THESE DE DOCTORAT DE L'UNIVERSITE PARIS VI

présentée pour obtenir le titre de

DOCTEUR DE L'UNIVERSITE PARIS VI

Spécialité Informatique

par

Marc LHUILLIER

Sujet :

Une approche à base de composants logiciels pour la conception d'agents. Principes et mise en œuvre à travers la plate-forme Maleva.

Soutenue le 17 février 1998
devant le jury composé de:

M. Jean-Paul Barthès	Rapporteur
M. Jean-Pierre Briot	Directeur
M. Patrice Carle	Examinateur
M. Jacques Ferber	Co-Directeur
M. Jean-Paul Haton	Président du Jury
Mme Anne Nicolle	Rapporteur
M. Oscar Nierstrasz	Examinateur
M. Jean-François Perrot	Examinateur

TABLE DES MATIERES

INDEX DES FIGURES, SCHEMAS ET EXEMPLES

REMERCIEMENTS

Cette thèse n'aurait pas vue le jour sans le concours de plusieurs personnes que je tiens à remercier pour leur aide, pour leur assistance et pour leur soutien personnel. Il n'y a pas d'ordre particulier dans la lecture de ces remerciements, et je tiens à m'excuser pour ceux que je vais oublier, tant ils sont nombreux.

Je tiens bien évidemment à remercier les membres de ma famille pour leur contribution, directe et indirecte, à la réalisation et à la concrétisation de ce travail.

Ayant changé de directeur de thèse, et sachant combien il a été délicat pour Jean-Pierre Briot d'intégrer mon travail en cours de route, je lui suis particulièrement reconnaissant de son aide, tant sur le plan rédactionnel que sur le plan personnel, tout au long de la rédaction (et même après) de ce mémoire. Je suis reconnaissant à Jacques Ferber pour son aide et l'autonomie qu'il m'a laissée. Je tiens à remercier Jean Charles Pomerol pour la confiance qu'il m'a accordée.

Je tiens à remercier les membres du Jury, Jean-Paul Barthès, Patrice Carle, Jean-Paul Haton, Anne Nicolle, Oscar Nierstrasz et Jean-François Perrot, pour leurs commentaires et remarques sur ce mémoire et pour m'avoir fait le plaisir d'accepter de participer à mon jury.

Je tiens à remercier l'ensemble des membres du laboratoire, à commencer par les doctorants, les docteurs et les secrétaires. Plus particulièrement, mes pensées vont pour les membres de mon équipe, merci Alexis, Claude, Stéphane, Eric, Laurent, Madeleine et Zahia pour les anciens, et pour ceux qui n'ont pas encore terminé, merci Min et Steffen.

Je tiens enfin à remercier mes amis, qu'ils soient du laboratoire ou de l'extérieur, et qui, pendant ces longues années m'ont supporté et réconforté. Merci à Bruno, Eric et Nico, Vincent et Claude, Pierre-Yves, Anne-Gaëlle, Rémi. Un merci particulier à Jean-Max pour son apport gastronomique.

Enfin un merci à tous ceux que j'ai oublié.

I INTRODUCTION

Cette thèse a pour objectif l'étude et l'analyse des modèles de conceptions à base de composants logiciels pour la réalisation de Systèmes Multi-Agents. L'idée centrale de nos travaux consiste en la définition d'un modèle de composant qui permettent une conception incrémentale des agents et de l'environnement. Associé à cette vision de la conception nous désirons assurer une meilleure réutilisation des composants existants pour la conception de nouveaux agents.

La problématique fondamentale, qui découle de la conception d'agents selon une approche incrémentale, est la réutilisation de comportements existants pour la définition de nouveaux agents. De plus, l'ajout de nouveaux comportements à un agent existant s'accompagne le plus souvent de la réécriture de celui-ci. Cette exigence est due en partie à la nature centralisée (de la structure ou du contrôle dans un agent) des modèles d'agents et aussi aux contraintes des langages utilisés pour la conception. Il est par conséquent très difficile, voire impossible de concevoir des agents aux capacités comportementales et morphologiques évolutives.

Pour offrir un modèle de conception qui permette la réutilisation mais aussi l'ajout de nouveaux comportements à un agent, nous proposons un modèle à base de composants logiciels. Ce modèle dissocie complètement la notion de données de la notion de contrôle au sein des agents. Pour présenter nos travaux, nous nous restreindrons volontairement à l'analyse de systèmes multi-agents situés en environnement topologique. Cependant nos travaux peuvent s'appliquer à d'autres catégories d'agents aussi bien qu'à d'autres modèles d'environnements.

En nous basant sur l'analyse de plusieurs architectures d'agents et de différents modèles de conception, nous définissons un modèle de composant qui permet de lier réutilisabilité et flexibilité tout en conservant les qualités respectives des modèles plus classiques tels que les modèles à base d'objets, d'acteurs ou les modèles modulaires. Le modèle que nous proposons s'inspire des modèles à base de composants logiciels (ou modèle componentiel) et permet une décentralisation du contrôle à l'intérieur de l'agent. Cette décentralisation assure alors une meilleure intégration des composants prédéfinis.

Nous proposons dans cette thèse :

1. Un modèle d'agent basé sur un modèle à base de composants. Un agent y est alors vu comme un ensemble de sous composants en interactions. Le problème revient alors à déterminer ce qu'est un composant et comment

il interagit avec les autres. L'approche componentielle[1] impose, dans notre cas, de savoir gérer le contrôle entre les différents composants d'un agent.

2. Un modèle de composant, un modèle de composition fonctionnelle et un modèle de composition structurelle. A la différence des modèles componentiels classiques, nous faisons une distinction très forte entre les flux de données (les informations utiles) et les flux de contrôle (l'activation des composants). L'intérêt de notre approche est, d'une part, de mieux modéliser les compositions et la réutilisabilité des composants, et d'autre part, de permettre une flexibilité compositionnelle nécessaire à tout système évolutif et dynamique.

3. Différentes classes de composants. Des composants 'comportementaux' pour réaliser un agent et des composants de contrôle/données pour gérer les interactions entre les composants comportementaux. En partant d'un schéma de conception basé sur le triplet perception-délibération-action, nous définissons des composants généraux tels que des capteurs, des composants comportementaux de base tels que la fuite, le suivi, et finalement des composants d'action. Du fait de la distinction forte que nous faisons entre les flots de données et les flots de contrôle entre les composants, nous proposons des composants de gestion du contrôle tels que des composants pour la synchronisation, la diffusion... et des composants de gestion des données qui sont la réplique des composants de contrôle mais qui ne traitent que des données, comme par exemple un composant pour la fusion de plusieurs flots de données en un seul.

Pour mettre en œuvre notre modèle componentiel, nous avons réalisé une plate-forme nommée Maleva (Modular Architecture for Living and Evoluting Agents). La plate-forme Maleva est avant tout une plate-forme de conception mais, et ce afin de tester les agents qui y sont réalisés, nous y avons intégré des outils de simulation. Maleva propose donc:

- Un environnement de programmation classique (Smalltalk, Delphi...) lié à un environnement de conception visuelle simple permettant de créer, à partir du modèle componentiel, différents types d'agents. Nous entendons par différents types d'agent, des agents aux structures différentes (le type et le nombre des composants) et des agents aux architectures différentes (l'agencement de ces composants et la gestion du contrôle).

- Un environnement pour la simulation de Système Multi-Agent, c'est-à-dire un ensemble d'outils destinés à la gestion de la simulation et des résultats de celle-ci, et bien sûr, un environnement de simulation censé être une réification d'un environnement réel ou imaginaire.

[1] Le terme 'Componentiel' doit être entendu comme faisant référence aux Composants Logiciels.

- La possibilité d'intégrer des outils ou des applications existantes pour faciliter la conception d'une nouvelle plate-forme. Maleva a été conçue comme un ensemble de classes, objets, composants et outils qui permettent de réécrire Maleva suivant la catégorie d'applications à laquelle elle sera destinée.

Ce mémoire sera découpé en deux parties :

1/ La première partie correspond à l'étude du domaine et des problèmes liés à la conception des Systèmes Multi-Agents. Nous aborderons essentiellement les différentes architectures d'agents et en particulier les architectures à base d'objets, d'acteurs et les architectures modulaires. Nous en profiterons pour traiter des différents modèles de communications inter-modules avec leurs défauts respectifs.

Nous verrons que les langages de programmation classiques tels que les langages objets et les langages d'acteurs ne sont pas suffisants, non pas parce qu'ils ne sont pas assez puissants mais parce qu'ils ne modélisent pas les interactions mais uniquement les structures ou les messages. Il faut, si l'on veut obtenir un système efficace, modéliser plus finement les interactions entre les entités[2] définissant les agents.

Il est primordial si l'on veut concevoir facilement des agents de bénéficier d'outils de conception simples, efficaces et adaptés aux problèmes engendrés par la simulation Multi-Agents tels que l'adaptation, l'évolution, le parallélisme. Les outils que proposent Maleva sont de plusieurs niveaux, ils vont de l'utilisation d'un langage orienté objet et/ou composant pour la conception de composants de base à la simulation proprement dite en passant par la « programmation » d'agents par composition visuelle de composants préexistants.

2/ Dans une seconde partie nous traiterons des réalisations, des choix et des outils proposés dans la plate-forme Maleva. Maleva a été conçue dans plusieurs environnements afin de tirer parti de chacune de leurs particularités. Cette thèse s'appuiera principalement sur la réalisation de Maleva faite dans le langage Delphi. Le choix de Delphi comme exemple de conception a été voulu dans la mesure où le langage lui-même met déjà en pratique la notion de composant logiciel. Ceci nous permettra alors, de voir certaines des différences qui peuvent exister entre des approches à base de composants pour la conception d'interface utilisateur, et des approches orientées conception de SMA.

Nous sommes partis de l'étude de plusieurs plates-formes de simulation pour la plupart réalisées au sein de l'équipe Miriad[3]. Cette étude nous a montré

---

[2] Entité fait référence aux objets, aux acteurs ou aux modules.

[3] L'équipe Miriad (Méthodologie des Interactions et Recherche en Intelligence Artificielle Distribuée) a disparu en tant qu'entité pour se fusionner avec d'autres équipes de recherche au sein d'un thème nommé OASIS (Objets et Agents pour Systèmes d'Information et de Simulation).

que le problème principal (outre le fait que les architectures d'agent sont toutes différentes d'une plate-forme à l'autre) était la réutilisabilité et l'ouverture vers d'autres applications.

Maleva tire pleinement partie du modèle componentiel dans la mesure où la plate-forme est elle-même conçue sur ce modèle. Maleva a donc été conçue comme une application basée autour d'une architecture modulaire et avant tout comme un système ouvert. Nous entendons par système ouvert un système qui admet des composants (applications, bibliothèques de fonctions...) écrits dans différents langages. Par système modulaire nous voulons un système qui puisse être développé de manière incrémentale par ajouts successifs de composants à l'application de base. L'idée maîtresse devient alors la réutilisabilité des composants existants, associée à la flexibilité de la composition nécessaire aux architectures ouvertes.

Cette thèse s'inspire donc naturellement des travaux sur les Systèmes Multi-Agents appliqués à la simulation, et des travaux entrepris sur des plates-formes de conception de SMA. Nous appliquons des techniques empruntées au Génie Logiciel notamment sur quelques architectures Client/Serveur et systèmes temps réels, et nous verrons l'utilisation de langages de programmation visuelle de type L4G. Bien entendu cette thèse s'inspire fortement des travaux sur les modèles de conception à base de composants logiciels dont nous étudierons les principes.

Partie 1

Un modèle à base de composants pour la conception d'agents

II ÉTAT DE L'ART ET PROBLEMATIQUE

La problématique principale de nos travaux se situe à la frontière de l'Intelligence Artificielle Distribuée et du Génie Logiciel et nous pourrions la résumer ainsi :

Est-il possible de concevoir des 'objets'[4] *représentant des comportements et/ou des structures primitives qui, une fois associés entre eux, permettront de définir un agent ? Est-il alors possible de réutiliser ces objets afin de concevoir de nouveaux agents ? Pouvons nous prendre en compte les notions d'évolution comportementale et structurelle ?*

Pour répondre à ces questions, nous devrons définir ce que sont ces 'objets' et nous devrons définir le terme 'associer'. Nous verrons, par la suite, que nous pourrons remplacer le terme 'objets' par le terme *composant* et le terme 'associer' par *composer*.

Maleva propose de définir une architecture d'agent comme un ensemble de composants en interaction, chaque composant étant lui-même défini comme un ensemble de sous composants. L'architecture proposée permet une approche incrémentale de la conception d'agent et de Systèmes Multi-Agents et l'obtention d'un modèle d'agent flexible, dynamique et distribué. Nous nous restreindrons, dans notre modèle, à la conception d'agents situés en environnement topologique. Le même modèle à base de composants est utilisé pour la conception de la plate-forme ce qui permet de modifier l'apparence et le fonctionnement du système en fonction des besoins de la simulation et de l'utilisateur.

Nous allons, dans un premier temps, aborder quelques architectures classiques d'agents telles que les architectures verticales, horizontales, hybrides et modulaires. Dans un deuxième temps, nous aborderons la notion de composant logiciel.

II.1 UN AGENT

L'objectif de ce chapitre n'est pas de définir ce qu'est un agent en terme de capacité comportementale mais de donner à la fois une définition fonctionnelle et une définition structurelle de celui-ci dans une approche de réalisation de SMA.

Nous étudierons dans un premier temps quelques définitions (non exhaustives) de la notion d'agent pour aborder par la suite les propriétés que nous estimons nécessaires de fournir à un agent.

[4] Au sens générique du terme.

II.1.1 PRINCIPES

S'il existe beaucoup de définitions d'un agent dans la littérature, elles sont soit très générales soit très spécifiques. Nous allons tout de même essayer d'en extraire certains principes.

Y. Demazeau et J-P. Muller [Demazeau 90] [Demazeau 91]définissent un agent comme une entité qui agit de manière rationnelle par rapport à ses buts et à ses croyances.

Y. Shoham [Shoham93] les définit comme des entités agissant dans un monde (leur environnement) qui peut être partiellement connu.

J. Ferber [Ferber95], quand à lui, les caractérise comme étant une entité réelle ou abstraite, dont le comportement est défini en fonction de la perception de l'agent, de ses variables d'état interne et de ses interactions avec les autre agents et avec l'environnement. Il ajoute qu'un agent peut se reproduire.

D'autre descriptions [Moffat95] ajoutent à un agent la flexibilité. Bien que peu étudiée, la flexibilité est un des aspects les plus important du modèle d'agent dans le cas de simulation faisant intervenir des modifications (adaptation, apprentissage..) comportementales et morphologiques.

Les mécanismes d'adaptation courants font intervenir des modifications au niveau des modules de contrôle, par exemple dans une architecture de subsomption, les fonctions de sélection (poids, priorités) qui font partie du mécanisme de contrôle peuvent être modifiées afin d'adapter le comportement de l'agent aux contraintes environnementales ou à son but.

Ces modifications n'entraînent pas, en général, de modification structurelle de l'agent. Ce pourrait pourtant être le cas, par exemple la focalisation de la perception d'un agent [Maës 91] doit être accompagnée d'une modification structurelle et pas seulement d'un filtrage plus spécifique des données perçues [Cariani 91].

Dans le cas de la perception, la focalisation peut se résoudre de plusieurs manière suivant la structure initiale de l'agent. Si l'agent ne possède à l'origine qu'un seul capteur « polyvalent », les modèles à base de composant permettent de le remplacer par un capteur plus spécifique alors que les modèles classiques sont obligés d'opérer un filtrage plus précis. Dans le cas d'un agent possédant plusieurs capteurs (par exemple dans la plate-forme ALIA[5], les robots possèdent un capteur de minerai, un capteur de balise, un capteur de base...), le concepteur est amené à faire un choix en fonction de l'architecture de l'agent. Les modèles classiques doivent soit inhiber les autres capteurs (architecture de

[5] La plate-forme ALIA, qui a été conçue par M. Delaye, a pour but l'étude de l'évolution structurelle et comportementale d'une société de robot en environnement inconnu. Nous reviendrons à plusieurs reprises sur cette plate-forme.

subsomption), soit accorder plus de temps de calcul au capteur concerné et moins de temps aux autres (architecture temps réel).

Dans les modèles componentiels les autres capteurs sont temporairement ôtés de l'agent, cette approche outre sa simplicité d'emploi, permet d'accorder automatiquement plus de temps de calcul au capteur restant ce qui dans le cas de systèmes temps réel est un confort supplémentaire.

II.1.2 PROPRIETES

Définir un agent est quelque chose de difficile car un agent ne peut pas être pris indépendamment, ni des autres agents du système, ni de son environnement, ni même de l'application pour laquelle il a été conçu. C'est parce qu'il est difficile de définir un agent en terme informatique qu'il apparaît plus accommodant de le définir en termes de capacité ou de propriétés. Les propriétés que nous allons évoquer et qui reprennent les définitions précédentes permettront de mieux appréhender les problèmes de conception d'un agent et a fortiori les problèmes de conception d'un SMA pour la simulation.

II.1.2.1 AUTONOMIE COMPORTEMENTALE

L'autonomie comportementale d'un agent peut être définie comme la capacité propre de pouvoir atteindre son but. Cette capacité doit être indépendante des autres agents du système.

II.1.2.2 FLEXIBILITE

La flexibilité est la propriété pour un agent de pouvoir modifier dynamiquement son comportement et/ou sa morphologie. Cette modification peut être un changement, une perte ou une augmentation. Dans la plate-forme ALIA [Delaye93], les agents robots pouvaient perdre une chenille et continuer à fonctionner (informatiquement parlant) tant bien que mal. Le comportement étant censé s'adapter à la structure du robot, toute modification de sa structure physique entraîne une modification de son comportement.

Un autre exemple courant de la flexibilité est celui des réseaux de communications, un nœud du réseau peut être amené à s'interrompre sans remettre en cause le fonctionnement global du système.

La propriété de flexibilité est très importante si l'on considère que les agents que nous concevons ne sont pas forcément (initialement) adaptés à l'environnement dans lesquels nous les faisons évoluer. La flexibilité va alors permettre une adaptation et une meilleure adéquation Agent/Environnement.

II.1.2.3 ADAPTATIVITE

L'adaptation est une capacité non nécessaire mais importante dans les SMA. Elle caractérise les systèmes où les agents cherchent à « améliorer » leur fonctionnement individuel ou collectif et, par la même, améliorer le système global. Par exemple, dans un système où les ressources accessibles sont primordiales à la survie des agents, s'adapter à une modification de la quantité ou à la nature des ressources peut être vu comme une forme d'adaptation.

Dans la plate-forme ALIA l'adaptation comportementale d'un robot, qui se situe avant ou après une modification structurelle, permet d'optimiser les fonction de déplacements ou de traitement d'une ressource.

La plate-forme Manta conçue par A. Drogoul, permet la simulation de comportements sociaux d'une société de fourmis [Drogoul93A]. L'apparition de sociogénèse correspond au passage de la totipotence à la spécialisation des fourmis. Cette adaptation avec retour possible (la spécialisation, dans ce cas, n'interdit pas de revenir à un comportement « oublié ») permet à l'ensemble de la colonie de survivre.

II.1.2.4 REACTIVITE

La réactivité doit être vue comme la particularité pour un agent de répondre directement à des stimuli, c'est à dire sans faire appel à la composante de délibération sur le comportement à adopter.

Eviter un trou (en s'arrêtant par exemple) ou fuir un prédateur peuvent être des comportements « câblés » ne prenant pas en compte les données internes de l'agent dans son processus de décision. Eviter un trou tout en fuyant un agent devient un problème plus complexe faisant intervenir des priorités dans l'acte de fuite ou d'évitement.

Faire coexister comportements délibératifs et comportements réactifs est une chose très complexe. Les travaux de Z. Guessoum dans sa thèse montrent qu'il est possible d'obtenir des architectures hybrides temps réel. Le problème à gérer est un problème de contrôle et nous verrons que la programmation componentielle permet d'obtenir des architectures hybrides sans système centralisé ou sans faire utilisation de mécanismes de priorité tels qu'ils sont utilisé par Brooks [Brooks 86].

II.1.2.5 MODULARITE

Nous avons volontairement ajouté la modularité comme une propriété à un agent car elle découle naturellement de la propriété de flexibilité, et elle permet de ne pas contraindre la structure de l'agent.

L'agent est alors conçu comme une « association » de modules plus ou moins spécialisés qui fonctionnent en parallèle et qui interagissent. Les interactions entre ces modules permettront d'obtenir un comportement global cohérent.

L'approche modulaire permet la flexibilité du système car elle impose des mécanismes de communication entre les modules qui par définition intègrent l'ajout, la suppression ou la modification d'un module. Un autre atout de la modularité est de pouvoir faire coexister comportements réactifs et délibératifs au sein d'un même agent. Les agents anticipateurs de P. Davidsson eux même inspirés des agents anticipateurs de E. Astor et B. Ekdahl [Davidsson94], [Davidsson95] ou l'architecture d'agents hétérogènes de Z. Guessoum en sont deux exemples récents.

En principe, un agent devrait idéalement posséder les propriétés énoncées précédemment. Dans la pratique il est rare de voir coexister l'ensemble de ces propriétés au sein d'un même agent, même si certains travaux [Delaye, Guessoum] ont tenté de le faire. Même modulaires, les modèles utilisés pour la conception d'agent ne sont pas, ou très peu, flexibles. Cela provient d'une part de la nature centralisée ou répartie du contrôle dans l'agent et d'autre part de la faible répartition du système.

Le modèle componentiel permet de résoudre le problème de la modularité et celui de la répartition, mais il ne résout pas celui du contrôle. Néanmoins ce modèle apparaît comme étant le plus adéquat pour définir un nouveau modèle. Nous verrons dans le chapitre suivant qu'il est possible de gérer une partie du problème du contrôle sans remettre en cause l'ensemble du modèle.

II.2 LES ARCHITECTURES

Si nous devions caractériser les différentes architectures d'agents, nous pourrions les classer dans deux grands groupes, les architectures verticales, dont la plus connue est l'architecture définie par Brooks, et les architectures horizontales, avec bien sûr tout un panel d'architectures hétérogènes qui intègrent à la fois un modèle horizontal et vertical.

II.2.1 ARCHITECTURES VERTICALES

Les architectures verticales dont la plus connue est l'architecture des agents de R. Brooks [Brooks86], caractérisent des agents modélisés en terme de fonctions de transition entre les données perçues et les actions exécutées. Un principe de décomposition verticale revient à associer à chaque action une fonction.

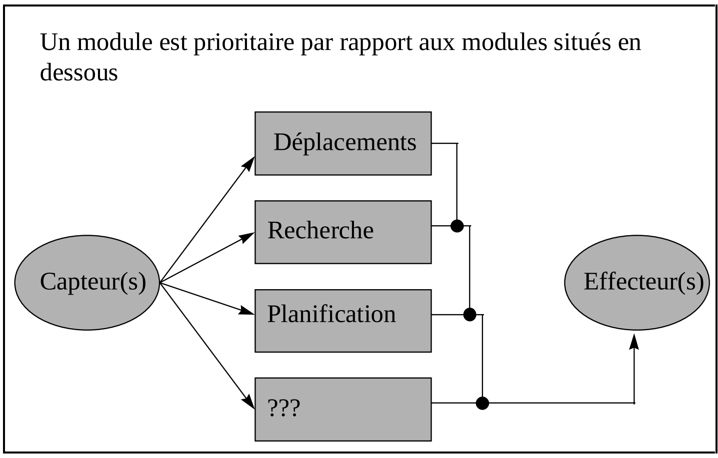


Figure 1 : Une Architecture Modulaire Verticale

Brooks a développé une architecture dite de subsomption. Cette architecture qui est adaptée aux agents réactifs (c'est à dire, dont la fonction de transition est simple) offre l'avantage d'une conception simple, car ne faisant intervenir que les 3 composantes : Perception - Sélection - Action. Elle est basée sur un mécanisme de priorité entre les fonctions de transition (partie Sélection). Cette architecture est un système modulaire (on peut regrouper certaines fonctions de transition en un groupe correspondant aux mêmes actions) vertical où les modules placés au-dessus sont prioritaires par rapport aux modules situés en dessous. Les modules peuvent fonctionner en parallèle et seul le module de priorité la plus forte verra son résultat appliqué.

Le défaut principal d'une telle architecture est qu'elle impose de définir les niveaux de priorité pour chaque module et que ces priorités semblent figées une fois pour toute.

Selon le même principe, il existe également la plate-forme MANTA [Drogoul93A] qui est caractérisée par une architecture verticale avec des tâches compétitives mais qui procurent une certaine flexibilité dans les relations de priorité entre les tâches. Cette architecture est identique à celle de Brooks à ceci près que la sélection d'une tâche à activer peut être définie dans un module indépendant de la tâche elle même.

II.2.2 ARCHITECTURES HORIZONTALES

Les architectures horizontales sont, quant à elles, utilisées pour la conception d'agents plus complexes. Une décomposition horizontale, à l'inverse d'une décomposition verticale, ne décompose pas l'agent en terme d'actions. Il s'agit en fait d'un découpage fonctionnel d'un agent en plusieurs modules distincts interagissants. Parmi ces architectures, nous pouvons citer les architectures à base de tableaux noirs, un tableau noir étant une zone de gestion et de maintien des données provenant des différentes composantes de l'agent KS (Knowledge Source) ainsi qu'une zone de gestion du contrôle de ces composantes [Cambier94] [Iffenecker92].

II.2.3 ARCHITECTURES HYBRIDES

Entre les deux approches, verticales et horizontales, il existe tout un éventail d'architectures hybrides qui font intervenir un modèle modulaire avec une approche centralisée ou décentralisée de la gestion des composants d'un agent. L'agent est alors conçu comme une « association » de modules plus ou moins spécialisés qui fonctionnent en parallèle et qui interagissent. Les interactions entre ces modules permettent d'obtenir un comportement global cohérent.

La plate-forme Dima développée par Z. Guessoum [Guessoum96] correspond à l'approche centralisée, elle est caractérisée par l'utilisation d'un module central chargé de la gestion des autres modules de l'agent. Ce module est aussi chargé de la gestion des priorités entre les modules, de la répartition des messages et de l'acheminement des messages. Cette approche permet de déterminer dynamiquement des priorités entre les modules et donc de combiner des composantes réactives avec des composantes cognitives.

Les agents anticipateurs sont un autre exemple d'architecture hybride simple. Ils peuvent être conçus comme l'ajout d'une composante d'anticipation à un agent réactif. Les travaux de P. Davidsson [Davidsson95] montrent comment faire cet ajout. L'agent réactif est défini par une composante réactive (Reactor), des senseurs (Sensors) et des effecteurs (Effectors). Concevoir un agent anticipateur revient à ajouter deux composantes, une composante d'anticipation (Anticipator) ainsi qu'un modèle du monde (World Model).

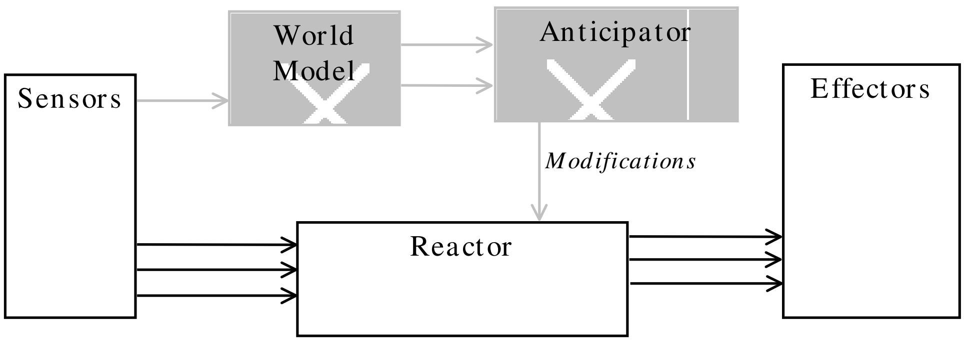


Figure 2: [Davidsson 95] The basic architecture of an anticipatory agent

Sur cette figure, nous voyons un agent Anticipateur. En noir, nous avons l'agent *réactif* initial, en gris nous avons fait figurer les deux composantes qui ont été ajoutées.

Le but de l'agent, tel qu'il est introduit dans l'article de M. Davidsson est de se déplacer vers un point donné (une base). L'agent perçoit toujours la base. Le rôle de la composante d'anticipation est, dans ce cas particulier, d'éviter les boucles d'aller-retour qui sont une caractéristique des agents réactifs en présence d'obstacles.

L'agent *réactif* fonctionne comme ceci: il perçoit la base, détermine l'orientation qu'il doit avoir (l'orientation de l'agent est définie comme étant, Nord, Est, Sud ou Ouest) afin de minimiser sa distance par rapport au but et enfin il avance.

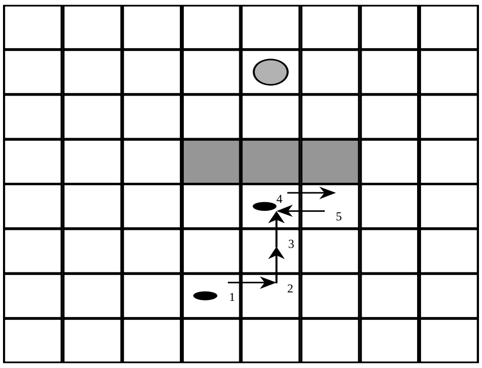


Figure 3: Le Comportement d'aller-retour

En absence d'obstacle, l'agent atteint toujours son but. En présence d'un obstacle (ici un mur) il se bloque en effectuant une succession d'aller-retour. Sur la figure ci-dessus, nous avons fait figurer la base (cercle gris), le mur (carrés grisés) et l'agent (ovale noir). Les chiffres indiquent les opérations de déplacements.

Les étapes 1,2 et 3 correspondent au comportement de se rapprocher du but (sans obstacle). L'étape 4 est un choix de se déplacer vers la droite (le mur interdit d'aller au Nord). L'étape 5 correspond alors à choisir la direction qui permet de se rapprocher du but, la seule direction possible (puisque le Nord n'est pas possible) revient à aller à gauche. L'agent revient donc à sa position antérieure. A partir de cette position, l'agent se bloque par une succession d'aller-retour.

Afin d'éviter ce blocage, il est nécessaire d'introduire une composante d'anticipation. Le rôle de cette composante est d'empêcher l'agent de revenir sur ses pas et, plus généralement, de détecter et de traiter les boucles infinies.

Pour fonctionner, un agent Anticipateur doit posséder un modèle du monde[6] (ou modèle d'environnement). Ce modèle du monde est une copie exacte

[6] Le monde d'un agent représente l'environnement topologique et l'ensemble des agents, données et objets qu'il contient.

du monde dans lequel évolue l'agent. La différence principale qui existe entre le modèle et le monde, est que le modèle doit « fonctionner » plus rapidement que l'original pour permettre de « tester » les conséquences des actions de l'agent.

Dans l'exemple ci-dessus, la composante d'anticipation va exécuter les étapes 4 et 5 avant même que l'agent ne soit arrivé devant le mur. A partir de là, la composante va détecter la boucle et modifier en conséquence la composante réactive.

Cet exemple est caractéristique des difficultés liées à la conception d'agents. En effet, sur la Figure 2, on s'aperçoit que pour passer d'un agent réactif à un agent Anticipateur, nous avons besoin d'une d'entrée *modifications*. Le rôle de cette entrée est de permettre à la composante *Anticipator* de modifier le choix de la direction avant que l'effecteur n'agisse. Pour cela, la composante *Anticipator* va changer certains paramètres internes de la composante *Reactor*, ces changements vont entraîner des modifications dans le choix des actions. *Le problème est alors de savoir ce que nous aurions fait si la composante Reactor n'avait pas été conçue avec une entrée modifications ?*

Toujours dans une approche modulaire, il existe des modèles où le module est lui-même un agent. Par exemple le modèle d'agent LRO3 [Roche93] identifie un agent à une société de sous-agents en interactions. Un agent est alors décomposé en deux niveaux, le niveau de compétence et le méta-niveau (voir la figure ci-dessous). Chacun de ces niveau est lui-même décomposé en plusieurs agents.

Au méta-niveau, nous avons un agent interface et un agent de contrôle. L'agent interface est chargé de la réception des messages, l'agent de contrôle gère les agents du niveau de compétence. Le niveau de compétence est composé d'un agent en charge des requêtes (équivalent d'un capteur), d'un agent en charge des réponses (équivalent d'un effecteur) et enfin un agent en charge des réclamations c'est à dire spécialisé dans le traitement des messages que d'autres agents n'ont pu traiter (partie comportementale).

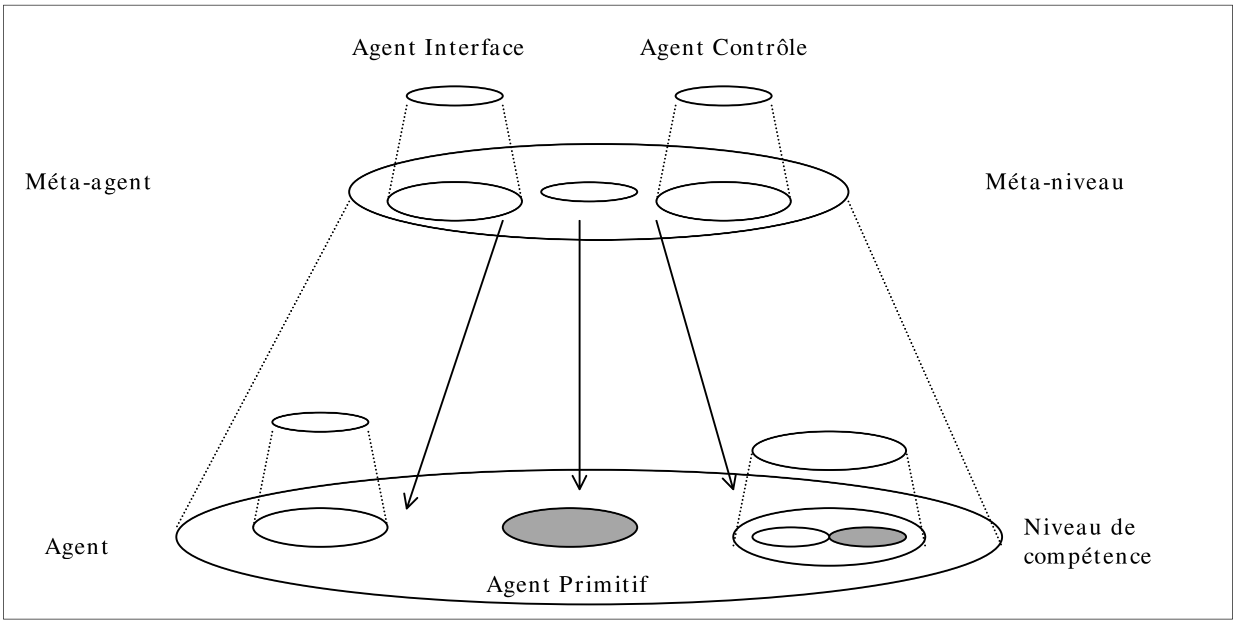


Figure 4: Structure réflexive et récursive d'un agent LRO3

Cette figure représente un agent LRO3 avec ses deux niveaux, cette approche est intéressante et nous y reviendrons dans le paragraphe III.1.3

II.2.4 REPARTITION FAIBLE/REPARTITION FORTE

Comme nous pouvons le voir autour des quelques exemples précédents, l'approche modulaire de la conception d'agent semble être la solution la plus adaptée car elle autorise différentes architectures. Si cette approche permet une grande souplesse de conception, elle impose cependant de définir les interactions entre les différentes composantes de l'agent. Une composante peut être un module [Maës 93, Iffenecker92, Guessoum96], une cellule [Delaye95, Konstantas93], un composant logiciel [Yoo97] et, bien sûr, dans cette thèse un composant Maleva. Les interactions entre les composantes sont assurées par des mécanismes de communications. Les communications sont effectuées par des envois de messages.

Dans une approche distribuée, les différentes composantes d'un agent peuvent fonctionner comme des processus indépendants. Dans une approche répartie, elles peuvent être localisées sur des machines différentes. Dans un cas comme dans l'autre, les interactions (donc les envois de messages entre les composantes) déterminent la nature faiblement répartie ou au contraire fortement répartie du système [Konstantas 93].

Les systèmes faiblement répartis sont les systèmes où les modules communiquent avec les modules dont ils connaissent les noms. Konstantas va même jusqu'à dire que les noms des modules sont connus dans tout le système.

Les systèmes fortement répartis stipulent que les noms des modules à qui on s'adresse sont des données volatiles qui ne sont pas connues par le module qui effectue la requête. Dans son modèle Cell, D. Konstantas impose pour un module de ne connaître que les noms de ses « ports de communication ».

Pourquoi le fait de ne pas connaître le nom des modules à qui on s'adresse est-il important ? Connaître le nom de son interlocuteur impose de connaître l'ensemble des interlocuteurs possibles et de savoir comment gérer leur absence (que celle-ci soit temporaire ou définitive). Si cela apparaît relativement facile à gérer il n'en va pas de même de la gestion de l'apparition de nouveaux modules dont on ne connaît ni le nom ni le rôle.

Si nous reprenons l'exemple précédent , l'agent Anticipateur, nous voyons que la composante *Sensor* transmet des données à la composante *World Model*. Cette transmission n'est possible que dans le cas où la composante *Sensor* connaît la composante *World Model*, ce qui revient à dire que l'architecture initiale de l'agent a été conçue afin de prendre en compte ces nouvelles composantes.

Nous avons opté, dans Maleva pour une approche fortement répartie. Les composantes de l'agent ne connaissent pas les autres composantes. Cette approche doit permettre une totale flexibilité des composantes de l'agent. N'importe quel composant peut dès lors être ôté ou modifié sans remettre en cause ni les composants existants ni les interactions entre ces composants.

Pour nous résumer, un composant destiné à la conception d'agent doit posséder des caractéristiques telles que :

- Une interface indépendante du comportement interne, afin d'assurer une totale dissociation entre comportement interne et gestion des messages inter-composant.

- Une activité indépendante des autres composants, afin d'assurer une certaine autonomie au composant.

- La possibilité de se composer avec d'autres composants afin de permettre de concevoir de nouveaux composants de plus haut niveau.

Ces différentes caractéristiques, qui ne sont pas exhaustives, seront étudiées dans le chapitre suivant.

III Modele Componentiel

Il est très difficile, sinon impossible, de décrire l'ensemble des modèles de conception de Systèmes Multi-Agents tant ceux-ci sont variés. Cependant et compte tenu que la plate-forme Maleva, et la majorité des plates-formes de conception/simulation développées au sein de l'équipe RTCD - Miriad, utilisent des architectures Objets, Acteurs ou Modulaires, nous nous emploierons ici à décrire ces modèles de conception.

Nous verrons que, malgré de grandes capacités de modélisation, les modèles objets et acteurs ne sont pas assez puissants pour permettre la conception de SMA où les agents soient flexibles et évolutifs.

Les systèmes modulaires, même s'ils apportent la flexibilité nécessaire à la conception d'agents évolutifs, introduisent des contraintes (par exemple un mécanisme de contrôle centralisé ou une interface de communication homogénéisée) qui vont à l'encontre de la réutilisabilité des modules préconçus.

L'introduction d'un nouveau modèle de conception destiné à améliorer la flexibilité des structures d'agent est nécessaire, c'est le but du modèle componentiel que nous avons défini.

Le but de ce chapitre est de montrer qu'à partir d'un modèle objet classique nous pouvons obtenir un modèle composite. Nous montrerons qu'à partir de ce modèle composite associé au modèle Acteur nous obtiendrons un modèle modulaire. Enfin nous montrerons qu'à partir du modèle modulaire nous pouvons facilement obtenir un modèle componentiel.

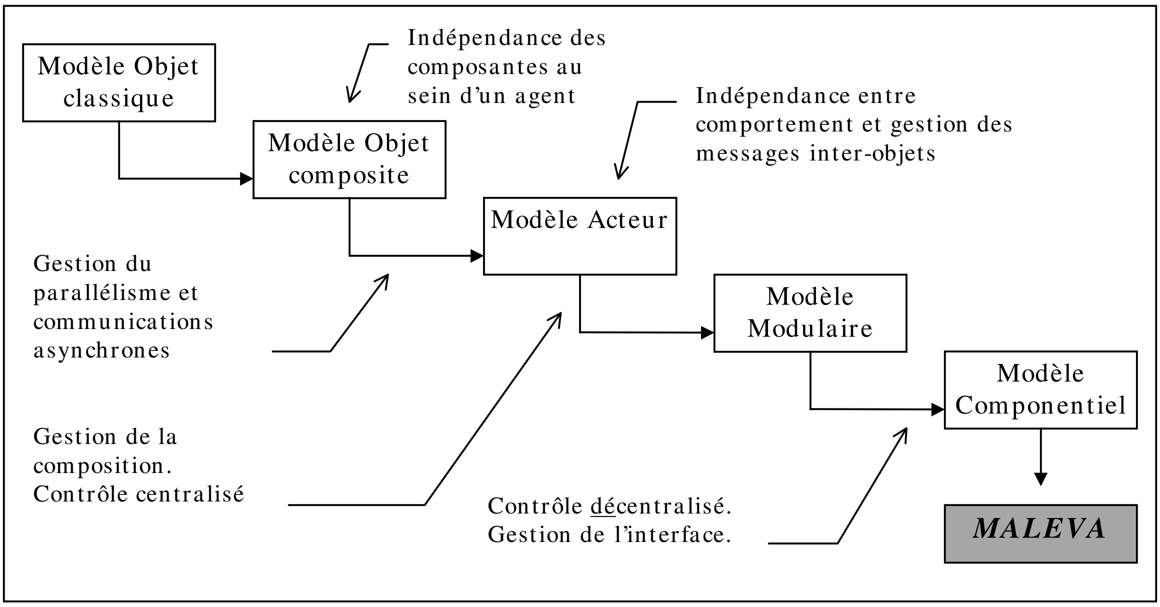


Figure 5: D'un modèle à l'autre

Ce chapitre est donc décomposé comme suit. Nous étudierons dans un premier temps les approches de conception (d'agent, de SMA et d'environnement de simulation) à partir des modèles objets, acteurs et modulaire tels que ceux utilisés dans différentes plates-formes de simulation. Nous expliciterons par la suite le but du modèle componentiel et nous finirons par définir le modèle componentiel et son application.

III.1 QUELQUES TYPES DE MODELES CONCEPTUELS

Les techniques et outils de développement de Systèmes Multi-Agents font appel à des langages de plus en plus adaptés aux systèmes distribués ou répartis. On peut cependant noter que pour la plupart ces langages sont basés sur des modèles objets, acteurs et/ou des approches modulaires.

Les modèles objets sont utilisés car ils permettent de bénéficier des avantages des langages objets (héritage, polymorphisme et encapsulation). De plus ces modèles permettent une approche incrémentale de la conception d'agent et permettent la réutilisation des objets déjà définis. Les langages acteurs permettent d'ajouter un parallélisme au sein du système, nous verrons d'ailleurs que cela introduit des contraintes au niveau de la simulation. Enfin le modèle modulaire assure une certaine dynamicité au système en permettant l'ajout de modules plus ou moins spécialisés. Il autorise la conception d'architectures différentes et hétérogènes au sein d'une même application, ce qui à terme permet de rendre chaque agent plus adapté à son environnement ou à son but.

Chacun de ces modèles va influer directement ou indirectement sur la structure de l'agent (l'ensemble de ses composantes) ainsi que sur le mode d'activation (séquentiel/parallèle) et le mode de communication (synchrone/asynchrone).

III.1.1 MODELES OBJET

Le modèle objet est certainement le modèle le plus utilisé pour la conception d'agents et de SMA. Dans l'équipe Miriad le langage de référence reste encore Smalltalk (Smalltalk80, VisualWorks etc...) bien certaines plates-formes commencent à être conçues avec des langages tels que C++, Java et, pour Maleva le langage Delphi. L'utilisation d'un langage tel que Smalltalk est d'autant plus normale que de nombreux outils de conception ont été réalisés dans cet environnement et, par conséquent, facilitent la réalisation de plate-forme de SMA.

L'utilisation d'un langage orienté objet est naturelle dans le sens où nous identifions naturellement un agent à un objet. En règle générale, un agent est une instance d'une classe et la conception de nouveaux agents se fait par héritage de classes existantes.

Ce qui caractérise les agents basés sur le modèle objet est l'utilisation de deux modèles conception, le modèle classique ou le modèle composite. Le plus souvent nous avons à faire à une architecture classique avec décomposition de l'agent en plusieurs composantes. L'agent y est vu comme une composition de plusieurs sous objets représentant respectivement les différentes caractéristiques physiques et/ou comportementales de l'agent.

Modèle classique

Dans une approche classique, l'approche la plus souvent utilisée est de représenter un agent par une classe qui contienne les trois composantes principales de l'agent: la composante Capteur(s) associée à la perception, la composante Comportement(s) associée à la délibération et enfin la composante Effecteur(s) associée à l'action. Chaque composante, et en particulier la composante comportementale, peut bien entendu être réalisée différemment suivant la plate-forme.

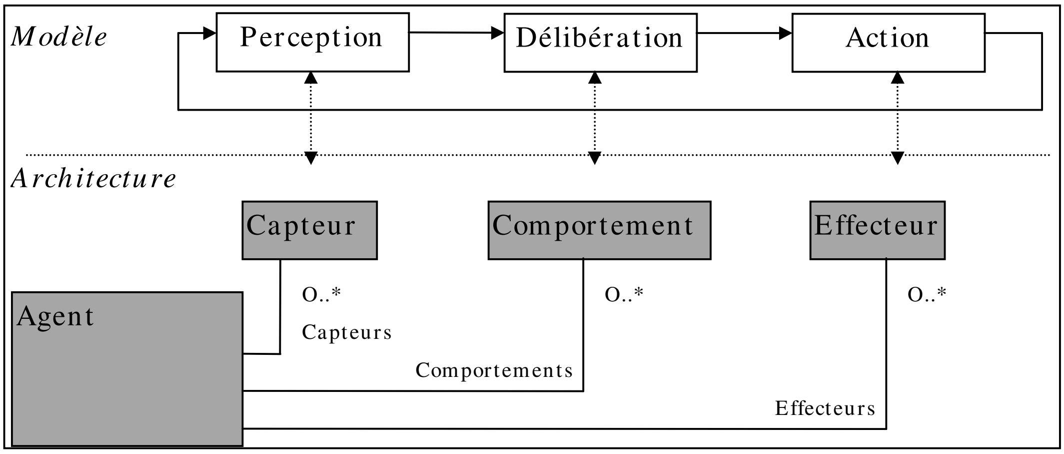


Figure 6: Architecture d'agent (Syntaxe UML) basée sur un modèle classique

Sur cette figure, nous voyons un exemple typique d'architecture d'agent où chaque composante (Perception, Délibération et Action) est décomposée en objets distincts[7].

Exemples d'utilisation

Dans la plate-forme ALIA dont la simulation traite de l'évolution morphologique et comportementale d'une société de robots en environnement hostile, un agent robot est représenté comme instance de la classe Agent qui possède 4 champs (composantes) qui sont respectivement Capteurs, Comportement, Effecteurs, Structure. Chacun de ces champs est une liste de pointeurs sur des instances, ce qui permet pour un agent de posséder plusieurs instances d'une même catégorie, par exemple un robot peut posséder un capteur d'agent et un capteur de balises. L'agent possède une sorte de contrôle centralisé qui permet d'inhiber certains capteurs, et effecteurs, afin de rendre effectif l'évolution morphologique de l'agent.

[7] En gris, figurent les classes que l'on supposera ici, sous classes de Object (Smalltalk).

Le mode d'activation des agents est séquentiel et celui de ses différentes composantes aussi. Le mode de communication se fait par envoi de messages synchrones

Dans la plate-forme Manta dont la simulation principale traite de la sociogénèse dans une colonie de fourmis, un agent fourmi est représenté comme une instance de la classe Ant, dont le seul champ représente l'ensemble des tâches qui entreront en compétition. Dans cette plate-forme une fourmi ne possède pour ainsi dire ni capteur ni effecteur. Toutes les fourmis possèdent la même morphologie, il n'apparaît donc pas utile de réifier Capteur et Effecteur sous forme d'objet. Le mode d'activation des agents est séquentiel et compte tenu que les fourmis ne communiquent pas directement entre elles mais par l'intermédiaire de l'environnement, on ne peut pas parler de mode de communication mais si celui-ci devait se faire (compte tenu de l'architecture actuelle) il se ferait par envoi de messages synchrones.

Ces deux applications utilisent la même architecture d'agent basée sur le modèle classique. Or, ce modèle a malheureusement le défaut d'être un modèle rigide. Quelque soit la structure des objets Capteur, Comportement et Effecteur, l'agent ne pourra pas contenir de nouvelles composantes telles qu'une composante de communication ou une composante d'anticipation. Ceci est dû au fait que les sous objets sont explicitement nommés dans la structure et que l'ajout d'une nouvelle composante reviendrait à réécrire la classe Agent ou à définir une sous classe contenant des champs supplémentaires. L'introduction de nouveaux champs dans la classe d'agent imposera de redéfinir l'interface de communication afin de faire correspondre les types[8] des objets entre eux.

Le même modèle à trois composantes deviendrait plus souple (donc plus flexible et plus réutilisable) s'il adoptait une architecture qui gère la composition, c'est à dire l'ajout de nouvelles composantes à une structure existante, il s'agirait alors d'une architecture basée sur un modèle Composite.

Modèle composite

Le modèle composite est un modèle d'architecture basé sur la composition d'objets au sein d'un objet englobant. Le modèle de base définit une architecture arborescente et hiérarchique composée d'objets composites et d'objets feuilles. Les objets composites sont les objets susceptibles de contenir d'autres objets, et les objets feuilles sont les objets de base.

[8] Deux objets ne peuvent communiquer entre eux que s'ils possèdent un type commun. Le type étant un sous groupe de leur interface [Gamma96].

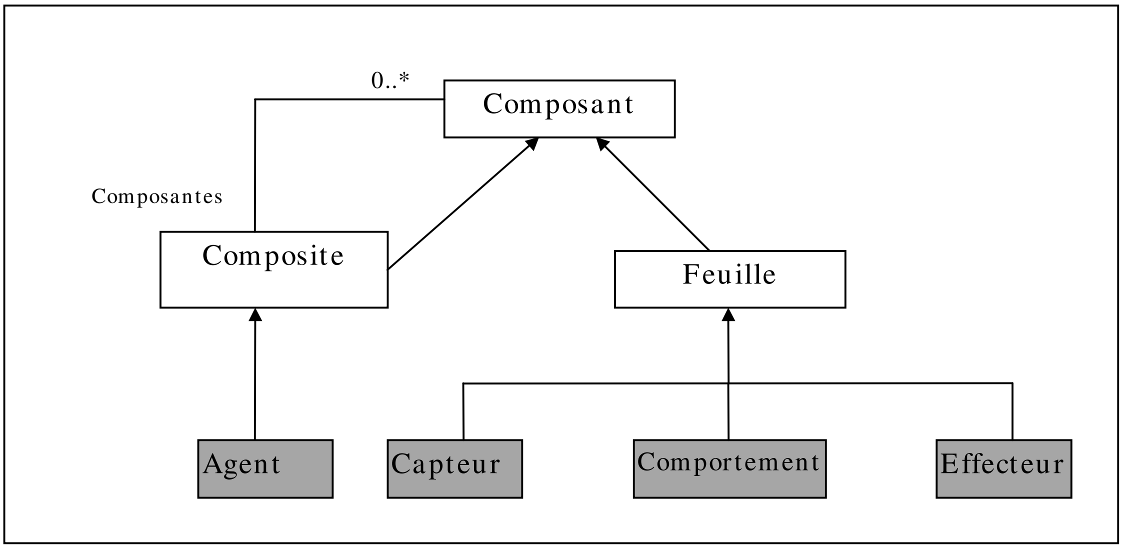


Figure 7: Un exemple d'agent basé sur un modèle composite

L'architecture de base (classes Composant, Composite et Feuille) est inspirée du modèle de conception proposé par [Gamma96, pp189], sa description est faite en syntaxe UML[9]. Sur cette figure, un agent est défini comme étant un objet composite, nous avons considéré que les autres classes étaient des feuilles bien que cela ne soit pas obligatoire. Un agent peut alors posséder un nombre quelconque d'objets composites ou feuilles.

La grande différence entre ce modèle et le précédent est la définition de la classe Agent. Il n'y a plus de nommage explicite des sous composantes et par conséquent toute nouvelle sous classe de Composite ou de Feuille peut être ajoutée au modèle d'agent existant. L'introduction d'une composante de communication se fait alors naturellement en ajoutant statiquement (avant compilation ou instanciation) ou dynamiquement une instance de la classe dans l'agent.

Certaines applications (dont la plate-forme Geisha que nous verrons plus bas) utilisent le principe du modèle composite pour concevoir des SMA. En règle générale, les composantes Capteur, Comportement et Effecteur sont des sous-classes de la classe Agent (ou de son équivalent) ce qui permet de modéliser un agent comme un ensemble de sous agents en interaction au sein d'un agent englobant.

[9] La syntaxe UML [Muller97] utilise les termes association, composition et agrégation. Les notions de composition et l'agrégation, telles que définie dans UML, sont différentes de celles utilisées dans cette thèse, nous n'utiliserons donc pour ce diagramme que l'association.

La plate-forme Geisha[10] est un exemple d'une telle architecture où l'agent est remplacé par la cellule. La cellule est elle même composée de sous cellules jusqu'au niveau des cellules « indivisibles ».

Qualités du modèle Objet

Que l'on utilise le modèle classique ou composite, l'intérêt du modèle objet est double : on a une représentation d'un agent (une classe) dans une structure simple, et nous pouvons réutiliser, hériter ou augmenter facilement les classes existantes. Nous verrons par la suite que le modèle composite peut être utilisé comme support pour le modèle modulaire.

- Avoir une représentation d'un agent sous forme d'une classe permet de l'identifier facilement (un agent est représenté par une instance unique dans le système). Elle permet également d'implémenter simplement et efficacement des opérations de sauvegarde, restauration, affichage (interface utilisateur), mais aussi de créer et détruire des agents (gestion de la simulation), ces opérations étant couramment utilisées dans les simulations.

- Réutiliser les modèles existants permet de créer de nouvelles classes d'agents par héritage des classes prédéfinies. Dans Alia, on peut créer de nouveaux types de capteurs ou d'effecteurs. Dans Manta cela n'est pas utile car la morphologie des agents importe peu et seul le comportement des agents permet de les différencier. Dans Geisha, il suffit de modifier (ou d'ajouter) une classe de cellules pour modifier (ou augmenter) le comportement global de toutes les cellules qui la contiennent.

Défauts du modèle Objet/Composite

1/ Dans le modèle composite, on doit offrir un outil de conception d'objets composites qui facilite la création d'agents. Il faut de surcroît gérer le maintien de cette structure qui n'est plus définie au niveau de la classe mais au niveau de chaque instance. L'introduction d'un modèle de conception du type Monteur [Gamma96] c'est à dire un objet responsable de la conception d'objets composites devient nécessaire.

2/ Chaque composante doit adopter une interface de communication compatible avec les autres composantes, c'est à dire posséder un type commun. Dans le modèle composite, si la classe Composite permet de déterminer les méthodes minimales pour la gestion de la composition, il n'en demeure pas moins qu'un objet doit pouvoir « dialoguer » avec n'importe quel autre objet composant l'agent. Si nous reprenons le cas, par exemple, de la focalisation de la perception, l'objet Comportement doit pouvoir indiquer à l'objet Capteur qu'il doit se spécialiser sur la perception d'un agent. Il faut donc que ces deux objets sachent quels messages envoyer (objet Comportement) et quelles méthodes implanter (objet Capteur). De plus,

[10] Geisha est une plate-forme de simulation conçue par D. Tarbourdeau. Les agents représentent des cellules vivantes. Chaque cellule doit « s'associer » à d'autres cellules afin de former des organismes de plus haut niveau.

chaque objet doit gérer la présence ou non des objets avec lesquels il sera amené à communiquer.

3/ Parallèlement au principe précédant, il faut dorénavant gérer les communications intra-agent. Le concepteur ne connaissant pas à l'avance la structure de l'agent, il faut traiter l'envoi de message entre les composantes et notamment savoir à qui et comment envoyer un message. Dans le modèle objet classique, les champs étant nommés, il n'y a pas de problème, le mode de communication standard fait l'affaire tandis que dans le modèle composite il faut identifier la (ou les) instance(s) à qui on veut envoyer un message, envoyer le message et gérer le(s) résultat(s). L'introduction d'un modèle de communication autre que le point à point devient nécessaire, nous y reviendrons dans les modèles qui suivent.

4/ Le mode d'activation reste séquentiel, les méthodes sont activées les unes à la suite des autres, et donc les objets, et par conséquent les agents, sont activés séquentiellement. Bien que le parallélisme ne soit pas dans tous les cas une nécessité au vue de la plupart des simulations, il doit quand même être offerte au futur utilisateur/concepteur de la plate-forme et ceci afin de permettre de réaliser de nouvelles architectures.

L'utilisation du parallélisme est préférable et nécessaire dans certains cas. Lorsque le modèle simulé est naturellement distribué, et, que les différents types d'agents (ou les composantes des agents) possèdent des cycles d'activations non corrélés[11]. C'est le cas, par exemple, dans les architectures temps réel [Guessoum96]. Lorsque le modèle propose des agents qui doivent fonctionner en parallèle. C'est le cas des architectures fonctionnant en réseaux et nous renvoyons le lecteur aux travaux de Yoo [Yoo97].

Comme nous le voyons, si les LOO possèdent des qualités nécessaires à la conception de SMA (en supposant que les classes soient bien conçues), ils pèchent par une trop grande rigidité des classes, par un mode de communication trop simple et un mode d'activation séquentiel.

L'introduction d'un nouveau modèle, qui permette de gérer certaines de ces contraintes, est nécessaire. Les langages d'acteurs comme nous allons le voir permettent de pallier à certains de ces problèmes, notamment en définissant d'autres modèles de communications et en permettant le parallélisme d'activation des agents.

III.1.2 Modele Acteur

Le modèle acteur est souvent employé dans la conception de SMA et de plates-formes de simulation (Mages, Dima, Maleva). Ce modèle est proche du

[11] Nous reviendrons par la suite sur la notion de cycle d'activation.

modèle objet. Il peut d'ailleurs s'implanter comme une sur-couche du modèle objet auquel il ajoutera des fonctionnalités de communication et d'activation parallèle (Actalk, Mering IV). Nous allons nous intéresser aux principes et à l'utilisation d'un langage d'acteur dans le cas de plates-formes de simulation de SMA. Nous verrons aussi que l'emploi d'un modèle Acteur peut introduire des contraintes au niveau des simulations et notamment quand à la gestion du temps et des synchronisations des agents.

III.1.2.1 MODELE ACTEUR : PRINCIPES

Si l'on se réfère aux travaux de Hewitt [Hewitt 77], de Agha [Agha 86] et les langages acteurs tels que ACTI, ABCL, Plasma II, Actalk [Briot 89][Briot94] et Mering [Carle92][Ferber91], on peut définir les langages acteurs et donc les acteurs en regroupant leurs fonctionnalités autour de 5 thèmes:

- **Le parallélisme :** L'ensemble des calculs sont effectués a priori de façon parallèle.
- **la communication :** Les acteurs peuvent communiquer par envoi de messages asynchrones bufferisés. Le principe même de ce type de communication suppose un envoi de message sans retour avec continuation en argument. Ce type de communication permet d'envoyer un message à un autre objet sans attendre que cet objet soit inoccupé (c'est à dire qu'il soit dans l'état de recevoir le message) et surtout sans avoir à attendre le résultat du message.
- **l'autonomie :** Il n'y a pas de partage de données entre les différents acteurs si ce n'est bien sûr les données partagées. L'autonomie provient du fait qu'un acteur peut dès lors traiter et manipuler ses données sans avoir à se soucier des autres acteurs du système.
- **la dynamicité :** Les accointances entre les acteurs sont dynamiques et par là même permettent de modifier le réseau des communications.
- **la localité :** Un acteur n'a qu'une vue partielle de son environnement et ne peut donc pas communiquer avec un autre acteur s'il ne le connaît pas directement ou indirectement (indirectement dans le cas ou par exemple un acteur est transmis comme continuation d'un calcul).

Les acteurs peuvent donc être vus comme des processus indépendants possédant leur propre zone de code et leur propre zone de données. Chaque Acteur à un comportement minimal qui correspond à traiter les messages en attente dans leur boîte aux lettres.

A titre d'exemple nous allons voir le langage Actalk qui est souvent utilisé pour la conception de noyau de SMA. Maleva dans sa version Smalltalk utilise cet outil afin de faire correspondre Acteur et Composant.

III.1.2.2 ACTALK

Actalk est un framework de programmation par acteurs développé sous le langage Smalltalk comme une extension même de ce langage. Les acteurs Actalk sont des objets Smalltalk développés autour de deux grandes classes. La classe Actor qui gère le comportement à la réception d'un message et la classe ActorBehavior qui correspond au comportement de l'acteur.

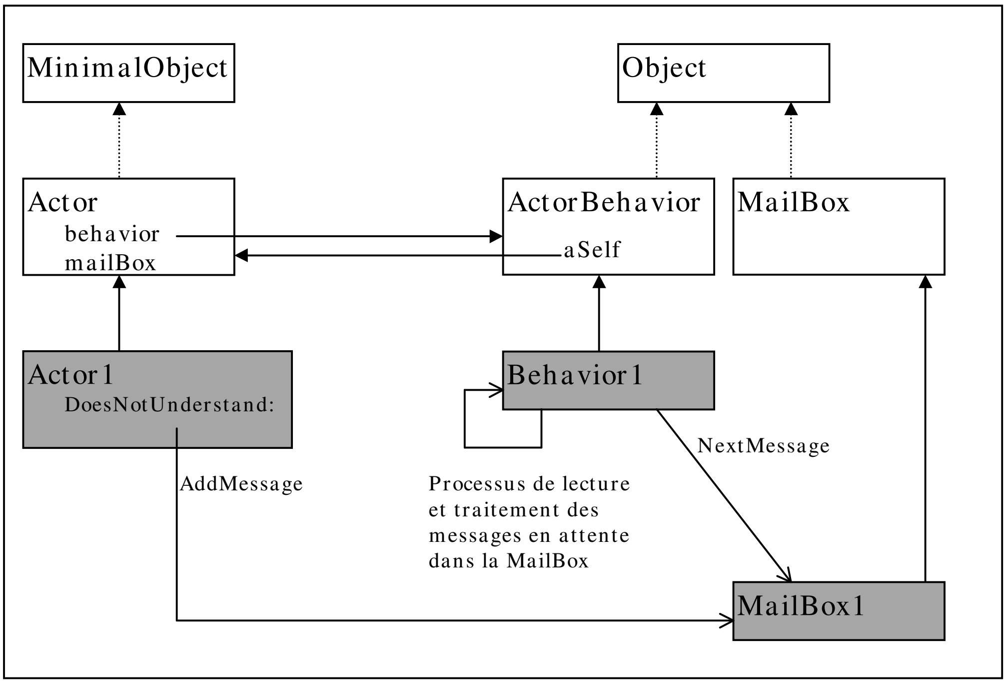


Figure 8: Modèle Acteur d'Actalk

La figure précédente représente le schéma général du modèle Actalk dans une version déjà ancienne[12]. Dans une version plus récente, le modèle de base contient trois grandes classes. La classe <u>ActorBehavior</u> a été scindée en deux classes <u>BehaviorObject</u> et <u>Activity</u>, et enfin la troisième classe <u>Actor</u>.

Les acteurs communiquent par envois de messages « classiques »[13]. A la réception d'un message, l'acteur génère un message de signification d'erreur *doesNotUnderstand:* (en effet il n'implémente pas, en général, la méthode invoquée) qui aura pour conséquence la réification et l'ajout de ce message dans la boite aux lettres. Un processus Smalltalk, défini au niveau du comportement,

[12] Nous présentons cette version car c'est elle que nous avons utilisé dans la première version de Maleva. De plus la comparaison entre les différentes versions d'Actalk tend à montrer l'évolution de ce framework vers un modèle componentiel.

[13] C'est à dire un message qui intègre le Receveur.

ira alors régulièrement prendre un message en attente dans la boite aux lettres pour le traiter. Dans la version plus récente, la classe Activity permet d'associer différentes stratégies de prise en compte des messages en attente dans la boite aux lettres. Par exemple, un acteur Actalk permet d'activer différents comportements distincts pour traiter plusieurs messages.

Hormis les qualités propres des langages d'acteurs, cette architecture a des qualités supplémentaires qui sont :

- d'une part de dissocier le gestionnaire de communication du comportement de l'acteur. Ceci permettra d'associer différents gestionnaires de messages pour des comportements (objets) identiques. Ceci est très important si l'on considère que nous voulons obtenir une architecture ouverte qui puisse intégrer des objets provenant de langages ou de systèmes différents et qui par conséquent n'obéissent pas aux mêmes protocoles d'envoi de message. Il s'agit là d'un aspect très important de la programmation de composantes réutilisables, la plate-forme Cell de D. Konstantas implémente la même notion en distinguant, dans une cellule, le noyau de la membrane. Le noyau est chargé de fournir des services utilisés localement (équivalent au comportement de l'acteur) tandis que la membrane est chargée de fournir des services avec l'extérieur (équivalent de l'acteur en tant que gestionnaire de messages).

- d'autre part de s'adapter facilement au modèle objet de base (à un héritage près). Un Agent y est alors conçu non plus comme une composition d'objets mais comme une composition d'acteurs. Le modèle composite peut parfaitement fonctionner sur un modèle d'acteurs. Nous verrons d'ailleurs que cela nous renverra naturellement vers le modèle modulaire. La seule contrainte pour passer d'un modèle à l'autre revient à traiter différemment les envois de messages puisqu'en mode acteur nous travaillons essentiellement avec des continuations.

- de faciliter le parallélisme intra et inter-agent. L'utilité d'une telle décomposition revient à décomposer l'agent en unités indépendantes, chaque unité peut dorénavant être partagée par plusieurs agents ce qui permettra d'assurer une uniformité dans la sémantique des services proposés.

Le parallélisme bien qu'étant un atout dans les Systèmes Multi-Agents qui sont par nature distribués peut cependant devenir une contrainte pour certaines simulations et notamment la gestion de la simulation et du temps. En effet l'ajout du parallélisme introduit des contraintes de synchronisation qui se retrouvent aussi bien au niveau du SMA qu'au niveau de l'agent si celui ci est conçu comme un ensemble de composantes en compétition.

III.1.2.3 PROBLEME DU TEMPS

La simulation d'agents situés[14], impose certaines contraintes quand à la gestion du temps. Dans la plupart des plates-formes destinées à la simulation, un agent obéit à un cycle d'activation qui représente la succession des opérations de perception-délibération-action. La notion de cycle d'activation est importante pour la simulation d'agents et plus particulièrement pour la notion d'agents situés. Associé au cycle d'activation nous pouvons alors déterminer le cycle de simulation qui correspond en quelque sorte au cycle des cycles d'activation, c'est-à-dire à la succession des opérations d'exécution du cycle d'activation pour l'ensemble des objets du monde.

Cycle d'activation

Comme nous l'avons dit, un cycle d'activation correspond à l'exécution de l'ensemble des opérations de perception-délibération-action. Bien entendu la durée réelle du cycle dépend de l'architecture de l'agent qui peut, par exemple, ne pas avoir de composante de perception ou de délibération, ou à l'inverse dont la composante délibération soit complexe par l'utilisation de sous composantes de communication ou de gestion d'une base de règles.

L'utilisation d'un cycle d'activation permet de déterminer le nombre d'itérations successives qui auront été nécessaires, pour un agent donné, pour accomplir un but donné.

Le nombre de cycles permettra par la suite de comparer l'efficacité (relative) des agents du monde. Plus l'agent mettra de cycles moins il sera efficace.

Enfin le cycle d'activation permet de déterminer un commencement et une fin d'activation pour un agent, et par conséquent, un état stable (cohérent) de celui-ci. Ceci permet une bonne gestion de la simulation même, par exemple les sauvegardes ou la génération de courbes .

Cycle de simulation

Le cycle de simulation ou UTS (Unité de temps de Simulation [Cazoulat95]) correspond donc à l'activation de l'ensemble des objets du monde[15].

Le cycle de simulation permet d'activer des objets du monde qui ne sont pas des agents, par exemple dans le cas de Manta, les cases (l'environnement

[14] Un agent situé est, par définition, un agent qui se trouve situé dans un environnement, qu'il perçoit et où il se déplace. Pour une définition plus précise, voir « Les systèmes Multi-Agents » [Ferber95], pp 16,17.

[15] Dans certaines plates-formes, l'environnement est activé durant le cycle de simulation. De plus il existe des objets (balises, bases, ressources...) qui ne sont pas des agents (on les nomme souvent des objets actifs) mais qui doivent agir à chaque cycle.

est discrétisé en cases adjacentes) doivent êtres activées afin de permettre la propagation de stimuli.

L'utilisation d'un cycle de simulation permet de déterminer le nombre d'itérations successives nécessaires, pour une société d'agents, pour accomplir un but donné. Par exemple, dans Alia, il faut environ 4000 à 10000 cycles pour qu'une société de robots s'adapte correctement à son environnement. Dans Mages IV (voir plus loin), c'est le nombre moyen de cycles[16] nécessaire à la capture d'une proie qui permet de comparer les différentes architectures d'agents et par la même de déterminer la meilleure.

Le cycle permet aussi de figer le monde (environnement + agents + données) à un instant donné et d'en extraire des graphiques, par exemple dans Manta, l'évolution de la société de fourmis permet d'en extraire des courbes d'évolution qui sont fonction du nombre de cycles [Drogoul93].

Enfin et le plus important, le cycle de simulation permet de synchroniser les couples agent/agent et les couples agent/environnement. La notion de synchronisation est très importante car elle permet d'assurer la bonne corrélation qui peut exister entre la perception qu'un agent a du monde qui l'entoure et les actions qu'il va y effectuer.

III.1.2.4 SYNCHRONISATION DES AGENTS

La synchronisation des agents dans un environnement est d'une part destinée à la gestion des contraintes d'accès à des données partagées (ressources) et d'autre part la gestion efficace de la simulation (sauvegarde, restitution).

Que le modèle d'agent soit basé sur un modèle Objet (donc un modèle séquentiel) ou sur un modèle Acteur (donc essentiellement parallèle) il faut, dans le cas d'accès à une ressource partagée, gérer le maintien de données cohérentes. Dans Alia, deux robots ne doivent pas consommer simultanément la même ressource, dans Mages, deux prédateurs ne peuvent pas occuper la même case (on considère ici qu'une case est une ressource) au même moment.

Si l'on veut pouvoir sauvegarder le monde à un instant donné, il faut que les agents et l'environnement soient dans un état stable (en début ou fin de cycle d'activation), et ceci ne peut se faire qu'en synchronisant les différentes entités que nous manipulons.

Comment synchroniser ?

On utilise dans ce cas un gestionnaire qui se nomme suivant les cas un Scheduler, un Synchronisateur, un Activateur mais dont le fonctionnement est

[16] On utilise, dans ce cas, les cycles de simulation au lieu de cycle d'activation. En effet, à certains niveaux de coopération les prédateurs chassent en groupe, et, comparer les activations individuelles ne correspond plus à rien.

quasiment identique et qui correspond finalement à déterminer un cycle de simulation.

J. Ferber utilise dans son modèle [Ferber95] un Synchronisateur dont la définition est : « Il s'agit d'une horloge qui évite que des actions effectuées de manière simultanée soient interprétées comme étant totalement séparées dans le temps… assure qu'à un moment donné tous les agents agissent simultanément et que l'environnement ne réagisse qu'ensuite… », et qui fait référence à la figure suivante.

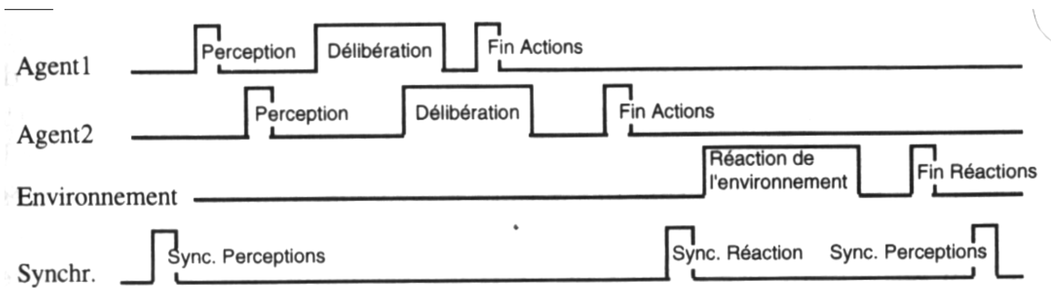


Figure 4.49: Diagramme de synchronisation pour deux agents et un environnement

Figure 9: Synchronisateur ([Ferber95], page 233) pour activation parallèle

Comme nous le voyons sur la Figure 9, le synchronisateur détermine d'une part, le début et la fin du cycle d'activation (la période comprise entre Sync. Perceptions et Sync. Réaction) et d'autre part le début et la fin du cycle de simulation (la période entre deux Sync Perception). Le Sync. Réaction permet de synchroniser les agents du monde et de traiter l'ensemble des actions que ceux-ci désirent effectuer. Il est à noter que le modèle d'agent de J. Ferber fonctionne sur un modèle Acteur et que par conséquent l'événement Sync. Réaction ne peut intervenir qu'une fois que l'ensemble des agents a envoyé un Fin Action, il s'agit donc d'un Sémaphore.

Dans ce cas, on peut traiter la sauvegarde à deux instants, soit au moment du Sync. Réaction auquel cas il faut enregistrer aussi les actions des agents puisque celles-ci ne sont pas encore réellement appliquées à l'environnement, soit au moment du Sync. Perception.

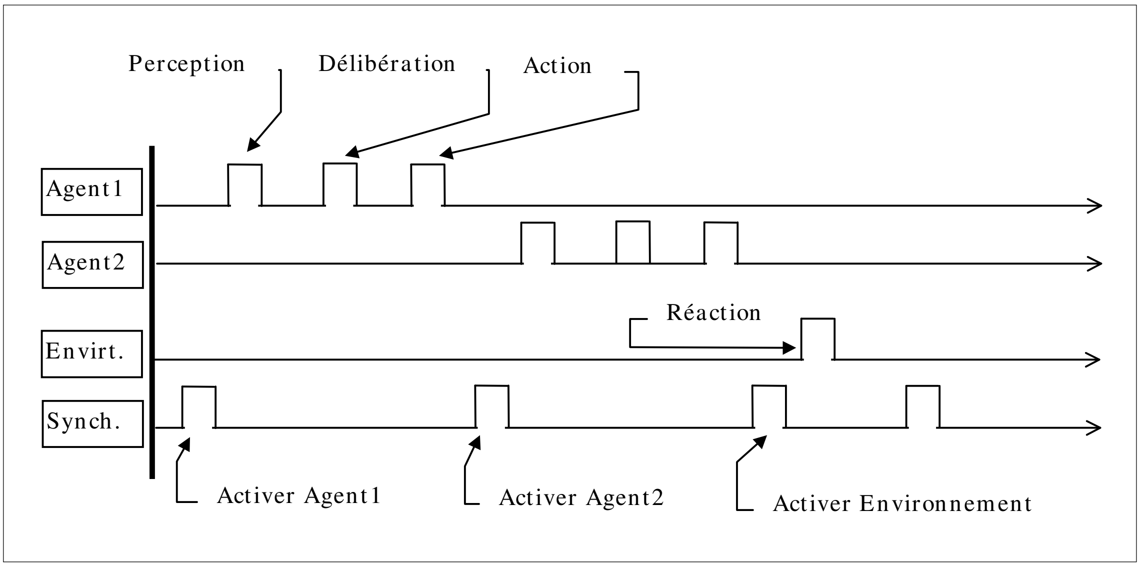


Figure 10: Synchronisateur pour activation séquentielle.

Dans le cas d'activations séquentielles, c'est-à-dire qu'un agent n'est activé qu'une fois que son prédécesseur a fini son cycle, on ne parle plus de synchronisation mais d'activation (d'où le nom parfois utilisé d'Activateur au lieu de Synchronisateur). Le Sync. Réaction n'est plus utile. En effet si l'on active l'environnement en dernier, on est assuré que chaque agent a déjà effectué son cycle d'activation (Cf. Figure 10).

Pourquoi synchroniser ?

Si la synchronisation des agents et de l'environnement sert à la gestion des ressources partagées ou à la gestion de la simulation, elle permet aussi d'assurer une certaine cohérence quand à l'activation des agents.

Si il n'y avait pas de synchronisation, c'est à dire que chaque agent exécute son cycle d'activation indépendamment des autres avec une durée réelle qui soit par conséquent dépendante de son architecture et de sa complexité, dans ce cas, les agents aux architectures les plus simples s'exécuteraient plus rapidement et plus souvent que des agents plus complexes.

Cette indépendance est souvent recherchée, par exemple dans les architectures hybrides temps réel [Guessoum96]. Le but est d'offrir un agent qui possède à la fois une partie cognitive et une partie réactive qui puisse répondre en temps réel à des stimuli externes. Si le but est louable, l'absence de synchronisation peut parfois introduire des incohérences quant à la simulation. Il est d'ailleurs à noter que bien souvent ces architectures proposent une composante de contrôle centralisée dont le but même est de gérer les problèmes de synchronisation.

Exemple d'utilisation

A titre d'exemple, dans la plate-forme Mages IV la simulation est basée sur un exemple inspiré du proie/prédateurs avec différents niveaux de communications.

Mages IV est une plate-forme de conception et de simulation de SMA conçu par T. Bouron [Bouron91, Bouron92], le modèle d'agent y est conçu autour du modèle Objet/Acteur défini par Actalk [Briot89]. L'architecture proposée permet la conception d'agents à granularité variable. Au premier niveau les agents (dans l'exemple proposé, les prédateurs) sont tous réactifs, alors qu'au dernier, ils développent des stratégies de chasse basées sur des communications de haut niveau (Actes de Langages).

Si la simulation a pour but de déterminer la ou les meilleures architectures d'agents pour un scénario donné, nous nous apercevons que les agents de haut niveau passent plus de temps à communiquer qu'à chasser ce qui va au détriment du but final qui est d'attraper la proie. Dans ce cas, la proie qui reste un agent purement réactif peut agir beaucoup plus rapidement que les prédateurs, autrement dit pendant que les prédateurs « discutent » entre eux, la proie s'enfuit.

On pourra bien sûr objecter que ceci est normal puisque dans la réalité cela se passe comme cela, mais doit-on prendre en compte le temps mis par chaque agent pour communiquer sachant que cette durée n'a rien à voir ni avec la réalité ni avec le but recherché. Dans un système incorporant un synchronisateur, les communications seraient intégrées dans le cycle d'activation de l'agent et n'influeraient pas sur le temps de réponse et donc sur la simulation.

En résumé, munie d'un synchronisateur, une plate-forme utilisant des agents conçus avec un modèle Acteur facilite la conception, le maintien et la gestion de la simulation. Dans Maleva, le synchronisateur porte le nom de Scheduler (Séquenceur) que nous étudierons dans le chapitre suivant.

Un autre exemple de l'utilisation des langages d'acteurs pour la conception de SMA est donné dans le modèle de programmation LRO3 [Roche93]. Le modèle d'agent défini dans LRO3 ne peut communiquer, d'après les auteurs, que par envoi de message asynchrone même s'il est capable d'utiliser des protocoles de communication éventuellement complexes, d'où l'apparente analogie agent/acteur. Cependant comme nous le verrons dans le paragraphe suivant, l'architecture proposée dans LRO3 propose un modèle d'agent multi-niveau qui revient à concevoir un agent comme une société d'agents de granularité plus faible.

Si le modèle Acteur permet de gérer les contraintes liées au parallélisme au sein d'un agent et d'un SMA, il n'en demeure pas moins qu'il reste le plus souvent une sur-couche d'un modèle objet et ne permet pas d'obtenir un modèle totalement flexible, en effet un acteur doit connaître l'acteur à qui il s'adresse.

Il reste à gérer les contraintes de composition et de gestion d'objets composites, nous allons alors utiliser un nouveau modèle de conception : le modèle modulaire.

III.1.3 MODELE MODULAIRE

Dans un système modulaire, un agent est une association de plusieurs modules spécifiques qui peuvent fonctionner en parallèle et donc en concurrence. Chaque module peut lui-même être composé de plusieurs sous-modules etc... Nous retrouvons ici le modèle Composite présenté auparavant associé à un modèle Acteur. Les applications basées sur ce modèle sont nombreuses, par exemple les travaux de P.Maës et des membres de son équipe [Blumberg & Galyean 95] sur les agents autonomes, les travaux de Z. Guessoum sur Dima, les travaux précités de C. Delaye, A. Drogoul .

Dans un système modulaire, un module est décrit par ses bornes d'entrées, ses bornes de sorties et sa fonction. La fonction d'un module représente l'ensemble des fonctions de transition entre les bornes d'entrées et les bornes de sorties. Le contenu du module n'est pas connu et n'a d'ailleurs pas à l'être, on a coutume de dire qu'il s'agit d'une boite noire. En ce sens, un module n'est pas différent d'un objet, on n'accède à un objet que par l'intermédiaire de ses méthodes (son interface) et le code interne de l'objet (son comportement) n'a pas à être dévoilé.

Chaque module peut représenter un comportement ou une partie physique de l'agent, par exemple il existe des modules de détection (des capteurs), des modules associés aux déplacements (des effecteurs), des modules de communication, des modules de sélection de tâche etc.

Cette approche offre plusieurs intérêts :

- Il reprend l'essentiel des modèles composites.
- Les modules peuvent être réutilisés plus facilement et plus efficacement. Les agents peuvent être construits par utilisation de modules pré-existants qui ont « déjà fait leurs preuves ». Les travaux de Min Y. Yoo sur la conception modulaire de réseaux de Pétri traitent de la validation/vérification d'un système par validation/vérification de chaque composante séparément des autres.
- Les modules peuvent fonctionner en parallèle et donc en concurrence. La concurrence (compétition) entre les modules permet de déterminer s'il en est de plus « efficaces » que d'autres dans un cas donné.
- Elle permet de faire coexister comportements réactifs et délibératifs. Les architectures hybrides en sont un bon exemple.

- Elle permet de modifier l'architecture suivant les besoins. En effet, un module n'étant défini que par ses bornes et sa fonction, les fonctions de transition peuvent employer n'importe quel modèle.

- Elle permet une approche visuelle de la conception d'agents. Il suffit d'un éditeur graphique pour ajouter un nouveau module à un agent.

Elle à cependant quelques défauts ou manques :

- Elle impose une gestion des messages inter-modules. Cette gestion des messages doit en principe résoudre les problèmes de partage d'information entre les modules, les problèmes de pannes.

- Au même titre que les modèles précédants, elle impose un protocole de communication entre les modules. Deux modules ne communiquent que s'ils se connaissent. La notion de connaissance veut alors dire qu'ils s'identifient et qu'ils savent ce qu'ils peuvent envoyer comme message. Lorqu'un objet (un acteur, un module) s'adresse à un autre objet, il doit connaître à la fois le destinataire, l'identificateur du message (la méthode) et les paramètres (nombre, type...). Une erreur, par exemple un mauvais identificateur est appelé, provoque soit une erreur de compilation, soit une interruption du programme (dans les langages interprétés) avec génération d'un message d'erreur (par exemple le doesNotUndertand: de Smalltalk).

- Elle complexifie la gestion informatique de l'agent (sauvegarde, restitution, debogage). Plus le système est découpé en sous parties, plus il devient difficile de gérer l'ensemble. Si l'on ajoute un mécanisme d'activation parallèle, deboguer devient une tache encore plus ardue.

- Elle fait disparaître l'agent au bénéfice des modules. Sauf pour certains modèles, il n'existe plus d'agent (structure) mais que des modules « connectés ». Comment faire alors pour visualiser cet agent dans l'environnement, comment récupérer les données intéressantes autrement dit comment gérer l'interface Agent/Utilisateur.

- Le modèle modulaire n'est pas conçu pour permettre la création de nouveaux modules par composition de modules pré existants. Ceci n'est pas en désaccord avec la définition initiale des modules, un module peut effectivement être composé de sous modules bien que le terme association soit plus correcte. En effet la composition qui est la base des modèle de conception composites va à mon sens plus loin que la simple mise en commun de plusieurs modules comportementaux, c'est d'ailleurs l'objet du modèle componentiel et des paragraphes suivants.

- Le modèle impose souvent un module de contrôle centralisé. Ce module doit gérer la transmission des messages aux autres modules du système mais surtout il doit activer ou inhiber les modules suivant les besoins.

Exemple d'utilisation

Le modèle LRO3 est un bon exemple d'utilisation des modèles acteur et modulaires. Le modèle introduit deux niveaux; le méta-niveau qui gère les messages et le niveau de compétence qui défini le comportement.

Nous retrouvons dans cette architecture la dissociation classique, employée dans les langages d'acteurs, du gestionnaire de message et du comportement propre de l'acteur. Le modèle LRO3 réifie chaque composante comme étant un agent d'un niveau inférieur jusqu'à un niveau indécomposable où les agents sont considérés comme étant atomiques. Nous retrouvons donc bien une approche modulaire par décomposition récursive de l'agent.

La dissociation gestion des messages et comportement permet comme l'indique les auteurs de prendre en compte la complexité des problèmes à résoudre et la diversité des solutions possibles qui nécessitent souvent l'emploi de protocoles de communication variables d'un agent à un autre.

Une particularité intéressante de ce modèle est de réifier les messages dans une structure qui contient :

- l'agent expéditeur : c'est-à-dire l'agent qui envoie le message, on peut supposer d'après l'article qu'il s'agit de l'agent Interface défini au méta-niveau.

- l'agent destinataire : comme précédemment on peut supposer qu'il s'agit de l'agent interface du client.

- la société d'appartenance de l'agent : ce qui doit représenter l'agent englobant de l'agent destinataire, en fait le groupe des sous agents du même niveau que l'agent destinataire.

- le contenu du message.

Et facultativement les champs suivants :

- le type du message : il s'agit d'une requête, d'une réponse ou d'une réclamation.

- un agent de continuation : nous sommes totalement dans une approche Acteur.

- un agent de délégation : l'agent qui va traiter le message au cas ou l'agent destinataire ne serait pas capable de le faire.

Cette structure propose de définir (facultativement) la notion d'agent de délégation, c'est-à-dire de pouvoir définir un autre destinataire dans le cas où le premier ferait défaut. Il est paradoxal de constater que c'est l'agent émetteur qui spécifie l'agent délégation et non pas le récepteur. Dans la plupart des modèles, c'est l'agent récepteur qui effectue les délégations. De plus, il semble que l'on ne puisse définir qu'un seul agent délégation ce qui est très limitatif quand aux possibilités de résolutions d'un problème donné.

Que manque-t'il à ce modèle ou quels sont ses défauts ? D'une part on ne trouve pas la définition d'un agent atomique, on ne connaît pas ni sa structure ni la façon de le concevoir. D'autre part, le modèle d'agent semble ne pas être conçu pour des agents situés, on ne trouve nulle part la notion de structure physique. La notion de message même si elle est prise en compte au Méta-niveau et notamment par l'agent Interface semble avoir été conçue dans l'optique d'envoi de messages asynchrones bufferisés avec continuation en argument. Ce type d'envoi de message même s'il est indispensable dans une approche distribuée impose que l'ensemble des agents du système obéissent au même principe ce qui peut aller à l'encontre d'une optique d'intégration d'agents provenant d'architectures différentes. Si la notion d'agent est importante, celle de groupe d'agent ne l'est pas moins. Même si le modèle proposé dans LRO3 stipule que l'on puisse avoir différents protocoles de communication, ceux-ci ne semblent être utilisés que dans une optique de communication agent/agent c'est-à-dire un mode de communication de type point à point. Or il est important d'offrir d'autres modèles tels que le Broadcast ou le Multicast que nous traiterons dans le chapitre suivant.

III.2 LE MODELE COMPONENTIEL

Le modèle componentiel (ou modèle à base de composants logiciels) est un modèle de programmation relativement récent lorsqu'il est appliqué à la conception d'agents. Ce modèle est essentiellement implanté dans les langages objets visuels (Visual Basic, Visual C++, Visual J++, Smalltalk... et bien sur Maleva) et il est encore essentiellement destiné aux applications Multimédia ou à la conception d'Interface Homme Machine.

Ce modèle s'inspire à la fois du modèle objet pour la conception, du modèle acteur pour les principes de parallélisme et enfin du modèle modulaire pour l'approche de conception par composition. Le principe de ce modèle est celui des modèles modulaires avec un niveau de flexibilité plus grand associé à une approche visuelle de la composition. La composition est la possibilité de conception de composants à partir de composants pré-existants et la flexibilité est basée sur la réification et la gestion des communications par un gestionnaire dépendant ou indépendant[17] du composant lui même.

[17] Suivant les architectures.

III.2.1 INTRODUCTION

Quelle que soit l'architecture adoptée dans la conception d'un agent dans un système donné, cette architecture impose certaines contraintes sur le modèle employé. Dans le modèle à base de tâches compétitives [Brooks 86][Drogoul 93A], la notion de compétition est intrinsèque et peut être obtenue avec un modèle objet classique car les tâches sont indépendantes les unes des autres. A l'inverse dans certaines architectures horizontales, les composantes d'un agent (objets ou modules) doivent s'activer parallèlement et doivent cependant communiquer entre elles. Le modèle objet classique n'étant pas à la base conçu pour ce genre d'approche, on doit alors se tourner vers les langage objets distribués ou les langages d'acteurs qui proposent de nombreuses primitives de communications. Si ces langages permettent d'appréhender certaines contraintes de parallélisme, l'utilisation d'un modèle modulaire devient obligatoire dès lors qu'il s'agit de modéliser un agent comme une société de sous agents c'est à dire adopter une approche récursive et incrémentale de la conception de SMA.

III.2.2 BUTS DE NOTRE MODELE

Dans le modèle componentiel appliqué aux Systèmes Multi-Agents, un agent n'est pas vu comme une entité fixée (dont la structure ou l'architecture est statique) et dont le comportement est régi par cette structure. Il ne s'agit plus d'un bloc monolithique mais d'une association à un moment donné d'un ensemble de composants fonctionnant en coopération ou en compétition.

Nous concevons un agent comme un <u>regroupement</u> de <u>petites structures</u>, lesquelles possèdent à la fois leur <u>propre comportement</u>, leur <u>propre interface</u> et leur <u>propre contrôle</u>. *Nous appellerons cette structure/ interface associée à ce comportement un <u>composant</u>.*

Une des particularités de cette approche est de pouvoir alors réutiliser les structures pré-définies pour concevoir de nouvelles structures de plus haut niveau, soit par augmentation des possibilités (héritage et polymorphisme) comme dans les langages objets, soit par association comme dans le modèle modulaire, soit enfin par <u>composition fonctionnelle et structurelle</u>. La composition fonctionnelle correspond à la mise en commun des fonctionnalités de chaque composant tandis que la composition structurelle correspond à la conception de structure (composant) de plus haut niveau par agrégation.

Un des but du modèle componentiel est de pouvoir concevoir et créer des composants (des agents) visuellement à partir de bibliothèques de composants pré-existants. Afin de faciliter la tâche de conception des futurs utilisateurs de la plate-forme, nous avons entrepris la réalisation d'un framework de programmation orienté graphique. Nous reviendrons sur Maleva dans le Chapitre consacré aux réalisations.

Un autre but est de pouvoir faire s'activer chaque composant indépendamment des autres. Chaque composant peut s'activer en parallèle des autres avec des périodes d'activation variables. La variabilité des périodes d'activation[18] va alors influer sur le flux de contrôle et le flux de messages entre les composants..

Le modèle componentiel s'inspire naturellement du modèle modulaire, un composant est une entité définie d'un point de vue externe par ses bornes d'entrée, de sorties et sa fonction.

Pour favoriser la programmation visuelle, on doit y ajouter des mécanismes de communication adaptés au modèle.

III.2.3 PRINCIPES

Les modèles à base de composants sont articulés autour de 3 grands concepts qui sont respectivement le composant, les interactions et la composition. Le composant représente l'entité de base que l'on manipule, en tant qu'objet, il permet la réutilisation et l'héritage. L'interaction permet la composition fonctionnelle des composants et la réalisation de l'application, la composition offre une possibilité supplémentaire de conception de composants de niveau supérieur à partir de composant plus simples, dans notre cas elle permet en plus une composition structurelle.

III.2.4 EXECUTION PARALLELE ET PROBLEME DU FLUX DE CONTROLE

Il existe beaucoup de modèle à base de composants logiciels, la différence principale que l'on peut trouver entre eux est due au traitement et à la gestion du contrôle. Dans les langages tels que Delphi, C++ c'est-à-dire des langages essentiellement destinés à la conception d'IHM, les composants sont rarement des processus, par conséquent il n'y a pas de problème de flux. Dans d'autres outils (framework, plates-formes, environnements de développements) telles que MaxTalk [Brethon97], Bric [Ferber & Magnin93], chaque composant est un processus à part entière. Dans de telles approches il est important de distinguer la notion de flux de contrôle et de flux d'information. Dans MaxTalk, le contrôle est une donnée particulière qui arrive sur une borne[19] particulière et se nomme Bang.

III.2.4.1 LE PROBLEME DU FLUX DE CONTROLE/ D'INFORMATION

Les modèles Objets et Acteurs diffèrent essentiellement par la nature des communications qu'ils définissent. Un Objet communique par envoi de

[18] Dans Maleva, la période d'activation est représentée par la valeur du cadencement du composant.

[19] Nous définirons plus loin la notion de borne.

message synchrone tandis qu'un Acteur communique par envoi de message asynchrone bufferisé. Or les modes de communications influent sur la nature du flux qui lie l'émetteur au récepteur. On parlera soit de flux de contrôles lorsque seul le(s) contrôle(s) est transmis au récepteur, soit de flux d'informations lorsque seules les informations seront transmises au récepteur.

III.2.4.2 FLUX DE CONTROLE

Dans un modèle Objet, le contrôle se transmet au travers du message. Lorsque un Objet émetteur A envoi un message à un objet récepteur B, il lui donne le contrôle. Ce contrôle est rendu à A lorsque B a terminé le traitement qu'il devait effectuer.

Le flux de contrôle tel qu'il est défini dans un modèle Objet est rigide, le modèle n'est donc pas flexible comme nous le souhaitons. En effet, si A communique avec B (donc A connaît B), B se doit d'exister sinon le contrôle est perdu, les langages objets courants provoquent une erreur (génération d'un message d'erreur ou génération d'une interruption dans le meilleur des cas) dans de tels cas. Certains modèles font apparaître la gestion de l'existence du récepteur bien qu'ils n'indiquent pas comment celle-ci est faite ni même les principes mis en œuvre pour gérer la disparition d'un objet possédant le contrôle.

III.2.4.3 FLUX D'INFORMATION

Dans un modèle Acteur, le flux de contrôle n'est pas défini. Lorsque un Acteur A envoie un message à un acteur B, il ne transmet que des informations (flux d'information) mais il garde son contrôle. En fait chaque acteur possède son propre contrôle.

Cependant il est possible de créer un flux de contrôle si le message est accompagné d'un arrêt du contrôle de la part de l'émetteur A, lequel peut alors reprendre son contrôle dès que le récepteur B lui transmet un signal.

Ce signal peut être défini comme un message qui va redonner le contrôle à A et bloquer B, et ainsi de suite. Comme nous allons le voir, cette technique correspond soit à synchroniser les messages soit à définir un mécanisme de continuation explicite [Carle 92].

III.2.4.4 CONTINUATION IMPLICITE

Le mécanisme de synchronisation (ou continuation implicite) est utilisé dans les protocoles de type Client/Serveur entre plusieurs applications. Une communication ne peut s'établir que si les deux applications sont actives et prêtes à dialoguer. Le protocole gère alors la transmission effective des informations et les problèmes en cas d'échecs. Maleva utilise ce mécanisme pour les communications entre les composants

Il est à noter que comme tout protocole, ce mécanisme est d'un niveau plus bas que la communication proprement dite, il ne fait que gérer le bon acheminement de l'information de l'émetteur au récepteur.

Les liens DDE (Dynamic Data Exchange) employés par certaines applications sous Windows, sont un bon exemple de mécanisme de synchronisation entre deux applications distinctes.

III.2.4.5 CONTINUATION EXPLICITE

Le mécanisme de continuation explicite permet quant à lui de créer plusieurs flots de contrôle ou de synchroniser les messages.

La continuation explicite permet de désigner le destinataire du résultat d'une requête qui n'est pas forcément l'émetteur initial de la requête. Donc si A possède le contrôle, il peut envoyer une requête à B avec une continuation.

Si la continuation est C, alors A, B et C possèdent leur propre contrôle et nous restons dans le cadre des langages d'acteurs classiques. Comme A,B,C possèdent leur propre boite aux lettres, chacun de ceux-ci peut continuer à travailler indépendamment des autres et des messages qu'ils s'envoient.

En revanche, si A se désigne comme continuation et se bloque (il cesse toute activité) en envoyant un message à B, et si il reprend le contrôle dès que B lui répond, alors A se synchronise par rapport au message qu'il a envoyé à B et nous revenons aux mécanismes d'envois de messages des LOO classiques.

Ce mécanisme à l'inverse du précédent n'est pas d'un niveau inférieur à celui de la communication, il impose de définir la continuation et à l'intégrer au message.

Si la plupart des modèles de conception componentielle différencient flux de contrôle et flux de données, la plupart d'entre eux ne traitent en fait que le flux de données. Dans Maleva nous avons opté pour une approche qui permette de totalement dissocier ces flux.

III.3 LE COMPOSANT

Comme nous l'avons dit précédemment, le modèle componentiel est fondé autour des trois grands principes qui sont respectivement le composant, la composition et les interactions. Nous nous attacherons ici à reprendre chacun de ces principes et à distinguer les différences conceptuelles qui peuvent exister entre les différentes applications bâties autour du modèle componentiel.

III.3.1 INTRODUCTION

En langage orienté objet la notion d'unité est étroitement dépendante de la notion d'objet. L'interface de l'objet (ce qui est accessible) est définie de manière syntaxique plus que de manière fonctionnelle, or la définition contraint et limite les interactions entre deux objets. La contrainte principale est que l'émetteur d'un message (l'émetteur) est obligé d'adopter la syntaxe de l'appelé (le récepteur), les deux entités doivent appartenir au même type [Gamma96], et par conséquent, l'émetteur est obligé avant même de savoir si le récepteur peut répondre à ses questions, de connaître l'ensemble (ou une sous partie) des méthodes d'accès de celui-ci. Ce problème qui intervient lors de l'implantation du programme se répercute et s'amplifie lorsque un même émetteur A veut interroger deux récepteurs B et C et que ceux ci n'adoptent pas la même syntaxe.

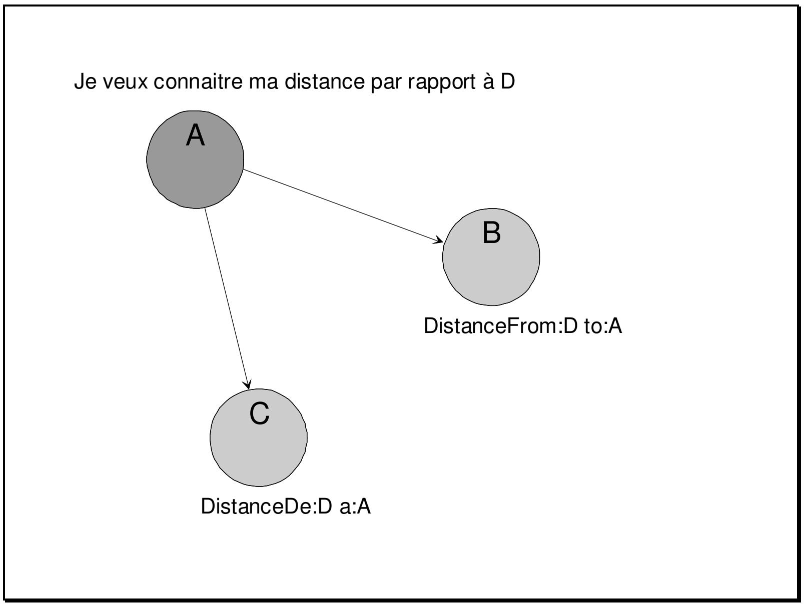


Figure 11 : Pour une même interrogation, deux syntaxes différentes

Sur cette figure nous avons volontairement imaginé deux objets récepteurs B et C qui proposent une même fonctionnalité pour deux syntaxes différentes. Sans pour autant prétendre résoudre complètement ce problème de syntaxe, les composants doivent être définis de telle sorte qu'un composant puisse interroger n'importe quel autre composant sans a priori connaître ni la syntaxe ni si le client peut effectivement répondre à la question (sur la Figure 11, rien ne permet de savoir si B ou C définissent les méthodes appelées). Dans le cas où justement un composant ne possède pas de méthode d'accès, il faut que le composant appelant puisse quand même continuer son traitement sans devoir s'interrompre à chaque appel pour vérifier la validité ou l'existence d'une réponse du client.

Le problème de la syntaxe et lié à celui de la répartition et particulièrement, comme nous l'avons vu au paragraphe II.2.4 à celui de la répartition forte. Pour résoudre le problème, il est nécessaire de définir un modèle de composant qui dissocie totalement l'interface du composant de son comportement.

III.3.2 DEFINITION

Il est difficile de donner une définition stricte du terme composant, O. Nierstrasz [Nierstrasz95] le définit comme : "*static abstraction with plugs*", ce qui veut dire selon l'auteur une entité d'une certaine durée de vie, unité indépendante de l'application qui l'utilise, entourant le programme qu'elle encapsule d'une membrane plus ou moins opaque et possédant des voies bien définies pour interagir et communiquer avec d'autres composants.

III.3.2.1 COMPOSANT

Nous donnerons une description plus concise en disant qu'un composant est vu comme une entité définie par un comportement interne et un ensemble de bornes externes (plugs, ports)[20]. Le comportement interne n'est pas visible de l'extérieur et n'est accessible qu'au travers des bornes. Il peut représenter une application, un programme, une fonction, un objet... Les bornes servent aux composants pour la gestion des interactions et des communications avec les autres composants du système. Dans la pratique, les composants sont vus comme des processus autonomes pouvant communiquer par l'intermédiaires de canaux de communication reliant des bornes d'entrée/sortie.

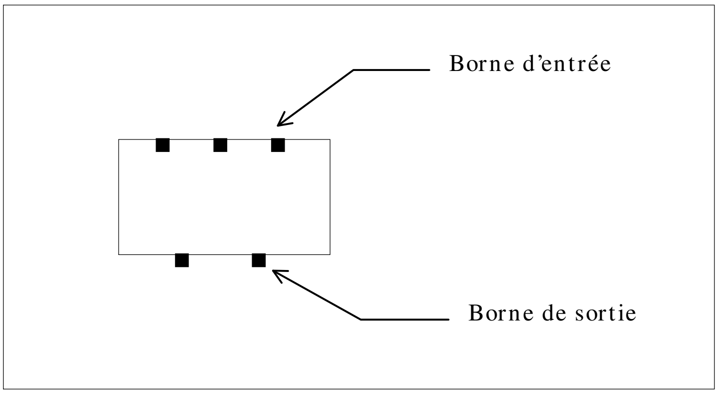


Figure 12: Composant

[20] Dans la littérature les termes plugs, bornes, ports désignent la même chose

III.3.2.2 COMPOSITION

La composition est le critère qui différencie le plus les modèles à base de composants. La composition peut être définie comme une composition fonctionnelle ou une composition structurelle.

Composition fonctionnelle

M. Nierstrasz nomme composition fonctionnelle l'opération qui consiste à définir les connexions entre les composants, la composition fonctionnelle est un des atouts les plus important du modèle componentiel, c'est ce qui le rend flexible.

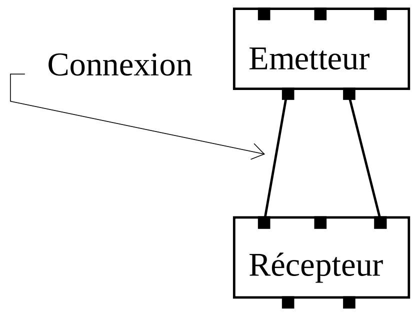


Figure 13: Composition fonctionnelle

Sur cette figure, nous voyons un exemple de composition fonctionnelle entre deux composants. Une connexion représente un canal de communication entre l'émetteur et le récepteur, elle ne peut relier qu'une borne de sortie à une borne d'entrée.

La composition fonctionnelle n'est cependant pas suffisante pour permettre de tirer pleinement parti du modèle. Certains outils comme le modèle Gluons [Pintado93] ne proposent que la composition fonctionnelle et d'autres comme Vista [Mey 95], Bric [Ferber 95] et dans une moindre partie Cell [Konstantas 92] offrent une possibilité supplémentaire de composition qui n'est plus associée aux communications mais à la structure du composant, c'est la composition structurelle.

Composition structurelle

Le principe de composition structurelle est relativement simple : plusieurs composants peuvent former un nouveau composant. Vu de l'extérieur, c'est à dire des autres composants du système, il n'y a qu'un seul composant auquel on s'adresse, alors que dans la réalité, ce composant est composé de plusieurs sous composants. Cette composition s'apparente à la composition fonctionnelle tout en permettant de cacher les sous composants et donc de les rendre inaccessibles. En fonction de la granularité des composants, on parlera alors de composants *simples* pour les composants de base (composants atomiques) et de composants *complexes* ou *composites* pour les composants « composés ». Nous reviendrons sur la composition dans le chapitre consacré aux exemples d'applications.

III.3.2.3 INTERACTIONS

Concevoir une application consiste à faire interagir plusieurs composants, pour cela il faut définir des mécanismes de communication qui soient compréhensibles par l'ensemble des composants du système.

Avant de définir ces mécanismes de communication, un composant doit permettre de connaître et d'identifier les fonctionnalités qu'il propose. Ceci se fait par des bornes (plugs, ports) d'entrée/sortie. De même que dans l'approche orientée objet, un objet est défini par un ensemble de méthodes dont seulement quelques unes seront accessibles (définies dans la partie interface de l'objet), dans l'approche à base de composants, les bornes servent à déterminer les fonctions (noms de méthodes, types et nombre de paramètres). Sur la Figure 13, les bornes 'accessibles' sont représentées par les carrés noirs. Ceux qui sont à l'intérieur représentent alors les bornes d'entrées qui permettront à un composant de recevoir des données, et les carrés à l'extérieur représentent les bornes de sorties par l'intermédiaire desquelles le composant pourra émettre ses données. Ainsi défini il est possible de savoir ce qu'un composant propose et ce dont il a besoin.

Faire communiquer deux composants revient alors à associer des bornes de sortie de l'un (l'émetteur) à des bornes d'entrées de l'autre (le récepteur). C'est ce que l'on nomme la composition fonctionnelle. Il faut alors pouvoir d'une part spécifier ces associations et d'autre part effectuer le transfert de données de l'émetteur vers le récepteur.

Pour spécifier les associations, les outils de conception les plus courants définissent la notion de connexion[21] (Cf. Figure 13) [Mey93][Mey95][Konstantas92][Ferber95], chaque connexion représente un canal de communication mono ou bidirectionnel[22] entre une Borne de sortie et une Borne d'entrée. Certains outils [Pintado93] permettent de vérifier si les deux bornes en relation acceptent les même types de paramètres. D'autres [Ferber 95] proposent d'associer à une borne de sortie plusieurs bornes d'entrées afin de permettre l'envoi d'un même message à un ou plusieurs récepteurs ce qui permet de modéliser des envois de message de type Point à Point, Broadcast ou Multicast [Fisher 95][Moffat 95]. Dans Maleva la vérification de la concordance des paramètres et le modèle d'envoi de message est géré par un objet spécial, le Gestionnaire de Message.

---

[21] le terme connexion se retrouve sous les termes de lien, link ou encore canal de communication.

[22] Suivant l'application.

Les mécanismes d'envoi effectif des données sont propres à chaque système mais en règle générale - un composant étant vu comme un processus - il s'agit d'un envoi de message asynchrone bufferisé. Dans Maleva nous avons opté pour une plus grande souplesse, le Gestionnaire de Messages permet tout type d'envoi de message (synchrone, asynchrone, asynchrone bufferisé).

III.3.3 EXEMPLES D'APPLICATIONS

Nous allons étudier quelques applications tournant autour du modèle componentiel. En particulier nous étudierons les modèles introduits dans l'environnement Vista, dans le modèle Cell, dans le modèle Bric et pour finir dans la plate-forme Maleva.

III.3.3.1 VISTA

Vicky de Mey [Mey93] [Mey95] a développé un environnement pour la conception de systèmes à base de composants nommé Vista. Cet environnement traite de quatre aspects du modèle à base de composants: le composant, la composition, les interactions utilisateur et la gestion des composants. Un composant y est vu comme étant le couplage d'un comportement et d'une présentation (interface). On retrouve alors, ce qui est une définition correcte des composants de base, la notion de comportement interne, et la notion de bornes d'entrées/sorties (celles ci étant nommées ports externes). Le comportement est responsable à la fois de ce pour quoi il a été conçu et de la gestion des compositions, l'interface est responsable de la représentation d'un composant (i.e. son apparence visuelle). Le composant est défini par un ensemble de ports externes qui représentent l'ensemble des fonctions accessibles par les autres composants. La composition fonctionnelle se fait alors par des canaux de communication définis par des ports actifs et des liens.

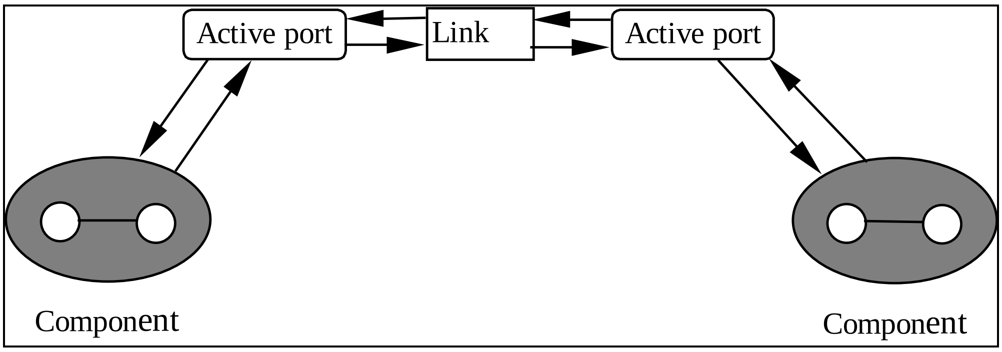


Figure 14: Vista, Composants et Communications

Sur la figure précédante nous voyons deux composants (association Comportement + Présentation) communiquant par l'intermédiaire de ports (Active Port) reliés par des liens (Link). Nous voyons ici la notion de canaux de communications que nous retrouverons dans Maleva sous la forme de Connexions.

Dans le modèle Vista, un composant composite est défini par des bornes d'entrée/sortie que l'utilisateur peut sélectionner parmi les ports des sous composants. L'utilisateur peut alors définir des composants composites de nature différente bien qu'étant structurellement identiques. L'auteur ne spécifie pas si un port déjà utilisé par deux sous composants peut être accessible de l'extérieur (ce qui revient à rendre possible les connexions multiples sur un même Port).

Les modèles à base de composants ne sont pas uniquement utilisés pour la conception d'applications de type IHM ou Multimédia, certains travaux étudient des modèles apparentés pour la conception d'applications fortement distribuées ou pour la conception d'agents et de SMA.

III.3.3.2 Cell

Le système Cell [Konstantas92] est caractérisé par la notion de cellule qui remplace celle de composant, ce modèle est basé sur la distinction entre le noyau qui représente l'ensemble des objets locaux appartenant au même espace d'adressage, et la membrane qui est responsable de la vision externe des objets exportables. Le noyau s'apparente alors au comportement interne d'un composant et la membrane à la vision externe que l'on a du composant. Le modèle propose bien sûr des ports de communication permettant aux membranes de communiquer entre elles. Ce que l'auteur propose dans son système c'est la possibilité pour les cellules d'importer (exporter) des objets qui augmenteront la taille du noyau, la membrane rendant les nouveaux objets accessibles de l'extérieur. L'objectif principal de ce modèle est de rendre accessibles des objets provenant de machines distinctes et/ou conçus dans des langages différents.

III.3.3.3 Bric

Le modèle Bric développé par J. Ferber [Ferber95] est une approche des systèmes à base de composants (Bric veut dire Block-like Representation of Interactive Components) pour la conception d'agents et de SMA. La conception d'une application (Agents et SMA) correspond à une approche Bottom-Up par la réutilisation et l'augmentation des composants existants. Comme dans les modèles précédents il est fait une distinction entre le comportement interne du composant et sa représentation externe. Le comportement d'un composant simple peut être décrit à partir de réseaux de Pétri, RDP simples, colorés avec arc inhibiteurs, la représentation externe permet de faire correspondre aux entrés/sorties du RDP des bornes de communication. La composition fonctionnelle reprend la notion de canaux de communications où peuvent migrer des RDP. La composition structurelle de plusieurs composants se fait par la création d’un composant englobant qui reprend les bornes inutilisées de ses sous composants (Cf. Figure 15).

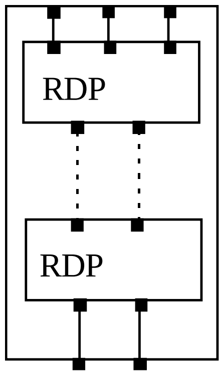


Figure 15: Composants Bric et composant englobant

A l’inverse des modèles présentés ci-dessus, où les connexions sont bidirectionnelles, le modèle Bric impose des connexions orientées de l’émetteur vers le récepteur. Autrement dit une requête est représentée par deux connexions, une pour la requête proprement dite et l’autre pour la réponse (ceci est dû au formalisme des RDP). L’intégration d’applications externes (par exemple tableurs, traitement de textes ...) impose l’utilisation de composants permettant le passage d’un type de connexion à un autre. Pour ce faire, J. Ferber introduit des composants primitifs qui se différencient des composants simples par le fait que leur comportement n’est pas décrit par des RDP.

III.3.3.4 MALEVA

Dans les modèles présentés, un composant peut être vu comme un processus autonome qui traite des données en entrée et produit des données en sortie. La disponibilité des données en entrées d’un processus (d’un composant) est suffisante pour en déclencher l’activité. La conception d’une application revient alors à décrire les flots de données entre les processus. Or les modèles de flots de données ne peuvent être appliqués aux systèmes où les données sont présentes de façon continue [Perez90] (par exemple la présence d’un agent prédateur à coté d’un agent proie), dans un tel cas, le composant qui doit traiter cette donnée boucle indéfiniment. Pour résoudre ce problème, J. Ferber propose un synchroniseur global (accessible de l’ensemble des agents) pour la perception (Sync. Perceptions) et pour l’action (Sync. Réactions) les activités intermédiaires fonctionnant en parallèle. L’utilisation d’un synchroniseur (Cf Figure 9) est séduisante, nous en définissons d’ailleurs un dans Maleva, mais elle n’est pas suffisante.

Nous avons opté dans Maleva pour une approche par distinction Donnée/Contrôle, chaque composant n’est activé que si il reçoit un contrôle externe (d’un autre composant) ou interne (d’un Scheduler), la conception de l’application revient alors à spécifier les flots de données **et** les flots de contrôle.

Notre approche permet alors de résoudre d'une part le problème : « Quel composant doit être activé et quand doit-il être activé ? » et d'autre part d'un problème non spécifié dans les modèles précédents : « Comment gérer la multiplicité des composants en amont ou en aval d'un composant donné ? ». Ce problème apparaît, par exemple, lorsque nous voulons utiliser plusieurs capteurs (Capteurs d'environnement) pour un seul comportement (par exemple un comportement de fuite). Dans ce cas, un composant doit pouvoir soit prendre en compte l'ensemble des données soit faire un choix entre ces données. Le chapitre suivant montrera notre approche à partir d'un exemple simple.

Le Composant Maleva

Un composant Maleva est donc une entité définie par un comportement interne, par un ensemble de bornes d'entrées, par un ensemble de bornes de sorties, et par un gestionnaire de message.

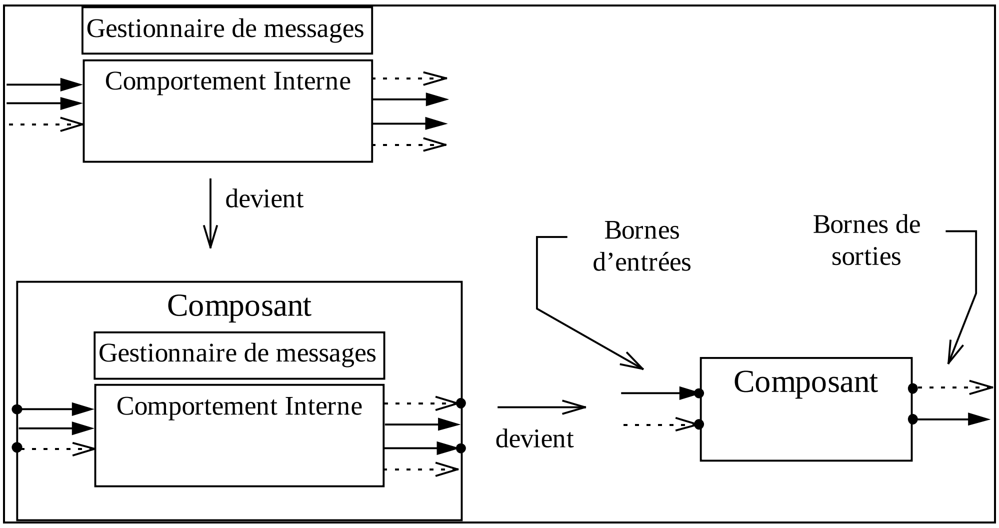


Figure 16: Un composant Maleva

En terme objet, le comportement est décrit par une classe (classe TSMAComponent) qui est responsable du comportement du composant et de la gestion des sous composants. Il définit des méthodes particulières représentant ses bornes d'entrées sorties (Figure 16 en haut). Ces méthodes seront ensuite indiquées au gestionnaire de message (Figure 16 en bas à gauche), lequel les rendra accessibles aux autres composants (Figure 16 en bas à droite).

Les bornes d'entrées/sorties sont de deux natures différentes, en trait continu sont représentés les flots de données et en pointillé sont représentés les flots de contrôle. Un composant Maleva n'est pas vu comme un processus autonome, il n'est activé que s'il reçoit un contrôle. Celui-ci peut être un contrôle externe (les flots de contrôle) ou par un contrôle interne (un contrôle est transmis périodiquement par un Scheduler qui gère l'ensemble des composants du système).

Le gestionnaire de messages est responsable de la vision externe du composant (il gère les bornes). Il est en outre responsable de la gestion des messages (données ou contrôles) qui transitent à travers ces bornes et il est enfin responsable de la gestion des canaux de communications entre différents composants (les connexions).

L'utilisation d'un gestionnaire de message indépendant du comportement permet, pour une même classe de comportement, avoir différents types de gestionnaires de messages et inversement pour un même gestionnaire avoir différentes classes de comportement (cas le plus courant).

Interactions

Les interactions entre composants se font par envoi de messages, la nature de ceux-ci (synchrone, asynchrone, asynchrone bufferisé) dépend des gestionnaires de messages associés aux composants qui communiquent. Chaque message transite d'un émetteur vers un récepteur par un canal spécifique nommé Connexion. L'ensemble des connexions entre deux composants représente alors le flot de communication. La notion de connexion se retrouve dans les travaux de V. de Mey avec la notion de Link, dans ceux de X. Pintado [Pintado91] avec ses Gluons.

Une connexion représente une structure de donnée qui possède au minimum les champs émetteur, récepteur, borne de sortie, borne d'entrée et paramètres.

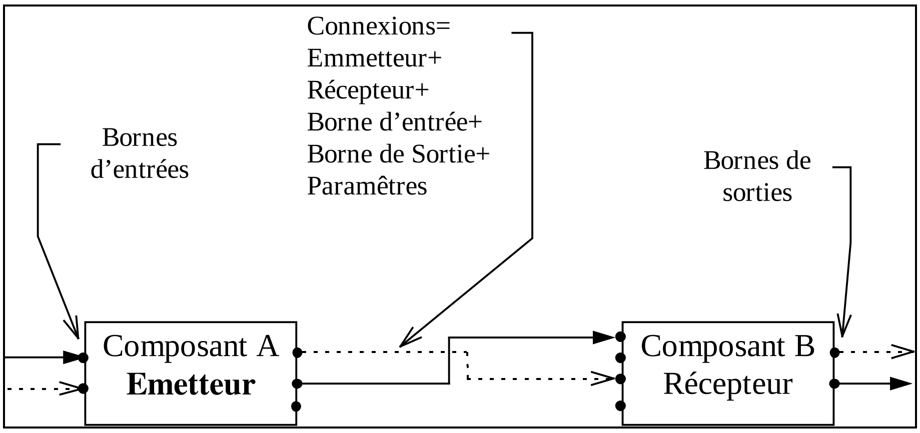


Figure 17: Connexions: flot de données/ contrôles

Ce sont ces connexions que l'utilisateur doit spécifier lorsqu'il conçoit l'application. Il a alors à sa disposition le moyen textuel (une méthode MakeConnexion) ou, de manière visuelle, par l'utilisation d'un des outils (boites à liste) de Maleva qui permet d'associer les bornes entre elles.

Composition de composants

Dans Gluons, il n’y a pas de composition structurelle car seules les interactions (donc les connexions) permettent de définir des groupes de composants (composition fonctionnelle). Il n’y a donc pas de composant complexe.

Dans Vista, les concepteurs définissent des composants composites qui sont l’équivalent d’un composant Complexe. Un composant composite est défini par des bornes d’entrée/sortie que l’utilisateur peut sélectionner parmi les bornes des sous composants. Le comportement du composant Complexe devient alors celui de ses sous composants.

Dans Maleva, les composants Simples sont conçus dans le langage de la plate-forme (ici Delphi) par simple héritage de composants génériques de base et grâce à un outil de génération de code. Il faut définir les bornes d’entrées et de sorties du composant et après génération du squelette de la classe, définir le comportement du composant. Chaque nouvelle classe de composant est alors compilée et enregistrée dans une base. Le développeur pourra alors s’il le désire faire apparaître son composant dans une palette de composants, il deviendra alors accessible aux utilisateurs.

Les composants complexes sont conçus en 2 étapes, la première est l’étape de conception visuelle (fonctionnelle) qui permet d’obtenir des modèles de composants. Définir des modèles revient alors à ajouter des composants (les sous composants) dans un composant englobant (super composant) et de définir les connexions entre les bornes d’entrées et de sorties. Ce modèle est un prototype de composant, on peut alors instancier autant de prototypes que l’on désire et les tester dans la simulation, il s’agit là en quelque sorte d’une validation du prototype pour un problème donné.

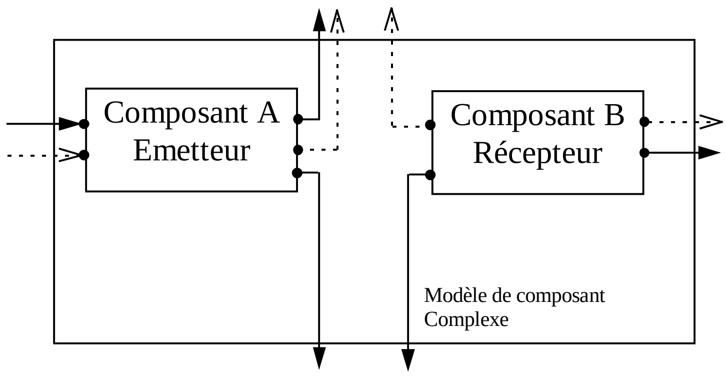


Figure 18: Composition de composants

Cependant le modèle de composant n’est pas tout à fait un composant à part entière dans la mesure où il ne définit pas ses propres bornes mais permet l’accès à la totalité de celles de ses sous composants. Les sous composants sont visibles et accessibles depuis l’extérieur ce qui va à l’encontre du principe d’encapsulation. Afin de faire correspondre à ce modèle un véritable composant nous devons passer par une deuxième étape qui est l’étape d’agrégation.

L'agrégation de composants

L'agrégation est l'étape qui consiste à partir d'un modèle fonctionnel de composant d'obtenir un modèle structurel. Cette étape est réalisée par un outil qui permet fonctionne en trois phases

Phase1 : On identifie les bornes des sous composants que l'on désire rendre accessibles et on crée les bornes dans le super composant en les renommant (en effet les bornes des sous composants peuvent avoir des noms identiques, par exemple si il existe deux sous composants identiques)

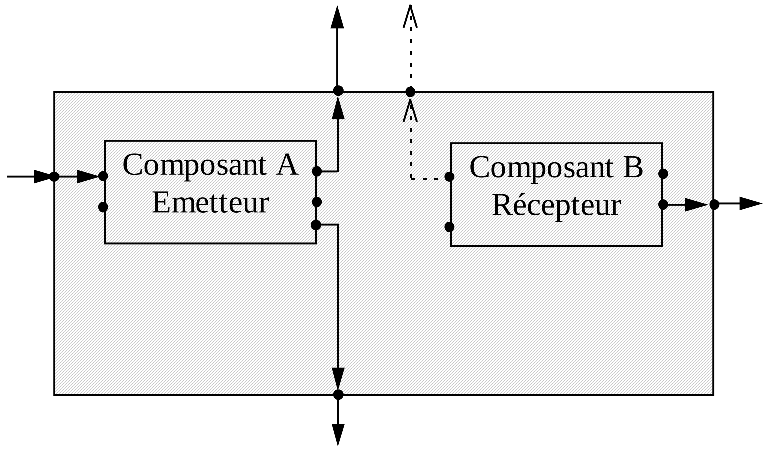


Figure 19: Agrégation de composants

Phase 2 : après la définition des bornes survient une phase de génération de code qui engendre une nouvelle classe de composant qui une fois compilée peut être intégrée à la palette des composants existants.

Phase 3 : dans Maleva, un composant composite n'est pas vu comme une coquille vide dont le seul comportement résulte de l'ensemble des comportement de ses sous composants. Afin d'augmenter l'indépendance de ce composant vis-à-vis des autres composants du systèmes, et afin de lui donner une réelle autonomie, Maleva lui permet d'avoir son propre comportement (attributs, méthodes...) et comme pour tout composant, il possède ses propres bornes d'entrée/sortie.

III.4 DES INTERACTIONS ENTRE LES COMPOSANTS

Les composants doivent pouvoir communiquer entre eux. Comme dans tout modèle objet ou acteur cette communication s'effectue par envoi de message. Il faut cependant spécifier la nature des messages

Synchrones/Asynchrone et le mode de communication Point à Point, BroadCast ou MultiCast.

III.4.1 MESSAGES SYNCHRONES/ASYNCHRONES

Maleva étant basé sur un modèle Objet et Acteur, les composants communiquent par envoi de messages Synchrones ou Asynchrones suivant le besoin. Les connexions entre les composants permettent de définir le type d'envoi de message qui sera effectué.

En mode Synchrone, l'émetteur est bloqué jusqu'au retour du contrôle tandis qu'en mode Asynchrone, l'émetteur garde son contrôle et peut continuer son activité.

III.4.2 UN MODE DE COMMUNICATION

Le mode de communication entre les composants peut être de 3 types (point à point, broadcast et multicast), il ne diffère pas du mode de communication entre Agents tel que ceux décrits par Fisher (Cf. Fisher 95).

III.4.2.1 LE POINT A POINT

Le mode de communication Point à Point est le mode de communication le plus usité. Il présuppose deux interlocuteurs, l'émetteur et le récepteur. Lorsque l'émetteur envoi un message il doit nommer explicitement le récepteur.

Il s'agit du mode de communication utilisé dans les langages Objets et Acteurs classiques. Le fait même de nommer le récepteur impose que celui-ci existe et qu'il soit à même de comprendre le message. Autrement dit, si le récepteur n'existe plus (il a disparu), ou si il n'implémente pas la méthode appelée, le système ne fonctionne plus.

Suivant le langage utilisé, ce genre de problème peut être réglé de plusieurs manières. Dans Smalltalk, un appel de méthode inexistante provoque l'appel d'une méthode (doesNotUnderstand:)[23], qui se répercute sur le récepteur ou toute sa hiérarchie. Dans les Langages Objets compilés, au mieux le compilateur provoque une erreur, il faut alors mettre en correspondance la méthode appelée et la méthode exécutée. Au pire, l'application s'arrête sans aucune explication.

Le mode de communication Point à Point classique n'apparaît pas comme une solution acceptable dès lors que nous avons à envoyer (diffuser) un même

[23] Ce qui n'est rendu possible que par le fait que les messages soient réifiés dans des objets.

message à plusieurs récepteurs. Le même problème apparaît lorsque nous avons un nombre indéterminé (qui puisse être modifié en cours d'exécution) d'objets à faire communiquer.

III.4.2.2 LE BROADCAST

Le mode de communication BroadCast permet de faire communiquer un nombre indéterminé d'objets/agents. L'émetteur s'adresse à l'ensemble des récepteurs présents. Il peut communiquer de manière épidémique (les messages sont retransmis de proche en proche par chaque objet récepteur) ou alors envoyer les message à tous les autres objets du système (réseau totalement connecté).

C'est le mode de communication le plus souvent utilisé pour les système dynamiques ou flexibles [Moffat 95]

Il permet la dynamicité puisque contrairement au Point à Point, il n'est pas nécessaire de nommer le récepteur (on s'adresse à tout le monde).

Il est tolérant aux pannes, si un des récepteur venait à disparaître le système fonctionne de manière identique que le récepteur soit remplacé (par un récepteur équivalent) ou non.

Il présuppose que l'émetteur et les récepteurs parlent la même langue. On en revient au problème du Point à Point cité précédemment lorsqu'un objet récepteur n'implémente pas la méthode appelée.

Son plus grave défaut est essentiellement de devenir inefficace lorsque le nombre de récepteurs devient trop important et que ceux-ci ne sont pas concernés par les communications. De plus il n'est pas sécurisé [Fisher 95] puisque tout le monde à accès aux données transmises au travers des communications.

Le mode de communication BroadCast, bien qu'intéressant n'apparaît pas non plus comme une solution acceptable dès lors que nous voulons un système viable (sécurisé) et efficace (rapide).

III.4.2.3 LE MULTICAST

Le MultiCast apparaît comme un compromis entre le mode Point à Point et le mode BroadCast. Il s'agit d'un BroadCast dans un groupe. On restreint donc le nombre de récepteurs aux personnes « intéressées » par les messages. Le MultiCast est implicite si les membres du groupe appartiennent à ce groupe sans en avoir demandé effectivement l'appartenance. On peut par exemple appartenir à un groupe dès lors que l'on possède un type (un sous-type) commun[24]. Le MultiCast est explicite si les membres du groupe en ont demandé l'appartenance, par exemple si l'on désire s'abonner à tel ou tel message.

On n'est plus aussi exhaustif que le BroadCast et on devient compositionnel vis à vis du Point à Point.

Le MultiCast présuppose alors une notion de groupe. Tout récepteur doit faire partie du groupe de l'émetteur, ce qui ne l'empêche pas de faire partie de plusieurs groupe à la fois.

Il impose cependant de désigner un gestionnaire de groupe, l'objet qui va devoir gérer l'insertion, le retrait et l'interrogation des membres du groupe. Il impose aussi, et on en revient alors au problème du BroadCast, que les récepteurs comprennent l'émetteur.

Hormis ce problème de compréhension mutuelle, le mode de communication MultiCast semble être le modèle de communication le mieux adapté aux systèmes dans lesquels le nombre d'individus (Objets, Composants, Agents) varie au cours du temps. Il permet la flexibilité du système, il est plus efficace que le BroadCast même si la sécurité des informations repose sur un gestionnaire de groupe.

III.4.2.4 LE MULTICAST ET LES GROUPES DANS MALEVA

Les différents composants utilisés lors de la création d'un agent, sont conçus indépendamment les uns des autres (le plus souvent par des personnes différentes) et diffèrent par le nombre et le nom des fonctions qu'ils mettent à disposition, et ce même si ces composants veulent offrir les même fonctionnalités.

Les problèmes de syntaxe des fonctions doivent être traités dans le mode de communication.

Comme nous l'avons vu auparavant, ce mode de communication nécessite un gestionnaire pour le groupe.

Dans Maleva, il n'y a pas à proprement parler de notion de groupe et de gestionnaire de groupe. Ces termes sont naturellement remplacés par la notion de composition et celle de gestionnaire de communication. Un composant peut alors être vu comme le groupe de ses sous composants de même que le gestionnaire de communication du super composant comme le gestionnaire du groupe de ses sous composantes puisque c'est lui qui gère l'acheminement des messages.

---

[24] Le terme type commun fait référence à interface de communication commune [Gamma96].

Etant donné que le gestionnaire de communication est totalement dissocié du comportement, celui-ci suivant sa catégorie peut mettre en œuvre différentes techniques de diffusion de message. A ce titre nous avons conçu deux classes de gestionnaires. La première travaille uniquement en mode Point à Point, elle ne traite que la première connexion définie pour une borne donnée même si plusieurs connexions existent. La deuxième classe traite du BroadCast en assurant l'envoi effectif des messages à l'ensemble des destinataires effectivement connectés à la borne de sortie du client, et assure le retour et la mise en commun des résultats si besoin est.

IV EXEMPLES DE CONCEPTION COMPONENTIELLE

Dans ce chapitre nous allons voir comment à partir de composants pré-existants nous pouvons concevoir un agent simple et à partir de cet agent concevoir des agents plus évolués. Nous adopterons donc une approche ascendante de la conception d'un agent.

Nous appréhenderons les problèmes liés à la conception componentielle et principalement les problèmes de la gestion du contrôle. Ces problèmes sont dus pour la plupart à l'approche ascendante et au fait que les composants n'ont pas été conçus a priori pour ce type d'agent.

Nous montrerons les différents principes des systèmes componentiels et notamment la flexibilité, l'approche ascendante et incrémentale de la conception et enfin nous traiterons les problèmes de la gestion des flots de contrôle et de données avec l'introductions d'opérateurs de gestion.

IV.1 EXEMPLE DE CONCEPTION COMPONENTIELLE

Nous allons maintenant voir si il possible d'adopter une approche ascendante pour la conception d'un agent avec les problèmes que cela va engendrer.

Dans un premier temps nous étudierons les composants de base, par la suite nous verrons des agents simples conçus par simple utilisation des composants existants, enfin nous verrons des comportement plus complexes obtenus par mise en commun de comportements plus simples.

Pour illustrer ce chapitre, nous allons tenter de concevoir des agents correspondant à une simulation d'un proie/prédateurs simple dans un environnement rectangulaire fini et continu. Nous concevrons donc des agents qui fuient (les proies) et des agents qui suivent (les prédateurs). Nous ne développerons pas de stratégie de chasse comme dans la plate-forme Mages.

Ce que nous désirons obtenir ce sont des agents Proie dont le comportement corresponde à un comportement de fuite face à des prédateurs ou dont le comportement par défaut corresponde à des mouvements aléatoire.

Pour les agents Prédateur, nous désirons qu'ils suivent les proies quand ils les perçoivent, et il serait souhaitable qu'ils se repoussent entre eux (afin d'occuper un espace plus grand) sinon qu'ils adoptent un comportement de mouvement aléatoire.

L'exemple Proie/Prédateurs réactif est inspiré des travaux de R. E. Korf [Korf92]. Le livre de J. Ferber [Ferber 95] est une grande source d'inspiration. Il est montré qu'un comportement peut être vu comme une somme de comportements plus primitifs.

IV.1.1 DES COMPOSANTS DE BASE

Les composants de base que nous allons utiliser dans cet exemple sont des composants dont les fonctionnalités se retrouvent dans la quasi totalité des applications proposant des agents situés (des agents représentant des agents physiques et qui peuvent se déplacer). Ces composants seront donc soit des composants capteurs, soit des composants de comportement de base (comportements primitifs) soit des composants effecteurs.

Il est à noter que les composants Capteurs et Effecteurs sont étroitement liés non pas à l'agent que nous allons concevoir mais à l'environnement dans lequel l'agent va évoluer. Autrement dit, nos agents peuvent être placés dans différents types d'environnements par simple remplacement des composants capteurs et effecteurs. Ceci correspond alors à notre premier principe de flexibilité : un agent est une composition dont la structure n'est pas rigide, tout composant peut être remplacé par n'importe quel autre composant pourvu qu'il fournisse la même fonctionnalité.

Composants Simples

Capteurs

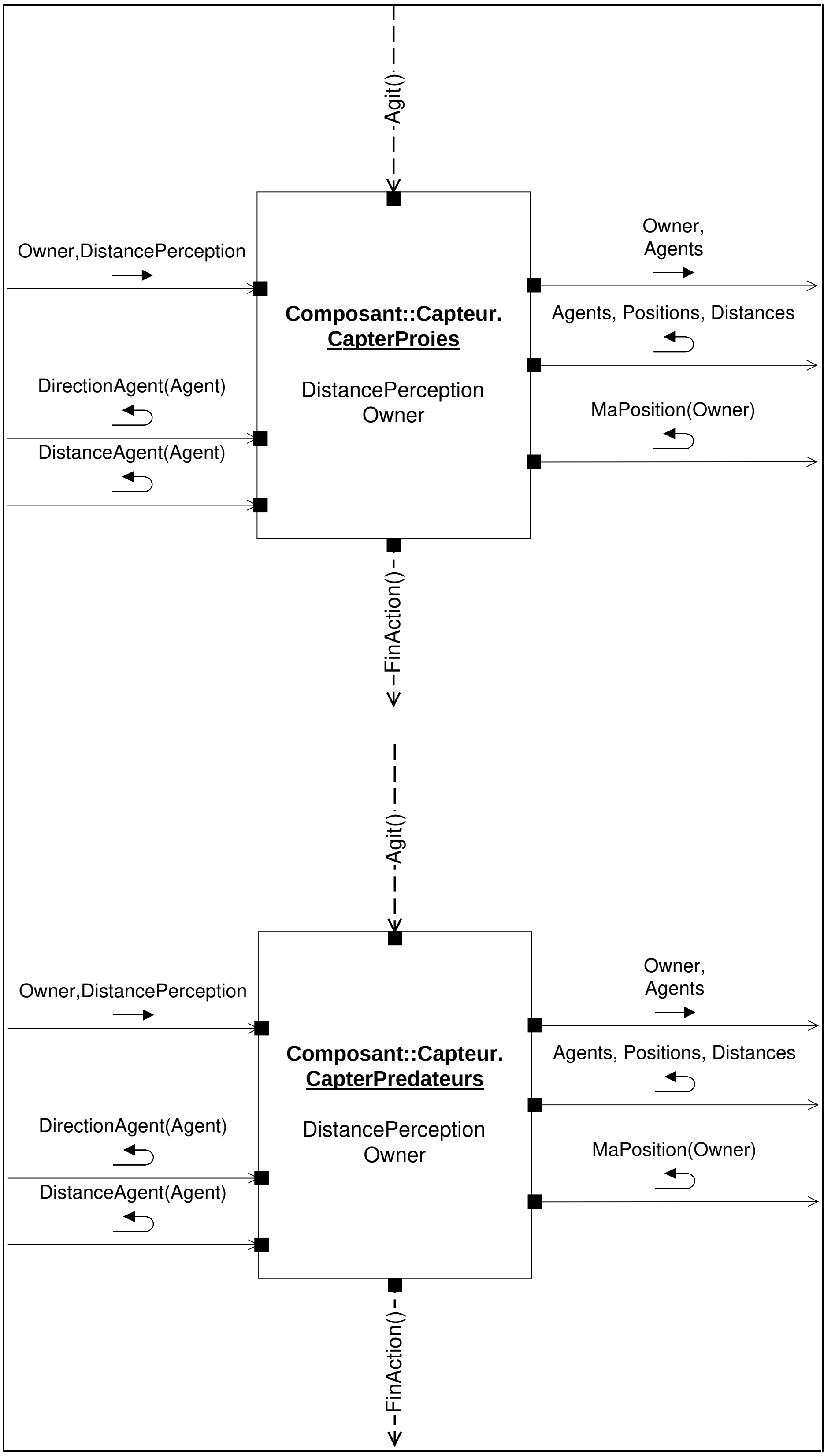


Schéma 1: Les Composants Capteurs

Nous avons deux types de capteurs associés aux types d'agent que nous allons simuler. Le premier capteur est un capteur de proie, le second est un capteur de prédateurs. Ces deux composants sont des instances particulières d'un composant générique capable de capter n'importe quel type d'agent.

Les diagrammes du Schéma 1 doivent se lire comme suit :

Nous avons un composant dont le rôle est de Capter des Proies (resp. Capter des Prédateurs). Il s'agit d'un composant passif (il ne s'agit pas d'un processus), il n'effectuera son traitement que s'il reçoit un contrôle sur sa borne nommée Agit(). Ce composant permet de connaître la direction et la distance d'une proie en interrogeant ses deux bornes DirectionProie et DistanceProie, on transmet la proie comme paramètre, le résultat est retourné lors de l'appel (appel de méthode classique avec retour de valeur). Enfin, ce composant retourne d'une part, la liste des agent détectés par l'intermédiaire de sa borne Agents, et d'autre part, le contrôle qui lui a été donné par sa borne nommée FinAction.

Le deuxième capteur fonctionne à l'identique du premier mais ne concerne que les agents prédateurs.

Composants Simples

Effecteurs

Dans cet exemple nous utiliserons un seul type d'effecteur. Celui-ci permet à n'importe quel agent d'agir dans l'environnement. Il s'agit d'un effecteur spécifique aux environnement continus (par opposition aux environnements discrétisés) qui permet dans ce cas précis à un agent de se déplacer dans une direction donnée et à une vitesse donnée, et de tuer un autre agent.

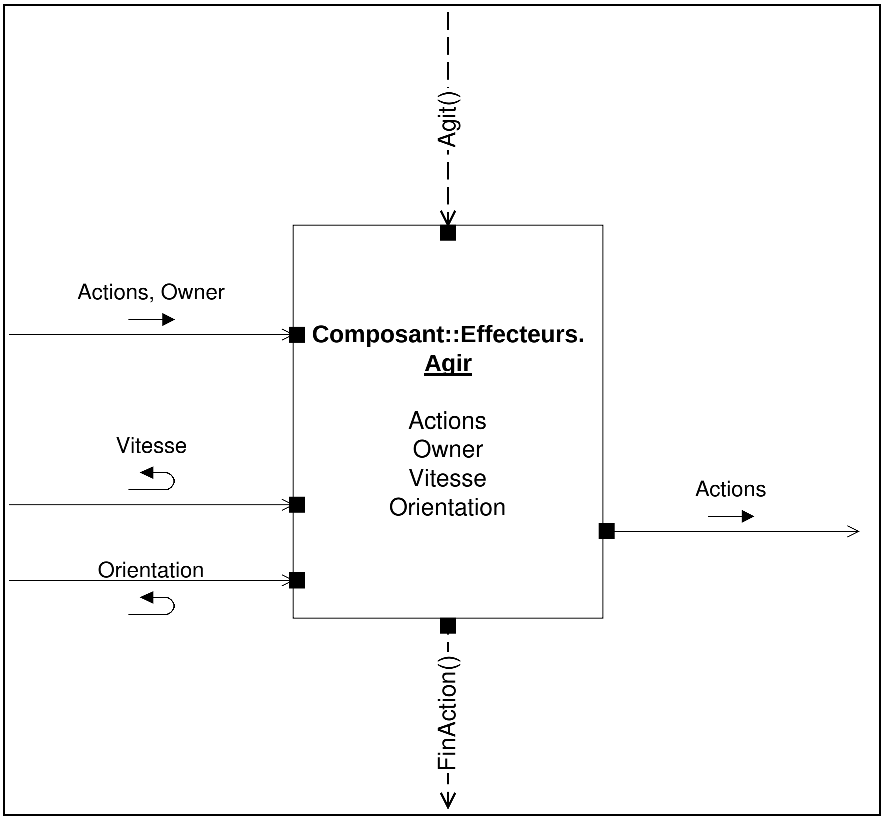


Schéma 2: Composant de Navigation pour Environnement Continu

Cet effecteur permet à un agent d'agir en lui transmettant l'ensemble des actions que l'agent désire effectuer, en contrepartie il agira sur ses paramètres Direction et Vitesse qui seront pris en compte au moment du déplacement effectif. Par souci de clarté, nous n'avons pas indiqué les flots entre Effecteur et environnement. Comme tout composant, l'effecteur transmet son contrôle à la fin de son traitement.

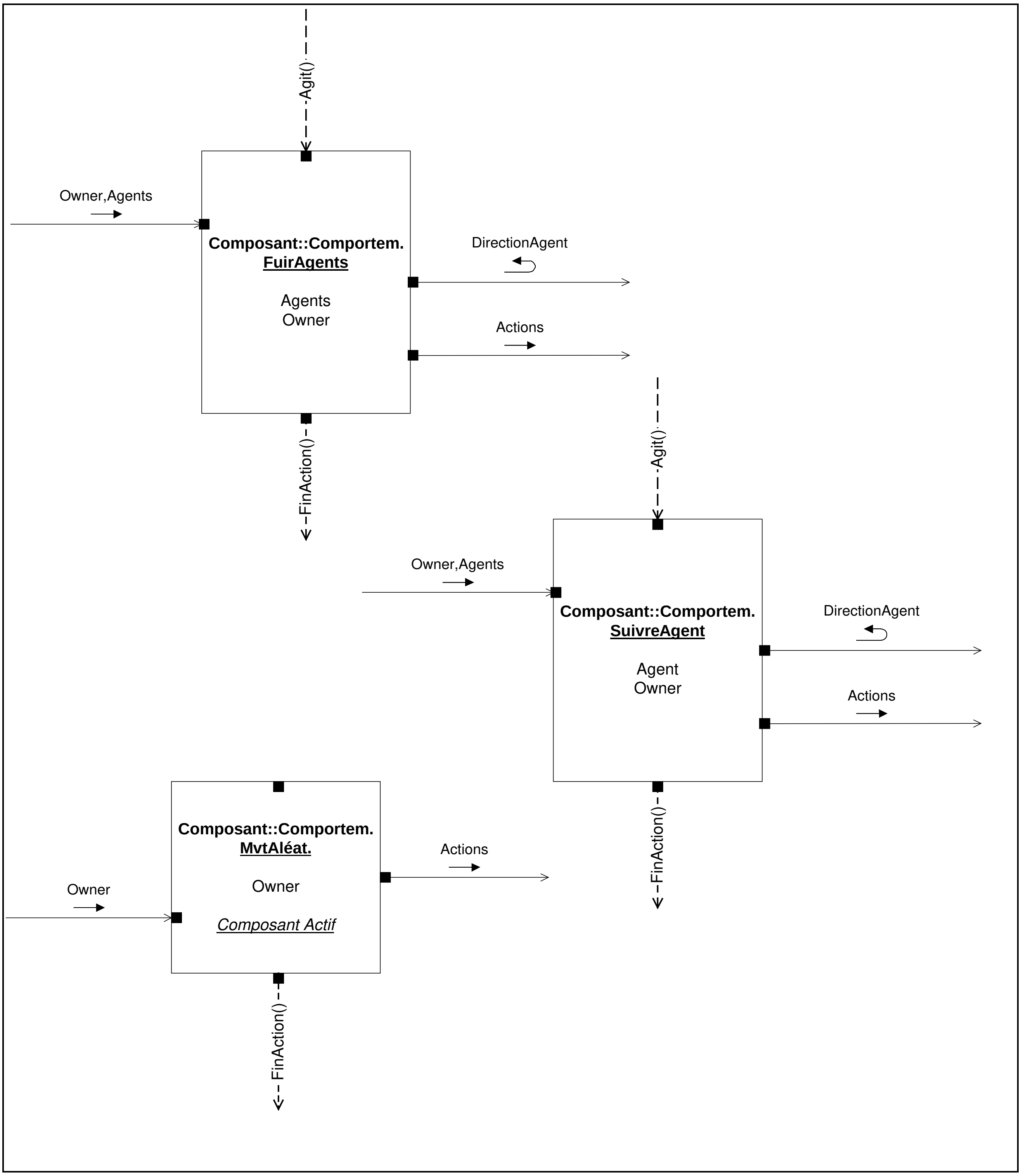


Schéma 3: Les composants Comportementaux basiques

Nous montrons ici quelques composants comportementaux de bas niveau tels que les comportements de fuite, d'attraction et de mouvement aléatoire. Les deux comportements de fuite et d'attraction ont besoin de données telles que la direction de l'agent (des agents) à fuir ou à suivre. Ces données sont envoyées par les composants Capteur et transiteront par des flots de données. Tout ces composants possèdent des paramètres internes leur permettant d'effectuer leur traitement.

Un comportement de fuite ou de suivi possédera des paramètres permettant de déterminer l'angle et la vitesse résultante de fuite ou de suivi. On peut en effet supposer qu'un prédateur aura tendance à suivre une proie plus qu'à fuir un autre prédateur et on ne fuit pas forcément en ligne droite.

Enfin un composant de mouvement aléatoire doit pouvoir contraindre les mouvement dans une direction donnée. On ne désire pas forcement avoir un mouvement brownien sur toute la surface mais par exemple vers le Nord avec un delta de +- 40°.

Ces paramètres ne sont pas constants pendant la durée de la simulation. Il faut par conséquent offrir à l'utilisateur la possibilité d'accéder et de modifier ces paramètres, d'où la nécessité d'une interface pour chaque composant. Chaque classe de composant ayant ses propres paramètres, il faut que l'interface soit suffisamment générique et modulaire pour admettre de nouveaux types de paramètres. Nous verrons dans le Chapitre IX un outil de gestion de l'interface utilisateur des composants.

IV.1.2 Des agents au comportement simple

Possédant les composants nécessaires à la conception de nos agents, nous allons maintenant les construire. En premier lieu, nous allons créer un agent qui suit une proie, puis un agent qui fuit des prédateurs et enfin un agent qui se déplace de manière aléatoire.

Pour créer nos agents simples, nous allons composer nos composants de base en reliant certaines bornes entre eux. Il s'agit donc ici de composition fonctionnelle uniquement. Nous verrons par la suite la composition structurelle.

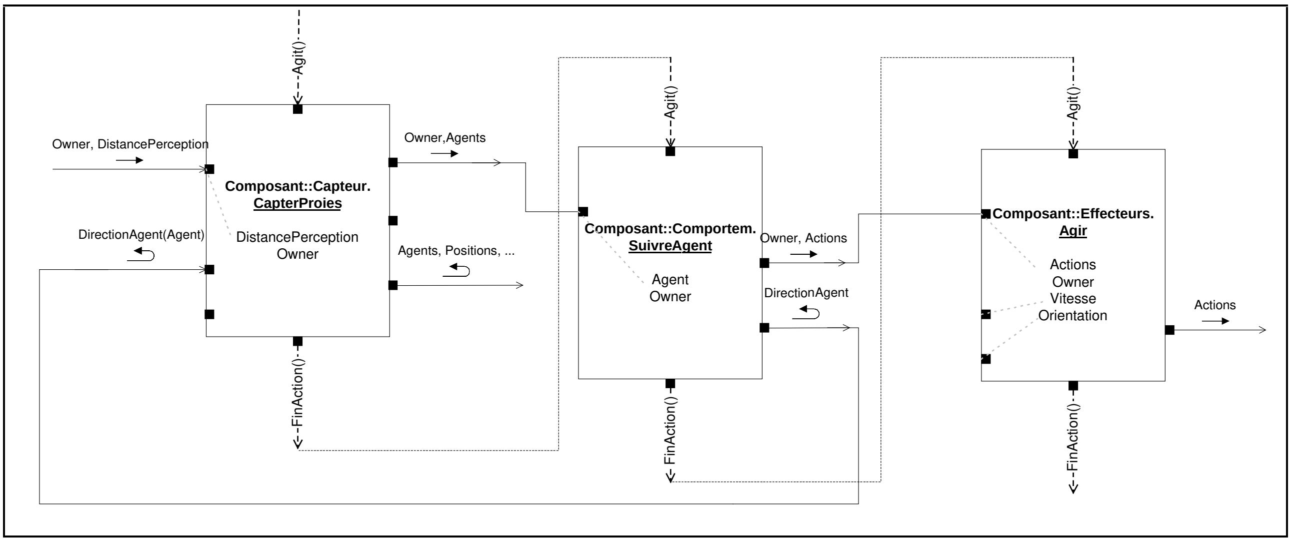


Schéma 4: Un agent qui suit une Proie

Sur le Schéma précédant, nous avons un "agent"[25] qui suit une proie. Nous avons connecté le Capteur de proie au comportement de suivi et le comportement de suivi à l'effecteur. Tous les composants sont passifs (Cadencement=0), nous avons donc un mécanisme séquentiel à l'intérieur de l'agent. Le contrôle est donné au Capteur par le Scheduler.

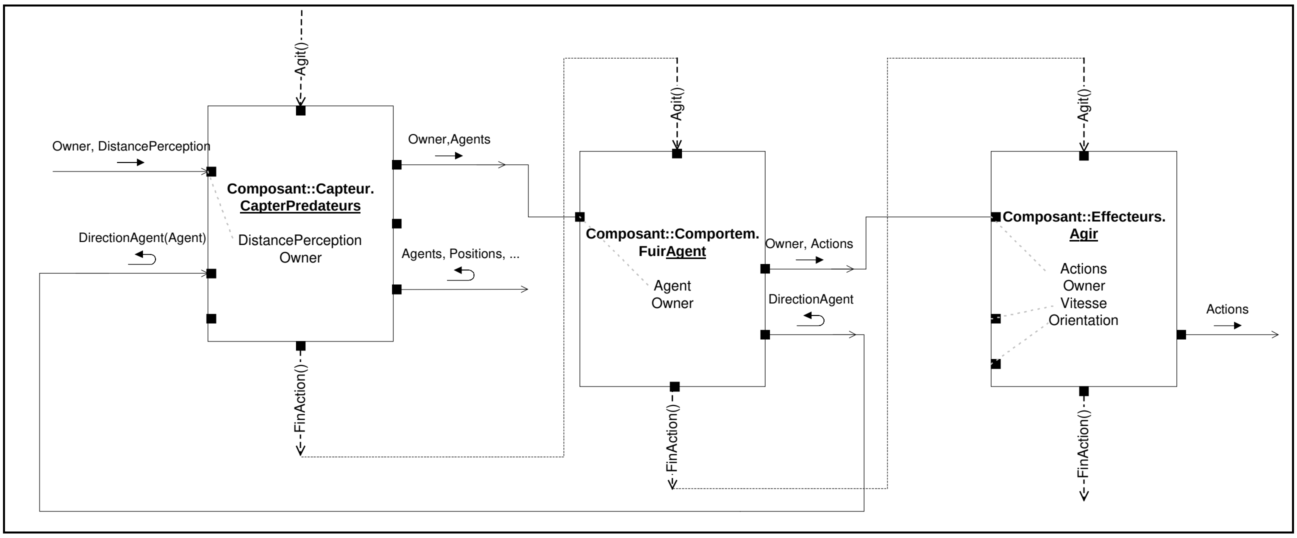


Schéma 5: Un agent qui fuit des prédateurs

Le même principe est appliqué pour concevoir un agent fuyant des prédateurs (Schéma 5). Nous voyons qu'à partir d'un même modèle à base de trois composants (Capteur + Comportement + Effecteur) nous pouvons concevoir deux types d'agents différents.

---

[25] Il ne s'agit pas réellement d'un agent, nous n'avons pas utilisé de composant Agent dans le schéma.

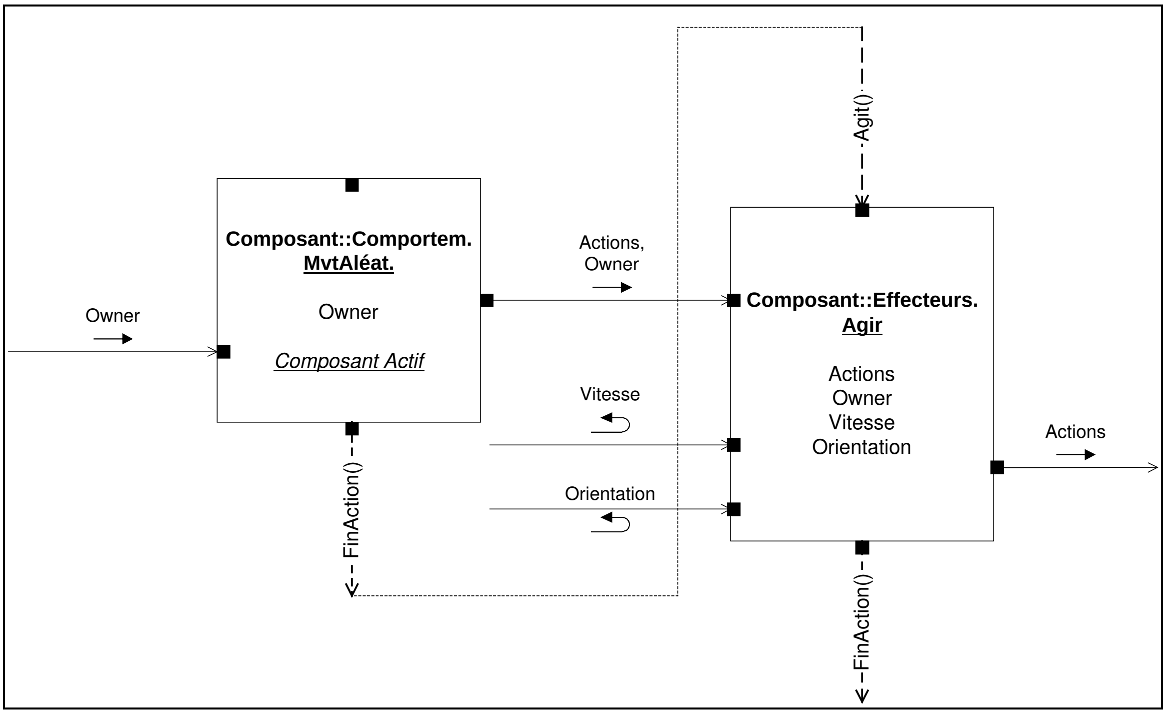


Schéma 6: Un agent qui se déplace aléatoirement

Enfin, le dernier agent est un agent qui ne perçoit rien (il ne possède pas de capteur), et son comportement consiste à se déplacer dans une direction choisie de manière aléatoire. Sur ce Schéma, nous voyons que cet agent n'a pas besoin de percevoir son environnement, il n'a par conséquent pas besoin de synchroniser sa perception vis à vis de l'environnement. Le composant de mouvement aléatoire peut alors être défini comme un Composant Actif. Ceci signifie que le composant fonctionne comme un processus sans attendre de recevoir de contrôle. Nous aurions pu faire de même pour les composants capteurs des deux agents précédants.

IV.1.3 DES AGENTS AU COMPORTEMENT PLUS COMPLEXES

Comme nous venons de le voir, il semble facile de concevoir des agents mono - comportementaux de manière rapide. Cependant bien que notre exemple ne requière pas d'agents aux comportements complexes, les agents que nous avons conçus n'ont pas de comportements suffisamment évolués et ne correspondent pas encore à ce que nous voulons simuler.

Le principe que nous allons montrer maintenant est qu'il est possible à partir des agents que nous avons déjà conçus de créer de nouveaux agents aux comportements plus évolués (ou moins restreints). Pour ce, nous avons deux (voir trois) possibilités à notre disposition

1. Première approche, pour obtenir des agents possédant des comportements plus complexes, nous pouvons remplacer les composants comportementaux que nous avons utilisés par des composants plus sophistiqués.

Nous pourrions en effet utiliser un composant comportemental permettant à la fois de suivre un certain type d'agent tout en permettant de fuir d'autres agents. Il faudrait que ce composant possède deux entrées pour les capteurs et sache différencier les proies des prédateurs. Il faut par conséquent que les agents proies et les agents prédateurs soient différentiables (i.e. qu'ils appartiennent à des classes différentes).

Cette approche est séduisante mais elle n'apporte rien à notre exposé et elle impose d'utiliser de nouveaux composants voir à en concevoir de nouveaux si ceux-ci n'existent pas. Or pour l'instant nous voulons utiliser uniquement ce que nous avons à notre disposition.

2. Deuxième approche, nous possédons les comportements nécessaires dans des composants distincts, nous pourrions les mettre en commun dans un seul agent.

Cette approche est beaucoup plus intéressante car elle permet de réutiliser des composants ayant déjà fait leur preuve. En effet nous avons déjà pu tester nos agents de base et voir si ceux-ci fonctionnent correctement dans l'environnement. C'est un des intérêt de l'approche ascendante que de pouvoir valider un modèle au fur et à mesure de sa conception.

La question revient alors à savoir comment associer différents comportements dans un seul agent.

3. Enfin nous avons une troisième possibilité comme étant la résultante des deux première, prendre les composants comportementaux et les mettre en commun dans un seul composant (un composant comportemental englobant) qui pourrait alors remplacer n'importe lequel d'entre eux.

Cette approche est attrayante car elle s'adapte bien à notre démarche incrémentale. Nous verrons par la suite qu'elle peut naturellement découler de la deuxième approche.

Nous concevrons d'abord nos agents plus complexes en prenant la deuxième approche puis nous reviendrons sur la troisième approche.

Avant de commencer la conception proprement dite de nos agents plus complexes nous devons avant tout nous attarder sur les problèmes de conception engendrés par l'approche componentielle, problèmes engendrés aussi par l'approche modulaire. Ces problèmes reviennent à traiter les flots de contrôle et les flots de données ce que nous allons étudier dans les paragraphes suivant.

IV.1.4 PROBLEMES DE COMPOSITION.

Si nous reprenons notre exemple de proie/prédateurs nous savons d'ores et déjà créer des comportements simples. L'étape suivante est donc d'associer différents comportements entre eux.

D'après l'énoncé nous voulons pour un prédateur qu'il suive une proie tout en fuyant ses congénères. Il s'agit là de ce que nous qualifierons d'un comportement additif. Ceci correspond en fait à prendre des données provenant de plusieurs composants et de faire un traitement sur l'ensemble.

En ce qui concerne la proie (et le prédateur) nous voulons qu'elle fuie les prédateurs si elle les perçoit sinon nous voulons qu'elle se déplace de manière aléatoire. Il s'ait cette fois ci d'un comportement que nous allons qualifier de comportement disjonctif. Ceci correspond à prendre des données d'un composant (fuite) ou celles d'un autre composant (mouvement aléatoire) mais pas les deux.

Comportements Additifs

Suivre Proie ET Fuir Prédateurs

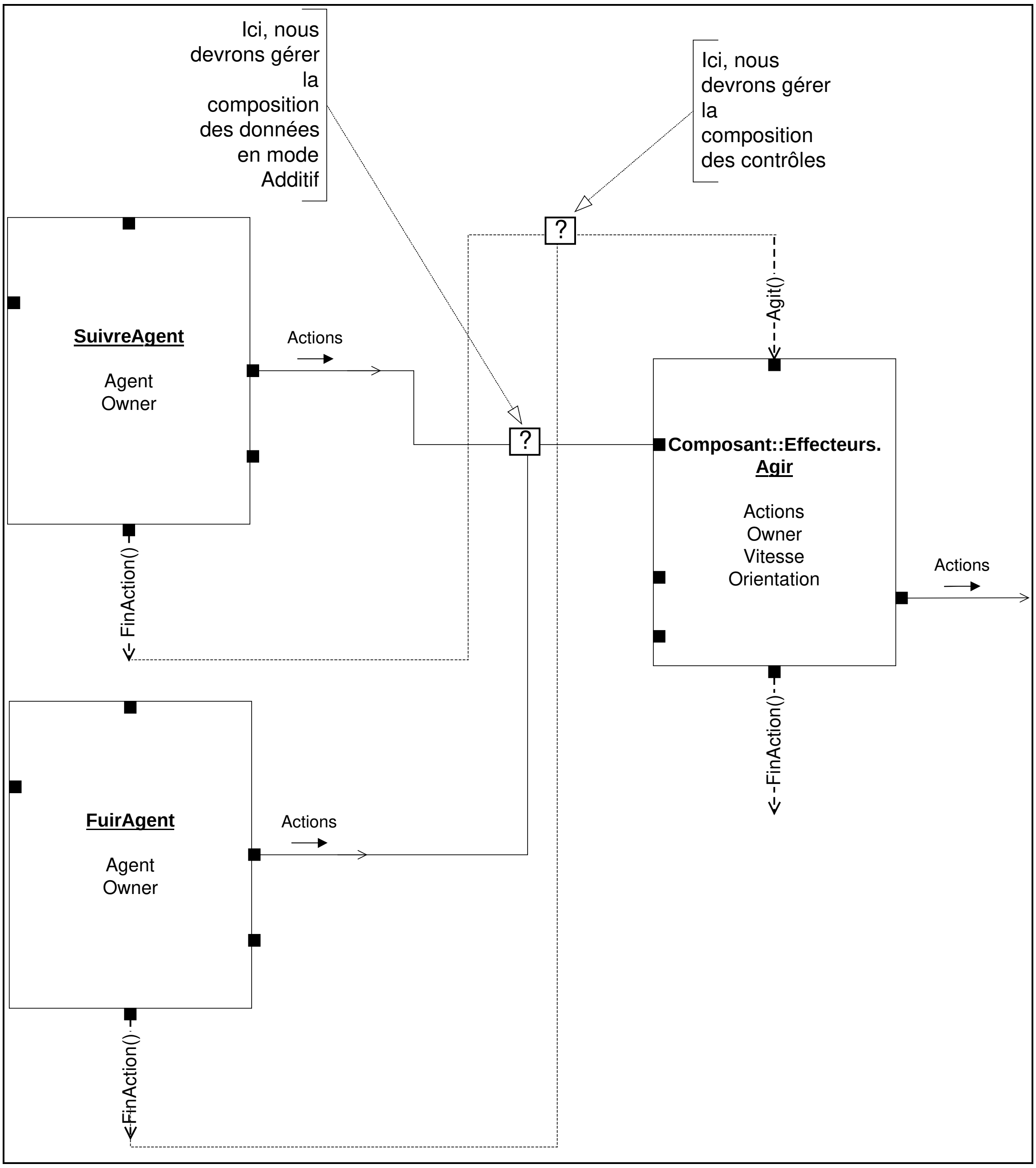


Schéma 7: Un agent qui suit un proie et fuit des prédateurs

Sur le schéma précédent, nous pouvons voir la mise en commun des deux comportements de fuite et d'attraction reprenant les

Schéma 4 et Schéma 5. Nous n'avons pas fait figurer les capteurs pour clarifier le Schéma. Nous avons mis en commun ces deux comportements en n'utilisant qu'un seul Effecteur.

Nous voyons donc sur le Schéma 7 que l'effecteur reçoit deux contrôles provenant des deux comportements. Nous oublierons pour l'instant le fait que, en temps normal, une borne d'entrée ne reçoit qu'une seule connexion. Si l'effecteur fonctionne de manière correcte, il devrait traiter le premier flot de données (générer des actions dans l'environnement) et ensuite traiter le second. Or notre but est qu'il traite les deux flots de données au même moment. Le problème que nous devons traiter revient à additionner les données provenant des composants comportementaux et par la suite transmettre le résultat à l'effecteur.

Comportements Disjonctifs

Fuir Prédateurs OU (absence de prédateurs) Mouvement Aléatoire

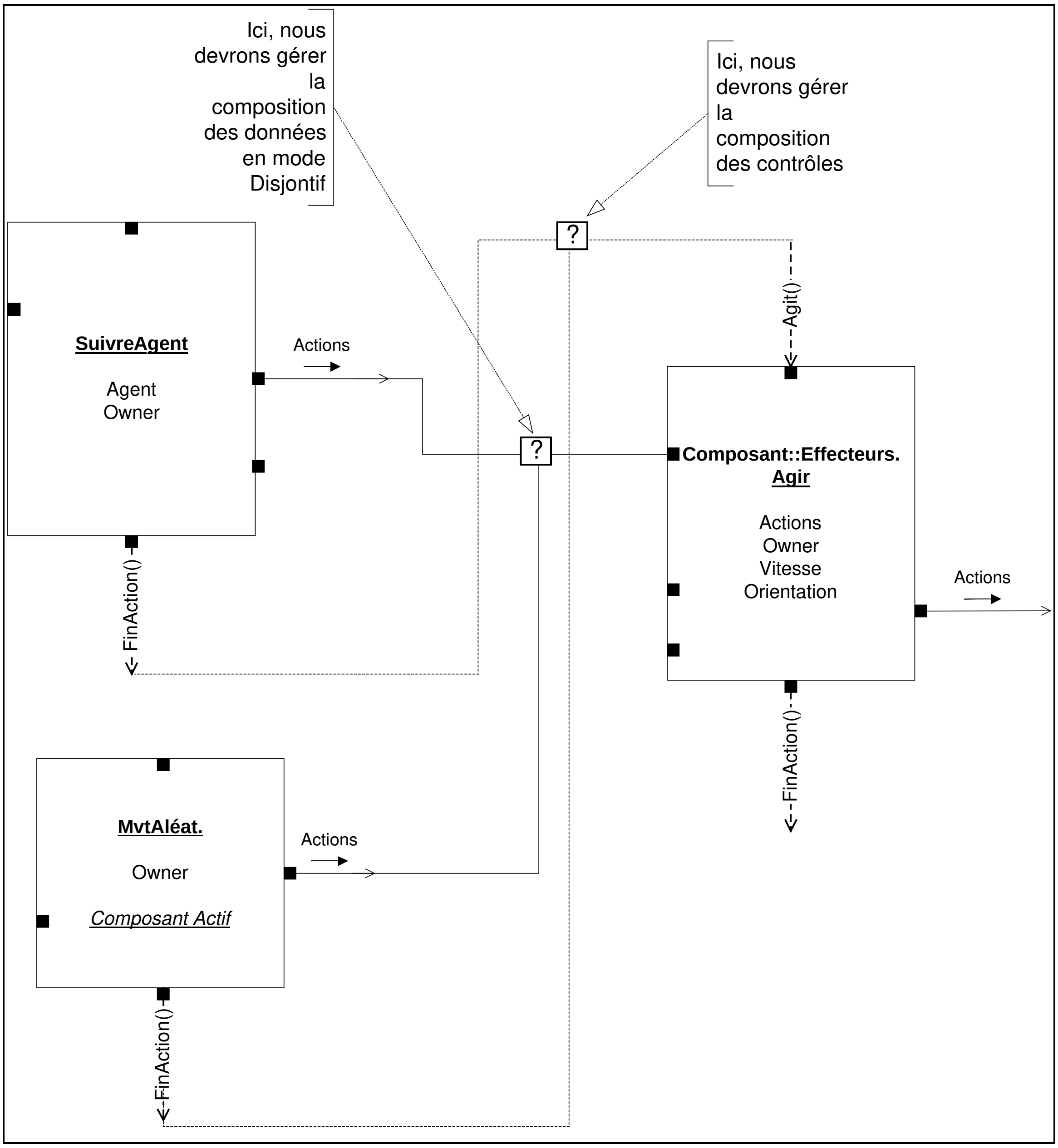


Schéma 8: Un agent qui fuit des prédateurs ou se déplace aléatoirement

Sur le schéma précédent, nous pouvons voir la mise en commun des deux comportements de fuite et de déplacement aléatoire reprenant les

Schéma 4 et Schéma 6. Nous voyons donc sur ce Schéma que l'effecteur reçoit deux données et deux contrôles provenant des deux comportements, cas que nous avons vus précédemment. A la différence du cas précédant, notre agent doit soit fuir soit bouger aléatoirement mais pas les deux. Si l'agent ne perçoit pas de prédateurs il fonctionne correctement car le comportement de fuite ne génère pas d'action mais s'il en perçoit il faut inhiber le comportement de mouvement aléatoire. C'est le deuxième problème que nous devrons gérer.

Traiter des flots de données et de contrôle est un problème difficile qui est souvent indiqué dans les articles [Moffat 95] mais très rarement détaillé. En fait le contrôle est souvent directement intégré aux architectures d'agents.

Dans les agents à architecture verticales, de type subsomption, le contrôle est géré par les priorités entre les modules. Dans les architectures verticales avec des taches compétitives, on ne prend en compte qu'un seul flot (celui dont la priorité est la plus forte).

Dans les architectures horizontales à base de tableaux noirs, ou les architectures hybrides [Guessoum 96], les contrôles sont traités par un module (un tableau noir, un automate à états finis...) spécifique et distinct. Ce module sert à régler les synchronisations entre les contrôles et à prendre en charge les priorités entre les flots.

Utiliser un mécanisme de priorité intégré aux modules ou utiliser un module annexe pour gérer l'ensemble des problèmes de contrôle revient à déterminer un fois pour toute la structure de l'agent or nous avons postulé dans notre modèle pour une structure flexible.

Nous avons donc choisi dans Maleva de poursuivre dans la continuité de notre approche componentielle et de concevoir des composants de gestion de flots de données et des composants de gestion de flots de contrôles. Ces composants à l'inverse des composants précités, ne règlent pas l'ensemble des problèmes de contrôle à l'intérieur d'un agent mais ils sont censés régler des problèmes locaux entre N composants en interactions. C'est le concepteur/utilisateur qui détermine et résout les problèmes de contrôle26.

IV.2 DES COMPOSANTS POUR LA GESTION DES DONNEES ET POUR LA GESTION DES CONTROLES.

Comme nous l'avons vu dans l'exemple précédent, le fait d'utiliser des composants pré-existants et le fait d'adopter une approche ascendante impose de gérer les flots de données entrantes concernant les mêmes informations et, par là même, de gérer les flots de contrôle entrants associés.

Typiquement, multiplier le nombre de composants en amont d'un composant qui traite les données résultantes (dans notre exemple un effecteur traitant des actions provenant de différents composants comportementaux),

[26] Le terme contrôle ainsi employé fait référence aux problèmes du contrôle des flots de données et des flots de contrôle.

impose de pouvoir gérer, d'une part le fait que nous recevons des données de même nature mais provenant de composants différents, et d'autre part le fait que ces données sont naturellement associées à un flot de contrôle.

Pour résoudre ces problèmes nous avons développé dans Maleva différents types de composants traitant des données ou des contrôles. Il s'agit en général de composants binaires (ils agissent comme des opérateurs) mais certains sont des composants N-Aires.

Nous n'expliciterons pas l'ensemble des composants de contrôle de Maleva, nous étudierons les plus importants et particulièrement ceux qui nous seront utiles pour notre exemple.

Le premier composant de contrôle que nous allons voir est un composant nommé Switch. Comme son nom l'indique (il s'agit d'un composant inspiré du composant électronique du même nom) il permet de commuter des données (et des contrôles) vers une sortie ou vers une autre.

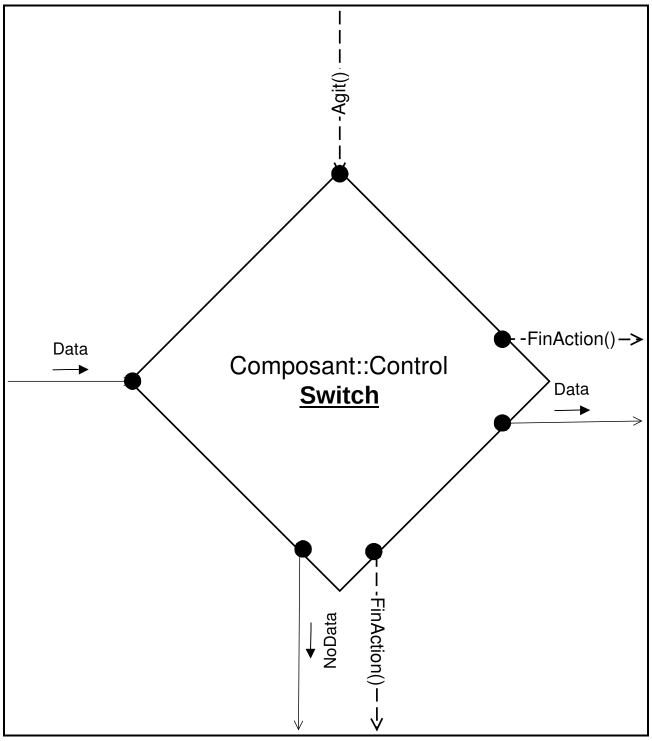


Schéma 9: Un composant Switch

Le principe de ce composant est simple :

C'est un composant qui lorsqu'il est activé traite la présence ou non de données (Data) en mémoire. Si des données sont présentes, il les transmet avec un contrôle (FinAction) vers la sortie à droite sinon il transmet un contrôle vers la sortie en bas. Ce composant permet donc de déterminer si des données sont présentes à un moment donné.

Le deuxième composant est un composant à deux entrées de contrôle et une seule sortie. Il s'agit bien entendu d'un synchroniseur. Les synchroniseurs sont des opérateurs qui ne s'activent que lorsqu'ils reçoivent l'ensemble des contrôles qu'ils sont censés recevoir, ils peuvent être couplés en cascade pour synchroniser N contrôles (un composant N-Aire étant composé de N-1 composants Binaires).

Afin de leur donner une autonomie nécessaire, nous leur donnons un cadencement à 1. Ils fonctionnent donc comme des processus.

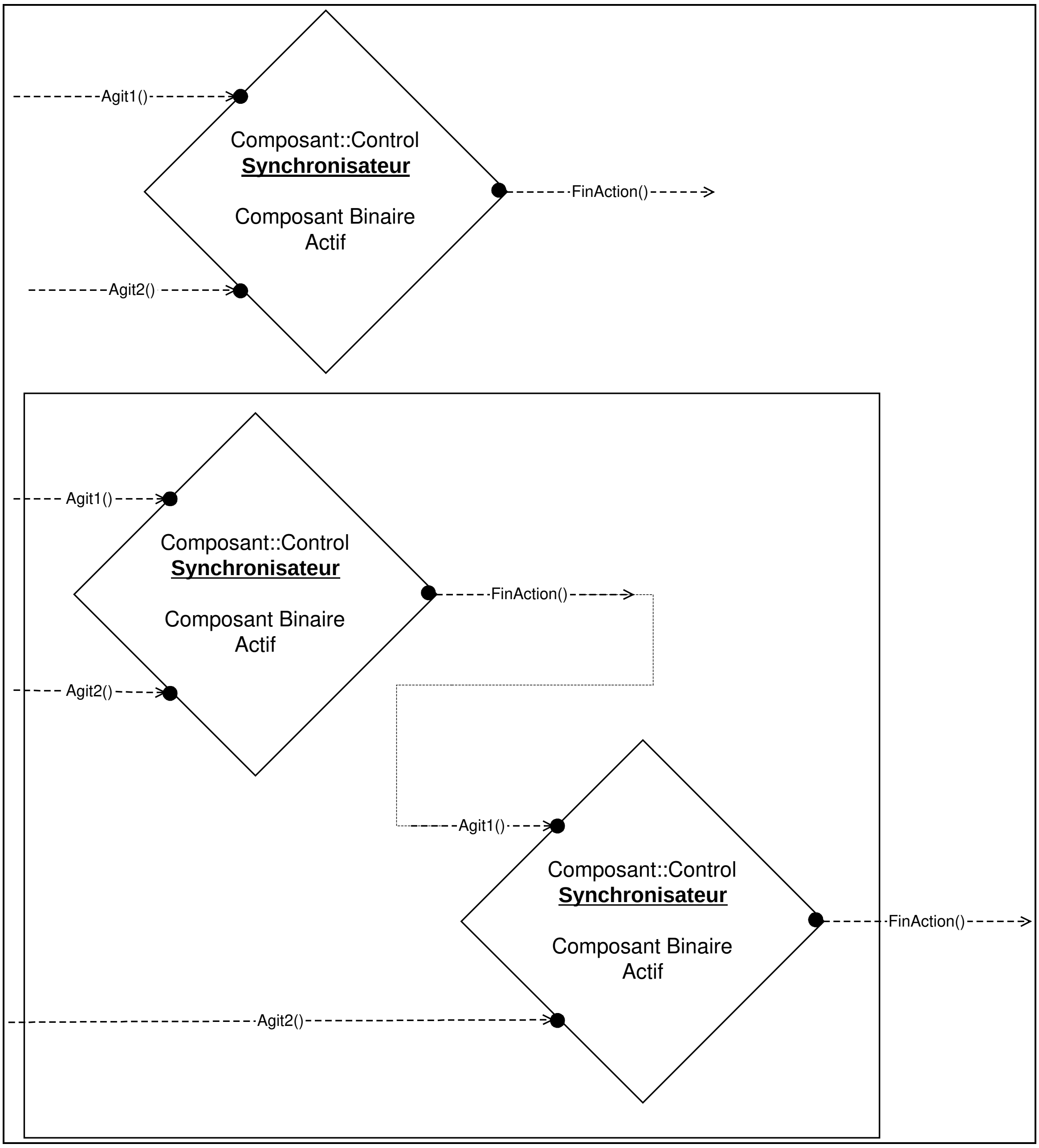


Schéma 10: Un composant Synchroniseur à 2 et à 3 entrées

Pour ajouter des données provenant de deux composants nous avons conçu un Composant Additionneur. Ce composant est un composant Actif qui dès qu'il a reçu les données sur ses deux bornes d'entrées transmet la somme sur sa borne de sortie. Ce composant transmet aussi un contrôle en sortie.

Pour envoyer des contrôles et/ou des données à plusieurs composants, nous avons réalisé un composant Répartiteur.

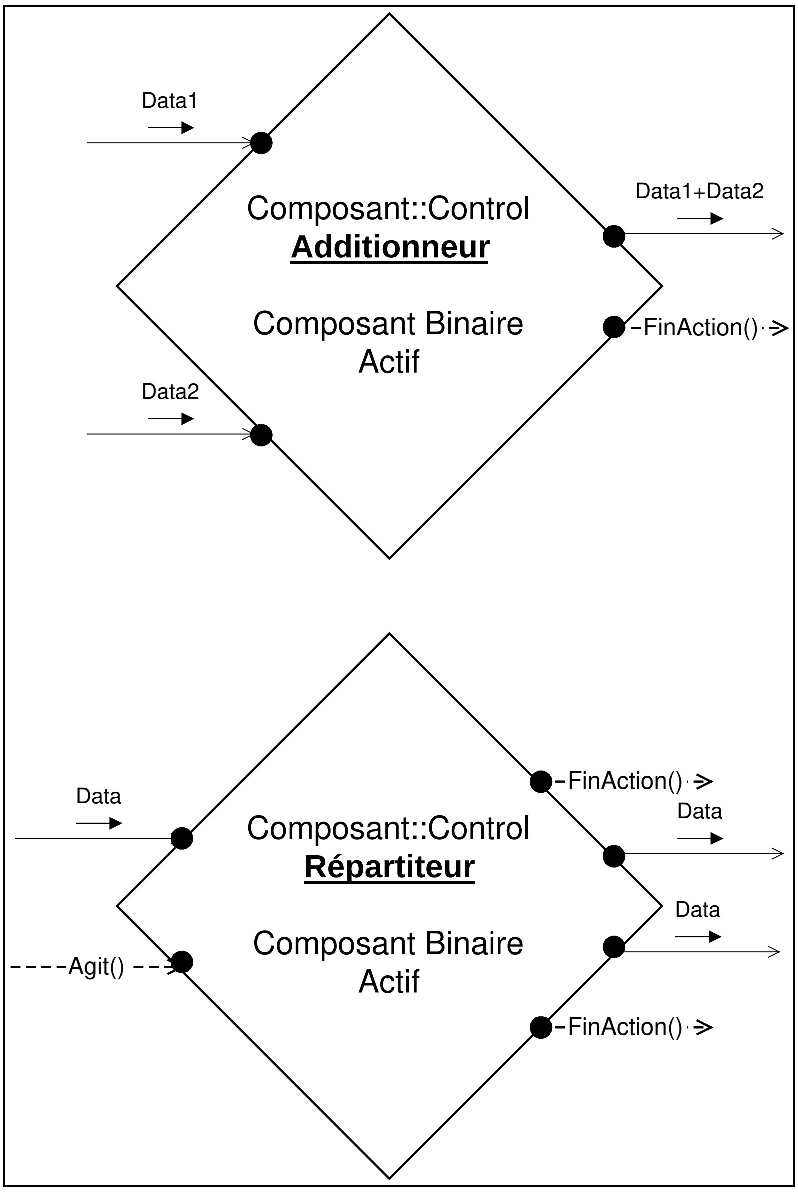


Schéma 11: Un composant Additionneur et un composant Répartiteur

Nous avons donc un ensemble de composants de contrôle qui nous permettent de gérer soit les flots de données soit les flots de contrôle, soit les deux.

Munis de nos composants de contrôle, nous pouvons reprendre les Schéma 7 et Schéma 8 et les modifier pour obtenir un agent qui puisse à la fois suivre une proie **et** fuir un prédateur, ou un agent qui puisse fuir un prédateur **ou** se déplacer aléatoirement.

Comportements Additifs

Suivre Proie ET Fuir Prédateurs

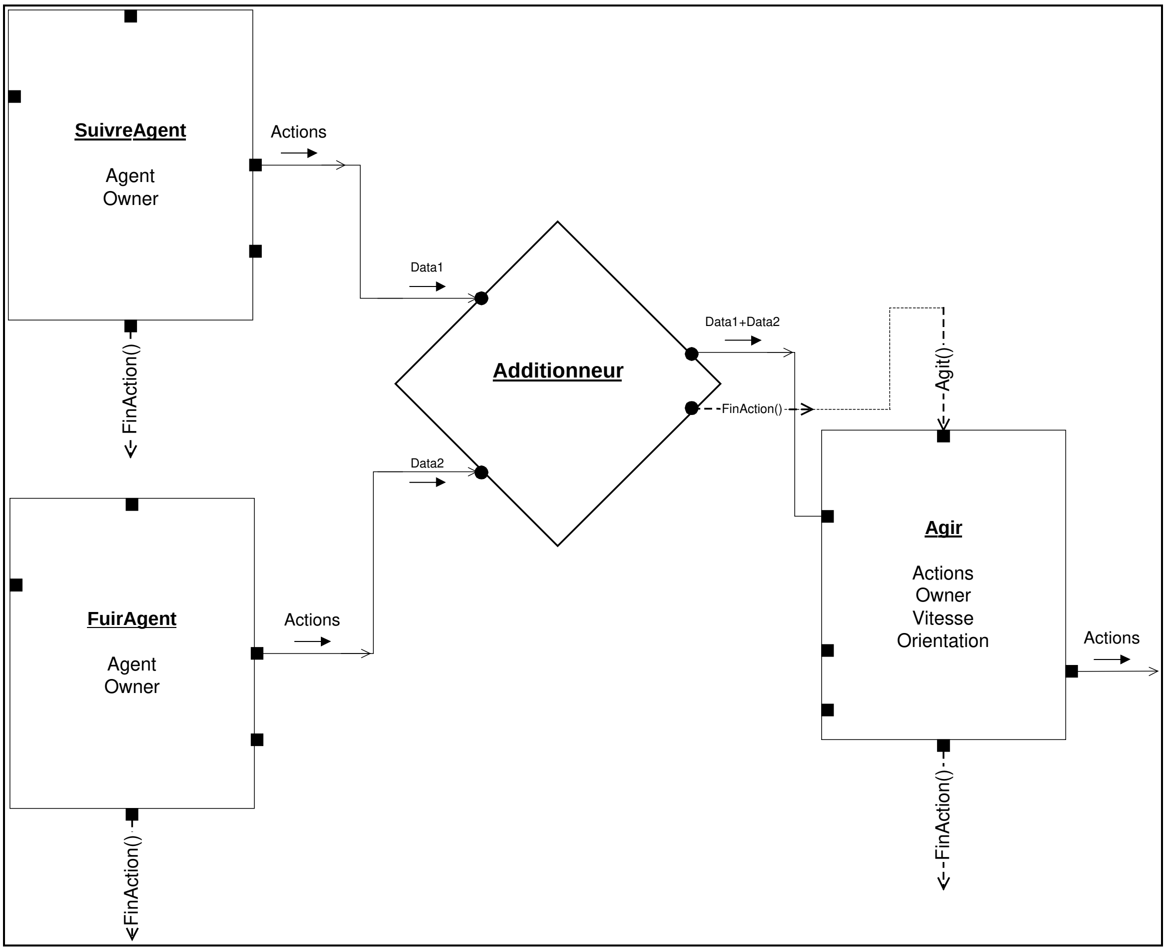


Schéma 12: Un agent presque Prédateur

L'utilisation de l'opérateur d'addition permet d'attendre l'arrivée des données provenant des deux comportements, de les ajouter dans une liste et de transmettre le résultat à l'Effecteur.

Nous voyons sur ce Schéma, que l'Additionneur ne reçoit pas de flot de contrôle des composants comportementaux, en effet, il n'a pas besoin de contrôles en entrée car il est actif. Il peut de plus générer un contrôle (en réalité il transmet celui qu'il a reçu du Scheduler). Nous avons donc avec ce seul composant réglé les deux problèmes de composition du Schéma 7

Nous nous rapprochons de plus en plus de notre définition d'un agent Prédateur à qui il manque cependant le comportement aléatoire.

Comportements Disjonctifs

Fuir Prédateurs <u>OU</u> (absence de prédateurs) Mouvement Aléatoire

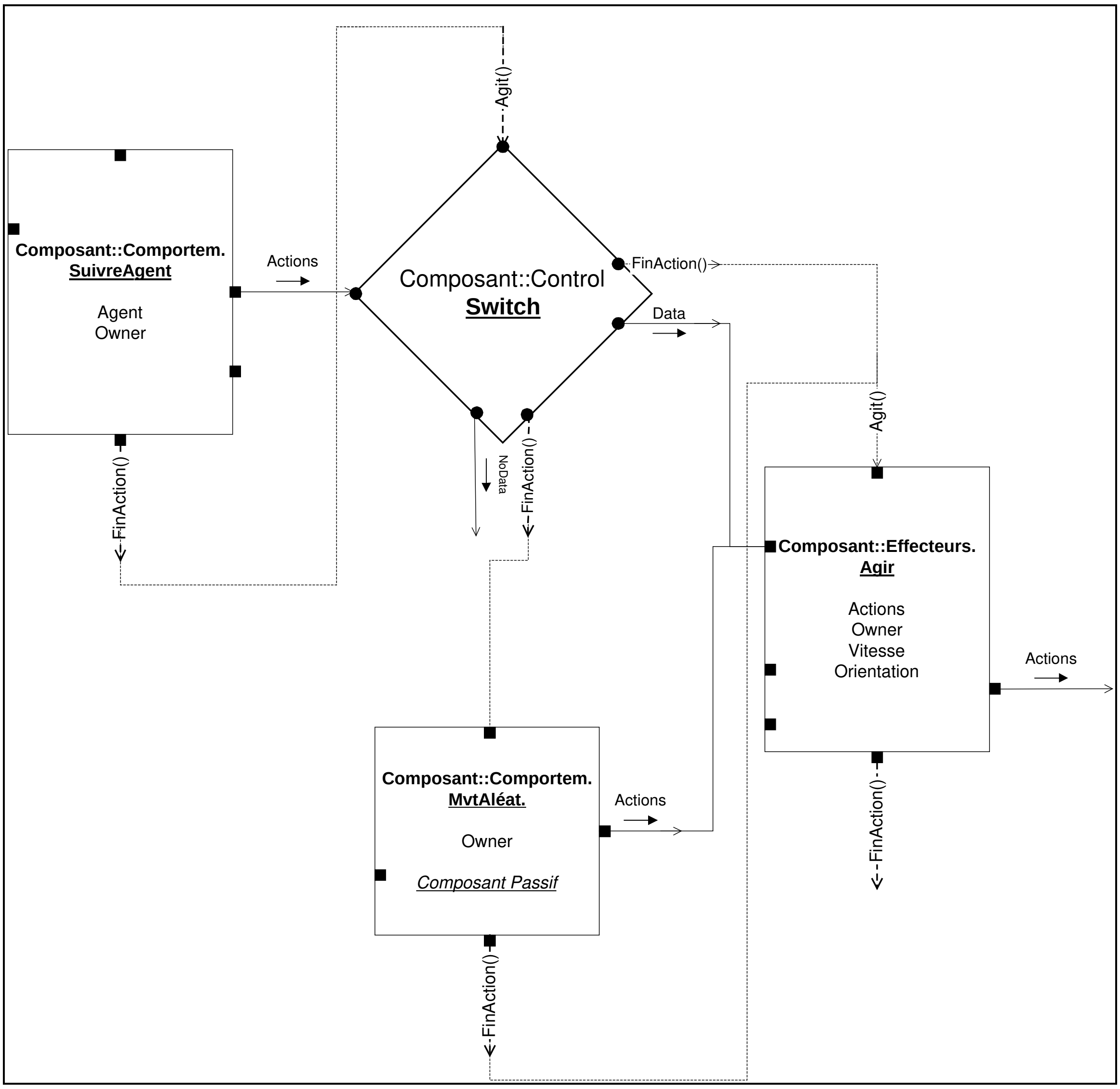


Schéma 13: Un agent Proie

L'introduction du composant Switch permet alors d'envoyer les données provenant du comportement de fuite vers l'effecteur (si le composant de fuite a produit des données) sinon on transmet le contrôle au comportement de mouvement aléatoire qui est lui même relié à l'effecteur. Nous avons donc ici une architecture de type verticale avec une priorité induite par la présence ou l'absence de données. Nous nous apparentons donc à une architecture de subsomption.

Comme nous pouvons le voir sur ce Schéma, l'effecteur reçoit deux connexions sur sa borne supérieure (contrôle) et deux autres sur sa borne d'entrée supérieure (données). Ceci est possible car les composants Maleva peuvent recevoir plusieurs connexions sur une même borne. Cependant, les composants ne reçoivent qu'un seul contrôle et qu'une seule donnée à chaque fois.

Nous avons donc finalement conçu notre agent Proie, il ne nous reste plus qu'à terminer notre agent prédateur. Pour cela nous ajouterons un composant de Mouvement Aléatoire et un Switch en respectant le même principe que pour l'agent Proie.

Comportements Additifs et Disjonctifs

Suivre Proie ET Fuir Prédateurs OU Mouvement Aléatoire

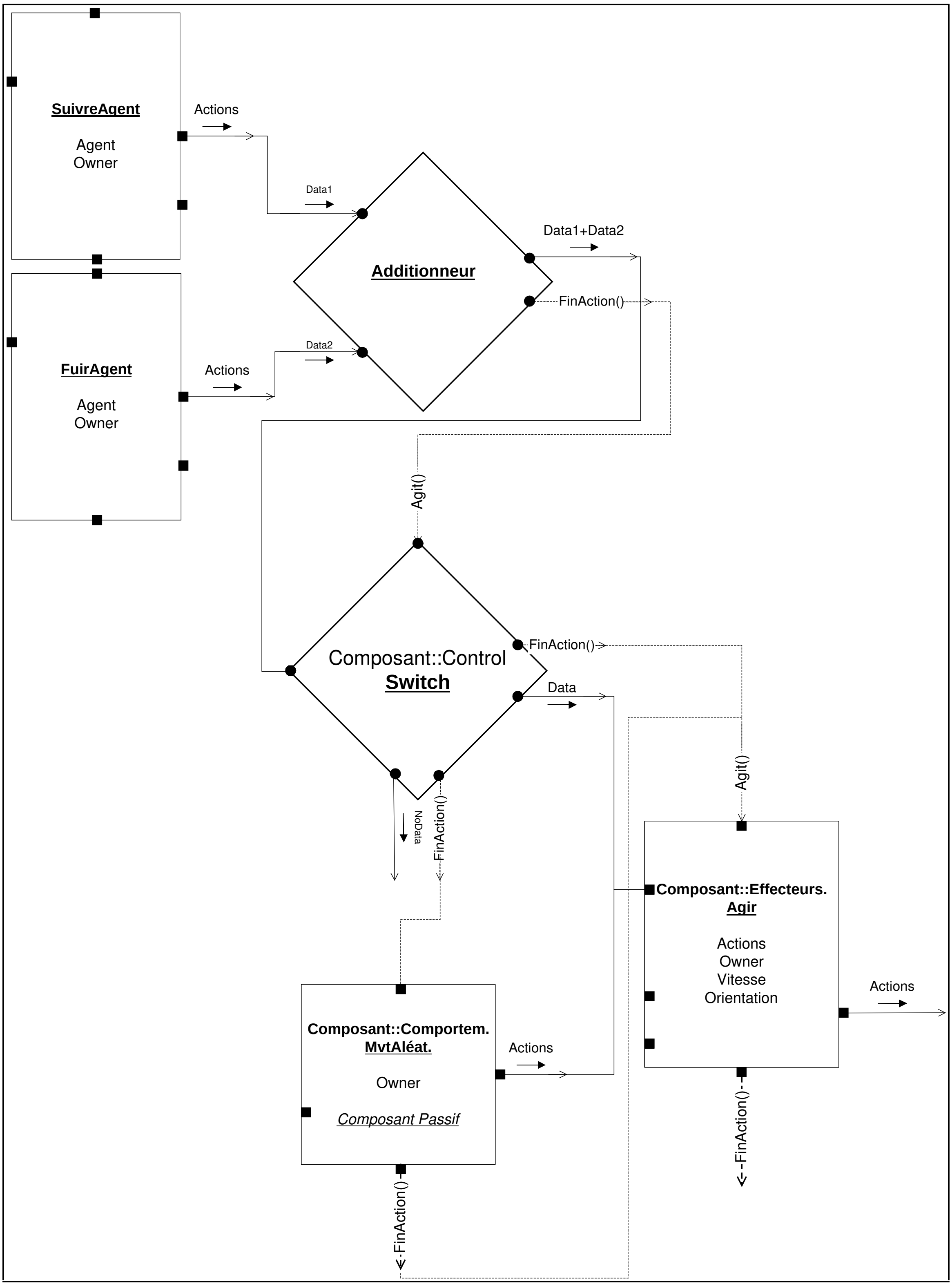


Schéma 14: Un agent Prédateur complet

IV.3 DE LA COMPOSITION FONCTIONNELLE A LA COMPOSITION STRUCTURELLE

Nous avons vu comment à partir de composant comportementaux simples, nous avons conçu un agent au comportement complexe en utilisant uniquement la composition fonctionnelle. L'idéal serait maintenant de créer un nouveau composant comportemental comme étant la résultante des compositions fonctionnelles des composants simples.

Pour créer notre nouveau composant, nous allons utiliser la composition structurelle qui nous permettra d'obtenir un composant composite. Nous allons reprendre le Schéma précédant auquel nous allons ôter les composants Capteur et le composant Effecteur. Tout ce qui reste deviendra le composant composite *Comportement Complexe*.

Nous pouvons voir sur le Schéma suivant ce que cela donne. Pour ne pas surcharger le Schéma, nous n'avons pas mis toutes les données et tous les contrôles.

Nous obtenons donc un composant Comportemental regroupant les trois comportements primitifs que nous avions à la base. Il ne s’agit cependant pas encore d’un composant proprement dit pour une raison simple. Par définition un composant est représenté par des flots entrants, des flots sortants et une fonction interne. Sur ce schéma, la fonction est définie par les trois sous composants mais les flots entrants et sortants ne sont pas encore définis, les flots sont directement liés aux sous composants.

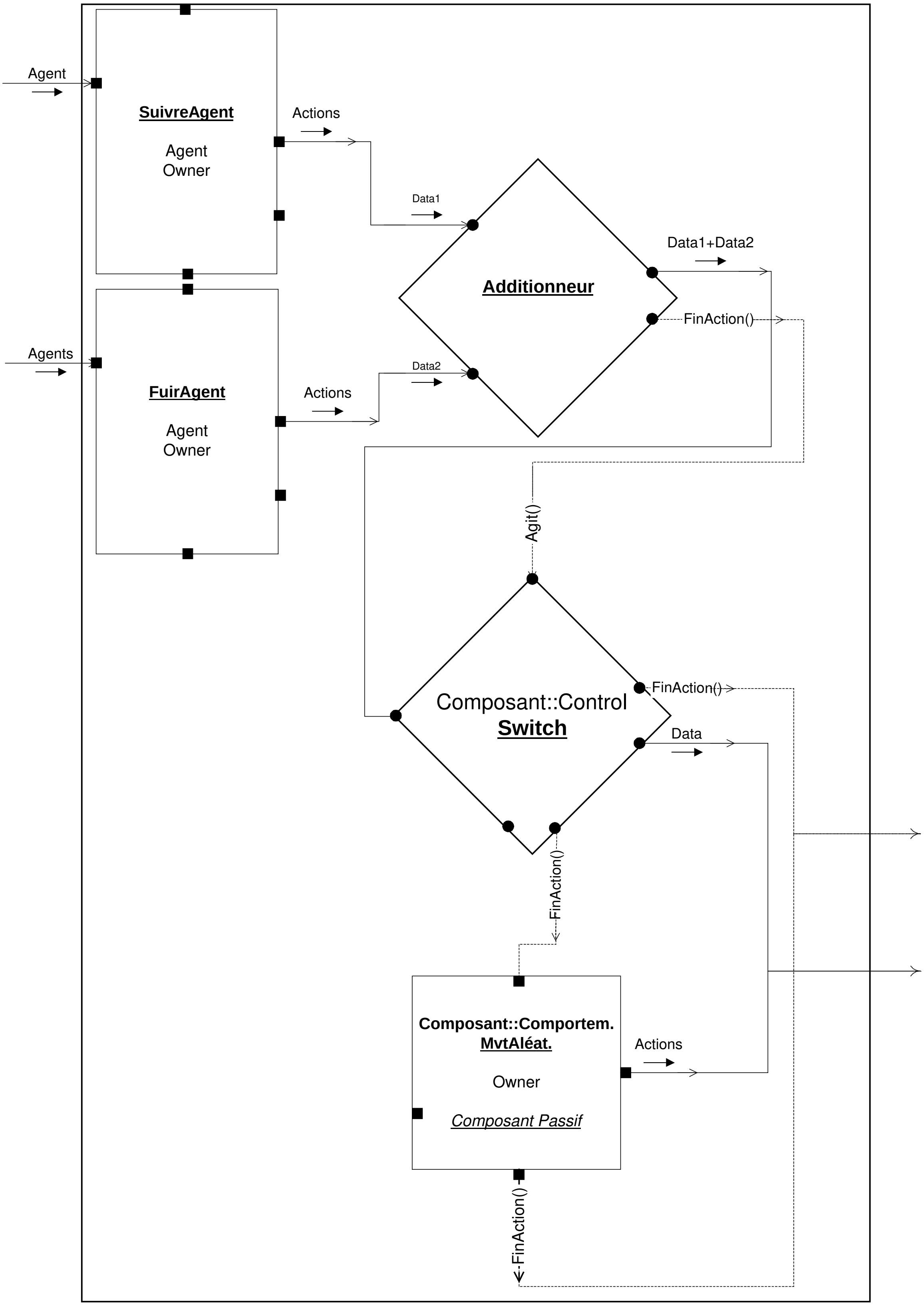


Schéma 15: Un composant Comportement Complexe

Pour obtenir un véritable composant il faudrait remplacer les flots qui pénètrent dans le composant par des flots qui s'arrêtent à sa périphérie. Le schéma deviendrait ceci

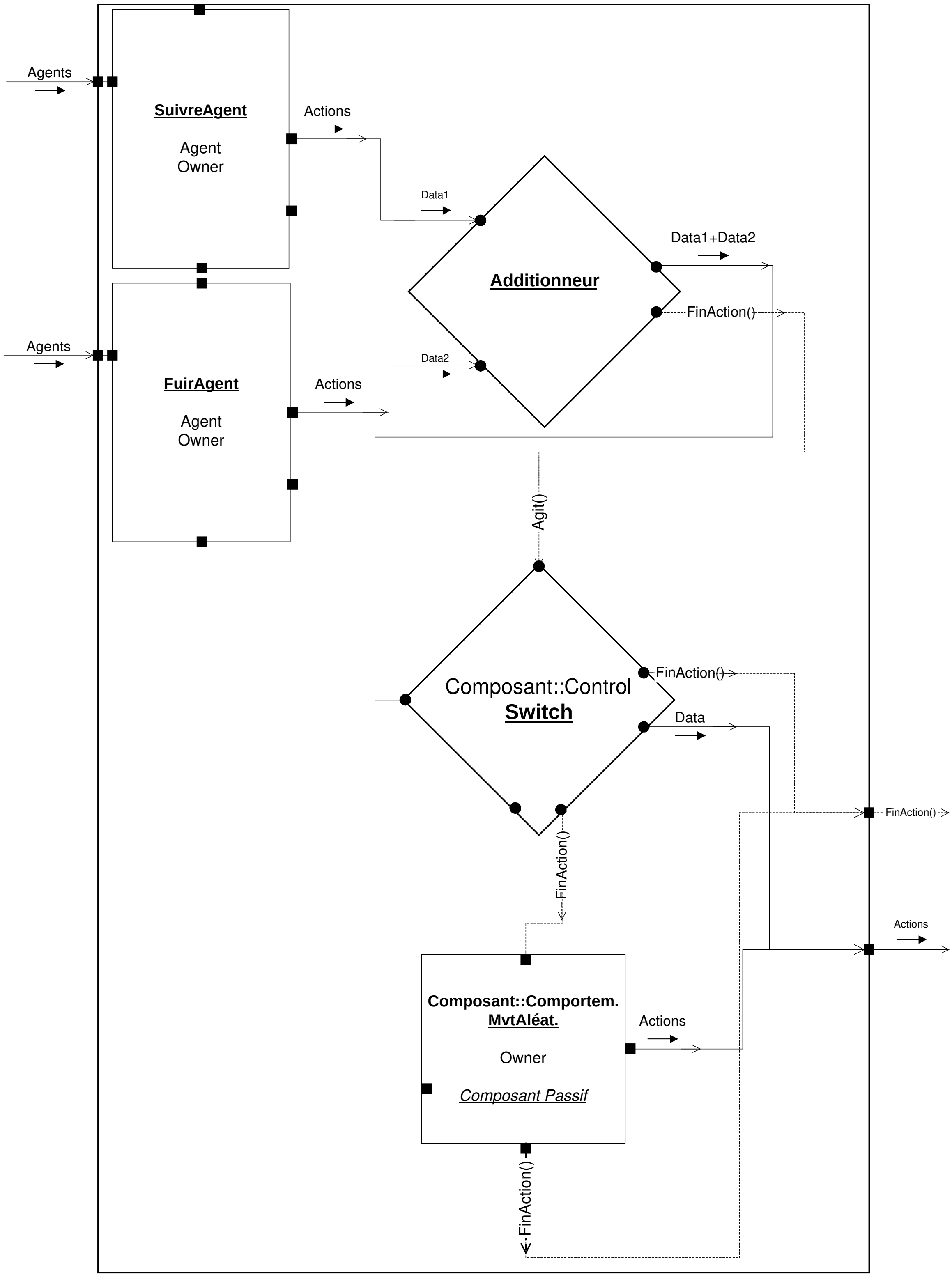


Schéma 16: Un composant Composite: Comportement Complexe

Il reste cependant à nommer les flots, en effet ceux ci sont dupliqués puisque provenant ou se dirigeant vers des composants identiques (Capteurs), ils possèdent donc les mêmes identificateurs (Agents).

Nous avons heureusement à notre disposition un mécanisme permettant de nommer les flots comme bon nous semble sans répercussion ni sur l'envoi ni sur la réception des message. Ce mécanisme est géré par le gestionnaire de communication, ceci est aussi appliqué dans le modèle Cell de Konstantas avec ses ports de communications.

Nous avons enfin conçu nos agents de manière systématique, mais fonctionnent-ils ?

Théoriquement oui, mais en pratique non.

En fait nous n'avons pas terminé notre développement. Il manque encore ce que nous pourrions appeler le passage des composants à l'agent. En effet sur les différents schémas que nous avons vus, nous n'avons que regroupé des composants sans avoir déterminé ni ce qu'était l'agent ni qui il était.

Déterminer ce qu'est l'agent c'est regrouper les composants qui le définissent dans un composant spécial qui est un composant Agent.

Déterminer qui est l'agent c'est lui donner une identité à part entière, ici c'est principalement lui donner un type (Proie, Prédateur), un nom, une apparence visible et des éditeurs permettant de modifier les paramètres internes.

IV.4 DES COMPOSANTS A L'AGENT

Sur les Schémas précédents, rien ne permet de déterminer où est l'agent ni même qu'il s'agit d'un agent Prédateur (ou d'une Proie), par conséquent, les Capteurs ne peuvent pas détecter cette classe (catégorie) d'agent pas plus, d'ailleurs que l'agent Proie.

Dans Maleva nous avons conçu un composant spécial nommé Agent qui est un composant « vide » et dont le rôle est justement de pouvoir identifier et catégoriser des agents. Comme tout composant, il est défini par ses bornes et sa fonction, mais il implémente aussi des fonctions spéciales associées à la gestion des agents telles que des fonctions de sauvegarde et de restitution sur fichier, des fonctions de représentation graphique...

IV.4.1 LE COMPOSANT AGENT

Le composant Agent est un composant spécial car il est plus qu'un composant, c'est un encapsulateur dont le rôle est de pouvoir interagir avec d'autre composants (en ce sens il s'agit d'un composant à part entière) et avec le reste de l'application et l'utilisateur.

Le premier rôle du composant Agent est donc de pouvoir regrouper ses composantes en un « tout ». En effet, et afin de ne pas perdre la notion d'agent au profit de celle de composant, il faut définir un objet censé représenter l'agent. Pour ce faire nous appliquons le même principe de regroupement que nous avons utilisé précédemment. L'agent devient donc le composant avec lequel un autre agent va interagir.

Le deuxième rôle de ce composant est d'offrir des fonctions générales de traitement appliquées à la simulation Sauvegarde/Restitution et Interface. La sauvegarde/restitution sur fichier et une interface avec l'environnement de conception. Bien que cela paraisse simple, il est assez compliqué de sauvegarder dans un fichier un composant dont la structure n'est pas fixée à priori. Il faut garder des références sur chacun des sous composants avec les flots de données et de contrôle et enfin gérer les composants partagés.

Enfin l'agent a pour rôle d'assurer une interface avec l'utilisateur. L'utilisateur doit pouvoir interagir avec l'agent d'une part dans l'environnement de conception et d'autre part dans l'environnement de simulation.

Dans l'environnement de conception, nous disposons d'un ensemble d'outils permettant de représenter l'agent avec l'ensemble de ses composants. Ces outils permettent de définir, ajouter, retrancher et modifier les composants ainsi que les connexions entre les composants.

Dans l'environnement de simulation, l'agent possède une représentation graphique et un éditeur de propriétés. La représentation graphique à pour but d'identifier l'agent dans l'environnement et de permettre de le sélectionner, de le positionner... L'éditeur de propriété (version Delphi) est un outil générique permettant à tout instant de visualiser et modifier des propriétés (attributs d'un objet) d'un composant en laissant à la charge du concepteur de définir ces propriétés et les fonctions d'accès (Cf. §IX.4.4.5).

V 1ERE CONCLUSION

Le modèle de composant que nous avons présenté permet de lier les atouts des langages Objets et Acteurs aux qualités des modèles à base de composants.

Afin de permettre une gestion plus fine de la composition fonctionnelle mais aussi une meilleure gestion de l'activation des composants à l'intérieur d'un agent, nous avons totalement dissocié la notion de flot de données de la notion de flot de contrôle.

Concevoir un agent revient alors à composer, de manière structurelle et fonctionnelle, différents composants comportementaux au sein de l'agent. La gestion du contrôle revient à définir et ajouter des composants spécifiques. Ces composants ont à charge de traiter les compositions associées aux données et celles associées aux contrôles.

Notre modèle de composant semble bien adapté à notre approche incrémentale de la conception et il ne fait aucune hypothèse sur un modèle particulier d'agent.

Si nous souhaitons adopter une démarche de conception inverse, c'est à dire une approche descendante de la conception d'agents, nous devrons alors étudier différents problèmes dus à cette approche. Un de ces problèmes sera de définir un modèle d'agent « générique » et un autre problème, dans le cas d'agents aux capacités évolutives, sera de pouvoir gérer l'ajout et/ou la disparition dynamique d'un composant au sein de cet agent.

Notre modèle de composant, et notamment notre distinction forte entre les flots de contrôle et de données, permet de prendre en compte des modifications dynamiques dans la structure d'un agent. L'utilisation des composants de contrôle permet dans une certaine mesure de traiter les disparitions, mais encore faut-il que l'agent ait été conçu en tenant compte de ces modifications.

Pour définir un modèle d'agent « générique », nous devrons définir un composant de haut niveau (puisqu'un agent est un composant). Ce composant servira de modèle (de squelette) dans lequel nous placerons des composants comportementaux pour définir nos agents.

Pour définir ce composant, il faudra étudier différentes architectures et différents modèles d'agents et il faudra trouver des régularités dans les différents modèles. A partir de ces régularités, nous pourrons définir des patrons (des patterns) de conception. Nous concevrons d'une part des composants comportementaux simples et génériques (c'est déjà le cas pour les composants de fuite, de suivi, capteur et effecteurs). Ces composants devront obéir à certaines contraintes de composition afin d'interdire ou, au contraire, d'automatiser certaines connexions. Une fois définies certaines constantes, nous étudierons et définirons des composants composites dans lesquels nous pourrons placer ces composants comportementaux simples. Pour cela nous devrons aussi définir des schémas de connexion basés sur l'identification des sous-composants. En effet, si nous voulons définir des composants composites « génériques », nous devrons, un peu comme un puzzle, typer les sous-composants et les connexions afin qu'ils prennent leur place naturelle au sein de l'agent. Il serait alors intéressant de pouvoir contraindre (par exemple en typant les bornes d'entrées / sorties) le nombre et le type de composants pouvant être connectés à un composant donné.

Ce travail de fond sur l'identification des patterns nécessite de développer une grande quantité de modèles d'agents et d'exemples, puis de les analyser. Ce travail, dont la durée dépend de la complexité des agents à modéliser (il faut en effet revoir les modèles existants pour les appliquer dans une vision componentielle), dépasse l'objet de cette thèse et fera l'objet d'études ultérieures.

Une autre direction d'étude que nous souhaitons examiner correspond à pouvoir gérer l'ajout dynamique d'un nouveau composant au sein d'un agent sans avoir auparavant conçu un composant spécifique destiné à prendre en compte cet ajout. Il s'agit d'un autre problème qui se situe à un autre niveau de notre modèle. En effet, prendre en compte l'ajout dynamique d'un composant, ajout non supervisé par l'utilisateur, demande une gestion centralisée de la composition (nous ne sommes pas en contradiction avec notre vision décentralisée du contrôle, la gestion centralisée représente, ici, la gestion des composants qui interagissent avec le nouveau composant). Le problème revient à déterminer et créer les connexions entre les composants existants et le nouveau composant et bien sur gérer la distribution du contrôle avec ce composant.

Partie 2

Les réalisations

VI INTRODUCTION

Comme nous l'avons dit auparavant, notre modèle de composant à été conçu avant tout pour la conception d'agent et particulièrement celle des agents situés en environnement simulé.

Il existe une multitude de plates-formes dédiées à la simulation de Systèmes Multi-Agents mais il n'existe pas de véritable analyse logicielle sur ces plates-formes. Ceci est en particulier dû au fait que la simulation est une petite partie de la recherche en IAD et à notre avis surtout parce qu'il n'est pas facile d'analyser un système aussi complexe qu'un SMA et encore moins de déterminer des outils de conception qui facilitent la création, la simulation et l'analyse de tels systèmes.

Les plates-formes existantes, pour la plupart, ne décrivent que les architectures des agents et les interactions agents/agents, et peu ou pas du tout le modèle sous-jacent, ni même l'environnement simulé, les interactions agents/environnement et agents/utilisateur. De plus ces plates-formes ne sont pour la plupart conçues que pour certains types d'agents (souvent des agents assez complexes) et bien qu'elles se disent génériques ne permettent pas de concevoir des simulations où sont étudiés des agents aux structures évolutives ou flexibles.

Afin de mettre en pratique notre modèle componentiel, nous avons conçu la plate-forme Maleva.

Cette plate-forme propose un ensemble d'outil qui permettent :

- De concevoir des composants.
- De composer les composants précedemment conçus.
- De concevoir des agents.
- De concevoir des environnements topologiques.
- De simuler les agents dans un environnement.

Nous commencerons par étudier quelques plates-formes de simulation et extraire certains principes fondamentaux.

Nous décrirons dans une seconde partie notre modèle componentiel et nous finirons par la réalisation effective de Maleva dans le langage Delphi.

VII LA PLATE-FORME MALEVA

Il existe beaucoup d'articles traitant de simulation mais très peu d'entre eux traitent de la plate-forme qui a permis la conception et la simulation des agents. Nous étudierons dans ce chapitre les plates-formes que je connais le mieux soit pour avoir participé à leur développement, soit pour les avoir étudiées dans le détail. Il s'agit des plates-formes Alia [Delaye93], Manta [Drogoul93], Mages [Bouron92], Simpop [Bura[27]]. La dernière en date étant la plate-forme de simulation de L. Magnin [Magnin 96] spécialement conçue pour la gestion des contraintes environnementales sur les simulations d'environnements «réels».

Toutes les plates-formes de simulations n'introduisent pas la notion d'environnement simulé. En effet il n'est pas nécessaire de disposer d'un tel type d'environnement pour que des agents puissent communiquer et résoudre des problèmes.

Il est cependant nécessaire de définir un environnement :

- Dans le cas de simulation faisant intervenir des agents situés, c'est à dire des agents mobiles ayant des caractéristiques physiques telles qu'une vitesse, une position...
- Dans le cas de simulation faisant intervenir des contraintes physiques sur les agents, par exemple des obstacles.
- Dans les simulations d'environnements réels. Par exemple la simulation d'une forêt dans le cas d'une étude sur l'écosystème ou par exemple une pièce (ou un terrain de football) pour des simulations en robotique.

Dans cette thèse, nous nous plaçons volontairement dans l'analyse de plates-formes de simulations possédant un environnement simulé et des agents situés.

Une plate-forme de conception et de simulation de Systèmes Multi-Agents est une application qui doit permettre de:

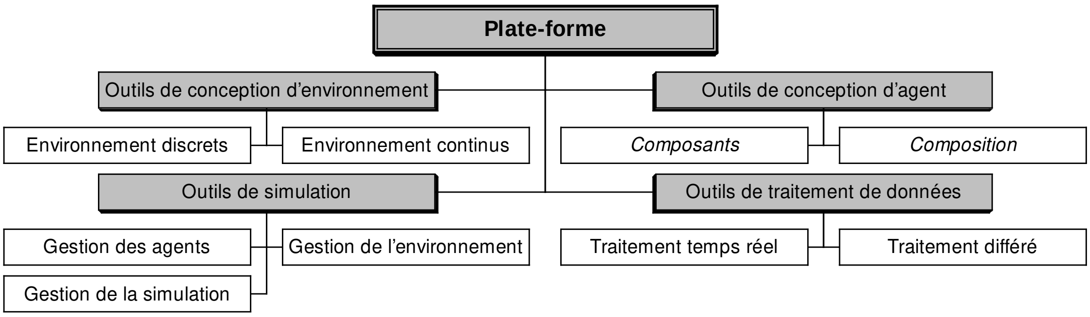
- concevoir des agents. Il s'agit de la programmation des composantes mais aussi de la conception des agents et enfin de la définition des paramètres de chaque composante (ex. la distance de perception d'un capteur).

[27] En cours de rédaction.

- de les activer dans un environnement simulé. Il s'agit de faire « fonctionner » les agents, c'est à dire de leur permettre de résoudre leur but mais aussi de gérer les agents (sauvegarde, restitution, placement...).

- de traiter les données des différentes simulations. Il s'agit principalement de récupérer les données individuelles (âge d'un agent, paramètre des composantes, but...) ou les données collectives (nombre d'agents, localisation...) et de permettre de les traiter (texte, données numériques dans un tableur, courbes, analyse statistique...).

Le schéma suivant montre les différents aspects d'une telle plate-forme.



Les outils de conception d'environnement ne seront pas traités car Maleva ne propose pas, a proprement parler, d'outils de conception d'environnement. En revanche, Maleva propose plusieurs environnements (Environnements discrets, continus...) qui sont des composants logiciels, nous proposons un ensemble de classes génériques permettant de concevoir de nouveaux environnements.

VII.1 LA PLATE-FORME DOIT DISPOSER D'UN ENVIRONNEMENT DE SIMULATION.

Un environnement de simulation représente la réification informatique d'un espace physique où les agents sont censés percevoir et agir. L'environnement est concret si les agents ont ou subissent des contraintes physiques, il est abstrait dans le cas contraire. Par exemple les agents représentant des fourmis dans Manta ou des robots dans ALIA subissent des contraintes physiques dans leur perception mais aussi dans leurs actions. Les robots ne peuvent pas se déplacer sur n'importe quel type de terrain (sinon ils s'épuisent). Les fourmis ne perçoivent qu'une partie de leur environnement, elles ne perçoivent que les stimuli déposés à certains endroits de la fourmilière et suffisamment proches pour être perceptibles.

Lors de la réalisation d'un environnement, le concepteur est naturellement amené à étudier le modèle d'environnement. Il existe deux grandes catégories d'environnements qui sont les environnements discrets et les environnements continus.

Environnements Discrets

Un environnement discret est un environnement découpé en cases qui forment un pavage de celui-ci. Il existe une multitude de dicrétisations possibles. Nous avons la discrétisation carrée (Manta, Mages, Maleva), rectangulaire (Maleva), hexagonale (Simpop, Maleva) qui correspondent à des pavages complets de l'environnement (tout l'espace est utilisé). Il existe également des pavages incomplets comme par exemple des cartes du monde (Maleva) qui correspondent à des pavages partiels ou quelconques.

La discrétisation de l'environnement a pour but de simplifier sa définition et de faciliter sa gestion. Mais la discrétisation a pour conséquence de définir les interactions entre les agents et l'environnement. Ces interactions se font au niveau de la perception de l'environnement par les agents ou au niveau des actions des agents dans l'environnement[28]. Chaque type d'interaction peut donner lieu, comme nous allons le voir, à une catégorie de discrétisation.

Dans le cas où les distances de perception des agents[29] sont identiques à un coefficient multiplicateur près[30], il peut être utile de discrétiser l'environnement. Une case de l'environnement représente alors une surface équivalente à la zone de perception de l'agent. Dans ce cas, les agents ne perçoivent que les données présentent dans la case où ils se trouvent (ou les cases adjacentes si le coefficient multiplicateur est supérieur à 1). L'application d'une discrétisation spatiale permet alors une optimisation, en temps de calcul, non négligeable. L'utilisation de cette technique est rare, en effet, dans le cas où les distances de perceptions sont variables (cas le plus courant), il faut que les tailles des cases se modifient en conséquence, ce qui rend ce modèle difficile à mettre en œuvre.

Dans le cas de l'action, une case représente en général un pas de déplacement d'un agent. De même que dans le cas précédent, on suppose que les agents se déplacent (perçoivent) d'une case vers une autre case adjacente, ou de plusieurs cases à chaque pas bien que dans ce cas nous devions gérer des problèmes de distances (Cf. §V, **Figure 21** et **Figure 22**).

Hormis l'aspect algorithmique et optimisation, la discrétisation d'un environnement est utilisée pour plusieurs raisons :

[28] L'agent perçoit par l'intermédiaire de ses capteurs et agit par l'intermédiaire de ses effecteurs. Capteurs et Effecteurs sont donc étroitement liés aux agents mais aussi à l'environnement et à sa dicrétisation.

[29] On suppose ici que les agents ne perçoivent pas tout l'environnement mais qu'ils ont une perception circulaire (ou angulaire) de rayon R fixé.

[30] Mr Lalande, dans sa plate-forme MACACA, traite de ce cas particulier et utilise un algorithme spécifique qui optimise (en temps) la perception.

- Cela permet de simuler plus facilement des phénomènes qui sont continus dans l'espace et variables dans le temps tels que des phénomènes de propagation de données (stimuli, température, humidité...) et de gérer particulièrement simplement les contournements d'obstacles. Dans Manta, la propagation de stimuli est une des particularité principale de l'environnement. Le fait que l'environnement soit discrétisé permet alors de lier efficacité et réalisme, le temps de calcul pour la propagation des stimuli est suffisamment court pour être applicable et la discrétisation (une case représente un pas de déplacement d'un agent) est suffisamment fine pour que l'environnement s'apparente à l'environnement réel (une fourmilière).

- Cela permet de définir plus facilement l'environnement physique (murs, obstacles ...) et donc de déterminer plus facilement des contraintes physiques locales appliquées aux agents. Dans Mages deux agents ne peuvent pas occuper la même case, dans Manta ils ne peuvent pas traverser les murs (un mur occupe l'espace d'un case). Ces contraintes locales sont facilement identifiables car la discrétisation de l'environnement se fait sur la base du déplacement d'un agent, autrement dit, un agent peut ou ne peut pas se déplacer sur une case.

- Cela permet enfin de traiter plus facilement des conflits entre plusieurs agents (gestion des évitements, collisions...). Toujours dans l'esprit d'une discrétisation par rapport au déplacement, il devient plus facile de déterminer les collisions entre deux agents même s'il n'est pas évident de les gérer. Dans la plupart des plates-formes, cette détection est faite sans pour autant que la gestion des collision soit traitée. Ce problème a d'ailleurs été efficacement traité dans la plate-forme de Sième de Laurent Magnin que nous étudierons par la suite.

Environnements continus

Un environnement continu, par opposition à discret, est un environnement où les données manipulées ne sont pas discrètes. Ce type d'environnement est plus difficile à gérer que le précédent mais il simule plus finement un environnement réel. La Plate-forme ALIA proposée par Claude Delaye propose un environnement continu représentant la surface d'une planète (Mars), les travaux de L. Magnin dans sa plate-forme Sième se font aussi dans un environnement continu et une des applications qui en a été faite proposait un environnement représentant un terrain de football.

L'utilisation d'un tel type d'environnement se fait pour plusieurs raisons :

- La taille de l'environnement n'est pas contrainte. L'environnement est de grande taille ce qui n'est pas possible dans un environnement discret où le nombre de cases est un facteur important.

- Les phénomènes continus peuvent être modélisés sous forme de fonctions dans l'espace et le temps f(x,y,z,t), il existe cependant des difficultés pour traiter des phénomènes locaux tels que le contournement d'obstacles (ceci fait intervenir des fonctions dont le temps de calcul devient trop important par rapport à la fonction première de la simulation).

- Les agents peuvent être modélisés dans l'environnement sous forme de vecteurs. C'est la cas par exemple dans Maleva où les agents sont représentés (dans l'environnement) par des vecteurs de direction et de vitesse.

- Les conflits entre les agents doivent être gérés de manière globale et non pas locale comme dans les environnements discrets.

Les travaux de L. Magnin traitent essentiellement de cet aspect de la simulation concernant la gestion des contraintes physiques réelles appliquées à un agent. Ses travaux sont à l'heure actuelle appliqués à la robotique mobile bien qu'ils puissent s'appliquer à toute simulation faisant intervenir des agents dans un environnement réel. Dans Sième, l'agent est défini comme une entité, L. Magnin lui ajoute une structure physique (représentation géométrique du volume occupé par l'agent), des propriétés internes et des propriétés externes (vitesse, direction, poids...). Les propriétés externes sont accessibles par l'environnement. Cette particularité permet alors de déléguer la gestion de ces propriétés à l'environnement (en temps réel). L'environnement propose un ensemble de règles qui sont fonction de l'espace et du temps et qui peuvent s'appliquer à une ou plusieurs propriétés d'un agent. De ce fait, l'environnement devient à même de déterminer le moment où une propriété doit être modifiée et de calculer la nouvelle valeur de cette propriété. L'ajout de règles de gestion de conflits permet de gérer les collisions en offrant alors la possibilité de décrire ce que l'on doit faire et surtout quand on doit le faire (par exemple une balle doit rebondir quand elle rencontre un mur). La contrepartie d'une telle richesse est qu'il faut modéliser l'ensemble des interactions physiques entre les agents ou objets de l'environnement (chocs Balle/Mur, Balle/Agent, Agent/Mur, Agent/Agent...).

VII.2 LA PLATE-FORME DOIT PERMETTRE DE CREER DIFFERENTS TYPES D'AGENT.

Par définition, un Système Multi-Agents est un système où cohabitent des agents. Ces agents peuvent être de même nature (agents identiques à quelques paramètres internes près) où des agents aux structures et comportements différents. La plate-forme doit par conséquent offrir la possibilité de concevoir différents types d'agents. Différents types d'agents signifie des agents aux capacités différentes (possibilités comportementales), aux structures (composantes) différentes ou aux architectures (modèles) différentes.

La capacité d'un agent représente ce qu'il peut faire. Deux agents aux capacités différentes sont par exemple deux robots dont l'un serait spécialisé dans le transport et l'autre dans le forage. Cette capacité est définie lors de la conception ou par apprentissage/adaptation. Dans un cas comme dans l'autre, la capacité d'un agent peut influer sur sa structure. Les travaux de M. Delaye traitent essentiellement de l'adéquation entre capacité et structure[Delaye93] et certains de nos travaux ont porté sur l'évolution morphologique (structure physique) et comportementale (capacités comportementales) d'une société de robots [Lhuillier91].

La structure d'un agent représente ses composantes. Dans le cas d'une architecture de subsomption (Cf. Figure 1), la structure est définie par les différents modules qui eux mêmes définissent la capacité initiale de l'agent. Modifier la structure revient à modifier le comportement. Dans le cas d'une architecture à base de composants logiciels [Yoo97], le nombre et la nature des composants définissent la structure d'un agent. Par exemple, deux agents aux structures différentes seraient deux robots dont l'un ne ferait que du transport (minerais par exemple) et l'autre que du forage. Dans un tels cas leur structure physique est différente, le premier étant mieux adapté aux déplacements et le second à l'extraction et à la perception.

Enfin des agents aux architectures différentes sont des agents aux composantes différentes avec un mécanisme d'exécution différent. Par exemple une fourmi de la plate-forme Manta et un robot de la plate-forme Alia sont des agents aux architectures différentes. Dans le premier cas, nous avons une architecture de subsomption tandis que dans le second cas nous avons une architecture composite. Les mécanismes d'exécutions des agents sont différents, dans Manta, le comportement de l'agent obéit à la perception tandis que dans Alia, c'est l'inverse, c'est le comportement qui définit la perception.

Si concevoir des agents à capacités différentes ou des agents de structures différentes n'est pas un problème conceptuel en soit, faire cohabiter des agents ayant des architectures différentes peut devenir complexe voir impossible suivant l'architecture initiale des agents. La plupart des plates-formes ne proposent qu'une seule architecture d'agent (horizontale, verticale ou hybride), nous verrons par la suite qu'une approche modulaire permet d'utiliser des composantes identiques avec des mécanismes de gestion différents et que ceci permet alors de concevoir des architectures différentes.

VII.3 LA PLATE-FORME DOIT PERMETTRE DE SIMULER LES AGENTS

Le rôle principal d'une plate-forme de simulation est de permettre de simuler le fonctionnement des agents, c'est à dire de les activer (à tour de rôle ou simultanément) et de permettre une modification de l'environnement dans la cas où celui-ci ne serait pas passif. Le mécanisme d'activation des agents dans une simulation peut se faire suivant deux grands principes qui sont une activation séquentielle ou parallèle et dans les deux cas cette activation revient à permettre à l'agent d'effectuer ce que l'on appelle son cycle d'activation.

Activation parallèle

L'activation parallèle consiste à associer à chaque agent une tâche (un processus). Les langages d'acteurs, comme nous le verrons plus loin, offrent les outils nécessaires à la réalisation de tels systèmes. Ce mécanisme d'activation est encore peu utilisé dans les simulations car il impose des contraintes sur l'architecture des agents et impose de définir des mécanismes de synchronisation qui limitent une grande partie de l'intérêt même de cette approche. Parmi les principaux problèmes de synchronisation, nous avons à gérer l'état de l'environnement, c'est-à-dire la validité et la viabilité des données que les agents perçoivent et manipulent dans l'environnement. Par exemple, si un agent consomme une ressource (par exemple de l'eau) dans l'environnement, il faut que cela se répercute sur la perception des autres agents. De même puisqu'il s'agit des données de l'environnement, nous devrons gérer les interactions au niveau des actions, deux agents qui consomment la même ressource peuvent créer des problèmes d'exclusion mutuelle. Enfin au niveau de la gestion même de la simulation, le simple fait de vouloir sauvegarder le monde (agents+environnement+données) ne peut se faire que lorsque tous les agents et objets (par exemple l'environnement) sont dans un état interne stable et satisfaisant (début ou fin de leur cycle interne).

Activation séquentielle

L'activation séquentielle qui est plus souvent utilisée (dans les plates-formes étudiées) correspond au principe de n'activer les agents que les uns après les autres dans un ordre qui a été précédemment défini. Ce mode d'activation est plus souvent utilisée dans les simulations car elle résout une grande partie des problèmes de synchronisation et elle permet de définir la notion de cycle. Un cycle correspond alors à l'activation de l'ensemble des agents et des objets actifs du monde (ex. l'environnement s'il n'est pas lui même un agent), les agents perçoivent, délibèrent et agissent dans le même cycle.

VII.4 OBTENIR DES RESULTATS PROVENANT DE LA SIMULATION

La plate-forme doit permettre d'obtenir des résultats, c'est à dire de visualiser, traiter des données sur un agent, sur un groupe ou sur la population entière.

Un des problèmes que nous avons à gérer est dû au fait que l'on ne sait pas a priori ce que l'on va vouloir visualiser, ni sous quelle forme (graphiques, données numériques, données textuelles...). Il n'existe pas, dans les plates-formes que nous avons étudiées, d'approche générale (c'est à dire qui offre la possibilité de traiter tout type de données), mais, plus souvent, comme dans la plate-forme de R. Cazoulat [Cazoulat95] une approche spécifique. L'utilisateur peut visualiser l'évolution d'un certain nombre de paramètres numériques prédéfinis (population, âge, mortalité...), ces paramètres sont fixés et vouloir définir de nouvelles structures de données impose de réécrire les outils de visualisations.

Soit la plate-forme implémente des outils de visualisation (c'est le cas le plus fréquent et particulièrement les plates-formes conçues à partir du langage Smalltalk) soit elle permet l'utilisation d'outils externes (par exemple Excel).

Dans le premier cas, nous pouvons alors visualiser les courbes en temps réel car l'utilitaire a été conçu pour. Bien entendu, dans un tel cas, la conception de nouvelles simulations (agents environnements) passe nécessairement par la redéfinition de cet utilitaire.

Dans le second cas, l'utilisation d'un outil externe permet de gagner en temps de développement et de mise à jour. Ce gain est contrebalancé par un traitement et une visualisation a posteriori des données. La plate-forme génère les données dans un ficher qui sera traité par la suite (après la simulation) par l'outil.

Quelle est la meilleure solution ? Il est difficile de répondre car beaucoup de paramètres doivent être pris en compte comme la disponibilité de l'outil externe, le nombre de données, le type de traitement à effectuer. Nous reviendrons sur nos choix dans le Chapitre IX.

Comme nous l'avons vu, il existe un grand nombre d'architectures possibles. Chacune d'entre elle possède des qualités et des défauts mais, quoi qu'il en soit, on ne peut pas a priori[31], dire qu'il en existe une qui soit plus adaptée qu'une autre. Ces architectures ont cependant en commun une approche modulaire de la conception de l'agent. C'est cette approche modulaire que nous étudierons dans le chapitre suivant.

[31] Avant la définition même du problème à traiter.

VIII ENVIRONNEMENT DE SIMULATION

Une plate-forme de simulation se doit d'offrir des outils destinés à la simulation. Dans le cas de SMA situés, il faut définir un environnement de simulation, c'est à dire une réification informatique d'un environnement réel ou virtuel.

Ce chapitre est destiné à l'étude des différents types d'environnements les plus souvent employés dans les plates-formes de simulation.

Ce que nous ne verrons pas dans ce chapitre, c'est un modèle d'environnement tel que proposé par Laurent Magnin dans sa thèse [Magnin96], en effet il s'agit d'une approche particulière de la notion d'environnement essentiellement destinée à la simulation d'environnements physiques réels et essentiellement appliquée à la robotique mobile.

VIII.1 L'ENVIRONNEMENT

Dans l'étude des systèmes multi-agents (SMA), l'environnement a un rôle privilégié, il peut servir à la fois de support de communication entre agents, il peut servir de support physique pour les déplacements des agents, il peut encore servir de support de données. La diversité des fonctions de l'environnement, la complexité même de sa définition mais aussi les problèmes dus à la discrétisation de celui-ci en font un système difficile à modéliser. Il apparaît donc naturel d'accorder un chapitre à l'étude de l'environnement.

Dans ce chapitre nous entendrons comme environnement tout ce qui n'est pas agent. Il faut alors faire une distinction entre l'environnement vu de l'environnement et l'environnement vu des agents.

Pour lui même, l'environnement est caractérisé par un modèle qui définit sa taille, ses contraintes. Pour l'agent, l'environnement est défini par ses contraintes et par les objets qu'il contient.

Ce chapitre sera donc découpé en plusieurs paragraphes:

- En premier lieu nous verrons les différents modèles d'environnement, leurs caractéristiques, leurs défauts.

- En deuxième partie nous verrons les différents rôles de l'environnement, en tant que support de communication, support physique et support de données ce qui introduira les interactions agent/environnement de type perception.

- Enfin nous aborderons en fin de ce chapitre la problématique des environnements distribués et des interactions de type action.

VIII.1.1 LES DIFFERENTS ENVIRONNEMENTS.

Jusqu'à présent, dans les différentes plates-formes de simulation que j'ai eu à étudier, l'environnement est défini autour de 2 axes, chacun de ces axes correspondant à un problème. Le premier axe est la discrétisation de l'environnement et le second est la taille de l'environnement ou tout au moins ses limites physiques.

VIII.1.1.1 LES ENVIRONNEMENTS DISCRETS/CONTINUS.

Les environnement discrets.

Un environnement discret (par opposition à continu) est un environnement découpé en cases adjacentes qui couvrent la totalité ou une partie de sa surface. La taille d'une de ces cases définit en général le pas de déplacement d'un agent de l'environnement. C'est actuellement le cas pour les plates-formes Manta et Mages, où les agents qu'ils soient représentés par des fourmis, des robots ou des chasseurs ne peuvent se déplacer que d'une case vers une autre case adjacente.

La discrétisation de l'environnement permet de faciliter un bon nombre de calculs, mais il permet aussi de définir très facilement ce qui compose l'environnement. *Au niveau des données accessibles par un agent, le problème est de savoir si il est plus rentable de recalculer une valeur pour une coordonnée précise (cas des environnements continus) ou si il est plus rentable de calculer la valeur pour chaque case et de laisser celle-ci la traiter.*

Dans les environnements discrets, chaque case devient alors à la fois l'environnement pour l'agent qui s'y trouve mais elle sert aussi à définir les possibilités de déplacement pour l'agent (donc ses contraintes). Suivant le type de déplacement que l'on veut rendre possible à un agent (déplacement horizontal, vertical, diagonal), il faut définir la forme d'une case. Dans le cas des plates-formes Manta et Mages, les cases sont carrées, et dans le cas de SimPop[32] les cases sont hexagonales. Nous n'avons adopté dans Maleva pour aucune solution particulière, nous offrons différents types d'environnement (différents composants) et différents types de discrétisations. Cependant nous préférons les cases hexagonales pour différentes raisons.

Les cases carrées aussi bien que les cases hexagonales permettent de couvrir la totalité de la surface de l'environnement. Elles sont toutes identiques, ce qui facilite leur définition. En revanche, le calcul des accointances d'une case carrée est facile puisque se faisant dans les 4 directions tandis que dans le cas des cases hexagonales, ce calcul se complexifie par le décalage sur deux lignes des cases.

---

[32] La plate-forme de simulation SimPop (pour Simulation de Population) a été développée par S. Bura dans l'optique de la simulation de l'évolution de zones urbaines.

Dans le cas des cases carrées, un déplacement diagonal est généralement appliqué en deux étapes (voir figure ci-dessous) tandis que les cases hexagonales de par leur nature permettent ce déplacement.

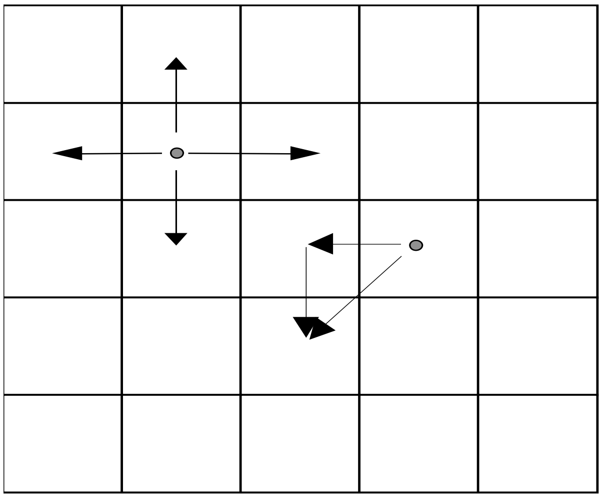


Figure 20: Un environnement avec discrétisation carrée

Enfin, et le plus important, les cases hexagonales permettent une définition plus précise de l'environnement car toutes les cases adjacentes sont équidistantes ce qui n'est pas le cas des cases carrées ou le problème des diagonales reste entier.

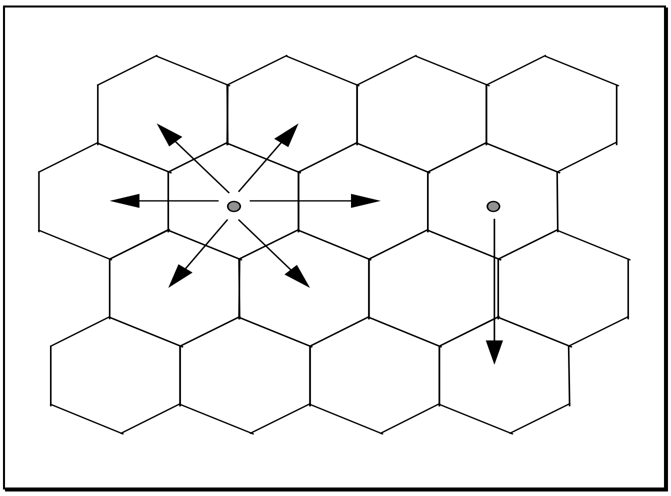


Figure 21: Un environnement avec discrétisation hexagonale

Cependant les cases hexagonales, si elles ajoutent les diagonales ne sont pas parfaites. Elles complexifient les calculs de déplacement car elles ne permettent pas de faire des déplacements suivant les huit axes mais seulement sur six axes (Cf. Figure 21), il manque soit l'horizontale soit la verticale (cas de la Figure 21). En effet, si les accointances sont disposées horizontalement (verticalement), un agent ne peut pas se déplacer verticalement (horizontalement) mais par une succession de diagonales, on en revient donc au problème des carrés.

Le problème est le même que celui des carrés tant que l'agent ne se déplace que d'une seule case, mais dans le cas où l'agent désire faire un saut de plusieurs cases dans une direction donnée, il faut alors, pour l'environnement, calculer la case de destination. C'est à ce moment que la complexification des calculs se fait sentir. Dans le cas des cases rectangulaires, savoir si un point fait partie d'une case est un calcul simple par rapport aux coordonnées et dimension des rectangles. Dans l'optique des cases hexagonales, il faut faire un calcul plus important à cause de la géométrie même des hexagones.

Nous avons entrepris dans Maleva de réécrire l'algorithme même de recherche de la case de chute. Afin d'optimiser les calculs, une première approximation doit retourner un ensemble de cases possibles (entre 7 et 15) et un second calcul sur les distances doit retourner la case. Nous obtenons donc, pour deux discrétisations différentes, deux algorithmes différents de recherche de case.

Bien que ceci paraisse être un problème de développement, on peut commencer à voir tout le problème de la définition d'un environnement supposé générique. Car si la généricité apporte la facilité de définition, d'utilisation et de maintenance, ces facilités s'obtiennent aux dépends de la fonctionnalité. Les calculs précités font intervenir des nombres réels ce qui en temps de calcul n'est pas négligeable contrairement aux calculs sur des nombres entiers (cas des cases rectangulaires).

Les environnements continus.

Par opposition aux environnements discrets, on peut utiliser des environnement continus. Ce qui différencie le plus ces deux types de mondes, c'est la définition même de l'environnement mais c'est aussi les interactions qui vont avoir lieu entre les agents et l'environnement et les calculs qui vont en découler.

La définition de l'environnement est un problème du fait que l'on manipule à la fois des données discrètes et continues. Cet aspect sera étudié dans les paragraphe Environnement support physique et Environnement support de données.

En ce qui concerne les interactions agents/environnement, on peut différencier deux types d'interactions dépendantes de la définition du monde. Dans le cas des environnements discrets, le rôle d'une case est double. Nous avons vu qu'elle servait de micro environnement pour l'agent qui s'y trouvait, et elle servait de contrainte pour les déplacements des agents. Ces rôles ne peuvent pas être modélisés dans un environnement continu. En effet on ne peut pas définir des contraintes de déplacement (mur, eau, montagne) ou des contraintes de propagation (absorption, diffusion), sauf dans les plates-formes dédiées à ce genre de simulation.

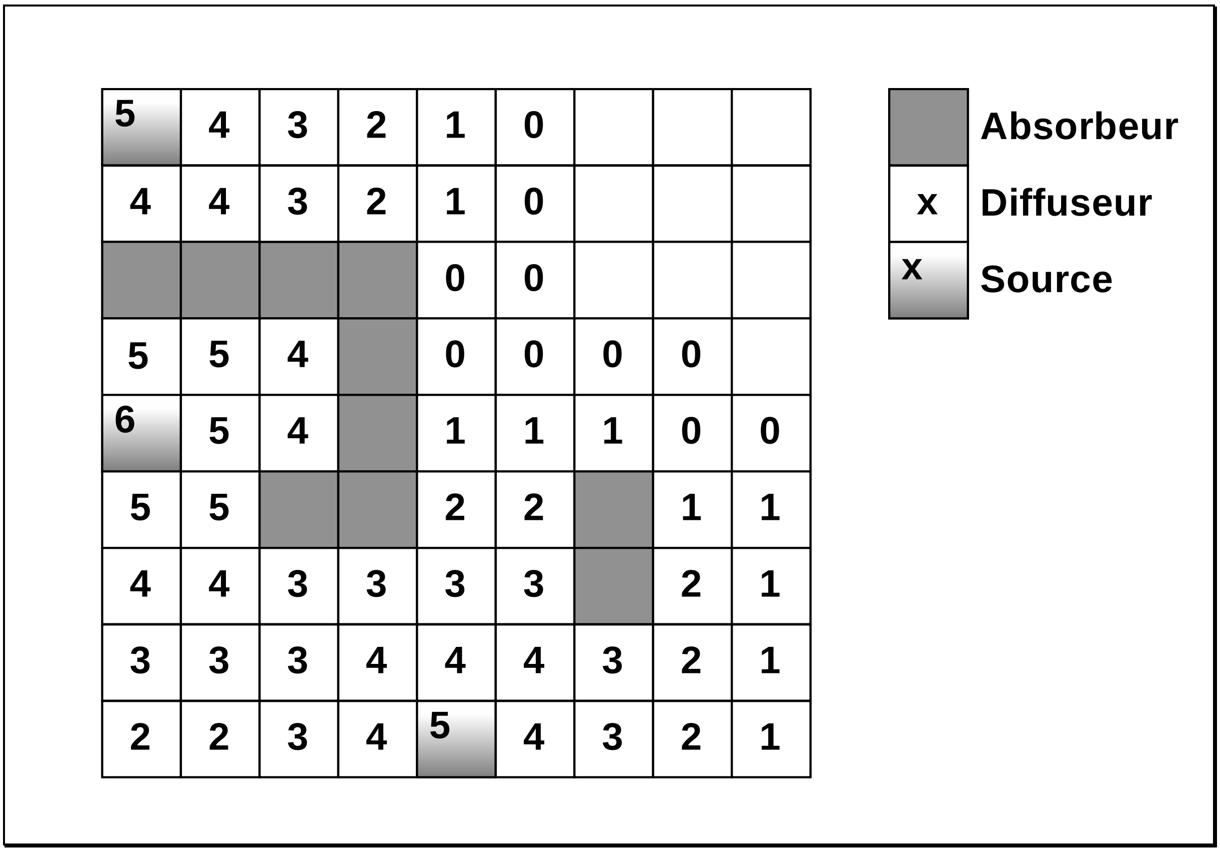


Figure 22: La propagation des stimuli dans Manta

A titre d'exemple dans la plate-forme Manta, où le rôle des stimuli est primordial[33] [Drogoul93A], la propagation des phéromones dépend de l'intensité des phéromones à un endroit donné - diffusion - et de la configuration spatiale de la fourmilière - absorption - (Cf. Figure 22). L'algorithme de propagation de ces stimuli est réalisé par propagation de gradient sur les cases. Cela permet de tenir compte à la fois des diffuseurs et des absorbeurs. On ne peut pas concevoir de faire ce type de propagation dans un environnement continu ou alors il faudrait faire des calculs très complexes et totalement prohibitifs en temps de traitement.

Nous avons donc pu entrevoir les rôles de chaque type d'environnement. On a ainsi pu s'apercevoir que chacun d'eux à des qualités et des défauts. Définir un environnement générique c'est alors prendre en compte les qualités

[33] La plate-forme Manta développée par A. Drogoul sert à simuler le comportement d'une fourmilière. Les agents manipulés sont donc des fourmis mais aussi des agents environnementaux tels que la lumière, l'humidité etc...

respectives des deux modèles en éliminant leurs défauts. Sans vouloir prétendre résoudre complètement le problème, nous avons sur Maleva modélisé l'environnement comme étant à la fois continu et discret.

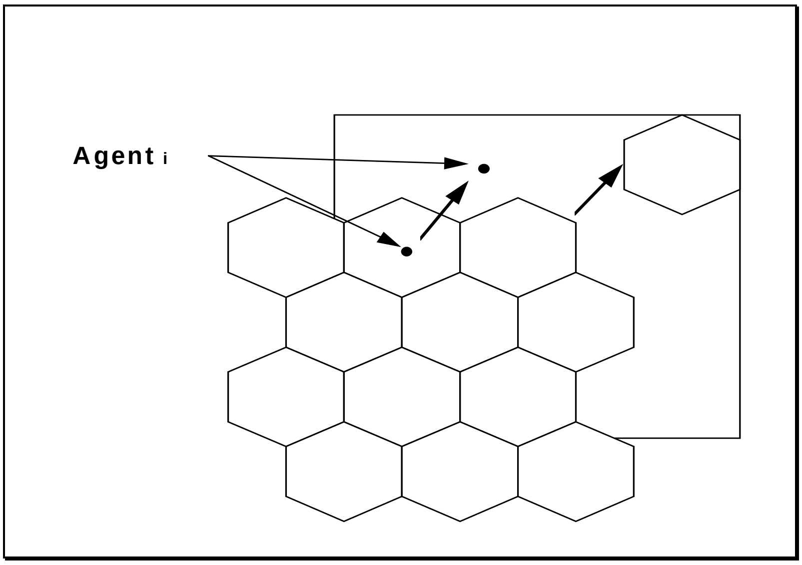


Figure 23: L'environnement multicouches de Maleva

Dans la première version de Maleva[34], l'environnement est multicouches (Cf. Figure 23), la couche inférieure correspond à un environnement continu, et les autres aux différentes discrétisations de celui-ci. Un agent placé dans l'environnement appartient à une couche supérieure (donc à une case), celle-ci appartient à la couche continue. Un agent n'a pas de connaissance de sa couche, il ne connaît que sa case. Toutes les interactions de l'agent seront faites par l'intermédiaire de sa case. La case, elle, est à même de faire correspondre les actions de l'agent à des actions correspondantes dans la couche continue. Par exemple, pour se déplacer d'une case vers le haut, l'agent envoie la commande ***déplacerN*** (pour déplacer vers le Nord) qui sera traduite par la case en une commande ***déplacer*** (possédant des arguments tels que l'orientation et la distance) qui pourra être comprise par la couche continue.

VIII.1.1.2 LES ENVIRONNEMENTS FINIS/TOROÏDAUX.

Quel que soit le type de simulation que nous allons être amenés à étudier, nous devons faire un choix sur les limites physiques du monde. Dans certains cas, un monde fini de taille relativement petite est suffisant. Il arrive cependant que l'on soit parfois tenu de définir un environnement de taille moyenne, voire de grande taille.

La taille d'un environnement est donné par le nombre de déplacements possibles des agents. J'appelle un environnement de petite taille un

[34] Nous traitons, ici, de la version Smalltalk de Maleva.

environnement dont la taille est inférieure à 25x25, de moyenne taille s'il est inférieur à 50x50. Pour les environnements de grande taille nous devons faire un choix, soit nous allons utiliser un environnement effectivement de grande taille, c'est à dire dont les distances sont grandes, soit nous allons utiliser un environnement toroïdal auquel cas il peut s'apparenter à un environnement infini.

Ici aussi nous sommes confrontés aux problèmes du choix d'environnement finis ou toroïdaux.

Les environnements finis sont des environnements qui contraignent les agents dans un espace déterminé par la taille de l'environnement.

Les environnements toroïdaux contraignent la surface de l'environnement sans contraindre les déplacements des agents qui s'y trouvent.

Ces contraintes d'espace vont donc déterminer le nombre maximal d'agents que peut contenir le monde si on considère que les agents ne peuvent pas se chevaucher.

Ce qui différencie le plus ces deux modèles d'environnement, c'est encore une fois la façon dont les agents peuvent accéder à certaines données. Les environnements finis possèdent une origine des coordonnées qui permet la localisation des agents et leur orientation. Les environnements toroïdaux peuvent aussi posséder une origine, mais celle-ci ne peut être utilisée par les agents car les localisations et les calculs des distances se font de manière différente.

VIII.1.1.3 LE CHOIX D'UN TYPE D'ENVIRONNEMENT

Après avoir étudié les différents types d'environnements possibles, leurs qualités et défauts, nous nous sommes penchés sur les paramètres qui permettent de déterminer quel type d'environnement choisir pour un type de simulation donné. Les critères de choix ne sont ni exhaustifs ni forcément judicieux dans la mesure où l'on peut passer d'un type d'environnement à un autre pour peu que l'on augmente la complexité de ceux-ci.

Comment choisir l'environnement correspondant à nos besoins:

- Pour le choix de la discrétisation .

La définition de l'environnement est-elle complexe? (i.e. le nombre de données manipulées est-il important). Si la réponse est oui il convient de s'orienter vers un environnement discret.

L'environnement est-il de grande taille? Si la réponse est oui, il convient alors de s'orienter vers un environnement continu, en effet le nombre de cases variant comme le produit de la largeur par la hauteur, la taille de l'environnement influe grandement sur la réponse.

Les déplacements des agents sont-ils contraints? Si oui il convient de prendre un environnement discrétisé afin que chaque case puisse gérer les contraintes des agents qui s'y trouvent.

L'environnement est-il actif? Si la réponse est oui, comme par exemple dans Manta avec la propagation de stimuli, l'utilisation d'un environnement discret simplifie énormément la tâche.

- Pour le choix fini/toroïdal.

L'environnement est-il continu? Dans le cas d'un environnement continu, la réponse importe peu bien qu'il soit plus naturel de prendre un environnement fini puisque la taille de l'environnement n'influe pas sur la taille des données. Dans le cas d'un environnement discrétisé, par contre, et compte tenu du fait que le nombre de cases est limité, il semble logique de prendre un environnement fini si l'espace n'a pas besoin d'être grand et un environnement toroïdal dans le cas contraire.

Le nombre d'agents est-il important? Suivant le type de simulation et de contraintes de mouvement des agents, si par exemple les agents ne peuvent pas simultanément occuper le même endroit, le nombre d'agents va naturellement influer sur la taille de l'environnement et par conséquent sur son approche discrétisée ou continue.

VIII.2 LE ROLE DE L'ENVIRONNEMENT DANS LA SIMULATION.

Le rôle de l'environnement dans une simulation est multiple. Un environnement peut servir de support de communication entre les différents agents. Il peut servir de support physique pour un agent mobile. Enfin, il peut servir de support de données pour les agents.

Chacun de ces rôles peut influer directement ou indirectement sur le modèle de l'environnement et, par conséquent, sur le choix que l'on va devoir effectuer pour déterminer s'il doit être continu/discret et fini/infini.

VIII.2.1 L'ENVIRONNEMENT SUPPORT DE COMMUNICATION.

L'environnement sert de support de communication lorsque les agents ne doivent pas (ou ne peuvent pas) communiquer directement entre eux.

La plate-forme Manta est un exemple d'un tel cas, les fourmis ne peuvent pas communiquer directement entre-elles, elles utilisent des phéromones. Bien que l'on puisse apparenter les stimuli à des objets du monde, il n'en demeure pas moins qu'il s'agit d'une forme de communication. La communication est alors locale (un stimulus ne se propage pas dans tout l'environnement), partielle (tous les agents du monde ne perçoivent pas tous les stimuli), temporaire (elle dépend de la durée de vie du stimulus) et décroissante (la valeur d'un stimulus décroît jusqu'à ce que le stimulus disparaisse). L'environnement est fini et discret, la propagation des stimuli s'effectue comme indiqué sur la Figure 22.

On pourrait cependant imaginer un mode de communication différent. Par exemple, si nous prenons un émetteur visuel, alors la communication est locale (la source est localisée), totale (tous les agents peuvent la voir), sa durée est considérée illimitée et sa distance de perception peut être considérée comme infinie. Dans un tel cas, l'environnement pourrait être continu et infini, les données sont calculables (elles sont constantes) en n'importe quel point de l'environnement et à n'importe quel moment.

Comme nous le voyons, le type de communication peut influencer la nature et le choix de l'environnement

VIII.2.2 L'ENVIRONNEMENT SUPPORT PHYSIQUE.

Dans certains types de simulation de SMA l'environnement ne sert que pour définir les contraintes physiques du fonctionnement des agents. Ces contraintes peuvent prendre différentes formes. Par exemple dans l'étude des systèmes proie-prédateurs, l'environnement n'a quasiment aucun rôle, sauf peut être celui de permettre de localiser un agent et donc de calculer sa position par rapport à un autre. En effet, si l'on prend un environnement fini, qu'il soit discret ou continu importe peu, la distance entre deux agents se fait uniquement par une simple différence entre les positions respectives de chacun d'eux (il faut bien entendu avoir une origine). Mais dans le cas un environnement toroïdal (donc considéré comme infini au niveau des déplacements), le calcul devient plus complexe puisqu'il faut alors faire un calcul pour les quatre directions, (Nord, Sud, Est et Ouest) comme nous pouvons le voir sur la Figure 20, et, par conséquent, connaître la taille de l'environnement.

On peut donc voir sur un exemple très simple que la nature de l'environnement influe sur la complexité des calculs, mais surtout ce que l'on doit retenir, ou plutôt la question que l'on doit se poser, c'est: "**un agent défini, modélisé et construit en dehors de toute contrainte environnementale doit-il et peut-il fonctionner dans n'importe quel type d'environnement ?**"

Les calculs doivent-ils être laissés à l'agent où doivent-ils être faits par l'environnement ?

Revenons à la définition de l'environnement. J'ai dit précédemment que pour ce genre de calcul, le fait que l'environnement soit continu ou discret importait peu. Ce n'est malheureusement pas toujours le cas, car si un environnement continu peut se ramener à un environnement discret où le pas de discrétisation est très fin, il reste le problème de la définition de l'environnement qui alors ne s'effectue pas de la même façon.

Un autre exemple provenant cette fois ci de la plate-forme de simulation ALIA (développée par M. Claude Delaye). Sur cette plate-forme, et dans l'étude de l'évolution et l'adaptation de robots miniers sur une planète (Mars), l'environnement est continu mais sa définition est discrète. En effet, dans l'étude que j'ai entreprise avec M. Delaye, la nature du sol était un facteur très important, il nous a fallu définir des zones de gisement, ce qui se fait très facilement sur un environnement discret puisque les données manipulées sont discrètes (présence ou absence et quantité de minerais sur une case), mais il a fallu aussi définir le relief de l'environnement ce qui dans un environnement continue revient à définir une altitude pour un point donné.

Pour nous faciliter le travail nous avons été contraint à discrétiser l'environnement afin de pouvoir utiliser un éditeur de relief qui permette une définition du monde qui soit aussi bonne possible mais pas pour autant fastidieuse. Nous pouvons donc voir que si dans certains cas utiliser un monde continu s'avère un choix utile, la définition de celui-ci pose de grands problèmes.

VIII.2.3 L'ENVIRONNEMENT SUPPORT DE DONNEES.

L'environnement sert aussi de support de données, dans les systèmes Multi-Agents, l'environnement sert à localiser les objets[35] et il permet aux agents d'accéder à ces objets.

Le rôle de l'environnement est donc de permettre aux agents d'accéder aux objets qui le composent et de les manipuler. L'environnement sert donc de base de données pour les agents.

Nous allons donc tenter dans ce paragraphe d'étudier les principes d'accès aux données.

Nous avons vu au paragraphe précédent que les données peuvent être de type numérique, par exemple une localisation, un calcul de distance. Ces données sont propres au modèle de l'environnement.

[35] Le terme objets fait référence aux objets appartenant à l'environnement mais qui ne définissent pas celui-ci.

Les données peuvent aussi être propres aux paramètres de l'environnement, il peut s'agir du relief, du taux d'humidité ou de luminosité.

Enfin les données peuvent être propres aux objets de l'environnement, il peut s'agir de minerais, d'agents.

Nous pouvons donc dire que l'environnement est caractérisé par un modèles, par des paramètres et par des objets. Donc nous devons étudier les modèles de perception des agents en fonction de ces caractéristiques.

VIII.3 L'ENVIRONNEMENT DANS MALEVA

Comme nous avons pu le voir, le choix et la définition d'un environnement est un processus très complexe. L'expérience de Maleva dans sa première version nous a servi. Il est quasiment impossible de prétendre définir un environnement générique qui puisse à la fois servir de support pour des données discrètes et des données continues. L'approche multi-couche, même si elle reste séduisante est trop complexe à gérer et elle impose un surcoût de calcul proportionnel au nombre de couches.

Nous avons alors modifié notre approche avec la version Delphi de Maleva. Etant donné que nous nous plaçons dans un modèle componentiel, l'environnement est par conséquent un composant[36]. Nous avons donc, et cela devient naturel dans notre approche, plusieurs types d'environnement. Nous proposons de base, un environnement discret avec discrétisation totale carrée, un environnement discret avec discrétisation partielle et un environnement continu. Dans les deux premiers composants, le fait que l'environnement soit fini ou toroïdal est une propriété (un attribut) du composant.

Bien qu'apparemment faisant partie de l'agent, **les composantes de perception et d'action sont définies en fonction de l'environnement et pas en fonction de l'agent**[37].

Par conséquent, pour chaque type d'environnement, il existe un composant capteur et un composant effecteur associé.

Ces composants permettront à n'importe quel agent de changer d'environnement à condition de changer pour des capteurs et effecteurs correspondants.

[36] Dans l'implémentation, et compte tenu des problème de conception, il ne s'agit pas d'un composant Maleva mais d'un composant Delphi. Nous verrons la différence dans le chapitre suivant.

[37]Vu de l'agent, un composant de perception ne sert que de filtre entre les données perçues et les données traitées.

IX MALEVA

Nous avons consacré un chapitre entier à Maleva et au langage Delphi pour permettre d'étudier le langage de développement et la réalisation qui a été faite. Nous étudierons dans un premier le langage de base, c'est à dire Delphi et nous verrons à ce titre quelles sont ses particularités. Dans un deuxième temps nous nous intéresserons à l'implémentation de Maleva et présenterons les outils qui sont proposés.

IX.1 PRINCIPES DE CONCEPTION

Le principe qui nous a guidé dans la conception des différentes versions de Maleva à été en premier lieu d'utiliser le langage de programmation tel qu'il est défini. Nous n'avons pas voulu réécrire de nouvelles descriptions de classes comme cela est possible dans les langages réflexifs (Smalltalk, Mering et PtitLoo). Il eut été en effet assez simple de créer une nouvelle syntaxe de définition de classe et de composant mais nous avons voulu permettre aux futurs utilisateurs de rester dans le cadre du langage d'origine afin de pouvoir réutiliser les classes existantes. Les seules modification permises dans la définition d'une classe consistent en l'ajout de champs permettant une description sémantique du composant, ce que nous verrons plus loin.

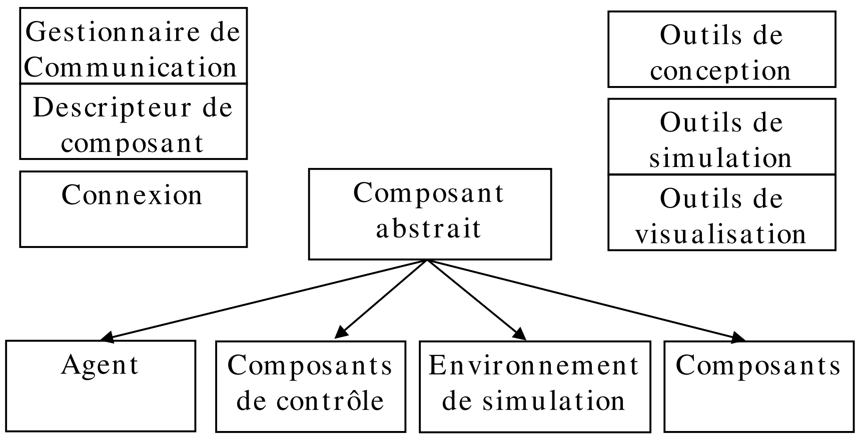


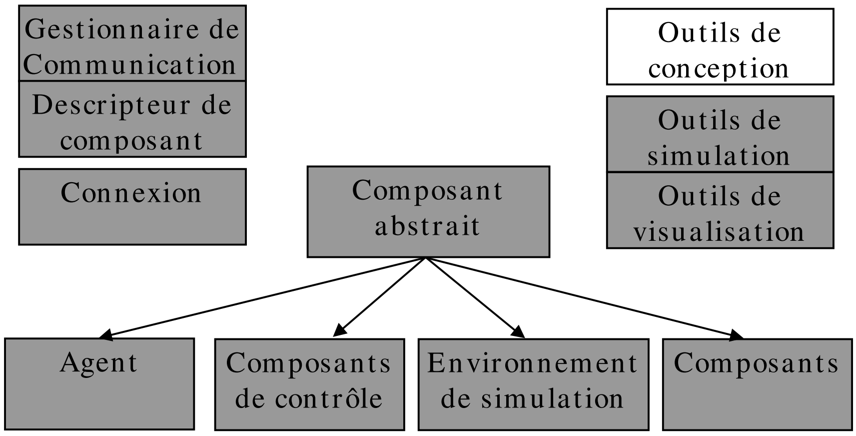
Figure 24: Les grandes composantes de Maleva

Maleva est une application écrite autour de 3 grandes parties :

- un noyau, l'ensemble des classes de base permettant la création de composants logiciels et la communication inter-composants.
- un ensemble d'outils de conception, des classes et des composants permettant de faciliter le travail de conception de SMA

- un ensemble d'outils de simulation, des classes et des composants permettant de faciliter la simulation de SMA.

IX.1.1 MALEVA : UN OUTIL DE CONCEPTION DE COMPOSANTS



L'approche visuelle de la programmation ne peut pas se faire sans un ensemble d'outils de conception. L'obtention d'une plate-forme de simulation nécessite des outils de simulation et de visualisation des différents éléments de la simulation.

Les outils proposés dans Maleva dépendent essentiellement du langage d'implémentation. Un langage tel que Smalltalk offre un grand nombre d'outils permettant le traitement des instances d'une classe. Il dispose d'inspecteurs d'objets et de deboggeurs intégrés. Dans Delphi qui, rappelons-le, est un langage compilé non réflexif, nous avons dû concevoir quelques outils de base que nous développerons plus loin.

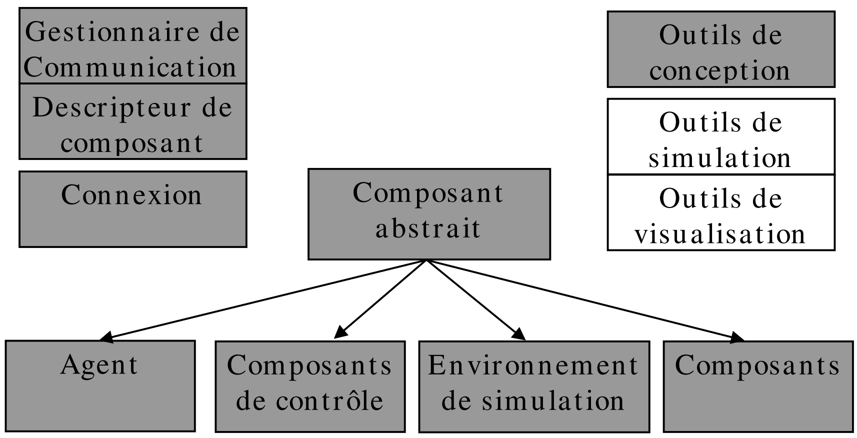
Les principaux outils présents dans la plate-forme Maleva sont :

- Outil permettant la création des composants. Il est parfois difficile de demander à l'utilisateur de concevoir un composant, il devrait alors connaître la syntaxe de définition de la classe, la ou les méthodes de conception des bornes d'entrée/sortie... Pour l'aider dans cette tâche nous avons développé sous Delphi un outil permettant de générer un squelette de classe. Dans la version Smalltalk cela n'existe pas car l'éditeur facilite déjà ce travail.

- Outil permettant la création et la manipulation des connexions. Dans la version Smalltalk nous avons utilisé Editalk un éditeur de graphe développé par R. Voyer et gracieusement mis à notre disposition. Dans Delphi nous avons développé un éditeur de connexion et sommes en train de concevoir un éditeur de graphe. Le rôle de ces éditeurs est de permettre à l'utilisateur de définir visuellement les connexions entre les composants (d'où le nom parfois abusif de programmation visuelle).

- Outil permettant de visualiser, modifier les attributs d'un composant. En Smalltalk, nous possédons déjà les inspecteurs d'objets qui permettent de visualiser et modifier l'ensemble des attributs d'une instance. Cet inspecteur est d'ailleurs critiquable dans le sens où il ne respecte pas quelques-uns des principes des Langages Orientés Objets. En premier lieu, l'inspecteur d'objets permet l'accès à l'ensemble des attributs d'un objet (le langage ne permet pas de définir des attributs privés). En deuxième lieu cet inspecteur permet de modifier ces attributs en y accédant directement alors que, normalement, il devrait utiliser des méthodes d'accès à ces attributs. Quoi qu'il en soit nous avons utilisé cet inspecteur pour permettre aux utilisateurs d'accéder aux structures des composants et des agents. En Delphi il existe aussi un inspecteur d'objet qui malheureusement ne peut pas être utilisé dans une application compilée (cet inspecteur ne peut être utilisé que pour la modification des propriétés d'un composant avant compilation). Nous avons donc développé un éditeur d'objet qui permet d'effectuer les mêmes opérations mais après compilation. A la différence de Smalltalk, Delphi permet de différencier les attributs privés, protégés, publics et publiés d'un objet. L'éditeur d'objet peut alors ne rendre accessible aux utilisateurs qu'une catégorie d'attribut (les attributs publiés que l'on nomme propriétés). Seuls les propriétés sont accessibles par l'inspecteur et ce uniquement au travers de méthodes d'accès en lecture et en écriture pour chaque attribut. Nous reviendrons sur les propriétés par la suite.

IX.1.2 MALEVA : UN OUTIL DE SIMULATION



Les outils de simulation qu'une plate-forme SMA doit proposer sont d'une part des outils permettant l'activation des agents et d'autre part l'accès à un environnement de simulation et des outils pour la gestion des agents (sauvegarde...). Ces outils ne sont intégrés que dans les versions Smalltalk et Delphi.

Dans la version Smalltalk, un environnement est un composant Maleva, il est donc activé comme les agents et peut être remplacé dynamiquement par n'importe quel autre composant. Dans la version Delphi il s'agit d'un composant Delphi c'est-à-dire un composant d'interface utilisateur.

Nous ne développerons pas plus les principes des environnements de simulation et nous renvoyons le lecteur au §V. Nous pouvons cependant dire que Maleva propose un panel assez complet de type d'environnement tels que les environnements continus, discrets, finis ou infinis (toroïdaux), de plus Maleva propose un ensemble de classes permettant de développer de nouveaux environnements.

Outre l'environnement de simulation, Maleva propose un composant nommé Scheduler dont le rôle est d'activer les composants à tour de rôle en mode séquentiel ou parallèle. Nous avons vu que les composants Maleva étaient fondés sur un modèle de communication distinguant les flux de données des flux de contrôle, le rôle du Scheduler est donc tout simplement d'initier les flux de contrôle des composants.

IX.2 LE NOYAU DE L'APPLICATION

L'architecture de Maleva est quasiment constante quel que soit le langage hormis les différences dues à la nature même du langage (notions de métaclasse, utilisation de la réflexivité, parallélisme, langage typé ou non ...).

Le noyau de Maleva se décompose suivant le schéma suivant :

- Le niveau Gestionnaire de Communication associé au Descripteur
- Les Connexions.
- Le niveau Composant avec ses sous classes.

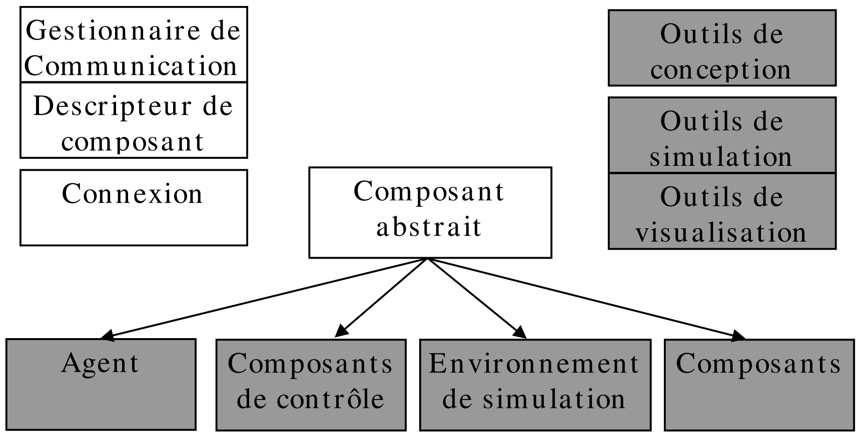


Figure 25: Le noyau de Maleva(en clair)

IX.2.1 LE NIVEAU GESTIONNAIRE DE COMMUNICATION ET LES CONNEXIONS

Les composants communiquent entre eux par l'intermédiaire de canaux de communication qui sont réifiés par des connexions. Chaque composant est associé à un gestionnaire de communication qui lui est propre.

Le Gestionnaire de Communication est une classe, qui prend en charge l'ensemble des connexions entre le composant auquel il est associé et les autres composants du système. Le Gestionnaire de communication est responsable de l'acheminement des messages vers les autres composants (les autres Gestionnaire de Communication), de la réception des messages, de la gestion des protocoles de communication s'il y a lieu.

Une connexion est une structure contenant les références sur l'émetteur (Client ou Cl), le récepteur (Serveur ou Se), la méthode appelante, la méthode appelée, les paramètres (Pa), le protocole (Pr) d'envoi de message (synchrone, asynchrone) et le résultat (Re) par défaut. On peut donc symboliser une connexion par le Tuplet <Cl, Se, Mo, Mi, Pa, Pr, Re> où Mo représente la méthode appelante (Méthode Output) et Mi la méthode appelée (Méthode Input). Mo correspond alors à la borne de sortie du composant émetteur et Mi à la borne d'entrée du composant récepteur.

L'envoi de message entre deux composants se fait suivant le schéma suivant :

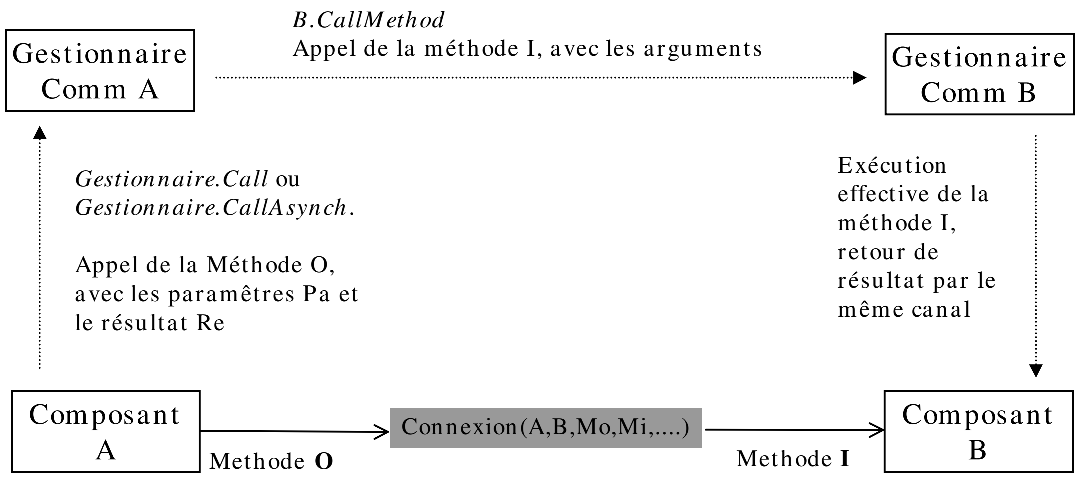


Figure 26: Schéma général d'envoi de message entre deux composants

Sur la figure précédente, nous voyons le schéma général simplifié d'envoi de message entre deux composants (nous avons schématisé un appel de type Point à Point), en ligne pointillé sont représentés les messages effectivement envoyés, en ligne continue la connexion définie par l'utilisateur.

La méthode O du composant A fait un appel auprès de son gestionnaire en effectuant l'appel à *Gestionnaire.Call(...)* ou l'appel à *Gestionnaire.CallAsynch(...)* ce qui permet de définir d'une part le protocole de communication (synchrone ou asynchrone), et d'autre part la méthode appelante (**Mo**) ainsi que les paramètres **Pa** et le résultat par défaut **Re**. Connaissant A et Mo, le Gestionnaire de Communication peut alors déterminer la ou les connexions dont les deux premiers arguments sont A et Mo et retrouver **B** et **Mi**. Avec ces connexions, il peut alors envoyer le message au gestionnaire de communication de B qui lui même demandera l'exécution effective de la méthode **Mi** auprès de son composant.

Envoi de message en mode Synchrone

Dans le cas d'un appel synchrone, l'envoi de message est basé sur le modèle objet standard, le gestionnaire de communication A appelle la méthode *CallMethod* du gestionnaire de communication B, le résultat de l'appel de Mi est renvoyé au gestionnaire de communication de A par le même canal, le résultat par défaut **Re** est alors remplacé par le nouveau résultat (en cas d'échec le résultat Re est gardé tel qu'émis par A) et le contrôle est rendu à A en lui indiquant qu'il y a eu un problème.

Envoi de message en mode Asynchrone

Dans le cas d'un appel asynchrone, le schéma d'envoi de message est légèrement différent. Dans ce cas les gestionnaires de message des composants doivent posséder une boite aux lettres susceptible d'entreposer les messages en attente de traitement. Le schéma devient donc ceci :

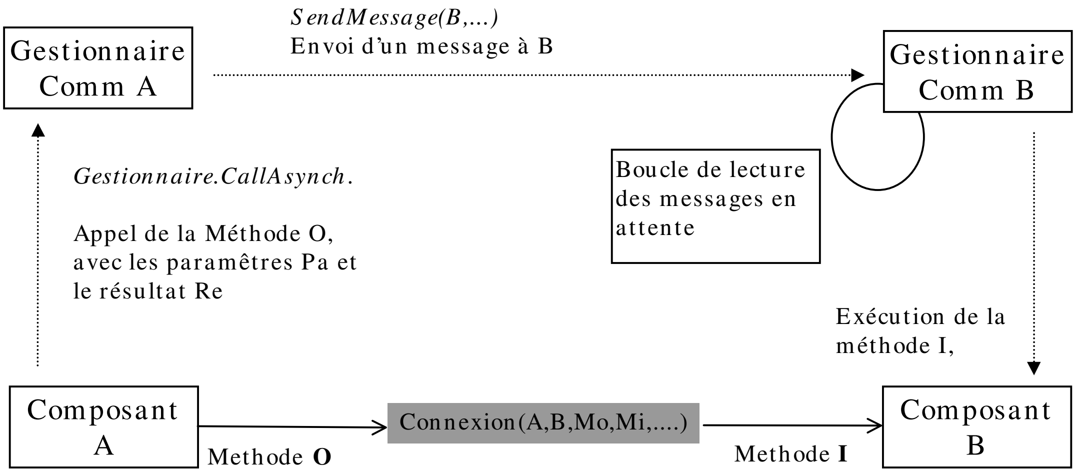


Figure 27: Schéma d'envoi de message Asynchrone entre deux composants

L'appel de *CallAsynch* provoque non plus l'appel de la méthode *CallMethod* vers le receveur mais un envoi de message (*SendMessage*[38]) à celui-ci. Le message qui contient les mêmes paramêtres que dans l'envoi de message synchrone est alors mis dans une boite aux lettres. La boite aux lettres est lue périodiquement par un processus indépendant qui provoque en cas de message en attente un traitement identique à celui d'un envoi de message synchrone à ceci près que nous avons changé de contexte d'exécution.

Le résultat doit être renvoyé par un autre canal (une autre connexion), ce qui revient à faire un envoi de message avec continuation implicite ou explicite.

[38] SendMessage est un envoi de message système propre à Windows.

Apports de ce mécanisme

1. Les schémas précédents permettent de voir que A ne connaît ni B ni Mi, ceci permet alors de remplacer, en cours d'exécution, B par n'importe quel autre composant ou Mi par n'importe quelle autre borne d'entrée. En cas d'échec, par exemple si le composant receveur n'existe plus, le résultat par défaut étant renvoyé, le composant émetteur peut continuer son activité sans problème.

2. On peut dès lors effectuer un envoi de message vers plusieurs destinataires (envoi de message en mode Multicast), il faut pour cela que le gestionnaire de communication gère l'envoi vers ceux-ci et surtout la réception des résultats. Dans le cas d'un envoi de message synchrone il s'agit simplement d'une mise en commun des résultats Re, dans le cas d'un envoi asynchrone on procède de même en synchronisant les retours des résultats.

3. On peut facilement changer les protocoles de communication entre composants en changeant uniquement les gestionnaires associés à ces composants. Nous avons par exemple deux classes de gestionnaire, l'une qui traite le cas du Multicast et l'autre les envois Point à Point. Dans le deuxième cas, ce gestionnaire n'est utilisé que si la connexion entre Mo et Mi est unique ce qui permet d'optimiser le temps de transfert d'information. Enfin l'utilisation d'un gestionnaire indépendant permet d'intégrer des composants provenant ou s'exécutant sur d'autres machines ou d'autres systèmes d'exploitation comme nous le verrons en étudiant les objets COM.

IX.2.2 LE NIVEAU DESCRIPTEUR OU META

Le niveau Descripteur de Maleva, c'est la partie description fonctionnelle d'un Composant. Si nous voulons pouvoir associer (composer) des composants pour qu'ils puissent travailler ensemble, il faut définir les interactions entre ces composants. Les connexions relient des bornes de sorties (Fonctions ou Méthodes de sorties) à des bornes d'entrées (Fonctions ou Méthodes d'entrées) de plusieurs composants.

Décrire de manière fonctionnelle les composants, c'est offrir l'ensemble des fonctions accessibles au niveau du composant (Cf. Figure 28: Le rôle du Descripteur de composant). Comme l'ensemble des fonctions d'un composant ne correspond pas forcément aux fonctions accessibles de celui-ci (les fonctions à usage interne ne doivent pas être accessibles ni visibles de l'extérieur), il convient de pouvoir les décrire. C'est le rôle du niveau Descripteur.

Pour simplifier nous pouvons dire que le rôle du Descripteur est de permettre de connaître l'ensemble des fonctions d'entrées et des fonctions de sorties d'un composant et leur prototype (nombre de paramètres, type des paramètres). En ce sens nous pouvons dire que le descripteur joue le rôle de méta-classe (au sens de méta-classe à la Smalltalk).

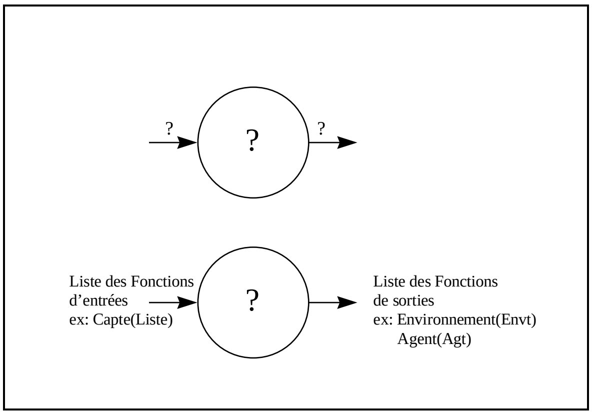


Figure 28: Le rôle du Descripteur de composant

Un autre rôle du descripteur est de décrire le composant lui-même. En programmation visuelle on utilise une représentation graphique explicite qui est en général une icône, une image ou toute représentation graphique qui puisse sembler suffisamment explicite. Dans Maleva, un composant à une représentation graphique sous forme d'une image qui est censée représenter le rôle du composant.

Pour ce qui est des composants nécessaires à la conception d'un agent, cette image contient un > à gauche s'il s'agit d'un composant de type Capteur, un ^ s'il s'agit d'un composant Comportemental et enfin un > à droite s'il s'agit d'un composant Effecteur. Cependant à l'usage on s'aperçoit que ce genre de représentation est nécessaire mais pas suffisante, en effet une image représentant un poisson avec un > à gauche représente un capteur mais représente-t'elle un capteur de poisson (un capteur qui capte des agents poissons) ou un capteur spécialisé pour un agent poisson ?

On s'aperçoit donc qu'une représentation sous forme d'image n'est pas suffisante, il faut pouvoir y ajouter une description sémantique. Le rôle du Descripteur est donc d'associer à un composant mais aussi aux connexions une description de leur fonction.

Dans la version Smalltalk, le descripteur est une classe capable de retrouver les méthodes externes (les bornes du composant) d'un composant, celles ci sont regroupées dans deux protocoles nommés InputMethods et OutputMethods. Chacune de ces méthodes possède une description (sous forme d'un commentaire) permettant de leur associer une description sémantique.

Les versions PtitLoo et Mering IV permettent d'utiliser les mêmes principes que la version Smalltalk car elles bénéficient aussi d'une réflexivité permettant de redéfinir les notions de classes et de méthodes. Définir une classe ou définir une méthode revient alors à définir leur contenu (le code) et leur fonction (la description). Les classes ainsi définies restent totalement compatibles avec les classes existantes.

La version Delphi est différente car il s'agit d'un langage compilé n'offrant pas de possibilités réflexives. Dans cette version, les descriptions sémantiques d'un composant ne sont pas définies dans la classe même mais à l'extérieur de celle-ci. Nous verrons plus loin comment un composant peut associer une description sémantique à ses bornes d'entrée/sortie.

IX.2.3 LE NIVEAU COMPOSANT

Un composant comme nous l'avons déjà vu est avant tout un objet. Le composant correspond au comportement de l'entité que nous tentons de modéliser. Son rôle revient alors à traiter les interrogations faites par ces bornes d'entrées et si besoin est à demander un complément d'information par l'intermédiaire des ses bornes de sorties.

L'autonomie du composant vis-à-vis des autres est assurée par sa boucle d'activité interne. Ses interactions avec les autres composants sont assurées par ses connexions.

Dans la version Smalltalk, les composants sont sous-classe de la classe ActorBehavior définie dans le noyau Actalk (Cf. §III.1.2.2). Les composants ont donc toutes les particularités des acteurs de même que dans la version Mering IV. Dans la versions PtitLoo un composant est un objet classique descendant directement de la classe Objet.

Dans la version Delphi les composants sont des objets particuliers descendants de la classe TComponent (classe des composants de base du langage Delphi) afin de bénéficier de certaines particularités de leurs ancêtres, mais comme nous l'avons dit les composants Delphi ne sont pas tout à fait des composants tels que cette thèse les décrit (ils ne permettent pas la composition), nous avons donc créé une nouvelle classe de composant nommée TSMAComponent dont tout descendant est un composant Maleva possédant son Gestionnaire de Communication et des bornes d'entrée/sortie de base.

IX.3 VERSION PASCAL (DELPHI)

L'utilisation de Smalltalk permet une conception rapide au détriment d'une exécution lente. La programmation componentielle (ou modulaire) a ceci comme défaut qu'elle multiplie le nombre de processus ce qui a comme conséquence de ralentir très nettement le système. Par exemple, sur notre exemple de conception componentielle d'agents proie et prédateurs, nous avons entre 5 et 8 composants par agent, ceci multiplié par une centaine d'agents (cela dépend bien entendu de la taille de l'environnement) revient à faire s'exécuter entre 500 et 800 processus simultanément, sans compter les interactions avec l'utilisateur. Maleva Smalltalk si elle bénéficie d'un langage très puissant est grandement handicapée par le fait que Smalltalk veut rester totalement indépendant du système d'exploitation dans le but d'être portable. Or l'utilisation d'outils externes (DLL, liens DDE, liens OLE...), qui est un apport indéniable voire indispensable, ne peut se faire qu'au détriment de la portabilité même du langage.

Autrement dit, nous ne pouvons à la fois utiliser ou intégrer des outils externes et fonctionner sur différents systèmes. Nous avons donc envisagé de construire une nouvelle plate-forme Maleva dont les prétentions sont d'offrir à la fois une grande facilité de développement avec une exécution rapide et la possibilité d'intégrer des outils provenant d'autres langages.

Mon choix s'est alors naturellement porté sur le langage Delphi. L'utilisation de ce langage a été en grande partie liée au fait que Delphi est un langage de programmation visuelle, il est rapide et efficace et surtout il intègre la notion de composants même si ceux-ci sont différents de ceux définis dans Maleva.

Delphi est un langage Pascal développé par Borland, dont le but est de faciliter la programmation sous environnement Windows. Delphi offre certaine particularités supplémentaires par rapport au Pascal traditionnel.

Les atouts de Delphi sont nombreux :

- En premier lieu Delphi est un langage visuel bi-directionnel (la modification d'un composant est répercutée sur le code généré et inversement) simple permettant la génération d'exécutables très performants.
- Delphi permet d'intégrer directement dans le code des composants écrits dans différents langages tels que VisualBasic, Visual C++, il intègre des possibilités d'utilisation de liens DDE, OLE qui nous serviront pour communiquer avec d'autre applications.
- Bien qu'étant un langage sans notion de méta-classe, Delphi propose quelques fonctions méta telles que l'accès aux attributs (l'équivalent de AllInstanceVarNames de Smalltalk) et méthodes d'un objet.
- Delphi permet une totale distinction entre l'objet et son interface d'utilisation. Cette distinction sera une des clefs de la programmation sous Maleva.

IX.3.1 UN LANGAGE DE DEFINITION DE COMPOSANTS

Un des éléments de base de la programmation sous Delphi est le composant. Les composants Delphi sont de deux types, les composants visibles (composants d'interface) et les composants invisibles (composants d'application ou composants de contrôle).

Les composants d'interface (ils servent à créer des interfaces d'application) sont des composants visibles dont le rôle est d'interagir avec l'utilisateur, comme tout langage de ce type, Delphi propose tout un panel de composants tels que les boutons, les listes, les boites à cocher etc...

Les composants d'application sont des composants invisibles ayant les mêmes particularités que les composants d'interface, mais qui ne peuvent pas interagir avec l'utilisateur. Delphi propose par exemple des composants liés aux communications entre applications ou à l'intérieur d'une même application (liens DDE, OLE), des composants d'accès aux bases de données...

Cette distinction entre les deux types de composants est avant tout due au système d'exploitation Windows et à son système de fenêtrage, tout composant visible possède son propre gestionnaire de message et occupe par conséquent plus de mémoire et de ressources qu'un composant invisible. Par souci d'efficacité et parce qu'il n'ont pas besoin d'être visibles, les composants Maleva sont des composants d'applications. Comme les gestionnaires de message sont amenés à envoyer et recevoir des messages, ils s'apparentent aux composants d'interface sans pour autant interagir avec l'utilisateur. Enfin certains composants tels que les composants Scheduler et Environnement, que nous étudierons plus loin, sont quant à eux des composants d'interface.

Donc les composants Maleva sous Delphi, sont soit des composants de contrôle, soit des composants d'interface, soit des composants hybrides. Mais rassurons-nous, pour les futurs concepteurs il n'y a aucune différence dans la manière d'implémenter ceux-ci.

Définition de Composants Delphi/Maleva

Un composant est conçu dans une unité (Unit) séparée. Dans cette unité nous dissocions la partie définition et visibilité de la classe (partie *interface*) de la partie implémentation (partie *implementation*). La partie interface sert à rendre visible l'objet du reste de l'application de même qu'elle sert à documenter les structures utilisées. La partie implémentation représente le code effectif des structures et des objets.

La syntaxe d'une unité (vide) se fait comme ceci :

```
Unit Nom de l'unité ;
Uses
   { Liste des unités utilisées } ;
Interface {Définition des structures visibles de l'extérieur}
Implementation {Implémentation des structures}
```

```
Begin
   {Corps de l'unité}
end.
```

Un composant est un objet sous classe de TComponent et possède des attributs et des fonctions aux propriétés qui seront définies par le concepteur.

La définition d'un composant Delphi ou d'un composant Maleva, qui se fait donc dans la partie Interface de l'unité, obéit à la syntaxe suivante :

```
Nom de la classe = class(Nom de la super classe)
private
   {Liste des attributs et Méthodes privées}
protected
   {Liste des attributs et Méthodes protégées}
public
   {Liste des attributs et Méthodes publiques}
published
   {Liste des attributs et Méthodes publiées}
   property Nom de la propriété :type
     [read Nom de la méthode d'accès en Lecture]
     [write Nom de la méthode d'accès en Ecriture];
end;
```

Exemple 1: Définition d'un composant Delphi/ Maleva

La définition d'une classe permet alors de spécifier quatre parties **private**, **protected**, **public** et **published** qui correspondent respectivement à rendre leur contenu invisible, partiellement visible, totalement visible ou faisant partie de l'interface de la classe. Hormis des considérations purement techniques, ces parties servent essentiellement à spécifier ce que peuvent ou ne peuvent pas voir les futurs utilisateurs de cette classe.

Par exemple, voici la définition de la classe générique de composant capteur et celle du Capteur de proie que nous avons utilisé (Cf. §IV.1.1) :

```
TGenericDetector=class(TSMANaireControlComponent)
private
   FDistance:single;        {Distance de perception}
   function GetFDistance : single ;
   procedure SetFDistance(distance :single);
protected
```

```
   MyAgent:TSMAAgent;        {Agent possesseur => capteur non
partageable}
   MyEnvironment:TSMAEnvironnement;          {Environnement}
   FType:TClass;            {Type d'agent à détecter}
public
   constructor Create(AOwner: TComponent); override;
published
   procedure SetAgentOwner(Param:PSMACallParam); virtual;
   procedure SetEnvironnement(Param:PSMACallParam); virtual;
   Procedure SetAgentClassToPerceive(param:...); virtual;
   Procedure SetDistanceOfPerception(param:...); virtual;
   procedure Detect(Param:PSMACallParam); virtual;
   procedure GetAgentDistance(Param:PSMACallParam); virtual;
   procedure GetAgentDirection(Param:PSMACallParam); virtual;
   property DistanceDePerception: single
            read GetFdistance
            write SetFDistance;
   end;
```

Exemple 2: Définition de la super-classe générique des composants capteurs

Jusqu'ici la définition d'un composant Maleva est totalement identique à celle d'un composant Delphi à une super classe près. Les composants Delphi héritent de *TComponent* tandis que les composants Maleva héritent de *TSMAComponent*.

La partie private définit un attribut (Fdistance) et deux fonctions (GetFDistance et SetFDistance) qui ne sont accessibles que par la classe elle même.

La partie protected définit des attributs qui ne seront accessibles que par les sous-classes.

La partie public définit ici une fonction spéciale (un constructeur) qui permet d'instancier la classe. Cette méthode est accessible par tout le monde.

Enfin la partie published propose un ensemble de méthodes qui représentent en Delphi la partie interface de la classe. Dans Maleva, cette partie va servir à définir les fonctions d'entrées et de sorties du composant.

En plus de ceci, nous avons une déclaration spéciale d'attribut :

```
property DistanceDePerception: single
            read GetFdistance
            write SetFDistance;
```

En Delphi nous disons qu'il s'agit d'une propriété qui rend publié l'attribut DistanceDePerception en lecture (GetFDistance) et en écriture (SetFDistance).

Autrement dit, lorsque nous faisons par exemple

if i=DistanceDePerception… (lecture de l'attribut) ou

DistanceDePerception := 10 (écriture de l'attribut)

en réalité nous appelons respectivement les méthodes GetFDistance et SetFDistance qui, elles, manipulent l'attribut caché Fdistance.

La fonction GetFDistance implémente ceci :

```
Function TGenericDetector.GetFDistance :single ;
begin
   result :=Fdistance ;
end ;
```

et la procédure SetFDistance implémente ceci :

```
Procedure TGenericDetector.SetFDistance(distance :single) ;
begin
   if distance>1000 then distance :=1000 ;
   Fdistance :=distance ;
end ;
```

La propriété DistanceDePerception est un attribut « virtuel » qui réprésente l'interface d'un attribut caché. Il fera en outre partie des attributs visibles et accessibles dans l'inspecteur d'objet que nous avons conçu et que nous verrons par la suite.

Cette particularité de ce langage est un des principes fondamentaux de la programmation componentielle qui distingue le contenu de l'objet de son interface. Le même principe peut s'appliquer à certaines méthodes liées à des événements utilisateurs. L'intérêt devient ici évident puisqu'il est possible de rendre accessible ou inaccessible un attribut et de plus il est possible de traiter les nouvelles valeurs avant que celles-ci ne soient effectivement affectées. Dans l'exemple précédent, nous obligeons la valeur distance à être inférieure à une valeur fixée a priori (ici 1000) et ce quels que soient les désirs de l'utilisateur.

Définition de nouveaux composants

Concevoir un nouveau composant de base revient alors à hériter d'un super composant existant comme suit :

```
Unit PreyDetector;
Uses {Liste des unitées}
Interface
TDetectorPrey=class(TGenericDetector)
private
protected
public
   constructor Create(AOwner: TComponent); override;
published
end;
```

Exemple 3: Définition de la classe capteur de Proie

Sur cet exemple nous voyons la définition de la classe de composant Détecteur de Proie. Comme nous pouvons le voir, cette définition est très succincte, l'essentiel ayant été déjà défini dans la super-classe.

Comme nous l'avons dit, jusqu'à présent la définition d'un composant Maleva correspond à celle d'un composant Delphi. Cependant créer des composants Maleva obéit à certaines règles que nous allons voir dans le paragraphe suivant.

IX.4 DE DELPHI A MALEVA

Pour concevoir des composants Maleva, nous avons deux possibilités, la première consiste à créer un composants Maleva classique descendant d'une classe prédéfinie et la seconde à transformer des composants Delphi (ou tout autre langage respectant le modèle) en composant Maleva.

Le passage de Delphi à Maleva se fait naturellement par utilisation des classes de base du noyau de Maleva (fichiers *Kernel*, *KernelControl*, *KernelDetector*, *KernelEffector*, *KernelAgent*...).

Dans Maleva, un composant est un composant Delphi auquel nous avons ajouté 1/ une structure de message unique, les messages seront véhiculés par les connexions, 2/ des méthodes qui permettent de différencier les méthodes classiques de celles définissant les fonctions d'entrées et de sorties de nos composants et enfin de donner une sémantique à ces fonctions.

IX.4.1 UNE STRUCTURE DE MESSAGE UNIQUE

Delphi est un langage typé, c'est-à-dire un langage où nous devons obligatoirement associer un type à une variable donnée. Certains y voient une contrainte et d'autres y trouvent à la fois de la clarté dans la lisibilité et surtout une assurance de ne pas faire n'importe quoi.

Le fait que nous utilisions un langage typé induit automatiquement une contrainte de format de message entre composants. Si nous ne voulons pas limiter le type de paramètres que nous utiliserons lors d'un appel de méthode (c'est le cas par exemple pour les événements systèmes sous windows) nous devons créer des structures générales qui englobent les différents types de données possibles.

Dans Maleva nous avons les connexions qui ne peuvent pas contraindre la nature des données qui vont y transiter. Nous avons donc défini une structure générale TSMACallParam (définie dans le fichier *Kernel.pas*) qui regroupe les différents types de données que nous sommes à même de rencontrer dans les SMA. Cette structure n'est pas exhaustive et on peut naturellement la modifier.

```
type
   PSMACallParam = ^TSMACallParam; {pointeur sur la structure}
   TSMACallParam
      resultList:TList;
{Un représentant pour les listes, ex. agents perçus par les
capteurs}
      identification:string;
{Un représentant pour les chaines de caractères, ex. Un nom
d'agent}
      point:TPoint;
{Un représentant de point, ex. la position d'un agent}
      dist:single;
{Un représentant des valeurs réelles, ex. la distance d'un
agent}
      Sender,Owner:TSMAModule;
{Deux représentants des composants, ici l'émetteur du message
et son responsable (celui qui gère la mémoire)}
      p:pointer;
{Un représentant pointeur non typé permettant l'accès à des
données complexes, ie. Un objet}
end;
```

Exemple 4: Définition de la structure d'un message

Chaque composant Maleva qui défini des bornes (fonction) d'entrée/sortie devra utiliser le type PSMACallParam qui représente un pointeur sur la structure TSMACallParam. Nous pouvons voir l'utilisation de ce type dans la définition de la classe TGeneriqueDetector (Cf. Exemple 2, partie published).

IX.4.2 CONCEPTION D'UN COMPOSANT MALEVA

Comme en Delphi, un composant Maleva doit être défini dans une unité. Chaque unité permet de définir une ou plusieurs classe de composants avec leurs représentations. Une unité peut faire référence à d'autres unités et l'ensemble des classes présentes dans chaque unité seront utilisables dans l'application compilée.

Les étapes de conception sont les suivantes:

1.Création de l'unité

2.Création de la classe.

3.Identification et enregistrement des fonctions d'entrées, sorties.

4.Identification et enregistrement du composant.

5.Association d'une image au composant.

La plupart de ces étapes sont automatisées et il n'y a en général rien a écrire, mais afin que les futurs utilisateurs sachent comment faire nous nous attarderont sur chacune d'entre elle. De plus nous montrerons que la conception de composant est une chose aisée et que la syntaxe est celle du Pascal standard.

Création de l'unité

La définition d'une unité Maleva est identique à celle d'une unité Delphi, elle doit cependant au minimum faire référence à l'unité Kernel et si besoin à l'unité KernelBis. Les unités prédéfinies dans Maleva sont :

Kernel.pas et *KernelBis.pas*

Ces fichiers représentent pour le premier le Noyau de Maleva, définition des classes TSMAComponent et TSMAGestionnaireComm définition des structures TSMACallParam et TSMAConnexion, et pour le second un autre gestionnaire de Communication TSMAGestionnaireBis (à titre d'exemple).

KernelAgent.pas

Unité de définition de la classe TKernelAgent et de la classe TGeneralBehavior.

KernelControl.pas

Unité de définition des composants de contrôle. TSMAControlComponent et toute ses sous-classes.

KernelDetector.pas

Unité de définition de la classe TGenericDetector. Cette classe permet de définir de nouveaux capteurs.

KernelEffector.pas

Unité de définition de la classe TGeneralEffector. Cette classe permet de définir de nouveaux effecteurs

TEnviron.pas

Unité de définition de la classe TSMAEnvironnement. Tout environnement est sous-classe de celle-ci.

Création de la classe.

Celle-ci obéit à la même syntaxe que celle de Delphi. Les méthodes définissant les fonctions d'entrée et de sortie doivent être placées dans la partie published de la classe. Elles obéissent toutes à la syntaxe :

```
procedure Nom de la procédure(Param:PSMACallParam); virtual;
```

ou

```
procedure Nom de la procédure(Var Param:PSMACallParam);
virtual;
```

Exemple 5: Définition des bornes d'entrée/ sortie

Le paramètre Param est un pointeur sur la structure que nous avons décrite précédemment.

Tout ceci représente la partie Interface du composant, nous allons maintenant voir la partie Implémentation.

Identification et enregistrement des fonctions d'entrée, sorties.

Un composant est une classe, pour permettre de différentier les méthodes des fonctions d'entrée/sortie il faut les enregistrer auprès du gestionnaire de communication associé au composant. L'enregistrement se fait dans la méthode de construction de l'instance (méthode Create), par appel des méthodes RegisterInputMethsWithDescription pour définir les bornes d'entrées et RegisterOutputMethsWithDescription pour définir les bornes de sorties (Cf. Exemple 8). Ces deux méthodes sont définies au niveau du gestionnaire de communication, celui-ci étant référencé dans le composant par la variable Gestionnaire.

La syntaxe des méthodes est :

```
gestionnaire.registerInputMethsWithDescription(
[liste de chaînes de caractères correspondant aux noms
des méthodes],
[liste correspondant aux description des méthodes]);

gestionnaire.registerOutputMethsWithDescription(
[idem],
[idem]);
```

Exemple 6: Enregistrement des méthodes d'entrée/ sortie

Le premier paramètre est une liste des chaînes de caractères qui correspondent aux nom des méthodes que nous désirons définir comme fonction d'entrée (respectivement sortie), la deuxième liste correspond alors à la description sémantique des méthodes. Le gestionnaire de communication enregistrera ces listes, et les rendra accessible aux différents éditeurs.

Il existe les méthodes inverses, UnRegisterInputMeths et UnRegisterOutputMeths qui permettent de retirer des définitions qui ne sont plus nécessaires. Par exemple si l'on veut renommer une fonction d'entrée dans une classe alors que celle-ci a été enregistrée dans sa super-classe, la nouvelle classe peut « désenregistrer » l'ancienne méthode et enregistrer la nouvelle.

Identification et enregistrement du composant.

Afin que la plate-forme Maleva puisse connaître les composants qui ont été conçus il faut procéder à leur enregistrement. Un composant enregistré deviendra alors accessible aux utilisateurs dans la palette des composants.

L'enregistrement d'un composant se fait dans le corps de l'unité par l'appel des méthodes RegisterModuleWithDescription et RegisterClass dont la syntaxe est la suivante:

```
    RegisterModuleWithDescription(
       nom de la classe,
       chaîne représentant le nom de la classe,
       chaîne de description de la classe,
       chaîne représentant le groupe de la classe);
    RegisterClass(nom de la classe);
```

Exemple 7: Enregistrement du composant

- Le paramètre nom de la classe est celui défini par la classe.
- La paramètre chaîne nom de classe détermine le nom du composant qui apparaîtra dans l'éditeur.
- Le paramètre chaîne de description représente la description sémantique de la classe.
- Le paramètre groupe représente le groupe dans lequel le composant apparaîtra (les composants sont regroupés par thèmes dans la palette).

Par exemple le constructeur de la classe TGenericDetector est écrit comme suit :

```
    constructor TGenericDetector.Create(AOwner: TComponent);
    begin
       inherited create(AOwner);
       ...                       {Initialisation des
attributs}
       gestionnaire.UnRegisterInputMeths([’BeginControl’]);
    {méthode enregistrée dans la super-classe, renommée
Detect}
       gestionnaire.RegisterInputMethsWithDescription(
       ['Detect',
        'GetAgentDistance',
        'GetAgentDirection',
        'SetAgentOwner',
        ...],
       ['Détection des agents correspondant à FType',
        'Calcule la distance à un agent',
        'Calcule la direction vers un agent',
        'Mémorisation de l''agent possesseur',
```

```
        ...]);
end;
```

Exemple 8: Constructeur de la classe TGenericDetector

L'utilisation de la méthode UnRegisterInputMeths('BeginControl') permet de ne plus prendre en compte BeginControl comme borne d'entrée et de la remplacer par Detect qui est plus explicite pour un Capteur.

La classe est enfin enregistrée auprès de Maleva comme suit :

```
begin  {Corps de l'unité}
  registerModuleWithDescription(
    TGenericDetector, {nom de la classe}
    'CAPTEUR',        {nom du composant}
    'Capteur Générique', {descritpion du composant}
    'Capteurs');      {thème ou groupe fonctionnel}
  registerClass(TGenericDetector);
end.
```

Association d'une image au composant

Cela consiste à associer une image graphique (Bitmap 24x24) définie dans un fichier séparé (fichier de ressource graphique .res), l'image ainsi définie apparaîtra dans la palette des composants accessibles. L'association se fait automatiquement par Maleva qui met en concordance le nom de l'image dans le fichier et celui de la classe. Un des intérêt de cette approche et de permettre de modifier l'image sans recompilation du code, et donc d'offrir à l'utilisateur la possibilité d'utiliser des représentations auxquelles il est plus habitué.

IX.4.3 INTEGRER DES COMPOSANTS DELPHI

Bien qu'étant des composants « spéciaux », les composants Maleva peuvent « encapsuler » des composants de base de Delphi. C'est la cas par exemple pour les composants de communications (DDE, Sockets...). Les composants Maleva ne sont pas conçus de manière totalement identique aux composants Delphi, la structure de communication est unique tandis que celle des composants Delphi est définie lors de la conception du composant[39]. Les composants Delphi n'ont pas de gestionnaire de communication ni ne notion de connexions. Si nous voulons intégrer des composants Delphi[40] au système Maleva il faut créer un composant qui encapsulera le composant que nous voulons intégrer. Le principe

[39] Le concepteur détermine la structure des messages pour chaque méthode lors de la conception du composant.

[40] Ou des composant VBX, OCX écrits dans d'autres langages.

revient à écrire un nouveau composant Maleva qui reprend les principales méthodes proposées par un composant Delphi.

Par l'intermédiaire des composants de communication, nous pouvons alors définir de nouveaux composants correspondant aux applications que nous désirons utiliser. Si nous voulons utiliser un Tableur (Ex. Excel) pour traiter des données, nous pouvons alors définir un composant Excel que nous intégrerons dans notre application avec un composant DDE.

Un exemple de composant « encapsulateur » est proposé dans la plate-forme pour la génération de graphiques avec l'utilisation d'un composant VisualBasic (TChart). Delphi est un langage apparemment très en vogue ce qui explique que nous trouvions sur Internet un nombre grandissant de composants très utiles tels que des composants Excel (Un tableau aux fonctions identiques à une feuille de calcul Excel), Word (Un petit traitement de texte compatible Word), des composants permettant d'adopter des protocoles de communication ou de transfert de données (HTTP, FTP, TCP/IP...). On peut par exemple en quelques heures faire de Maleva une application fonctionnant sur Internet en réutilisant uniquement les composants fournis par Delphi[41].

M. Le Cerf a, par exemple, utilisé et augmenté Maleva dans le cadre de sa thèse (collaboration Laforia et INRETS) sur la simulation de carrefours autoroutiers. Dans cette version, il a été ajouté (entre autre) un composant Sockets permettant de se connecter sur un site serveur et de récupérer des données (% d'occupation au sol) en temps réel.

Intégrer des composants provenant d'autres langages

Une des particularités du langage Delphi est de permettre l'intégration de composants provenant d'autres langages (Visual C++, Visual Basic ...) pour peu que ces composants obéissent au standard des composants ActiveX. Maleva utilise d'ailleurs un de ces composants qui permet l'affichage en temps réel de graphes de données.

IX.4.4 LES OUTILS DE MALEVA

Comme nous l'avons vu, Delphi est un langage de conception très puissant, Maleva bénéficie bien sûr de ces atouts. Mais la plate-forme ne se résume pas uniquement à la conception de classe. Du point de vue de l'utilisateur qui désire concevoir ou simuler un SMA, il faut adjoindre à ces capacités des outils spécialisés.

La plate-forme propose un ensemble d'outils qui sont regroupés en trois pages Simulation, Conception et Résultats. On peut passer d'une page à une autre en sélectionnant la page où on désire aller grâce à des onglets.

[41] Avec la dernière version Delphi 3.0

La page Simulation représente la simulation proprement dite. C'est dans cette page que la simulation des agents a lieu. Elle regroupe les composants Scheduler, Environnement et les outils gestion de la simulation tels que Sauvegarde/Restitution...

La page Conception représente les outils de conception des agents par composition de composants existants. Elle regroupe les éditeurs permettant de créer les connexions entre les composants. Elle propose l'ensemble des composants Maleva ayant été auparavant enregistrés.

La page Résultats est un exemple d'intégration et d'utilisation d'un composant externe (TChart) permettant de visualiser en temps réel des courbes.

IX.4.4.1 Simulation

La page Simulation regroupe 4 grandes composantes, les outils de Sauvegarde/Restitution, les outils de Sélection, L'environnement de simulation et le Scheduler.

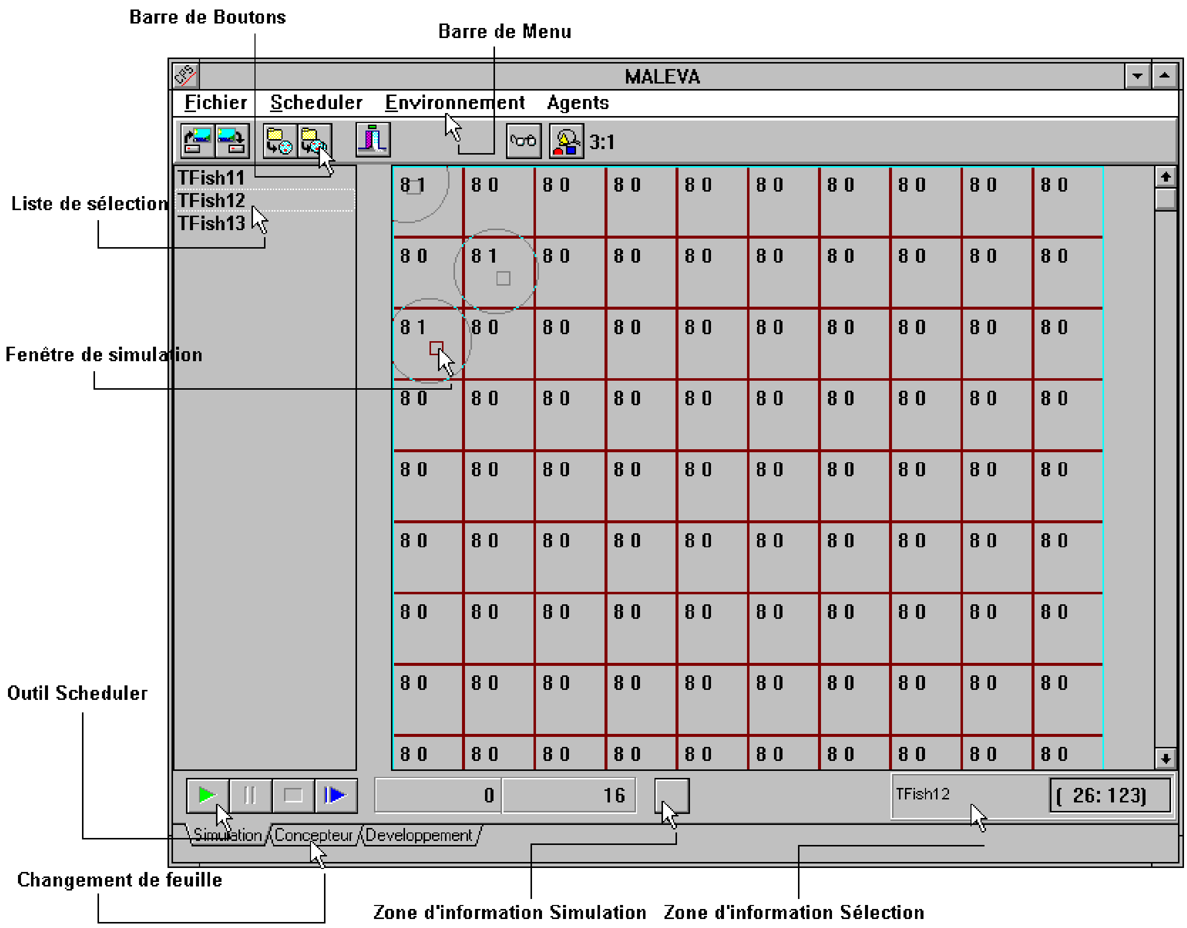
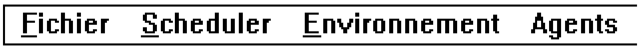


Figure 29: La Page Simulation, environnement discret

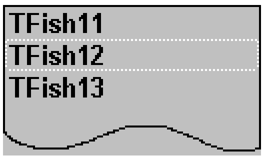
Sur cette figure, nous voyons l'ensemble de la page. Cette figure est la figure se trouvant dans le fichier d'aide fourni avec la plate-forme (appli.hlp).

Nous voyons en haut les outils de Sauvegarde/Simulation (Barre de menu et Barre de boutons)



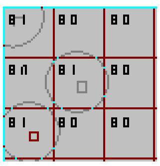
les outils de Sauvegarde/Restitution permettent de sauvegarder la simulation en cours (boutons de gauche). Sauvegarder la simulation revient à créer deux fichiers correspondant respectivement à l'environnement (.evt) et aux agents (.ags). Le bouton représentant une paire de lunettes permet de faire tourner la simulation sans affichage (plus rapide) et celui représentant une loupe avec à sa droite deux chiffres permet de faire un Zoom de l'environnement.

Au milieu à gauche l'ensemble des agents présents à ce moment dans l'environnement (Liste de sélection)



Cette liste permet de sélectionner un agent, l'agent sélectionné apparaît dans la zone Information Sélection.

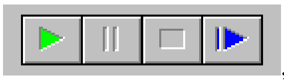
A droite l'environnement de simulation (Fenêtre de simulation)



dans cet exemple l'environnement est discret, chaque case affiche deux valeurs qui représentent le nombre d'agents sur la case et le nombre de cases voisines (le nombre de voisins dépend de la nature de l'environnement Toroïdal ou non, ou de la définition des accointances). Les agents sont ici représentés par des rectangles gris, les cercles représentent les distances de perception des agents.

L'environnement est un composant Maleva d'interface (visible), il peut être discret ou continu, il existe pour l'instant 4 composants Environnements fournis avec la plate-forme, Générique, Discret, Continu, Discret avec discrétisation incomplète (monde). Passer de l'un à l'autre revient alors à changer le composant, par exemple passer d'un environnement discret à un environnement continu revient alors à changer le composant TEnvironnementDiscret par le composant TEnvironnementContinu.

En bas nous avons le Scheduler

,

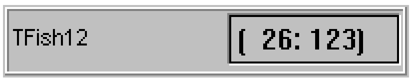
le Scheduler est le composant permettant de lancer la simulation en pas à pas ou en boucle. Il permet d'interrompre la simulation à tout moment et d'inspecter les agents ou l'environnement.

Il existe deux composants Scheduler, l'un fonctionne en mode 16 bits activé par la boucle principale de l'application, l'autre fonctionne en mode 32 bits comme un processus à part entière.

A la droite du Scheduler nous avons :

• le nombre de cycles d'activation et nombre de composants membres (Zone Information Simulation), qui représente le nombre total de composants dans la simulation.

• le type et la position de l'agent sélectionné (Zone Information Sélection)



• la zone de changement de page qui permet de passer d'une page à une autre. Ici nous avons la page Simulation visible et les deux pages (Concepteur et Developpement) accessibles.

A partir de cette page, nous pouvons modifier certains composants tels que le Scheduler ou l'Environnement. Le composant que nous sommes amenés à changer le plus couramment est le composant Environnement, en effet chaque type de simulation peut imposer des contraintes quand à la nature de celui-ci. Nous pourrions changer l'environnement pour qu'il ressemble à ceci.

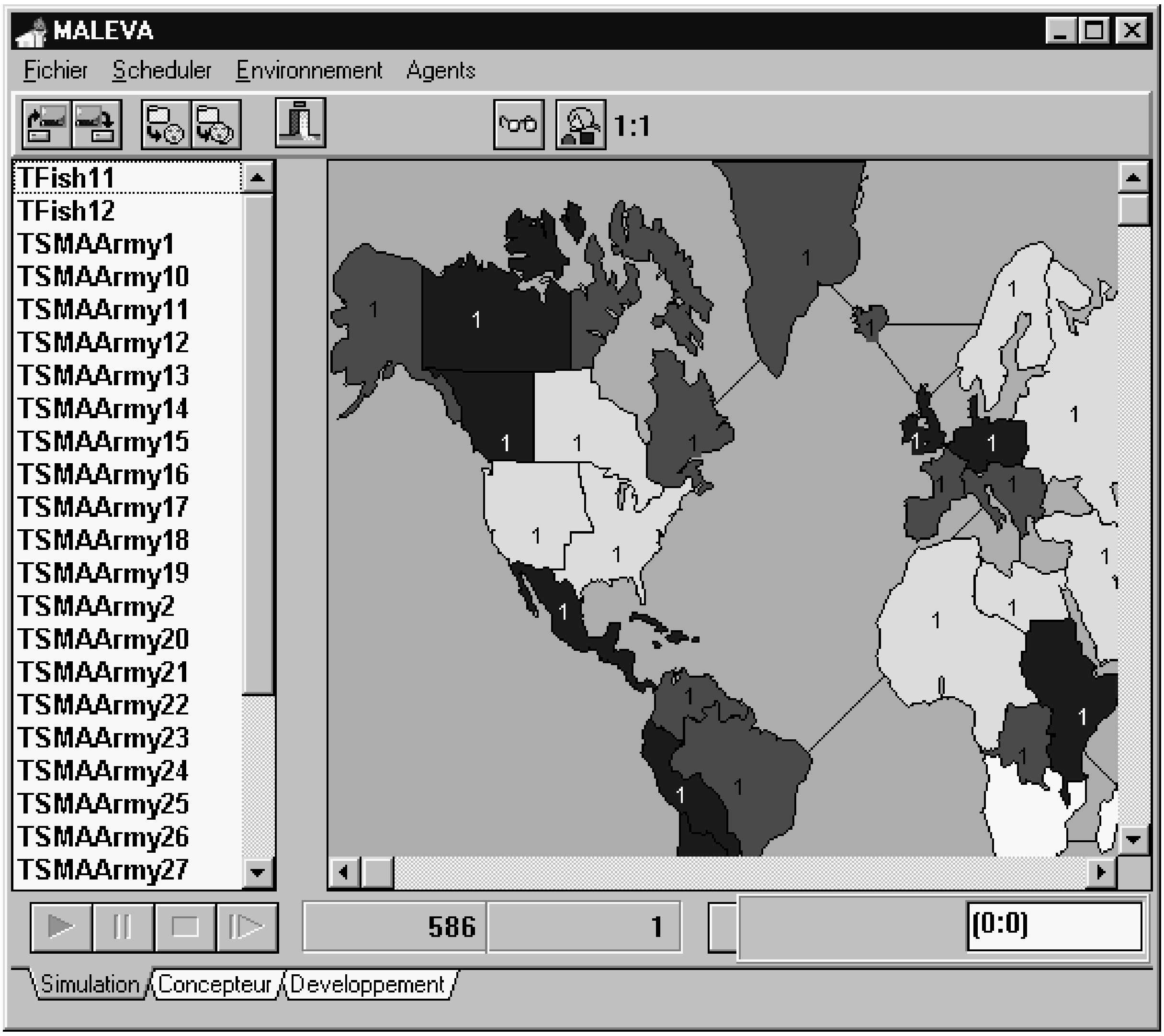


Figure 30: La page Simulation, un autre environnement

Dans l'exemple[42] ci-dessus, nous avons utilisé un environnement discret représentant le monde. Il s'agit d'un cas particulier d'environnement discret, les cases (elles représentent les pays) ne sont pas de taille identiques et elles ne forment pas une discrétisation complète de la surface. Les seules opérations que l'utilisateur est amené à faire consiste à remplacer l'environnement précédent par celui-ci en effectuant un simple glisser-déposer.

IX.4.4.2 CONCEPTION

La page Conception est la page où l'on peut concevoir les agents par regroupement de sous composants. Elle offre comme la page Simulation des outils de Sauvegarde/Restitution et génération de code qui correspondent cette

[42] Dans la version actuelle de Maleva, la fenêtre est légèrement différente car certains composants (toute la partie basse de la fenêtre) ont été réécrits, et n'ont plus la même apparence.

fois ci à un agent[43]. Comme la page Simulation, la page Conception est découpée en plusieurs zones.

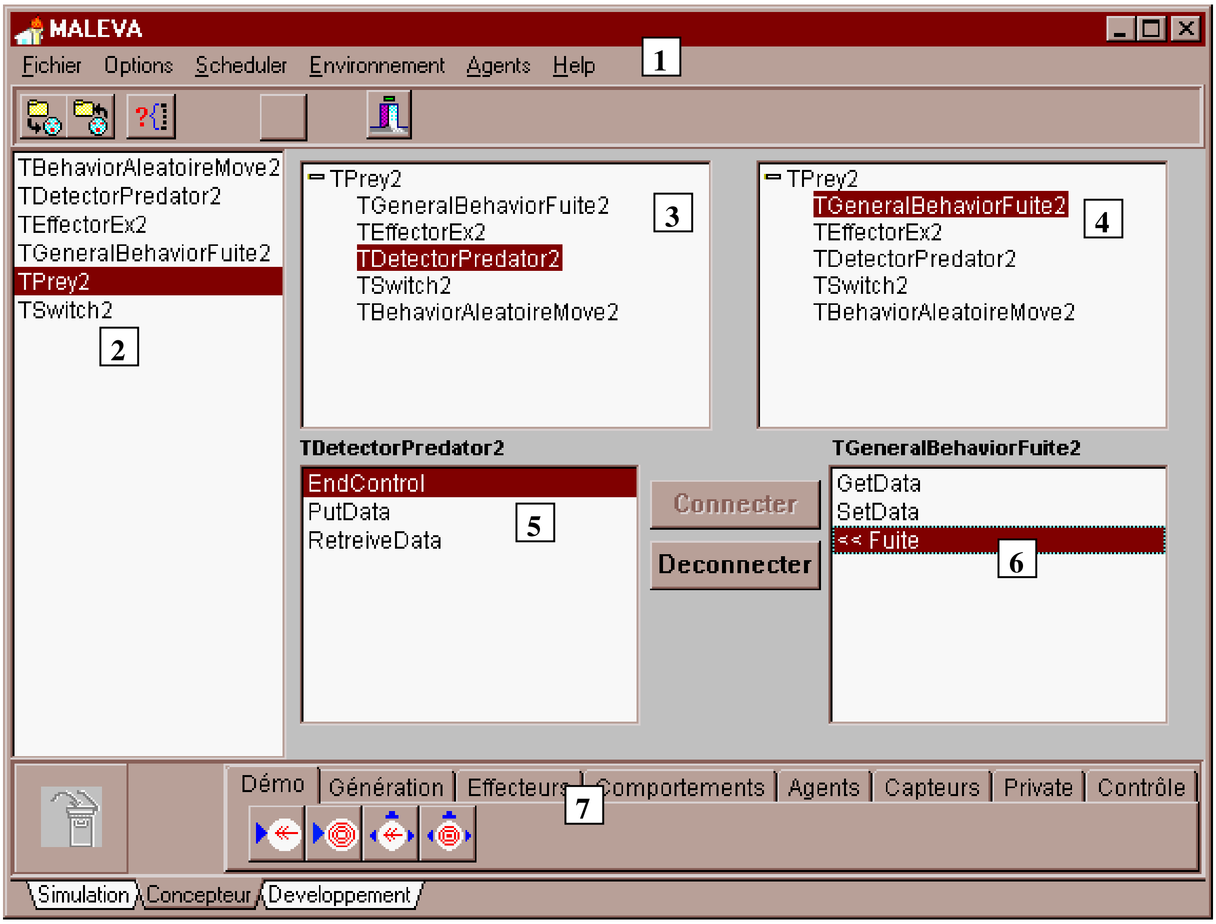


Figure 31: La page Conception

La partie supérieure de cette page (Barre de Menu et Barre de Bouton, **1**) correspond aux outils de Sauvegarde/Restitution mais cette fois ci ces outils s'appliquent aux composants en cours de conception.

La zone de Sélection (liste à gauche, **2**) permet de sélectionner un composant instancié (présent dans le système)..

Les deux Listes (haut centre, **3** et **4**), sont les listes de sélection du composant Client et du composant Serveur.

Les deux Listes (bas centre, **5** et **6**), sont les listes de sélection des fonctions de sortie (pour le client) et d'entrée (pour le serveur).

La poubelle (bas gauche) permet de détruire un composant ou une connexion.

[43] On crée ici un agent que l'on reprendra en un ou plusieurs exemplaires dans la page simulation.

La zone de boutons (en bas, **7**) représente la palette des composants (classes) accessibles.

Principe d'utilisation

I. On charge un ensemble de composants déjà sauvegardé (**1**) ou on sélectionne (instancie) des composants dans la zone de boutons (**7**).

II. Tous ces composants apparaissent dans la zone de sélection (**2**).

III. On double clique un composant dans la zone de sélection, il apparaît alors dans les zone Client (**3**) et Serveur (**4**) (sur l'exemple c'est le composant TPrey2).

IV. On déplace (Drag & Drop) d'autres composants de **2** vers **3**, ils deviennent alors sous composants et apparaissent alors dans les zones **3** et **4** (ici tous les autres composants sont sous composants de TPrey2).

V. On sélectionne un composant Client dans la zone **3** (TDetectorPredator2) et un composant Serveur dans la zone **4** (TGeneralBehaviorFuite2), pour chacun d'eux apparaîssent les fonctions d'entrée et de sortie dans les zone **5** et **6**.

VI. Il ne reste qu'à sélectionner certaines fonctions de sortie du client (ici EndControl) et à les connecter à certaines fonctions d'entrée du serveur (ici Fuite).

VII. On répète les étapes V et VI pour chaque sous-composant et chaque connexion.

VIII. On sauvegarde le nouveau composant (Menu) ou on génère une nouvelle classe de composant (Bouton).

Cette fenêtre permet donc de concevoir des agents (ou des composants) par composition de composants pré-existants. Nous sommes en train de développer un éditeur de graphe qui permette un affichage et un traitement entièrement graphique (comme le propose l'outil Editalk utilisé dans Maleva Smalltalk).

IX.4.4.3 RESULTATS

La page Résultats est la page où l'on peut visualiser les résultats des simulations sous différentes formes. Cette page est la moins aboutie car elle est plus dépendante de la simulation (de la nature des données à visualiser) que les autres pages.

Le principe de fonctionnement est simple, à chaque agent dont nous voulons reprendre des données à visualiser, nous associons un composant spécialisé (dans la figure ci-dessous il s'agit d'un composant TChartFxInterface) qui fait l'interface entre l'agent et le composant de visualisation utilisé (un agent ne sait pas forcément comment s'adresser à un composant de visualisation donné).

Dans l'exemple ci-dessous, nous utilisons un composant de visualisation TChartFx écrit en VisualBasic qui permet d'afficher des courbes sous différentes formes, il permet de sauver les données, de les voir sous forme textuelle, de les modifier ...

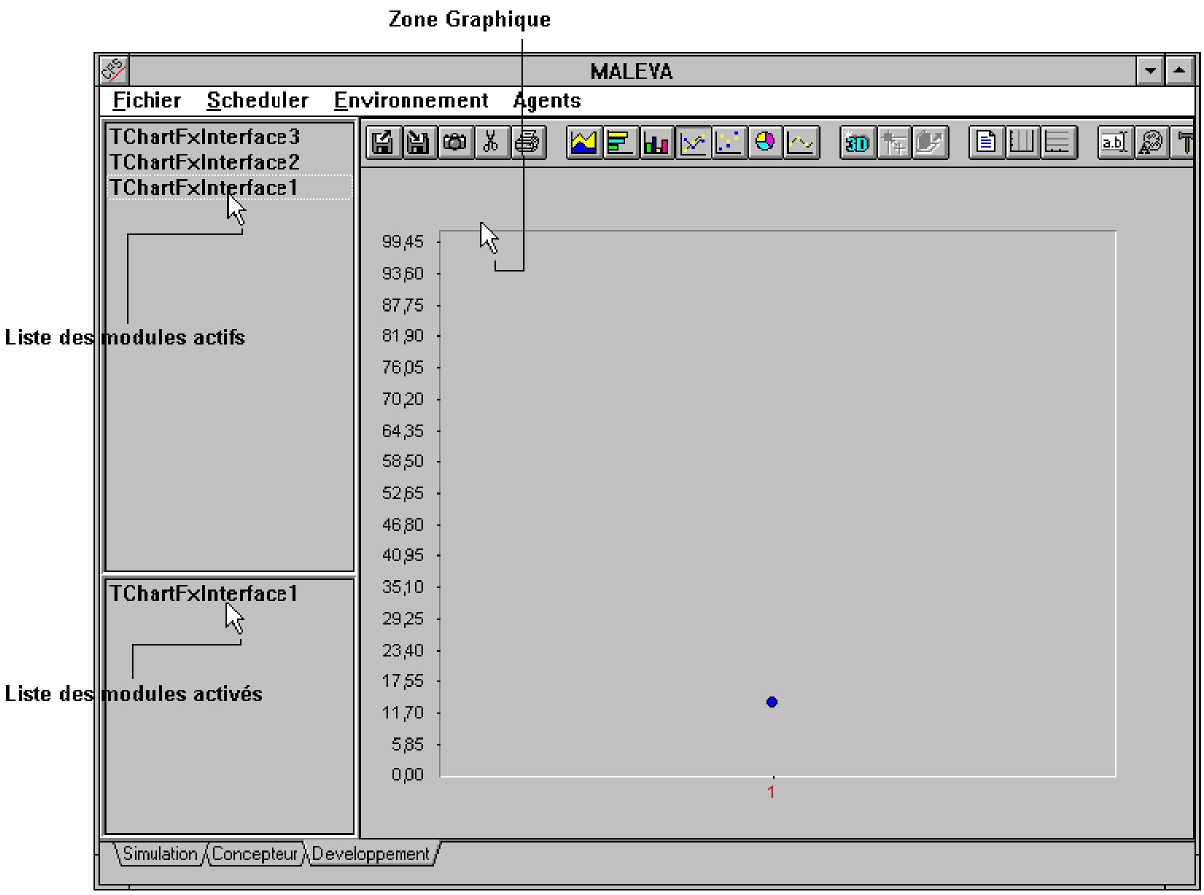


Figure 32: La Page Résultats

Le principe de cette page est d'offrir la possibilité de visualiser des données provenant d'un composant donné. Les composants de l'exemple (TChartFxInterface) sont associés à des agents auxquels ils fournissent une interface de communication avec l'objet de visualisation (TChartFX).

Afin de ne pas surcharger les courbes, nous dissocions les composants d'interface actifs (liste supérieur) de ceux activés (liste inférieure). Les composants actifs peuvent devenir activés par simple déplacement dans la liste inférieure. Un composant activé peut alors transmettre ses données à TChartFX lequel se charge de l'affichage des courbes.

IX.4.4.4 Autres Outils

Maleva propose des outils qui n'apparaissent pas directement dans la plate-forme, ces outils permettent entre autre d'inspecter des composants, de les visualiser, de les espionner ou de générer de nouvelles classes de composants. Nous ne verrons ici que quelques uns de ces outils.

IX.4.4.5 Inspecter les Composants

Maleva propose aussi un outil générique nommé Inspecteur de Composants. Il s'agit d'un outil identique à l'inspecteur d'objets fourni par Delphi que nous avons réécrit (celui fournit avec Delphi n'est pas utilisable en mode RunTime).

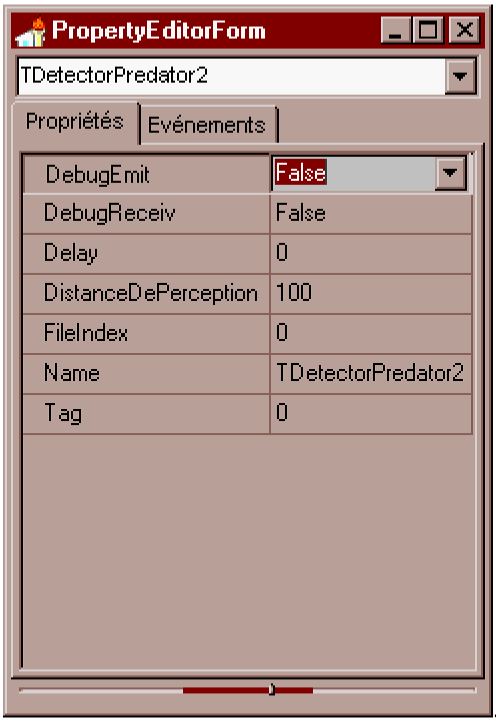


Figure 33: L'inspecteur de composants

Cet Inspecteur de composants propose deux volets (Propriétés et Evénements) qui permettent respectivement de « voir » la valeur des chaque propriété publiée d'un composant donné et de voir les événements définis pour ce composant.

Il offre la possibilité de changer ces propriétés en respectant le type de celle ci. Les valeurs discrètes (Booléens, Enumération) apparaîtront dans des listes, les valeurs composées (Ensembles) apparaîtront comme des sous propriétés avec pour chacune la possibilité de la sélectionner, les valeurs continues (Entiers, Réels) dans des boites d'édition.

Par exemple une propriété Booléenne fera apparaître une liste avec les valeurs True et False, une propriété Ensemble (Set) fera apparaître chacun des éléments avec la possibilité d'ajouter ou de retrancher des éléments de l'ensemble et une propriété Entier (Integer) fera apparaître une zone de sélection avec la valeur courante.

Une des possibilité les plus pratiques de l'inspecteur d'objet est de permettre de définir un éditeur spécifique pour un type de propriété donné, nous avons en effet repris l'ensemble des spécificités de l'inspecteur d'objets de Delphi.

Prenons par exemple une propriété de type TColor (le type TColor est défini comme une valeur entière), l'éditeur classique proposera une zone de sélection pour une valeur de type Integer (le code RVB de la couleur). L'utilisateur pourra modifier cette valeur avec la contrainte de connaître le passage de la couleur qu'il désire avec le code RVB correspondant. Afin de faciliter les modifications de ce genre de propriété, nous pouvons définir un éditeur spécifique (ici TColorPropertyEditor) qui offrira une liste avec toutes les couleurs définies dans le système ou si on le désire une fenêtre avec les couleurs.

Dans le volet Evénements, nous pouvons visualiser les événements d'un composant. Un composant Delphi (et par voie de conséquence, un composant Maleva) possède ce qu'on appelle des événements. Un événement représente la volonté pour un composant d'avertir qu'il s'est passé quelque chose. Par exemple pour un bouton, nous avons des événements prédéfinis comme le fait que le bouton ait été cliqué, appuyé, relaché... Un événement est représenté de façon interne par un composant comme une variable pointant sur une méthode dont le prototype a été défini. Traiter un événement revient alors, pour celui qui veut la gérer, à créer une méthode correspondant au prototype et de prévenir le composant (le bouton) que l'on désire que cette méthode soit appelée si il y a un événement.

Pour les composants Maleva, nous avons défini certains événements tels que OnBeforeAct, OnAct, OnAfterAct (événements déclenchés avant, pendant et après qu'un composant soit activé), OnDebugEmit et OnDebugReceive (événements déclenchés lorsqu'un composant reçoit (resp. transmet) un message avec un autre composant), et enfin OnConnexionChange qui est invoqué lorsqu'un composant voit ses connexions changer (ajout, retrait..).

Afin de montrer en temps réel les valeurs des propriétés des composants, l'inspecteur d'objet peut être couplé avec un autre composant ComposantRefresh qui est un composant servant à faire rafraîchir certaines vues. L'opération de rafraîchissement revient alors à traiter l'événement OnAct de ce composant et à demander aux vues de se réafficher. Nous pouvons donc dire que la plate-forme Maleva est écrite avec des composants Maleva.

Le code correspondant correspond à ceci :

```
Var UpdateComposant :TSMAComposant ;

procedure TForm1.Initialise;
Begin
   {On suppose que le composant est instancié ici}
   ...

   UpdateComposant.OnAct:=OnComposantRefreshAct;
end;
```

Dans cette partie, nous (TForm1) demandons au composant de nous avertir s'il est activé, et nous lui disons que la méthode à appeler est OnComposantRefreshAct (laquelle doit avoir une structure correspondant au prototype défini par OnAct, rassurons-nous, ceci est fait automatiquement par le générateur de Code).

```

procedure TForm1.OnComposantRefreshAct(Sender:TObject);
var heap:THeapStatus;
begin
   ...
   systeminfolabel1.refresh;
   heap:=getHeapStatus;
   selection.caption:=format('%d',[heap.totalfree]);
end;
```

Dans cette méthode nous écrivons le traitement à effectuer lorsque l'événement est déclenché.

Le volet Evénement de l'Inspecteur d'objets permet de visualiser l'ensemble des événements définis au niveau d'un composant et de voir les méthodes de gestion associées sans pouvoir modifier ces méthodes (pour des raisons techniques ceci n'est pas encore possible).

IX.4.4.6 Visualiser les Composants

L'outil Visualisation de Composant permet de visualiser un composant présent dans le système. Son rôle est essentiellement informatif, il n'autorise aucune modification des liens existant entre les composants.

Pour une classe de composants donné, et si ce composant possède des instances, il permet de sélectionner une de ces instance et de regarder l'ensemble de ses fonctions d'entrée et de sortie, il permet d'examiner chacune des connexions afin de savoir avec qui ce composant est en relation.

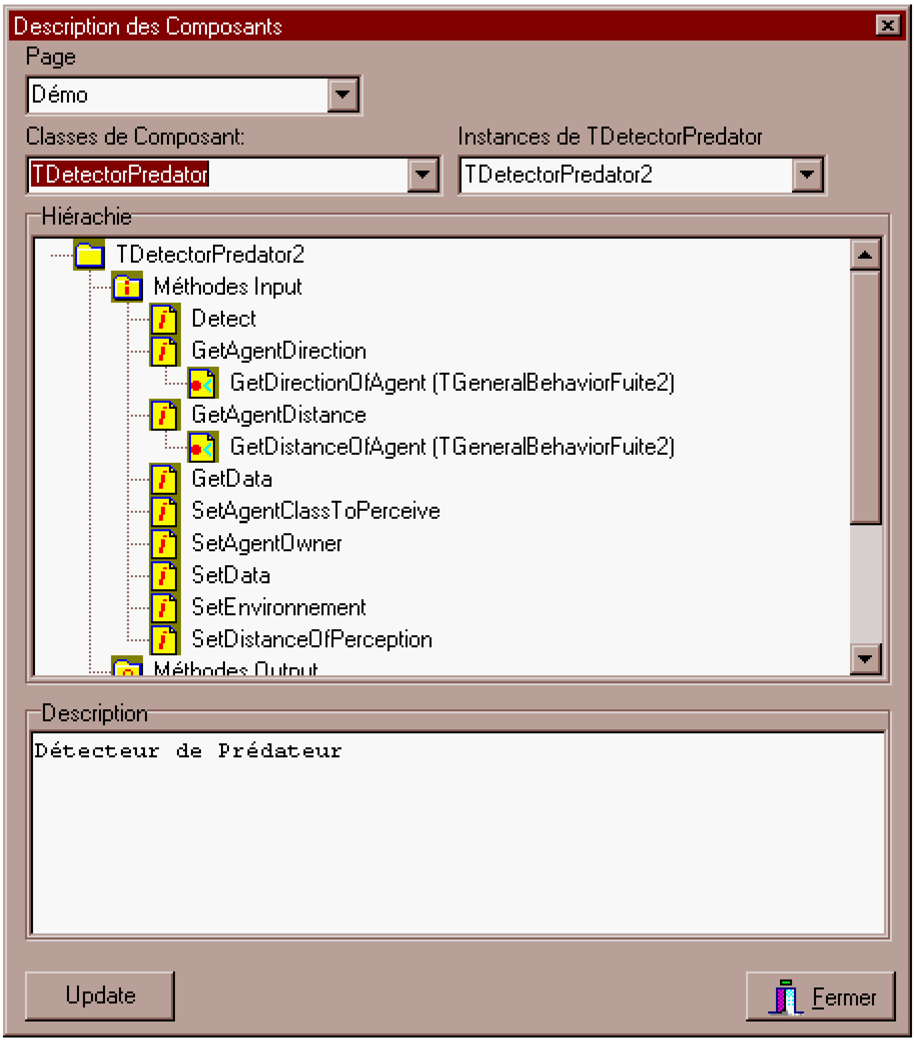


Figure 34: L'outil visualisation de Composants

Enfin cet outil permet de voir la description sémantique associé à chaque classe de composant, à chaque instance et à chacune des bornes d'entrée/sortie.

Dans l'exemple ci-dessus, nous voyons la description d'une instance (TDetectorPredator2) de la classe TDetectorPredator définie dans le groupe de composant nommé Démo (Cf. la page de Conception, Figure 31, volet **7**).

A partir de cette fenêtre nous pouvons donc visualiser tout composant présent dans le système. Nous pouvons alors voir l'ensemble de ses méthodes Input et Output et savoir avec qui elles sont connectées. Sur l'exemple la Méthode Input GetDirectionOfAgent est connectée une seule fois (une seule ligne) avec le composant TGeneralBehaviorFuite2, lequel deviendra accesible en cliquant sur la ligne correspondante.

IX.4.4.7 Espionner les Composants

Il s'agit d'un outil qui permet de visualiser les communications d'un composant donné, on peut alors voir les connexions en entrée et en sortie qui sont activées. C'est un outil assez rudimentaire conçu autour d'un composant qui en fait va remplacer le composant à espionner en détournant l'ensemble des connexions à son profit. Il ne lui restera plus qu'à afficher les messages et à les renvoyer à leur destinataire.

IX.4.4.8 Creer des Composants

Créer des composants est une opération nécessaire et pour certains d'entre eux (les composants composites) cela devient fastidieux. Pour faciliter la tâche du développeur, Maleva propose un outil de génération de code de composant nommé ComponentGenerator. Il s'agit d'un outil permettant à partir d'un ensemble de composants sélectionnés dans la page Conception de créer un nouveau composant qui 'agrège' ceux-ci. Cet outil crée en fait le code d'une nouvelle classe qui possédera comme champs chacun des sous-composants sélectionnés.

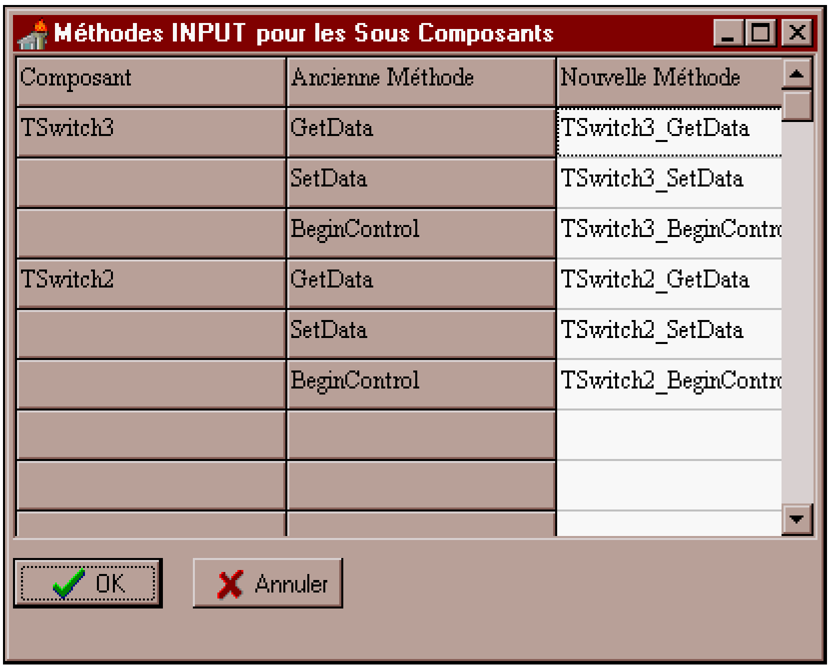


Figure 35: Outil de définition des bornes d'un composant Composite

Comme nous l'avons vu agréger des composants revient à créer un nouveau composant qui reprenne l'ensemble des fonctions d'entrée et de sortie des sous-composants, avec bien entendu la possibilité de renommer les fonctions dupliquées. Le générateur de composant permet d'automatiser tout ceci. Son rôle est de reprendre l'ensemble des fonctions, de proposer un nouveau nom pour chacune de celle ci, et bien entendu de générer une nouvelle classe de composant qui pourra par la suite être compilée et intégrée à la plate-forme.

Dans l'exemple de la Figure 35, nous avons voulu prendre un exemple d'agrégation de deux composants identiques, deux composants Tswitch, dans un composant parent, la fenêtre récapitule alors les bornes dupliquées (ici les bornes d'entrée) et propose de les renommer. Il fera de même avec les bornes de sorties et dès qu'il n'existera plus de problèmes, il génèrera le code correspondant.

X REALISATIONS ET APPLICATIONS

Ce chapitre est consacré aux différentes versions de Maleva qui ont été réalisées, nous décrirons principalement les deux versions majeures. Nous aborderons l'architecture générale de la plate-forme pour ensuite nous attacher à décrire plus finement la dernière version. Nous décrirons les différentes réalisations dans une approche chronologique.

La plate-forme Maleva (Modular Architecture for Living and EVolving Agents) a été écrite dans plusieurs langages. Chacune de ces versions a permis de bénéficier des particularités respectives de chaque langage, de plus certaines modifications ont été apportées au fur et à mesure. Deux versions ont été écrites par moi-même et deux versions du noyau ont été écrites en collaboration avec des étudiants lors de projets de stage de DEA ou de DESS.

Nous avons réalisé l'étude et la conception de Maleva en nous définissant dans deux objectifs principaux qui n'ont pas variés au cours du temps. Le premier objectif est d'offrir un ensemble d'outils de conception et de simulation souples et suffisamment puissants pour être appliqués à des simulations longues et complexes. Nous reviendrons sur le premier but qui nous a démontré la nécessité de réécrire la plate-forme dans un langage plus efficace. Le deuxième objectif est de permettre aux possesseurs[44] de la plate-forme de modifier celle-ci aussi bien dans son apparence (son interface utilisateur) que dans son fonctionnement propre (le noyau ou plus certainement les classes de base). Ce deuxième objectif a nécessité une approche de la conception centrée sur la réutilisation des composantes de la plate-forme. Pour ce faire, et comme nous le verrons plus loin, Maleva a été conçue autour d'un noyau (kernel), ce noyau est totalement dissocié de l'interface de conception (conception des composants et conception des agents) aussi bien que de l'interface de simulation.

X.1 VERSION SMALLTALK

X.1.1 REALISATION

Maleva a été écrite en Smalltalk dans sa première version. Le choix du langage a été simplifié par les besoins lors de la mise en place du projet. En effet Smalltalk est portable entre différents systèmes (Unix, Mac, PC) sans nécessiter une recompilation du code. Smalltalk offre des outils d'aide à la

[44] Il s'agit des personnes qui utilisent Maleva pour concevoir une simulation. Il ne s'agit donc pas des utilisateurs finaux.

conception d'Interface Homme Machine (IHM) qui permettent un développement rapide d'interfaces utilisateur. Smalltalk offre le couple classe/méta-classe et la réflexivité du langage permet de travailler à n'importe quel niveau du langage.

L'essentiel des plates-formes existantes au sein de l'équipe Miriad étant écrites dans ce langage, le principe de réutilisation des composantes déjà existantes en a été d'autant plus facilité.

Le noyau de Maleva y a été conçu autour du frameword Actalk tandis que les outils de conception visuelle ont été conçus autour du framework Editalk. Nous ne reviendrons pas sur Actalk dont nous avons déjà parlé. Pour la partie Interface de conception, nous avons intégré Editalk, un éditeur graphique générique. Editalk nous a permis de construire un outil de conception visuel pour nos composants, cet outil permet de visualiser les composants ainsi que les connexions qui leurs sont associées. Il faut noter que cette version de Maleva ne proposait pas les composants partagés, c'est à dire les composants pouvant posséder plusieurs connexions sur une même borne.

Smalltalk étant un langage relativement répandu, il nous a permis d'utiliser des outils de conception tels que Actalk et Editalk qui ont grandement facilité le travail de conception.

Enfin pour concevoir les outils de simulation et particulièrement les outils de gestion, nous avons de plus utilisé des classes telles que le BOSS (Binary Object Storage System) et CPOK (Parser C) qui permettent respectivement de sauvegarder des objets sans avoir à gérer les références (circulaires ou non) entre eux et de développer et intégrer des bibliothèques de fonctions dynamiques (Dynamic Link Library DLL sous Windows) ou des bibliothèques de fonctions statiques sous Unix et Mac.

X.1.2 APPLICATION

Munie de ces outils puissants, la plate-forme Maleva a été conçue rapidement ce qui a permis de l'utiliser dans une première application. La plate-forme Selva a été conçue par Dario Caruso [Caruso94], dans le cadre d'un projet de simulation de l'évolution d'une forêt équatoriale. Ce projet a été mené en collaboration avec le CIRAD-Forêt et encadré par Mme Sylvie Gourlet Fleury [Gourlet97]. Maleva a été fournie et modifiée par M. Caruso afin de mieux s'apparenter aux volontés des utilisateur finaux.

La complexité de la simulation (il a été nécessaire de modéliser différentes espèces d'arbres ainsi que le sol, le climat et d'autres paramètres) qui impose une vue locale (centrée sur l'arbre) et globale (centrée sur la forêt) a montré les limites de Maleva.

Dans une approche componentielle, un agent est au minimum composé de 5 voir 6 composants de base : Capteur, Effecteur, Comportement, Interface et Agent; le sixième composant optionnel correspond au composant de communication inter-agents et n'est utile que pour les agents communiquants (ce n'est pas le cas pour les arbres). Si nous multiplions le nombre de composants d'un agent par le nombre d'agents (dans Selva la simulation se fait sur des parcelles de plusieurs hectares ce qui représente quelques milliers d'arbres) et que nous rajoutons les connexions nous arrivons très vite à saturation.

Dans la simulation proprement dite, le cycle d'activation[45] est censé représenter une durée qui, dans la réalité, représente l'ensemble des comportements de l'agent simulé pour une période donnée. Dans le cas d'une espèce d'arbre, il a été décidé dans un premier temps de définir le cycle d'activation comme correspondant à une année réelle de la vie d'un arbre. Le but étant de descendre à un cycle de six mois pour finir à trois mois (tous les 3 mois interviennent des paramètres tels que la température, le taux d'humidité qui influeront sur la vie de l'arbre).

La plate-forme Maleva ne permettait pas de simuler, en un temps raisonnable, autant d'agents avec un pas de temps correspondant à trois mois et ce pour une simulation représentant 5 années de la vie de la forêt.

Afin de la modifier, nous avons alors entrepris d'offrir la possibilité aux composants d'être partagés. Le but était, dans le cadre de Selva, de partager un seul composant comportemental entre tous les arbres d'une même espèce. Ce travail a été entrepris par M. Caruso tandis que dans le même temps nous analysions les qualités et défauts de cette version.

Maleva version Smalltalk est un outil puissant dans la rapidité de modélisation et de prototypage. Malheureusement, elle possède de nombreux défauts dont le principal est la faible vitesse d'exécution. Parmi les autres défauts on peut citer les manques du langage[46] en ce qui concerne l'Interface Homme Machine (IHM), le peu d'ouverture vers d'autres langages et applications.

X.2 VERSIONS PTITLOO ET MERING IV

Comme nous l'avons vu, la version Smalltalk de Maleva péchait par un certain nombre de défauts. Nous avons donc entrepris d'étudier l'approche Maleva à partir d'autres langages tels que PtitLOO et Mering. Ces deux versions n'ont concerné que le noyau compte tenu du temps accordé à leur réalisation.

---

[45] Nous rappelons que le cycle d'activation correspond à l'activation d'un agent au cours d'un cycle.

[46] Il s'agit de Smalltalk 80 2.5

Maleva-PtitLOO, il s'agit d'une version Lisp/C du noyau de Maleva qui permet de bénéficier de la puissance et de la portabilité du langage. PtitLOO est un langage Lisp Orienté Objet développé par J. Ferber, il permet de développer rapidement des applications mêlant du code Lisp et du code C. Ce langage propose une IHM dynamique et puissante, mais hélas la rapidité du langage en partie interprété n'est toujours pas suffisante. La version PtitLOO a été écrite par deux étudiantes lors d'un projet de DESS GLA, la durée du projet n'a pas permis d'aller plus loin dans le développement de la plate-forme et d'offrir un outil suffisamment complet pour être utilisable tel quel. Cependant, ce projet, a permis de montrer toute la puissance d'une application « ouverte » à d'autres outils ou à d'autres langages (le C par exemple), l'aspect intégration d'outils existants est un atout primordial qui ne doit pas se résumer à l'intégration de composants supplémentaires conçus dans le langage d'origine et dédiés à la plate-forme.

L'autre version du noyau de Maleva a été écrite en Mering IV qui est un langage d'acteur réflexif [Carle92][Ferber91]. Cette version a été conçue lors d'un projet de DEA, l'utilisation du langage Mering a permis de bénéficier de la puissance de la réflexivité du langage et de son parallélisme fort. De plus Mering offrant tous les protocoles de communications dont nous avions besoin (Point à Point, Multicast, messages Synchrone et Asynchrone bufferisé), le conception du noyau en a été grandement simplifiée. L'idée principale de ce projet était de modéliser la notion de composant logiciel et d'étudier plus finement d'une part le parallélisme et la distribution des composants et d'autre part les mécanismes de communication avec continuation implicites. Comme dans le projet précédent, la durée n'a pas permis d'aller plus loin et d'offrir une application complète.

X.3 VERSION DELPHI

L'ensemble des études et réalisations précédentes ont permis de prendre en compte plusieurs critères discriminants quant au choix du langage de réalisation et des outils que la plate-forme doit fournir. Le modèle de composant a aussi été affiné.

Les critères du choix du langage peuvent être énumérés comme suit :

- Un langage rapide, de préférence un langage compilé. La portabilité semble être un défaut si elle ne s'acquiert qu'aux dépends de la rapidité d'exécution.
- Un langage facile à comprendre, afin que les futurs utilisateurs puissent mettre la main à la pâte.
- Un langage permettant de concevoir rapidement et facilement une IHM puissante et souple.
- Un langage ouvert vers les autres applications.

Le modèle de composant doit proposer :

- Les composants atomiques et les composants composites.

- Les composants partagés, afin de minimiser la complexité.
- Les gestionnaires de communications totalement dissociés des comportements des composants, afin d'offrir plusieurs modèles de gestion des communications.
- Les connexions mono-directionnelles avec contrôle dissocié des données, afin d'offrir des possibilités supplémentaires de composition fonctionnelle et de répondre aux problèmes de modifications dynamiques des relations entre composants.

Enfin les outils de conception/simulation doivent offrir :

- Des outils de visualisation des composants simples, composites.
- Des outils pour la composition fonctionnelle, afin de faciliter les connexions.
- Des outils de conception structurelle, afin de faciliter la création de nouveaux composants composites.
- Des outils de simulation, un Scheduler...
- Des outils d'inspection, notamment un éditeur de propriétés pour les composants. Smalltalk en propose un très pratique (voire trop). Parmi les autres outils, il faut au moins un outil de visualisation de courbes ou de traitement de données (type tableur).

La dernière version de Maleva a été écrite en Delphi. Le choix de Delphi a été fait pour plusieurs raisons dont la plus déterminante fut que le langage lui-même est un langage de conception à base de composants logiciels même si ceux-ci ne correspondent pas tout à fait à la notion de composants logiciels tels que décrits dans cette thèse.

Delphi est un excellent langage de programmation qui malheureusement à le seul défaut de ne pas être portable (pour l'instant) sur différents systèmes. Cependant tant par les outils de conception de l'IHM que par la rapidité du code compilé, je pense que cette dernière version reste la meilleure et la plus efficace.

Nous ne traitons pas plus ici de la version Delphi que nous étudierons plus finement. Cependant, et afin de rester dans le plan de lecture, nous allons parler de l'application qui a été réalisée à partir de cette version.

X.3.1 APPLICATION

En collaboration avec l'INRETS[47], M. Vincent Le Cerf [Le Cerf97] a entrepris d'utiliser Maleva pour concevoir un outil d'étude et d'analyse de carrefour autoroutier. Le but de cet outil est de fournir des informations pertinentes à partir de données numériques reçues en temps réel. La chaussée est décomposée en un ensemble de surfaces rectangulaires adjacentes. Les données proviennent de caméras et sont traitées en amont de l'outil, elles sont traduites en des valeurs de taux d'occupation (il s'agit d'un pourcentage de l'occupation) de ces surfaces.

[47] Intitut National de Recherche sur les Transports et leur Sécurité.

Le but de la plate-forme est d'une part de modéliser l'environnement, c'est-à-dire les surfaces et leur taux d'occupation; d'autre part de modéliser des compositions de surface afin de faire correspondre ces surfaces à celles qu'occupent des véhicules (voitures, camions...) et enfin d'en extraire les informations viables telles que déterminer la nature des véhicules, leur orientation et leur vitesse.

La difficulté de l'étude est d'arriver à déterminer à partir d'informations imparfaites et « parcellaires » (les données d'une seule surface ne permettent pas d'obtenir des informations viables) des informations exactes. Une difficulté supplémentaire est liée à la nature mobile ou immobile des informations, en effet un véhicule peut être mobile, auquel cas les taux d'occupation vont se déplacer, ou un véhicule peut être immobile, auquel cas les taux d'occupation vont stagner voire augmenter (plusieurs véhicules appartiennent à la même surface).

M. Le Cerf a donc utilisé Maleva pour deux raisons principales, la première étant sa disponibilité et la seconde étant liée à son approche Multi-Agents. L'aspect Multi-Agents était attrayant pour la modélisation de l'espace et des véhicules. L'approche componentielle de Maleva a été un plus dans la mesure où l'on peut facilement identifier un composant à une surface donnée et par conséquent mettre en correspondance la composition componentielle et la composition des surfaces.

Comme pour Selva, M. Le Cerf a modifié Maleva. Il a redéfini l'interface pour la faire correspondre aux besoins et a ajouté de nouveaux composants (des composants d'accès réseaux de type TCP-IP et autres) pour adapter la plate-forme aux contraintes temps réel de l'application ainsi qu'à l'utilisation de données réparties.

A l'heure actuelle, Selva ne correspond plus du tout à ce qu'était Maleva à l'origine, cependant, et c'était un des buts initiaux, les modifications apportées à l'original ont été faites en un minimum de temps.

XI CONCLUSION FINALE

L'objectif de cette thèse a été d'offrir un modèle de conception simple et des outils de réalisation pour les agents et les systèmes multi-agents. Nous avons alors fait plusieurs choix. Le principal choix a été de définir un modèle d'agent à base de composants logiciels. Pour ce faire il a été nécessaire de prendre en compte les différentes approches componentielles et de les appliquer aux SMA. Nous avons pour cela définit un modèle de composant différent des modèles classiques.

L'apport principal de ce modèle à base de composants est de favoriser la réutilisabilité des composants, applications et outils déjà existants. L'approche proposée permet de tirer parti des modèles plus anciens, modèles objets, acteurs et modulaires, pour concevoir des agents et des systèmes multi-agents plus flexibles et plus dynamiques.

La distinction qui est faite entre les flots de données et les flots de contrôle a permis de prendre en compte certaines contraintes qui sont dues aux différentes formes de composition. Mais cette distinction a aussi facilité la prise en compte des contraintes qui sont liées à la simulation, et notamment pour tout ce qui concerne les notions de cycles de simulation et la gestion de la synchronisation.

Afin d'intégrer ce modèle de conception dans une plate-forme, il est apparu nécessaire de fournir des outils de conception, de manipulation et de gestion. Maleva est le fruit de plusieurs travaux d'étude et de conception. Les différentes versions qui ont été réalisées ont permis de prendre en compte différents points de vue, et les applications qui ont été réalisées ont montré les qualités mais aussi les défauts et les manques de la plate-forme.

Parmi les autres choix qui ont été fait, on peut citer les différents types d'environnements, l'utilisation d'un Scheduler pour le séquencement des activations. Même si, forcément, ces choix contraignent l'espace des applications possibles de Maleva, ils ont été étudiés et conçus dans une optique de simplification de la réutilisation.

Parmi les travaux futurs, nous entreprenons de compléter le modèle par une plus grande rigueur quand à la composition fonctionnelle. Nous désirons, entre autre, permettre de faire des vérifications de types ainsi que des opérations de manipulation sur ceux-ci. Nous pouvons d'ailleurs citer les travaux de L. Bellisard [Bellisard97] qui, dans une approche componentielle, propose de tels outils.

L'identification d'invariants dans différents modèles d'agents nous permettra de déterminer des patrons de conception. Les patrons décriront des schémas de composition identiques entre plusieurs agents. Un patron correspondra alors à la composition fonctionnelle de plusieurs composants que nous déterminerons après une phase d'étude et d'implémentation de différents modèles d'agents dans Maleva. Cela fera l'objet d'un stage de DEA. Un patron, une fois déterminé, sera intégré dans un composant composite (composition structurelle) qui pourra alors être réutilisé dans d'autres agents.

La définition de patrons de conception devra contraindre le type des sous-composants et la nature des connexions entre ceux-ci. Leur application, à travers des composants composites devra intégrer ces contraintes de types et devra être traité à un méta-niveau. Nous définirons alors de nouveaux composants, des méta-composants, qui devront être capables de gérer les composants et les connexions au sein de l'agent. Ils devront pour cela distinguer et identifier les composants et les connexions. La seule définition des classes ne permet pas de distinguer les composants et leur rôle, nous utiliserons alors les définitions sémantiques des composants et nous utiliserons le typage des connexions afin de permettre une gestion de ceux-ci au méta-niveau.

Les composants que nous avons conçus sont pour la plupart des composants de bas niveau. Ils sont essentiellement destinés à la conception et à la réalisation d'un agent simple situé. Une des évolutions de Maleva va consister à définir des composants de plus haut niveau. Pour cela nous intégrerons des composants de granularité plus forte, par exemple nous intégrerons des composants pour la communication entre agents (composants de négociations [Yoo97]).

Nous souhaitons séparer le comportement du gestionnaire de communication qui lui est associé. Le but est de permettre une modification dynamique du gestionnaire en cours d'exécution du composant. En effet, et afin de rendre les composants totalement indépendants à la fois du langage d'origine et du système d'exploitation, nous désirons intégrer les mécanismes d'interaction (les envois de messages ainsi que les protocoles) dans des composants à part entière. Pour cela nous devrons concevoir des composant de bas niveau qui puissent définir différents protocoles de communication entre composants. Le gestionnaire de communication, bien qu'associé au comportement par des canaux de communications qui ne peuvent pas être des connexions, pourra alors être conçu comme une composition des sous-composants intégrant les différents protocoles.

Enfin, nous désirons intégrer plus finement l'aspect réparti de Maleva. Pour ce faire nous intégrerons l'architecture COM (Common Object Modelling) [COM, DCOM] et plus particulièrement son approche distribuée DCOM. A l'instar de CORBA, COM permet la répartition d'objets sur des machines distantes voir sur des systèmes d'exploitations différents.

REFERENCES BIBLIOGRAPHIQUES

[Agha 86] G. Agha « Actor : a model of concurrent computation in distributed systems», MIT press, 1986.

[Bellisard 97] L. Bellisard, M.C. Pellegrini, M. Riveill « Intregration and Distribution of Legaccy Sofware with OLAN », in 2nd France-Japan Workshop Object Based Parallel and Distributed Computing (OBPDC'97), Octobre 1997, IRIT.

[Blumberg & Galyean 95] B. M Blumberg, T. A. Galyean « Multi-Level Direction of Autonomous Creatures for Real-Time Virtual Environments», 1995.

[Bouron 91] T. Bouron « COMMAS : A Communication Model for Multi-Agents System », in Modelling and Simulation : Proceedings of ESM'91, Simulation Councils, Copenhague, 1991.

[Bouron 92] T. Bouron « Structure de Communication et d'Organisation pour la Coopération dans un Univers Multi-Agents », Thèse de Doctorat, Université de Paris VI, 1992.

[Brazier 96] F.M.T Brazier, P.A.T. van Eck and J. Treur « Modelling Cooperative Behaviour for Ressource Access in a Compositional Multi-Agent Framework », in J.L. Fiadero and P.-Y. Schobbens (Eds), Proc. 2nd Workshop of the ModelAge Project. Sesimbra Portugal, Jan. 15-17, 1996, pp. 27..40.

[Brethon 97] J.M. Brethon, D. Polzin « Maxtalk », projet DESS G.L.A. 1997.

[Briot89] J. P. Briot « Actalk : a Testbed for Classifying and Designing Actor Laguages in the Smalltalk-80 Environment. », in Proc of ECOOP'89 Nottingham, UK, pp 109..129.

[Brooks 86] R.A. Brooks « A robust layered control system for a mobil robot », IEEE Journal of Robotic and Automation, 2(1), 1986, pp. 14..23.

[Brooks 90] R.A. Brooks « Elephant Don't Play », Journal of Robotic and Autonomous Systems, Vol 6, N°1,2, 1990, pp. 3..15.

[Bura 93] S. Bura, F. Guérin-Pace, H. Matian, D. Pumain, L. Sanders « Multi-Agents Systems and the Dynamics of a Settlement System. », in Simulating Societies Symposium, Siena, C. Castelfranchi (Ed.), 1993.

[Bura & al 94] S. Bura, C.Delaye, G. Dumortier, M. Girard, L. Magnin et al., « La norme PLAGE (Plate-forme générique d'expérimentations et de simulations. », in Evolution Artificielle, 1994.

[Cambier 94] C.Cambier « SIMDELTA, un système multi-agent pour simuler la pêche sur le Delta Central du Niger », Thèse de Doctorat, Université de Paris VI, 1994.

[Cariani 91] P.Cariani « Emergence and Artificial Life », ArtificialLife II, SFI Studies in thes Sciences of Complexity, vol. X. C.G. Langton, C. Taylor, J.D. Farmer & S. Rasmussen, Addison Wesley, 1991, pp. 775..797.

[Carle 92] P.Carle « Mering IV : Un langage d'acteur pour l'intelligence artificielle distribuée intégrant objet et agents par réflexivité compilatoire », Thèse de Doctorat, Université de Paris VI, 1992.

[Carouso 94] D. Carouso : « Modélisation d'une forêt tropicale par une approche multi-agents. », Rapport de stage de DEA IARFA, Université de Paris VI, 1994.

[Cazoulat 95] R. Cazoulat : « Modélisation et simulation de la dynamique de population d'agents. », Thèse de Doctorat, Université de Caen, 1995, §3.3.1.

[Davidsson 95] P. Davidsson : « A Linearly Quasi-Anticipatory Autonomous Agent Architecture : Some Preliminary Experiments », in LNAI 1087, Architecture and Modelling, Camberra, Australia 1995, pp 189..203.

[Delaye 94] C. Delaye, L. Magnin : « Systèmes multi-agents, simulation et règles environnementales (l'expérience Plages)», Journées Systèmes Multi-Agents, P.R.C.-G.D.R. IA, Paris, 16 décembre 1994.

[Delaye 95] C. Delaye : « », Thèse de Doctorat, Université de Paris VI, 1995.

[Demazeau 90] Y. Demazeau, J.-P. Müller, Eds. Decentralized A.I. Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW' 89), Elsevier/North-Holland.

[Demazeau 91] Y. Demazeau, J.-P. Müller, Eds. Decentralized A.I.2 Proceedings of the Second European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW' 90), North-Holland.

[Drogoul 93A] A. Drogoul : « De la Simulation Multi-Agents à la Résolution Collective de Problèmes : Une Etude de l'Emergence de Structures d'Organisation dans les Systèmes Multi-Agents », Thèse de Doctorat, Université de Paris VI, 1993.

[Drogoul 93B] A. Drogoul, B. Corbara, D. Fresneau : « Manta : New experimental results on the emergence of (Artificial) Ant Societies. », in Simulating Societies Symposium, Siena, C.Castelfranchi (Ed.).

[Ferber 90] J. Ferber : « Conception et programmation par Objet », Hermès, Paris, 1990.

[Ferber 91] J. Ferber, P. Carle : « Actors and Agents as Reflective Concurrent Objects : A Mering IV Perspective », IEEE Trans on Systems, Man and Cybernetics. 21, 1991.

[Ferber 93] J. Ferber, L. Magnin : « Conception de systèmes multi-agents par composants modulaires et réseaux de Petri », P.R.C.-G.D.R. IA, Montpellier, 17 décembre 1993.

[Ferber 95] J. Ferber « Les Systèmes Multi-Agents : Vers une intelligence collective », InterEdition, 1995.

[Fisher 95] M. Fisher « Representing ans Executing Agent Based Systems », Lecture Notes in Artificial Intelligence, LNAI 890, Wooldridge & Jennings ed. , pp 307..323.

[Fodor 86] J. Fodor « La modularité de l'esprit», Editions de minuit, 1986.

[Kumeno 95] F. Kumeno, Y. Tahara, A. Ohsuga, S. Honiden « Evolutional Agents : Field Oriented Programming Language, Flage », IEEE ASPEC, 1995, pp. 189..198.

[Gamma 96] E. Gamma, R. Helm, R. Johnson, J. Vlissides « Design Pattern : Catalogue de Modèles de Conception Réutilisable », Thomson Publishing, 1996.

[Girard 98] M. Girard-Faugère « ADAM : un gestionnaire d'applications réparties en environnement dynamique», Thèse de Doctorat, Université de Paris VI, 1998.

[Giroux 93] S. Giroux « Agents et Systèmes, une Nécessaire Unité », Thèse de Doctorat, Université de Montréal, 1993.

[Guessoum 96] Z. Guessoum « Un environnement opérationnel de conception et de réalisation de systèmes multi-agents », Thèse de Doctorat 96/09, Université de Paris VI, 1996.

[Hewitt 77] C. Hewitt « Viewing control structures as patterns of messages passing », Artificial Intelligence 8 (3), 1977.

[Iffenecker 92] C. Iffenecker « Un système Multi-Agents pour le support des activités de conception de produits », Thèse de Doctorat 93/01, Université de Paris VI, 1992.

[Konstantas 93] D. Konstantas « Cell : A Framework for a Strongly Distributed Object Based System », Thèse de Doctorat 2598, Université de Genève 1993.

[Korf 92] R. E. Korf « A simple solution to pursuit game », Working Papers of the 11h International WorkShop on Distributed Artificial Intelligence, pp 183-194.

[LeCerf97] V. Le Cerf, M. Pintado « An Adaptative Model of Camera-Driven Urban Intersections Observation », Workshop on dynamic scene recognition, juin 1997, Toulouse.

[Lhuillier91] M. Lhuillier « Evolution morphologique et comportementale d'une société de robots», Rapport de stage de DEA IARFA, Université de Paris VI, 1991.

[Magnin 96] L. Magnin « Modélisation et simulation de l'environnement dans les systèmes multi-agents : Application aux robots footballeurs », Thèse de Doctorat 96/17, Université de Paris VI, 1996.

[Maës 91] P. MAës « A bottom-up mechanism for behavior selection in an artificial creature », in From Animal to Animats. Proceedings of the First International Conference Simulation of Adaptative Behavior. Meyer J.A. & Wilson S.W. (Eds). MIT Press 1991, pp. 238..246.

[Maës 93] P. MAës « Behavior-based artificial intelligence », in From Animal to Animats II. Proceedings of the Second International Conference Simulation of Adaptative Behavior. Meyer J.A., Roiblat H.L. & Wilson S.W. (Eds). MIT Press 1993, pp. 2..10.

[Mercier 94] F. Mercier « Modélisation de la Dynamique des Systèmes Ecologiques, à l'aide des techniques de l'Intelligence Artificielle, en particulier des Systèmes Multi-Agents », Rapport bibliographique de DEA, Laboratoire de Biométrie, Génétique et Biologie des Populations, U.C.B. Lyon I, Février 1994.

[Mey 93] V. de Mey & S. Gibbs « A Multimedia Component Kit : Experiences with Visual Composition of Applications », in Visual Objects 1993, pp 29..49.

[Mey 95] V. de Mey « Visual Composition of Software Applications », in Object Oriented Software Composition 1995, pp 275..302.

[Moffat 95] D. Moffat, N.H. Frijda « Where there's a Will there's an Agent », Lecture Notes in Artificial Intelligence, LNAI 890, Wooldridge & Jennings ed. , pp 245..260.

[Muller 97] P.A. Muller « Modélisation objet avec UML », Eyrolles, 1997.

[Nierstrasz 94] O. Nierstrasz & T.D. Meijler « Requirement for a composition Language », ECOOP 94 Workshop on Models and Languages for Coordinatin of Parallelism and Distribution, Italy, LNCS 924, Sprinter-Verlag 1994.

[Nierstrasz 95] O. Nierstrasz & L. Dami « Component Oriented Software Technology », in Object Oriented Software Composition 1995, pp 3..28.

[Pintado 91] X. Pintado « Gluons : Connecting Software Components », Working Paper in Object Composition 1991, pp 73..83.

[Perez 90] J.P.Perez « Systèmes Temps Réels » Dunod, 1990.

[Reiss 90] S. Reiss « Connecting Tools Using Message Passing in the FIELD Environment », IEEE Sotware, July 1990, pp. 57..66.

[Roche 93] C. Roche, Y. Dumond, S. Stinckwich « LRO3 : un modèle pour la programmation par objets parallèle », in Génie Logiciel & Systèmes Experts, N° 32 Septembre 1993, pp 48..57.

[Shoham 93] Y. Shoham « Agent-oriented programming », Artificial Life 60, 1993, pp 139..159.

[Vinoski 97] S. Vinoski « CORBA : Integrating Diverse Applications Within Distributed Heterogeneous Environments », IEEE Communication Magazine, Vol 14, N°2, Fevrier 1997.

[Yoo 97] M.Y. Yoo « Modélisation des Systèmes Coopératifs et Validation de leurs Protocoles de Communication », Rapport de Pré-Soutenance de Thèse, Laforia, Juin 1997.

[COM, DCOM] Architectures COM et DCOM, Draft fournis sur le site Microsoft. Notes techniques fournies avec DirectX. (http : //www.microsoft.com).

[Delphi2] Delphi 2.0 Manuels fournis avec le langage. Borland 1995.

[Delphi3] Delphi 3.0 Manuels fournis avec le langage. Borland 1997.

[Smalltalk] Smalltalk80, Visual Works. Manuels fournis avec le langage.