Technique et Science Informatiques, Hermès Science publ. www.hermes-science.com

Entre programmation par composants et langages d'experts

Rendre la modélisation individu-centrée plus accessible à l'utilisateur

David Houssin* — **Stefan Bornhofen*** — **Sami Souissi**** — **Vincent Ginot***

* *INRA, Station d'Hydrobiologie Lacustre de Thonon-les-Bains, BP 511, 74203 Thonon les bains Cedex. ginot@avignon.inra.fr*

** *Université des Sciences et Technologies de Lille 1. CNRS - UA 8013 ELICOStation Marine, BP 80, 62930 Wimereux. sami.souissi@univ-lille1.fr*

RÉSUMÉ. *L'irruption récente de la modélisation individu-centrée dans le domaine de la dynamique des populations fait de plus en plus appel au concept multi-agents. Mais elle n'a pas encore été suivie d'une offre logicielle permettant à un utilisateur non informaticien de construire et d'utiliser ses propres modèles. Dans ce domaine, il est pourtant souvent possible d'écrire l'intégralité de la dynamique avec des agents réactifs. Toute l'information sur le système est alors portée par des agents dont le comportement est simple. La construction d'un modèle par assemblage de composants s'en trouve facilitée. Notre approche rejoint des concepts informatiques récents sur les modèles objets dynamiques, le flot d'exécution de tâches, ou l'élaboration de langages d'experts.*

ABSTRACT. *The raise of individual based modelling in ecology has not yet been followed by softwares enabling the biologists to tailor there own models. A way to build such tools is to ground them on the multi-agents concept and on component programming. In population dynamics, it is often possible to translate all the dynamical aspects of a model into agents. Hence, information on the system is only supported by agents, and the components involved in model building only deal with a few number of structures. This greatly facilitate the design and use of these components. Our approach and needs join a number of recent works in the field of dynamics objects models, micro-workflow or end-user programming.*

MOTS-CLÉS : *simulations multi-agents (SMA) ; dynamique des populations ; modélisation individu-centrée ; programmation orientée utilisateur ; atelier d'expert.*

KEY WORDS : *multi-agents systems (MAS) ; population dynamics ; individual-based modelling (IBM) ; component programming ; end-user programming.*

1. Introduction

La modélisation individu-centrée est de plus en plus employée en écologie et plus particulièrement en dynamique des populations. Et de plus en plus, elle repose sur le concept informatique des systèmes multi-agents (SMA) qui apporte des ouvertures indéniables en simulation, et dont on trouvera un descriptif détaillé dans [FER 95] par exemple. Pour autant, la mise en œuvre d'un modèle individu-centré et plus encore d'un système multi-agents est délicate et demande un investissement important [HOG 90]. Le candidat modélisateur aura donc intérêt à utiliser un environnement de développement adéquat. Ce domaine est actuellement en pleine effervescence et nous en donnerons quelques exemples plus loin. Mais force est de reconnaître avec [LOR 99] que l'on se trouve soit en présence de simulateurs trop spécifiques pour répondre à de nouvelles questions, soit en présence de plates-formes effectivement plus génériques mais reposant sur un langage de programmation bien difficile d'accès au non informaticien.

Une deuxième contrainte est qu'avec un SMA il est encore difficile d'utiliser un formalisme, équivalent à un système d'équations différentielles par exemple, dans lequel on puisse exprimer son modèle, le conserver, le comparer à d'autres ou l'exporter vers un autre environnement de travail. En pratique, le modèle fait partie intégrante du simulateur chargé de son exécution, et une personne extérieure à sa construction, à commencer par le chercheur lui-même, aura toujours du mal à comprendre en quoi consiste le modèle, et une défiance bien naturelle vis-à-vis d'un simulateur boîte noire dont il n'est pas bien convaincu qu'il fasse exactement ce qu'il est supposé faire. Car si le chercheur doit pouvoir effectuer des simulations, il doit aussi pouvoir les interpréter. Et pour cela il a besoin de bien savoir ce que fait la machine.

Inaccessibilité et manque de lisibilité sont donc à nos yeux deux freins importants au développement des SMA dans le domaine des simulations individus-centrées. On pourrait en ajouter un troisième, qui sort un peu du cadre informatique de cet article : l'absence d'une véritable méthodologie de conception, de validation et d'utilisation de ce type de modélisation. Après un rapide panorama de quelques environnements de développement plus ou moins couramment utilisés en écologie, nous présentons les grandes lignes des travaux que nous avons pu mener sur ce sujet. Nous concluons en essayant de situer notre démarche par rapport aux évolutions récentes en informatique dans ce domaine.

2. Simulateurs et environnements de développement utilisés en écologie

Selon une classification proche de celle introduite par [LOR 99], on peut classer les environnements de développement en trois catégories, du plus général au plus spécifique. Sans prétendre être exhaustif, nous en donnons quelques exemples

utilisés en écologie, qui nous semblent à la fois significatifs et surtout suffisamment documentés pour être concrètement utilisables.

Les plates-formes génériques reposent sur un langage de programmation et facilitent l'écriture de SMA quel que soit leur domaine d'application. En écologie, la plus connue est sans conteste *Swarm* [MIN 96], largement utilisée par la communauté des chercheurs en Vie Artificielle, et que bien des simulateurs évoqués plus bas utilisent. *StarLogo* [RES 96] vient déjà loin derrière, et repose sur un macro-langage qui permet de piloter des agents sur une grille spatiale. L'offre française, dont une bonne partie se trouve résumée sur le site AFIA[1], est importante, même s'il s'agit souvent de prototypes encore en développement. Certaines de ces plates-formes ont eu des applications en écologie ou en biologie. Citons *Geamas* [MAR 97], sous Java, qui vient du monde de la géophysique. Son originalité est d'offrir une structure d'agents par couche (notion de groupe et de relations hiérarchiques entre agents) et des processus d'agrégation et de désagrégation entre ces niveaux. *Dima* [GUE 98], également en Java, propose des modules comportementaux (communication, perception, raisonnement…) pour créer des agents aussi bien réactifs que cognitifs. *MadKit* [GUT 97], toujours sous Java, met l'accent sur la notion de rôle avant même celle d'agent. Elle vise l'écriture de modèles distribués sur plusieurs ordinateurs à travers la toile. Le langage *Oris* [HAR 00], dont la sémantique est proche de C++, est interprété et permet de modifier en ligne le comportement d'un modèle de réalité virtuelle.

Les plates-formes orientées écosystème proposent des utilitaires plus spécifiquement dédiés à la simulation des écosystèmes. Egalement basées sur un langage de programmation, elles gardent un fort degré de généricité. *Ecosim* [LOR 98] repose sur une importante bibliothèque C++ d'objets et un moteur de simulation à événements discrets. *Cormas* [BOU 98], sous Smalltalk, est dédié à la modélisation des interactions entre sociétés humaines et ressources renouvelables. Son originalité est de proposer un support spatial évolué intégrant des échelles multiples et des liens dynamiques avec les SIG.

Les plates-formes dédiées, ou « simulateurs ». Sauf exception, il s'agit plus d'un environnement performant de paramétrage et d'utilisation d'un modèle que d'un outil de développement à proprement parler, sauf à entrer (en général laborieusement !) dans la programmation du simulateur pour l'adapter à un nouveau modèle. A titre d'exemples et pour montrer la diversité des systèmes écologiques étudiés, *Manta* [DRO 92] s'intéresse à l'émergence de la spécialisation des tâches dans une société d'insectes. *SugarScape* [EPS 96] laisse des agents s'organiser pour exploiter deux ressources, l'une abondante et bon marché, le sucre, l'autre rare et chère, l'épice. *Formosaïc* [LIU 98] modélise les dynamiques forestières morcelées. *BacSim* [KRE 98] s'intéresse aux dynamiques micro-biologiques. *Wesp-Tool* [LOR 99], dédiée à la dynamique des métapopulations, est basée sur la notion de rôles (œuf, juvénile,

[1] http://www-poleia.lip6.fr/~guessoum/afia.html

adulte…) et sur la formalisation de leurs transitions. *Osmose* [SHI 01] s'appuie sur une base de données piscicole pour paramétrer des espèces virtuelles de poissons entrant en interaction par prédation.

Si l'on se place du point de vue d'un biologiste souhaitant écrire lui-même ses modèles, le choix est en fait assez restreint. Si l'on devait sélectionner un outil dans chaque catégorie, nous prendrions *StarLogo* dans la première, car il permet effectivement de commencer à écrire des modèles en quelques minutes avec quelques macros simples. Dans la deuxième catégorie nous choisirions *Cormas*, parce que le langage Smalltalk est un des plus accessibles au non informaticien, et parce qu'il existe déjà une communauté importante d'utilisateurs. Dans la troisième catégorie, l'outil le plus souple nous paraît être *Wesp-Tool*, mais en sachant qu'il est encore trop proche des simulateurs à structure rigide et que l'utilisateur se retrouvera vite à écrire du C++ pour aller modifier ce qui l'intéresse.

Il semble pourtant possible de proposer un outil de création et d'utilisation de modèles individus-centrés qui soit vraiment accessible au biologiste, et ceci sans trop restreindre son domaine d'application. L'idée est que la tâche de l'informaticien n'est plus d'offrir un modèle, ou dans le meilleur des cas des agents clef en main comme dans les simulateurs actuels, mais des composants beaucoup plus élémentaires à partir desquels un utilisateur pourra construire ses agents, et un environnement de simulation dans lequel il pourra les faire évoluer et les étudier. Nous nous situons donc dans l'optique d'une programmation par composants, opposable dans une certaine mesure à une programmation plus classique par langage.

3. Décomposer une tâche en primitives

L'idée de base de notre démarche est de faire porter l'effort de codification non pas au niveau de l'agent dans sa globalité ou au niveau de son rôle, mais au niveau des différentes actions (les tâches) que ce dernier doit exécuter, et même au niveau encore plus élémentaire de la manière dont ces tâches peuvent se décomposer. Car on peut estimer que ces dernières répondent souvent à un schéma très classique : il faut localiser l'information, la trier, la traiter, puis mettre à jour le nouvel état du système. On peut donc imaginer décomposer une tâche en un enchaînement de sous-actions élémentaires, et à l'inverse créer une nouvelle tâche en sélectionnant et en paramétrant des sous-actions bien choisies. Dans la pratique, ces sous-actions que nous appellerons des primitives par analogie avec leur usage en informatique, prennent la forme de petits modules informatiques qui échangent de l'information. Mais pour que cette décomposition en primitives soit efficace sans retomber dans la complexité d'un langage de programmation, il faut qu'elles soient simples à utiliser et qu'elles restent en nombre limité. Il faut également que l'information qu'elles vont échanger ait une structure aussi simple que possible. Ceci conduit à des choix architecturaux importants que nous allons évoquer brièvement.

3.1. *Une architecture adaptée au travail des primitives*

- *Des agents à structure simple*. Un agent se résumera pour nous à un objet qui comporte deux dictionnaires. Un dictionnaire d'attributs qui va fixer son état, et un dictionnaire ordonné de tâches exécutées séquentiellement, qui va fixer son comportement. Si le dictionnaire de tâches est vide, c'est un objet passif. Pour construire ses agents, l'utilisateur doit donc remplir ces deux dictionnaires. En ce qui concerne les attributs, et toujours par soucis de simplicité, nous nous contenterons de deux types d'attributs : les attributs standards qui possèdent un nom, une valeur et une unité, (exemple, une « taille » en cm), et les attributs de type « liste » qui sont identiques mais permettent de stocker une liste de valeurs. Pour les tâches, on peut fournir à l'utilisateur quelques tâches prédéfinies pour la plupart spécifiques à son domaine telles que vieillir, se reproduire, se déplacer, lire un scénario, ou encore se métamorphoser en un autre agent. Mais dans la plupart des cas il aura besoin de bâtir des tâches adaptées à son modèle. Comme nous le verrons plus loin, il le fera en assemblant puis en paramétrant des primitives bien choisies.

- *Des modèles « tout agent »*. Une manière simple de réduire la complexité des tâches et des primitives est de réduire le nombre d'objets sur lesquels elles doivent travailler. En particulier, on peut essayer de faire en sorte que tout ce qui appartient au modèle soit décrit par des agents aux comportements plus ou moins complexes. Toute l'information sur le système modélisé (son état) est alors exclusivement portée par des agents, et même plus précisément par leurs dictionnaires d'attributs. Les primitives n'auront donc à s'échanger que des agents ou des structures de données qui contiennent l'état de ces agents. Nous pensons que bon nombre de modèles en dynamique des populations peuvent s'écrire avec trois types d'agents seulement : **1- Les animats**, pour reprendre la terminologie introduite par [WIL 87] et par l'International Society for Adaptive Behavior (ISAB). Ils joueront pour nous le rôle classiquement tenu par les individus et les objets passifs en modélisation individu-centrée. **2- Les agents de l'espace**. Notre option « tout agent » nous oblige à faire le choix d'un espace discret formé d'agents cellules. Ceci nous coupe des modèles d'espace continu, mais en retour l'espace y gagne un comportement et le monde des automates cellulaires nous devient accessible. **3- Les agents non situés**. Dans tout modèle, il faut un support d'information, notamment pour les scénarios environnementaux (variables de forçage), et pour la collecte et la visualisation des résultats importants. Ce rôle sera typiquement celui d'un troisième type d'agent, « l'agent non situé ». Ces agents ne sont pas plongés dans l'espace comme les animats, mais sont toujours accessibles (par leur nom) par les autres types d'agents ainsi que par l'utilisateur. Un exemple d'agent non situé est l'agent « Simulateur » qui porte les caractéristiques de la simulation : pas de temps, temps écoulé et durée totale.

- *Moules d'agents et agents réels*. Dans la pratique, l'utilisateur ne complète pas directement les agents réels, mais des instances de la classe `Moule` (figure 1). Une fois ces moules définis, et pour créer les agents réels, le système utilise des instances

de la classe `Agent` dans lesquelles le dictionnaire des attributs du moule est recopié. Chaque agent a donc bien sa propre identité, avec comme valeurs initiales les valeurs du moule. Pour le dictionnaire de tâches, pas de copie, mais un simple accès au dictionnaire du moule. Tous les agents ont donc le même comportement. L'utilisateur souhaitera naturellement lier certains paramètres des tâches à des attributs de l'agent. En écriture conventionnelle ces paramètres seraient des méthodes d'appel à des variables d'instances de l'agent. Ici, on le fait en instaurant des liens dynamiques entre paramètres des tâches et attributs (figure 1).

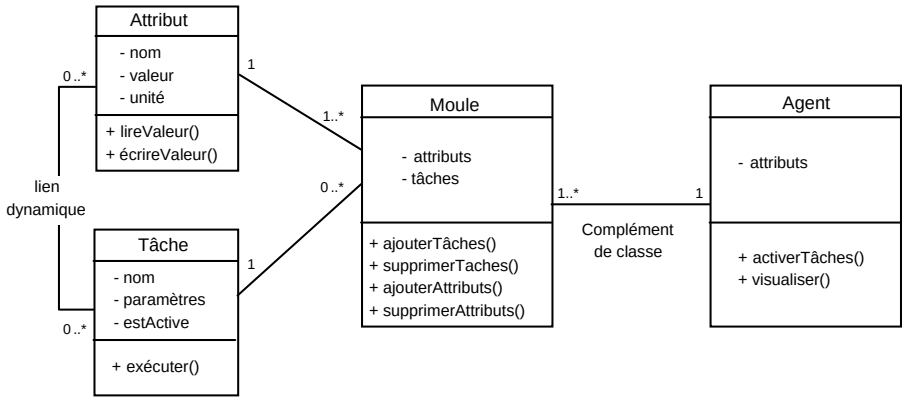

Figure 1. *Agents et moules d'agents. Un agent est une instance de la classe* `Agent`*, mais son comportement est apporté par une instance d'une autre classe, le* `Moule`

Le principe de ces liaisons dynamiques est que tâches et primitives héritent de la classe `Objet-Parametrable` qui gère ces liens. Tout paramètre susceptible de recevoir une liaison dynamique doit être un objet de la classe `Paramètre` et doit se trouver dans le dictionnaire de paramètres de `ObjetParametrable`. Pour instaurer une liaison, le système remplace `Paramètre` par un autre objet, le `Capteur`, qui redéfinit la méthode #valeur pour aller chercher dynamiquement cette valeur dans un attribut, alors que #valeur de `Paramètre` renvoie simplement la valeur stockée dans cet objet. Pour cela, `Capteur` dispose d'une variable d'instance qui fournit le chemin d'accès à cet attribut et qui comprend deux champs : le premier désigne l'agent possesseur, et le deuxième l'attribut visé. Ce faisant, un capteur au sein d'une tâche ou d'une primitive peut aussi bien pointer vers le porteur de la tâche (mon_nomAttribut), que vers tout agent accessible dans le contexte de cette tâche, et en particulier la cellule (maCellule_nomAttribut), un agent non situé désigné par son nom (nomAgent_nomAttribut), ou encore un agent que la primitive peut recevoir en argument (son_nomAttribut), voire même la cellule de ce dernier (saCellule_nomAttribut). Un exemple en sera donné figure 4. Dans cette architecture, un agent ne s'adresse donc pas à proprement parler à un autre agent, et il n'y a pas de notion de message. Il lance l'exécution des tâches de son moule, et ces

dernières ont si nécessaire accès aux attributs de tous les agents accessibles dans leur contexte d'exécution.

- *Quatre structures et trois mondes pour les agents.* Pour assurer les quatre opérations que sont la définition, la vie, la synchronisation et la mémorisation des agents, nous proposons de leur donner une structure quadruple : (1) **le moule** et (2) **l'agent réel** déjà présentés, (3) **l'image**, qui est un instantané de l'état de l'agent à un instant donné, et (4) **l'historique**, qui mémorise les états successifs de l'agent dans le temps. Ces quatre structures sont héritées de la même classe, qui gère en particulier l'accès aux attributs. On accède donc de la même manière à l'état d'un agent réel, de son moule, de son image, ou de son historique. Ces structures sont plongées dans trois mondes différents, le monde réel, le monde image, et le monde mémoire. Ces trois mondes partagent également la même structure si bien que la localisation et le tri des agents se fait de la même manière quel que soit le monde auquel on s'adresse. Il en résulte que bien souvent, une primitive composant une tâche n'a pas besoin de savoir sur quel type de monde et sur quel type de structure d'agent elle travaille. Bien plus, elle l'ignore en général, et c'est ce qui sert de base à la gestion de la synchronisation entre agents.

- *Gestion du temps, synchronisation et ordonnanceur.* Par simplicité, nous avons retenu une gestion de type horloge à pas fixe. Pour la synchronisation, deux structures sont utilisées : le monde réel et le monde image. En mode séquentiel, l'ordonnanceur donne la main séquentiellement aux agents existants au début du pas de temps courant selon un tirage aléatoire sans remise, et fournit toujours le monde réel aux tâches à travers la variable `unMonde`. En mode parallèle il faut que tous les agents « voient » la même chose au cours du pas de temps. Pour cela, l'ordonnanceur crée une image du monde avant d'appeler les agents concernés, et la bascule vers les tâches, utilisant le fait que tâches et primitives n'ont pas besoin de savoir sur quel monde elles travaillent (figure 2). Grâce à l'objet `Capteur` et à ses liens dynamiques, il leur suffit de toujours lire dans la variable `unMonde` que leur fournit l'ordonnanceur, et de toujours écrire dans `leMondeRéel`. Les agents sont également appelés aléatoirement sans remise, mais la résolution des éventuels conflits de ressources reste à la charge du modélisateur. Dans la pratique, le mode parallèle est plutôt utilisé pour les cellules, et le mode séquentiel pour les animats. L'ordonnanceur permet également à l'utilisateur de spécifier le type de synchronisation à appliquer, ainsi que l'ordre d'appel des actions élémentaires sur un pas de temps : activation et sauvegarde des différents groupes d'agents, mise à jour de la visualisation, définition des résultats à mémoriser (figure 3).

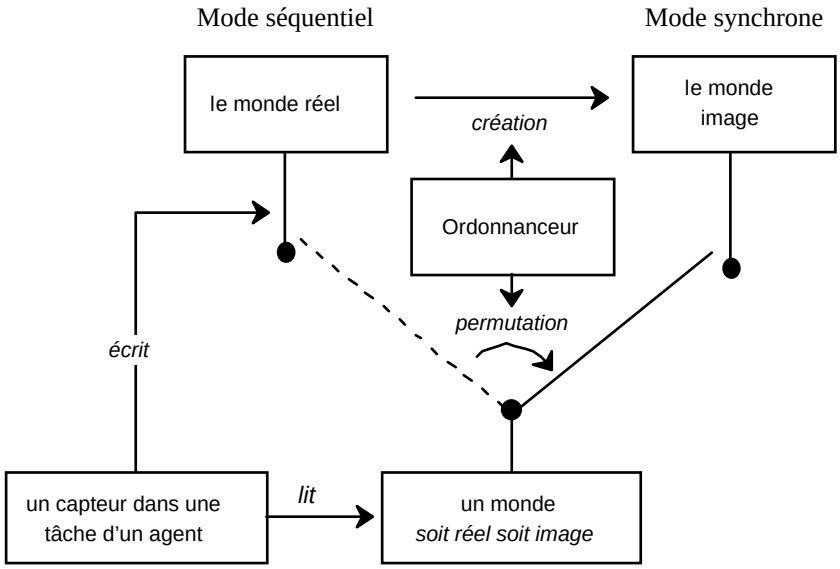

Figure 2. *Synchronisation. En mode séquentiel, les primitives lisent et écrivent dans le monde réel. En mode parallèle, l'ordonnanceur crée une image du monde et la bascule vers les tâches afin que tous les agents concernés voient la même chose*

Figure 3. *Ordonnanceur. Il sert à orchestrer le déroulement d'un pas de temps, avec notamment la séquence d'appel et de sauvegarde des différents types d'agents. Il est configuré ici selon l'exemple de [DEA 79]. On notera que l'utilisateur a défini un agent non situé appelé « Observateur », et qu'il a choisi de faire jouer d'abord les cellules en mode synchrone puis les animats en mode séquentiel, et enfin l'agent observateur*

3.2. *Construire une tâche avec des primitives*

Argument échangé et flot d'exécution. Les primitives vont donc s'échanger des agents, ainsi que leurs historiques (les agents pourront avoir de la mémoire). Mais il peut aussi être utile de s'échanger directement des données reflétant l'état des agents. Nous avons opté pour un tableau unique à trois dimensions, le `Tableau3D`. La première dimension est le nombre d'agents différents que l'on souhaite traiter en même temps, la deuxième est le temps (utilisée dans le cas d'historiques), et la troisième est fournie par la donnée elle-même (utilisée si attribut de type « liste »). On remarquera que ce tableau n'intègre pas la dimension des différents attributs d'un même agent. L'intégrer présente en effet peu d'intérêt car on traite rarement ensemble des attributs de nature différente, et deux inconvénients majeurs : la perte d'homogénéité du tableau et l'impossibilité d'y conserver des valeurs en provenance d'agents différents ayant tout de même au moins un attribut commun. En ce qui concerne le flot d'exécution, toujours par soucis de simplicité, nous nous contenterons d'une séquence linéaire de primitives à argument unique, l'argument de sortie d'une primitive formant l'argument d'entrée de la primitive suivante.

Construire une tâche consiste donc à sélectionner des primitives et à les enchaîner. Le tableau 1 présente les 25 primitives que nous avons actuellement définies, réparties en 6 groupes : recherche, sélection, traduction, calcul, clôture et contrôle.

Type	Fonction	Primitives
Recherche	Point de départ typique d'une tâche. Pas d'argument en entrée, renvoie en sortie des agents (réels ou mémoires) ou un `tableau3D` contenant les valeurs mémorisées dans le temps d'un attribut.	Moi, MaCellule, TousLesAnimats, ToutesLesCellules, MonVoisinage, MonHistorique, MonHistoriqueAttribut
Sélection	Parmi la liste des agents fournis en entrée, en sélectionne certains (ou un seul) sur le nom, sur des conditions à satisfaire, ou aléatoirement	TrierSurNom, TrierSurValeurs, SousSélectionAléatoire, ChoixFinal
Traduction	Passer des animats à leurs cellules ou vis versa, passer des agents aux historiques ou au `tableau3D`, passer d'une cellule à son voisinage…	DeAnimatsVersCellules, DeCellulesVersAnimats, DeAgentsVersHistoriques, DeAgentsVersValeurs, Voisinage
Calcul	Compter les objets reçus en entrée, réaliser des calculs mathématiques impliquant des attributs d'agents, réaliser des calculs simples (somme, moyenne, max, min…) sur un `tableau3D`…	Compter, ModifierAttributs, CalculSurT3D
Clôture	Plus typiquement en fin de tâche. Pour sauvegarder une valeur (ou un vecteur) d'un `tableau3D` dans un agent, se déplacer ou tuer un agent.	SauverValeurs, DéplacementCiblé, Tuer
Contrôle	Pour sortir du mode strictement enchaîné des primitives : arrêter le flot d'exécution, l'exécuter en boucle ou dérouler la liste fournie en entrée pour donner les objets un par un aux primitives suivantes.	Conditions, TantQue / FinTantQue, DéroulerUneListe / FinDéroulerUneListe

Tableau 1. *Répartition des primitives en six groupes fonctionnels*

Parmi ces primitives, la primitive de calcul `ModifierAttributs` occupe une place centrale car elle utilise un formalisme mathématique permettant d'écrire les interactions entre agents (figure 4).

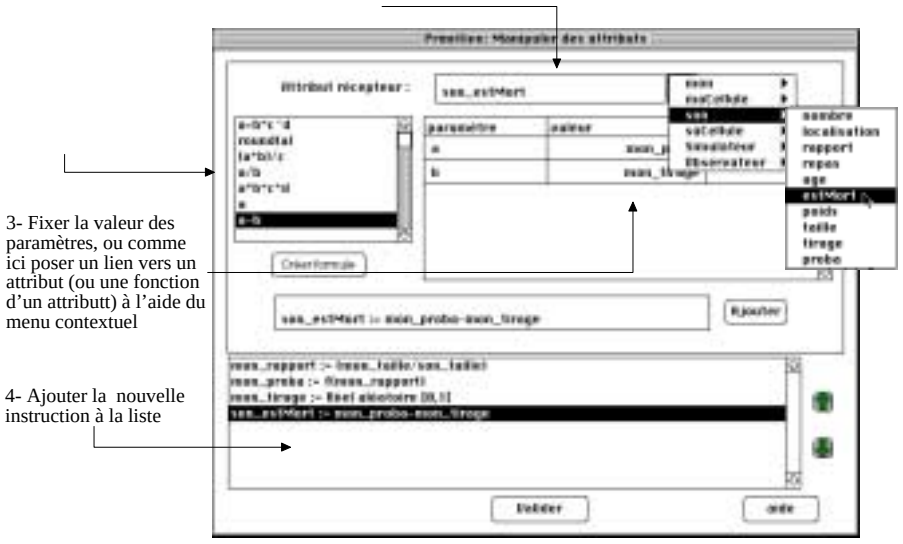

Figure 4. *La primitive* `ModifierAttributs`*, telle qu'utilisée dans l'exemple de [DEA 79]. (voir texte et figure 5). Cette primitive est au cœur de tous les calculs impliquant des attributs d'agents, dès lors qu'ils sont visibles dans son contexte d'exécution. L'accès aux attributs se fait par les liens dynamiques évoqués au 3.1*

Une fois que l'utilisateur a spécifié sa séquence de primitives, un interpréteur contrôle la cohérence des enchaînements, et en particulier la nature et la dimension de l'argument échangé. Il crée ensuite le code Smalltalk de la nouvelle tâche qui viendra s'ajouter, sous la forme d'une nouvelle classe, aux autres tâches de l'utilisateur et aux éventuelles tâches déjà fournies par le simulateur. Cette nouvelle tâche possède sa propre interface formée des interfaces spécifiques des primitives qui la compose. Les tâches créées sont donc elles-mêmes très flexibles et peuvent être conservées par l'utilisateur pour des agents et des modèles différents, au même titre que les tâches prédéfinies. Cette particularité fait en grande partie l'originalité de notre projet et le distingue des simulateurs et des bibliothèques d'objets traditionnels : l'utilisateur complète dynamiquement sa propre bibliothèque de comportements. Une tâche, quelle soit prédéfinie ou construite par l'utilisateur, est considérée comme une primitive sans argument. Elle peut donc elle-même entrer dans la composition d'une tâche plus complexe (récursivité). Un exemple simple est de lui adjoindre une primitive de contrôle : la tâche s'exécute *si*, ou bien s'exécute *tant que*.

La capacité de ce concept à permettre la création de modèles variés a été testée en reprenant trois exemples parmi les plus célèbres de la littérature dans notre domaine : le modèle de [DEA 79] souvent considéré comme fondateur en modélisation individu-centrée, l'incontournable « jeu de la vie » de John Conway [GAR 70] pour illustrer le fonctionnement en automate cellulaire, et le modèle « Sugar Space » de [EPS 96] qui fit grand bruit dans les milieux économistes et qui met aux prises des agents plus ou moins bien adaptés, mais qui peuvent évoluer au cours des générations, face à un environnement parfois rétif. Le lecteur pourra se référer à ces articles pour plus de détails ou consulter le site http://www.avignon.inra.fr/mobidyc où nous présentons ces exemples et où nous décrivons plus en détail la plate-forme que nous avons développée selon ces concepts. Son nom est MOBIDYC, acronyme de MOdélisation Basée sur les Individus pour la DYnamique des Communautés. Elle reprend l'architecture que nous venons de décrire et propose un environnement complet de création et d'utilisation de modèles dans le domaine de la dynamique des populations, incluant la définition d'expériences simulatoires permettant de les tester.

Nous nous contenterons ici d'illustrer l'usage des primitives avec la tâche de chasse déjà complexe des agents du modèle de [DEA 79]. Rappelons ce comportement. A chaque pas de temps représentant une journée, chaque poisson joue à tour de rôle et peut atteindre tous ses congénères. Il en choisit un au hasard et le pourchasse. Il arrive parfois à le capturer selon une probabilité de capture qui dépend du rapport de taille entre les deux poissons. S'il le capture, il le mange et augmente son poids en conséquence. S'il le rate il le laisse tranquille et cherche une autre proie. Le cycle de chasse s'arrête lorsque tous les poissons ont été pourchassés ou lorsque le poisson n'a plus faim. Le poisson suivant joue à son tour, et ainsi de suite pour tous les poissons. L'enchaînement de primitives qui en résulte est illustré sur la figure 5. On remarquera en particulier la primitive de contrôle `Dérouler-LesEntrées` qui permet de construire le cycle de chasse des poissons, ainsi que l'incontournable primitive `ModifierAttributs` qui permet de faire les calculs entre le prédateur et sa proie (figure 4).

Le principe de la décomposition des tâches en primitives joint à une architecture adaptée semble donc fécond pour aider l'utilisateur dans la création de ses modèles en dynamique des populations. A ce jour, quatre biologistes (DEA ou doctorants), ont pu tester ce concept. MOBIDYC étant encore en version béta, la prise en main s’est effectuée par une visite de deux ou trois jours dans nos locaux, une bonne moitié étant d’ailleurs passée à discuter du modèle plutôt que de l’outil. Les utilisateurs ont ensuite réalisé l’essentiel des simulations chez eux. L’objectif de la version finale, avec aide en ligne et tutoriaux, est qu’un utilisateur non informaticien puisse se lancer seul en une journée.

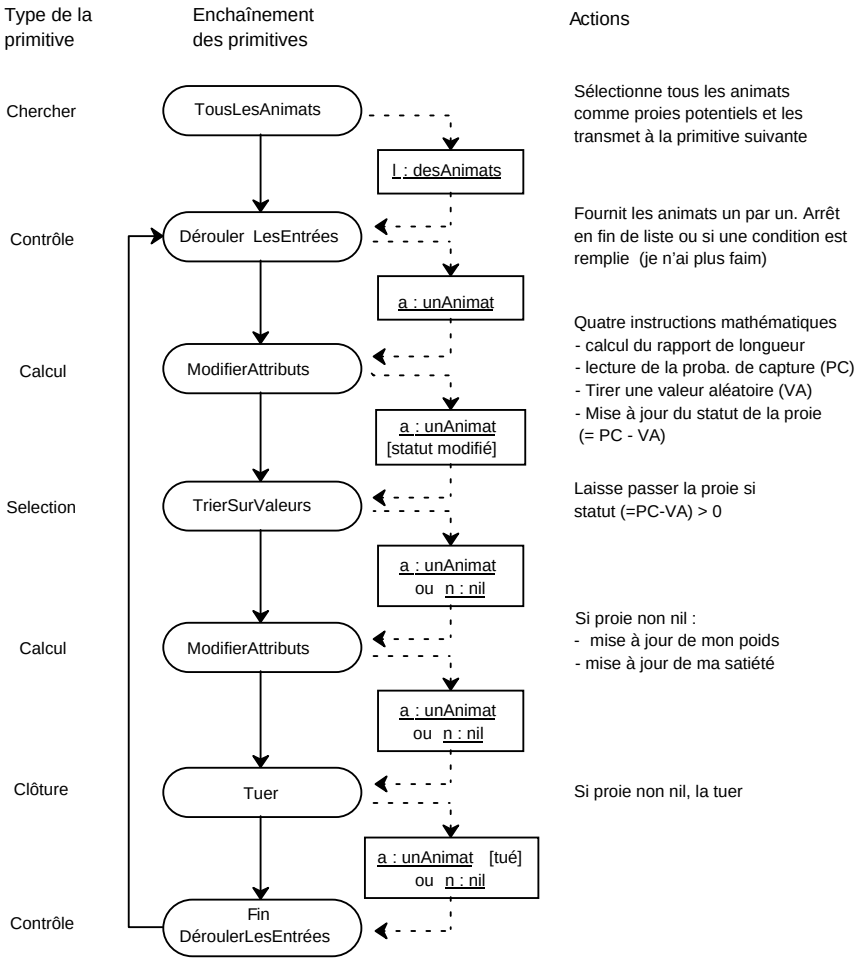

Figure 4. *Construire une tâche en enchaînant des primitives. Diagramme d'activité UML de la tâche de chasse du modèle de [DEA 79] (voire texte)*

Le deuxième axe de nos travaux a cherché à rendre ces modèles plus lisibles et a montré l'intérêt d'intégrer un langage de description des modèles dans les environnements de développement.

4. Vers un langage de description des agents et des modèles en dynamique des populations

4.1. *Les motivations*

Trois raisons au moins militent en faveur d'un langage de description des modèles individus-centrés en écologie, la première et la plus importante ayant été évoquée en introduction :

1- *La lisibilité*. Dans les simulateurs utilisés aujourd'hui en écologie il n'existe aucun lien formel entre le discours du modélisateur décrivant son modèle et les résultats qui sortent de la machine. Comment un observateur, en particulier un expert du domaine, peut-il s'assurer que la machine exécute bien le modèle, et donc faire confiance aux interprétations qui découlent de son exécution ? Cette question du formalisme est cruciale pour le développement des simulations multi-agents en écologie et nous pensons que tant qu'il ne sera pas résolu, ce type de modélisation ne sera pas crédible car trop difficilement vérifiable.

2- *La durée de vie des modèles*. En absence de formalisme, un modèle sera sauvegardé sous la forme de fichiers binaires, le Binary Object Streaming Service (BOSS) de Samlltalk dans notre cas. Avantage, il n'y a aucune traduction entre le modèle qui s'exécute et celui qui est sauvegardé. Il s'agit du même codage. Inconvénient, les objets du modèle doivent correspondre bit pour bit aux objets manipulés par la plate-forme. La moindre modification faite ultérieurement sur un objet utilisé par un modèle fait que ce dernier ne sera plus rechargeable.

3- *La lenteur des calculs et le coût mémoire*. Bien qu'il soit en partie compilé, Smalltalk reste un langage interprété et sa gestion mémoire est assez lente. Ce langage est donc idéal pour le prototypage d'applications complexes, mais moins adapté à l'exécution d'un modèle de simulation exigeant de grosses ressources calcul et mémoire. Un langage permettrait de traduire un modèle élaboré interactivement sous Smalltalk, puis de le recompiler vers un module plus rapide chargé de son exécution.

4.2. Intégrer un langage à une plate-forme de développement de modèles

Le fait de se restreindre à un domaine d'application particulier et de se limiter à des modèles « tout agent » ont facilité la définition d'un tel langage et son intégration à une plate-forme. Ceci, joint à la souplesse de Smalltalk, fait qu'un prototype opérationnel a pu être proposé en quelques mois seulement [HOU 98]. Ce travail à vocation exploratoire était surtout destiné à montrer la faisabilité d'un tel langage et à réfléchir à une architecture pouvant en tirer le meilleur parti. Ceci a conduit à enrichir la plate-forme des éléments suivants :

- un langage de description des modèles baptisé ATOLL,
- un outil de traduction des objets constitutifs du modèle vers ce langage,
- un P-code intermédiaire entre Smalltalk et le langage cible,
- un interpréteur syntaxique d'ATOLL producteur de P-code,
- un interpréteur P-code -> Smalltalk,
- un interpréteur P-code -> C++,
- un moteur C++ dépourvu d'interface pour l'exécution rapide des simulations.

L'architecture générale est illustrée sur la figure 6. Bien que pour des raisons pratiques une petite interface de programmation en ATOLL ait été écrite, il ne s'agit pas d'inviter le biologiste à écrire directement dans ce langage : le progrès ne serait pas très grand. Le principe est que le modèle, construit et testé intéractivement sur la plate-forme, est ensuite traduit vers le langage. Cette traduction est facilitée par la structure hiérarchique des différentes composantes d'un modèle : modèle -> Entités -> Agents -> Tâches -> Primitives. On notera également le choix technique d'un P-code intermédiaire pour éviter d'avoir à écrire l'interpréteur syntaxique dans plusieurs langage.

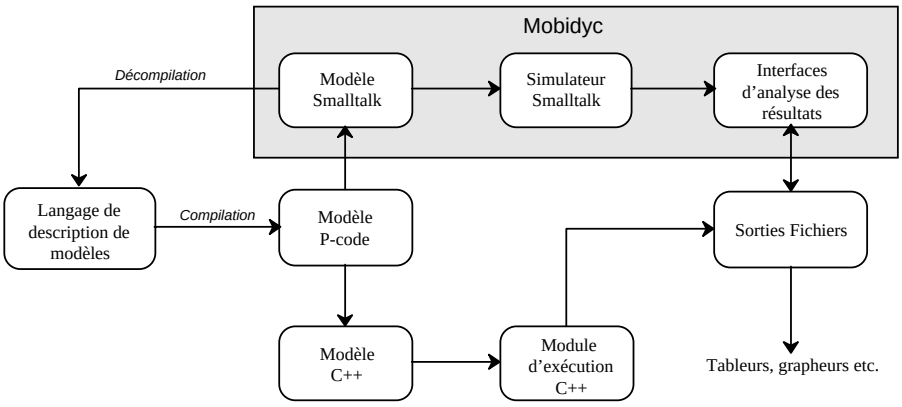

Figure 6. *Intégration d'un langage formel. L'utilisateur utilise la plate-forme interactive pour construire et tester son modèle. Ce dernier est ensuite traduit vers le langage pour en donner une sauvegarde indépendante de la plate-forme. Il peut ensuite être recompilé vers un module d'exécution plus rapide dépourvu de toute interface*

L'ensemble fonctionne dans son intégralité sur des modèles (très) simples, l'exercice, en temps limité, ayant consisté à définir et tester sa cohérence sans entrer dans les détails. Ces exemples ont en particulier montré que le module d'exécution C++ laisse espérer un gain d'un facteur au moins 5 en temps calcul par rapport à

Smalltalk dès que l'on dépasse quelques milliers d'agents, et un gain du même ordre sur le nombre d'agents que pourront contenir les modèles. Le travail a surtout porté sur la syntaxe du langage et du P-code correspondant au détriment du traducteur, qui est pourtant un élément clef de l'ensemble. A cette époque en effet, le travail sur les primitives paramétrables telles que nous les entendons aujourd'hui n'était pas terminé. Nous décrivons brièvement la syntaxe du langage pour en tirer quelques enseignements et le lecteur pourra se référer à [HOU 98] pour plus de détails sur l'organisation de la grammaire et la structure du P-Code.

La grammaire du langage est assez limitée et porte principalement sur :

- Des mots-clefs délimitant différentes zones de texte qui correspondent aux principaux objets du modèle et de la simulation : `[ Entite ]`, `[ Stades ]`, `[ Attributs ]`, `[ Tache ]`, `[ Batch ]`.

- L'appel et la structuration des tâches : *`faire`*, *`si`*, *`alors`*, *`sinon`*.

- Des locutions réservées qui seront converties en primitives de filtres ou de conditions : *`les agents sont`* (ou *`ne sont pas`*) `des Liste_d'agents` ; *`dans un rayon de`* `rayon_action` ; *`en`* `une_saison` ; *`pour une probabilité de`* `un_réel`.

- La référence aux attributs : *`mon`*, *`ma`*, *`cible`*, *`cellule`*.

- Des mots réservés s'ajoutent à cette grammaire qui sont les noms des primitives ou des tâches prédéfinies, ou encore des noms de paramètres prédéfinis de certaines primitives.

Compte tenu de la présence de mots réservés qui appartiennent au domaine d'application spécifique auquel ces modèles sont dédiés, on peut le considérer comme un macro-langage ou un langage de script. Reprenons un exemple proche de notre poisson du modèle de [DEA 79] qui souhaitait manger ses congénères. Dans l'état actuel du langage la partie qui concerne ce poisson, ici un brochet, s'écrirait (mots réservés en italique) :

```
[ Entite ]
Nom : Brochet ;

[ Stades ]
adulte

[ Attributs ]
rayonPredation adulte = 2 ;
poids adulte = 300 g ;

[ Tache ]
proies = ( juvéniles de gardon, adulte de brochet ) ;
grossir alors ( affectation poids = mon poids + poids_cible ) ;
déplacement adulte faire ( marcheAléatoire [ 2 ] ) ;
prédation adulte
   si ( mon age > 3 semaines )
        et dans un rayon de rayonPrédation
```

```
et les agents sont des proies
et ( taille_cible < 5 cm )
alors ( grossir, tuer ) ;
```

Ce langage, bâti rapidement, comporte bien des défauts et, plus grave, certaines ambiguïtés. Nous nous attacherons simplement à dégager quelques intérêts et les défauts majeurs. Au chapitre des choses intéressantes :

- Le découpage par mots-clefs de zones de texte. Il permet à l'interpréteur de comprendre différemment une même syntaxe selon les zones et d'envisager une lecture hyper-textuelle.
- Une structuration peut être un peu rigide mais récursive des tâches qui sont de type *alors, si alors,* ou *si alors sinon*. Elles sont récursives au sens où une instruction (donc dans la liste des *alors* ()) peut elle-même être une tâche qui peut contenir ses propres conditions (ex. la tâche grossir qui se trouve intégrée à la tâche prédation).
- Le fait de pouvoir remplacer le nom d'une primitive par une locution plus claire.
- La référence explicite aux propriétaires des attributs.
- Le fait de pouvoir définir certains objets avant de les utiliser (ex. les *proies* et la tâche grossir).
- Le fait d'intégrer la syntaxe mathématique standard.

Au chapitre des défauts majeurs :

- Aucune structure pour définir un éventuel paramétrage des tâches et des primitives. Il se fait soit de manière explicite mais peu lisible derrière la primitive (ex. `marcheAléatoire [ 2 ]` qui signifie « se déplacer aléatoirement dans un rayon de deux cellules ») soit de manière implicite, via des mots prédéfinis qui sont les paramètres de certaines primitives. Sur notre exemple, *proies* n'est pas une variable comme la syntaxe le laisserait croire mais un mot réservé qui représente le paramètre de la primitive de tri `les agents sont des`. Ceci conduit à multiplier les mots réservés.
- L'absence d'arguments explicites entre primitives rend le langage plus naturel mais pas très clair. Par exemple rien ne dit que les primitives de préconditions de la prédation renvoient la liste filtrée des agents aux instructions de la tâche, ni que ces instructions sont exécutées en boucle sur la liste des agents filtrés.
- L'absence de variables temporaires pousse à un usage immodéré des mots réservés ou à leur remplacement par des attributs d'agents.
- L'ordonnancement de l'exécution des différentes tâches des agents n'est pas clair.

On peut y ajouter une lisibilité qui devient vite moyenne dès que les choses se compliquent. Considérons par exemple les instructions suivantes que l'on aurait pu ajouter dans la zone [`Attributs`] :

```
puissance = a * x ^ b
poids adulte = puissance ( 0.0035 g/cm^3.35 * ma taille cm^3.35 )
```

La première ligne définit une fonction (puissance) dont la classe est créée dynamiquement par l'interpréteur syntaxique. La seconde définit un attribut contextuel (qui porte un lien dynamique vers un autre attribut) chez le brochet adulte (le poids) qui est une fonction puissance de sa taille, et fournit à cette fonction le paramétrage désiré avec les unités (facultatives mais à écrire sans blancs ni parenthèses). Cette instruction mélange donc définition, paramétrage et unités. Cela fonctionne mais ce n'est pas un modèle de lisibilité.

Malgré ces défauts deux fonctionnalités importantes sont couramment utilisées par notre plate-forme : la décompilation (traduction) de certaines primitives (notamment `Conditions`, et `ModifierAttribut`), qui fournit aux interfaces utilisateurs des sémantiques claires et concises, et la compilation des fonctions et des formules mathématiques, ce qui permet à l'utilisateur de créer les formules qu'il souhaite, notamment dans `ModifierAttribut`.

5. Discussion : entre programmation par composants et langages d'experts

A la genèse du projet, fin 1995, l'idée qu'un utilisateur non informaticien puisse créer lui-même ses propres modèles n'était pas encore très répandue. Il fallait donc faire avec les moyens du bord. Depuis, les choses ont évolué, et il est intéressant de comparer notre démarche, celle de biométriciens non nécessairement experts en informatique, aux tendances qui se dessinent aujourd'hui en informatique. Nous avons vu dans les précédentes parties que penser un environnement de développement de modèles individus-centrés nous avait conduit à :

1) choisir un concept informatique permettant de représenter ces modèles,

2) développer des outils permettant de les construire et de les utiliser,

3) essayer de proposer un formalisme qui les rendent compréhensibles par un expert du domaine et indépendant de leur plate-forme d'exécution.

En ce qui concerne le point 1, nous avons montré en introduction que le concept multi-agents semblait adapté. En ce qui concerne les points 2 et 3, nous avons opté pour une programmation par composants et pour un langage informatique assez classique, dont le prototype nous montre d'ailleurs déjà certaines limites. Sans prétendre à l'exhaustivité, étudions maintenant quelques approches informatiques actuelles sur ces deux derniers points.

5.1. *De la programmation par composants à la gestion des tâches*

Donner un comportement aux agents en assemblant des composants élémentaires n'est pas une idée neuve. [FER 95] est un des premiers à l'avoir proposé avec *BRIC*, une métaphore des circuits électroniques où le comportement d'un agent se

programme en interconnectant les bornes d'entrées et de sorties de composants judicieusement choisis. Ces composants sont soit eux-mêmes des assemblages de composants (récursivité), soit des composants dits élémentaires que l'on programme à l'aide de réseaux de Pétri colorés, soit des composants dit primitifs par analogie avec les primitives prenant en charge des tâches spécifiques. Par rapport à notre approche, l'intérêt est que l'on peut non seulement créer des comportements en agençant des composants élémentaires, mais également créer de nouveaux composants selon la sémantique des réseaux de Pétri.

[YOO 99] reprend cette idée en se focalisant sur les protocoles de coopération entre agents, c'est-à-dire sur la séquence de messages que s'envoient deux agents coopératifs en interaction. Même si ses préoccupations sont assez éloignées des nôtres, il est intéressant de noter qu'il conclut également qu'une bonne manière de créer des systèmes évolutifs dans ce domaine est de bâtir ses agents à partir de composants. Il propose donc un langage spécifique de construction de composants coopératifs basé sur la notion d'état / transition et montre qu'il se traduit facilement en réseaux de Pétri en vue de la validation formelle.

Plus proches de nous semblent les préoccupations de [RAZ 01] qui travaille sur la notion d'atelier d'experts, proposée en particulier par [NAR 93]. Pour ces auteurs, un atelier d'experts est un logiciel qui doit se transformer en une application précise entre les mains d'un expert métier sans connaissances préalables en programmation. On reconnaît bien là nos objectifs. Dans un contexte de programmation par objet, une difficulté tient à l'aspect statique des classes. Or dans un atelier d'expert, il va bien falloir créer les classes des objets qui seront instanciés, et ceci dynamiquement. Razavi propose d'utiliser le concept de Modèles Objets Dynamiques développé par [RIE 00], par ailleurs très proche de ce que cet auteur avait pu développer pour ses propres applications. Les MOD sont basés sur la notion de Type Objet [JOH 97]. L'idée de base est d'associer une classe normale à une instance d'une autre classe qui sert de prototype (on parle aussi de méta-objet) en ce sens qu'elle contient les informations nécessaires à l'évolution dynamique de la première classe. Les MOD adoptent ce schéma Type Objet sur les entités manipulées par l'application et sur leurs attributs. Razavi propose d'étendre ce concept aux primitives et à leur ordonnancement car le schéma de base des MOD n'en prévoit pas l'usage. Type Objet est donc utilisé trois fois, d'où l'appellation To3 proposée par cet auteur [RAZ 00]. Notre approche, confrontée au même problème, converge également vers ce schéma. En effet, les instances de la classe `Moule` apportent des compléments dynamiques, sous la forme de la description des attributs et de la fourniture des comportements, à la classe `Agent` dont les instances, les agents réels de la simulation, contiennent les valeurs des attributs. Et les comportements sont décrits par un enchaînement de primitives adaptées au métier. Le `Moule` joue donc pour nous le rôle de prototype et comme dans le modèle de Razavi, un élément fondamental est que les primitives du moule ont librement accès aux instances de la classe `Agent` et à la valeur de leurs attributs.

Une des difficulté de la programmation par composants est d'organiser et de gérer le flot d'exécution de ces derniers pour obtenir le comportement souhaité. Pour notre part, nous avons délibérément figé la séquence de nos primitives en un enchaînement linéaire. Dans ce domaine, [MAN 01] ouvre des perspectives intéressantes avec la notion de micro-workflow. L'idée est de gérer les flux de tâches non plus grâce à un modèle global monolithique mais à l'aide, encore, de composants indépendants chargés de tâches spécifiques. Trois grands types de composants forment le noyau de ce système. Des composants d'exécution traitent chaque procédure : exécution du code et gestion des données correspondantes (entrées/sorties), qui peuvent être typées et ne se limitent pas nécessairement à un seul argument. Des composants de synchronisation séparent le comment du quand. Ils reposent sur des préconditions bâties sur le modèle ECA (Evénement-Condition-Action). Des composants de procédé contrôlent l'enchaînement des procédures : séquences linéaires, appels aux primitives du domaine, conditions, répétitions, itérations, branchements et jonctions. Certains de ces composants jouent donc typiquement le rôle de nos primitives de contrôle. D'autres composants, moins directement impliqués dans l'organisation du flux, permettent le suivi, la mémorisation, l'intervention utilisateur, ou encore la distribution sur plates-formes multiples du workflow. Un des gros intérêts de l'approche est sa modularité : on se limite aux composants utiles à son application.

5.2. *Des langages de spécifications aux langages d'experts*

Dans un travail portant sur la spécification, le prototypage et la vérification des SMA, [HIL 00] donne une vision synthétique des langages de spécification les plus utilisés. Il se limite aux langages formels car il s'agit pour lui d'aboutir au codage. Ceci exclut donc OMT ou UML par exemple, et il sépare :

- *Les langages orientés modèles* qui décomposent les systèmes en ensembles, fonctions et relations. Un de leur intérêt est de s'appuyer sur la logique mathématique pour exprimer les contraintes. Ils sont donc bien adaptés à la spécification d'objets manipulant des données.
- *Les langages algébriques* qui reposent sur la notion de type abstrait de données et comme leur nom l'indique sur les notations algébriques permettant de décrire équations et contraintes.
- *Les langages basés sur la logique temporelle* qui s'attachent à énoncer des prédicats ou des invariants à l'aide d'opérateurs temporels qui s'ajoutent à ceux de la logique classique. Ils sont adaptés aux systèmes réactifs c'est-à-dire ceux qui entretiennent une interaction avec leur environnement.
- *Les langages graphiques* basés sur la notion d'état/transition, qui s'adressent à des systèmes réactifs qui possèdent un nombre fini d'états et un comportement qui spécifie les transitions entre ces états. Le formalisme le plus utilisé est celui des réseaux de Pétri suivi sans doute par celui des statecharts [HAR 87].

Pour spécifier des comportements réactifs, [HIL 00] propose une approche multi-formalismes combinant les avantages structurants des langages orientés modèles et la richesse des langages graphiques à états finis. On a alors tous les éléments pour aller de la spécification à la validation de toute une gamme de SMA, en passant par leur prototypage et leur développement opérationnel.

Mais force est de constater que la sémantique de ces langages est souvent difficile. Les langages graphiques comme les réseaux de Pétri sont a priori plus abordables mais pas toujours très lisibles : il est difficile de déduire un comportement d'un simple regard aux schémas, et ces derniers deviennent vite inextricables si les comportements se compliquent. C'est d'ailleurs pourquoi Hilaire propose d'utiliser les statecharts dont il montre qu'ils sont plus intuitifs que les réseaux de Pétri et surtout plus compacts.

Pour pallier cette difficulté, une autre voie de recherche, déjà vieille d'une bonne dizaine d'années, est celle de la méta-modélisation. L'idée, dont on trouvera un aperçu dans [REV 01] par exemple, est de fournir à un expert non informaticien un formalisme métier dans lequel il décrit son modèle conceptuel. C'est le méta-modèle, formé par exemple d'opérateurs et de variables pour des applications en mathématiques. Par bien des aspects la solution que nous proposons s'en rapproche donc beaucoup. Mais ce formalisme est lui-même basé sur un méta-formalisme indépendant du domaine d'expertise, et pour l'essentiel guidé par les notions d'entité et de relation. Le système doit ensuite automatiquement générer le code exécutable et l'interface utilisateur. L'intérêt est que le générateur de code, dont le principe peut-être très différent selon les approches, peut lui aussi être en grande partie indépendant du domaine. Mais toute la difficulté est de manipuler des concepts suffisamment puissants pour ne pas retomber dans la complexité des langages classiques et suffisamment non ambiguës pour en permettre le codage.

La notion de langage d'experts (ou de formalisme métier) qui se dégage de tous ces travaux nous paraît fondamentale. Nous en retenons qu'il n'a pas vocation à l'universalité. Il est développé pour un domaine d'application défini, ce qui doit à la fois simplifier sa conception et le rendre compréhensible par l'utilisateur final. Un langage d'experts doit manipuler les objets du domaine. Ce sont pour nous les 25 primitives, mais également les attributs et les agents, ainsi que tout ce qui permet de définir une simulation, voire un ensemble d'expériences simulatoires. Pour satisfaire nos objectifs, il doit être à la fois un langage de spécification (qui décrit le modèle de façon lisible pour un expert biologiste, indépendamment de la plate-forme d'exécution) et un langage d'implémentation (qui permet de créer le code exécutable). A la lumière de ce qui se fait actuellement, on peut cependant penser que l'imagination est encore de mise pour élaborer un tel langage, mais qu'un langage informatique de type classique, même s'il se veut un peu plus proche du langage naturel comme ce que nous avons cherché à faire avec ATOLL, semble devoir rapidement montrer ses limites. A l'image de ce que propose Hilaire plusieurs sémantiques seront sans doute à combiner, et nous pensons en particulier à :

- *Des sémantiques algébriques*, en particulier pour l'écriture des primitives de calcul. Car tout utilisateur, même non informaticien, en garde quelques notions héritées de ses cours de mathématiques. Il est même possible de combiner données numériques et symboliques (des connaissances) comme le proposent [PAG 99]. Un des enjeux de ces langages et l'expression des contraintes. Car il serait bien utile que les primitives encapsulent certaines informations comme le domaine de variation des variables et des paramètres : certaines valeurs peuvent en effet être interdites.

- *Des sémantiques graphiques* plus adaptées aux aspects dynamiques. On pense bien sûr aux transitions entre agents-rôles pour écrire le cycle de vie des espèces, mais il faut également décrire la séquence des primitives composant une tâche particulière, ou encore la séquence d'appel des agents telle que la gère l'ordonnanceur. Une difficulté sera sans doute de représenter les interactions entre agents, un élément important pour la compréhension des modèles.

- *Un contexte hypertextuel* pour améliorer la lisibilité, et relier les différents niveaux de description et d'utilisation d'un modèle. On songe bien sûr à XML, et c'est avec cet état d'esprit qu'ATOLL a été scindé en zones par mots-clefs.

Dans notre domaine des simulations en écologie, [LOR 99] déjà cités ont ouvert la voie avec le langage Wesp-DL. Bien que trop proche de C++ pour être utilisable par un non informaticien et trop limité dans les types de modèles qu'il permet de décrire, il associe en effet un diagramme pour les changements de rôle des agents (les transitions), et un langage formel, incluant intelligemment SQL pour la manipulation des données, pour les comportements. Pour [RAZ 01], l'association entre MOD, micro-workflow, et un langage d'experts dérivé par exemple de celui des tableurs devrait fournir une base conceptuelle puissante pour la conception d'environnements de développement accessibles aux experts d'un domaine.

Dans le domaine des systèmes à compartiments, le logiciel précurseur que fut Stella™ et qui fit de nombreux émules comme Simulink® de Matlab® représente à nos yeux l'archétype de ce que doit être un outil de modélisation orienté utilisateur : il offre une programmation très accessible, d'ailleurs ici encore par composants, joint à une traduction formelle rigoureuse du modèle sous la forme du système d'équations différentielles associé. Notre approche entend faire un pas dans cette direction dans le domaine de la modélisation individu-centrée, en proposant une architecture « tout agent » adaptée à une décomposition des tâches en primitives, et en essayant de contribuer à la mise au point d'un langage permettant de décrire ces modèles. Et comme nous l'avons signalé en introduction, il ne s'agit que d'une première étape. Car derrière l'outil apportant une plus grande facilité d'écriture et une indispensable validation informatique du modèle (ce que « fait » la machine est bien ce qui est décrit dans la spécification du modèle), l'utilisateur doit également avoir un retour sur la qualité de ses simulations : combien de répliquats pour obtenir telle précision en sortie ? Quelle est l'influence du pas de temps, du choix du type de synchronisation ? De quel comportement atomiste des agents le système modélisé tire-t-il telle ou telle propriété ? Derrière les outils facilitant la construction, c'est

bien toute la méthodologie de conception, de validation, d'utilisation et d'interprétation de ce nouvel outil de modélisation qu'il faudra également développer.

Remerciements

Ce travail a été financé dans le cadre du programme CNRS Environnement, Vie et Sociétés « Biodiversitas » à travers l'action thématique du GIP HydrOsystèmes « Le poisson dans son milieu » (BV 2/98006). Mobidyc et son code sont librement disponibles (version bêta) dans un esprit de club d'utilisateurs. Il utilise VisualWorks 5i.4 de la société Cincom qui fonctionne sur pratiquement tous types d'ordinateurs et qui existe en version gratuite pour les utilisations non commerciales.

6. Bibliographie

[BOU 98] BOUSQUET F., BAKAM I., PROTON H., LE PAGE C., Cormas : Common-Pool Resources and Multi-agent Systems. Congrès IEA-98-AIE, *Lecture Notes in Artificial Intelligence* n°1416 : 805-814. Springer.

[DEA 79] DEANGELIS D.L., COX D.K., COUTANT C.C., Cannibalism and size dispersal in yong-of-the-year largemouth bass : experiment and model. *Ecological Modelling*, 8 (1979) 133-148.

[DRO 92] DROGOUL A., CORBARA B., FRESNEAU D., Applying EthoModeling to Social Organization in Ants. In *Biology and Evolution of Social Insects*, Leuven University Press, Leuven, pp. 375-383, 1992.

[EPS 96] EPSTEIN J.M., AXTELL R. L., *Growing artificial societies. Social science from the Bottom Up*. Cambridge, MIT Press, 1996.

[FER 95] FERBER J., *Les systèmes multi-agents. Vers une intelligence collective*, Paris, InterEditions, 1995.

[GAR 70] GARDNER M., The fantastic combinations of John Conway's new Solitaire Game "Life". *Scientific American*, 23 (4): 120-123.

[GUE 98] GUESSOUM Z., DIMA: Une plate-forme multi-agents en Smalltalk. Revue Objet, 3(4): 393-410, 1998

[GUT 97] GUTKNECHT O., FERBER J., {MadKit}: Organizing heterogeneity with groups in a platform for multiple multi-agent systems, Rapport Interne LIRMM,1997.

[HAR 87] HAREL D., Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8 (3) 231-274.

[HAR 00] HARROUET F., Oris : s'immerger par le langage pour le prototypage d'univers virtuels à base d'entités autonomes. Thèse de doctorat, Université de Bretagne Occidentale (Brest), 2000.

[HIL 00] HILAIRE V., Vers une approche de spécification, de prototypage et de vérification de systèmes multi-agents. Thèse de doctorat de L'Université de Franche-Comté. Spécialité Informatique. Janvier 2000.

[HOG 90] HOGEWEG P., HESPER B., Individual-oriented modelling in ecology. *Math. Comput. Modelling*, 13 (6) 83-90.

[HOU 98] HOUSSIN D., ATOLL : un langage de description de modèles dédié à la dynamique des peuplements. Rapport de DEA, Université Paris VI, 1998.

[JOH 97] JOHNSON R., WOOLF B., Type Object. Pp. 47-66 In MARTIN R., RIEHLE D., BUSCHMANN F., *Pattern Languages of Program Design.* Addison-Wesley, 1997.

[KRE 98] KREFT J-U., BOOTH G., WIMPENNY J.W.T., BacSim, a simulator for individual-based modelling of bacterial colony growth, *Microbiology* (1998) 144, 3275–3287

[LIU 98]. LIU J., ASHTON P.S., FORMOSAIC : an individual-based spatially explicit model for simulating forest dynamics in landscape mosaics. *Ecological Modelling* 106 (1998) 177-200.

[LOR 98] LOREK H., SONNENSCHEIN M., Object-oriented support for modelling and simulation of individual-oriented ecological models. *Ecological Modelling*, 108 (1998) 77-96.

[LOR 99] LOREK H., SONNENSCHEIN M., Modelling and simulation software to support individual-based ecological modelling. *Ecological Modelling*, 115 (1999) 199-216.

[MAN 01] Manolescu D-A., Micro-workflow. A workflow architecture supporting compositional object-oriented software development. PHD-Thesis, Univ. Illinois. http://micro-workflow.com

[MAR 97] MARCENAC P., Modélisation de systèmes complexes par agents. *Technique et Science Informatiques*, 16 (8): 1013-1038.

[MIN 96] MINAR H., BURKHART R., LANGTON C., ASKENAZI M., The swarm simulation system: a toolkit for building multiagent simulations. Santa Fe Institute Working Paper n° 96-06-042.

[NAR 93] NARDI B.A., *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, 1993.

[PAG 99] PAGE M., GENSEL J., BOUDIS M., AMIA : an environment for knowledge-based discrete-time simulation, *Actes AAAI Spring Symposium on Hybrid systems and AI*, Stanford University (CA US), (22-24 mars) 1999.

[RAZ 00] RAZAVI R., Type Cube: Foundation for an Architectural Style aimed at Building Adaptive and Flow-Independent Software. *Position paper to OOPSLA'2000 workshop on Best-practices in Business Rule Design and Implementation*. 15-19 October 2000, Minneapolis, USA. http://AdaptiveModeling.com

[RAZ 01] RAZAVI R., Outils pour les langages d'experts – Adaptation, *Refactoring*, Réflexion. Thèse de doctorat de l'Université Paris VI, Spécialité Informatique, option IA. novembre 2001. http://www-poleia.lip6.fr/~razavi/xps/phd

[REV 01] REVAULT N., YODER J. W. Adaptive Object-Models and Metamodeling Techniques. In Ecoop'01 Workshop Reader, Ákos Frohner (ed), LNCS, 2001, Springer-Verlag -- Report of workshop at Ecoop'01, University Eötvös Loránd, Budapest, Hungary. http://www-poleia.lip6.fr/~revault/publications.html.

[REZ 96] RESNICK M., Beyond the Centralized Mindset. *Journal of the Learning Sciences* 5 (1), 1-22.

[RIE 00] RIEHLE D., TILMAN M., JOHNSON R., « Dynamic Object Model » *Proc. of the 2000 Conference on Pattern Languages of Programs (PLoP 2000), Washington University* Tech. Rep. N° wucs-00-29. http://www.riehle.org/papers/2000/plop-2000-dom.html

[SHI 01] SHIN Y.-J., CURY P., Exploring fish community dynamics through size-dependent trophic interactions using a spatialized individual-based model. *Aquatic Living Resources.* 14 (2001) 65-80.

[WIL 87] WILSON S.W., Classifier Systems and the Animat Problem. *Machine Learning*, 2: 199-228.

[YOO 99] YOO M.-J., Objets et Agents pour systèmes d'information et de simulation. Une approche componentielle pour la modélisation d'agents coopératifs et leur validation. Thèse de doctorat, Université Paris 6, 1999.

Article reçu le 8 février 2001
Version révisée le 20 août 2001
Rédacteur responsable : Zahia Guessoum

David Houssin *est ingénieur en informatique. Après un DEA en intelligence artificielle, il a été responsable de développement dans une grande maison française d'édition de jeux vidéo, et occupe aujourd'hui un poste identique dans une entreprise européenne de technologies embarquées.*

Stefan Bornhofen *est mathématicien et informaticien, diplômé de l'Université de Mainz en Allemagne. Il se passionne pour les jeux informatico-mathématiques dédiés à la simulation de comportement d'objets physique. Il travaille actuellement en France en tant que développeur d'applications logicielles à l'IUT de La Rochelle.*

Sami Souissi *est maître de conférence à la station marine de Wimereux, Université de Lille 1. Docteur en biométrie et doté d'une double compétence statistiques / modélisation, son domaine d'étude est l'écologie numérique et l'étude des systèmes complexes, plus particulièrement en dynamique des populations marines.*

Vincent Ginot *est docteur en biométrie, ingénieur de recherche à l'unité de statistiques spatiales INRA d'Avignon. Son thème de recherche actuel porte sur les modèles individus-centrés, tant sur le plan de leur construction que sur la méthodologie de leur usage.*