logiciels et le support, des interfaces. Le niveau « *Change Management* » permet de gérer la distribution. Les besoins de l'utilisateur viennent du niveau « *Goal Management* ». La recomposition s'effectue par planification ; elle ne vise que la satisfaction des besoins spécifiés initialement. De plus, le système compose un ensemble fixé *a priori*. Le point fort du système nous semble venir du choix du mécanisme de contrôle de la composition ; en effet, les auteurs, conscients de la nécessité de décentralisation pour répondre aux exigences des domaines d'application pris en charge, sont passés d'un management layer centralisé à une version distribuée, où les composants sont répartis sur plusieurs nœuds. Aucun des nœuds n'a une vision globale : un nœud connaît seulement les composants qu'il héberge. Pour valider une configuration globale, les nœuds utilisent un protocole de diffusion restreinte à plusieurs tours : le *gossip protocol*. À chaque tour un nœud envoie l'état en cours à un autre nœud choisi aléatoirement : un état est constitué de choix indiquant, pour un composant donné, quelle interface fournie satisfait quelle exigence. Le nœud complète avec ses composants et le transmet à un autre nœud au tour suivant. Le protocole s'arrête après un certain nombre de tours. Le processus de reconfiguration est donc bien coopératif et distribué.

3.5 Orchestration de services

Pour tenir compte du caractère variable et imprévisible de l'environnement ambiant, C. Vergoni *et al.* [8] proposent d'utiliser une approche ascendante pour la composition des applications. Le système proposé met en œuvre une orchestration de services à travers un assemblage de composants LCA (*Lightweight Component Architecture*) qui représentent des composants logiciels traditionnels ou des *proxies* de services logiciels. La construction est dite opportuniste car elle est provoquée par l'apparition des services dans l'environnement. Le système composé est une orchestration de services qui décrit du flux d'événements entre composants. Un conteneur supporte l'exécution de l'orchestration ainsi que sa modification dynamique à travers une interface qui permet cette modification. Les orchestrations construites sont exportables, via leurs conteneurs, sous forme de services composites SLCA (*Service Lightweight Component Architecture*).

La spécification d'une application est une description d'un assemblage de composants sous forme de règles qui peuvent être introduites et mises en œuvre dynamiquement. Ces règles expriment des objectifs limités et locaux. Elles sont exprimées sous forme d'aspects (au sens de la programmation par aspects ou AOP [13], une autre forme de composition par « tissage » dont nous n'avons pas parlée en introduction) portant sur les assemblages (« aspects d'assemblage »). L'application est réalisée « tardivement » à partir de la spécification. Le système de composition s'appuie sur le protocole UPnP pour détecter l'apparition et la disparition de services, et réagit à ces événements en déclenchant un tissage d'aspect d'assemblage. La construction des applications est ainsi dynamique et réactive aux variations de l'environnement, sans intervention extérieure. Chaque dispositif matériel peut supporter un système de composition local. Au final, les interactions entre les différents systèmes locaux auto-organisés font « émerger » un comportement d'application au niveau global. Mais, le système ne donne aucune garantie quant à la pertinence (adéquation avec l'environnement physique, logique et utilisateur) de ce comportement émergent.

De notre point de vue, la réactivité et l'auto-organisation, ainsi que l'absence d'expression d'un objectif global pour le système construit, font des systèmes multi-agents (en particulier les systèmes multi-agents adaptatifs ou AMAS, cf. section 4) de bons candidats pour supporter la mise en œuvre.

4 Système proposé

Dans leur étude sur les approches de composition dynamique de services, A. Urbieta *et al.* soulignent différentes propriétés que doivent posséder les systèmes capables de prendre en charge cette composition [16]. La difficulté ne vient pas des dispositifs qui sont largement disponibles mais la composition est rendue difficile plutôt parce que la majorité de ceux-ci sont isolés et n'ont pas de réelles capacités de coopération. Ils soulignent également que le contexte dans lequel évoluent ces dispositifs est dynamique et présente une forte distribution. Il s'agit alors de « découvrir les composants les plus adéquats dans ce contexte et de les composer au mieux ». D'où la nécessité d'avoir un système qui s'adapte au contexte environnemental, qui a un modèle d'exécution de type chorégraphie plutôt qu'orchestration et qui est composé d'éléments autonomes qui requièrent une intervention minimale des utilisateurs. « Autonomie », « distribution » et « adaptation » conduisent à l'utilisation de technologies multi-agents.

4.1 Théorie des AMAS

Les Systèmes Multi-Agents (SMA) constituent un paradigme particulièrement adapté dès lors que l'on désire définir un système décentralisé, adaptable, muni d'une faculté de perception qui lui permet de répondre dynamiquement aux contraintes de son environnement. Il s'agit de considérer un ensemble d'agents regroupés au sein d'un système dans lequel tout agent peut agir et interagir en bénéficiant de ses capacités de perception et de décision. Un système multi-agent adaptatif ou AMAS est avant tout un SMA. Il se caractérise donc par une décentralisation du contrôle au sein d'agents capables de perception et de décision [17]. La notion clef des AMAS est l'auto-organisation, celle-ci supporte l'adéquation d'une part, de la fonction globale par rapport aux besoins, et d'autre part, du système avec son environnement : la fonction du système étant issue de la composition des fonctions locales, changer l'organisation de ces fonctions locales, modifie par là même la fonction globale. En s'auto-organisant face aux pressions de son environnement et en suivant un critère de coopération, le système définit une fonction globale dite adéquate. Ce critère de coopération implique qu'un AMAS ne peut entamer d'action qui soit préjudiciable à son environnement. Maintenir un état coopératif au sein des constituants du système garantit une adéquation parfaite et continue avec son environnement. Pour un agent AMAS une attitude coopérative consiste à résoudre au mieux ces situations conflictuelles avant qu'elles n'occasionnent

des dysfonctionnements. Ces situations sont appelées « Situation Non-Coopératives » (SNC) et sont classées en fonction du moment où elles sont susceptibles d'être détectées dans le cycle perception-décision-action de l'agent [18] :

1. **Perception** – Incompréhension : l'agent n'est pas capable d'extraire l'information d'un signal. – Ambiguïté : l'agent attribue plusieurs interprétations à un même signal.

2. **Décision** – Incompétence : l'agent n'est pas capable d'exploiter pour son raisonnement l'information extraite d'un signal. – Improductivité : l'information extraite d'un signal ne permet de tirer aucune conclusion.

3. **Action** – Concurrence : l'agent estime que son action aura les mêmes conséquences que celles d'un autre agent (objectifs communs). – Conflit : l'agent estime que son action sera incompatible avec celle d'un autre agent (objectifs antagonistes). – Inutilité : l'agent estime que son action n'aura aucune conséquence sur le monde qui l'entoure.

4.2 Modèle

Notre solution repose sur l'utilisation d'un système multi-agent adaptatif pour assembler des composants logiciels dans un environnement ambiant, puis adapter l'assemblage, de manière dynamique et autonome. De façon générale, ces systèmes sont appropriés pour la conception d'applications dont la spécification n'est pas ou mal connue ; ils semblent *a priori* adaptés pour la construction et le maintien en production d'applications ambiantes.

Avant de définir les différents agents de notre système et afin de mieux appréhender notre modèle, nous proposons dans un premier temps de préciser certains termes.

Dans un environnement ambiant nous retrouvons un certain nombre de dispositifs (un *smartphone*, une *box* internet par exemple). Une ressource est une entité physique directement liée à un dispositif (l'écran d'un *smartphone*, la mémoire d'une *box* internet etc.). Elle possède un ensemble de propriétés non fonctionnelles (la définition vidéo d'un écran par exemple, la qualité audio d'un haut-parleur etc.). Les dispositifs peuvent embarquer des composants logiciels (les applications embarquées dans un *smartphone*, le logiciel de lecture vidéo d'une *box* internet etc.). Un composant fournit des services au travers de ses interfaces fournies. Un service est rendu uniquement si toutes les interfaces requises du composant sont « résolues », en d'autres termes s'il existe pour chaque interface requise du composant, un autre composant qui lui fournit ce service requis. L'instance d'un composant est à un composant ce que l'objet est à une classe en programmation objet : pour un même composant il est possible au cours du temps d'en obtenir plusieurs « exemplaires physiques ». L'instance d'un composant est la brique élémentaire pour la construction d'application. Une ressource peut être partageable (plusieurs instances de composants l'utilisent) ou non partageable (une seule instance peut l'utiliser à un instant donné). Une instance de composant peut être soumise à des contraintes pour pouvoir fonctionner et proposer ses services (l'accès à une ressource par exemple). Dans un environnement ambiant, une application est alors le résultat de la composition de plusieurs instances de composant dont les contraintes sont respectées et les dépendances sont résolues.

Notre système de composition automatique est composé d'un ensemble d'agents autonomes capables de créer et de connecter dynamiquement des instances de composants entre elles, en respectant leurs contraintes, pour produire des applications les plus adaptées aux besoins de l'utilisateur. Cependant le rôle de ce système ne s'arrête pas là : il perdure pendant l'exécution des applications, gère et reconfigure dynamiquement les assemblages qu'il maintient en réaction à certains événements. Ces derniers proviennent à la fois de l'environnement direct de l'utilisateur (arrivée de nouveaux dispositifs, changement de lieu de l'utilisateur etc.) mais également du middleware dans lequel se composent et s'exécutent la ou les applications (connexion rompue entre instances de composants par exemple).

J.-P. Briot avait déjà noté l'intérêt d'utiliser les agents comme assistants à la composition de composants [7]. C'est cette idée que nous reprenons en nous appuyant sur un système multi-agent adaptatif. Ce dernier est constitué de deux types d'agents : les agents composants et les agents instances de composants. Quel que soit son type, un agent interagit directement avec un sous ensemble d'agents qui sont à sa portée de perception. Dans le monde AMAS, cette portée est définie sous le terme de voisinage. Dans un environnement ambiant on imagine très facilement que la portée de perception d'un agent et donc son voisinage peut correspondre à la portée du réseau dans lequel se trouve l'agent.

4.2.1 Agents composants et agents instances

Un agent composant gère un composant. Il connaît donc les interfaces requises et fournies de celui-ci. Il communique et interagit localement avec un voisinage d'agents composants, c'est-à-dire un ensemble d'agents dont le composant fournit (resp. requiert) l'une des interfaces qu'il requiert (resp. fournit). Son rôle principal est de créer des agents instances lorsqu'il le juge utile. De façon générale, il interagit avec son voisinage d'agents composants et d'agents instances. Lorsqu'un dispositif apparaît dans un environnement donné, l'ensemble des agents composants contenus dans celui-ci est automatiquement créé. Lorsqu'un utilisateur éteint un dispositif ou que ce dernier est défectueux, l'ensemble des agents composants et des agents instances liés à ce dispositif est supprimé. Si pour un réseau donné un dispositif devient hors de portée de celui-ci, l'ensemble des agents (composants et instances) liés au dispositif ne fait plus partie du voisinage des agents interagissant sur ce réseau.

Un agent instance gère une instance de composant. Il est lié à un agent composant. Il possède un voisinage d'agents instances. Il interagit avec des agents composants et des agents instances. Son objectif est de trouver les « meilleurs » agents instances fournissant les services qu'il requiert au travers de ses interfaces requises. En réponse à une demande d'un autre agent, un agent instance renvoie un niveau de compétence (*capability*). Celui-ci est représenté par un pourcentage fonction de son aptitude à répondre à cette demande. La notion de compétence d'un agent instance est indirectement reliée aux ressources utilisées par l'instance de composant. Pour l'illustrer, prenons l'exemple suivant : supposons qu'il existe dans l'environnement un composant « ComposantSon » qui fournisse une interface « Son » et que celui-ci soit associé à un dispositif « Téléphone ». La ressource « Enceintes Téléphone » liée à ce dispositif définit une propriété non fonctionnelle « QualitéAudio » égale à « Mono ». Lorsqu'un agent demande à l'agent instance en charge de gérer une instance « InstanceSon » s'il peut lui fournir du son dont la « QualitéAudio » est égale à « 5.1 », la réponse à cette demande pourrait être par exemple « 50% ». L'agent à l'origine de la demande en conclurait que sa demande peut aboutir sans complètement répondre à son besoin et que par conséquent, il existe potentiellement dans son voisinage des agents susceptibles de mieux répondre à son besoin.

4.2.2 Interaction

Dans les systèmes multi-agents adaptatifs, les agents s'auto-organisent grâce à leur comportement coopératif obtenu par leurs interactions ou en réaction à un événement. Ce dernier peut provenir directement du système (création d'un nouvel agent composant, création d'un nouvel agent instance, fin d'une demande de connexion, fin d'une demande de reconnexion) ou bien du middleware (connexion physique rompue entre deux instances de composant). Pour que les agents interagissent et coopèrent nous avons défini un ensemble de messages qu'ils sont capables de s'envoyer et de recevoir. Nous les détaillons ci-dessous.

AcceptProvided (itf, acc, cap) – L'agent *acc* accepte de fournir l'interface *itf*. Il fournit son niveau de compétence cap en rapport à la requête qui lui avait été formulée.

Connect (itf, req) – L'agent *req* annonce qu'il veut connecter son interface requise *itf*. Un timeout est lancé à la suite de cette demande de telle sorte que l'agent *req* puisse prendre une décision si personne ne lui a répondu à la fin de ce timeout.

ConnectOk (itf, acc) – L'agent *acc* valide la demande de connexion à son interface fournit *itf*.

PropagateCapability (itf, sender, cap) – L'agent *sender* prévient que son niveau de compétence *cap* a évolué pour son interface fournit *itf*.

ReplaceConnect (itf, req) – L'agent *req* demande un remplacement de connexion pour l'interface *itf*. Ce message est utilisé lorsque l'agent a trouvé une meilleure connexion que celle qu'il utilisait. Un timeout est lancé à la suite de cette demande de telle sorte que l'agent *req* puisse prendre une décision si personne ne lui a répondu à la fin de ce timeout.

ReplaceConnectOk (itf, acc) – L'agent *acc* accepte le remplacement de connexion pour l'interface *itf*.

RequestProvided (itf, req, params) – L'agent *req* cherche une interface fournie *itf* avec un certain nombre de paramètres.

RequestRequired (itf, req, params) – L'agent *req* recherche une interface requise *itf* avec un certain nombre de paramètres.

4.2.3 Comportements

La conception d'un système multi-agent implique de décrire le comportement de chaque agent, dans notre cas les agents composants et les agents instances. Dans les systèmes multi-agents adaptatifs nous parlons plus précisément de comportement coopératif car la coopération dans le monde AMAS est le moteur de l'auto-organisation des agents. Une phase essentielle dans la conception de tels systèmes est le recueil pour chaque agent des SNC. L'ensemble de ces situations ont été décrites dans la section 4.1. Lorsqu'elles sont identifiées, le concepteur décrit pour chacune d'entre elles un comportement ramenant l'agent en situation de coopération. Ainsi quelle que soit la situation dans laquelle il se trouve, l'agent se comporte toujours de façon coopérative avec les autres agents du système. Le comportement de l'ensemble de nos agents ne peut être décrit de façon exhaustive car il est directement relié à la nature intrinsèque du composant : les agents responsables d'un composant « Télécommande » n'auront pas exactement le même comportement que ceux responsables d'un composant « Lecteur Vidéo ». Nous pouvons cependant dégager quelques règles génériques :

1/ Lors de la création d'un agent composant (suite à l'arrivée d'un nouveau dispositif) celui-ci se considère inutile dans l'environnement. Pour se rendre coopératif il prévient son voisinage qu'il fournit un ensemble de services (envoi d'un *RequestRequired* pour l'ensemble des interfaces fournies du composant).

2/ Lors de la création d'un agent instance, celui-ci prévient son voisinage qu'il est à la recherche de composants fournissant les interfaces qu'il requiert (envoi d'un *RequestProvided* pour l'ensemble des interfaces requises du composant).

3/ Si un agent composant reçoit une requête à la recherche d'une interface qu'il fournit (*RequestProvided*), il crée un agent instance et lui transfère la demande.

4/ Si un agent instance reçoit l'acceptation d'une requête à la recherche d'une interface que le composant requiert (*AcceptProvided*), l'agent met à jour sa liste d'agents ayant accepté la demande pour cette interface. Cette liste est constamment triée par niveau de compétence (*capability*). Si l'instance de composant que gère cet agent est déjà connectée à une autre instance et que l'agent accepteur a un niveau de compétence meilleur pour cette interface, l'agent fait une demande de remplacement de connexion (*ReplaceConnect*) à l'agent accepteur. De plus, si le niveau de compétence global de l'agent a augmenté, celui-ci propage à son voisinage sa nouvelle compétence (*PropagateCapability*).

Un ensemble de comportements plus spécifiques reste à la charge du concepteur/développeur qui utilisera ce modèle. Si par exemple nous nous situons au niveau des agents instances, nous pouvons nous poser la question de quel comportement adopter lorsque nous voulons faire une demande de connexion effective (*Connect*) : que faire si l'agent pour l'une de ses interfaces requises n'a eu aucune réponse à sa demande (situation d'inutilité) ? Se supprimer ? Attendre encore un moment ? Que faire si, pour une interface requise, plusieurs agents à niveau de compétence équivalent ont répondu favorablement à la requête d'un agent (situation de conflit) ? Prendre n'importe lequel ? Demander à l'utilisateur de faire un choix ?

4.3 Discussion

En regard de la grille d'analyse (cf. 3.1), nous pouvons caractériser notre système ainsi :

- Les composants et le système produit par composition : nos agents sont un support à la composition de composants logiciels qui sont contenus dans des dispositifs distribués dans l'environnement. Le système multi-agent adaptatif cherche à produire des assemblages pour former des applications ambiantes.
- Le système de composition : d'un point de vue technique, la composition entre composants logiciels est rendue possible par l'unification (*matching*) syntaxique entre interfaces requises et interfaces fournies. Pour cela chaque agent instance de composant recherche les « meilleurs » agents instances fournissant des services qu'il requiert au travers de ses interfaces requises. Le contrôle de la composition est ici décentralisé au travers des agents instances qui sont totalement autonomes et capables de se composer avec autrui. La combinatoire de la composition est contrôlée par plusieurs éléments. Tout d'abord chaque agent instance limite ses recherches à son voisinage. Ensuite, il ne demande à connecter ses interfaces requises que sur les agents instances qu'il considère les plus compétents. Lors de l'entrée ou la sortie de dispositifs et donc de composants dans l'environnement, le système n'est pas perturbé : il se réajuste par auto-organisation de ses agents composants et agents instances.
- L'expression des besoins et des buts : du point de vue du système le besoin n'est pas explicite. Les applications disponibles pour l'utilisateur se construisent automatiquement au travers des composants se trouvant à sa portée. Pour atteindre son but l'utilisateur interagit avec des dispositifs. Les actions de l'utilisateur ou les modifications de l'environnement amènent le système à réorganiser dynamiquement les assemblages de telles façons qu'ils soient toujours les plus adéquats possibles avec les besoins de l'utilisateur.

Ainsi la technologie AMAS supporte la création et le maintien en production des applications dans un environnement ambiant. En utilisant notre modèle, il reste à la charge du concepteur la description de comportements pour l'ensemble de ses agents composants et instances. Ces comportements sont plus simples à appréhender car ils sont locaux et qu'ils ne font a priori aucune hypothèse sur la fonction globale du système.

4.4 Prototype

Le prototype que nous avons développé [2] est une première validation expérimentale de notre système. Cette version fait les hypothèses simplificatrices suivantes :

- l'unification (*matching*) sémantique entre composants qui est un problème ouvert n'est pas considérée : nous ne travaillons pas sur le sens ni du composant, ni du produit de la composition ;
- les composants possèdent une seule interface fournie et plusieurs interfaces requises (simplification de la combinatoire).

Notre implémentation s'exprime sous forme d'architecture logicielle. Pour cela nous nous appuyons sur le langage SpeADL [20] qui introduit des abstractions utiles aux systèmes multi-agents. L'outil MAY (*Make Agents Yourself*), qui est intégré à Eclipse sous forme de plugin, permet

de transformer automatiquement nos descriptifs SpeADL en agents exécutables implémentés en Java. L'environnement dans lequel s'exécute notre système est entièrement simulé (simulation des dispositifs, d'un annuaire distribué pour la gestion du voisinage et du *middleware*).

A ce jour nous avons implémenté un premier système supportant cette architecture. Nous sommes partis d'un scénario prenant en compte des dispositifs multimédias accessibles dans une maison puis nous avons implémenté pour l'ensemble de nos composants les comportements spécifiques des agents composants et des agents instances.

5 Conclusion et perspectives

Avec les systèmes ambiants, nous sommes aujourd'hui confrontés à de nouveaux défis en matière de construction et de déploiement de logiciels. Dans ce cadre, l'approche traditionnelle descendante qui consiste à développer un produit logiciel à partir de besoins exprimés par un client et à baser la validation du produit sur ces besoins, pourrait laisser la place à une démarche beaucoup plus ascendante et opportuniste dans laquelle le produit logiciel pourrait émerger de la composition d'éléments disponibles dans l'environnement. De par la dynamique des systèmes ambiants (mobilité des composants, volatilité des services et des ressources, ouverture…) et leur naturelle décentralisation, les systèmes multi-agents sont candidats à la réalisation de ces assemblages dynamiques : des agents, répartis et embarqués au sein des dispositifs qui constituent le système ambiant, peuvent coopérer pour assembler les composants puis déployer et maintenir les applications émergentes produites.

Dans ce travail, nous avons fait une première expérimentation qui a montré la faisabilité de la composition opportuniste et ascendante (en faisant quelques hypothèses simplificatrices) et un prototype a été développé. Les agents coopératifs (agents composants et agents instances) interagissent localement pour résoudre des « situations non coopératives » relatives à la composition entre interfaces fournies et interfaces requises, ou à la modification des liaisons (reconfiguration) pour une meilleure qualité de service. L'autonomie des agents et la distribution du contrôle sont des éléments essentiels dans la construction de ces systèmes dont l'environnement est mal connu et en constante évolution. L'utilisation d'AMAS permet d'aborder de façon ascendante la composition autonome d'applications ambiantes : les applications se créent et se reconfigurent grâce aux comportements locaux des agents.

Parmi les problèmes ouverts, il y a celui du contrôle de la combinatoire (nombre de compositions possibles) et de la validation du produit qui émerge de la composition. A quelle condition, jusqu'à quelle limite compose-t-on ? Faut-il contrôler les applications qui émergent et quelles applications doit-on laisser émerger ? Quel rôle peut jouer l'utilisateur dans le contrôle ? Faut-il contrôler *a priori* la composition (par exemple, au moyen de statistiques sur le comportement passé de l'utilisateur, ou plus généralement en identifiant des situations contextuelles particulières) ou *a posteriori* (par exemple, au moyen de *feedbacks* de l'utilisateur et de mesures de satisfaction) ?

On peut se demander si l'explicitation de besoins peut effectivement disparaître (dans une approche purement ascendante, c'est l'apparition du produit qui crée, ou pas, le besoin), si les besoins (fonctionnels) peuvent être déduits du comportement des utilisateurs. On peut aussi s'interroger sur la prise en compte des besoins selon leur nature : dans notre cas d'étude, il semble que les besoins fonctionnels soient à l'origine de la création d'un assemblage (configuration) alors que les besoins non fonctionnels semblent moteurs dans les opérations de reconfiguration.

Dans ce travail, nous avons également fait abstraction d'un certain nombre de problèmes techniques relatifs à l'assemblage et à l'interopérabilité qu'il conviendrait d'analyser plus finement avant d'y apporter des réponses. Plus généralement, une réflexion approfondie doit certainement être menée en termes d'usages, d'acceptabilité, de sécurité…

6 Références

[1] Weiser M., The *Computer for the Twenty-First Century*, Scientific American 265, 3, September 1991.

[2] Denis G., *Composition autonome et émergence au moyen d'agents-composants coopératifs*, Rapport de Master 2 Recherche, Université Paul Sabatier, Toulouse, 2011.

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

[4] Szyperski C., *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[5] Bass L., Clements P., Kazman R., *Software Architecture in Practice*, Addison Wesley, 1998.

[6] Medvidovic N., Taylor R.N. *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Trans. Softw. Eng., 26(1):70–93, 2000.

[7] Briot J.-P. Composants logiciels et systèmes multi-agents, A. El Fallah Seghrouchni and J.-P. Briot, eds, *Technologies des systèmes multi-agents et applications industrielles*, Chapitre 5. Lavoisier, Paris, France, 147-187, 2009.

[8] Vergoni C., Tigli J.-Y., Rey G. et Lavirotte S., Construction Bottom-up d'applications ambiantes en environnements partiellement connus a priori. *7ème journées francophones Mobilité et Ubiquité – Ubimob*, 2011.

[9] Lau K.-K., Wang Z. Software Component Models, *IEEE Transactions on Software Engineering* 33(10):709-724, October 2007.

[10] Stavropoulos T. G., Vrakas D., Vlahavas I., A survey of service composition in ambient intelligence environments, *Artificial Intelligence Review*, 24 September 2011.

[11] Thomas L., Luner J., Roman G.-C., Gill C., Achieving coordination through dynamic construction of open workflows, *10th ACM/IFIP/USENIX International Conference on Middleware*, p268-287, Urbanna (Illinois), 2009.

[12] Vallée M., *Un intergiciel multi-agent pour la composition flexible d'applications en intelligence ambiante*. Thèse de doctorat, Ecole Nationale Supérieure des Mines, St-Etienne, 2009.

[13] Kiczales G., Lamping J., Mendhekar A., Maeda C., Videira Lopes C., Loingtier J.-M., Irwin J. Aspect-Oriented Programming. *European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, 220–242. Springer-Verlag, 1997.

[14] Kramer J, Magee J., A rigorous architectural approach to adaptive software engineering, *Journal of Computer Science and Technology* 24(2): 183–188 Mar. 2009.

[15] Sykes D., Heaven W., Magee J., Kramer J., Exploiting non-functional preferences in architectural adaptation for self-managed systems, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10).* ACM, New York, NY, USA, 431-438.

[16] Urbieta A., Barrutieta G., Parra J., Uribarren J., A Survey of Dynamic Service Composition Approaches for Ambient Systems, *1st Int. Conf. of Ambient Media and Systems*, 2008, Canada.

[17] Georgé J.-P., Gleizes M.-P., Experiments in Emergent Programming Using Self-organizing Multi-Agent Systems. *4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'05)*, Springer Verlag, p. 450-459, septembre 2005.

[18] Georgé J.-P., Gleizes M.-P., Glize P., Conception de systèmes adaptatifs à fonctionnalité émergente : la théorie des AMAS, *Revue d'Intelligence Artificielle*, Hermès Science Publications, Vol. 17, N. 4/2003, p. 591-626, 2003.

[19] Noël V., Arcangeli J.-P., Gleizes M.-P., Une approche architecturale à base de composants pour l'implémentation des systèmes multi-agents, *Revue des Nouvelles Technologies de l'Information*, à paraître, 2012.