Une approche architecturale à base de composants pour l'implémentation des Systèmes Multi-Agents

Victor Noël, Jean-Paul Arcangeli, Marie-Pierre Gleizes

Une approche architecturale à base de composants pour l'implémentation des Systèmes Multi-Agents

Victor Noël, Jean-Paul Arcangeli et Marie-Pierre Gleizes

Institut de Recherche en Informatique de Toulouse
Université de Toulouse
118, route de Narbonne, 31 062 Toulouse Cedex, France
{Prenom.Nom}@irit.fr

Résumé. Motivés par le développement des Systèmes Multi-Agents (SMA), nous explorons dans cet article la production de supports de développement orientés agent spécialisés en utilisant des architectures logicielles à composants. L'objectif de ce travail est de faciliter le passage de la conception du SMA, en termes de types d'agents et d'interactions, à son implémentation, à l'aide de ce que nous nommons une micro-architecture. Celle-ci est un moyen de prendre en compte les exigences que la conception orientée agent ne considère pas. À l'aide d'un exemple réel, nous mettons en évidence les spécificités des applications SMA et les implications architecturales de celles-ci. La principale contribution de cet article réside dans la définition du modèle de composants SPEAD (Species-based Architectural Design) qui introduit un type spécifique de composants, le *transverse*, qui permet de réaliser l'interconnexion entre les agents du système et leur plateforme d'exécution. Cette abstraction est complétée par deux autres, l'*espèce* et l'*écosystème*, qui supportent au niveau de la micro-architecture la réalisation des concepts manipulés dans les SMA. Nous présentons une implémentation de SPEAD sous forme d'un langage de description d'architectures, utilisable en conjonction avec JAVA. Ce langage est utilisé dans notre équipe pour supporter le développement dans le cadre de projets de recherche.

1 Introduction

La **modélisation** d'un problème et/ou de sa solution par un Système Multi-Agent (SMA) est une manière de concevoir des applications. De tels **systèmes** sont composés d'entités, les **agents**, qui interagissent au sein d'un **environnement** faisant office de médium d'interaction (Ferber, 1995). La particularité de cette modélisation est que l'effort de conception est focalisé sur la définition des **comportements individuels** des agents et de leurs **interactions**, pour permettre au système d'exhiber un **comportement global** répondant aux exigences initiales (Ferber, 1995; Demazeau, 1995).

Ainsi, le contrôle de l'exécution du SMA est distribué dans les agents. Les comportements, qui sont basés sur des interactions plus ou moins complexes entre les agents, ont le plus souvent

une dimension locale, c'est-à-dire qu'ils ne s'appuient pas sur des informations globales au système (Ferber, 1995), mais seulement sur des informations venant de l'environnement ou fournies par les autres agents avec qui ils communiquent. Cela fait d'ailleurs des SMA une approche particulièrement adaptée aux systèmes distribués, décentralisés ou acentrés.

La modélisation par SMA est principalement utilisée dans le domaine de l'Intelligence Artificielle Distribuée (résolution de problèmes, systèmes adaptatifs, etc.), de la Simulation (modélisation sociale), de la Robotique Collective, de l'Intelligence Ambiante, etc. Le travail de notre équipe de recherche est avant tout centré sur l'ingénierie des SMA pour répondre à des problèmes complexes, dynamiques et partiellement spécifiables (Gleizes et al., 2008) [1].

Nous sommes particulièrement intéressés par les problèmes relatifs au développement des SMA, et, dans ce cadre, nous étudions et nous proposons des méthodes et des outils pour aider à leur développement. Il s'agit là d'un enjeu majeur pour la communauté SMA : l'utilisation effective des technologies multi-agents par un plus large public ou par l'industrie du logiciel impose d'une part de réduire l'écart entre conception et implémentation, donc de permettre une implémentation directe des concepts et des techniques multi-agents (Schelfthout et al., 2002; Bordini et al., 2005; Molesini et al., 2007), et d'autre part de limiter les coûts de production et de maintenance en favorisant la flexibilité, la composition et la réutilisation. Cet article présente donc une contribution à ce problème, fondée sur une démarche d'architecture logicielle et le concept de composant logiciel.

De nombreux travaux visent la **conception** de ces systèmes et un certain nombre de **méthodes de développement** en ont résulté (Bergenti et al., 2004; Henderson-Sellers et Giorgini, 2005). Ces méthodes aident à identifier et modéliser les agents du SMA réalisé, leurs moyens d'interaction, directs ou indirects (échange de messages, environnements partagés, organisations sociales, etc.) et les éléments de l'environnement. Pour cela, elles proposent des modèles d'agents et de SMA. Ensuite, en se basant sur différentes **approches**, ces méthodes proposent des guides pour définir le comportement des agents exploitant les mécanismes d'interaction : toute la difficulté dans cette étape est de déterminer les comportements individuels adéquats qui permettront au système global d'avoir le comportement global escompté. Nous différencions l'approche, qui apporte une solution pour résoudre un problème avec les SMA, de la méthode, qui accompagne l'application de cette approche, éventuellement en allant jusqu'à l'implémentation. Pour le reste de cet article, nous nous intéressons seulement à l'application des approches, éventuellement dans le cadre de méthodes. En effet, l'objectif de notre travail est de supporter ce qui dans ces méthodes est considéré comme un travail d'implémentation.

À partir du résultat de la phase de conception, le **développeur du SMA** n'a « plus qu'à » **implémenter** le système, les agents, leurs mécanismes d'interactions, tout l'outillage associé (interfaces graphiques, *scheduler*…) et à l'intégrer à des systèmes informatiques existants le cas échéant. Heureusement, la plupart des méthodes fournissent des **outils** et des **supports de développement** (Bordini et al., 2006, 2009) pour faciliter ce travail en fonction du résultat de l'application des méthodes. Les supports de développement SMA proposent des ensembles d'abstractions de programmation adaptées aux modèles qu'ils supportent. En revanche, le plus souvent, ils accompagnent le modèle utilisé dans les méthodes ou proposent leurs propres modèles d'agents, et ne prennent pas en compte, comme nous le montrons ensuite, le fait que chaque application a des exigences qui doivent être traitées après la conception du SMA lui-même. En quelques mots, chaque application d'une approche SMA à un problème donné

1. Pour des exemples, voir `http://www.irit.fr/SMAC`.

introduit ses propres « types d'agent » en fonction de leur dynamique, des mécanismes d'interaction qu'ils utilisent, de leurs comportements, etc, mais aussi en fonction de l'environnement et de toutes les exigences qui ne sont pas traitées par les approches elles-mêmes telles que la connexion à des systèmes externes ou la production d'interface graphique pour contrôler ou observer le système. Nous avons précédemment traité la question de façon détaillée et mis en avant les motivations de nos travaux pour combler le fossé entre conception et implémentation dans les SMA dans Noël et al. (2010).

À l'opposé de ces travaux proposant des supports de développement, en nous inspirant des approches utilisant les architectures logicielles (Bass et al., 2003; Clements et al., 2003) et les composants logiciels (Bachmann et al., 2000; Szyperski et al., 2002), nous nous sommes intéressés à des moyens de produire des briques réutilisables pour construire, « à la carte », des supports de développement orientés agent qui soient **adaptés** à une application donnée et aux besoins de ses développeurs. Pour cela, nous proposons de fabriquer des **composants** et des **architectures à composants** à l'aide d'un langage de description d'architecture (ADL) (Medvidovic et Taylor, 2000) pour réaliser l'architecture logicielle supportant le SMA et le cadre nécessaire à la programmation des agents à exécuter. L'apport de cet article à cette problématique est multiple : en terme de caractérisation de la spécificité des SMA en rapport avec cet objectif, en terme de modèle de composants et d'ADL pour appliquer cette approche et définir des briques réutilisables répondant aux spécificités des SMA, et enfin en terme d'outil pour supporter ce développement.

L'article suit le plan suivant : dans la partie 2, nous motivons la proposition à l'aide d'un exemple tiré d'une application réelle, l'analysons et généralisons cette analyse aux SMA. Cela nous permet d'en tirer un ensemble de besoins auxquels nous répondrons. Dans la partie 3, nous présentons la proposition en terme de modèle de composants et d'ADL. Nous présentons l'outil développé pour faciliter son utilisation, et nous décrivons brièvement quelques composants produits ainsi que des retours d'expérience. Dans la partie 4, nous illustrons notre proposition dans le cadre de notre exemple. Dans la partie 5, nous plaçons la contribution dans un contexte méthodologique. Dans la partie 6, nous discutons la proposition et la positionnons par rapport à d'autres travaux. Enfin nous concluons avec une analyse de la contribution en réponse aux besoins exprimés précédemment.

2 De la conception à l'implémentation des SMA : analyse et motivations

Un Système Multi-Agent est constitué d'**entités** passives (objets, bases de données...) ou actives (agents, objets actifs...) qui interagissent au sein d'un **environnement** (plan spatial, réseau, structure organisationnelle...) en utilisant des **mécanismes d'interaction** (action et perception : envoi de messages, écriture et lecture sur tableau noir, accès à des capteurs ou actionneurs physiques tels qu'une caméra ou les roues d'un robot, mobilité logicielle sur le réseau...). L'environnement fait office de médium d'interaction (stigmergie [2], organisation structurée, observation du voisinage spatial ou social, tableau noir, gestionnaire d'événements...). Un agent (cognitif, réactif...) a une dynamique définie par son **cycle de vie** et paramétrée par un

2. Dépôt et captage de « phéromones » dans un environnement situé (modèle d'interaction souvent utilisé en intelligence artificielle).

comportement et un **état** (connaissances, objectifs...) qui le caractérisent en tant qu'individu. La dynamique et le comportement nécessitent des mécanismes internes à l'agent (protocoles d'interaction, aptitudes computationelles, moteur de raisonnement...) exploitant éventuellement les mécanismes d'interaction.

La conception complète d'un SMA couvre un large panel de choix de conception, allant de la définition des agents, de leurs modalités d'interaction et de leurs comportements, à la résolution d'exigences qui ne sont pas spécifiquement orientées agent (exécution, visualisation...), ni couvertes par les méthodes SMA qui sont orientées vers la fonctionnalité du système.

2.1 La conception dans le cadre des SMA

La conception SMA est obtenue au moyen de méthodes qui permettent de décrire comment les agents interagissent, avec quel comportement, dans quel environnement, pour répondre aux besoins fonctionnels du système à construire.

En particulier, une spécificité des SMA est la façon dont les interactions entre les différents éléments du système, les agents, sont effectuées. Contrairement aux pratiques usuelles dans les architectures à composants (par exemple avec les configurations des composants composites dans les ADL) où l'organisation des composants est décrite de façon plus ou moins statique, les agents d'un SMA ont la capacité de choisir dynamiquement, à l'exécution, avec qui ils échangent de l'information. En revanche, ce qui est fixé est l'ensemble des mécanismes d'interaction mis à leur disposition.

Par exemple, l'envoi de messages est un moyen d'interaction très utilisé dans les SMA. Dans ce cas, l'organisation des agents résulte du fait que chaque agent connaît d'autres agents, en a une représentation, maintient ses représentations et les fait éventuellement évoluer selon son comportement. De plus, lorsque l'environnement fait office de médium pour les interactions, il contraint celles-ci.

2.2 Exemple : jeu sérieux

Pour illustrer nos motivations et les concepts présentés ici, nous décrivons maintenant un exemple de SMA tiré d'un système actuellement développé au sein de notre équipe de recherche dans le cadre du projet GAMBITS [3].

L'objectif du système est d'adapter dynamiquement les valeurs des paramètres de configuration d'un jeu sérieux [4] pour l'entraînement d'opérateurs de surveillance maritime. En fonction de *feedbacks* venant du joueur, le système adapte la configuration du jeu au niveau du joueur et permet un apprentissage progressif. En effet, certains paramètres du jeu sont modifiables et ont un impact sur l'apprentissage du joueur : en ajustant ces paramètres en accord avec un objectif pédagogique, il est donc possible d'adapter au mieux cet apprentissage.

Pour répondre à ce besoin, l'application de l'approche SMA nommée AMAS (Adaptive Multi-Agent System) (Gleizes et al., 2008) amène les concepteurs du système à définir le SMA en terme d'agents coopératifs, de mécanismes d'interaction et de comportements. Dans cette approche, un système peut s'auto-organiser selon trois « niveaux » possibles : modification des paramètres internes des agents, réorganisation du système (changement des connexions

3. Game-Based Strategic Intelligence : `http://www.gambits.fr/`, financé par la DGCIS (Direction Générale de la Compétitivité, de l'Industrie et des Services).

4. Un jeu sérieux est un jeu dont l'objectif premier est autre que le divertissement (Alvarez, 2007).

entre les agents) et évolution du système (*i.e.* création et destruction d'agents). Toutes ces réorganisations sont déclenchées par les agents en suivant des règles locales qui correspondent à leurs comportements, conçues à l'aide de l'approche AMAS.

Résultat de l'application de l'approche à GAMBITS. Les agents du système interagissent en utilisant :

- l'envoi de messages : communication asynchrone, les messages reçus sont stockés dans une boîte aux lettres interne à l'agent qu'il peut consulter, et
- l'observation (perception de l'environnement) : accès en lecture à une partie de l'état d'un agent que ce dernier publie à destination des autres agents.

Les messages reçus par un agent sont traités par son comportement, qui peut modifier son état, une partie de celui-ci étant observable. Les agents possèdent des connaissances, et, entre autres, connaissent d'autres agents avec qui ils interagissent à l'aide de ces deux mécanismes.

Dans GAMBITS, quatre types d'agents ont été définis. Ils diffèrent soit pas les mécanismes d'interaction utilisés (par exemple, certains agents ne reçoivent pas de messages, certains n'observent pas l'état des autres agents, d'autres encore ne sont pas observables, etc.), soit par leur comportement et leur rôle dans le système.

L'utilisateur du système peut modifier l'état de certains des agents pour envoyer un *feedback* au système et provoquer son auto-organisation.

Le cycle de vie de chacun des agents est le suivant :

1. l'agent perçoit son environnement — par exemple il consulte les derniers messages reçus et observe l'état d'autres agents — puis interprète les informations perçues pour mettre à jour ses connaissances ;
2. il décide des prochaines actions à effectuer en fonction de règles internes de décision ;
3. enfin, il effectue les actions choisies : envoyer des messages à d'autres agents, changer son état interne et/ou observable, créer d'autres agents ou disparaître (« suicide »).

Nous ne détaillons pas davantage le comportement, celui-ci ayant peu d'impact sur la suite de cet article.

Exigences et choix de conception supplémentaires. En plus du résultat de la conception SMA, un certain nombre d'exigences et contraintes supplémentaires existent. Elles sont supplémentaires dans le sens où elles ne sont pas traitées par l'approche SMA choisie ou apparaissent après son application. Par exemple, elles ont pu être identifiées par le client, par les développeurs ou par le fait qu'une approche SMA particulière est appliquée et introduit ses propres contraintes.

Dans le cas de GAMBITS, le système est cadencé par une horloge globale. Pour des raisons de synchronisation du système, induites par la définition des comportements, ce cadencement doit garantir que tous les agents ont effectué la partie perception et décision de leur cycle de vie avant de tous effectuer leurs actions.

En particulier, le système exécute en boucle les actions suivantes :

- mise à jour de l'état des agents modifiés par l'utilisateur ;
- exécution de la partie perception/décision de tous les agents du système ;
- exécution de la partie action de tous les agents du système ;
- mise à jour des interfaces de visualisation.

De plus, il est demandé de mettre en place un mécanisme pour créer les agents du SMA à partir d'une description d'un scénario de jeu. Le scénario contient les informations du problème qui se trouvent représentées par des agents dans le système.

Par ailleurs, l'arrêt de l'exécution du système est conditionnée par un certain nombre d'informations liées aux agents. Celles-ci doivent donc être récupérées à intervalles réguliers. Ces informations sont aussi celles qui peuvent être visualisées.

Enfin, de par l'organisation de l'équipe de développement, il existe deux contextes différents d'exécution pour le SMA : mise au point et intégration. Dans le premier cas, il est nécessaire de faire la mise au point du système principalement à travers la modification des comportements des agents. Pour cela, le concepteur veut avoir accès à des informations sur l'état interne des agents à travers des interfaces graphiques de visualisation. Dans le second cas, lors de l'intégration avec le reste du système (principalement le moteur de jeu et l'interface utilisée par l'utilisateur) seuls des *logs* textuels sont utilisés.

2.3 Analyse

À travers l'analyse qui suit, nous généralisons au développement des SMA les remarques que l'on peut faire à propos de l'exemple et nous en tirons un ensemble de besoins pour supporter leur implémentation.

Spécificité de l'application à développer. La conception du SMA présenté a mis en évidence plusieurs types d'agents : en plus des spécificités liées à la méthode utilisée ou au domaine d'application, comme les moyens d'interaction utilisés ou le cycle de vie de l'agent, certains besoins comme le cadencement ou la manière de créer les agents ont un impact sur la façon dont sont structurés les agents et leurs comportements.

Dans GAMBITS par exemple, les comportements sont décrit en terme d'interprétation des perceptions, de mise à jour de connaissances et de règles de décisions. Pour cela, il faut mettre à disposition des comportements les mécanismes d'interaction comme l'envoi et la réception de messages ou l'observation d'autres agents. Mais en plus, la manière dont ils sont cadencés fait que ce comportement doit être déclenché par le système, tout en garantissant que les messages envoyés de façon concurrente arrivent à destination, et que la consultation de l'état d'autres agents ne bloque pas leur exécution. Enfin, la nécessité de visualiser l'état des agents implique la mise en place de mécanismes de consultation et d'affichage de cette information, et cela de manière désactivable et sans effet de bord sur la façon dont le système s'exécute.

Ces besoins viennent du choix de l'approche, des choix de conception faits avec l'approche, du domaine d'application ou encore de la façon dont est organisée l'équipe de développement.

Spécificité des SMA. Chaque méthode, chaque approche et même chaque application (donc chaque SMA à produire) a ses **propres types**[5] **d'agents** avec leurs propres mécanismes d'interactions, leurs propres manières de les exploiter et ce de façon adaptée au problème à traiter.

C'est en ce sens que la **programmation agent** se différencie de paradigmes de programmation plus classiques. Par exemple, programmer en objet, c'est traduire tous les concepts manipulés, et en particulier les interactions entre objets, en terme de définition et d'appel de méthode. Différemment, les SMA mettent en avant une diversité de modes d'interaction, entre

5. Nous entendons le terme « type » au sens large et non formel.

autres envoi de messages, communication de groupe, interaction indirecte à travers l'environnement, etc.

Chaque type d'agents peut être vu comme une machine abstraite avec ses entrées (capteurs), ses sorties (actionneurs), sa sémantique et sa dynamique. Alors, programmer le comportement d'un agent c'est programmer la machine abstraite. Chaque type d'agents apporte en quelque sorte son propre **paradigme de programmation** avec ses propres primitives ou concepts de programmation : ainsi programmer un agent, c'est utiliser cet ensemble cohérent de concepts pour la programmation de son comportement. Plus de détails sur ce point de vue peuvent être trouvés dans Noël et al. (2010).

Cela nécessite donc de mettre en place les abstractions adaptées à l'application à construire, et de les implémenter pour pouvoir produire un système exécutable. Donc, implémenter un SMA demande un effort supplémentaire pour à la fois mettre en œuvre ce paradigme de conception et de programmation spécifique, puis pour l'utiliser pour développer son SMA.

De plus, comme l'exemple le met en avant, la différence entre l'agent en tant que tel et l'environnement, dans lequel il « vit » et interagit, est fondamentale dans les SMA. En effet, cela est causé par la décentralisation du contrôle dans les agents, l'utilisation importante de mécanismes d'interaction et la création dynamique d'agents. Ainsi, à l'exécution, il est nécessaire de différencier les entités logicielles que sont les agents de leur **plateforme d'exécution**.

Conception micro-architecturale. Nous proposons donc de faire la différence entre deux phases de conception :

1. **Conception SMA** : application de la méthode SMA qui produit une description du système en terme d'abstractions telles que les types d'agents du système, leurs mécanismes d'interactions, les éléments de l'environnement dans lequel ils interagissent, etc.
2. **Conception micro-architecturale** [6] : mise en place des abstractions adaptées aux concepts introduits lors de la conception SMA et prise en compte du reste des exigences de l'application à développer (cadencement des agents, synchronisation, visualisation, architecture interne des agents, exécution des comportements, etc). La micro-architecture réalise donc à la fois les agents eux-mêmes mais aussi leur plateforme d'exécution.

Contrairement à l'architecture du SMA (résultat de la conception SMA), sorte de « macro-architecture » qui s'intéresse à définir le comportement global du système à travers la définition des comportements des agents, la micro-architecture (résultat de la conception micro-architecturale) se concentre sur l'architecture des éléments de cette « macro-architecture » pour réaliser ce qui supportera leur exécution.

En effet, autant la première a pour objectif de répondre aux exigences fonctionnelles — voire non-fonctionnelles comme l'adaptation ou la performance — du système qui sont traitables par les approches SMA, autant la seconde a pour objectif de construire les abstractions nécessaires à la première, et ce en prenant en compte toutes les exigences non considérées par ces approches. Ces objectifs ne sont pas nouveaux en tant que tels dans le contexte des architectures logicielles, mais se matérialisent de façon particulière dans le cadre des SMA.

À l'issue de ces deux phases, la micro-architecture est implémentée, et celle-ci sert ensuite de support à l'implémentation du SMA qui exploite les abstractions mises en œuvre par la micro-architecture.

6. Ce terme a été utilisé dans un autre cadre (Johnson, 1997) qui, comme nous le verrons dans la section 6, n'est pas si éloigné des idées avancées ici.

Pour clarifier le discours, nous qualifions dans la suite de « métier » les abstractions introduites lors de la conception SMA et de « opératoires » les abstractions introduites et les exigences traitées ensuite. Ainsi la plateforme d'exécution réalise l'environnement au sens métier mais aussi tous les mécanismes nécessaires à l'exécution des agents et à la satisfaction des autres exigences opératoires.

2.4 Besoins pour la réalisation de la micro-architecture

La mise en avant de l'importance de la micro-architecture nous permet d'identifier deux dimensions d'étude de l'aide au développement des SMA :

- la dimension architecturale se focalise sur la réalisation de la micro-architecture elle-même, et s'appuie sur la différenciation entre agents et plateforme d'exécution.
- la dimension méthodologique s'intéresse à la définition des abstractions fournies par la micro-architecture, et à leur utilisation pour programmer un SMA.

Cette partie identifie trois besoins spécifiques pour concevoir et implémenter des SMA répondant à la dimension architecturale, cœur de notre contribution. Ils sont accompagnés d'explications illustrées par GAMBITS. La dimension méthodologique sera abordée dans la partie 5.

Définition de mécanismes d'interconnexion entre les agents et la plateforme d'exécution. Ce point reflète le besoin d'exprimer et de réaliser les mécanismes mis à disposition des agents par la plateforme d'exécution, mais aussi ce que cette plateforme d'exécution requiert des agents pour les faire fonctionner. Par exemple, pour envoyer des messages il est nécessaire que la plateforme d'exécution fournisse un mécanisme d'envoi de messages, mais aussi requiert de l'agent qu'il fournisse un moyen de déposer les messages dans sa boîte aux lettres. Ou encore, la plateforme requiert de l'agent qu'il lui fournisse un moyen d'exécuter son comportement.

Tous ces mécanismes répondent à la fois à des besoins d'interaction métiers, mais aussi à des besoins opératoires.

En quelque sorte, le besoin est d'interconnecter deux « systèmes » : l'architecture de la plateforme d'exécution et l'architecture de l'agent. En effet, même si dans les SMA, conceptuellement parlant, certaines interactions sont directes entre agents, il n'y a en pratique que des interactions qui passent architecturalement par la plateforme.

De plus, il est souhaitable que ces mécanismes soient réutilisables dans différentes applications. Typiquement, le mécanisme d'envoi et réception de messages doit pouvoir être réutilisé dans plusieurs applications orientées agent.

Définition de types d'agents dédiés. Ce besoin résulte de la présence de différents types d'agents au sein d'une application. Ces types d'agents ont la particularité d'être dédiés au SMA à construire, de par les mécanismes d'interaction qu'ils utilisent ainsi que leur architecture interne et la façon dont celle-ci fonctionne et est liée à la plateforme qui les exécute. Les types d'agents doivent permettre, lors de la conception, d'être le point de départ de l'explicitation des choix architecturaux qui répondent aux exigences de l'application.

De plus, il est nécessaire de pouvoir composer divers mécanismes d'interaction réutilisables pour définir un type d'agent. Ainsi, les agents dans l'exemple du jeu sérieux sont connec-

tés en même temps à la plateforme à travers un mécanisme d'envoi et réception de messages, un mécanisme d'observation et de publication d'état, un mécanisme de cadencement, etc.

Création et interconnexion dynamique d'agents. Le dernier besoin micro-architectural concerne l'instanciation des types d'agents, c'est-à-dire la création dynamique d'agents à l'exécution. Il s'agit ici de connecter dynamiquement (mais suivant un schéma prédéfini à travers la définition du type d'agent) les agents et la plateforme d'exécution : chaque agent doit être connecté à la plateforme à travers chacun de ses mécanismes d'interconnexion. Cette connexion ne peut pas être effectuée au démarrage du système, mais à l'exécution car les agents peuvent être créés ou détruits dynamiquement.

Cela a bien sûr des implications sur la manière de définir ces mécanismes et sur la manière de définir les types d'agents. En effet, un mécanisme doit parfois être instancié à la fois du côté de la plateforme d'exécution et du côté de l'agent. Pour le mécanisme de réception de message, une partie du mécanisme et de son état est propre à l'agent : sa référence ; et une partie est propre à la plateforme : la distribution des messages. De même, pour la visualisation, une partie du mécanisme est du côté de l'agent pour récupérer et éventuellement traiter les informations propres à l'agent et l'autre partie est propre à la plateforme pour agréger ces informations et les afficher.

3 Le modèle de composants SPEAD

Nous proposons ici les modèles nécessaires pour définir les micro-architectures telles que proposées dans la section 2, mais aussi les implémenter de telle sorte que les mécanismes et les composants créés soient aptes à être réutilisés, et de telle façon que la création de nouvelles micro-architectures soit aisée pour les concepteurs de systèmes.

Précisément, nous proposons :

- SPEAD (Species-based Architectural Design), un modèle de composants supportant la réalisation de la micro-architecture dans les SMA.
- SPEADL (Species-based Architectural Description Language), un ADL permettant de décrire ces architectures à composants et de les implémenter.

Tous les termes non définis ici sont ceux communément admis (Medvidovic et Taylor, 2000) dans les ADL. En particulier, un composant est considéré comme étant un élément composable au sein d'une architecture, elle-même étant considérée comme une composition de composants. Une telle composition forme un composant dit composite. Nous introduisons seulement la différenciation entre classe et type de composants : une classe de composants est définie par sa description et son implémentation. La description joue le rôle de type mais aussi d'implémentation dans le cas de composants composés d'autres composants. De plus, selon que l'on se place à l'implémentation ou à l'exécution, un composant est considéré respectivement comme une classe ou une instance.

Nouvelles abstractions. Les abstractions introduites par SPEAD, au-dessus des abstractions classiques des modèles de composants, sont :

- L'**écosystème**, une architecture à composants réalisant la micro-architecture d'un SMA, telle que définie précédemment. La plateforme d'exécution du SMA est donc réalisée par l'écosystème qui est responsable de la création et de l'exécution des agents.

- L'**espèce**, une architecture à composants réalisant un type d'agent dans un écosystème donné. L'espèce peut être vue comme une classe à partir de laquelle des agents (individus de l'espèce) peuvent être construits dynamiquement par « instanciation ».
- Le **transverse**, un composant spécial ayant pour fonction de plonger les espèces dans l'écosystème. A l'exécution, il connecte les agents à la plateforme d'exécution : il est à la fois pour partie situé de manière unique dans la plateforme, et pour partie dupliqué dans chacun des agents qui l'utilisent.

Ainsi, un écosystème [7] est une architecture — définie en terme de composants et de transverses — qui fournit des moyens de créer des individus d'espèces d'agents qu'il propose. Une espèce est définie au sein d'un écosystème par son architecture et les transverses de l'écosystème qu'elle « utilise ». Le transverse est en particulier le moyen qui permet de réaliser et de réutiliser des mécanismes d'interaction utilisés par les espèces d'agents.

Notons que les notions d'espèce et d'écosystème diffèrent des notions d'agent et d'environnement utilisées dans les SMA : en effet, les premières recouvrent autant l'aspect métier du SMA à produire (types d'agents, interactions, organisation...), que l'aspect opératoire nécessaire à la réalisation pratique du SMA (moteur de règles, distribution, cadencement, visualisation...). Ces notions enrichissent donc celles définies par les méthodes de développement SMA et guident la construction de la micro-architecture.

Ces abstractions sont architecturales : elles permettent de rendre explicite la définition de types d'agents, une spécificité des SMA. Mais ce sont aussi des abstractions d'implémentation : elles facilitent la réalisation et l'implémentation de systèmes où l'on doit mettre en relation un ensemble d'entités interagissant à travers des mécanismes d'interconnexion, et ce de façon dynamique. Ceci sera détaillé dans la conclusion dans une analyse vis-à-vis des besoins exprimés précédemment.

SPEAD Les principaux éléments du modèle SPEAD sont graphiquement représentés dans la figure 1. Pour simplifier sa présentation, nous le définissons ici selon trois aspects :

- description : les éléments nécessaires à la description de composants quand on utilise le langage de description d'architecture SPEADL ;
- implémentation : les éléments nécessaires à l'implémentation de composants quand on utilise le langage de programmation JAVA en s'appuyant sur du code généré automatiquement à partir de la description ;
- exécution : la façon dont les composants et espèces sont instanciés et initialisés.

Ces distinctions ne sont pas faites dans la figure de façon à rendre celle-ci plus claire (seul le point de vue classes — descriptions et implémentations — est représenté dans la figure).

Nous présentons d'abord un modèle de composants qui servira de base à la proposition. Ce modèle pourrait éventuellement être remplacé par des modèles plus courants tels que celui de UML 2 ou de Fractal. Cela n'a pas été fait pour de simples raisons pratiques : les concepts introduits ici sont trop différents de ce qui se fait dans ces modèles pour permettre d'en donner une définition et implémentation claire et facilement utilisable pour développer des SMA. Nous introduisons ensuite la contribution au-dessus de ce modèle de composants.

7. Notons que le terme écosystème utilisé ici ne doit pas être confondu avec l'écosystème logiciel (Bosch, 2009) : bien qu'ils partagent leur origine écologique, un écosystème est ici un produit du développement et non le contexte dans lequel celui-ci est fait et contraint. Ainsi, le terme écosystème est naturellement bien adapté pour caractériser l'entité qui définit des espèces et est responsable de faire « vivre » les individus de ces dernières.

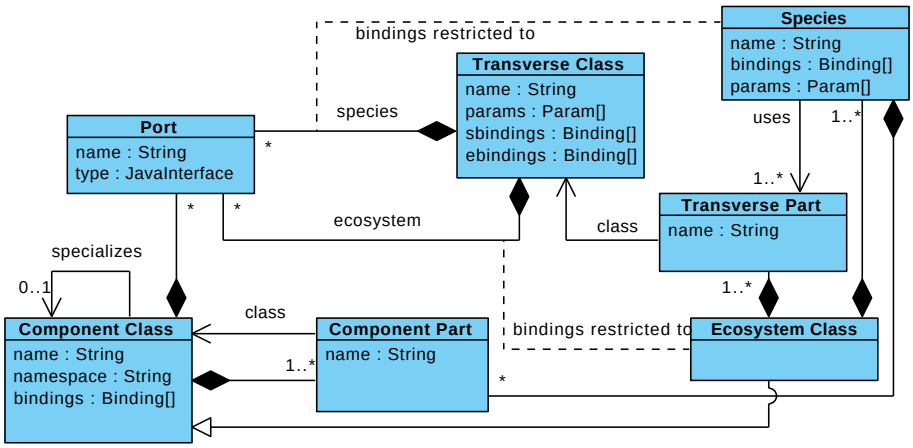

FIG. 1: Méta-modèle de SPEAD (UML2).

3.1 Modèle de composant de base

Pour ce modèle de composants, qui nous sert de base pour introduire ensuite la contribution, nous avons fait un certain nombre de choix. Nous nous sommes inspirés de modèles de composants existants, de nos besoins pratiques et de nos préférences personnelles. Nos objectifs lors de la définition de ce modèle ont été de permettre de facilement décrire des composants, de les implémenter, de les utiliser et de les réutiliser.

Description avec SPEADL. Une classe de composant est décrite par un type de composant, éventuellement accompagné d'une configuration (voir l'élément `Component Class` dans la figure 1 et le mot-clef `component` dans la figure 2a). Le type d'un composant est décrit par un nom ainsi que par des ports qui sont typés par des interfaces [8] et une direction (fournit ou requiert) (voir l'élément `Port` dans la figure 1 et les mots-clefs `provides` et `requires` dans la figure 2a).

La configuration d'un composant décrit la structure interne du composant en termes de parties [9], celles-ci étant réalisées par des composants connectés par des connecteurs (voir l'élément `Component Part` dans la figure 1 et le mot-clef `part` dans la figure 2c). Un composant est donc composite ou primitif selon qu'il est décrit par une configuration ou non.

Chaque partie d'un composite a un nom et un type de composant. Dans une configuration, tous les ports requis des parties doivent être connectés soit à un port fourni d'une autre partie, soit à un port requis ou fourni du composant composite (voir le mot-clef `bind` dans la figure 2c). De plus dans un composite, les ports fournis peuvent être connectés à un des ports fournis d'une de ses parties. Ces connexions doivent respecter les types des ports.

En l'état actuel du modèle, les connecteurs entre composants sont de type simple « appel-retour » et il n'est pas possible de définir de nouveaux connecteurs.

8. Une interface est ici représentée par une interface JAVA.
9. Nous utilisons le terme « partie » de la même façon qu'en UML dans les structures composites.

```
component Sequencer {
  provides globalTick: Tick
  requires tickPerceiveAndDecide: Tick
  requires tickAct: Tick
  requires tickExecutorManager: Tick
}
```

(a) Composant primitif.

```
component UI {
  requires control: SchedulingControl

  provides startRunning: Do
  provides stopRunning: Do
}

component GraphicalUI specializes UI {
  part gui: SchedulingControllerGUI {
    bind control toThis control
  }
}
```

(b) Spécialisation.

```
component ParameterAgentComponent[Ref] {
  provides receive: Push[Request] = rh.receive
  provides perceiveAndDecide: Do = beh.perceiveAndDecide
  provides act: Do = beh.act
  provides getValue: Pull[Double] = actualValue.getValue

  requires send: Send[Request, Ref]
  requires me: Pull[Ref]
  requires die: Do
  requires logActivity: LogActivity

  part actualValue: ValueManager
  part objectiveValue: ValueManager
  part rh: RequestHandler[Ref] {
          bind send toThis send
          bind neighboursInfo to nm.neighboursInfo
  }
  part nm: NeighboursManager
  part beh: Behavior[Ref] {
    // ... connexions comme pour rh ...
  }
}
```

(c) Composant composite.

FIG. 2: Descriptions de composants en SPEADL.

Les types de composants peuvent être paramétrés par des types. Ce n'est pas représenté dans la figure 1, en revanche un exemple est visible dans la figure 2c avec le composant `ParameterAgentComponent` paramétré par `Ref`. Ces paramètres de type peuvent être utilisés dans l'implémentation ainsi que dans les composites en tant qu'arguments des types des parties et des ports.

Enfin, un mécanisme de spécialisation est proposé : seules les descriptions de composants non composites peuvent être spécialisées, et aucun port ne peut être ajouté ou redéfini (voir l'association `specialization` dans la figure 1 et le mot-clef `specializes` dans la figure 2b). En revanche, lors d'une spécialisation introduisant des parties, les ports fournis précédemment définis peuvent être connectés aux ports de ces parties. La seule chose qui peut donc être raffinée est la configuration du composant. Par exemple, cela permet de définir différents composites substituables à un même type de composant.

Implémentation avec JAVA. Ce modèle de composants est ici couplé avec le langage de programmation JAVA. À partir des descriptions de classes de composants précédentes, des classes JAVA qui suivent toutes le même schéma sont générées. Nous avons veillé à ce que l'articulation entre la description et l'implémentation des composants soit la plus flexible possible en évitant toute répétition et vérification manuelle de la cohérence. Pour cela, nous exploitons la génération de code et le typage statique fort du langage cible. Ce code généré agit en quelque sorte comme un conteneur des composants une fois instanciés : il contient du code pour gérer leur démarrage, les connections entre parties de composites, etc. Nous ne détaillons pas ici les choix faits pour ce code généré par manque de place, mais cette information est accessible dans l'implémentation de référence présentée section 3.3.

En particulier, ces classes ont le même nom que le composant, les mêmes paramètres de type et permettent d'implémenter les ports fournis en exploitant les ports requis. `Sequencer` et `ParameterAgentComponent` en sont des exemples respectivement figures 3a et 3c.

Implémenter un composant (primitif ou composite) se fait en étendant sa classe JAVA générée et en se conformant aux règles suivantes :

```java
public class SequencerImpl extends Sequencer {
  // methode abstraite dans la classe mere
  @Override
  protected Tick globalTick() {
    return new Tick() {
      @Override
      public void tick() {
        // utilisation de ports requis
        tickPerceiveAndDecide().tick();
        tickAct().tick();
        tickExecutorManager().tick();
      }
    };
  }
}
```

(a) Composant primitif.

```java
// instanciation de l'implementation GambitsImpl
Gambits.Component sma =
        Gambits.createComponent(new GambitsImpl());
// demarrage du composant
sma.start();
// utilisation d'un des ports
sma.loadScenario().load("scenario1.xml");
```

(b) Instanciation d'un composant.

```java
public class ParameterAgentComponentImpl
    extends ParameterAgentComponent<Ref> {
  // parametre utilise uniquement dans l'implementation
  private final IParameter parameter;
  public ParameterAgentComponentImpl(IParameter parameter) {
          this.parameter = parameter;
  }
  // methodes abstraites dans la classe mere
  @Override
  protected ValueManager make_actualValue() {
          return new ValueManagerImpl(parameter);
  }
  @Override
  protected RequestHandler<Ref> make_rh() {
          return new RequestHandlerImpl(parameter.getId());
  }
  // ... implementation des autres parties ...
}
```

(c) Composant composite.

FIG. 3: Implémentations de composants en JAVA.

- pour chaque port fourni (et non connecté, dans le cas de composites), une implémentation de l'interface du port doit être fournie (voir par exemple la méthode `globalTick` dans la figure 3a) ;
- pour chaque partie d'un composite, une implémentation JAVA conforme à sa classe doit être fournie (voir par exemple les méthodes commençant par `make_` dans la figure 3c) ;
- une implémentation pour un *hook*[10] à exécuter au démarrage du composant peut aussi être fournie.

Une implémentation peut concrétiser les paramètres de type de la classe qu'elle étend.

Dans l'implémentation des ports fournis et du *hook* de démarrage, il est possible d'utiliser les ports requis du composant et le cas échéant les ports fournis des parties du composite (voir par exemple dans la figure 3a).

Exécution. La figure 3b montre un exemple d'instanciation d'un composant en JAVA.

Les instances de composants peuvent être dans un état « initialisé » ou « démarré ». Cet état est géré par le code généré comme suit. « Initialisé » est l'état qui résulte de la création du composant (*i.e.* l'instanciation de son implémentation). Après appel de `start()` sur le composant, celui-ci passe dans l'état « démarré » dès que tous ses ports requis et fournis ont été connectés au sein du composite dans lequel il existe, et que :

- pour un composite, ses parties ont été connectées selon la configuration, et démarrées ;
- son *hook* de démarrage a été exécuté.

Aucun mécanisme n'est proposé directement au niveau du modèle de composants pour arrêter les composants. S'il est nécessaire d'introduire des actions spécifiques pour l'arrêt d'un composant, alors cela doit être exprimé à travers la définition d'un port.

10. Une traduction de ce concept n'existant pas en français, nous avons préféré garder le terme anglais. Implémenter un *hook* permet au développeur d'« accrocher » des morceaux de programmes à exécuter à des moments déterminés.

```
ecosystem Proto {
  provides createParameter: CreateParameter[AgentRef]
  provides createCriteria: CreateCriteria[AgentRef]
  provides createMeasured: CreateMeasured[AgentRef]
  provides createConstraint: CreateConstraint[AgentRef]
  provides loadScenario: Push[IScenario] = sl.loadScenario
  provides updateValue: Push[IMeasured] = su.updateElement

  transverse mo: Monitoring
  transverse em: ExecutionManager {
    bind startRunning to ui.startRunning
    bind stopRunning to ui.stopRunning
    bind logState to mo.logState
  }
  transverse vp: ValuePublisher
  transverse vr: ValueReader{
    bind infraGetValueOf to vp.getValueOf
  }
  // ... declarations des autres parties transverses

  part s: Sequencer {
    bind tickPerceiveAndDecide to pd.tick
    bind tickAct to act.tick
    bind tickExecutorManager to em.testSystemState
  }
  // ... declarations des autres parties composants

  species ParameterAgent(e: IScenarioElement,
                         name: String) {
    part infra : ParameterAgentArchitecture[AgentRef] {
      bind send to m.send
      bind me to m.me
      bind die to s.die
      bind informExecutionManager to em.informChange
    }
    // ... declarations d'autres parties composants
    uses act {
      bind cycle to infra.act
    }
    uses em {
      bind logActivity to mo.logSysActivity
    }
    uses mo(e)
    // ... declarations des autres
    // utilisations de parties transverses ...
  }
  // ... declarations des autres especes ...
}
```

FIG. 4: Description d'un écosystème en SPEADL.

```
transverse Monitoring {
  species(e: IScenarioElement) {
    provides logSysActivity: StateLog
  }
  ecosystem {
    provides logState: StateLog
  }
}
```

FIG. 5: Description d'un transverse en SPEADL.

3.2 Écosystème, espèce et transverse

Nous introduisons maintenant les trois abstractions proposées dans le cadre de ce travail. Elles s'appuient sur le modèle de composants de base présenté précédemment.

L'écosystème a pour principal objectif de réaliser la plateforme d'exécution des agents qu'il contient. L'espèce représente un type d'agent instanciable. Le transverse représente les liens qui existent entre l'espèce et son écosystème. Ainsi les transverses ont deux « côtés » : ils sont présents au sein des écosystèmes et font le pont avec les espèces qui les utilisent.

De la même façon que pour les composants, nous distinguons les classes d'écosystèmes et de transverses, des instances de ces derniers. Les espèces sont des classes, les agents étant leurs instances. Notons aussi que les espèces sont définies au sein d'un écosystème.

Description avec SPEADL. Les classes de transverses sont décrites par un type composé d'un nom et de paramètres de type (voir l'élément `Transverse Class` dans la figure 1 et le mot-clef `transverse` dans la figure 5). À cela s'ajoutent deux ensembles de ports correspondant aux deux « côtés » du transverse : le premier est lié à l'écosystème dans lequel le transverse sera utilisé (voir l'association `ecosystem` dans la figure 1 et le mot-clef `ecosystem` dans la figure 5) et le second à l'espèce qui l'utilisera (voir l'association `species` dans la

```
public class ProtoImpl extends Proto {
  @Override
  protected CreateParameter<AgentRef> createParameter() {
    return new CreateParameter<AgentRef>() {
      public AgentRef create(final IParameter param) {
        // methode generee pour chaque espece
        ParameterAgent agent = createParameter(
          // classes implementant l'espece
          new ParameterAgentImpl(param),
          param, param.getId());
        agent.start();
        return agent.m().me().pull();
      }
    };
  }
  @Override
  protected Sequencer make_s() {
        return new SequencerImpl();
  }
  @Override
  protected Scheduled make_act() {
        return new ScheduledImpl();
  }
}
```

FIG. 6: Implémentation d'un écosystème en JAVA.

```
public class GraphicalMonitoringImpl extends Monitoring {

  // GUI generale
  private GUI g = new GUI();
  // panel pour les infos du systeme
  private GlobalCriticalityViewer globalViewer;

  public GraphicalMonitoringImpl() {
    globalViewer = new GlobalCriticalityViewer();
    g.addChart(globalViewer.getChartPanel());
  }

  @Override
  protected StateLog logState() {
    return new StateLog() {
      @Override
      public void log(Double value, Double criticality) {
        globalViewer.appendValues(value, criticality);
      }
    };
  }

  @Override
  protected Agent make_Agent(IScenarioElement e) {
    return new AgentImpl(e);
  }

  public class AgentImpl extends Agent {
    // panel pour les infos de chaque agent
    private ScenarioElementViewer viewer;

    public AgentImpl(IScenarioElement e) {
      this.viewer = new ScenarioElementViewer(e);
      g.addChart(viewer.getChartPanel());
      viewer.getChartPanel().updateUI();
    }

    @Override
    protected StateLog logSysActivity() {
      return new StateLog() {
        @Override
        public void log(Double value, Double criticality) {
          viewer.appendValues(value, criticality);
        }
      };
    }
  }
}
```

FIG. 7: Implémentation d'un transverse en JAVA.

figure 1 et le mot-clef `species` dans la figure 5). De plus, un ensemble de paramètres (un nom et un type de donnée [11]) est associé au côté espèce .

L'écosystème, qui peut-être utilisé comme un composant ordinaire, est décrit par un type de composant, une configuration et des descriptions d'espèces (voir l'élément `Ecosystem Class` dans la figure 1 et le mot-clef `ecosystem` dans la figure 4). La configuration, en plus de permettre de définir des parties composants, permet de définir des parties transverses (voir l'élément `Transverse Part` dans la figure 1 et le mot-clef `transverse` dans la figure 4). Les parties transverses sont, comme les parties composants, nommées et ont un type. Les connexions respectent les mêmes règles que pour les composites, avec en plus la possibilité de connecter les transverses, mais en se restreignant au côté écosystème de leur type (voir la contrainte `bindings restricted to` liée à `Ecosystem Class` dans la figure 1).

La description d'une espèce, au sein d'un écosystème, est composée d'un nom, d'une configuration et d'un ensemble de paramètres (un nom et un type de donnée) (voir l'élément `Species` dans la figure 1 et le mot-clef `species` dans la figure 4). La configuration d'une espèce est composée de parties composants, et fait référence aux parties transverses de l'écosystème qu'elle utilise (voir l'association `uses` dans la figure 1 et le mot-clef `uses` dans

11. Un type de donnée est ici représenté par une classe JAVA.

la figure 4). De plus, elle définit les connexions entre les transverses — en considérant leur type restreint à leur côté espèce (voir la contrainte `bindings restricted to` liée à `Species` dans la figure 1) — et les parties composants. Pour chacun des transverses référencés, les paramètres de l'espèce peuvent lui être passés en argument s'il le nécessite (voir par exemple le transverse `mo` dans la figure 4).

Implémentation avec JAVA. De la même façon qu'avec le modèle de base, des classes JAVA pour les transverses et les écosystèmes sont générées. Implémenter un transverse se fait alors en étendant sa classe JAVA générée et en suivant les règles suivantes :

- pour chacun des ports fournis du côté écosystème, une implémentation de l'interface du port doit être fournie ;
- pour le côté espèce, une classe doit être définie (voir la classe interne `AgentImpl` dans la figure 7) : pour chacun des ports fournis de ce côté, une implémentation de l'interface du port doit être fournie ;
- pour les deux côtés, une implémentation du *hook* de démarrage peut être fournie ;
- une méthode permettant de créer une instance de son côté espèce doit être implémentée, prenant en paramètre les paramètres de la description (voir la méthode `make_Agent` dans la figure 7).

Dans l'implémentation des ports fournis et du *hook* de démarrage du côté écosystème, seuls les requis du côté écosystème sont accessibles. Dans l'implémentation des ports fournis et du *hook* de démarrage du côté espèce, les requis des deux côtés sont accessibles.

L'implémentation d'un écosystème est similaire à l'implémentation d'un composite (voir la figure 6). Implémenter une espèce signifie définir une classe (qui étend une classe générée) qui fournit une implémentation JAVA pour chacune de ses parties (composants).

Les espèces ne sont décrites et implémentées qu'à travers les transverses qu'elles utilisent, leurs architectures internes définies en terme de composants et leurs paramètres. Ainsi, sans aucune implémentation autre que celles décrites ici, pour chaque espèce, un mécanisme de création d'individu est disponible. Ce mécanisme est accessible pour l'implémentation des ports fournis et du *hook* de démarrage d'un écosystème. Il permet de créer une instance d'une des espèces de l'écosystème en lui donnant une implémentation de l'espèce et des arguments pour les paramètres de l'espèce.

En revanche, pour décrire des modes de création particuliers (choix des implémentations pour le composant réalisant l'architecture interne et choix des paramètres), plus couramment appelés fabriques, et accessibles à l'extérieur de l'écosystème, on pourra définir des ports dans l'écosystème qui implémenteront ces fabriques en utilisant le mécanisme de création d'individu des espèces (voir par exemple le port `createParameter` dans la figure 6).

Exécution. L'écosystème possède les mêmes particularités que les composants composites concernant les états et le démarrage. Dans un écosystème, démarrer un transverse signifie démarrer son côté écosystème et exécuter son *hook* de démarrage.

Les espèces se comportent différemment de par leur nature : en effet, étant similaires à des classes de composants déclarées dans un écosystème, elle ne sont pas démarrées en tant que telles. En revanche, créer un individu de l'espèce entraîne l'instanciation de son implémentation ainsi que le côté espèce de chacun des transverses de l'écosystème que l'espèce utilise.

Ces instanciations sont suivies du démarrage de toutes les transverses, puis de l'architecture interne. Lorsque le côté espèce d'un transverse est démarré, son *hook* de démarrage est exécuté.

Ainsi, tous les mécanismes de connexion entre les agents et la plateforme d'exécution sont gérés par le modèle de composant et le code généré.

3.3 Implémentation et exploitation de SPEADL

SPEADL a été complètement implémenté sous la forme d'un éditeur textuel et d'un générateur de code. Cet outil est nommé MAKE AGENTS YOURSELF (MAY) et est distribué sous licence GNU General Public License (GPL) [12]. L'implémentation repose sur *Eclipse* et le *framework Xtext* [13]. L'éditeur fournit la coloration syntaxique, la complétion automatique, la vérification de type pour les connexions entre composants (types paramétriques inclus). Le code généré avec l'outil ne demande aucune modification et toute implémentation se fait à travers l'extension de classes JAVA comme présenté précédemment. Ainsi la programmation des composants est flexible et incrémentale : la description est considérée comme du code, toute modification est répercutée sur le code généré, et le code implémenté est directement impacté par la mise à jour de ce code généré.

Expérience et utilisateurs. L'outil et le modèle sont utilisés quotidiennement par une demi-douzaine de doctorants au sein de notre équipe de recherche, ainsi que par la *start-up* UPETEC [14], *spin-off* de notre équipe de recherche, dans le cadre de projets industriels. Ils sont de plus utilisés pour introduire la programmation par composant, donc restreinte au modèle de composants de base, à des étudiants de Master. Aucune tentative d'évaluation de l'outil et du modèle n'a été faite en dehors de discussions informelles. Celles-ci montrent un apprentissage du modèle un peu difficile (et en partie causé par la découverte de la programmation orientée composant) mais une appréciation très positive sur le long terme. Les points positifs sont la prise en compte tôt d'exigences non explicitées par la conception SMA, la facilitation de la maintenance et la réutilisation, en particulier celle des mécanismes d'interaction.

Bibliothèque de composants. À travers l'utilisation de l'outil dans notre équipe, une bibliothèque de composants a été constituée. Elle comporte principalement des mécanismes utilisés en pratique dans des projets.

La présence de quelques grands groupes de composants est à noter, par exemple :

- cadencement : un certain nombre de composants permettent de gérer le cadencement du système, certains étant de simples composants (horloges, interfaces graphiques, *thread pools*), certains des transverses (cadencement d'un groupe d'agents, ou depuis l'agent) ;
- interaction : plutôt présents sous la forme de transverses, ces composants permettent de mettre en place l'envoi de messages par référence directe, l'appel synchrone par référence, des mécanismes de *broadcast*, etc ; il existe aussi des composants à utiliser dans les écosystèmes pour la distribution ou la sérialisation de données ;

12. `http://www.irit.fr/MAY`
13. `http://www.eclipse.org/Xtext/`
14. Emergence TEChnologies for Unsolved Problems — `http://www.upetec.fr/`.

– cycle de vie et contrôle des flux : plutôt utilisés dans les architectures internes des agents, ces composants implémentent différents types de cycles de vie, comme le traitement séquentiel de messages, ou des boucles.

Ces composants sont mis à disposition à travers le gestionnaire de projets et de dépendances Maven [15] sous licence GNU Lesser General Public License (LGPL) sur le site web de l'outil.

4 Application

Nous détaillons ici la micro-architecture conçue pour l'exemple décrit dans la partie 2.2.

La figure 8 montre une représentation graphique de celle-ci, où une seule des quatre espèces est détaillée. Le formalisme utilisé est le diagramme composite de UML 2, avec quelques adaptations pour représenter les transverses. Celles-ci sont en deux parties, une dans l'écosystème (stéréotypée «`transverse`»), et une dans les espèces qui l'utilisent (stéréotypée «`uses`»). Les noms des ports ne sont pas indiqués et ceux des interfaces reflètent leur rôle pour simplifier la lecture. Des parties de la description et de l'implémentation de cet exemple sont visibles dans les figures 2 à 7.

Cette micro-architecture a été décrite et implémentée par un des utilisateurs de notre modèle. Les concepteurs ont choisi de définir une espèce par type d'agents. Cela se justifie par le fait que chacune des espèces n'utilise pas les mêmes mécanismes d'interaction (dans le cas contraire, une espèce aurait pu couvrir plusieurs types d'agents de la conception SMA).

La décomposition de l'écosystème résulte de la séparation des préoccupations suivantes :

– interaction avec l'extérieur du système : chargement de scénarios et mise à jour de ceux-ci durant l'exécution du système (`ScenarioLoader` et `ScenarioUpdater`).
– cycle de vie du système, avec la séparation de l'exécution des agents en deux pas (les deux `Scheduled`, chacun étant responsable, lorsque déclenchés l'un après l'autre, d'exécuter l'ensemble des agents du système). L'horloge `Clock` cadence le système, tandis que l'ordre des opérations à exécuter est décrit dans `SysLifecycle`.
– exécution globale du système, récupération de statistiques et arrêt lorsque certaines conditions globales sont atteintes (`ExecutionManager`).
– visualisation du système et contrôle de l'horloge (`Monitoring` et `UI`) : seul `ExecutionManager` dépend d'eux pour leur envoyer des informations de *log* et ces derniers peuvent être facilement remplacés par une version texte par exemple.

L'espèce représentée est connectée à l'écosystème à travers les transverses répondant à des préoccupations opératoires :

– `Scheduled` exécute les agents.
– `ExecutionManager` récupère des informations dans chacun des agents.
– `Monitoring` affiche des informations sur les agents et le système.

Elle est aussi connectée à des transverses répondant à des préoccupations métiers :

– `Messaging` pour l'envoi de messages (par références directes).
– `ValuePublisher/Reader` pour publier/lire l'état d'autres agents (par références nommées).

15. `http://www.maven.org/`

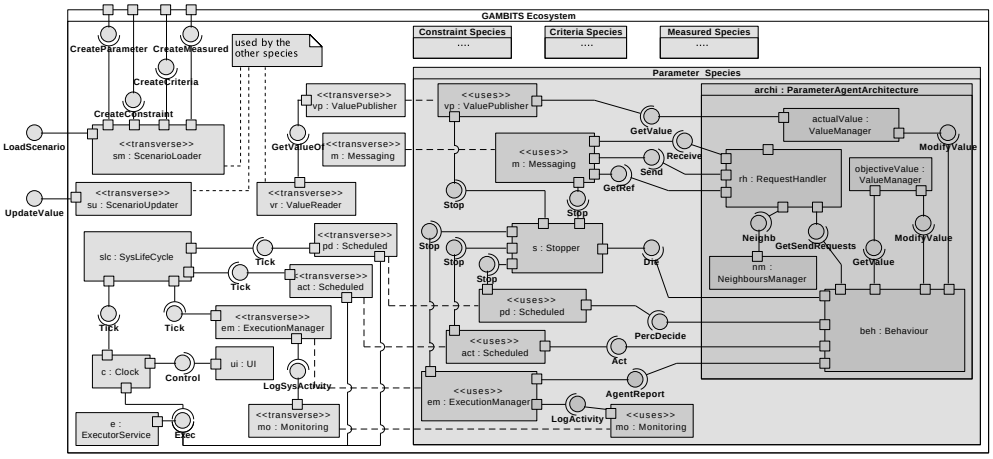

FIG. 8: La micro-architecture du SMA Gambits.

L'architecture interne de l'espèce contient aussi un composant pour stopper tous les transverses qui doivent l'être quand l'agent disparaît du système, en plus des composants réalisant les différentes parties du comportement des agents.

Enfin, la création d'agents est proposée à travers un port par espèce pour s'abstraire de celles-ci et de leur architecture. Ces ports sont connectés à `ScenarioLoader` qui est un transverse : les espèces qui l'utilisent fournissent un moyen d'initialiser certaines informations que les instances doivent connaître à la création du système. Lors de l'initialisation de son côté espèce, sa référence est donnée aux agents précédemment créés, et ensuite, les références de ces agents lui sont communiquées. Ainsi toute la logique d'initialisation est encapsulée dans ce transverse et son implémentation.

Les composants `Scheduled`, `Messaging`, `Clock` et `ExecutorService` ont été pris dans la bibliothèque, et les autres ont été développés pour l'application.

5 Considérations méthodologiques

Dans les sections précédentes, nous nous sommes focalisés sur la dimension architecturale de notre travail. De façon à le resituer dans un contexte méthodologique, nous présentons ici une méthode ainsi que les raisons de son introduction.

Comme nous l'avons vu, une différenciation est faite entre la réalisation de la conception SMA et le support des abstractions de cette dernière. En effet, dans l'exemple et dans beaucoup d'application orientées agent, l'idée est de permettre au développeur SMA de se concentrer sur la définition des comportements de ses agents, et éventuellement de faire de la mise au point à ce niveau sans se préoccuper de la façon dont les mécanismes qu'il utilise sont implémentés.

SPEARAF (Species to Engineer Architectures for Agent Frameworks) est une méthode qui s'appuie sur SPEAD et un processus pour supporter le passage de la conception du SMA, telle qu'envisagée par les méthodes existantes, et son implémentation avec des préoccupations de qualité du logiciel, telles qu'envisagées dans le domaine des architectures logicielles.

L'idée ici est de fournir des espèces d'agents spécifiques qui soient adaptées aux besoins métiers : cela permet aux développeurs de l'application de s'appuyer sur celles-ci autant pour concevoir que pour implémenter le SMA **en exprimant ce que les agents « font »** en faisant abstraction des préoccupations opératoires, c'est-à-dire de **« comment » les agents font ce qu'il font ou sont exécutés**.

Pour cela, nous utilisons une approche par *framework* au sens des *hotspots* et des *frozenspots* (Markiewicz et de Lucena, 2001). L'objectif est de fabriquer pour les agents des architectures dont certains des composants restent abstraits (*i.e.* n'ont pas d'implémentation). Ainsi, les *hotspots* du *framework* basé sur ces architectures seront ces composants destinés à être implémentés par les utilisateurs du *framework* : leurs ports fournis représentent les points à implémenter et les ports requis représentent les primitives de programmation disponibles. Inversement, les développeurs du *framework* fournissent des implémentations pour le reste des composants qui forment les *frozenspots* du *framework*. Cet aspect de notre travail a la particularité de n'avoir au final qu'un seul lien avec SPEAD : celui-ci permet de réaliser les abstractions nécessaires à la conception SMA. Techniquement, les espèces et les transverses n'apportent rien directement à la production de *framework*. En revanche, cela justifie à nouveau l'utilisation de composants (en plus des avantages classiques des architectures logicielles à composants), entre autres par l'utilisation du concept de port requis.

SPEARAF propose un processus en deux étapes correspondant respectivement à la fabrication d'un *framework* par l'expert en architectures et composants, puis à l'utilisation de ce *framework* par l'expert métier (le programmeur du SMA). Ce dernier comble les *hotspots* avec les comportements (métier) des agents.

Voici une ébauche de la façon dont on peut faire correspondre les concepts utilisés par SPEARAF aux concepts de SPEAD :

- l'identification des abstractions nécessaires à l'utilisateur du *framework* ainsi que la définition de la dynamique des agents de l'espèce guide la conception de l'architecture interne de l'espèce et le choix des composants abstraits et concrets.
- l'identification de ce qui compose l'écosystème, que ce soit l'environnement SMA (métier), d'exécution, de déploiement ou l'interface avec l'utilisateur, guide la conception de l'architecture de l'écosystème.
- l'identification de ce qui permet à l'architecture de l'espèce d'interagir avec l'architecture de l'écosystème guide le choix ou la conception des transverses.

6 État de l'art et discussion

Les travaux que nous présentons ici se répartissent en deux catégories. Les trois premiers points concernent des travaux qui peuvent être considérés comme comparables à nos objectifs d'aide au développement des SMA. Ils portent principalement sur l'aspect architectural de la contribution : utilisation de composants, introduction de l'abstraction transverse, création dynamique d'agents. Les deux derniers points concernent des travaux qui nous permettent de situer les concepts introduits ici par rapport au domaine des architectures logicielles. Ils concernent les artefacts que l'on peut construire à l'aide de l'approche et du modèle de composants.

Systèmes multi-agents et composants logiciels Plusieurs travaux existent dans le domaine des SMA qui exploitent des composants pour fabriquer des agents (Briot, 2009). Nous retiendrons Generic Agent Model (GAM) (Brazier et al., 1999), MALEVA (Briot et al., 2007) et Magique (Routier et al., 2001). Ces travaux portent sur l'utilisation de composants pour fabriquer des architectures d'agents. GAM est un modèle d'agent générique fait de composants avec certains mécanismes d'interaction. MALEVA, de son côté, permet de concevoir le comportement de l'agent avec des composants. Magique permet de faire de même en y ajoutant du dynamisme à l'exécution. Ces travaux se focalisent sur l'aspect comportemental de l'agent, proposent un type d'agents (*i.e.* une espèce d'agents) figé et ne considèrent pas la partie plateforme d'exécution du système.

Composants et Connecteurs On peut trouver des ressemblances entre les composants transverses et les connecteurs. Par exemple, Medium (Cariou et al., 2002) propose une manière d'implémenter des connecteurs dans un environnement distribué tout en facilitant la réutilisation. On y retrouve la définition d'un type de composants faisant office de moyen d'interaction, qui dans notre approche est réalisé par le composant transverse complété par l'écosystème. En revanche, nous nous focalisons sur l'intégration d'un composant, l'agent, dans une infrastructure ce qui différencie conceptuellement Medium de notre travail. Dans ces travaux, les connecteurs sont placés au même niveau que les composants qu'ils connectent, alors que dans notre cas, le transverse est un moyen de plonger un composant au sein d'un autre. De plus, l'existence des transverses est justifiée par des besoins de création et de connexion dynamique de composants.

Création dynamique de composants Dans le domaine des ADL, une attention particulière a été donnée à la problématique de la création dynamique de composant. Citons par exemple les travaux faits autour de Darwin (Magee et Kramer, 1996) qui définissent des architectures contenant des composants instanciables à l'exécution. La description contient aussi une définition des liens à mettre en place entre ce composant et les autres composants de l'architecture. Dans le cas de l'instanciation directe de Darwin, il n'est pas possible de référencer les ports fournis des composants créés pour éviter des situations où plusieurs ports fournis sont connectés à un même port requis. Pour pallier cela, il est nécessaire d'utiliser les références de services, ainsi le modèle de composant définit un moyen de « référencer » les composants.

À l'inverse, notre approche propose de placer le composant créé (l'espèce) et son créateur (l'écosystème) à deux niveaux différents : la relation qui les lie est celle de contenu et conteneur. Ainsi le composant est connecté au reste du monde par son conteneur et le choix de transverses définira les moyens de communication qui lui sont donnés. On peut ainsi définir un transverse qui réalise une connexion de type port collection — les ports fournis par les instances créées dynamiquement sont « référencées » par un entier par exemple — ou au contraire utiliser des transverses comme dans les SMA. Cela ne permet pas de vérifier l'évolution de la structure vis-à-vis de l'implémentation des composants, contrairement aux mécanismes d'instanciation dynamique fournis par Darwin que l'on peut coupler avec le π-calculus. En revanche, par construction, il est possible de s'assurer que structurellement parlant, les connexions entre les différents composants sont valides (*i.e.* tout port requis est connecté).

Conteneurs, styles architecturaux et *frameworks* Lorsque l'on utilise le modèle de composant proposé, à travers la définition d'espèces et leur mise en œuvre au sein d'écosystèmes à l'aide de transverses, nous avons avancé que l'on définissait une sorte de machine abstraite capable d'exécuter le comportement des agents. De plus, les précédentes comparaisons font ressortir une relation de type contenu-conteneur entre les instances des espèces et leur écosystème. Avec ce point de vue, à chaque fois qu'une espèce est définie dans une application donnée, nous définissons en quelque sorte un modèle de programmation, voire de composant, qui sera respecté par les instances des espèces. Et par la définition d'une micro-architecture, nous fabriquons un conteneur capable d'exécuter de telles instances : ceci rejoint la définition de *framework* de composant dans Johnson (1997) [16], où d'ailleurs le terme micro-architecture est utilisé pour caractériser l'architecture du *framework* lui-même, par opposition à l'architecture fabriquée avec le *framework*.

On peut pousser l'analogie plus loin et considérer la définition d'un écosystème comme la définition d'un style architectural (ensemble de contraintes sur les éléments, composants et connecteurs, d'une architecture) et l'implémentation de certains de ses éléments. Ainsi dans GAMBITS, nous retrouvons dans la définition des espèces la mise en place du style *peer-to-peer* par envoi de messages. Les contraintes posées sur les composants sont claires, à travers la définition de l'espèce, en revanche, les transverses ne capturent pas de manière satisfaisante les contraintes sur les connecteurs. Plus de travail serait nécessaire sur ce point.

Dans Loiret et al. (2009), les auteurs proposent un moyen de définir des concepts spécifiques à un domaine qui guident la génération et l'implémentation d'un *framework* de composants dédié. Ce travail se différencie du nôtre par le fait qu'il repose sur un modèle de composants qui sert de base à la création du modèle de composants dédié (et donc du *framework* de composants dédié), alors que nous définissons directement un modèle de composants et qu'une architecture l'implémente pour réaliser le *framework* de composants dédié.

Lignes de produits et *frameworks* Comme expliqué dans la partie méthodologique de notre travail, le modèle permet la définition de *frameworks* par l'utilisation de composants sans implémentation. En effet, nous utilisons ce principe pour fournir la micro-architecture du SMA au développeur de l'application. Celui-ci se focalise sur l'implémentation des composants de comportement des agents, et la programmation est contrainte par les ports à fournir et ceux disponibles.

On retrouve l'utilisation de *frameworks* dans les travaux sur les lignes de produits. La principale différence entre l'approche par ligne de produit et celle proposée est liée à l'objectif poursuivi : alors que les lignes de produits visent à créer des architectures servant de base à la fabrication d'application spécialisées, les architectures que nous produisons sont dédiées à une application donnée.

En effet, dans nos travaux, la réutilisation se cantonne surtout aux composants eux-même, et non aux espèces et écosystèmes produits (donc le *framework* produit). Il est pertinent de vouloir réutiliser des morceaux d'architecture pour faciliter la construction de micro-architectures communes à un même domaine. Le besoin existe dans le domaine des SMA, mais le modèle proposé ici ne répond pas à celui-ci.

16. En dehors du fait qu'un *framework* y est considéré comme étant implémenté à l'aide d'un langage de programmation orienté objet.

7 Conclusion

Dans le but de faciliter le passage de la conception à la mise en œuvre, nous proposons une démarche pour construire la micro-architecture d'un Système Multi-Agent (SMA). En effet le développement des SMA a la particularité de nécessiter deux phases de conception distinctes : la conception du SMA et la **conception micro-architecturale**. Notre démarche s'appuie sur le modèle de composants SPEAD qui se démarque des modèles existants par la description de classes de composants, les « **espèces** », instanciables dynamiquement au sein d'autres composants, les « **écosystèmes** ». En particulier, ce mécanisme repose sur le concept de composant « **transverse** » qui permet de connecter les espèces aux écosystèmes. Nous présentons aussi une implémentation de ce modèle sous forme de **langage de description d'architecture** utilisable en conjonction avec JAVA : elle se présente sous la forme d'un éditeur pour le langage et un générateur de code vers JAVA, le tout étant intégré à *Eclipse*. Cette démarche et l'outil sont utilisés dans de nombreux travaux au sein de notre équipe, dans des projets industriels ainsi que dans le cadre d'enseignements de Master.

Nous présentons maintenant une analyse des abstractions introduites dans la contribution au regard des besoins que nous avions identifiés. Cette analyse s'opère selon deux points de vues : architectural et implémentation.

En tant qu'abstraction architecturale, l'espèce est conçue comme une abstraction qui est utile à la définition de la micro-architecture. Sa présence permet de renforcer l'objectif général de la conception architecturale qui est de rendre explicites les choix de conception le plus tôt possible et de documenter l'architecture.

Plusieurs éléments du modèle supportent le volet architectural de notre proposition :

- l'espèce est une abstraction de conception qui explicite l'existence d'entités créées dynamiquement et interagissant de façon spécifique.
- le transverse sert à expliciter le lien entre ces entités et leur plateforme d'exécution. Il est en quelque sorte une garantie de l'autonomie de l'agent : il est important que cette « exigence » (*i.e.* un agent ne peut être contrôlé par une entité extérieure) reste explicite et respectée jusqu'à l'implémentation. Dans l'exemple GAMBITS, les agents sont autonomes, mais certains d'entre eux peuvent être modifiés de l'extérieur, le transverse `ScenarioUpdater` rend ce contrôle extérieur explicite.
- la définition de ports, pour donner accès à la création d'individus d'une espèce en dehors d'un écosystème, permet d'expliciter sur quelles informations repose la création d'un agent, et quelle information est donnée en retour, tout en s'abstrayant du détail de l'espèce et de son architecture interne.

Du point de vue implémentation, on conçoit ces abstractions comme un moyen de faciliter celle-ci, en accord avec les points précédents :

- le transverse, unité d'implémentation réutilisable, est un moyen d'implémenter un lien entre une ou plusieurs entités et l'écosystème. Autrement dit, le transverse est un moyen d'implémenter et de mettre à disposition les ports requis par l'architecture d'une espèce en s'abstrayant de la mise en œuvre de la création dynamique d'individus. Enfin, il permet d'isoler l'implémentation de l'initialisation de la préoccupation qu'il réalise.
- l'espèce est le moyen de décrire comment composer et connecter à l'exécution ces liens et l'architecture interne de l'espèce. À la création, elle permet de composer les éléments de l'architecture de l'agent, transverses inclus, mais de plus elle réalise les différentes initialisations liées à chacune des préoccupations de chacun des transverses.

Ainsi, ces abstractions permettent à la fois de faire respecter les choix faits au niveau architectural, mais aussi réalisent en quelque sorte un patron de conception pour un problème récurrent dans les SMA : la création dynamique d'individus et leur connexion à la plateforme d'exécution.

Pour conclure sur cette analyse, nous pouvons dire qu'il est utile d'utiliser des espèces au sein d'un écosystème pour deux raisons principales :

1. conceptuelle : pour rendre explicite la définition d'un type d'agents ;
2. technique : pour faciliter la réalisation de systèmes où l'on doit mettre en relation un ensemble d'entités interagissant à travers des mécanismes d'interconnexion, et ce de façon dynamique.

Dans certaines applications, il n'est pas techniquement nécessaire d'utiliser ce genre d'abstractions, en particulier si l'application repose sur des *middlewares* adéquats pour interagir. Nous pensons en particulier à l'échange de messages qui est très répandu dans le domaine de l'informatique distribuée. En revanche, nos abstractions s'avèrent utiles par exemple quand :

– les interconnexions agent-plateforme (métiers et opératoires) sont diverses et plus spécifiques au domaine à traiter ; il est alors nécessaire d'introduire de nouveaux mécanismes de type *middleware* et de les composer, et nos abstractions facilitent cela.
– les besoins en documentation des choix architecturaux concernent les liens entre agents et plateforme d'exécution, nos abstractions permettent de les décrire de façon adaptée.

La suite de ce travail se focalisera principalement sur trois axes. Le premier se concentre sur l'éventuelle réification des abstractions transverse et écosystème pour tendre vers un modèle de composant plus général : en considérant le transverse comme une spécialisation de l'écosystème, ce modèle mettra le focus sur les capacités d'instanciation dynamique de composants par la composition d'instances d'espèces (nos transverses actuels) pour former d'autres espèces (nos espèces actuelles). Le deuxième s'appuie sur le fait que notre travail se situe dans le cadre d'une recherche plus générale sur le lien entre composants et agents. Entre autres, nous nous intéressons aux liens qui existent entre la définition d'une espèce d'agents et la définition d'un modèle de composants, d'un style d'architecture ou encore d'un conteneur de composants comme cela a été discuté dans la dernière partie : notre objectif sera de produire des conteneurs pour des architectures à composants auto-organisatrices utilisant une approche SMA comme moteur de l'adaptation. Le troisième, plus méthodologique, porte sur l'utilisation de notre approche pour supporter les lignes de produits logicielles SMA à travers la création de *frameworks*, sur les questions de réutilisation d'architectures que cela pose, comme évoqué dans la partie précédente, ainsi que sur l'étude de son intégration avec les méthodes de développement SMA existantes et leurs modèles d'agents dans un contexte de développement dirigé par les modèles.

Références

Alvarez, J. (2007). *Du jeu vidéo au serious game, approches culturelle, pragmatique et formelle*. Thèse de doctorat, Université de Toulouse.

Bachmann, F., L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, et K. Wallnau (2000). Technical concepts of component-based software engineering. Technical report, Software Engineering Institute, Carnigie Mellon.

Bass, L., P. Clements, et R. Kazman (2003). *Software Architecture in Practice* (2nd ed.). Addison-Wesley.

Bergenti, F., M.-P. Gleizes, et F. Zambonelli (Eds.) (2004). *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*. Klüwer Academic Press.

Bordini, R. H., L. Braubach, M. Dastani, A. El Fallah Seghrouchni, J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, et A. Ricci (2006). A survey of programming languages and platforms for multi-agent systems. *Informatica 30*, 33–44.

Bordini, R. H., M. Dastani, J. Dix, et A. El Fallah Seghrouchni (Eds.) (2005). *Multi-Agent Programming: Languages, Platforms and Applications*, Volume 15 of *Multiagent Systems, Artificial Societies and Simulated Organizations*. Springer.

Bordini, R. H., M. Dastani, J. Dix, et A. El Fallah Seghrouchni (Eds.) (2009). *Multi-Agent Programming: Languages, Tools and Applications*. Springer.

Bosch, J. (2009). From software product lines to software ecosystems. In *Proceedings of the 13th International Software Product Line Conference*, pp. 111–119. Carnegie Mellon University.

Brazier, F. M. T., C. M. Jonker, et J. Treur (1999). Compositional design and reuse of a generic agent model. *Applied Artificial Intelligence Journal 14*, 491–538.

Briot, J.-P. (2009). Composants logiciels et systèmes multi-agents. In A. El Fallah Seghrouchni et J.-P. Briot (Eds.), *Technologies des systèmes multi-agents et applications industrielles*, Chapter 5. Lavoisier.

Briot, J.-P., T. Meurisse, et F. Peschanski (2007). Architectural design of component-based agents: A behavior-based approach. In R. H. Bordini, M. Dastani, J. Dix, et A. El Fallah Seghrouchni (Eds.), *Programming Multi-Agent Systems*, Volume 4411 of *Lecture Notes in Computer Science*, pp. 71–90. Springer.

Cariou, E., A. Beugnard, et J.-M. Jézéquel (2002). An architecture and a process for implementing distributed collaborations. In *Proceedings of the Sixth International Enterprise Distributed Object Computing Conference*, pp. 132–143. IEEE Computer Society.

Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, et J. Stafford (2003). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley.

Demazeau, Y. (1995). From interactions to collective behaviour in agent-based systems. In *Proceedings of the First European conference on cognitive science*, pp. 117–132.

Ferber, J. (1995). *Les systèmes multi-agents : vers une intelligence collective*. InterEditions.

Gleizes, M. P., V. Camps, J.-P. Georgé, et D. Capera (2008). Engineering systems which generate emergent functionalities. In D. Weyns, S. A. Brueckner, et Y. Demazeau (Eds.), *Proceedings of the International Workshop "Engineering Environment-Mediated Multi-Agent Systems"*, Volume 5049 of *Lecture Notes in Computer Science*, pp. 58–75. Springer.

Henderson-Sellers, B. et P. Giorgini (Eds.) (2005). *Agent-Oriented Methodologies*. Idea Group Publishing.

Johnson, R. (1997). Frameworks = (components + patterns). *Communications of the ACM 40*(10), 39–42.

Loiret, F., A. Plsek, P. Merle, L. Seinturier, et M. Malohlava (2009). Constructing domain-specific component frameworks through architecture refinement. In *35th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 375–382.

Magee, J. et J. Kramer (1996). Dynamic structure in software architectures. In *ACM SIGSOFT Software Engineering Notes*, Volume 21, pp. 3–14. ACM.

Markiewicz, M. E. et C. J. P. de Lucena (2001). Object oriented framework development. *Crossroads 7*, 3–9.

Medvidovic, N. et R. Taylor (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering 26*(1), 70–93.

Molesini, A., E. Denti, et A. Omicini (2007). From AOSE methodologies to MAS infrastructures: The SODA case study. In A. Artikis, G. O'Hare, K. Stathis, et G. Vouros (Eds.), *Proceedings of the 8th International Workshop "Engineering Societies in the Agents World"*, pp. 283–298. NCSR "Demokritos".

Noël, V., J.-P. Arcangeli, et M.-P. Gleizes (2010). Between design and implementation of multi-agent systems: A component-based two-step process. In *Proceedings of the 8th European Workshop on Multi-Agent Systems*. Université Paris Descartes.

Routier, J.-C., P. Mathieu, et Y. Secq (2001). Dynamic skills learning: A support to agent evolution. In D. Kudenko et E. Alonso (Eds.), *Proceedings of the Artificial Intelligence and the Simulation of Behaviour, Symposium on Adaptive Agents and Multi-agent systems*.

Schelfthout, K., T. Coninx, A. Helleboogh, T. Holvoet, E. Steegmans, et D. Weyns (2002). Agent implementation patterns. In *Proceedings of the Workshop on Agent-Oriented Methodologies, 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 119–130.

Szyperski, C., D. Gruntz, et S. Murer (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.

Summary

Motivated by the development of Multi-Agent Systems (MASs), we investigate in this paper the production of dedicated agent-oriented development supports by using component-based software architectures. The objective of this work is to ease the transition between the design of the MAS, in terms of agents and interactions, to its implementation, by using what we call the micro-architecture. The last is a mean to take into account the requirements that are not tackled by the agent-oriented design. Using a real world example, we highlight the specificities of MAS applications and how they impact the architecture. The main contribution of this article is in the definition of the component model SPEAD (Species-based Architectural Design) that introduces a specific type of component, the *transverse*, which realises the interconnection between the agents of the system and their runtime platform. This abstraction is accompanied by two others: the *species* and the *ecosystem*. They support at the micro-architecture level the realisation of the concepts that are manipulated in MASs. We present an implementation of this model in the form of an architecture description language, which is usable along with JAVA, that is used in practice to support the development in the research projects of our team.