SFERES [1]
Un framework pour la conception de systèmes multi-agents adaptatifs
SFERES [1]
A framework for designing adaptive multi-agent systems

Samuel LANDAU [2], Stéphane DONCIEUX [3]
Alexis DROGOUL [2] et Jean-Arcady MEYER [3]

LIP6 – thème OASIS 8, rue du Capitaine Scott 75015 Paris
Email : {landau, doncieux, drogoul, jameyer}@poleia.lip6.fr
Web : http://www-poleia.lip6.fr/{~landau, ~doncieux, ~drogoul, ~jameyer}

RÉSUMÉ. *Nous présentons les principaux aspects d'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 la 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 et de ses deux principales composantes. La structure générale de SFERES, les choix de conception, et les possibilités d'extensions du framework y sont expliqués et illustrés. Nous concluons en mentionnant les applications qui bénéficient déjà de l'environnement et des facilités de SFERES, et en évoquant les perspectives de développement du framework.*

ABSTRACT. *We present the main aspects of a framework of artificial evolution and multi-agent simulation facilitating the integration of learning by evolutionnary algorithms in the design of adaptive multi-agent systems. In the first part of the article, we briefly present the concepts and the techniques of the various existing evolutionnary algorithms. The secund part is devoted to the presentation of the framework and its two main constituent. The general structure of SFERES, the design directions, and the extensions feasibilities of the framework are explained and illustrated there. We conclude by mentionning the applications that already take advantage from the environment and facilities of SFERES, and by evoking the development perspectives of the framework.*

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

KEY WORDS : *framework, multi-agent learning, artificial evolution, multi-agent simulation*

---

1. SFERES is a Framework for Encouraging Research on Evolution and Simulation, *http://miriad.lip6.fr/SFERES/*
2. équipe Miriad
3. équipe AnimatLab

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 des obstacles importants [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 diffèrent 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 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, comme on le verra, 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'agent pour une même technique.
La famille des méthodes d'apprentissage est particulièrement vaste et il nous a été nécessaire de choisir, pour construire *SFERES*, un premier sous-ensemble de techniques qui soit à la fois adapté aux différentes formes possibles d'apprentissage multi-agent (individuel, collectif, etc.) et suffisamment efficaces pour justifier leur utilisation [MIT 97]. La famille des algorithmes d'évolution artificielle, qui a été choisie, jouit d'un statut particulier : si les différents algorithmes qui la composent sont loin d'être toujours optimaux, ils sont par contre les plus génériques et se déclinent en une multitude de variantes – dont l'une pourra être plus adaptée que les autres pour un cas donné. L'apprentissage par évolution artificielle est ainsi, depuis quelques années, l'objet de beaucoup d'attentions en apprentissage automatique et se trouve être no-

tamment de plus en plus utilisé en IAD [WEI 96].
Dans la première partie de l'article, nous présentons donc brièvement les concepts et les techniques des différents algorithmes évolutionnistes existants. Nous insistons particulièrement, dans cette partie, sur, d'une part, leurs éléments communs (qui nous ont servi de base pour la conception du framework) et, d'autre part, la très grande diversité de leurs applications, notamment en IAD. La seconde partie est consacrée à la présentation du framework et de ses deux principales composantes. La structure générale de *SFERES*, les choix conceptuels qui ont sous-tendu notre démarche de conception, de même que ses possibilités d'extensions sont clairement présentés et expliqués. L'ensemble est illustré par des exemples simples. Nous concluons en mentionnant les applications qui bénéficient déjà de l'environnement et des facilités de *SFERES*, et en évoquant 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'appuient sur des modèles empruntés à quelques-uns des mécanismes connus de l'évolution [DAR 59] comme base de conception et d'implémentation. 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. Les grandes diversités des applications et implémentations des algorithmes évolutionnistes, ainsi que leur grande généricité en tant que technique d'optimisation ou d'apprentissage ont motivé la création du framework *SFERES*, dont la structure et le fonctionnement sont détaillés section 3.

2.1. *Principe des algorithmes évolutionnistes*

Le principe générale 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 astucieuse afin d'améliorer progressivement les performances. La base conceptuelle commune aux algorithmes évolutionnistes réside donc en la simulation de l'évolution de structures d'*individus* via des processus de *sélection*, de *mutation* et de *reproduction*. Ces processus dépendent des performances (*fitness*[4]) mesurées des structures d'individus dans un certain *environnement*. Plus précisément, les algorithmes évolutionnistes entretiennent une *population* de structures, qui évoluent suivant les règles de sélection et d'autres opérateurs, désignés sous le nom d'*opérateurs génétiques* – comme le *croisement* et la *mutation*. Ce sont lesdits

4. fitness : mesure de l'adaptation d'un individu à son environnement ; mesure de performance de l'individu dans des problèmes d'optimisation.

opérateurs qui sont les moteurs de la recherche dans l'algorithme. Chaque individu de la population se voit attribuer une mesure de sa fitness dans l'environnement. La reproduction concentre son attention sur les individus à haute fitness, exploitant ainsi l'information disponible du degré d'adaptation de l'individu. Le croisement et la mutation perturbent ces individus, fournissant des heuristiques générales pour l'exploration des solutions possibles représentées par chaque individu. Bien que simpliste d'un point de vue de biologiste, ces algorithmes sont suffisamment riches pour fournir des mécanismes de recherche adaptative robustes et puissants.

La fig. 1 illustre le principe de fonctionnement d'un algorithme évolutionniste. Pour

```
-- commencer avec une date initiale
t := 0
-- initialiser une population d'individus, usuellement de façon aléatoire
initialiser_population P(t)
-- évaluer la fitness de tous les individus de la population initiale
évaluer P(t)
-- boucler sur le critère d'arrêt (temps écoulé, fitness atteinte, etc.)
tant que non critère d'arrêt faire
    -- augmenter le compteur de temps
    t := t + 1
    -- sélectionner une sous-population pour générer la progéniture
    P'(t) := sélectionner_parents( P(t) )
    -- croiser les "gènes" des parents sélectionnés
    croiser( P'(t) )
    -- perturber stochastiquement la population générée
    muter( P'(t) )
    -- évaluer les fitness des nouveaux individus
    évaluer( P'(t) )
    -- sélectionner les survivants en se basant sur la fitness
    P(t+1) := sélectionner_survivants( P(t), P'(t) )
```

Figure 1. *Description d'un algorithme évolutionniste*

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, . . .].

2.1.1. *Algorithmes Génétiques*

Les Algorithmes Génétiques [HOL 75, De 75, GOL 89] se caractérisent par l'utilisation d'un génome (à l'origine une chaîne de *bit*) qui représente le codage de ce qui va être évalué (le *phénotype*). Le codage tient une place très importante, il joue un rôle central dans la connaissance que l'on peut introduire dans l'algorithme – pour accélérer la recherche de solutions. Classiquement, la sélection des survivants (cf. fig. 1) est une méthode où chaque individu a une probabilité proportionnelle à sa fitness d'être sélectionné ; les croisements (dans le cas de chaînes de *bit*) se font en échangeant des morceaux de génome et les mutations se font en inversant des *bit* lors d'un tirage aléatoire, suivant une probabilité généralement fixée au départ.

2.1.2. *Stratégie Évolutionnistes*

Les Stratégies Évolutionnistes [REC 73, SCH 75] opèrent très différemment. Les génomes sont composés d'un tableau de paramètres flottants et d'une matrice de covariances. Cette dernière sert à calculer les probabilités de mutation des éléments du tableau : les éléments du tableau sont mutés en se voyant ajouter une variable gaussienne de moyenne zéro et dont l'écart-type est calculé en fonction des paramètres de covariances qui sont eux aussi sujets à mutation. Les paramètres de covariance sont ajustés tout le long de l'algorithme : la mutation est adaptative et cette composante de chaque individu permet d'informer l'algorithme. Les stratégies évolutionnistes (ES, de l'allemand EvolutionsStrategie) n'utilisent généralement pas les croisements, mais ont plusieurs méthodes de sélection particulières, notamment : $(\mu, \lambda) - ES$ et $(\mu + \lambda) - ES$. Dans les deux cas μ parents génèrent λ descendants (avec $\lambda > \mu$), mais les méthodes différent sur la façon de sélectionner les μ individus pour l'étape suivante : dans $(\mu, \lambda) - ES$, ils sont choisis parmi les λ descendants, alors que dans $(\mu + \lambda) - ES$ ils sont choisis parmi les $\mu + \lambda$ individus, parents et descendants confondus.

2.1.3. *Programmation Évolutionniste*

La Programmation Évolutionniste [FOG 66, FOG 92] avait été conçue au départ pour faire évoluer des Machines à États Finis. La représentation du génome dépend du problème, il n'y a pas nécessairement de codage depuis une chaîne comme dans les Algorithmes Génétiques. Chaque individu génère un descendant, celui-ci est muté avec une probabilité qui peut décroître tout le long de l'algorithme, ou qui peut évoluer avec l'individu. Il n'y a pas de croisement et la sélection se fait en prenant les individus les plus adaptés parmi les parents et les descendants, réduisant ainsi la taille de la population suivante à la taille de la population initiale.

2.1.4. *Programmation Génétique*

Enfin, la Programmation Génétique [KOZ 92] a cela de particulier qu'elle fait évoluer des programmes informatiques. Les génomes de l'algorithme original était des structures d'arbres représentant des S-expressions semblables au langage LISP : un croisement consiste alors à échanger des sous-arbres. L'évalutation de l'individu se fait directement, en évaluant le programme qu'il code. Souvent, la Programmation Génétique n'emploie pas de mutation. Des variantes de cet algorithme imposent le respect d'une grammaire context-free [KOD 98] aux programmes-individus, permettant ainsi de circonscrire l'espace de recherche et d'imaginer des opérateurs de mutation consistants.

La fig. 2 récapitule les caractéristiques de chacun de ces algorithmes dans leur première version : la plupart ont évolué depuis. Par exemple, les dernières versions des stratégies évolutionnistes ont un croisement.

	algorithme génétique	stratégie évolutionniste	programmation évolutionniste	programmation génétique
création	~1960	~1960	~1960	~1980
individu	chaîne binaire interprétée	tableau de paramètres et matrice des corrélations	quelconque	arbre représentant un programme
sélection	probabiliste[a]	élitiste[b] schémas (μ, λ), $(\mu + \lambda)$	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*

2.2. *Justification de la création du framework*

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

Les aspects parallèle et aléatoire de la recherche de solution permettent d'utiliser avec succès les algorithmes évolutionnistes là où d'autre méthodes classiques d'optimisation (recuit simulé, remontée de gradient,...) sont inutilisables ou trop peu efficaces, car elles nécessitent une certaine forme du paysage de l'espace de recherche pour converger rapidement. La recherche simultanée de plusieurs solutions permet de se sortir de minima locaux, ou de combiner des solutions prometteuses pour en obtenir d'encore meilleures et la partie aléatoire de recherche d'une solution permet d'élargir l'exploration de l'espace, ce qui augmente la probabilité de se trouver à proximité d'un optimum.
Contrairement aux autres méthodes, les algorithmes évolutionnistes ne nécessitent pas de connaissances sur le problème à résoudre, ce qui en fait une bonne alternative lorsqu'aucune méthode déterministe d'optimisation n'est utilisable ou connue. On peut cependant plus ou moins informer l'algorithme, selon ce qu'on sait du problème, pour accélérer la recherche.
Le prix à payer pour la généricité et la souplesse d'utilisation est que la convergence n'est pas assurée, même si en pratique l'on trouve souvent des solutions satisfaisantes (mais pas toujours optimales).

2.2.2. *Grande diversité d'applications*

D'un point de vue 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 rapidement servi comme méthode d'apprentissage (prises de décision [HOL 75], systèmes multi-agents [FOG 95], robots [NOL 00],. . .), notamment dans le sillon tracé par la Vie Artificielle. C'est en particulier très prisé pour la conception des animats [WIL 87, WIL 90, MEY 90, . . .] ainsi que comme méthode d'apprentissage / 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[5],

Enfin, l'apprentissage multi-agent par évolution artificielle apporte aussi une réponse au problème du *credit assignment*. Par exemple, il est possible de choisir un type d'individu de l'algorithme évolutionniste qui représente les caractéristiques du *groupe* d'agents (et non pas de chaque agent) que l'on veut faire évoluer [HAY 95].

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 introduire un biais dans

5. *http : //www.robocup.org*, [KIT 95]

la recherche de solution pour accélérer le calcul. On est donc souvent amené à expérimenter différentes solutions pour un problème donné. La question de faciliter leur élaboration et leur réutilisation se pose donc très tôt avec une grande acuité : c'est la motivation principale pour l'élaboration du framework décrit ici.

3. Le framework

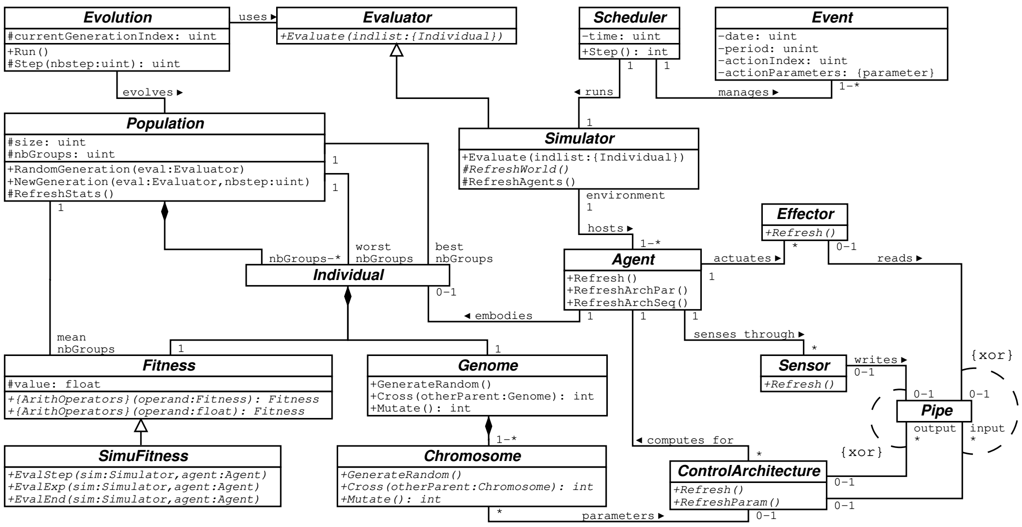

Figure 3. *Diagramme des classes de SFERES(cf. fig 8 pour un diagramme pleine page)*

Le framework est constitué de deux parties. La première est la sous-structure d'algorithmes évolutionnistes et permet d'implémenter les algorithmes évolutionnistes tels que présentés section 2.1. La seconde est la sous-structure de simulation multi-agent et permet d'implémenter des simulations dont les caractéristiques seront détaillées dans la section 3.2. Les deux parties sont liées par la mécanique de l'évaluation des individus de l'algorithme évolutionniste dans l'environnement de la simulation. L'évaluation porte sur les comportements des agents, chaque individu décrivant plus ou moins directement le comportement d'un agent de la simulation.

La figure 3 rassemble l'ensemble des classes abstraites du framework et décrit leurs associations.

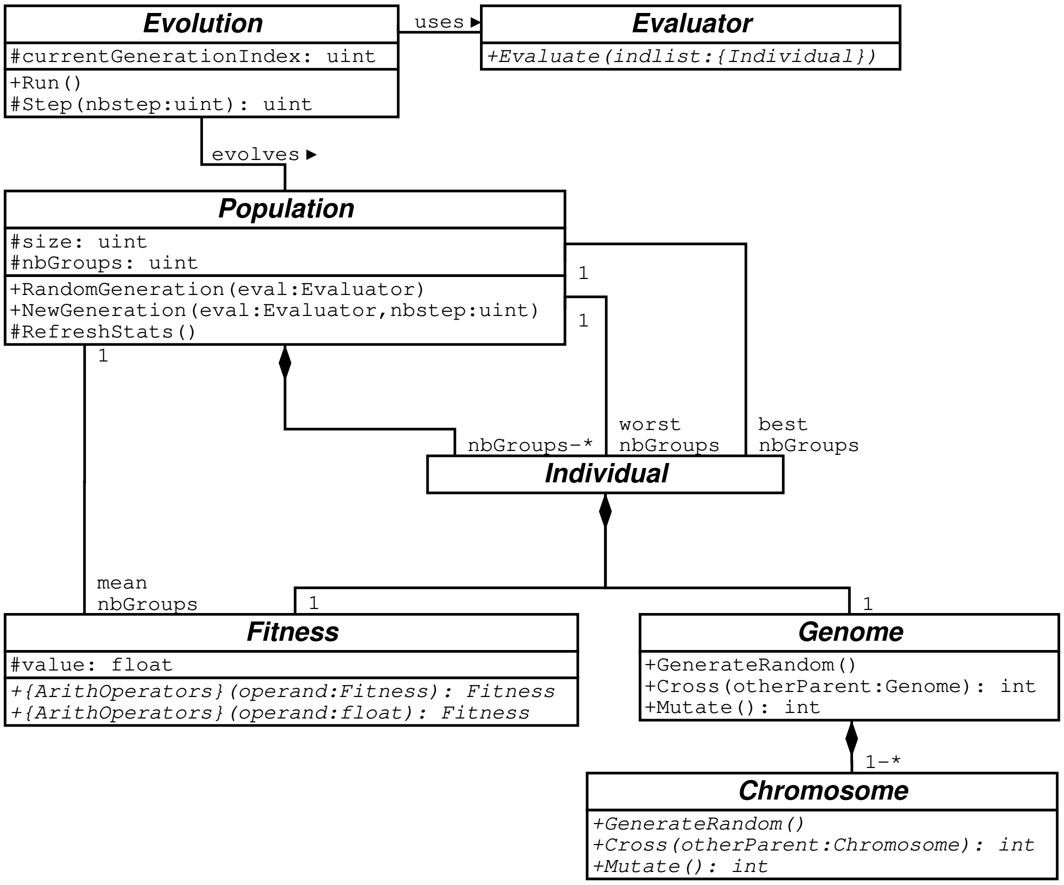

Figure 4. *Diagramme des classes de la partie évolutionniste de SFERES*

3.1. *La structure du framework : algorithme évolutionniste*

Nous allons détailler le processus qui nous a conduit aux choix structurels détaillés sur la figure 4. Si l'on fait abstraction de l'évaluation des individus, rappelons que les caractéristiques d'un algorithme évolutionniste donné (cf. fig. 2) reposent sur :

– le choix des structures "génétiques" des individus (nous dirons *génome*, même s'il peut n'y avoir aucune ressemblance avec le génome de la biologie), qui à leur tour influencent directement:
  – le choix des mutations,
  – le choix des croisements,

  qui ont le rôle d'opérateurs exploratoires,

– le choix des sélections, qui dépendent des fitness des individus et de la population, au sein de laquelle les sélections ont lieu. Les sélections ont le rôle d'opérateurs d'exploitation (de l'information de l'algorithme).

Nous allons présenter la hiérarchie des classes abstraites du framework pour l'évolution de façon que ces caractéristiques soient bien identifiées lors du processus de réification.

3.1.1. *Les classes abstraites pour l'évolution*

classes `Evolution` *et* `Evaluator`

Le framework devant permettre de résoudre différents problèmes en réutilisant le même algorithme évolutionniste, ou au contraire de tester différents algorithmes sur le même problème, il paraît judicieux d'abstraire le processus d'évaluation des individus. Les deux premières classes de la partie évolutionniste de *SFERES* sont donc une classe `Evolution` et une classe `Evaluator`. La classe `Evolution` gère tout le processus de l'algorithme évolutionniste et la classe `Evaluator` évalue un ou des individus (en exprimant leurs génotypes, puis d'une façon qui lui est propre, évalue leur qualité d'adéquation au problème posé et positionne leur fitness). La classe `Evolution` se sert de la classe `Evaluator` pour évaluer les individus qu'elle fait évoluer, ce processus d'évaluation étant indépendant du fonctionnement de l'algorithme.

classe `Individual`

L'évaluation n'est pas indépendante des données que `Evolution` manipule, les individus. Or chaque individu est caractérisé par un génome et une fitness. Il y a d'une part un lien fort entre `Evolution` et le génome – qui doit pouvoir être exploité/exprimé – et d'autre part un lien fort entre `Evolution` et la fitness – qui doit pouvoir être positionnée d'une façon très particulière, propre au problème. Il faut donc abstraire l'individu en une classe `Individual`, pour que l'on puisse choisir à l'utilisation une instance qui reflète ce couplage fort avec l'évaluation, tout en permettant à `Evolution` de manipuler indistinctement les individus. `Evolution` se servira de `Evaluator` en lui demandant d'évaluer une liste de `Individual`.

classes `Fitness` *et* `Genome`

L'abstraction de l'individu en une classe n'est pas non plus suffisante : il est tout à fait envisageable que l'on veuille résoudre un même problème en testant différents *codes génétiques*, d'autant plus que ce choix influence directement la dynamique de la recherche de solutions : un mauvais codage peut nuire en agrandissant trop l'espace de recherche, alors qu'un bon codage peut accélérer la recherche. Or le génotype est *a priori* structurellement indépendant de l'évaluation du phénotype, c'est d'ailleurs ce qui aura lieu le plus souvent : la fitness ne notera pas directement le génome. Cependant la possibilité reste de les lier plus avant, en concevant convenablement des classes dérivées. Par exemple, sur un codage génétique de type Programmation Génétique (cf. fig 2), on peut vouloir prendre en compte la taille de l'arbre généré, pour favoriser les

programmes–génomes les plus courts.
Nous abstrayons par conséquent les deux composantes de la classe `Individual`, en une classe `Fitness` et une classe `Genome`. Il faut que l'évaluateur sache interpréter et exploiter le génome et noter de façon correcte le phénotype induit : soulignons à nouveau le couplage fort entre la classe `Evaluator` et la classe `Fitness` d'une part et entre la classe `Evaluator` et la classe `Genome` d'autre part. L'évaluateur peut aussi ne pas se contenter d'exprimer le génome et de le laisser intact : lors des manipulations il peut aussi éventuellement le modifier, par exemple pour implémenter le Lamarckisme (transmission héréditaire des caractères acquis, ici par le biais du génome).
Par ailleurs, il y a aussi un lien entre l'algorithme et le génotype (cf. fig 1). En effet, lors de son exécution, l'algorithme doit pouvoir croiser des génome, ou les muter : ce sont les opérateurs d'exploration de l'algorithme ; la classe `Genotype` doit donc comporter ces méthodes.

classe `Chromosome`

Pour des raisons pratiques, la classe `Genome` est elle-même composée de chromosomes. La classe `Chromosome` va permettre de réutiliser des structures particulières pour les génomes, ce qui peut s'avérer très utile (codages différents de paramètres indépendants). Un cas particulier important étant celui de l'évaluation des individus à l'aide d'une simulation : les chromosomes vont faciliter l'apprentissage de comportements d'agents dotés de plusieurs architectures de contrôle, chaque chromosome contenant des paramètres pour au plus une architecture de contrôle.

classe `Population`

Les opérateurs d'exploitation aussi peuvent faire l'objet d'expériences, les algorithmes évolutionnistes se distinguant aussi sur leur façon d'exploiter les connaissances qu'ils construisent sur l'espace de recherche. La classe `Evolution` va donc déléguer à une autre classe le soin d'opérer les sélections. Comme celles-ci se font au sein de la population, c'est sur une classe `Population` que le choix se porte. La sélection dépend en effet de la structure interne de la population : la sélection peut par exemple exploiter une topologie particulière de l'ensemble des individus – la notion de proximité entre individus augmentant la probabilité de croisement.
Cette notion de topologie se voit considérablement enrichie si l'on introduit aussi la notion de groupes dans la population. Celle-ci peut se faire à peu de frais, il s'agit juste d'infléchir les méthodes de sélection selon que les groupes sont :

– hétérogènes (les individus appartenant à des groupes différents ne pouvant pas être croisés), ce qui permet par exemple d'implémenter un algorithme de coévolution,
– homogènes (les individus appartenant à des groupes différents pouvant être croisés), ce qui permet par exemple d'implémenter un algorithme évolutionniste à niches écologiques, dont on espère généralement que chaque niche va voir ses individus se spécialiser différemment des individus des autres niches (par dy-

namique sélectionnistes). Un exemple classique est l'algorithme génétique à populations migrantes proposés par De Jong (Deme GA, [De 75]).

À noter que, pour que la sélection puisse être indépendante de la façon dont est calculée la fitness, la classe `Fitness` doit être capable de donner une valeur commune représentative de l'adaptation de l'individu pour toutes ses classes dérivées. Un attribut flottant de `Fitness` va assurer cette fonction. En effet, la fitness peut être calculée et stockée sous une autre forme pendant l'évaluation, par exemple à l'aide d'un tableau, pour noter des caractères distincts de l'individu. La façon choisie de calculer l'attribut flottant en fonction de cette représentation interne (par exemple pour un tableau de nombre, il peut être obtenu par moyenne arithmétique ou géométrique) a une influence certaine sur la probabilité de l'individu d'être sélectionné.

Lors d'expériences avec des algorithmes évolutionnistes il est intéressant de garder une trace statistique des individus ayant évolué, pour éventuellement apporter des corrections à l'algorithme et aussi pour reconstituer le étapes du processus évolutif qui ont conduit à la solution. Ces statistiques sur les individus sont faites par la classe `Population` de façon générique : elle garde ainsi trace, à chaque génération, de l'individu le plus mal et le mieux adapté de chaque groupe, ainsi que de la fitness moyenne pour ce groupe.

Enfin, nous avons fait le choix de donner un rôle supplémentaire à la population. En plus d'être l'instance qui gère les sélections, la classe `Population` va être aussi capable d'avancer seule vers une nouvelle génération, ce qui consiste en croiser les individus sélectionnés de la sous-population et les muter après. Pour son propre fonctionnement l'instance de la classe `Evolution` n'a alors plus qu'à régulièrement lancer un message à l'instance de `Population` pour avancer d'une génération. À chaque lancer de message, l'instance de `Evolution` fournit en argument l'évaluateur (qui peut changer en cours d'évolution). Ce choix de délégation de tâche (qui devrait incomber à `Evolution`) permet de libérer `Evolution` de la gestion courante de l'algorithme et, en particulier, de lui permettre de distribuer des calculs sur plusieurs processus. Les algorithmes évolutionnistes étant très gourmands en ressources et souvent lents à converger – la convergence n'étant de plus pas assurée – inclure dans le framework la possibilité de distribuer les calculs peut s'avérer très utile. Par exemple, plusieurs processus esclaves instanciant une classe dérivée de `Evolution` qui leur fait attendre d'un processus maître les individus à faire évoluer localement pour un temps avant de les renvoyer, serait une autre implémentation, distribuée et élégante de Deme GA.

3.1.2. *Exemple de fonctionnement*

La figure 5 présente un exemple de déroulement simplifié d'un algorithme évolutionniste selon le framework de *SFERES* : au cours d'une génération, deux individus sont créés, croisés, mutés puis évalués ; l'un est sélectionné (i_k) et l'autre pas (i_j).

3.1.3. *Exemple de dérivation des classes*

Prenons comme exemple une Stratégie Évolutionniste (ES, cf. fig 2) distribuée.

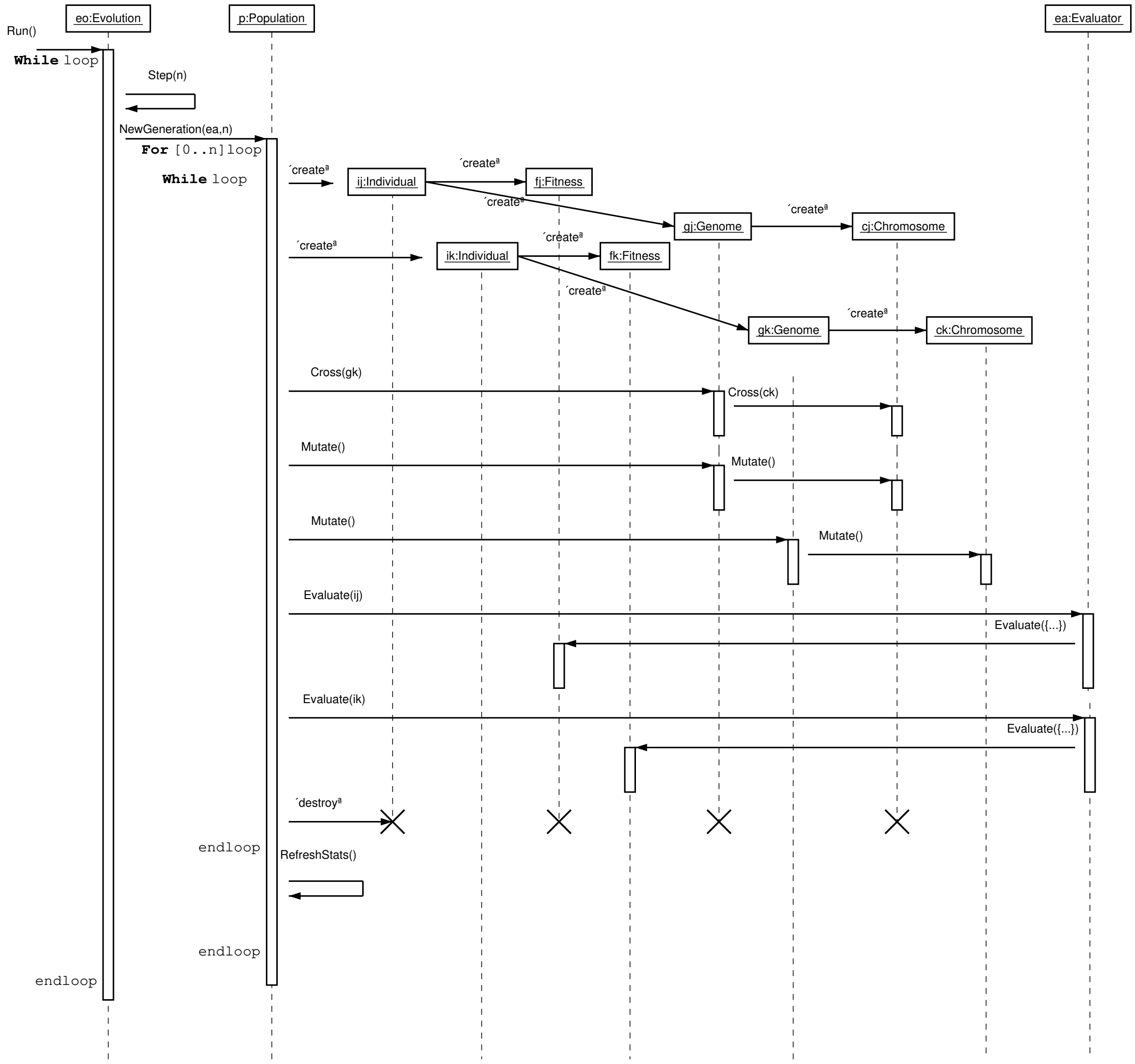

Figure 5. *Diagramme temporel illustrant la partie évolutionniste*

`MasterDistributedEvolution` *et* `SlaveDistributedEvolution`

Classes dérivées de `Evolution`. La première redéfinit la méthode `Run()` de façon à se mettre à l'écoute sur une interface de communication puis, lorsque les clients se sont enregistrés le calcul se poursuit normalement par la méthode générique de `Evolution` qui fait appel à `Step()` régulièrement. De façon symétrique, la seconde redéfinit `Run()` de façon à se connecter au serveur, puis le calcul se poursuit normalement.
Chaque classe redéfinit aussi `Step()` : le maître envoie à chaque esclave une liste

d'individus pour former sa population, puis attend les individus de la génération suivante, calculés sur chaque client ; le client reçoit sa population du serveur, la fait évoluer de nbstep générations et la renvoie. Le maître fait appel à `NewGeneration()` après avoir reçu les informations des esclaves et ceux-ci font appel à `NewGeneration()` entre la réception et le renvoi des individus.

`ESGenome`, `FloatArrayChrosome` *et* `CorrelationMatrixChromsome`

`ESGenome`, `FloatArrayChrosome` et `CorrelationMatrixChromsome` sont des classes dérivées de `Genome` et de `Chromosome`. `ESGenome` est composée de `FloatArrayChrosome` et de `CorrelationMatrixChromsome`. Sa méthode `Mutate()` est redéfinie de façon à employer les valeurs du second chromosome (`CorrelationMatrixChromsome`) pour les calculs de mutations des valeurs du premier chromosome ; les méthodes `Mutate()` des deux chromosomes sont surchargées aussi. La méthode `Cross()` est surchargée dans chacune des classes et ne fait rien.

`MasterDistributedPopulation` *et* `SlaveDistributedPopulation`

Classes dérivées de `Population`. On redéfinit `NewGeneration()` du maître pour qu'il ne fasse plus que des statistiques et celui des esclaves pour qu'ils n'en fassent pas (comme les individus sont redistribués sans cesse chez les clients, les statistiques de chacun d'eux sont de peu d'intérêt).

classe dérivée d'`Evaluator`

Elle n'a de contrainte que celle de pouvoir interpréter le tableau de flottants du génome d'ES.

3.2. *La structure du framework : simulation multi-agent*

L'originalité de *SFERES* comme framework d'évolution artificielle réside surtout en l'intégration d'un simulateur SMA pour évaluer les individus, ce que n'offrent pas la plupart des bibliothèques d'algorithmes génétiques [WHI 89, WAL 00, ...], lesquelles reposent sur une fonction d'objectif à atteindre (*"objective function"*). Après normalisation cette fonction d'objectif à atteindre n'est autre qu'une fitness explicite. Elle doit donc être totalement réécrite pour chaque problème, puisqu'elle doit prendre en charge à la fois l'expression du génome et son évaluation. Cette approche est certes adapté à des résolutions de problèmes d'optimisation de fonctions explicites, mais pas à des problèmes liés à une simulation (ce qui englobe une bonne part des problèmes d'apprentissage), car l'on ne bénéficie pas de structures permettant de décrire plus rapidement l'environnement et les agents d'un problème, et surtout de le faire une fois pour toutes indépendamment de l'évaluation du comportement des agents.
La simulation multi-agent dans *SFERES* sert à évaluer les individus, au travers des comportements des agents qui leur sont associés. Pour chaque individu, le simulateur

utilise le génome afin d'obtenir tout ou partie de la description du comportement de l'agent. L'utilisation peut être plus ou moins directe, elle peut consister en un simple paramétrage ou nécessiter plusieurs étapes d'interprétation ou de décodage : tout dépend de la structure de contrôle choisie. Celle-ci se décompose en sous-structures, des architectures de contrôle que l'on peut assembler à loisir. Ces architectures de contrôle peuvent être de tout type : numériques (réseaux de neurones, classeurs), symboliques (réseaux sémantiques,), hybrides, etc. Les aspects importants de la simulation multi-agent dans *SFERES* sont que pour chaque agent il peut y avoir plusieurs, architectures de contrôle, plusieurs capteurs et plusieurs effecteurs, et surtout que ce sont des composants que l'on peut ajouter, retirer ou reconfigurer tout le long de la simulation. Enfin, la simulation est à temps discret, ordonnancée par un échéancier.

3.2.1. *Les classes abstraites pour la simulation*

La structure pour la simulation est détaillée dans la figure 6.

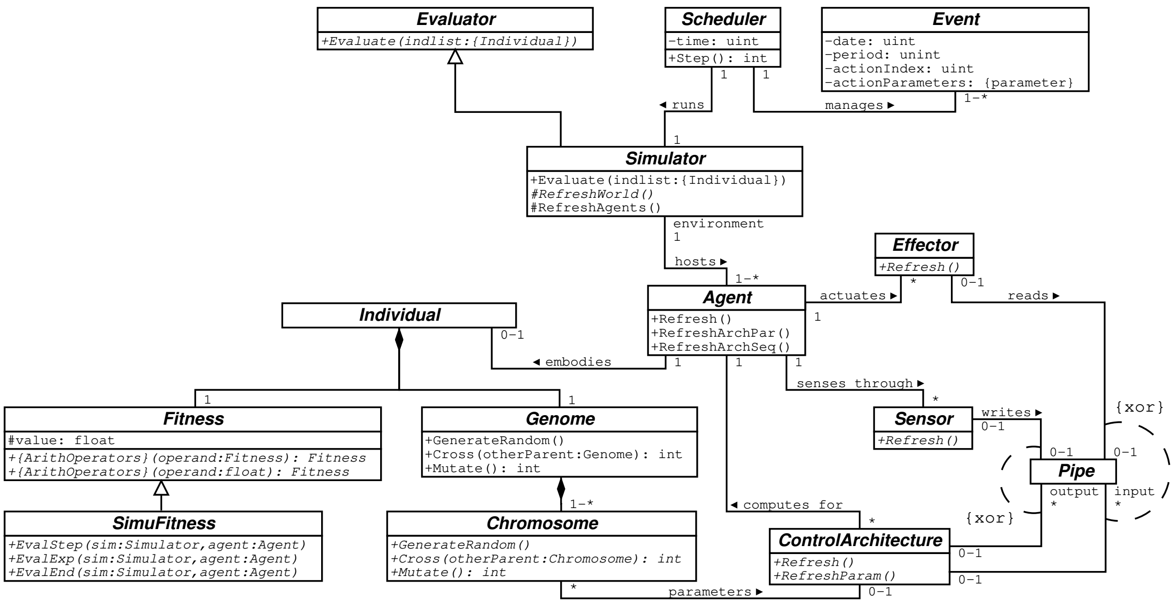

Figure 6. *Diagramme des classes de la partie simulation de SFERES*

classe `Simulator`

La classe de départ pour la simulation est la classe `Simulator`, dérivée de la classe `Evaluator`. C'est elle qui contient le modèle du monde à simuler : paramètres, lois de fonctionnement, etc. Son interface est constituée d'une méthode abstraite `RefreshWorld()` et de deux méthodes génériques `RefreshAgents()` et `Evaluate(Individual)`. Les deux premières s'occupent respectivement de rafraîchir les paramètres de l'environnement et des agents et la dernière surcharge la méthode abstraite de `Evaluator`. Leurs caractéristiques sont :

- `RefreshWorld()` : dépend de la simulation et est donc abstraite.
- `RefreshAgents()` : est générique grâce à l'abstraction de l'agent. La façon de faire le rafraîchissement d'un agent peut être toujours la même, même si ce rafraîchissement repose sur des caractéristiques propre à une simulation donnée.
- `Evaluate(Individual)` : est générique. Elle alterne appels à l'étape suivante de l'échéancier et diverses évaluations des fitness des agents.

Les classes dérivées de `Simulator` peuvent fournir des méthodes qui permettent d'agir sur les paramètres de l'environnement ou des agents ; celles-ci seront appelées par l'échéancier.

classes `Scheduler` *et* `Event`

`Scheduler` est la classe de l'échéancier qui, par appels successifs à la méthode générique `Step()`, fait avancer d'un pas de temps la simulation, pas de temps au cours duquel plusieurs actions peuvent être exécutées. L'échéancier gère une liste d'événements de type `Event`, chacun d'eux se déclenchant à une date et avec une période données. Un événement contient aussi l'indice de l'action du simulateur à exécuter et ses paramètres.
La fin d'une expérience, ou la fin de la simulation sont des événements particuliers qui ne sont pas liés à une méthode du simulateur. L'événement “fin d'une expérience” sert à évaluer en plusieurs fois un ou des individus, en leur faisant subir plusieurs tests. L'événement “fin de la simulation” sert juste à limiter la durée de la simulation.
Suivant l'échéancier, le simulateur peut ne pas être réinitialisé en début d'évaluation ou d'expérience par exemple. Il est ainsi possible de découper une évaluation en une suite d'expériences alors que l'évaluation totale se fait dans une simulation sans discontinuités, ou alors il est aussi possible de préserver le contexte de la simulation d'une évaluation à la suivante (contexte persistant).

classe `Agent`

L'agent est au centre de la simulation : il peut incarner ou pas un individu de l'algorithme évolutionniste. Si un agent est bien lié à un individu, son comportement est en partie ou complètement déterminé par le processus évolutif et, au cours de la simulation, la fitness de l'individu est positionnée. Sinon, le comportement de l'agent n'évolue pas d'une génération à la suivante. Il est important de noter que ceci n'empêche pas l'agent d'apprendre, même avec un apprentissage qui s'étale sur plusieurs simulations : les paramètres appris du comportement ne seront simplement pas le fait de l'évolution. Ceci est rendu possible par le fait qu'un agent peut être persistant d'une simulation/évaluation à la suivante et qu'il est inclus une méthode de paramétrage des architectures de contrôle (cf. ci-après la description de la classe `ControlArchi-`

`tecture`).
Un agent peut agir dans l'environnement de la simulation au moyen d'effecteurs (classe `Effector`), dont le nombre peut varier lors de la simulation. De façon analogue, l'agent peut percevoir des informations au moyen de capteurs (classe `Sensor`) et traiter les informations par des architectures de contrôle (classe `ControlArchitecture`). Un exemple d'agent sans architecture de contrôle serait un des véhicules de Braitenberg [BRA 86].
La classe `Agent` répond à 3 messages génériques :

- `Refresh()`, qui effectue un rafraîchissement des capteurs, des architectures de contrôle (par défaut de façon parallèle, cf. ci-dessous), des paramètres des architectures de contrôle et, enfin, des effecteurs.
- `RefreshArchPar()`, qui effectue un rafraîchissement parallèle des architectures de contrôle. Il consiste en l'utilisation des tubes de communication (classe `Pipe`) en entrée (resp. en sortie) des architectures de contrôle comme tampons en lecture (resp. en écriture) : les architectures sont toutes séquentiellement rafraîchies en lisant les tampons en lecture, puis les sorties des architectures sont séquentiellement écrites dans les tampons en écriture. Ainsi, bien que chaque opération soit faite séquentiellement, les architectures sont rafraîchies comme si elles l'étaient de façon simultanée.
- `RefreshArchSeq()`, qui effectue un rafraîchissement séquentiel des architectures de contrôle. Les architectures sont rafraîchies les unes après les autres, les tubes ne servent pas de tampon.

Les deux méthodes de rafraîchissement des architectures de contrôle ont deux rôles distincts. Le rafraîchissement parallèle permet de garantir la cohérence temporelle du flux de l'information circulant entre les architectures de contrôle, même en présence de récursions multiples. Pour n'importe quel rafraîchissement, quel que soit l'ordre de parcours des architectures de contrôle, l'information sera diffusée de la même façon. Cette méthode est intéressante, par exemple, lorsque le nombre et l'agencement des architectures de contrôles pour une expérience donnée sont susceptibles de changer au cours de la simulation. Le rafraîchissement séquentiel, par contre, ne garantit pas la cohérence temporelle : les effets du rafraîchissement dépendent de l'ordre du parcours des architectures. Il offre cependant l'avantage d'être plus réactif : il faut peu de rafraîchissements séquentiels successifs pour que l'information parcoure toutes les architectures de contrôle (en fait, un seul s'il n'y en a aucune sans entrées et qu'elles sont parcourues dans l'ordre depuis les capteurs jusqu'aux effecteurs) alors que, pour le rafraîchissement parallèle, il en faut au moins autant que d'architectures de contrôles en séquence entre les capteurs et les effecteurs. Cette méthode est donc plutôt intéressante lorsqu'il n'y a pas de récurrences, et que l'on veut un temps de traitement court des informations.

classe `Pipe`

Les tubes de communication (classe `Pipe`) gèrent le flux des données entre architectures de contrôle, capteurs et effecteurs. Ils permettent au signal de traverser séquentiellement ou parallèlement les architectures de contrôle, en ayant la possibilité de fonctionner comme des tampons en lecture/écriture. Un tube reçoit des données soit d'un capteur, soit d'une architecture de contrôle (il est alors une de ses sorties), et les envoie soit dans un effecteur, soit dans une architecture de contrôle (il est alors une de ses entrées).

classe `Effector`

La classe `Effector` est celle des effecteurs des agents. Elle répond au message `Refresh()` lancé en général par `Agent`. C'est une méthode abstraite – le rafraîchissement dépend de l'effecteur et du tube. `Effector` est lu lors du rafraîchissement du monde par le simulateur.

classe `Sensor`

De façon analogue, la classe `Sensor` est celle des capteurs des agents, qui répond au message `Refresh()` lancé en général par `Agent.Refresh()` est une méthode abstraite – le rafraîchissement dépend de l'environnement, de l'agent et du capteur. `Sensor` est positionné lors du rafraîchissement de l'agent auquel il est lié.

classe `ControlArchitecture`

Enfin, la classe `ControlArchitecture` est celle des architectures de contrôle. Comme pour `Sensor` et `Effector`, la méthode abstraite `Refresh()` est en général invoquée par l'agent. Une architecture de contrôle a un certain nombre d'entrées et sorties, qui viennent de ou vont vers des tubes. Un certain nombre de chromosomes peuvent être requis pour le paramétrage. Une architecture de contrôle est essentiellement une unité de traitement de l'information pour un agent.
Enfin, les architectures de contrôle répondent à un message `RefreshParams()`, rafraîchissement des paramètres. Celui-ci permet d'implémenter de l'apprentissage (tout algorithme classique), ou du développement dans le cas de réseaux neuronaux artificiels par exemple [GRU 94, KOD 98].

classe `SimuFitness`

Les simulations comme méthodes d'évaluation requièrent une façon particulière d'évaluer les fitness des individus. En effet, le concepteur/expérimentateur peut choisir d'évaluer progressivement la valeur de la fitness au cours de la simulation (à chaque pas de temps, à la fin de chaque expérience), et/ou de le faire en fin de simulation. On spécialise donc une classe `SimuFitness` dans ce but, cette classe est liée à un simulateur et à un agent donné. Chaque méthode d'évaluation a besoin d'avoir accès aux paramètres de l'environnement et à l'agent dont elle doit évaluer le comportement. La classe `SimuFitness` présente trois méthodes abstraites, qui sont appelées au moment opportun par la méthode d'évaluation du simulateur :

– `EvalStep(Simulator,Agent)`, qui ajourne la fitness à la fin d'un pas de temps,

– `EvalExp(Simulator,Agent)`, qui ajourne la fitness à la fin d'une expérience,
– et `EvalEnd(Simulator,Agent)`, qui ajourne la fitness à la fin de la simulation.

3.2.2. *Exemple de fonctionnement*

La figure 7 présente un scénario simplifié du fonctionnement d'un simulateur. Lors d'une évaluation, le simulateur instancie les agents, ainsi que leurs capteurs, effecteurs et architectures de contrôle, puis il boucle sur un passage à l'étape suivante via un appel `Step()` à l'échéancier. Ensuite, chaque appel boucle sur les évaluations individuelles de chacun des agents. En fin de simulation, le simulateur boucle sur une évaluation finale de chacun des agents, puis les détruit avant de rendre la main.
Le passage à l'étape suivante provoque un rafraîchissement des agents dans l'exemple, ce qui implique, pour chacun d'eux, la mise à jour de ses capteurs, puis celle de ses architectures de contrôle et de ses paramètres, et enfin celle de ses effecteurs.

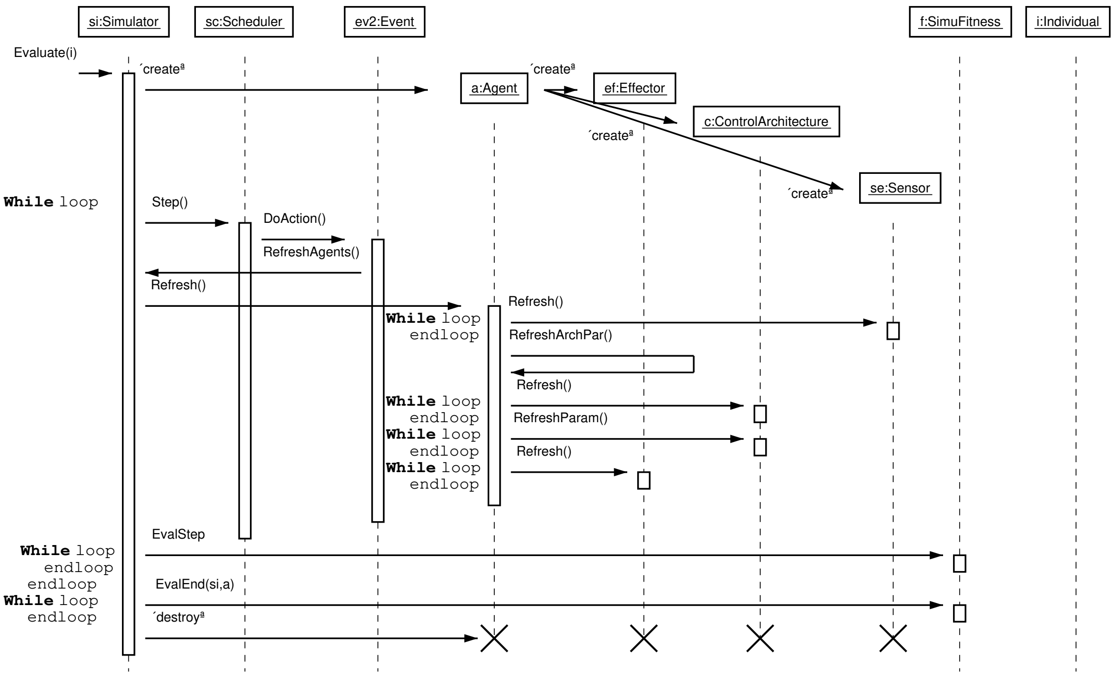

Figure 7. *Diagramme temporel de la partie simulation*

3.2.3. *Exemple de dérivation des classes*

Pour illustrer le fonctionnement de *SFERES* nous allons prendre comme exemple la simulation d'un système multi-robots dans un environnement de couloirs, à l'image

de celui de MICRobES [6]. Cet exemple fictif résume les classes de base d'un environnement inspiré de celui du simulateur pour MICRobES, et n'a pas vocation à fournir un modèle complet.
Chaque robot est doté de trois architectures de contrôle à base d'un réseau de neurones artificiels, de deux effecteurs (deux roues) et de trois types de capteurs : une ceinture de sonars, une jupe de pare-chocs (qui sont sensibles à la pression) et une caméra vidéo. Des agents dont le comportement n'évolue pas sont également introduits, ils représentent les personnes qui se déplacent aléatoirement dans les couloirs.

`RobotsSimulator`

Classe dérivée de `Simulator`. Elle contient un ensemble d'attributs qui sont les paramètres du monde (configuration des couloirs, positions des portes, meubles,...) et surcharge la méthode `RefreshWorld()`. Celle-ci va tenir compte des agents, des paramètres du monde qui peuvent changer en cours de simulation (via un événement de l'échéancier) – par exemple une porte qui s'ouvre ou se ferme – pour calculer l'état du monde de la simulation à l'étape suivante, ainsi que les différents paramètres de position et orientation des agents. Les informations des mouvements des agents sont calculées d'après les informations que chacun fournit sur l'état de ses effecteurs.

`ConcreteScheduler` *et* `ConcreteEvent`

Les classes concrètes de l'échéancier (dérivé de `Scheduler`) et des événements (dérivés de `Event`) sont très génériques. Les événements sont gérés par l'échéancier sous forme d'une liste ordonnée par la date des événements et ceux-ci savent associer un indice d'action à la méthode idoine du simulateur.

`MovingAgent`, `RobotAgent` *et* `PersonAgent`

`MovingAgent` est une classe abstraite dérivée de `Agent`, qui contient des informations de position et d'orientation. En sont dérivées les classes concrètes `RobotAgent` et `PersonAgent`, qui sont respectivement celle d'un robot et d'une personne. Aucune des méthodes génériques de rafraîchissement de `RobotAgent` ne nécessite une surcharge. En revanche, pour la classe `PersonAgent`, la méthode `Refresh()` est surchargée afin de calculer une valeur pour les effecteurs sans utiliser d'architecture de contrôle et avoir des déplacements choisis aléatoirement. Enfin, les agents robots dont le comportement est sujet à évolution incarnent chacun un individu lors de l'évaluation.

`WheelEffector` *et* `LegsEffector`

Ces deux classes dérivées de `Effector` sont respectivement la classe de la roue d'un robot et la classe des jambes d'une personnes. Un robot a deux roues (une droite

6. Maîtrise de l'Immersion de Collectivités de Robots en Environnement Standard, *http://miriad.lip6.fr/microbes/* [DRO 99]

et une gauche) et la classe `WheelEffector` contient essentiellement la vitesse de la roue. Pour une personne, `LegsEffector` contient le vecteur vitesse, positionné pseudo-aléatoirement à chaque pas de temps directement par `PersonAgent` et non pas suite à des calculs effectués par des architectures de contrôle comme pour les robots.

`RobotSensor`, `SonarSensor`, `BumperSensor` *et* `CameraSensor`

`RobotSensor` est une classe abstraite dérivée de `Sensor`, contenant la position d'un capteur sur un `RobotAgent` : la mesure d'un capteur dépend en effet de sa position sur le robot. En sont dérivées les classes `SonarSensor`, `BumperSensor` et `CameraSensor`, qui sont respectivement la classe d'un sonar, d'un pare-chocs et de la caméra. L'information des capteurs `SonarSensor` et `BumperSensor` est un flottant et celle de `CameraSensor` est une matrice de nombres représentants la couleur des pixels associés, captés par la caméra.
À noter que `PersonAgent` n'ayant pas de capteurs associés, il incombe au simulateur de tenir compte ou non du vecteur vitesse d'une personne, par exemple en cas de collision avec un objet, une paroi ou un autre agent.

`FloatPipe` *et* `MatrixPipe`

Ce sont les deux types de tubes utilisés pour cette simulation, dérivés de `Pipe`. `FloatPipe` est un tube qui transporte des nombres flottants et sert à connecter réseaux de neurones, capteurs (`SonarSensor` et `BumperSensor`) et effecteurs (`WheelEffector`). `MatrixPipe`, quant à lui, est un tube qui transporte une matrice de nombres et va servir pour la circulation de l'information en provenance de la caméra d'un robot.

`NeuralNetworkControlArchitecture`, `FloatNeuralNetworkControlArchitecture` *et* `MatrixNeuralNetworkControlArchitecture`

`NeuralNetworkControlArchitecture` est une classe abstraite dérivée de `ControlArchitecture`, implémentant les services d'un réseau de neurone, sans spécifier le type des entrées et des sorties. Ceux-ci sont spécialisés dans les classes dérivées `FloatNeuralNetworkControlArchitecture` et `MatrixNeuralNetworkControlArchitecture` par exemple, chacune ayant en sortie un certain nombre de flottants et, en entrée, des flottants pour la première et des matrices de nombres pour la seconde. Chacun de ces réseaux de neurones est connecté aux capteurs, effecteurs et éventuellement à d'autres réseaux par les tubes adéquats. D'autre part, chaque réseau de neurone va puiser dans les chromosomes qui lui sont associés des paramètres nécessaires à son fonctionnement, ou à sa façon de se paramétrer[7].

7. on fait alors en partie évoluer des paramètres d'apprentissage pour les réseaux de neurones, à ce sujet voir par exemple [FLO 96]

4. 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. Les directions futures de développement du framework sont :

– l'ajout de bibliothèques de classes implémentant d'autres algorithmes évolutionnistes (par exemple des algorithmes de coévolution ou des algorithmes n'ayant pas de notion de génération),
– l'ajout de bibliothèques de sous-classes communément utilisées dans les SMA (par exemple des environnement en 2 ou 3 dimensions, une gestion asynchrone des évènements) pour le simulateur,
– 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]),
– l'ajout de bibliothèques de classes permettant l'étude de problèmes classiques mettant en jeu de l'apprentissage.

Un certain nombre d'applications ont déjà été implémentées à l'aide de *SFERES*, mais les détailler n'est pas l'objet de cet article. Il nous paraît tout de même important de les mentionner, afin que le lecteur puisse se faire une idée des types d'utilisations possibles de *SFERES* (et donc de sa généricité). *SFERES* a ainsi été utilisé dans :

– *FROG*, une simulation proies-prédateurs : le prédateur est une caméra mobile au centre d'une demi-sphère, et les proies se déplacent sur la demi-sphère ; le prédateur capture une proie lorsque celle-ci est maintenue un certain temps dans la fenêtre de vision du prédateur.
– *BLIMP*, une simulation de ballon dirigeable [DON 99] : le ballon dirigeable est entraîné à maintenir un cap et une vitesse donnés en présence de vent.
– *MICRobES-SFERES*, une simulation multi-robots dans un environnement de bureau, précédemment invoquée : les robots, les personnes se déplacent dans l'environnement. L'apprentissage des robots porte notamment sur l'évitement d'obstacles, l'acquisition de capacités de proprioception ou la reconnaissance d'amers visuels dans l'environnement.

Cette liste est amenée à s'enrichir de nouveaux exemples, la vocation de *SFERES* étant d'intéresser d'autres équipes.

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.

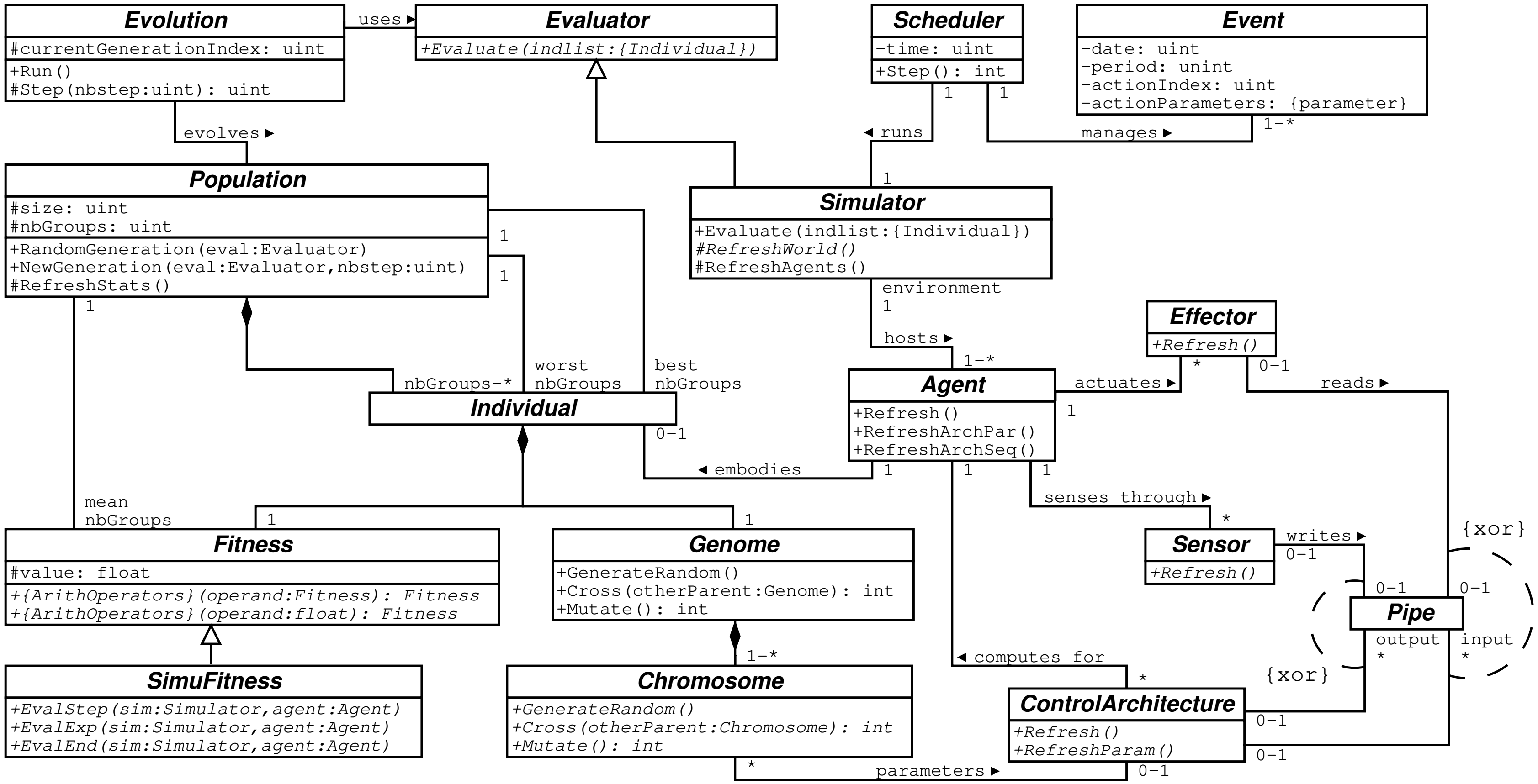

Figure 8. *Diagramme des classes de SFERES*

Bibliographie

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

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

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

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

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

[BRA 86] BRAITENBERG V., *Vehicles : Experiments in Synthetic Psychology*. MIT Press, 1986.

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

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

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

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

[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 ». In 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. et SPEARS W. M., « Using genetic algorithms to solve NP-complete problems ». In SCHAFFER J. D., Ed., *Proceedings of the Third International Conference on Genetic Algorithms*, p. 124–132. Morgan Kaufmann, june 1989.

[DON 99] DONCIEUX S., « Évolution d'architectures de contrôle pour robots volants ». In DROGOUL A. et MEYER J.-A., Eds., *Intelligence Artificielle située*, p. 109–127, Paris, octobre 1999. HERMES.

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

[DRO 99] DROGOUL A. et PICAULT S., « MICRobES : vers des collectivités de robots socialement situés ». In *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 96] FLOREANO D. et MONDADA F., « Evolution of Plastic Neurocontrollers for Situated Agents ». In MAES P., MATARIC M., MEYER J.-A., POLLACK J. et WILSON. S., Eds., *From Animals to Animats : Proc. of the Fourth International Conference on Simulation of Adaptive Behavior*, Cambridge, MA, 1996. MIT Press.

[FLO 98] FLOREANO D., NOLFI S. et MONDADA F., « Competitive Co-Evolutionnary Robotics : From Theory to Practice ». In PFEIFER R., BLUMBERG B., MEYER J.-A. et 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. et 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. et CARSE L. B. B., Evolving Multi-Agent Systems. In WINTER G., PÉRIAUX J., GALAN M. et CUESTA P., Eds., *Genetic Algorithms in Engineering and Computer Science*. Wiley and Sons, 1995.

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

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

[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. et VAN GUCHT D., « Genetic algorithms for the traveling salesman problem ». In 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, p. 65–89, 1992.

[GRU 94] GRUAU F., « *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm* ». PhD thesis, ENS Lyon – Université Lyon I, january 1994.

[HAY 94] HAYNES T., WAINWRIGHT R. et SEN S., « Evolving cooperation strategies ». Rapport technique UTULSA-MCS-94-10, The University of Tulsa, December 16, 1994.

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

[HEI 98] HEITKÖTTER J. et 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. et RICHARDSON J., « A classifier based system for discovering scheduling heuristics ». In GREFENSTETTE J. J., Ed., *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, p. 231–235, San Mateo, 1987. Morgan Kaufmann.

[HOF 94] HOFFMANN F. et 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, p. 297–314, july 1962.

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

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

[KOD 98] KODJABACHIAN J. et MEYER J.-A., « Evolution and Development of Neural Controllers for Locomotion, Gradient-Following, and Obstacle-Avoidance in Artificial Insects ». *IEEE Transactions on Neural Networks*, vol. 9, n° 5, p. 796–812, 1998.

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

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

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

[MEY 90] MEYER J.-A. et 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. et TAYLOR C. E., « Evolutionary Computation: An Overview ». *Annual Review of Ecology and Systematics*, vol. 20, p. 593–616, 1999.

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

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

[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. et SEKARAN M., Multiagent Coordination with Learning Classifier Systems. In Weiss and Sen [WEI 96], p. 218–233.

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

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

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

[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, p. 65–85, 1994.

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

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