SFERES : Un framework pour la conception de systèmes multi-agents adaptatifs

Samuel Landau* — **Stéphane Doncieux****
Alexis Drogoul* — **Jean-Arcady Meyer****

LIP6
Pôle IA – thème OASIS
8, rue du Capitaine Scott
75 015 Paris

{landau, doncieux, drogoul, jameyer}@poleia.lip6.fr
* *équipe Miriad*
** *équipe AnimatLab*

RÉSUMÉ. *Nous présentons les principaux aspects de SFERES, un framework d'évolution artificielle et de simulation multi-agent permettant de faciliter l'intégration de l'apprentissage par algorithme évolutionniste dans la conception de systèmes multi-agent adaptatifs. Dans une première partie de l'article, nous présentons brièvement les concepts et les techniques des différents algorithmes évolutionnistes existants. La seconde partie est consacrée à la présentation du framework, de ses principales composantes, de sa structure, des choix de conception et de ses possibilités d'extensions. Nous concluons en mentionnant les applications qui bénéficient déjà de l'environnement et des facilités de SFERES et en évoquant ses perspectives de développement.*

ABSTRACT. *We present a framework for artificial evolution and multi-agent simulation for the design of adaptive multi-agent systems by means of evolutionnary algorithms. In the first part, the concepts and techniques of existing evolutionnary algorithms are introduced. The second part describes the framework and its main components, the general structure of SFERES, the design directions and some extensions. We conclude with applications taking advantage of the environment and facilities of SFERES together with some development perspectives.*

MOTS-CLÉS : *framework, apprentissage multi-agent, évolution artificielle, simulation multi-agent*

KEYWORDS: *framework, multi-agent learning, artificial evolution, multi-agent simulation*

1. Introduction

L'une des problématiques majeures de la recherche actuelle en intelligence artificielle distribuée (IAD) est l'introduction de techniques d'apprentissage automatique dans les systèmes multi-agents (SMA), dans le but d'obtenir des SMA adaptatifs, c'est-à-dire des systèmes où les agents apprennent à se comporter, à interagir ou à s'organiser pour améliorer leurs performances collectives dans la réalisation d'une tâche. En raison de la complexité croissante des problèmes abordés par l'IAD, et des difficultés de conception qui en découlent, cette voie semble extrêmement prometteuse. Cependant elle se heurte encore à deux principaux obstacles [WEI 96]. Le premier obstacle, que nous avons décrit dans [DRO 98], est le problème du choix de la bonne technique d'apprentissage, qui nécessite de pouvoir évaluer sa pertinence par rapport à la tâche et par rapport au niveau (individuel ou collectif) auquel elle est censée s'appliquer. Ce choix implique de pouvoir comparer simplement les performances, sur un même problème (ou, dans le cas de l'IAD, pour la même instance d'une tâche multi-agent) d'un certain nombre de techniques qui différent par leur formalisme de représentation et leurs algorithmes. Le second obstacle, méthodologique et empirique, réside dans le choix du protocole d'apprentissage à utiliser (qui apprend, à quel rythme, dans quel contexte, etc.). Ce choix nécessite de disposer d'un environnement d'expérimentation robuste dans lequel il est possible de changer ce protocole sans avoir à modifier la technique d'apprentissage utilisée. Dans les deux cas, conduire et comparer des expériences successives qui diffèrent, soit par le choix de la technique, soit par celui du protocole, sont des activités qui vont jouer un rôle capital dans la conception et le déploiement de SMA adaptatifs.

SFERES [1] est un framework et un environnement de développement qui a été conçu pour fournir au concepteur les abstractions et les outils de base nécessaires à la réalisation conjointe de ces deux activités. Il combine deux sous-parties génériques qui seront détaillées dans cet article : un outil dédié à une forme particulière d'apprentissage, l'évolution artificielle, et un simulateur multi-agent. Le couplage de ces deux frameworks permet au concepteur de s'affranchir des contraintes qui existent à l'heure actuelle dans la majorité des outils ou bibliothèques dédiées à l'apprentissage multi-agent, en lui donnant la possibilité d'évaluer la pertinence de plusieurs techniques pour un même problème ou celle de plusieurs protocoles expérimentaux ou architectures d'agents pour une même technique.

La famille des méthodes d'apprentissage est particulièrement vaste et il a été nécessaire de choisir, pour construire SFERES, un sous-ensemble de techniques qui soit à la fois adapté aux différentes formes possibles d'apprentissage multi-agent (individuel ou collectif par exemple) et suffisamment efficaces pour justifier leur utilisation [MIT 97]. Les algorithmes d'évolution artificielle ont été retenus car ils bénéficient d'un statut particulier : s'ils sont loin d'être toujours optimaux, mais ils sont par contre les plus génériques et se déclinent en une multitude de variantes – dont l'une pourra

1. SFERES is a Framework for Encouraging Research on Evolution and Simulation, *cf.* `http://miriad.lip6.fr/SFERES/`

être plus adaptée que les autres pour un cas donné. C'est pour cette raison que l'apprentissage par évolution artificielle est, depuis quelques années, de plus en plus utilisé en IAD [WEI 96].

Dans la première partie de l'article, nous présenterons brièvement les concepts et les techniques des différents algorithmes évolutionnistes existants. Nous insisterons particulièrement, d'une part, sur leurs éléments communs, qui nous ont servi de base pour la conception du framework et, d'autre part, sur la très grande diversité de leurs applications, notamment en IAD. La seconde partie sera consacrée à la présentation de la structure générale de SFERES, des choix conceptuels qui ont sous-tendu notre démarche et de ses possibilités d'extensions. Nous mentionnerons ensuite un exemple simple ainsi que les applications qui bénéficient déjà de l'environnement et des facilités de SFERES avant de conclure sur les perspectives de développement du framework.

2. Introduction aux algorithmes évolutionnistes

L'appellation générique *algorithme évolutionniste* désigne des systèmes informatiques de résolution de problèmes [BRE 62, HOL 62] qui s'inspirent des mécanismes adaptatifs de l'évolution des espèces animales [DAR 59]. Différents algorithmes ont été proposés, les principaux ayant été conçus presque simultanément et indépendamment dans les années 1960 : les algorithmes génétiques, les stratégies évolutionnistes, la programmation évolutionniste et, plus récemment, la programmation génétique.

2.1. *Principe des algorithmes évolutionnistes*

Le principe général des algorithmes évolutionnistes est de tester en parallèle différentes solutions potentielles, appelées par la suite *individus*, à un problème posé, puis de retenir les plus efficaces et, à partir de ces solutions, d'en générer de nouvelles en les combinant de façon à améliorer progressivement leurs performances. La base conceptuelle commune aux algorithmes évolutionnistes réside donc en la simulation de l'évolution de générations successives de codes génétiques ou *génomes*, qui composent les individus, via des processus de *sélection* et de *reproduction*. Une première génération d'individus est tirée au sort. Chaque individu est testé et une note - valeur sélective ou *fitness* - lui est attribuée en conséquence. Seuls les individus à forte valeur sélective pourront se "reproduire", exploitant ainsi l'information disponible sur le degré d'adaptation de l'individu. Une seconde génération est crée en appliquant des opérateurs génétiques sur les structures des individus parents, comme des *recombinaisons* - échange de matériel génétique entre individus - ou des *mutations* - transformation aléatoire d'une partie du génome -. Ces transformations fournissent des heuristiques générales pour l'exploration des solutions possibles car ils apportent de la nouveauté dans le système. Chaque individu est de nouveau testé et le processus est reconduit de génération en génération jusqu'à l'obtention d'individus performants pour la tâche donnée. La figure 1 illustre le principe de fonctionnement d'un algorithme évolution-

```
t := 0
initialiser_population P(t)
évaluer P(t)
tant que non critère d'arrêt faire
        t := t + 1
        P'(t) := sélectionner_parents(P(t))
        croiser(P'(t))
        muter(P'(t))
        évaluer(P'(t))
        P(t+1) := sélectionner_survivants(P(t),P'(t))
```

Figure 1. *Principe de fonctionnement d'un algorithme évolutionniste*

niste. Bien que simpliste d'un point de vue biologique, ces algorithmes sont suffisamment performants pour fournir des mécanismes de recherche adaptative robustes et puissants.

De nombreuses variantes d'algorithmes évolutionnistes existent. Elles différent principalement sur la stratégie employée pour sélectionner les individus à conserver, sur le codage des solutions ainsi que sur les opérateurs génétiques utilisés.

Les algorithmes génétiques [HOL 75, DE 75, GOL 89] travaillent sur un codage de type chaîne binaire, alors que les *stratégies évolutionnistes* [REC 73, SCH 75] utilisent un vecteur de flottants. La *programmation évolutionniste* [FOG 66, FOG 92] permet de faire évoluer des machines à états finis et la *programmation génétique* [KOZ 92] travaille directement sur des programmes informatiques sous forme d'arbres de type S-Expressions.

Pour plus de détails sur les algorithmes évolutionnistes et des comparaisons des mérites de chacune des approches, consulter par exemple [HEI 98, BEA 93, MIT 99, WHI 94, BÄC 93].

	algorithme génétique	stratégie évolutionniste	programmation évolutionniste	programmation génétique
individu	chaîne de bit	tableau	quelconque	arbre
sélection	probabiliste[a]	élitiste[b]	élitiste	probabiliste
croisement	oui	non	non	oui
mutation	oui	oui	oui	non

a. la probabilité d'être sélectionné est fonction (croissante) de la fitness
b. la sélection se fait sur la valeur de la fitness : les « meilleurs » sont choisis

Figure 2. *Caractéristiques des principaux algorithmes évolutionnistes*

La figure 2 récapitule les caractéristiques de chacun des principaux algorithmes évolutionnistes – dans leur version « orthodoxe ». Beaucoup d'échanges ont eu lieu

entre ces techniques. SFERES permet de faciliter la conception des approches hybrides et d'évaluer la pertinence de chacune.

2.2. *Justification de la création du framework*

2.2.1. *Une méthode générique d'optimisation/d'apprentissage*

La sélection naturelle serait un mécanisme de création d'ordre à partir du désordre, qui repose sur la capacité de générer des variations de façon aléatoire sur lesquelles une sélection va filtrer les solutions trouvées – les plus adaptées. Ce mécanisme pourrait opèrer simultanément à plusieurs niveaux, de celui des constituants les plus intimes des cellules[KUP 00] à celui des organismes vivants dans leur ensemble[DAR 59].

Comme les variations sont aléatoires, les méthodes d'apprentissage par évolution artificielle ne nécessitent pas d'introduire de finalité donc de connaissance sur le problème à résoudre dans les opérateurs de recherche. Cette propriété en fait une bonne alternative lorsqu'aucune méthode déterministe d'optimisation n'est utilisable ou connue. Pour accélérer la recherche, il demeure cependant possible d'informer l'algorithme en biaisant les opérateurs de variation, selon ce qu'on sait du problème, mais la généricité des opérateurs est alors perdue. Notons qu'il faut tout de même le plus souvent introduire de la connaissance dans le mécanisme de sélection – même si ce n'est pas toujours nécessaire. Ce peu d'hypothèses faites sur les populations de solutions à trouver a pour conséquence de faire que les algorithmes évolutionnistes peuvent souvent être appliqués là où d'autres méthodes classiques d'optimisation (recuit simulé, remontée de gradient, ...) sont inutilisables ou trop peu efficaces, car elles nécessitent certaines propriétés de l'espace de recherche pour converger rapidement (voire pour converger tout court). La remontée de gradient, par exemple, impose l'utilisation de fonctions continûment dérivables.

Les aspects parallèle (plusieurs solutions sont recherchées de façon simultanée) et aléatoire (variations produites par les opérateurs) de la recherche permettent d'éviter les minima locaux du paysage de l'espace de recherche, par la combinaison de solutions éloignées ou par une variation suffisamment grande. Cependant, la généricité et la souplesse d'utilisation ont parfois pour conséquence le fait que la convergence n'est pas assurée. Si le temps de calcul est trop limité, on obtient des solutions satisfaisantes mais pas toujours optimales.

L'apprentissage multi-agent par évolution artificielle apporte également une réponse au problème du *credit assignment*. Ce problème se pose lors de l'apprentissage par plusieurs agents d'un système : lesquels faut-il récompenser ou punir, et dans quelles proportions, au vu de ce qu'a produit le système ? Une réponse possible avec l'évolution artificielle est de choisir un type d'individu de l'algorithme évolutionniste qui représente les caractéristiques du *groupe* d'agents que l'on veut faire évoluer [HAY 95], et non pas d'un agent pris individuellement – auquel cas il faudrait par exemple autant d'individus à évaluer qu'il y a de clones d'agents dans le système.

2.2.2. *Grande diversité d'applications*

Du point de vue de l'optimisation, un algorithme évolutionniste est une méthode stochastique ne requérant que des valeurs de la fonction à optimiser – on dit qu'il est d'ordre 0. La palette des applications est dès lors très large, puisque l'algorithme est applicable sans que le problème étudié satisfasse des hypothèses très spécifiques (continuité de la fonction, dérivabilité de la fonction,...). Il y a de ce fait plus d'applications aux algorithmes évolutionnistes qu'à toute autre méthode d'optimisation, car il suffit de disposer d'une fonction calculable, sans propriétés particulières.

Les algorithmes évolutionnistes ont été utilisés d'abord comme méthode d'optimisation numérique [DE 75, GOL 89, BÄC 93], puis le champ des applications s'est très diversifié [GRE 85a] :

– optimisation d'horaires [COL 90],

– optimisation de plans [DAV 85, HIL 87, FAN 94],

– conception de circuits électroniques, optimisation de gestion de flux,...

– et plus généralement, résolutions de problèmes NP-complets [GRE 85b, DE 89].

Les algorithmes évolutionnistes ont aussi abondamment été utilisés comme méthode d'apprentissage (systèmes de prises de décision [HOL 75], systèmes multi-agents [FOG 95], robots [NOL 00],...), notamment dans le sillon tracé par la Vie Artificielle, pour la conception des animats, animaux artificiels adaptatifs [WIL 87, WIL 90, MEY 90] ou la conception multi-agent :

– proie(s)-prédateurs, l'application sans doutes la plus représentée : [GRE 92, HAY 94, LUK 96, FLO 98, ...],

– coordination, coopération [SEN 96, ITO 95, BUL 96],

– fourragement [BEN 96],

– apprentissage d'un protocole de communication [WER 91],

– génération de plans [AND 95],

– prédiction et filtrage d'information, [CET 95, MOU 96],

– conception d'une équipe de football [AND 99, LAN 99] pour la RoboCup [2],

2.2.3. *Grande diversité d'implémentations*

À chaque problème que l'on désire résoudre à l'aide d'algorithmes évolutionnistes, il reste donc beaucoup de choix à faire : la façon de poser le problème – via le *codage génétique* et le choix de la fonction de *fitness* – ou la façon d'explorer l'espace des solutions – via les *opérateurs génétiques* et les choix de *sélection*. C'est par ces choix que l'on va pouvoir informer l'algorithme et éventuellement introduire un biais dans la recherche de solution pour accélérer un calcul particulier. On est donc souvent amené

2. *cf.* `http://www.robocup.org`, [KIT 95]

à expérimenter différentes techniques pour un même problème et c'est la motivation principale pour l'élaboration du framework décrit ici.

2.2.4. *Insuffisance des bibliothèques d'algorithmes évolutionnistes existantes*

Dans les bibliothèques d'algorithmes évolutionnistes [WHI 89, WAL 00, ...], l'évaluation d'un individu est composée uniquement du calcul ponctuel de la fonction d'évaluation, aucun autre élément n'est lié à l'évaluation d'un individu. Ce mécanisme, parfait pour l'optimisation de fonctions, n'est pas adapté à l'évolution du comportement d'agents pour les raisons suivantes :

– Un agent interagit avec son environnement et ces interactions doivent être considérées dans leur aspect temporel. Il serait réducteur de ne considérer qu'un instant au cours de la vie de l'agent pour attribuer une note de performances.

– L'évaluation du comportement des agents nécessite un simulateur (ou une interface vers des agents réels), ainsi que des systèmes de contrôle. Ces éléments sont indépendants d'une fonction d'évaluation particulière. La fonction d'évaluation, elle, en dépend, mais il est tout à fait possible d'étudier plusieurs fonctions d'évaluation sur un même simulateur.

3. Le framework

SFERES est constitué de deux parties. La première est la partie évolutionniste et représente les algorithmes évolutionnistes tels que présentés section 2.1. La seconde est la partie simulation et représente des simulations multi-agent. Les deux parties sont liées par l'évaluation des individus de l'algorithme évolutionniste dans la simulation. L'évaluation porte sur le comportement des agents simulés. Ce comportement est décrit plus ou moins directement selon les cas dans le génome sur lequel travaille l'évolution.

L'originalité de SFERES réside dans le couplage de la partie évolution artificielle et de la partie simulateur multi-agent pour évaluer les individus, ce que n'offrent pas la plupart des bibliothèques d'algorithmes évolutionnistes [WHI 89, WAL 00, ...]. Ce couplage est décrit section 3.3.

La figure 3 rassemble les classes abstraites du framework en les regroupant par parties. La relation entre les classes `Agent` et `Individual`, la classe `SimuFitness` et l'héritage de `Evaluator` par `Simulator` constituent le lien entre les deux parties du simulateur (*cf.* section 3.3).

3.1. *Partie évolutionniste*

La figure 4 rassemble les classes de la partie évolutionniste du framework. Ces classes peuvent être regroupées en deux parties : le moteur d'évolution et l'individu.

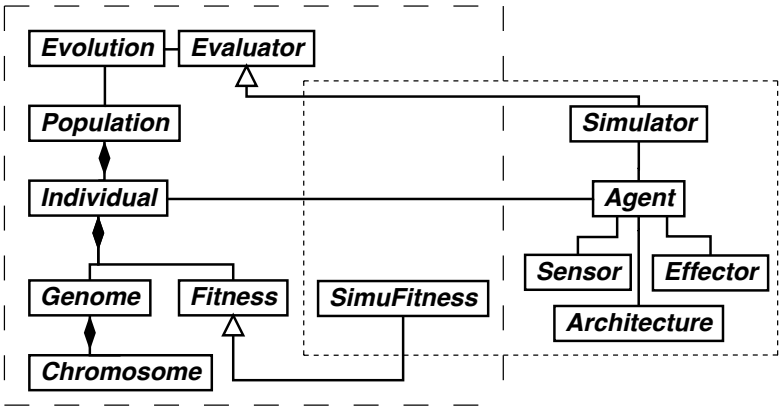

Figure 3. *Diagramme UML des classes de SFERES . Encadrée de tirets, la partie évolutionniste, et encadrée de pointillés, la partie simulation*

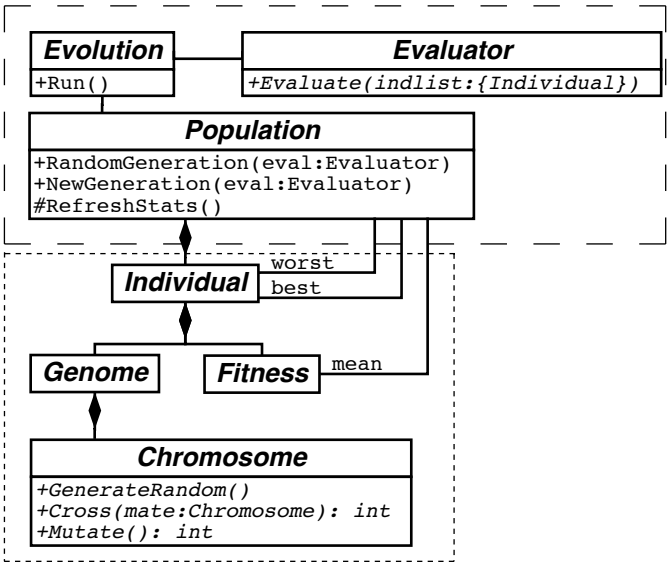

Figure 4. *Diagramme UML des classes de la partie évolutionniste de SFERES. Encadré de tirets, le moteur d'évolution, et encadré de pointillés, l'individu*

3.1.1. *Moteur d'évolution*

Le moteur d'évolution (cadre en tirets sur la figure 4), regroupe les classes gérant le déroulement de l'algorithme évolutionniste.

La class `Evolution` gère l'aspect technique des calculs. La répartition des calculs sur différentes machines, le choix des statistiques enregistrées pendant le déroulement de l'algorithme font partie de ses attributions.

La classe `Population` contient l'ensemble des individus et prend en charge la sélection de l'algorithme évolutionniste (*cf.* figure 1). La sélection est tout à fait indépendante du codage des solutions et des opérateurs génétiques. La classe `Population` a pour tâche de générer une nouvelle population à partir de la population courante et de mettre à jour les informations statistiques (performance du meilleur individu, moyenne...), dont la class `Evolution` gére la sauvegarde.

Au cours de la génération de nouveaux individus, la classe `Population` a besoin de les évaluer. Elle envoie alors un message à la classe `Evaluator`, qui les met à jour.

3.1.2. *Individu*

L'individu contient toutes les informations relatives à une solution potentielle : l'ensemble de structures et de paramètres composant la solution (classe `Genome`) ainsi que les performances de cette solution pour le problème posé (classe `Fitness`).

Un génome (classe `Genome`) est composé de chromosomes (classe `Chromosome`), qui contiennent le code génétique à proprement parler. La classe `Chromosome` contient également toutes les méthodes de manipulation de l'information génétique : principalement la génération aléatoire, le croisement et la mutation. Ces méthodes sont utilisées par la population au moment de la génération d'un nouvel individu. Leur implémentation est fortement liée à un codage génétique particulier.

La fonction d'évaluation, qui va définir la pression sélective exercée sur les individus, est contenue dans la classe `Fitness`. Du point de vue de `Population`, cette classe permet d'ordonner les individus en vue de leur sélection. Cette classe est le principal biais introduit par le concepteur pour diriger l'évolution vers les solutions souhaitées.

3.1.3. *Principe de fonctionnement*

Lors du lancement d'une expérience, `Evolution` exécute la méthode `Run()` qui fait appel à la méthode `NewGeneration()` de `Population`. Cette méthode génère de nouveaux individus et fait appel pour cela à `GenerateRandom()` lors de la première génération, puis à `Cross()` et `Mutate()` de la classe `Chromosome`. `Population` évalue alors la performance des individus nouvellement générés par un appel à la méthode `Evaluate()` de `Evaluator`. Le résultat de cette évaluation, stocké dans `Fitness`, servira à sélectionner les individus les plus performants lors du prochain `NewGeneration()`. À la fin de `NewGeneration()`, `Population` met à jour les différentes statistiques avec `RefreshStats()` avant de rendre la main à `Evolution`.

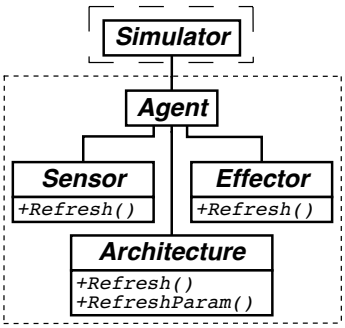

Figure 5. *Diagramme UML des classes de la partie simulation de SFERES. Encadré de tirets, le moteur de simulation, et encadré de pointillés, l'agent*

3.2. *Partie simulation*

La figure 5 rassemble les classes de la partie simulation du framework. Ici encore, on peut subdiviser les différents éléments en deux parties : le moteur de simulation et l'agent. Précisons d'emblée que, dans une implémentation particulière, le moteur de simulation et les agents, abstractions utiles au fonctionnement du framework, peuvent ne constituer qu'une interface avec un simulateur existant [GUT 97, GUE 98] ou des robots.

3.2.1. *Moteur de simulation*

Le moteur de simulation (classe `Simulator`) prend en charge la gestion de l'environnement des agents et le déroulement des simulations. `Simulator` est une classe abstraite dérivée d'`Evaluator`, permettant d'effectuer les évaluations des individus dans la simulation choisie. Il transmet les instances d'`Individual` aux instances d'`Agent` qui vont servir à évaluer leurs performances et il met à jour `SimuFitness`, classe dérivée de `Fitness` à laquelle on a ajouté des méthodes liées à l'évaluation du comportement d'un agent. Ces méthodes servent à évaluer le comportement d'un `Agent` au cours du temps, nous y reviendrons à la section 3.3).

3.2.2. *Agent*

Les interactions de l'`Agent` avec son environnement se font par le biais de capteurs (classe `Sensor`) et effecteurs (classe `Effector`), et l'information qui circule depuis les capteurs et vers les effecteurs est traitée par des `Architectures`. Ces classes forment à l'intérieur de l'agent un réseau dont la taille et la structure ne sont pas imposées et peuvent être commandées par un `Chromosome` (*cf.* section 3.3). Les `Chromosomes` peuvent aussi définir la structure interne de chacun de ces éléments.

3.2.3. *Principe de fonctionnement*

`Simulator` est appelé par le biais de sa méthode `Evaluate()` depuis `Population`. Les `Agents` ont accès à un `Individual` et peuvent utiliser son information génétique pour définir leur structure ou leurs paramètres. Au cours de l'évaluation, `Simulator` met à jour les instances de `SimuFitness`.

3.3. *Couplage entre les parties évolutionniste et simulation*

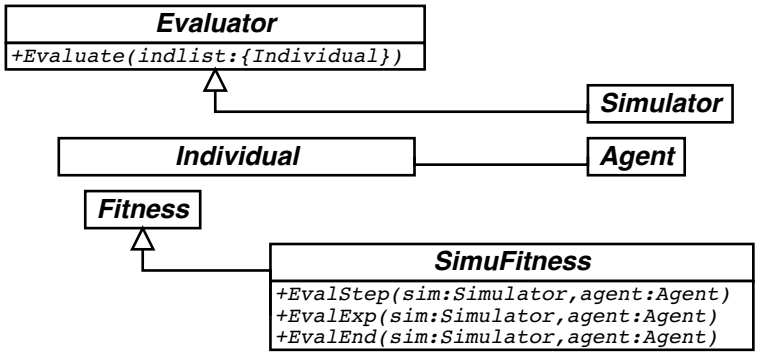

Figure 6. *Diagramme des classes du couplage entre les parties évolutionniste et simulation de SFERES*

La figure 6 représente la classe `SimuFitness`, les héritages et relations sur lesquels repose le couplage entre les parties évolutionniste et simulation de SFERES.

`Simulator`, la classe de base de la partie simulation du framework, hérite d'`Evaluator`. Elle dispose donc de la méthode permettant à `Population` de demander l'évaluation d'individus. Dans le cas de `Simulator`, cette méthode, `Evaluate()`, a la particularité de nécessiter une classe dérivée de `Fitness`, `SimuFitness`, pour effectuer ces évaluations.

`SimuFitness` a en effet des méthodes d'interface propres au calcul des performances d'un agent au sein de la simulation. Celles-ci sont nécessaires car à la différence du cas d'une optimisation de fonction, où le calcul de la valeur de la fonction au point défini par le génome est suffisant, lors de l'évolution de systèmes multi-agent, plusieurs critères sont à prendre en compte. Il ne suffit pas d'observer un agent à un instant donné pour avoir une idée de ses performances, il est nécessaire de pouvoir le suivre lors de ses interactions avec son environnement et les agents qui l'entourent. Le calcul de `SimuFitness` se fait donc pas à pas avec la méthode `EvalStep()`. Il peut ensuite être affiné à la fin d'une expérience et à la fin de l'évaluation avec les méthodes `EvalExp()` et `EvalEnd()`.

Enfin, `Agent` peut être lié au plus à un `Individual` qui est évalué. Ce lien permet de définir tout ou partie de l'agent à partir de l'information génétique contenue dans

`Genome`. L'agent, les capteurs, les effecteurs et les architectures d'agent peuvent ainsi être définis par les `Chromosomes`, aussi bien dans leur nombre, leurs connexions que dans leur structure interne. Aucune contrainte n'est imposée quant à la relation entre un `Agent` et un `Chromosome`. Les possibilités sont très variées, aussi une interface imposée restreindrait la portée générale du framework.

3.4. *Dérivations de classes*

L'implémentation de nouvelles expériences ou de variantes d'expériences existantes se fait simplement par dérivation des classes concernées tout en conservant les autres.

L'étude d'un nouvel algorithme de sélection ne nécessite que la réimplémentation de la classe `Population`. Il peut ensuite facilement être comparé aux stratégies utilisées précédemment en le faisant fonctionner sur des expériences déjà étudiées.

Le changement de simulation nécessite l'implémentation des classes liées à l'environnement simulé, c'est-à-dire `Simulator`, `Agent`, `Sensor` et `Effector`. Les `Architectures` utilisées pour contrôler les `Agents` n'étant pas connectées directement à l'environnement simulé, n'ont pas à être redéfinies.

La résolution d'un problème différent dans un environnement déjà implémenté ne nécessite que de changer la pression sélective appliquée aux individus. Il suffit donc d'implémenter une seule nouvelle classe : `SimuFitness`, toutes les autres classes étant identiques.

Le changement de l'architecture de l'agent requiert d'implémenter la nouvelle classe `Architecture`, ainsi que, généralement, la classe qui sert à la définition de sa structure ou de ses paramètres, à savoir le `Chromosome` associé.

De même, l'étude d'un nouveau codage génétique nécessite l'implémentation du nouveau `Chromosome` ainsi que, généralement, des classes qui vont l'utiliser (`Architecture` le plus souvent).

4. Illustration

Les agents considérés ici sont des robots à roues. Un premier type de robots, les proies, disposent uniquement de capteurs de proximité et sont capables de se mouvoir rapidement. Un deuxième type de robots, les prédateurs, disposent, en plus de capteurs de proximité, d'une caméra pour détecter les proies, mais ils se déplacent moins vite qu'elles. Les robots se déplacent dans une arène de taille donnée. Cette expérience est celle décrite dans [FLO 98]. On considère que lorsque un ou des prédateurs touchent une proie, ils la capturent.

4.1. *Apports comme outil de conception*

Dans un premier temps, pour se rendre compte de la complexité du problème, le concepteur peut implémenter manuellement le comportement des différents agents. Il n'utilise alors que la partie simulation de SFERES pour visualiser et évaluer les comportements en situation. Dans cette étape, pas d'évolution : les architectures des agents sont figées, entièrement conçues dès le départ et le concepteur peut suivre les évolutions de ces agents et acquérir différents points de repère qui lui permettront d'évaluer les performances de ses futurs agents adaptatifs.

Une deuxième étape dans l'étude de ce problème peut consister à améliorer des comportements pré-spécifiés en utilisant les techniques évolutionnistes. Dans ce cas, les agents sont spécifiés par le concepteur mais plusieurs paramètres ne sont pas fixés initialement ; ils seront soumis à évolution. De nombreuses techniques évolutionnistes existent pour l'optimisation de paramètres : *Algorithmes Génétiques*, *Evolution Strategies*... Le concepteur peut choisir parmi celles-ci et comparer leurs performances. Dans cette partie, seuls les élements que l'on souhaite optimiser par évolution doivent être modifiés : les architectures d'agent et éventuellement les capteurs ou effecteurs. Les agents évoluent dans le même environnement que lors de la première étape et il est tout à fait possible de faire s'affronter agents figés et agents adaptatifs, afin de comparer les stratégies préconçues aux stratégies acquises par apprentissage évolutionniste.

Pour étudier plus avant les systèmes adaptatifs, il est également possible de définir entièrement le comportement des agents par évolution (capteurs, effecteurs et architectures). Toutes les combinaisons sont ensuite possibles pendant l'évaluation d'un agent. Un prédateur peut ainsi être évalué en présence d'une proie définie manuellement, ou d'une proie adaptative, il peut être en présence de concurrents (ou d'alliés) prédateurs ayant le même comportement ou ayant un comportement différent mais tout de même acquis par évolution ou bien encore ayant un comportement figé.

Chaque agent peut avoir un type d'architecture différent des autres. Il est ainsi possible de faire s'affronter un prédateur contrôlé, par exemple, par un système de classeurs et une proie contrôlée par un réseau de neurones.

4.2. *Apprentissage agents/multi-agent*

L'interaction d'un agent avec son environnement ou avec d'autres agents passe par ses capteurs et effecteurs. Ceux-ci peuvent être modifiés ou remplacés indépendamment les uns des autres pour notamment tester différentes stratégies de communication entre agents. Les prédateurs peuvent ainsi communiquer, avec un même protocole, en utilisant différents supports (lumineux, sonore, gestuel...). Pour tester les différentes possibilités dans SFERES, il suffit de changer les différents capteurs/effecteurs impliqués (en dérivant `Sensor` et `Effector`). Dans les différents cas, l'architecture régissant le protocole de communication peut rester inchangée. Le choix des capteurs, effecteurs et de leurs paramètres peut aussi être fait automatiquement par évolution.

Le comportement de l'agent est principalement déterminé par ses architectures. Différentes possibilités peuvent être testées à capteurs et effecteurs identiques : contrôleurs par plans, cognitifs, réactifs... Chaque contrôleur peut être fixe ou bien conçu par évolution, que ce soit partiellement ou complètement. La classe à dériver est alors simplement `Architecture`, qui se chargera d'interpréter si besoin est les données et paramètres génétiques nécessaires à son fonctionnement.

Sur le problème proies-prédateurs, il est ainsi possible d'étudier les différentes stratégies de chasses des prédateurs : chasse solitaire en concurrence, ou bien chasse coordonnée en groupe. SFERES permet de comparer les deux approches et de tester différentes stratégies de coordination.

4.3. *Choix de la technique d'apprentissage évolutionniste*

Les techniques d'apprentissage évolutionnistes sont nombreuses. Certaines sont dédiées à l'optimisation de paramètres (*Algorithmes Génétiques*, *Evolution Strategies*), d'autres s'intéressent à l'évolution de structures exécutables (*Programmation Génétique*) et enfin de nombreuses techniques sont adaptées à un type de contrôleur particulier (*Codage Cellulaire* pour les réseaux de neurones, par exemple).

Chacune de ces techniques a un intérêt dans un contexte particulier. Certaines sont utilisables pour optimiser des comportements préparés à la main, alors que d'autres permettent une définition des architectures ex nihilo. SFERES permet de choisir la stratégie la plus adaptée en fonction des besoins du concepteur.

Pour passer d'une stratégie à une autre, la principale classe à modifier est la classe `Chromosome` – plus marginalement, `Genome`, qui est générique – et les opérateurs génétiques associés. Les classes interprétant l'information génétique (`Architecture` généralement) peuvent aussi être amenées à être modifiée si les méthodes sur lesquelles repose l'interprétation ne sont plus adaptées : c'est-à-dire si le support héréditaire (chromosomes en particuliers) n'a plus la même représentation (chaîne de bit, arbres par exemple). Il n'est pas nécessaire de modifier autre chose.

4.4. *Choix du protocole d'évaluation*

La méthode de sélection des individus qui vont s'affronter peut également être changée indépendamment des autres classes. Cette possibilité est très utile lors d'expériences de type coévolution proies-prédateurs. En effet, différentes méthodes existent pour choisir les individus qui vont être évalués simultanément et ceux qui vont être conservés d'une génération à l'autre. Un prédateur peut être évalué avec des prédateurs identiques génétiquement ou différents (il faut alors choisir lesquels : les meilleurs, choix aléatoire...). Il pourra de même être évalué face aux meilleurs proies de la génération courante, ou face à une groupe de proies choisies aléatoirement. Pour implémenter ces différentes méthodes de sélection, la seule classe à modifier est `Population` sur laquelle repose la sélection et le protocole d'évaluation.

Il reste encore à choisir le point le plus important dans les algorithmes évolutionnistes : la fonction d'évaluation. Comment doit-on récompenser les agents ? En fonction de leurs performances en groupe ou en fonction de leur habileté individuelle ? Différentes possibilités peuvent être testées en changeant la classe `SimuFitness`, toutes les autres classes étant identiques. Dans notre problématique proies-prédateurs, différentes évaluations sont utilisables, par exemple :

– individuelle : le nombre de proies capturées par un prédateur

– collective : temps moyen mis pour capturer une proie

– implicite : écosystème simulé, les prédateurs qui ne capturent pas de proie meurent au bout d'un certain temps

4.5. *Classes à dériver*

Pour ce qui est de la partie évolutionniste, les classes `Evolution`, `Population`, `Individual` et `Genome` sont génériques et seront réutilisées sans modifications la plupart du temps. Cependant, si les concepteurs désirent changer la politique de sélection des individus, il dériveront la classe `Population` et dériveront la méthode `NexGeneration()`, par exemple pour implémenter une politique de sélection élitiste plutôt qu'une politique probabiliste (*cf.* figure 2). Ceci est indépendant de la façon dont sont évalués les individus. D'autre part, selon les représentations génétiques que le concepteur désire utiliser, il devra dériver autant de classes de `Chromosome`. Celles-ci sont cependant indépendantes du problème étudié, et seront réutilisables sans modifications, par exemple pour implémenter une chaîne de bit d'algorithme génétique ou un arbre d'expression de la programmation génétique. Enfin, la classe `SimuFitness` est dépendante du problème étudié et de la façon dont est évalué l'individu auquel elle est associée. Elle doit donc être systématiquement dérivée.

Pour la partie simulation, la classe `Simulator` doit être dérivée car spécifique au problème étudié. Les méthodes de `Simulator` sur lesquelles repose le processus d'évaluation sont génériques, la dérivation consiste donc à gérer les lois de fonctionnement de l'environnement dans lequel sont plongés les agents. La classe `Agent` doit être dérivée pour tenir compte de ce qui est propre au simulateur avec lequel on l'utilise, le reste est générique (processus d'évaluation et interface avec les capteurs, effecteurs et architectures). Par exemple, si les proies et les prédateurs ne se différencient que par les capteurs et effecteurs qu'ils emploient, ils seront instances de la même classe dérivée de `Agent`. Enfin, les classes `Sensor` et `Effector` dépendent du simulateur et doivent être dérivées à chaque fois que l'on change de simulateur, et la classe `Architecture` doit être dérivée selon les architectures que le concepteur veut employer. L'architecture étant indépendante de l'agent, des capteurs, des effecteurs et du simulateur, chacune pourra être réutilisée sans modifications.

4.6. *Applications*

SFERES a ainsi été utilisé dans des simulations proies-prédateurs, dans la conception d'architectures de contrôle pour un pendule inversé, dans une simulation de dirigeable et une simulation multi-robots dans un environnement de bureau (liée au projet MICRobES [DRO 99]). Dans cette application, la partie simulation de la plate-forme est une interface pour un protocole courant, permettant de se connecter indifféremment à un simulateur existant ou directement aux robots. L'évolution peut alors se faire directement sur robots réels, comme dans [FLO 98] ou alors sur robots simulés, puis, une fois les calculs terminés, il est possible de passer sur robots réels.

5. Conclusion

Nous avons présenté le framework d'évolution artificielle et de simulation multi-agent SFERES, après avoir situé et expliqué l'intérêt d'un outil générique pour la conception de systèmes multi-agents adaptatifs.

Une plate-forme SFERES a été codée en C++ sous Linux, et un certain nombre d'applications ont déjà été implémentées. Nous sommes également en train d'enrichir les bibliothèques de la plate-forme d'un moteur de simulation physique, qui nous permettrait de concevoir de façon plus unifiée des simulations avec des robots, quel que soit par exemple leur mode de locomotion (robots à pattes, robots à roues,...). D'autre part, diverses techniques évolutionnistes ont aussi été implémentées : des approches classiques comme les algorithmes génétiques, les stratégies évolutionnistes et la programmation génétique, et aussi de nouvelles approches, notamment ATNoSFERES [LAN 01, PIC 01].

Le caractère générique du framework a déjà assuré un gain de temps de développement important dans la réalisation de certaines de ces applications, puisque plusieurs classes sont utilisées à l'identique dans le cadre d'applications différentes. La même classe de population est ainsi utilisé dans toutes les applications de type mono-agent et la classe chromosome couramment utilisée l'est dans des applications allant de l'optimisation de fonctions à la conception d'architecture de contrôle.

Les directions futures de développement de la plate-forme sont :

– l'ajout de bibliothèques de classes implémentant d'autres algorithmes évolutionnistes,

– l'interfaçage avec une plate-forme SMA complète

– l'ajout de bibliothèques de classes implémentant d'autres architectures de contrôle (par exemple des classeurs [HOL 75, GÉR 00], des contrôleurs flous [HOF 94]).

Remerciements

Nous tenons à remercier Jean-Pierre Briot, Zahia Guessoum, Jean-François Perrot, et les membres de l'équipe Framework pour leur soutien et les conseils qu'ils nous ont prodigués. Nous tenons également à remercier Agnès Guillot pour sa relecture avisée et les corrections qu'elle nous a suggérées.

6. Bibliographie

[AND 95] ANDRE D., « The Evolution of Agents that Build Mental Models and Create Simple Plans Using Genetic Programming », *Proceedings of the Sixth International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc., 1995, p. 248–255.

[AND 99] ANDRE D., TELLER A., « Evolving Team Darwin United », ASADA M., KITANO H., Eds., *RoboCup-98 : Robot Soccer World Cup II*, vol. 1604 de *LNCS*, Paris, France, juillet 1998 1999, Springer Verlag, p. 346–351.

[BÄC 93] BÄCK T., SCHWEFEL H.-P., « An Overview of Evolutionary Algorithms for Parameter Optimization », *Evolutionary Computation*, vol. 1, n° 1, 1993, p. 1–23.

[BEA 93] BEASLEY D., BULL D. R., MARTIN R. R., « An overview of genetic algorithms : Part 1, fundamentals », *University Computing*, vol. 15, n° 2, 1993, p. 58–69.

[BEN 96] BENNETT F. H., « Emergence of a Multi-Agent Architecture and New Tactics For the Ant Colony Foraging Problem Using Genetic Programming », MAES P., MATARIC M. J., MEYER J.-A., POLLACK J., WILSON S. W., Eds., *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior : From animals to animats 4*, Cape Code, USA, 9-13 septembre 1996, MIT Press, p. 430–439.

[BRE 62] BREMERMANN H. J., « Optimization through Evolution and Recombination », YOVITS, JACOBI, GOLDSTEIN, Eds., *Self organizing Systems*, Pergamon Press, Oxford, 1962.

[BUL 96] BULL L., FOGARTY T. C., « Evolution in cooperative multiagent environments. », SEN S., Ed., *Working Notes for the AAAI Symposium on Adaptation, Co-evolution and Learning in Multiagent Systems*, Stanford University, CA, mars 1996, p. 22–27.

[CET 95] CETNAROWICZ K., « The application of evolution process in multi-agent world (MAW) to the prediction system », LESSER V., Ed., *Proceedings of the First International Conference on Multi–Agent Systems*, MIT Press, 1995.

[COL 90] COLORNI A., DORIGO M., MANIEZZO V., « Genetic algorithms and highly constrained problems : The timetable case », GOOS G., HARTMANIS J., Eds., *Parallel Problem Solving from Nature*, Springer-Verlag, 1990, p. 55–59.

[DAR 59] DARWIN C., *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*, 1859.

[DAV 85] DAVIS L., « Job shop scheduling with genetic algorithms », Grefenstette [GRE 85a], p. 136–140.

[DE 75] DE JONG K. A., « An analysis of the behavior of a class of genetic adaptive systems », PhD thesis, Dept. of Computer and Communication Sciences, University of Michigan, 1975.

[DE 89] DE JONG K. A., SPEARS W. M., « Using genetic algorithms to solve NP-complete problems », SCHAFFER J. D., Ed., *Proceedings of the Third International Conference on*

Genetic Algorithms, Morgan Kaufmann, june 1989, p. 124–132.

[DRO 98] DROGOUL A., ZUCKER J.-D., « Methodological Issues for Designing Multi-Agent Systems with Machine Learning Techniques : Capitalizing Experiences from the RoboCup Challenge », rapport n° LIP6 1998/041, octobre 1998, LIP6.

[DRO 99] DROGOUL A., PICAULT S., « MICRobES : vers des collectivités de robots socialement situés », *Actes des 7èmes Journées Francophones Intelligence Artificielle Distribuée et Systèmes Multi-Agents (JFIADSMA'99)*, Hermès, 1999.

[FAN 94] FANG H.-L., « Genetic Algorithms in Timetabling and Scheduling », PhD thesis, University of Edinburgh, 1994.

[FLO 98] FLOREANO D., NOLFI S., MONDADA F., « Competitive Co-Evolutionnary Robotics : From Theory to Practice », PFEIFER R., BLUMBERG B., MEYER J.-A., WILSON S. W., Eds., *From Animals to Animats : Proc. of the Fifth International Conference on Simulation of Adaptive Behavior*, Cambridge, MA, 1998, MIT Press.

[FOG 66] FOGEL L. J., OWENS A. J., WALSH M. J., *Artificial Intelligence through Simulated Evolution*, John Wiley, New York, 1966.

[FOG 92] FOGEL D. B., « Evolving artificial intelligence », PhD thesis, University of California, San Diego, 1992.

[FOG 95] FOGARTY T. C., CARSE L. B. B., « Evolving Multi-Agent Systems », WINTER G., PÉRIAUX J., GALAN M., CUESTA P., Eds., *Genetic Algorithms in Engineering and Computer Science*, Wiley and Sons, 1995.

[GÉR 00] GÉRARD P., STOLZMANN W., SIGAUD O., « YACS : a new Learning Classifier System using Anticipation », *Proc. of the Third International Workshop on Learning Classifier Systems*, LNCS / LNAI, Springer Verlag, 2000.

[GOL 89] GOLDBERG D. E., *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley, 1989.

[GRE 85a] GREFENSTETTE J. J., Ed., *Proceedings of the International Conference on Genetic Algorithms and their Applications*, San Mateo, 1985, Morgan Kaufmann.

[GRE 85b] GREFENSTETTE J. J., GOPAL R., ROSMAITA B., VAN GUCHT D., « Genetic algorithms for the traveling salesman problem », Grefenstette [GRE 85a], p. 60–168.

[GRE 92] GREFENSTETTE J. J., « The Evolution of Strategies for Multi-agent Environments », *Adaptive Behavior*, vol. 1, n° 1, 1992, p. 65–89.

[GUE 98] GUESSOUM Z., « DIMA : Une plate-forme multi-agents en Smalltalk », *Revue Objet*, vol. 3, n° 4, 1998, p. 393–410.

[GUT 97] GUTKNECHT O., FERBER J., « MadKit : Organizing heterogeneity with groups in a platform for multiple multi-agent systems », rapport n° RR97188, december 1997, LIRMM.

[HAY 94] HAYNES T., WAINWRIGHT R., SEN S., « Evolving cooperation strategies », rapport n° UTULSA-MCS-94-10, dec 1994, The University of Tulsa.

[HAY 95] HAYNES T., SEN S., SCHOENEFELD D., WAINWRIGHT R., « Evolving a Team », SIEGEL E. V., KOZA J. R., Eds., *Working Notes for the AAAI Symposium on Genetic Programming*, Cambridge, MA, november 1995, AAAI.

[HEI 98] HEITKÖTTER J., BEASLEY D., « The Hitch-Hiker's Guide to Evolutionnary Computation (FAQ for comp.ai.genetic) », december 1998.

[HIL 87] HILLIARD M., LIEPINS G., PALMER M., MORROW M., RICHARDSON J., « A classifier based system for discovering scheduling heuristics », GREFENSTETTE J. J., Ed., *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, San Mateo, 1987, Morgan Kaufmann, p. 231–235.

[HOF 94] HOFFMANN F., PFISTER G., « Evolutionary Design of a Fuzzy Knowledge Base for a Mobile Robot », *International Journal of Approximate Reasoning*, vol. 11, n° 1, 1994.

[HOL 62] HOLLAND J. H., « Outline for a logical theory of adaptive systems », *Journal of the ACM*, vol. 9, n° 3, 1962, p. 297–314.

[HOL 75] HOLLAND J. H., *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*, Ann Arbor : University of Michigan Press, 1975.

[ITO 95] ITO A., YANO H., « The Emergence of Cooperation in a Society of Autonomous Agents », LESSER V., Ed., *Proceedings of the First International Conference on Multi–Agent Systems*, MIT Press, 1995, p. 201–208.

[KIT 95] KITANO H., ASADA M., KUNIYOSHI Y., NODA I., AWA E.-I. O., « RoboCup : The Robot World Cup Initiative », *Proc. of IJCAI–95 Workshop*, Menlo Park, CA, 1995, AAAI Press, p. 41–47.

[KOZ 92] KOZA J. R., *Genetic Programming : On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, 1992.

[KUP 00] KUPIEC J.-J., SONIGO P., *Ni Dieu ni gène. Pour une autre théorie de l'hérédité*, Seuil, Paris, novembre 2000.

[LAN 99] LANDAU S., « Prise de décision pour une équipe de robots-footballeurs », Master's thesis, Université Paris VI, LIP6, Paris, septembre 1999.

[LAN 01] LANDAU S., PICAULT S., DROGOUL A., « ATNoSFERES : a Model for Evolutive Agent Behaviors », *Proceedings of the AISB'01 Symposium on Adaptive Agents and Multi-Agent Systems*, 2001.

[LUK 96] LUKE S., SPECTOR L., « Evolving Teamwork and Coordination with Genetic Programming », KOZA J. R., GOLDBERG D. E., FOGEL D. B., RIOLO R. L., Eds., *Genetic Programming 1996 : Proceedings of the First Annual Conference*, Cambridge, MA, 1996, The MIT Press, p. 150–156.

[MEY 90] MEYER J.-A., WILSON S. W., Eds., *From Animals to Animats : Proc. of the First International Conference on Simulation of Adaptive Behavior*, London, England, 1990, A Bradford Book, MIT Press.

[MIT 97] MITCHELL T., *Machine Learning*, McGraw Hill, 1997.

[MIT 99] MITCHELL M., TAYLOR C. E., « Evolutionary Computation : An Overview », *Annual Review of Ecology and Systematics*, vol. 20, 1999, p. 593–616.

[MOU 96] MOUKAS A., « Amalthaea : Information Discovery and Filtering using a Multiagent Evolving Ecosystem », *Proc. of the Conference on Practical Application of Intelligent Agents and Multi-Agent Technology*, London, UK, 1996.

[NOL 00] NOLFI S., FLOREANO D., *Evolutionary Robotics : The Biology, Intelligence, and Technology of Self-Organizing Machines*, MIT Press/Bradford Books, Cambridge, MA, 2000.

[PIC 01] PICAULT S., LANDAU S., « Ethogenetics and the Evolutionary Design of Agents Behaviors », CALLAOS N., ESQUIVEL S., BURGE J., Eds., *Proc. of the 5th World Multi-*

Conference on Systemics, Cybernetics and Informatics (SCI'01), vol. 3, july 2001, p. 528-533.

[REC 73] RECHENBERG I., *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*, Frommann–Holzboog, Stuttgart, 1973.

[SCH 75] SCHWEFEL H.-P., « Evolutionsstrategie und numerische Optimierung », Dr.-Ing. Thesis, Technical University of Berlin, Department of Process Engineering, 1975.

[SEN 96] SEN S., SEKARAN M., « Multiagent Coordination with Learning Classifier Systems », Weiss, Sen [WEI 96], p. 218–233.

[WAL 00] WALL M., « GALib », `http://lancet.mit.edu/ga/`, 2000.

[WEI 96] WEISS G., SEN S., Eds., *Adaptation and Learning in Multi–Agent Systems*, Lecture Notes in Artificial Intelligence, Springer Verlag, Berlin, 1996.

[WER 91] WERNER G. M., DYER M. G., « Evolution of communication in articial organisms », LANGTON C. G., TAYLOR C., FARMER J., RASMUSSEN S., Eds., *Articial Life II*, Reading, MA, 1991, Addison Wesley, p. 659–687.

[WHI 89] WHITLEY D., « the Genitor genetic algorithm », `http://www.cs.colostate.edu/~genitor/`, 1989.

[WHI 94] WHITLEY D., « A Genetic Algorithm Tutorial », *Statistics and Computing*, vol. 4, 1994, p. 65–85.

[WIL 87] WILSON S. W., « Classifier Systems and the Animat Problem », *Machine Learning*, vol. 2, n° 3, 1987, p. 199–228.

[WIL 90] WILSON S. W., « The Animat Path to AI », Meyer, Wilson [MEY 90], p. 15–21.