Université de Montréal

Étude des environnements de microsimulation économique avec agents partiellement rationnels

par

Vincent Trussart

Département d'informatique et de recherche opérationelle
faculté des Arts et Sciences

Mémoire présenté à la faculté des études supérieures
en vue de l'obtention du grade de
Maître ès science (M.Sc.)
en Informatique

Octobre, 1999

Université de Montréal
Faculté des Études Supérieures

Ce mémoire intitulé:

Étude des environnements de microsimulation
économique avec agents partiellement rationnels

présenté par:

Vincent Trussart

a été évalué par un jury composé des personnes suivantes:

Nadia El–Mabrouk	Présidente–rapporteuse
François Major	Directeur de recherche
Jacques Robert	Codirecteur de recherche
Esma Aimeur	Membre du jury

mémoire accepté le: 99-10-12

Résumé

Dans ce mémoire, nous étudions la création d'un environnement de simulation macroéconomique basé sur des techniques de programmation orientées objets distribuées. Cet environnement sera utilisé pour valider une nouvelle approche en économie computationelle: la simulation microanalytique. La particularité de cette approche est que la simulation est effectuée sur une population d'agents indépendants apprenant leur comportement par l'utilisation de divers algorithmes d'apprentissage; les mesures d'intérêt de la simulation étant obtenues par l'agrégation des actions discrètes de chaque agent.

Sommaire

Ce mémoire s'oriente sur trois axes principaux: les systèmes distribués objets, les algorithmes d'apprentissage évolutifs et l'économie. Le but est de créer un environnement de simulation économique permettant de valider une nouvelle approche d'économie computationelle: la simulation microanalytique adaptative. La simulation microanalytique adaptative est une méthodologie simulant les actions d'agents ayant la capacité d'agir individuellement et d'apprendre de leurs actions. Les données d'intérêt sont extraites en faisant l'agrégation des différentes actions prises par les agents .

On verra que cette approche permet de prendre en compte la rationalité limitée des agents participants à une simulation économique. Une étude du projet *Aspen* sera ensuite effectuée; ce projet consiste à créer un outil de simulation microanalytique à grande échelle dont le but est de modéliser de grandes portions de l'économie américaine par le biais d'agents adaptatifs.

Nous relevons deux problèmes principaux freinant l'adoption des simulations microanalytiques adaptatives:

- la précision de l'approche demeure inconnue;
- les ressources informatiques requises sont imposantes.

Nous tenterons donc, dans ce mémoire, de résoudre ces deux problèmes. Pour résoudre le premier, un modèle simple de validation du simulateur sera créé; la simplicité de

ce modèle permet d'en calculer l'équilibre stratégique. Il suffit donc de vérifier si les résultats du simulateur concordent avec les résultats théoriques. L'aspect économique de ce modèle sera développé par Marcelin Joanis, un étudiant au second cycle en économie.

Ce modèle macroéconomique est composé d'un gouvernement, X Entreprises et Y Individus (où $X \ll Y$). Un produit P est vendu par les X entreprises; le prix de ce produit est fixé selon un algorithme d'apprentissage

Pour ce modèle, l'algorithme choisi est une variante de celui utilisé dans le projet Aspen, algorithme que ses concepteurs nomment *GALCS*. Chaque entreprise et le gouvernement embauchent des individus et un salaire leur est versé au début de chaque journée. Le comportement des individus est déterministe: ils achètent à chaque journée le plus de produits P qu'ils peuvent.

Après quelques semaines de simulations et de corrections au modèle économique et informatique, nous arrivons à la conclusion que les résultats produits par le simulateur et, par le fait même, par l'approche microanalytique adaptative concordent avec les résultats théoriques. En effet, les résultats sont encourageants; sauf pour quelques exceptions, la différence entre le prix moyen des entreprises et la valeur théorique est toujours de moins de 1%. Les résultats auxquels nous arrivons nous permettent d'affirmer qu'il existe un compromis entre la capacité d'apprentissage des agents et la taille de la simulation, entre précision et réalisme. En effet, des résultats plus précis pourraient être obtenus avec l'utilisation d'un algorithme plus performant et demandant plus de ressources informatiques que l'algorithme *GALCS*.

Pour les fins de validation, l'environnement de simulation sera construit de façon centralisée (par opposition à l'approche distribuée), en utilisant la technologie *Infobus*, dans le langage *Java*. La technologie Infobus facilite la création de protocoles de communication entre objets s'exécutant dans le même espace mémoire (même machine virtuelle *Java*).

Pour résoudre le second problème, l'environnement de simulation sera réimplanté dans un environnement distribué en utilisant la technologie *JavaSpaces*, de *JavaSoft*. Pourquoi l'environnement de simulation n'a-t-il pas été initialement codé avec *JavaSpaces*? Deux raisons:

- L'environnement est plus lourd que *Infobus*; trop lourd pour des fins de prototypage et de validation. La taille effective d'une simulation sur un seul ordinateur en utilisant *JavaSpaces* est beaucoup plus petite que celle que l'on peut obtenir avec *Infobus*.
- La raison principale: *JavaSpaces* n'était pas disponible lors de l'implantation du simulateur de validation.

La réimplantation du système en utilisant *JavaSpaces* permet l'exploitation de plusieurs caractéristiques très intéressantes fournies par cette technologie:

- Distribution des agents. Évidemment, en répartissant les agents sur plusieurs ordinateurs, la puissance et la lourdeur de leur module d'apprentissage peuvent être augmentées. Le nombre d'ordinateurs clients[1] se connectant au *JavaSpace* peut être accru au besoin. De plus, il serait assez simple d'implanter un algorithme de répartition automatique de la charge des ordinateurs clients par un déplacement des agents.
- Fractionnement du serveur. Cette capacité est assez étonnante. Étant donné que *JavaSpaces* force le programmeur à représenter son algorithme comme étant un flot d'objets et qu'il est possible de synchroniser "atomiquement" des transactions s'effectuant sur plusieurs serveurs *JavaSpaces*, il est facile de spécialiser certains *JavaSpaces* pour l'échange d'un sous-ensemble d'objets. Par exemple, dans le modèle de simulation, il est évident que si l'on augmente le nombre d'agents, (peu

1. Un ordinateur *client* est un hôte dans lequel les agents participant à la simulation s'exécutent.

importe sur combien d'ordinateurs ils se trouvent), le serveur exécutant le *JavaSpaces* doit gérer un trafic croissant d'objets-messages. Or, il serait très facile de fractionner le *JavaSpaces* en deux: un premier responsable des objets représentant les transactions monétaires (salaire, achat, taxe) et un autre responsable de tous les autres messages (nouvelle journée, journée terminée, publication des prix de vente, demande d'emploi, embauche, etc...). À la limite, il serait possible de créer autant de *JavaSpaces* qu'il y a de types de messages (objets, entrées) passés.

Étant donnée la conception modulaire des agents participant à une simulation, il est simple de changer l'algorithme d'apprentissage des agents ayant des capacités adaptatives. Or, la nature même de l'environnement de simulation impose plusieurs restrictions et contraintes dans le choix de ces algorithmes. Plusieurs algorithmes d'apprentissage seront donc étudiés pour en déterminer la pertinence dans le contexte de simulation à grande échelle d'un environnement dynamique et non supervisé où l'apprentissage se fait par renforcement. Parmi ces algorithmes, notons : *RL (Reinforcement Learning)*, *GALCS, LCS (Learning Classifier Systems)*...

Ce projet étant multidisciplinaire, il est essentiel que la définition de nouveaux modèles et que la paramétrisation des modèles à simuler soient simples et flexibles. Les composantes de base des deux environnements de simulation (local et distribué) sont codées en *Java*. Toutefois, le lancement de la simulation et la configuration des différentes variables sont faits en *Python* (*JPython*, pour être plus précis), un langage de scriptage orienté objet et interprété. Étant donnée l'intégration profonde de *JPython* à la plateforme *Java*, il est possible, par exemple, de créer une nouvelle classe *JPython* dérivée d'une classe *Java*. Donc, si un usager décide de modifier le comportement de la classe *Individu*, il n'a qu'à définir une nouvelle classe *Python* qui hérite de la classe *Java Individu*; la simulation peut s'effectuer en utilisant ce nouvel individu.

Table des matières

Table des figures

Liste des tableaux

Remerciements

Je tiens à remercier cordialement François Major, directeur de maîtrise; Jacques Robert, co-directeur; Marcelin Joanis pour ses brillantes observations économiques; Robert Gérin-Lajoie pour son support et ses observations toujours pertinentes et finalement le CIRANO (Centre Inter-universitaire de Recherche en Analyse des Organisations) et Bell/Émergis pour leur généreuse bourse d'études, sans laquelle je n'aurais pu faire cette maîtrise.

Introduction

Lors de l'élaboration de théories macroéconomiques, pour prédire le comportement des différents acteurs et l'équilibre des modèles, les économistes utilisent généralement des méthodes de résolution mathématique. Plus les modèles sont complexes, plus ils sont difficiles à résoudre. Parfois, représenter formellement le modèle à étudier s'avère tout autant laborieux.

Les techniques traditionnelles de simulation macroéconomique supposent que chaque *agent*[2] de la simulation agit de façon purement rationnelle[3]. Or, selon [Sar93], l'espérance rationnelle impose deux contraintes sur les modèles économiques:

- Le comportement rationnel des agents de la simulation;
- La cohérence entre les agents de la perception de l'environnement.

Or, lorsqu'un économiste implante un modèle de simulation traditionnel, l'espérance rationnelle impute beaucoup plus d'informations aux agents que l'économiste en possède; il doit donc inférer et estimer un comportement de façon arbitraire aux agents de la simulation.

2. Le terme *agent* décrit tout au long de ce mémoire un membre actif d'une simulation économique et non un agent informatique (bien qu'une implantation informatique de ces agents sera effectuée). Il peut donc être remplacé par *agent économique.*

3. Selon la science économique, des agents sont rationnels s'ils maximisent en fonction de préférences stables. Dans le contexte de la théorie, le concept de rationalité présume que les agents ont des anticipations correctes des actions des autres et qu'ils sont *bayesiens.*

Dans [Sar93], Thomas J. Sargent propose le concept de rationalité bornée. Pour montrer l'utilité des modèles à rationalité bornée, il décrit comment des modèles de simulation impliquant des agents adaptatifs permettent de résoudre certains problèmes difficilement traitables par des modèles à espérance rationnelle.

Une nouvelle approche de simulation

La simulation microanalytique adaptative est une méthodologie simulant les actions d'agents ayant la capacité d'agir individuellement et d'apprendre de leurs actions. Les données d'intérêt sont extraites en faisant l'agrégation des différentes actions prises par les agents .

Dans le contexte d'une simulation macroéconomique, cette approche offre plusieurs avantages par rapport aux techniques d'analyse macroéconomique traditionnelles. Parmi ces avantages, notons :

- La procédure de simulation ne requiert pas de représentation formelle pour les relations endogènes entre les agents du système. L'usager a donc une grande liberté lors de la modélisation du comportement individuel des agents. Par exemple, il est possible que le comportement d'une classe d'agents soit purement probabiliste et que celui d'une autre classe soit déterminé par un algorithme d'apprentissage. De plus, l'effet de certaines contraintes non-linéaires (taxes, réglementations..) peut être modélisé explicitement.
- La méthode de simulation étant centrée sur le comportement individuel des agents, l'usager peut bâtir un modèle microéconomique pour chaque agent au lieu de définir un modèle macroéconomique "global". De plus, beaucoup de données microéconomiques sont disponibles pour aider l'expérimentateur à modéliser le comportement des agents.

– Étant donné que chaque agent apprend de ses actions selon sa propre perception de l'environnement et qu'il n'a pas de connaissances *a priori* du modèle à résoudre, la simulation microanalytique ne souffre pas du problème de comportement purement rationnel des techniques d'analyse macroéconomique traditionnelles.

Deux obstacles majeurs freinent l'adoption de la simulation économique microanalytique:

– Le premier est la précision incertaine de la technique de simulation. En effet, étant donnée la nouveauté de l'approche, il existe peu de données permettant de valider et de mesurer la précision de la simulation microanalytique.
– Le second obstacle est le besoin énorme en ressources informatiques nécessaires pour effectuer une simulation à grande échelle. Il est effectivement très coûteux (en termes de ressources informatiques) de simuler le comportement détaillé de chaque individu.

Nous nous efforcerons, dans ce mémoire, de trouver des solutions à ces deux problèmes.

Validation de la technique de simulation

Pour valider la technique de simulation, nous tenterons de comparer les résultats obtenus par simulation microanalytique à l'équilibre stratégique[4] d'un modèle macroéconomique simple dont les composantes sont présentées à la section 2.1 (page 41). Pour fins de validation, une seule variable sera déterminée par le biais d'un algorithme d'apprentissage: le prix de vente du produit vendu par les entreprises (chaque entreprise fixe son propre prix). L'algorithme d'apprentissage utilisé pour déterminer les prix est une version modifiée de celui utilisé par le projet *Aspen*.

4. Dans le contexte d'un jeu économique, un équilibre stratégique est un point fixe qui spécifie pour chaque participant une stratégie qui lui est optimale étant donnée la stratégie des autres.

Distribution de la charge de simulation

Pour permettre d'augmenter la taille des simulations, plusieurs approches sont possibles. La première (la plus triviale) serait d'augmenter la puissance du matériel exécutant la simulation. On voit rapidement les désavantages d'une telle approche. Par exemple, si l'on suppose que la taille d'une simulation croît linéairement avec le nombre d'agents qui y participent, on devra doubler les performances du processeur pour doubler la taille de la simulation.

Une autre approche serait de programmer notre environnement de simulation pour qu'il puisse répartir sa charge de calcul sur tous les processeurs disponibles d'un ordinateur multiprocesseur.[5] Cette approche souffre évidemment des mêmes problèmes que la première. Il serait aussi possible d'utiliser une ferme d'ordinateurs combinée à une librairie de calcul distribués; il faut évidemment que la ferme soit composée d'éléments homogènes.

Comme nous le verrons plus loin, les systèmes distribués posent des contraintes au concepteur du système; la méthode de distribution de la charge étant la moindre. Le problème des défectuosités partielles est beaucoup plus important lors de la création d'environnements distribués robustes.

L'outil utilisé pour répartir la charge de calcul est *JavaSpaces*, un produit de *JavaSoft* permettant la création de systèmes distribués persistants orientés objets abordant de front les problèmes inhérents à ces systèmes.

5. Cette approche est utilisée par les développeurs du projet *Aspen* du *Sandia National Laboratories*. Leur implantation ne tourne que sur leur ordinateur multiprocesseur massivement parallèle *Intel Paragon*.

Chapitre 1

Revue de littérature

1.1 Simulations microanalytiques adaptatives

1.1.1 Définition

L'expression *simulation microanalytique* réfère à un modèle qui simule les actions primitives des différents agents décisionnels individuellement et qui génère des données macroéconomiques d'intérêt par l'agrégation des actions individuelles.

Selon [BPQA96], outre le projet *Aspen*, seulement deux modèles de simulation macroéconomique utilisant cette approche sont répertoriés : le modèle Bergmann-Bennet [BB86] et le modèle Orcutt-Caldwell-Weirtheimer [OCW76].

1.1.2 Projet *Aspen*

Aspen [BPQA96] est un projet du *Sandia National Laboratories* qui a pour mission de développer un système de simulation microanalytique à très grande échelle de l'économie américaine ayant, selon son auteur, le potentiel d'accroître les capacités d'analyse et de comparaison des différentes théories et politiques économiques. Contrairement aux

simulations traditionnelles de l'économie, le simulateur tente ici de modéliser le comportement de *chaque* entité économique (individu, entreprise...).

Les concepteurs *d'Aspen* tentent d'arnacher les capacités des nouvelles techniques d'apprentissage *évolutives* et la puissance toujours croissante des ordinateurs massivement parallèles en modélisant l'économie comme étant une séquence d'interactions discrètes entre agents ayant la capacité d'apprendre et s'exécutant de façon parallèle. Dans ce modèle les agents ayant la capacité de s'adapter à leur environnement et d'optimiser leur comportement[1], interagissent et communiquent entre eux au moyen de transactions, le plus souvent monétaires.

Les avantages d'un tel modèle sont :

- la modélisation cohérente et unifiée de l'économie;
- la possibilité d'analyser l'impact de la variation de variables exogènes (décisions légales, taxes, réglementations) de façon précise;
- la possibilité d'analyser de façon indépendante différents secteurs de l'économie.

Agents du modèle

Dans la composition actuelle du modèle *Aspen,* les agents sont dérivés de ces quatre types de base :

- Individu: Cet agent peut travailler ou recevoir des prestations d'aide sociale ou d'assurance chômage. Il dépense ses revenus en consommant (quatre types de biens distincts), en épargnant ou en investissant dans des obligations d'épargne.
- Firme: Chaque firme produit un des quatre types de biens possibles. Ces types sont : automobile, immobilier résidentiel, produits périssables essentiels (nourriture) et, en dernier lieu, produits non-durables dont la consommation dépend du

1. Les agents adaptatifs du projet *Aspen* utilisent un algorithme d'apprentissage simple décrit plus loin dans ce mémoire.

niveau de revenus. Les firmes déterminent le prix de vente de leurs produits selon un algorithme d'apprentissage (*GALCS*) qui tente de déterminer une stratégie optimale de fixation des prix.

- Gouvernement : Le gouvernement (agent unique) perçoit des impôts sur le revenu des particuliers et des firmes ainsi qu'une taxe de vente. Il est employeur (donc, il paie un salaire à ses employés), et il prend à sa charge le paiement de l'assurance sociale et des prestations de chômage. Il émet aussi des obligations d'épargne lorsqu'il est en déficit.
- Financier : Le domaine financier est composé de banques (dont le rôle est de maintenir des comptes d'épargne, de prêter de l'argent aux individus et aux firmes et d'investir dans les obligations d'épargne), d'une banque centrale décidant de la politique monétaire et d'un marché financier opérant sur les obligations d'épargne.

Plusieurs activités économiques ne sont pas simulées par le modèle actuel. Par exemple, le secteur militaire, le marché boursier et le secteur des services; il est toutefois prévu qu'elles y soient intégrées ultérieurement.

Prototype

Un premier prototype de simulateur a été développé en 1996. Ce modèle analyse l'évolution, sur 30 ans, d'une économie simplifiée composée de 1000 foyers, 4 firmes et un gouvernement. La figure 1.1 montre le flux monétaire de ce prototype.

1.2 Agents adaptatifs et rationalité limitée

Selon [Sar93], la simulation d'un modèle économique dans lequel un ou plusieurs agents ont un comportement appris (agents adaptatifs) permet de résoudre les problèmes

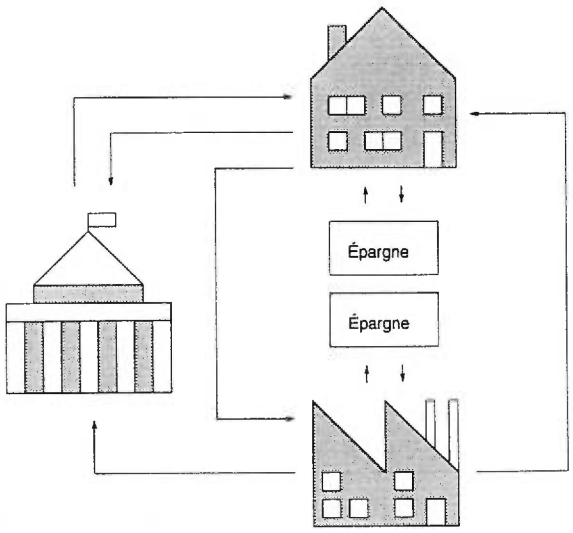

FIG. 1.1 – *Flux monétaire du premier prototype du modèle de simulation du projet* ASPEN *du* Sandia National Laboratories *composé d'un gouvernement, de 4 firmes et de 1000 foyers.*

reliés à l'espérance rationnelle étant donné que les agents n'ont pas de connaissances *a priori* et qu'ils ont la liberté d'expérimenter et de trouver certains comportements optimaux qui auraient autrement été inférés arbitrairement par l'économiste ou l'expérimentateur.

Sargent présente plusieurs modèles à rationalité limitée utilisant des agents adaptatifs et compare les résultats avec ceux qui seraient obtenus par des techniques traditionnelles impliquant l'espérance rationnelle. Parmi ces exemples, notons l'expérimentation de Marimon et Sunder (1992).

Marimon et Sunder [MS92] (cité par [Sar93]) ont fait des expérimentations en laboratoire pour étudier un modèle théorique ayant plusieurs points d'équilibre. Pour ce modèle, les solutions utilisant des méthodes de résolution purement rationnelles et celles utilisant des méthodes à rationalité limitée (à l'aide d'algorithmes adaptatifs) partagent les mêmes (deux) points d'équilibre; les caractéristiques de stabilité de ces deux modèles

Supervision	*Classe d'algorithmes*
Totale	Apprentissage manuel
.	Apprentissage par instruction
.	Apprentissage par analogie
.	Apprentissage par exemple
Nulle	Apprentissage par exploration

TAB. 1.1 – *Classes d'algorithmes d'apprentissage en fonction du degré de supervision requis tel que présenté par Michalsky, Carbonell et Mitchell [MCM83].*

étant par contre inversées. Les résultats de l'expérimentation de Marimon et Sunder valident pour ce problème l'approche de résolution à rationalité limitée. Selon [Sar93], ces résultats sont représentatifs des autres expérimentations de Marimon et Sunder; les résultats d'expérimentations sont en moyenne beaucoup mieux approximés par les techniques adaptatives que par les techniques purement rationnelles.

1.3 Algorithmes d'apprentissage

Il n'existe pas d'algorithme d'apprentissage universel; différents problèmes requièrent différentes approches. Michalsky, Carbonell et Mitchell [MCM83] ont proposé de classifier ces différentes approches en quelques groupes distincts, en ordre du degré de supervision requis par l'algorithme (voir le tableau 1.1).

La première catégorie est la plus simple car tout l'apprentissage est fait par un superviseur; les connaissances sont entrées manuellement. La seconde offre plus de flexibilité; le système doit pouvoir comprendre les instructions qu'il reçoit et les intégrer dans sa base de connaissances.

L'apprentissage par analogie est plus complexe que les précédents. Le système doit pouvoir trouver dans sa base de connaissances une situation s'apparentant à la situation

actuelle et ensuite la modifier pour usage ultérieur.

Les algorithmes de la quatrième catégorie (*par exemple*) acquièrent leurs connaissances par le biais d'informations associées à des échantillons ou des exemples provenant d'un environnement d'exécution. Ces informations peuvent être la valeur optimale (ou la classe) de l'échantillon (dans ce cas, on parle d'apprentissage supervisé) ou une mesure de la qualité de l'action exécutée (apprentissage par renforcement, défini plus en détails à la section 1.3.2).

Les algorithmes d'apprentissage par découverte et exploration tentent d'extraire l'information contenue dans les échantillons (données) et de classifier ces derniers; ces algorithmes sont donc totalement non supervisés. Dans cette catégorie pourraient se trouver les cartes de *Kohonen*.

Une autre méthode [dB94] de classification serait selon le type d'environnement dans lequel les algorithmes doivent faire leur apprentissage. En effet, l'environnement d'exécution peut être :

- statique ou dynamique;
- déterministe ou stochastique;
- discret ou continu.

Un environnement est dit dynamique s'il change en fonction du temps; un environnement statique reste toujours le même. Si un environnement est déterministe, il répond toujours de la même façon à une action; s'il est stochastique, les réponses à une action donnée (dans le même contexte) peuvent être différentes. Ce comportement probabiliste est dû soit à la nature même de l'environnement, soit à des perturbations causées par une mauvaise perception de l'environnement. Un environnement est fini s'il existe une quantité limitée d'états dans lesquels il peut se trouver et un nombre limité d'actions à prendre.

Comme nous le verrons plus loin, l'environnement de simulation économique microanalytique dans lequel nos agents devront évoluer est de type *dynamique*, *stochastique* et *discret* où l'apprentissage se fait à partir d'exemples (la quatrième catégorie définie précédemment). Pour que la simulation soit efficace, les agents devront résoudre le problème de la sélection de la prochaine action en se basant sur l'expérience acquise au cours des itérations précédantes.

Les prochaines sections sont consacrées à la description d'algorithmes qui pourraient être appropriés pour résoudre ces problèmes dans le contexte d'une simulation microanalytique économique à grande échelle.

1.3.1 Systèmes de production (*production systems*)

Les systèmes de production sont des processus de résolution de problèmes basés sur la comparaison avec des patrons (*pattern*). Ces systèmes sont composés d'un ensemble de règles de production, d'une mémoire interne et d'un cycle *reconnaissance-action*. Voici une description proposée par Luger et Stubblefield [LS97] :

- Les règles de production sont des paires *condition-action* et représentent une partie de la connaissance globale du système. La partie *condition* de la paire est un patron qui détermine quand l'usage de cette règle est approprié. L'action représente une des étapes nécessaires à la résolution globale du problème. Cette action agit sur la mémoire interne du système de production.
- La mémoire interne est une représentation de l'état courant de l'environnement. Cette mémoire est elle-même un patron qui est comparé à la partie *condition* de l'ensemble des règles de production. Lorsque la *condition* d'une règle correspond à l'état de la mémoire interne, l'action associée à cette règle devient potentiellement exécutable.

	Règles
1	ba $\rightarrow$ ab
2	ca $\rightarrow$ ac
3	cb $\rightarrow$ bc

TAB. 1.2 – *Règles d'un système de production simple.*

- Le cycle *reconnaissance-action* est la structure du contrôle (la séquence d'exécution) du système. En premier lieu, la mémoire interne est initialisée de façon à représenter l'environnement (la description du problème à résoudre). L'état du processus de résolution de problèmes est encodé sous forme de patrons dans la mémoire interne. Ces patrons sont ensuite comparés avec les conditions des règles de production. Le résultat de cette comparaison est un sous-ensemble des règles de production, nommé *ensemble conflictuel.* Les règles dont la condition s'apparie aux patrons dans la mémoire interne sont alors dites *activées.* Une des règles de l'ensemble conflictuel est alors sélectionnée par un algorithme de résolution de conflits et la partie *action* de la règle de production est ensuite exécutée. Après l'exécution de la règle, le cycle est répété en utilisant la mémoire interne modifiée par l'action exécutée. Le tableau 1.3, extrait de [LS97], montre une trace d'exécution d'un système de production simple.

Il y a plusieurs approches possibles pour résoudre les conflits (c'est-à-dire choisir une règle dans l'ensemble conflictuel). Par exemple, une approche triviale serait de choisir au hasard. Une approche plus intéressante serait d'inclure des heuristiques de sélection à l'algorithme de résolution de conflits. Cet algorithme est à la base de celui qui sera introduit à la section 1.3.3.

Itération	Mémoire interne	Ensemble conflictuel	Règle exécutée
0	cbaca	1,2,3	1
1	cabca	2	2
2	acbca	3,2	3
3	acbac	1,3	1
4	acabc	2	2
5	aacbc	3	3
6	aabcc	nil	Arrêt

TAB. 1.3 – *Itérations d'un système de production simple résolvant le problème du tri d'une chaîne de caractères. Pour cette exemple, la règle de résolution de conflits est de choisir le premier élément de l'ensemble conflictuel. Les règles de production utilisées sont définie dans le tableau précédant.*

1.3.2 Apprentissage par renforcement

Définition

Selon [dB94], un système d'apprentissage par renforcement est composé de deux ensembles : A l'ensemble des états dans lesquels peut se trouver l'environnement et B, l'ensemble des actions que le système peut exécuter sur l'environnement. Le système peut être défini comme une fonction (possiblement stochastique) $\psi = M(\phi,t)$ où $\psi \in B$ est l'action que le système exécute, $\phi \in A$ l'état de l'environnement et $t \in \aleph^+$ l'incrément de *temps* actuel (voir la figure 1.2).

La réaction de l'environnement est une fonction (elle aussi possiblement stochastique) $p = F(\psi,\phi,t)$ où $p \in \Re^+$ est la valeur de renforcement (positive ou négative) générée par l'environnement. Cette valeur est une mesure de la qualité de l'action ψ exécutée.

Un algorithme d'apprentissage par renforcement apprend si :

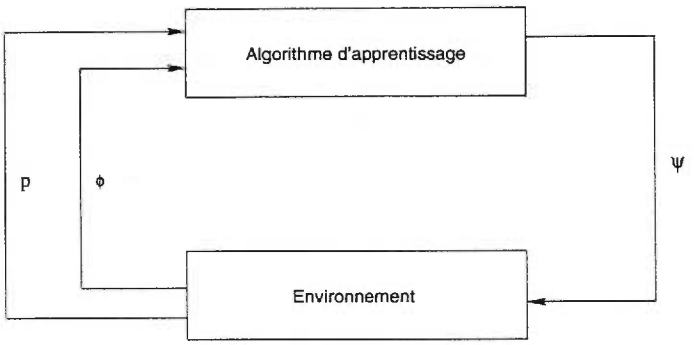

FIG. 1.2 – *Apprentissage par renforcement (tiré de [SB98]) où ψ est l'action exécutés, P est la valeur de renforcement et ϕ est l'état du système.*

$$\langle F(M(\phi_t,t),\phi_t,t)\rangle > \langle F(M(\phi_{t-1},t-1),\phi_{t-1},t-1)\rangle$$

où $\langle\,\rangle$ est utilisé pour dénoter la valeur espérée retournée par la fonction F.

Assignation des crédits

Une des composantes importante de ce type d'algorithme d'apprentissage est la fonction de renforcement (ou fonction d'assignation des crédits); un agent utilisant l'apprentissage par renforcement tente de maximiser les crédits futurs attribués par cette fonction. Par exemple, si un agent doit jouer au *backgammon*, la fonction pourrait retourner 0 en tout temps sauf en position terminale gagnante (+1) et en position terminale perdante (-1).

Étant donné qu'un agent tente de maximiser le renforcement futur, il apprend ultimement quels états mènent à la victoire et quels mènent à une défaite.

Pour qu'un agent puisse faire des choix, il doit avoir un moyen d'évaluer la qualité des états pour pouvoir par la suite prendre une décision. Cette fonction d'évaluation

est un appariement entre les états et la valeur de ces états; elle peut être approximée de plusieurs façons: table (*look-up table*), réseau de neurones... La valeur d'un état est définie comme la somme des renforcements estimée entre cet état et l'état terminal.

Approximation de la fonction d'évaluation

Initialement, l'approximation de la fonction d'évaluation est médiocre; l'appariement entre les états et les actions est initialisé aléatoirement. Pour que le comportement de l'agent s'améliore, ce dernier doit apprendre deux choses:

- une action ayant un effet néfaste immédiat ne doit pas être répétée;
- si toutes les actions associées à un état ont un effet néfaste immédiat, alors cet état doit être évité.

Soit $V^{\star}(x_t)$ la fonction d'évaluation optimale où x_t est le vecteur d'état, $V(x_t)$ l'approximation de la fonction d'évaluation et δ le facteur d'atténuation des crédits (ce facteur permet aux renforcements immédiats d'avoir plus de poids que les renforcements futurs). En tenant compte des erreurs d'approximation, on obtient $V(x_t) = e(X_t) + V^{\star}(x_t)$.

Si une table est utilisée pour approximer la fonction d'évaluation, les états doivent être discrets. La mise à jour d'un élément représentant un état dans la table est faite selon

$$\Delta w_t = \max_u(r(x_t,u) + \gamma V(x_{t+1})) - V(x_t)$$

où u est l'action choisie à l'état x_t, causant une transition vers l'état x_{t+1} et $r(x_t,u)$ est le renforcement reçu de l'environnement lorsque l'action u est prise à l'état x_t. Si l'espace des états est grand ou s'il est continu, l'utilisation d'une table est problématique; il est avantageux de la remplacer par un réseau de neurones.

Selon Maes [Mae97], une des caractéristiques attrayante des algorithmes d'apprentissage par renforcement est le fait que la convergence des agents vers le comportement optimal, sous certaines conditions, puisse être prouvée formellement. Or, ces conditions (nombre d'essais infini et environnement Markovien) sont rarement atteignables.

1.3.3 *LCS (Learning Classifier Systems)*

La discussion précédente sur les systèmes de production et sur les algorithmes d'apprentissage par renforcement nous permet d'introduire un type d'algorithme héritant de ces concepts : les *LCS* (*Learning Classifier Systems*).

Selon Mitchell et Forrest [MF97], les *LCS* sont basés sur trois principes: l'apprentissage, une réponse intermittente de l'environnement et une hiérarchie de modèles internes représentant l'environnement.

Boer introduit une notation formelle des *LCS* dans son mémoire [dB94]. Selon cette notation, un *LCS* χ est un tuple de la forme $\langle C,B,F \rangle$ où :

- C est un ensemble de classifieurs où $M = |C|$;
- B est la liste de messages;
- F est une fonction $F(C,B) \rightarrow (B',\Upsilon)$ qui prend en entrée un ensemble de classifieurs et une liste de messages et qui retourne une autre liste de messages et une sortie Υ (cette dernière peut être vide).

L'apprentissage, avec les *LCS*, se fait de deux façons : par l'application d'opérateurs génétiques sur C et par la rétribution des classifieurs. Le cycle de fonctionnement d'un *LCS* est composé de ces étapes [Dum95]:

- introduction de messages environnementaux produits par les capteurs de l'interface;

- collection des classifieurs qui reconnaissent les messages présents dans la liste;
- élection des classifieurs par un mécanisme d'enchère des classifieurs activables;
- activation des classifieurs sélectionnés et production d'une nouvelle liste de messages;
- extraction, de cette nouvelle liste, des messages destinés aux effecteurs et activation de ces effecteurs;
- rétribution des règles si l'action des effecteurs conduit à un renforcement positif;
- remplacement de la liste des messages par celle qui vient d'être produite par le système.

Classifieurs

Selon Boer, un classifieur est un tuple $\langle \Gamma, A, \Upsilon, S \rangle$ où :

- Γ est un ensemble de *conditions* de la forme $\langle g, C \rangle$ où g est un booléen et C est une chaîne de caractères dont l'alphabet est $\{0,1,*\}^l$ ($l \in N^+$);
- A est une chaîne dont l'alphabet est $\{0,1,\#\}^l$ ($l \in N^+$) représentant une *action*;
- Υ est une chaîne dont l'alphabet est $\{0,1,\#\}^l$ ($l \in N$) représentant la *sortie;*
- $S \in \Re$ est la force du classifieur.

Liste des messages

Cette liste, que nous avons précédemment notée B, est un ensemble de tuples (messages) de la forme $\langle \theta, \varphi \rangle$. Dans ces tuples, θ est de la forme $\{0,1\}^l$. Le classifieur responsable du placement de ce message dans la liste est $\phi \in C \bigcup \epsilon$. La taille maximale de la liste des messages est un paramètre du système.

La fonction de modification de la liste des messages

Pour définir la fonction F, nous avons besoin d'introduire deux nouvelles fonctions; une pour savoir si un classifieur a le droit de générer une sortie ($f_g(c,C) \rightarrow \{vrai,faux)$ $(c \in C)$) et une autre pour déterminer s'il peut placer un nouveau message dans la liste ($f_z(c,C) \rightarrow \{vrai,faux)$ $(c \in C)$) . De plus, un seul classifieur a le droit de générer une sortie ($|f_z(c,C) = vrai| = 1$).

La première partie de F détermine la nouvelle liste des messages:

$$F_B(C,B) = \begin{cases} \{\langle F_o(m,\gamma_{pref},A_c),c\rangle \mid c \in C \wedge c\, est\, active \wedge f_g(c,C)\} & si\, F_\gamma(C,B) = \emptyset \\ \emptyset & si\, F_\gamma(C,B) \neq \emptyset \end{cases}$$

La seconde partie de F est la fonction déterminant la sortie du système:

$$F_\Upsilon(C,B) = \begin{cases} \{\Upsilon_c \mid c \in C \wedge f_z(c,C)\} & si\, \exists c \in C \\ \emptyset & si\, \neg\exists c \in C \end{cases}$$

Appariement classifieur-message

Étant donnée la simplicité de l'alphabet des messages et des conditions, l'étape d'appariement se fait rapidement dans un *LCS*: si la distance de Hamming (en ignorant les bits associés au caractère #) entre la partie C de Γ et la composante θ d'un des messages de la liste B est nulle, alors le classifieur est *activé.*

Voici un exemple d'appariement entre des classifieurs et une liste de messages :

Étiquette	Message
a	0101
b	1010
c	1111

Condition	Satisfaite	Appariement
0101	oui	a
1101	non	
#101	oui	a
1###	oui	b,c
##00	non	
####	oui	a,b,c

Calcul de la mise

Lors de l'élection des classifieurs par un mécanisme d'enchère, la valeur de la mise doit être calculée à partir de la force de chaque classifieur. Par exemple, une fonction de calcul de la mise pourrait être :

$$B_c = f_{spec}(c) * S_c$$

où B_c est la mise du classifieur c, S_c est la force de c et $f_{spec}(c)$ est une mesure de la spécificité de c. Un classifieur s'appariant à moins de messages qu'un autre doit être considéré plus pertinent et avoir une mise plus forte. Une fonction possible pour $f_{spec}(c)$ serait :

$$f_{spec}(c) = \frac{L - W}{L}$$

où W est le nombre de "*" dans un classifieur et L est le nombre total de caractères.

L'algorithme de rétribution *bucket-brigade*

Cet algorithme, introduit par Holland ([Hol80], cité par [dB94]), modifie la force des classifieurs à chaque itération du système. Les classifieurs obtiennent rétribution de deux façons :

- en étant responsable d'une action causant un renforcement positif de l'environnement;
- en étant payé par un autre classifieur. Lorsqu'un classifieur active un message produit précédemment par un autre classifieur (la partie φ du message), il doit lui payer un pourcentage (ce pourcentage est un paramètre du système) de sa propre rétribution. Cette caractéristique permet de répartir le crédit plus équitablement entre les classifieurs.

Application des algorithmes génétiques aux *LCS*

Pour favoriser l'exploration des différents classifieurs possibles, les opérateurs génétiques courants doivent être appliqués sur l'ensemble des classifieurs du système. Le critère de sélection pour la reproduction des classifieurs est leur force (S). Lorsqu'une opération de croisement survient entre deux classifieurs, une force doit être attribuée aux descendants; cette valeur est une moyenne pondérée de la force des parents, étant donné qu'il n'existe pas de fonction d'évaluation (*fitness*) des classifieurs comme dans les algorithmes génétiques traditionnels.

Algorithme d'apprentissage *d'Aspen* : GALCS

L'algorithme utilisé dans *Aspen* est une version simplifiée d'un *LCS*. Dans *GALCS*, les classifieurs représentent les états dans lesquels l'agent peut se trouver (l'état est encodé à partir des senseurs de l'agent). À chaque état sont associées trois probabilités

(dont la somme est 1); chaque probabilité correspond à une action à soumettre aux effecteurs. L'apprentissage se fait par le renforcement ou la pénalisation des probabilités associées à un couple état-action.

Une variante de l'algorithme *GALCS* est utilisée dans ce mémoire; elle est décrite à la section 2.1.5.

1.3.4 Programmation génétique

Définition

La programmation génétique est une méthodologie de résolution de problèmes décrite par John R. Koza [Koz92] utilisant les méthodes et techniques des algorithmes génétiques introduits par les travaux de John Holland [Hol75]. L'idée principale de M Koza a été d'aborder la résolution de problèmes comme étant la recherche d'un programme informatique permettant de résoudre ce problème. Le paradigme de la programmation génétique fournit une méthode de recherche du programme informatique le plus apte (selon une fonction d'utilité) parmi l'espace des programmes admissibles (valides dans le langage choisi).

Selon [Koz90, Section 1, p. 7],

> "..indépendamment de la terminologie utilisée, le problème commun sous-jacent est la découverte d'un programme qui produit la sortie désirée lorsqu'on lui présente certaines entrées."

Plusieurs types de problèmes peuvent être résolus, en les reformulant de cette façon, par le moyen de la programmation génétique. Par exemple, notons :

- reconnaissance de patrons;
- recherche de stratégies dans un jeu;

- classification et formation de concepts;
- résolution d'équations différentielles;
- identification de fonction (symbolique) à partir d'un ensemble de données;
- régression;
- ...

La section suivante montre un exemple d'utilisation de la programmation génétique qui s'apparente au problème que nous tentons de résoudre dans ce mémoire. De plus, il montre comment les opérateurs génétiques peuvent être appliqués à la recherche de programmes informatiques.

Exemple d'utilisation

Un projet du laboratoire *Sandia*, dirigé par R.J. Pryor ([Pry98]), tente d'implanter un comportement autonome à des véhicules robotisés par le biais de la programmation génétique. Ces véhicules pourraient être utilisés pour des missions militaires ou humanitaires : opérations de déminage, surveillance, etc. Il est prévu que ces véhicules soient déployés par groupes variant entre quelques dizaines et quelques centaines pour résoudre une tâche précise.

Pour obtenir un comportement optimal, les concepteurs ont modélisé le problème par une grille à deux dimensions où se trouvent des murs et une cible à atteindre. Les robots sont placés arbitrairement sur la grille. La position de chaque robot est donnée par un couple d'entiers positifs (x,y) et la direction par un des quatre points cardinaux. Un robot ne peut détecter un obstacle que s'il lui fait face et il ne peut communiquer qu'avec les deux robots les plus rapprochés. De plus, il ne peut effectuer qu'une opération par exécution de son programme : ces opérations sont soit une rotation, soit un déplacement d'une case sur la grille. Le but de cet exercice est que chaque robot trouve la cible.

Les programmes générés par l'algorithme de programmation génétique sont représentés sous forme d'arbre composé de noeuds de deux types : fonctions (11) et terminaux (23).

Parmi les fonctions, notons:

- addition;
- soustraction;
- multiplication;
- division;
- comparaison;
- accès aux registres...

De plus, voici quelques exemples de noeuds terminaux utilisés pour *Robug:*

- direction;
- force du signal;
- position;
- l'impact d'une barrière.

Chaque individu de la population de programmes générés doit être évalué selon la fonction d'utilité:

$$u = \frac{1000.0}{1 - \sqrt{distance\,cible}}$$

De plus, pour limiter la taille des programmes, une fonction de pénalité est utilisée pour réduire l'évaluation des programmes selon leur nombre de noeuds.

$$p = \alpha \star nombre\,noeuds$$

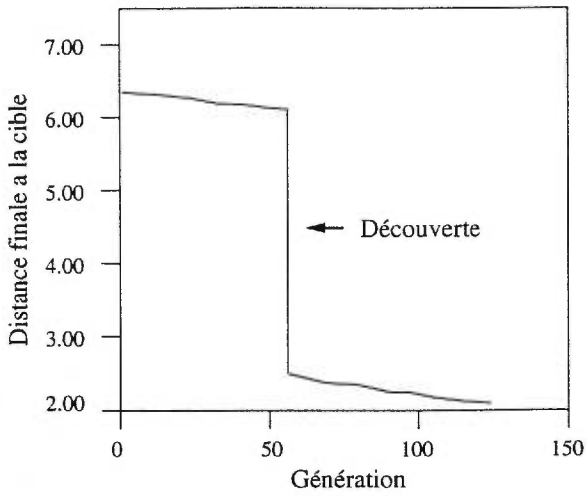

FIG. 1.3 – *Comportement typique de l'apprentissage par programmation génétique : la convergence s'améliore grandement lorsque les programmes générés* découvrent *une bonne approche de résolution.*

Pour créer une nouvelle génération d'individus, les opérateurs de reproduction, croisement et mutation sont utilisés; chaque opérateur possède une probabilité arbitraire d'utilisation (la somme des probabilités est 1).

Après quelques dizaines de générations, les programmes générés permettent aux robots d'atteindre leur cible. Le diagramme 1.3 montre l'évolution de la convergence des robots en fonction du nombre de générations.

1.4 Systèmes distribués objets : *JavaSpaces*

Les systèmes informatiques distribués ont un niveau de complexité ajouté et des contraintes qui leur sont propres; ces nouvelles contraintes n'existent tout simplement pas lorsqu'un système est construit de façon "locale". Selon [WWWK96], ces contraintes sont :

– latence augmentée;

- défectuosités partielles;
- hétérogénéité

Selon Waldo, Wyant Wollrath et Kendall, la gestion des défectuosités partielles est la contrainte la plus importante et elle doit être traitée explicitement. Par exemple, le système de fichiers distribué NFS[2] est un modèle de système local[3] traditionnel (API gestion de fichiers : *read*, *write*, *open*, *close* ...) converti en système distribué en désirant conserver l'*API*[4] du système local. Selon [WWWK96], les concepteurs de *NFS* ont abordé le problème des défectuosités partielles par le biais du "*soft mount*".

Il existe deux façons de gérer l'inaccessibilité d'un serveur de fichiers *NFS* : le "*soft mount*" et le "*hard mount*". Le "*soft mount*" expose les défectuosités du serveur ou du réseau aux clients par le biais de codes d'erreurs E/S standards et ces erreurs d'entrée/sortie sont beaucoup plus fréquentes que dans un système de fichiers local. Les applications client qui gèrent mal ces erreurs sont susceptibles de corrompre les fichiers qu'elles utilisent. Les "*soft mounts*" sont rarement utilisés aujourd'hui; la méthode "*hard mount*" lui est préférée. Un "*hard mount*" signifie qu'une application est suspendue jusqu'à ce que la communication avec le serveur puisse être rétablie. De cette façon, les applications du côté client sont isolées des défaillances partielles; le problème de cette approche est que si un serveur devient inaccessible, plusieurs postes client sont inutilisables.

Selon Waldo, cette façon d'aborder le problème des défaillances partielles est insuffisante: pour être plus robuste, l'*API* aurait dû être changé.

L'architecture *JavaSpaces* est conçue pour faciliter la création de systèmes distribués robustes permettant de résoudre facilement les problèmes intrinsèques de ces systèmes.

2. *Network File System*
3. *Local* en opposition à *distribué*
4. Interface de programmation *(Application Programmer Interface)*

Plus particulièrement, *JavaSpaces* offre des services de persistance distribuée et d'atomicité transactionnelle. De plus, *JavaSpaces* incite le concepteur à concevoir ses algorithmes comme un flot d'objets et le force à tenir compte des défis supplémentaires que représente la conception d'un système distribué robuste. *JavaSpaces* permet de résoudre les contraintes d'hétérogénéité (étant donné que chaque membre du système distribué s'exécute dans une machine virtuelle *Java*) et de défectuosités partielles (par le biais des *baux* et des transactions).

1.4.1 Concepts

Un *JavaSpace* contient des entrées[5] et des méthodes permettant de manipuler ces entrées. Les manipulations permises sont :

- l'écriture (*write*);
- la lecture (*read*, *readIfExists*);
- le retrait (*take*, *takeIfExists*);
- la notification (*notify*).

Les opérations de lecture, de retrait et de notification utilisent une entrée patron (*template*) sur laquelle l'appariement se fait. Lorsqu'une entrée patron est utilisée, ses champs servent de comparateurs. Si un champ est présent, il devra s'appareiller parfaitement; s'il est absent, il est exclu de la comparaison.

Supposons que cette instance de la classe *TestEntry* est écrite dans le *JavaSpace* (nous la nommerons entrée *E*):

5. Une entrée est un objet *Java* implantant l'interface *net.jini.space.Entry*

Classe : *TestEntry*		
attribut	type	valeur
nom	*java.lang.String*	"toto"
animaux	*java.util.Vector*	["foo", "bar"]

et que nous avons ces entrées patron, respectivement nommées *p1*, *p2* et *p3*:

Classe : *TestEntry*		
attribut	type	valeur
nom	*java.lang.String*	Null
animaux	*java.util.Vector*	["foo", "bar"]

Classe : *TestEntry*		
attribut	type	valeur
nom	*java.lang.String*	"toto"
animaux	*java.util.Vector*	["foo", Null]

Classe : *TestEntry*		
attribut	type	valeur
nom	*java.lang.String*	Null
animaux	*java.util.Vector*	Null

Les patrons *p1* et *p3* s'apparient à l'entrée présente dans le *JavaSpace*; les opérations de lecture et de retrait utilisant les patrons *p1* et *p3* pourraient retourner l'entrée *E*. De plus, si un client du *JavaSpace* s'était enregistré pour la notification de l'écriture d'une entrée s'appariant à un de ces deux patrons, il aurait reçu un événement (de type *net.jini.event.RemoteEvent*) l'informant qu'une telle entrée a été écrite dans l'espace.

Par contre, le patron *p2* ne s'apparie pas à l'entrée *E* étant donné que le mécanisme d'appariement ne s'effectue pas sur un graphe d'objets mais seulement sur les attributs immédiats (de premier niveau) d'un objet. Pour être plus précis, notons que la comparaison des attributs se fait sur leur représentation sérialisée (leur *java.rmi.MarshalledObject*[6] associé). Cette caractéristique permet d'augmenter grandement la performance générale des serveurs *JavaSpaces*, étant donné que la comparaison d'entrées peut s'effectuer sans reconstruire l'objet (autrement dit, le passage de représentation sérialisée (*MarshalledObject*) vers objet manipulable est évité). Toutefois, cette caractéristique ajoute cette contrainte : les attributs doivent être des objets (héritant de la classe *java.lang.Object*) et non des types primitifs (*int, long, float, double, byte, short, boolean...*). Lors de la comparaison des entrées, les types primitifs sont tout simplement ignorés.

Pour gérer les problèmes de défectuosités partielles, *JavaSpaces* offre, entre autre, le service de baux (*lease*s). Dans les *JavaSpaces*, l'obtention d'une ressource ou l'accès à un service est un privilège négocié entre le client et le *JavaSpace* pour une durée limitée. Ainsi, lorsque deux (ou plus) composantes d'un système distribué *JavaSpaces* n'arrivent plus à communiquer entre eux (segment du réseau coupé, effondrement d'un des participants...), le système peut continuer à s'exécuter de façon fiable. De plus, grâce aux baux, l'accumulation de ressources désuètes dans le *JavaSpace* est improbable.

Pour assurer la cohérence des données en empêchant la propagation de résultats partiels et erronés dans les systèmes distribués, *JavaSpaces* offre le service de transactions. Une transaction permet de grouper un ensemble d'opérations pour que d'un point de vue externe l'exécution de *toutes* les opérations du groupe soit un succès ou un échec. Les

6. Étant donné que *JavaSpaces* utilise *RMI* (*Remote Method Invocation*) comme protocole de communication, l'utilisation de la sérialisation d'objets introduite avec *Java* 1.1 est implicite. *Java* 2 (anciennement *Java* 1.2) introduit une nouvelle classe, "*java.rmi.MarshalledObject*", qui contient essentiellement une séquence d'octets représentant un objet sérialisé. Cette classe offre des méthodes de comparaison entre deux *MarshalledObject* : ce sont ces méthodes qui sont utilisées pour effectuer la comparaison entre deux entrées écrites dans un *JavaSpace*.

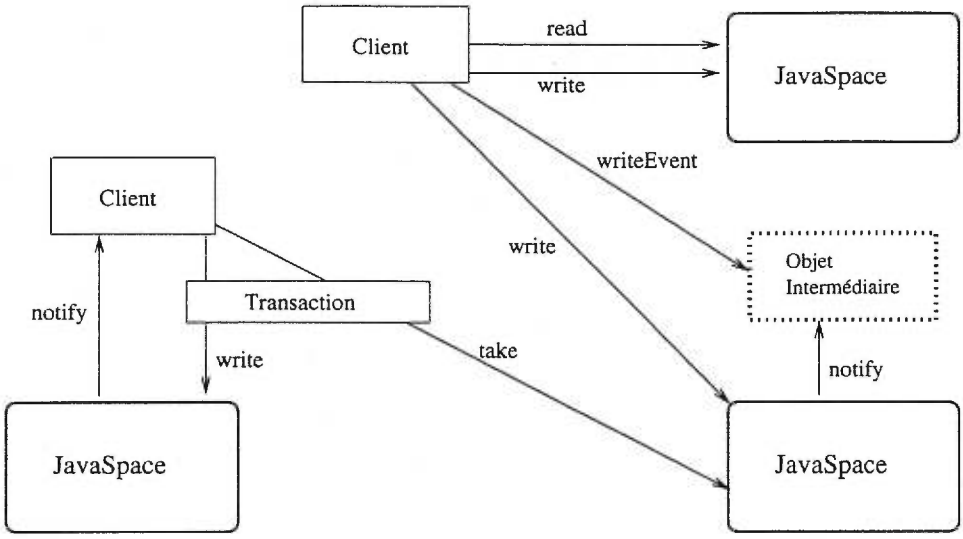

FIG. 1.4 – *Exemple de système distribué composé de plusieurs* JavaSpaces *utilisant les transactions et les événements distribués ainsi que les opérations de base* read, write, *et* take *(tiré de [Sun98b]).*

systèmes transactionnels standards utilisent souvent un moniteur de transactions (*transaction processing monitor*); ce dernier s'assure de la validité de l'implantation de la sémantique transactionnelle des participants à la transaction. JavaSpaces utilise une approche opposée, plus conforme à l'approche orientée objet: chaque objet participant à une transaction est responsable de la validité de son implantation du protocole transactionnel.

Tel que cité dans [Sun98c] :

> "Le but de ce système est de fournir l'ensemble *minimal* de protocoles et d'interfaces requises pour *permettre* aux objets d'un système d'implanter une sémantique transactionnelle commune et non l'ensemble maximal d'interfaces, protocoles et politiques pouvant assurer la validité de toute sémantique transactionnelle possible."

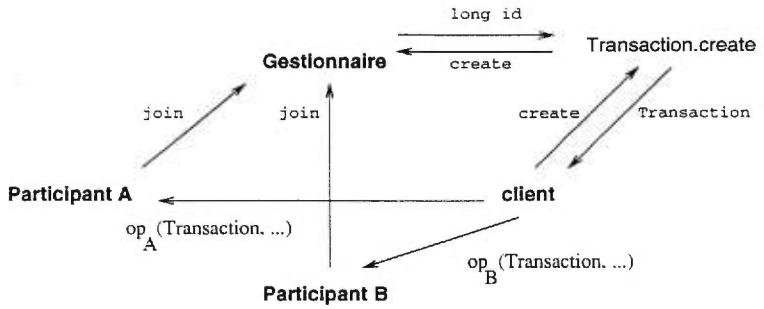

FIG. 1.5 – *Création d'une transaction distribuée comprenant deux propriétaires de ressources (*ParticipantA *et* ParticipantB*) par un client. (tiré de [Sun98c])*

Les transactions sont créées et gérées par un objet qui implante l'interface *TransactionManager* et elles possèdent une identité unique (un entier *long*) aux yeux de leur gestionnaire. Le diagramme 1.5 montre le processus de création des transactions. Un client désirant créer une transaction en fait la demande à un gestionnaire par le biais de la méthode statique *Transaction.create()*; cet objet *Transaction* peut ensuite être utilisé pour effectuer des opérations sur un service. Le propriétaire du service est nommé *Participant*; s'il accepte de participer à la transaction, il doit joindre cette dernière en faisant la demande au gestionnaire. Une transaction est terminée lorsque chaque participant approuve (*commit*) ou annule (*abort*). Pour qu'une transaction soit approuvée, chaque participant doit voter; le vote peut être "prêt", "inchangé" ou "échec". Si tous les participants ont voté "prêt" ou "inchangé", le gestionnaire ordonne à tous les participants "prêts" d'effectuer leurs modifications (*roll forward*). Si un participant a voté "échec", le gestionnaire ordonne à tous les participants "prêts" d'annuler leurs opérations (*roll back*).

Cette revue de littérature sur les simulations microanalytiques adaptatives, les algorithmes d'aprentissage et les systèmes distribués objets nous fournit les fondations sur lequelles nos prochains modèles de simulation seront bâtis. Le premier modèle, présenté au chapitre suivant, vise à valider l'approche de simulation.

Chapitre 2

Premier modèle : validation

Le premier modèle économique simulé est une version simplifiée du modèle de développement *d'Aspen*. Le but de ce modèle est de valider l'approche du simulateur et l'algorithme d'apprentissage qu'il utilise. Ce modèle est suffisamment simple pour que l'équilibre stratégique puisse en être calculé.

2.1 Description du modèle

2.1.1 Hypothèses économiques

Pour que l'équilibre stratégique de ce modèle économique puisse être calculable, certaines hypothèses doivent être posées :

- les entreprises disposent d'inventaires suffisants en tout temps;
- il y a plein emploi de la main-d'oeuvre;
- employeurs et employés sont liés par des contrats d'embauche à long terme;
- il n'y a que deux biens dans l'économie: le bien X et la monnaie;
- il n'y a pas de marché financier;

– le gouvernement et les entreprises s'endettent seulement à l'étranger (arrivée externe de capitaux).

2.1.2 Agents et processus

Plusieurs types d'agents ayant des fonctionnalités et des caractéristiques différentes peuplent ce premier modèle à simuler. Les agents des modèles locaux et distribués ont un comportement semblable; leurs diagrammes d'état se trouvent à la section 3.2.

Le gouvernement

Le gouvernement est un agent unique. Ses caractéristiques sont :

– prélève une taxe de vente v (%) sur chaque achat du bien X auprès des individus;
– prélève à la source un impôt proportionnel sur le revenu des individus de t (%);
– prélève un impôt sur les profits des entreprises de u (%);
– emploie e employés au salaire journalier de s.

Les entreprises

L'agent de type "entreprise" est multiple. Ses caractéristiques sont :

– produisent un bien homogène X à un coût unitaire c, constant pour les m firmes de l'industrie;
– l'entreprise j dégage des profits bruts

$$\pi_j = (p_j - c) * x_j$$

où x_j est la quantité vendue du bien X;

- à chaque période, l'entreprise j fixe son prix de vente du produit X selon un algorithme d'apprentissage (GALCS);
- emploie e employés au salaire journalier de s;
- chaque entreprise a un propriétaire unique;
- les profits et pertes nets (après impôts) sont imputés en entier aux propriétaires sous forme de dividendes (d).

Les individus

L'agent de type "individu" est multiple. Ses caractéristiques sont :

- Les n individus sont soit employés par l'une des firmes (me) ou par le gouvernement (e), soit propriétaires d'une firme (p). Ainsi,

$$(m+1)e + p = n$$

- à chaque période, les individus employés demandent

$$x_i = \frac{(1-t)s}{(1+v)\tilde{p}}$$

unités du bien X de l'une des m entreprises (où $\tilde{p}$ est le prix chargé par l'entreprise choisie).

- un individu choisit l'entreprise où il achète selon la distribution de probabilités suivante:

$$P(acheter\ de\ j) = \frac{k}{p_j^q}$$

où

$$k = \frac{1}{\sum_{j=1}^{m} p_j^q}$$

est une constante de normalisation;

- les individus propriétaires épargnent la totalité de leurs revenus.

2.1.3 Variables du modèle

Ce modèle simple comporte deux types de variables : les variables exogènes et endogènes. Les variables exogènes sont fixées alors que les variables endogènes émergent lors de la simulation.

Variables exogènes

- $v = 0$ (taxe de vente);
- $t = 0.2$ (taux d'imposition moyen des particuliers);
- $u = 0.16$ (taux d'imposition moyen des entreprises);
- $n = 504$ (nombre d'individus)
- $e = 100$ (nombre d'employés de chaque entreprise et du gouvernement)
- $p = 4$ (nombre de propriétaires d'entreprise)
- $m = 4$ (nombre d'entreprises)
- $s = 100$ (salaire journalier des employés des entreprises et du gouvernement)
- $q = 5$ (exposant de la distribution de probabilité de la décision de consommation)
- $c = 100$ (coût unitaire de la production du bien X)

Variables endogènes

La valeur de ces variables varie à chaque itération du simulateur:

- π_j (profits de l'entreprise j);
- p_j (prix chargé par l'entreprise j);

- x_j (quantité vendue par l'entreprise j);
- δ_j (écart entre p_j et le prix moyen de l'industrie $\overline{p} = \frac{1}{m}\sum_{j=1}^{m} p_j$).

La valeur initiale de p_j est tirée d'une distribution de probabilité uniforme ($p_j \sim U[50,150]$).

2.1.4 Équilibre stratégique

Le premier modèle économique présenté dans ce mémoire était suffisamment simple pour que le calcul de l'équilibre stratégique puisse être possible. [1]

Rappelons l'expression des profits de l'entreprise j :

$$\pi_j = (p_j - c)x_j$$

La demande à laquelle la firme fait face dépend de la probabilité que les consommateurs achètent leur produit, probabilité qui est calculée de la manière suivante:

$$P(acheter\ de\ j) = \frac{k}{(p_j)^q}$$

Soit :

$$Q = \sum_{i=1}^{n} x_i$$

la quantité totale demandée par les consommateurs. Nous avons alors:

$$x_j = \frac{k}{(p_j)^q} Q(p)$$

où $\frac{k}{(p_j)^q}$ peut s'interpréter comme la part de marché de l'entreprise j à une période donnée. Il importe de noter que, compte tenu de l'expression de la fonction de demande

1. Ce calcul de l'équilibre stratégique a été développé par Marcelin Joanis.

des consommateurs que nous avons retenue ($x_i = (1-t)s/\tilde{p}$ si $v = 0$), Q est une fonction du vecteur de prix p.

Exprimons maintenant la constante de normalisation k en fonction des prix. Nous avons nécessairement l'égalité suivante:

$$\sum_{j=1}^{m} \frac{k}{(p_j)^q} = 1$$

En isolant k dans cette équation, on obtient :

$$k = \frac{1}{\sum_{j=1}^{m} p_j^{-q}}$$

Nous avons donc que :

$$x_j = \frac{(p_j)^{-q}}{\sum_{j=1}^{m} p_j^{-q}} Q(p)$$

À l'équilibre stratégique, comme toutes les firmes sont identiques, nous aurons $p_j = p_k \, \forall j,k$. Soit $y = (1-t)s$, le revenu disponible de chacun des me consommateurs de l'économie, l'équation peut être réécrite de la manière suivante pour une entreprise à l'équilibre stratégique:

$$x_j = \frac{(p_j)^{-q}}{\sum_{j=1}^{m} p_j^{-q}} \cdot \frac{mey}{p_j} = \frac{(p_j)^{-q-1}}{\sum_{j=1}^{m} p_j^{-q}} \cdot mey$$

où mey est le revenu disponible total de l'ensemble des consommateurs. Remplaçons cette expression dans la fonction de profit :

$$\pi_j = (p_j - c) \cdot \frac{(p_j)^{-q-1}}{\sum_{j=1}^{m} p_j^{-q}} \cdot mey$$

La condition du premier ordre de la maximisation de l'équation précédente implique le prix d'équilibre suivant, identique pour toutes les entreprises:

$$p_j = \frac{c(m(-q-1)+q)}{q(1-m)}$$

En remplaçant dans cette expression les paramètres exogènes par leurs valeurs respectives (énumérées à la page 44) nous obtenons:

$$p_j = \frac{100(4(-5-1)+5)}{5(1-4)} = 100 \cdot \frac{19}{15} \approx 126.67$$

2.1.5 Algorithme d'apprentissage

L'algorithme d'apprentissage utilisé est une variante de *GALCS* (*Genetic Algorithm Learning Classifier System*), introduit dans la section 1.3.3 (nous n'utiliserons que trois variables endogènes contre quatre pour *GALCS).*

Pour ce modèle, l'algorithme tente d'apprendre l'action à effectuer a_i (augmenter le prix d'une unité, le maintenir à son niveau actuel ou le baisser d'une unité) en fonction des entrées surveillées. Ces trois entrées binaires (où 1 équivaut à un "oui" et un 0 à "non") sont :

- π_j : Les profits de l'entreprise j ont monté ou descendu suite à l'action a_{i-1}.
- p_j : Le prix chargé par l'entreprise j a monté ou descendu suite à l'action a_{i-1}.
- x_j : La quantité vendue par l'entreprise j a monté ou descendu suite à l'action a_{i-1}.

L'état de l'environnement, pour une journée donnée, est donc encodé sur une séquence binaire de trois bits. L'environnement peut donc se trouver dans 8 (2^3) états possibles. À chaque état est associé un classifieur qui est chargé de prendre une décision: augmenter,

maintenir, ou baisser le prix. Par exemple, le classifieur associé à l'état φ est un vecteur de probabilités (dont la somme est 1) de la forme suivante :

$$(p^d, p^s, p^a)$$

où :

- p^d est la probabilité de diminuer le prix d'une unité;
- p^s est la probabilité de garder le prix stable;
- p^a est la probabilité d'augmenter le prix d'une unité.

L'apprentissage se fait en modifiant la valeur des différentes probabilités au sein d'un classifieur. Par exemple, si le niveau de renforcement est $\delta = 0.1$ et que l'on désire effectuer un renforcement positif sur la décision de diminuer le prix, le classifieur *pourrait* changer de la façon suivante :

$$(0.3, 0.4. 0.3) \rightarrow (0.4, 0.35, 0.25)$$

Un des problèmes de cette approche est que peu importe la réponse de l'environnement, le renforcement reste le même; pour favoriser une certaine stabilité dans le comportement des agents, une notion de renforcement proportionnel pourrait être introduite. De plus, contrairement à l'algorithme *bucket-brigade* utilisé avec les *LCS*, la rétribution ne s'effectue que sur le dernier classifieur ayant généré une action.

La procédure d'apprentissage se résume donc à la procédure itérative : (au début d'une nouvelle journée i) :

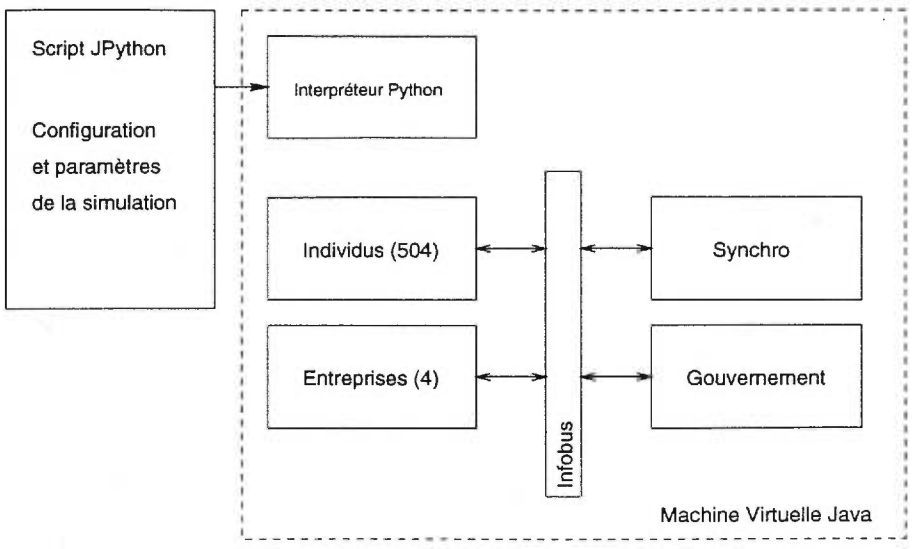

FIG. 2.1 – *Architecture informatique du premier modèle. Le canal de communication de ce modèle composé de 504 individus, 4 entreprises et un gouvernement est* l'Infobus. *Le démarrage et la paramétrisation de l'environnement de simulation se fait par le biais d'un* script JPython.

Étape	Action
Détermination de l'état actuel	φ_i
Calcul de l'évolution profit	$\Delta = \pi_i - \pi_{i-1}$
Renforcement du classifieur	$\varphi_{i-1} = \varphi_{i-1} + \delta$

Pour valider l'algorithme d'apprentissage et le simulateur, les entreprises du premier modèle devraient toutes fixer le prix de vente du produit X à 126.67 unités.

2.1.6 Modèle informatique

L'environnement de simulation du premier modèle est restreint à une seule machine virtuelle *Java*. L'architecture, présentée dans la figure 2.1, est centrée autour du protocole *Infobus*. Tel que cité dans [Sun98a], l'architecture *Infobus* est un ensemble d'interfaces facilitant la création d'applications composées de *Java Beans* qui échangent des données

de façon asynchrone.

Les membres d'un *Infobus* peuvent être de trois groupes : les producteurs, les consommateurs et les contrôleurs; les deux premiers étant les plus courants. Les contrôleurs sont spécialisés dans la médiation entre les producteurs et les consommateurs.

Pour recevoir les événements et échanger des données sur un *Infobus,* un consommateur doit s'y *inscrire.* Pour y parvenir, l'objet doit implanter les interfaces *InfobusMember* et *InfoBusDataConsumer*, obtenir une référence à l'objet Infobus et le joindre en tant que consommateur avec la méthode *Infobus.addDataConsumer(this).* Un producteur doit, pour sa part, implanter les interfaces *InfobusMember* et *InfoBusDataProducer* et joindre le bus en utilisant *Infobus.addDataProducer(this).* Les rôles ne sont pas exclusifs : un membre peut être à la fois consommateur et producteur.

Synchronisation

La gestion de la synchronisation entre agents est primordiale si l'on veut qu'une exécution parallèle puisse être possible. Une synchronisation trop forte implique une utilisation faible des ressources informatiques étant donné que les agents sont souvent en attente. Par exemple, le modèle de synchronisation du projet *Aspen* est basé sur une segmentation fine des journées; chaque agent du système doit, à la fin de chaque segment, suspendre son exécution. Or, pour favoriser l'exécution parallèle, nous avons tenté de relâcher ces contraintes de synchronisation.

L'approche utilisée dans ce modèle est basée sur l'exploitation des dépendances entre les différents agents participant à la simulation. L'élément de dépendance principal dans notre modèle économique est l'argent. Par exemple, un agent de type individu doit attendre le versement de son salaire par son employeur et la publication des prix par les firmes avant de pouvoir acheter des produits X.

L'objet (singleton) *ecosim.modele1.Synchro*, à la fois producteur et consommateur

sur *l'Infobus*, se charge de la gestion de la synchronisation. Pour y parvenir, il signale le début de chaque journée aux agents consommateurs d'événements du bus; l'agent synchroniseur peut envoyer ce message lorsque tous les agents actifs du système lui ont envoyé un message lui indiquant que leur journée de travail est terminée.

2.2 Résultats obtenus

Étant donnée la nature dynamique et stochastique de l'environnement de simulation, la compétition entre les entreprises et la simplicité de l'algorithme d'apprentissage, l'obtention d'un point de convergence stable est peu probable; un comportement oscillatoire (bruité) est prévisible. L'approche utilisée pour mesurer la performance de l'algorithme d'apprentissage est de calculer la moyenne du prix moyen des quatre entreprises après un certain point de stabilisation déterminé empiriquement. Après analyse du comportement des entreprises, il a été décidé que l'algorithme se stabilise vers la journée j=*500*.

La figure 2.2 montre l'évolution des prix lors d'une simulation durant quelques milliers d'itérations (*journées*). La moyenne du prix moyen des entreprises lorsque les variables exogènes sont fixées aux valeurs décrites précédemment est, après stabilisation, à 0.4% de la valeur prédite par l'équilibre stratégique.

Étant donnée la nature dynamique du simulateur, il est possible de perturber l'environnement de simulation en modifiant des variables exogènes *durant* la simulation. La figure 2.3 montre l'évolution de la moyenne des prix moyens des entreprises pour le produit X en fonction du temps, où le coût de production est changé à j=1000 (passant de c=100 à c=75). Cette variation du coût de production pourrait s'apparenter à une amélioration technologique faisant baisser le coût de production des entreprises. Le comportement du simulateur suit le modèle théorique. En effet, la moyenne entre les journées 500 et 1000 est, pour cette simulation, de 126.2. Après la perturbation, le prix

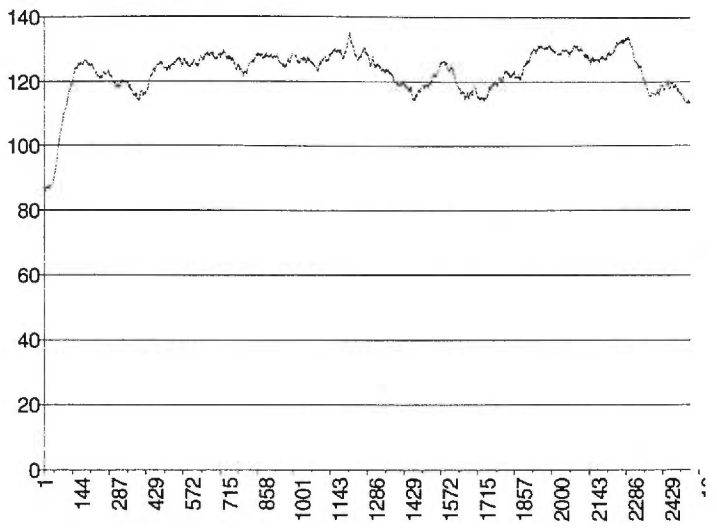

FIG. 2.2 – *Résultats de la simulation du premier modèle* (q=5)*, durant 3000 journées (le prix initial est déterminé aléatoirement). Le résultat moyen après stabilisation (journée j=500) est à 0.4% de la valeur prédite par l'équilibre stratégique.*

moyen passe à 95.3. Le tableau 2.1 montre les prix moyens observés et désirés, selon le développement présenté précédemment

Variation en fonction de q

On voit ici que l'algorithme d'apprentissage converge exactement vers l'équilibre stratégique, sauf pour $1 \leq q \leq 4$, où il devient imprécis. Nous supposons que la sim-

	Moyenne observée	Moyenne théorique	Erreur (%)
$c = 100$	126.2	126.67	0.4
$c = 75$	95.3	95	0.32

TAB. 2.1 – *Prix moyen des entreprises avec perturbation du coût de fabrication à la journée j=1000.*

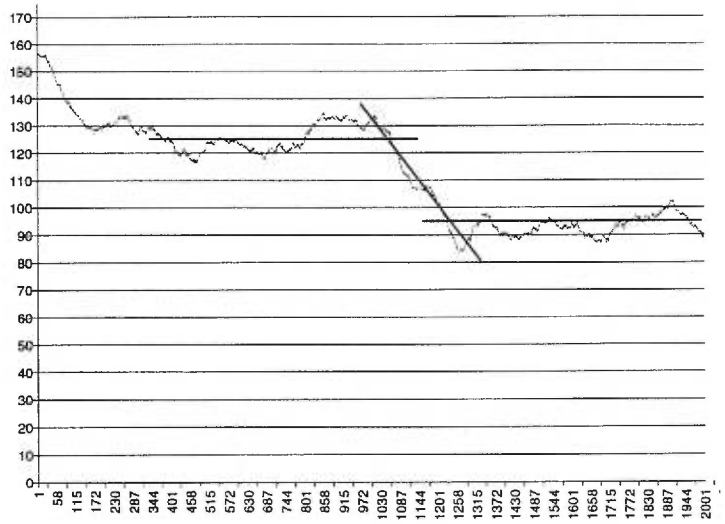

FIG. 2.3 – *Pour cette simulation, le coût de production* (c) *des firmes diminue à la journée j=1000; il passe de 100 à 75. Les firmes s'adaptent en baissant leur prix. Le prix moyen des entreprises se stabilise, après quelques itérations, à 0.32% de la moyenne théorique.*

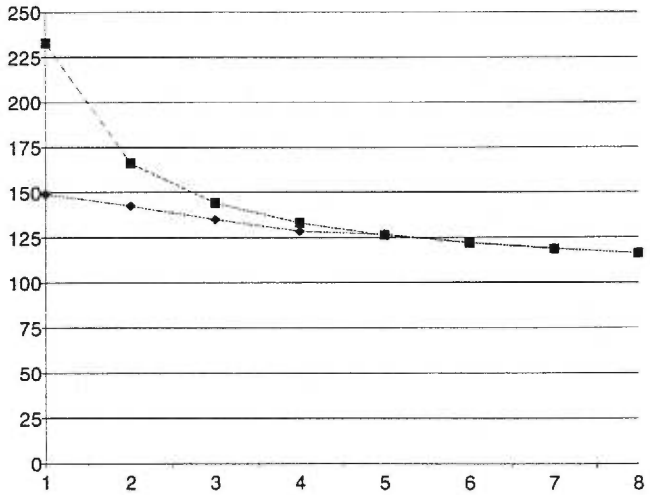

FIG. 2.4 – *Prix moyen des entreprises en fonction de* q *(exposant de la distribution de probabilité de la décision de consommation). On voit içi que l'algorithme est précis pour des valeurs de* $q \geq 5$.

q	$\tilde{p}$ théorique	$\tilde{p}$ observé
1	233	149
2	166.33	142.5
3	144.11	134.9
4	133	128.5
5	126.33	126.5
6	121.89	122.2
7	118.71	119
8	116.33	116.19

TAB. 2.2 – *Prix moyen (théorique et observé) des entreprises en fonction de* q.

plicité de l'algorithme d'apprentissage est la cause de cette imprécision. Les courbes représentées sur la figure 2.4 sont les fonctions de profit marginal pour plusieurs valeurs de q différentes. La fonction de profit marginal est la dérivée première de la fonction de profit; elle associe à chaque prix le profit additionnel généré par un incrément de prix d'une unité. Pour que le profit soit maximisé, il faut que le profit marginal soit nul (donc, la dérivée première); le simulateur devrait donc trouver une valeur proche du prix où la courbe croise la valeur de profit marginal égale à 0.

Or, on voit que la pente de la fonction s'affaiblit au fur et à mesure que q s'approche de 0; d'ailleurs, pour $q = 0$, le problème n'a pas de solution. Plus la pente est faible, plus l'algorithme échoue dans sa tâche d'atteindre la valeur théorique prédite. Par exemple, pour $q = 1$, au prix simulé, augmenter le prix de 1 aurait pour effet d'accroître les profits de 0.88%, ce qui n'est pas négligeable compte tenu des sommes en jeu.

Comparaison avec les résultats obtenus par *Aspen*

Nos résultats de simulation semblent concorder avec ceux du projet *Aspen*, tels que présentés dans le document [BPQA96]. Or, dans ce document, les concepteurs *d'Aspen* affirment que leur simulateur permet de prédire les variations cycliques des économies. Nous ne croyons pas qu'il soit possible d'affirmer que les cycles obtenus soient une ca-

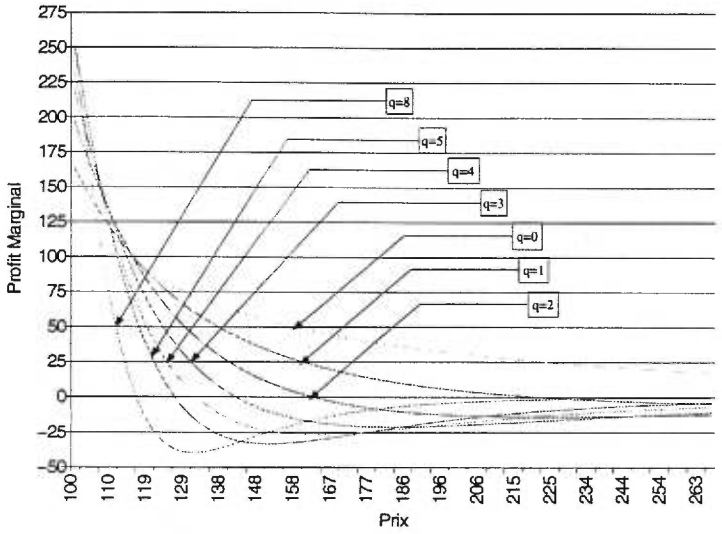

FIG. 2.5 – *Fonctions de profit marginal pour plusieurs valeurs de q (exposant de la distribution de probabilité de la décision de consommation) différentes. Pour que le profit soit maximisé, le profit marginal doit être nul.*

ractéristique émergeant du simulateur; nous mettons plutôt en cause la simplicité de leur algorithme d'apprentissage: *GALCS*. Par exemple, l'effet cyclique du simulateur *d'Aspen* pourrait être causé par méthode de rétribution de GALCS : la rétribution ne s'effectue que sur le dernier classifieur ayant généré une action (contrairement à l'algorithme *bucket-brigade*).

La technique de simulation étant maintenant validée, nous devons tenter de résoudre le deuxième problème des simulations microanalytiques : leurs besoins en ressources informatiques pour permettre une simulation à grande échelle. C'est l'objectif du second modèle de simulation présenté au chapitre suivant.

Chapitre 3

Second modèle : objets distribués

Pour parvenir à distribuer les agents économiques sur plusieurs ordinateurs, nous transformerons notre modèle informatique basé sur *InfoBus* pour qu'il utilise *JavaSpaces*.

La modification principale due à ce changement d'approche est l'éradication de toutes communications directes entre agents. Par exemple, dans le modèle *Infobus*, un individu peut acheter directement ses produits d'une firme : il s'agit tout simplement d'un appel de fonction entre un objet de type *ecosim.modele1.Individu* et un autre de type *ecosim.modele1.Entreprise*. Dans un *JavaSpace*, ce type de communication est proscrit : toute communication entre les éléments du système distribué doit se faire par le biais d'échange d'objets passant par un *JavaSpace*.

3.1 Entrées

Tous les messages passant sur *l'InfoBus*, dans le premier modèle, doivent être reconvertis en objets de type *entrée*[1]. Étant donné que les messages passés sont des objets, il est possible de définir une hiérarchie de classes représentant les entrées. De plus, toutes les communications directes entre objets (par appel de méthodes), qui étaient permises dans Infobus puisque les agents sont dans la même machine virtuelle *Java*, doivent être transformées en messages utilisant des entrées.

Le diagramme 3.1 montre la hiérarchie des classes entrées échangées dans le *JavaSpace*.

3.1.1 Description

Voici une description complète des différentes classes pouvant être écrites dans un *JavaSpace*.

SpaceEntry Classes de base dont héritent toutes les entrées pouvant être écrites dans un *JavaSpace*.

NewDayEntry Une entrée de ce type est écrite dans le *JavaSpace* par le *Synchro*. Ce dernier détermine que c'est le début d'une journée s'il a reçu un message de type *JobDoneEntry* pour chaque élément participant à la simulation.

JobDoneEntry Un participant à la simulation signale qu'il a terminé ses actions pour une journée par l'écriture d'une entrée de ce type dans le *JavaSpace*.

WorkNeededEntry Un individu désirant travailler pour une entreprise doit écrire une entrée de ce type pour signaler sa recherche d'emploi.

1. Les objets de type *entrée* doivent implanter l'interface "*net.jini.entry.Entry*" et avoir été traités avec le compilateur *rmi* (*rmic*).

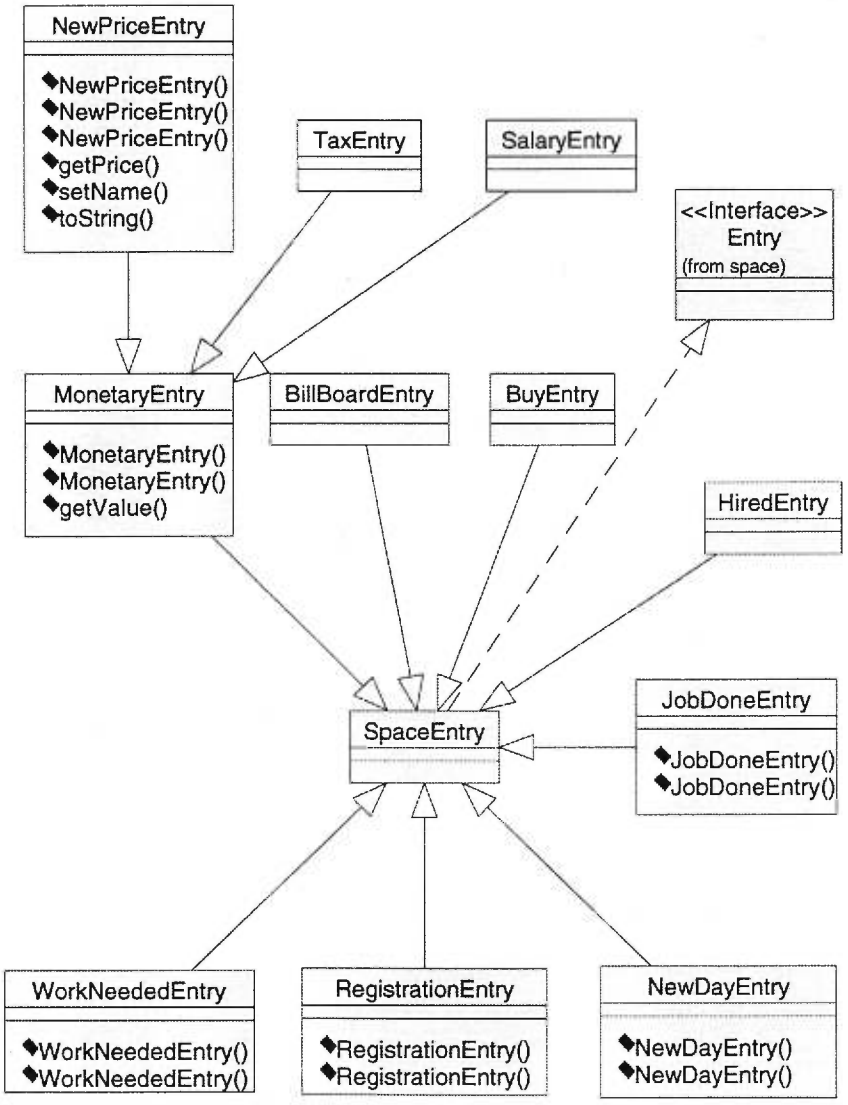

FIG. 3.1 – *Diagramme de classes* UML *montrant les relations entre les classes de type "entrées" présentent dans le modèle de simulation distribué. Ces classes seront membres du* JavaSpace.

HiredEntry Une entreprise désirant embaucher un individu effectue un *take* d'une entrée de type *WorkNeededEntry* et retourne un message de type *HiredEntry* à l'individu en question.

RegistrationEntry Si l'identité d'un participant n'est pas connue au moment du démarrage, ce dernier doit écrire une entrée de ce type dans le *JavaSpace* pour avertir le *Synchro* de sa présence.

BuyEntry Lorsqu'un individu désire acheter des produits d'une entreprise, il écrit une entrée de ce type permettant à la compagnie qui lui vend les produits de connaître la quantité d'items transigés.

BillBoardEntry Une entrée de ce type est écrite dans le *JavaSpace* par le *Synchro* au début de chaque journée. Cette entrée contient différentes valeurs d'intérêt public. Par exemple, si un secteur bancaire est ajouté au système, le taux d'intérêt de la banque centrale pourrait s'y trouver. Le *Synchro* est responsable de son écriture au début de chaque journée. Étant donné que chaque *BillBoardEntry* est associé à une journée, le *Synchro* doit aussi retirer du *JavaSpace* l'entrée associée à la journée précédente.

MonetaryEntry Cette entrée est la super-classe de toutes les entrées représentant un échange monétaire. L'attribut commun de ces entrées est le montant de l'échange.

NewPriceEntry Une entrée de ce type est écrite par chaque entreprise au début de chaque journée.

TaxEntry Entrée écrite lorsqu'une perception de taxe ou d'impôts (impôts sur le revenu des entreprises, des individus et taxe de vente) est effectuée. Le récepteur de ce type d'entrées est le gouvernement.

SalaryEntry Chaque entreprise écrit, au début de chaque journée, une entrée de ce type

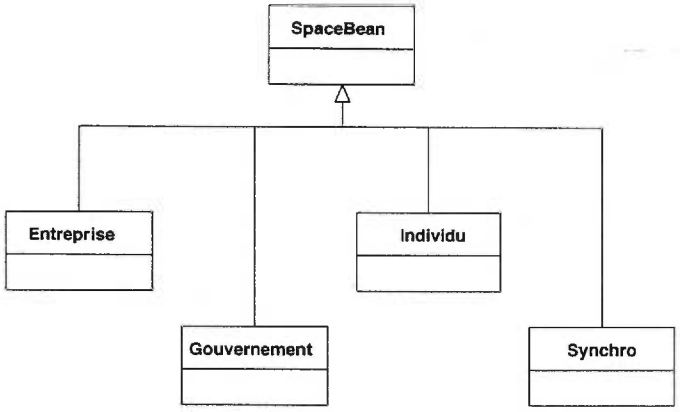

FIG. 3.2 – *Hiérarchie des agents participant à la simulation.*

représentant le salaire versé à chacun de ses employés (une entrée par employé). Le récepteur de cette entrée est l'individu travaillant pour cette entreprise.

3.2 Agents participant à la simulation

Le diagramme 3.2 (page 60) présente les différents agents participant à la simulation. Chaque agent est un client du *JavaSpace*: il possède une référence à ce dernier.

Les figures présentées dans les sections suivantes sont les diagrammes d'activité[2] associés à chaque agent s'exécutant dans l'environnement de simulation distribué.

Individus Le diagramme 3.3 (page 61) montre l'activité d'un agent de type *Individu* s'exécutant dans un *JavaSpace*. Ces agents sont créés au moment du lancement de la simulation par le script de démarrage et ils travaillent soit pour le gouvernement, soit pour une entreprise.

2. Tels que spécifiés par le standard *OMG UML 1.0*

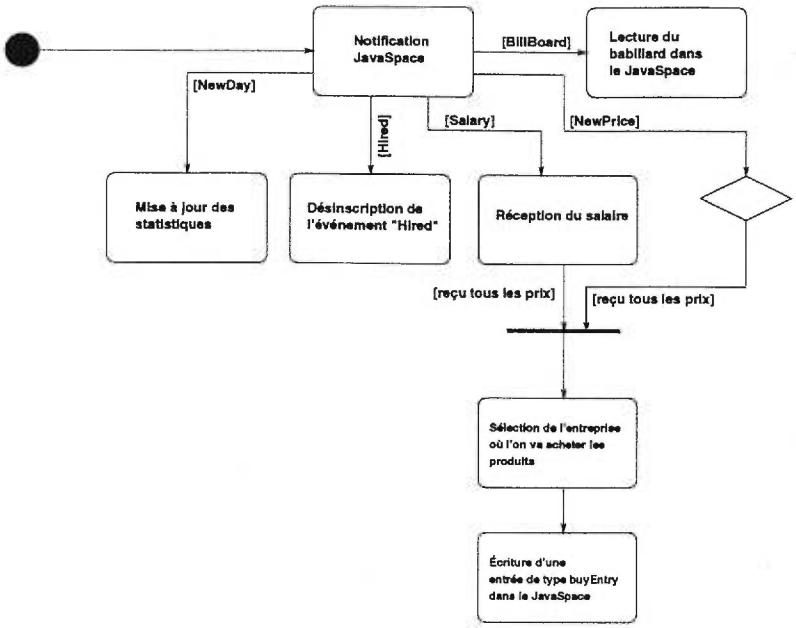

FIG. 3.3 – *Diagramme d'activité des agents* Individu.

Gouvernement Le diagramme 3.4 (page 62) montre l'activité d'un agent de type *Gouvernement* s'exécutant dans un *JavaSpace*. L'agent *Gouvernement* est unique et est créé par le *script* de démarrage initial. Dans ce modèle, la perception des taxes et des impôts est faite par l'entreprise et il n'y a aucune vérification de la part du gouvernement. La cohérence comptable est donc dépendante du comportement des agents; la tâche revient ultimement au programmeur.

Entreprises Le diagramme 3.5 (page 63) montre l'activité d'un agent de type *Entreprise* s'exécutant dans un *JavaSpace*.

Synchro Le diagramme 3.6 (page 64) montre l'activité d'un agent de type *Synchro* s'exécutant dans un *JavaSpace*. Cet agent supporte deux modes d'inscription: la préinscription et l'inscription différée.

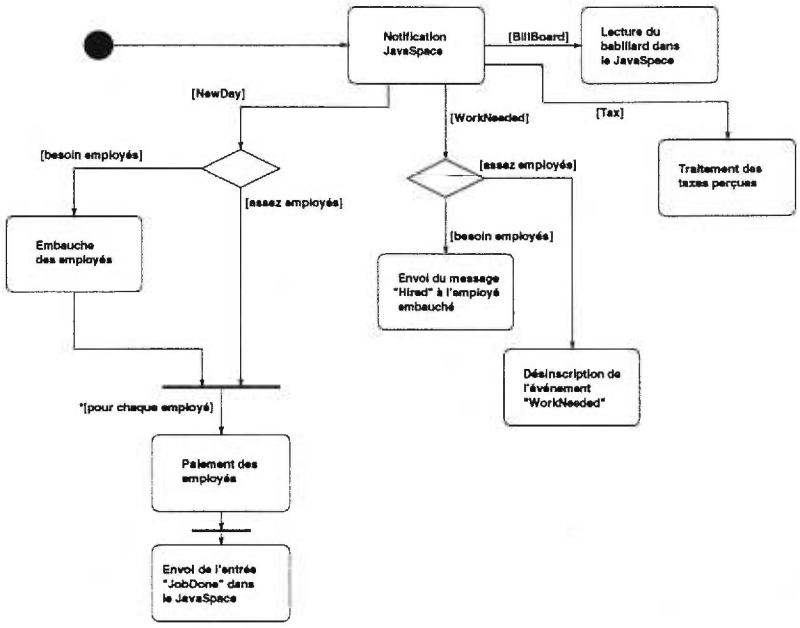

FIG. 3.4 – *Diagramme d'activité de l'agent Gouvernement*

Le premier mode suppose que tous les membres actifs sont connus au moment du démarrage de la simulation. Le second permet l'ajout de façon dynamique d'agents à l'environnement de simulation; cette capacité permet, par exemple, l'analyse de l'impact d'une augmentation de la population sur l'activité économique. Cet agent est aussi responsable de l'écriture des données publiques du système telles que les variables exogènes, les statistiques courantes, etc. Pour publier ces données il écrit, entre le moment où il a reçu tous les messages de type *JobDoneEntry* et celui où il déclare le début d'une nouvelle journée, une nouvelle entrée de type *BillboardEntry*. Nous verrons, dans la prochaine section, comment l'agent *Synchro* obtient ces données.

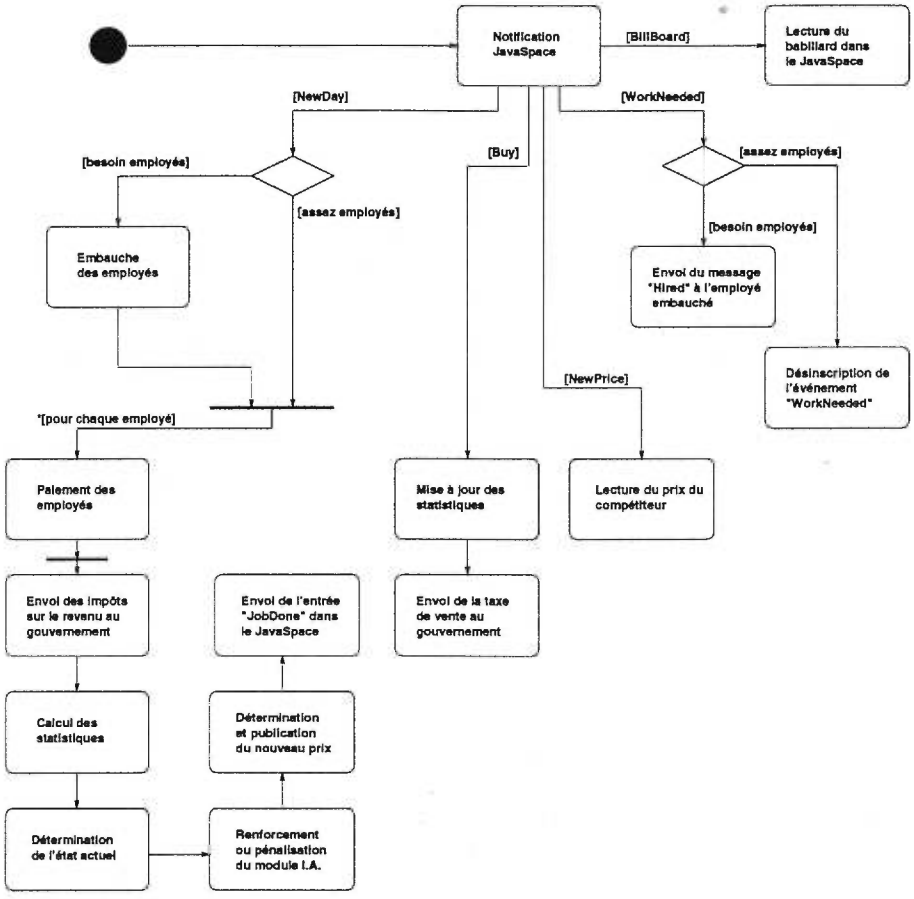

FIG. 3.5 – *Diagramme d'activité d'un agent Entreprise.*

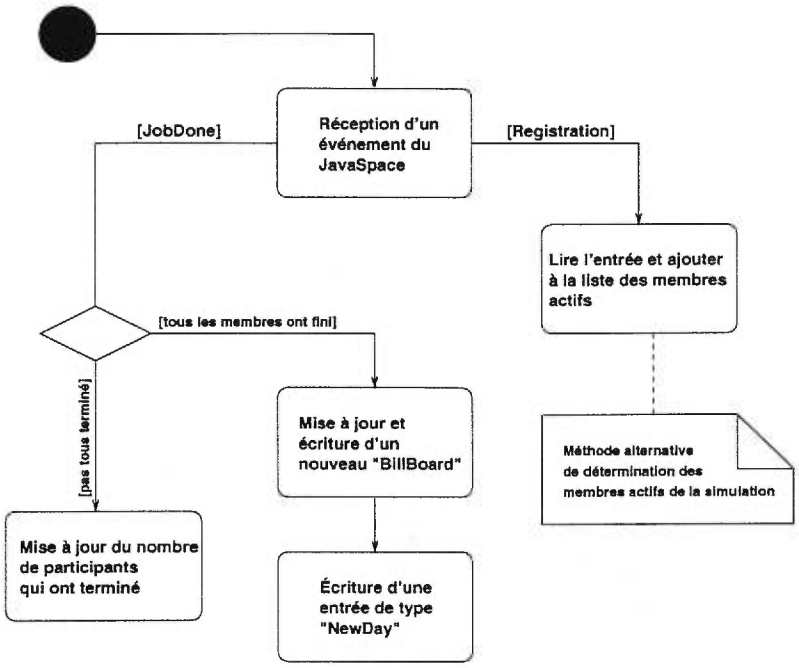

FIG. 3.6 – *Diagramme d'activité de l'agent Synchro.*

3.2.1 Configuration

Étant donné que le concepteur d'une simulation n'est pas nécessairement l'auteur du simulateur, le logiciel doit être facile à configurer, utiliser et modifier. Au lieu de définir un langage de description d'une simulation, un langage de scriptage tout usage a été intégré au simulateur : de cette façon, le langage de description et de configuration supporte mieux les modifications et l'évolution du simulateur. Le langage choisi est *JPython,* parce que c'est un langage intreprété facile à apprendre (même pour une personne ayant des connaissances de programmation limitées) tout en demeurant puissant (orienté objet, gestion d'exceptions...) et qu'il s'intègre complètement à la plate-forme *Java*. L'interpréteur *JPython* est lui-même codé en *Java*: il est donc possible de l'intégrer à une application *Java* et d'exécuter des scripts *JPython* à l'intérieur de la même machine virtuelle (*JVM*). De cette façon, les instances d'objets *Java* peuvent accéder aux

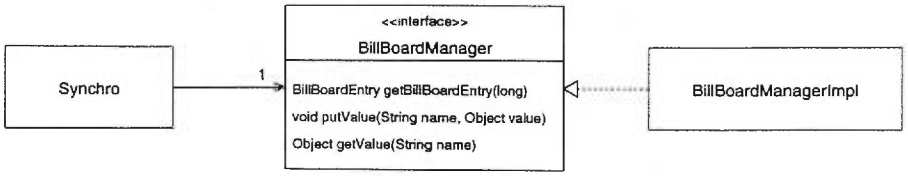

FIG. 3.7 – *Gestion des babillards.*

instances d'objets *JPython*; l'inverse est également possible. Les objets des librairies des deux langages sont mutuellement disponibles et instanciables. De plus, les exceptions des deux langages sont compatibles : par exemple, un script *JPython* peut intercepter et traiter une exception lancée par un programme *Java*.

Le lancement et la configuration des variables exogènes d'une simulation se font à l'aide d'un script de démarrage. Le script complet associé au modèle de validation se trouve en annexe A.

L'agent *Synchro* délègue la tâche de créer le babillard à chaque nouvelle journée à un objet qui implante l'interface *ecosim.space.BillBoardManager*. Cette approche de programmation, très fréquemment utilisée dans *Java*, permet de créer le gestionnaire de babillard dans le script de démarrage, du moment qu'il implante cette interface.

Pour introduire des perturbations dans le système, le gestionnaire doit tenir une table de modification des variables exogènes. Par exemple, pour que la variable représentant le coût unitaire (c) de production du produit X des entreprises chute de 100 à 75 à la journée $j = 300$, cette table de perturbations devrait contenir les entrées suivantes : "*0, c, 100*" et "*300, c, 75*"; une variable conserve sa valeur jusqu'à la prochaine perturbation.

L'utilisation de *JavaSpaces* pour la création du simulateur distribué apporte plusieurs avantages et quelques inconvénients. Le premier avantage est le plus évident : la distribution des agents. En répartissant les agents sur plusieurs ordinateurs, la puissance de leur module d'apprentissage peut être augmentée. De plus, il serait assez simple d'implanter

un algorithme de répartition automatique de la charge des ordinateurs clients par un déplacement des agents. Un autre avantage de *JavaSpaces*, plus étonnant, est la capacité de fractionner le serveur. Étant donné que *JavaSpaces* force le programmeur à représenter son algorithme comme étant un flot d'objets et qu'il est possible de synchroniser "atomiquement" les éléments de ce flot par le biais de transactions pouvant s'effectuer sur plusieurs serveurs *JavaSpaces*, il est facile de spécialiser certains serveurs pour l'échange d'un sous-ensemble d'objets-messages. Par exemple, dans le modèle de simulation, il est évident que si l'on augmente le nombre d'agents, (peu importe sur combien d'ordinateurs ils se trouvent), le serveur exécutant le *JavaSpaces* doit gérer un trafic croissant d'objets. Or, il serait très facile de fractionner le *JavaSpaces* en deux: un responsable des objets représentant les transactions monétaires (salaire, achat, taxe) et un autre responsable de tous les autres messages (nouvelle journée, journée terminée, publication des prix de vente, demande d'emploi, embauche, etc). À la limite, il serait possible de créer autant de *JavaSpaces* qu'il y a de types d'entrées échangées.

La version *JavaSpaces* du simulateur est par contre plus lourde que celle conçue avec *Infobus*. La première cause est reliée à la nature distribuée de l'environnement; le simulateur souffre des contraintes de latence et de délais réseau. L'atomicité transactionnelle des primitives de base des *JavaSpaces* (*read*, *write*...) contribue aussi à alourdir le système.

Conclusion

La simulation microanalytique adaptative, telle que décrite dans ce mémoire, est un outil d'économie computationelle valable; elle permet d'analyser le comportement d'une population d'agents dans un environnement macroéconomique avec une grande précision : au niveau de chaque membre y participant. De plus, les agents ont la capacité de s'adapter à leur environnement et ils sont à rationalité limitée : ces caractéristiques permettent à l'approche discutée dans ce mémoire d'être particulièrement utile pour modéliser des économies en transition.

Pour vérifier la validité de la technique, nous avons simulé un modèle dont l'équilibre stratégique est calculable. Les résultats obtenus concordent avec les prévisions théoriques, sauf dans quelques cas particuliers; ces erreurs sont attribuées à l'algorithme d'apprentissage simple utilisé par les agents ayant des capacités d'adaptation (dans le modèle de validation, les agents de type *entreprise*). De plus, nos résultats sont semblables à ceux obtenus par le simulateur du projet *Aspen*. Nous n'affirmons pas toutefois pouvoir prédire et expliquer les cycles économiques.

Nous sommes arrivés à la conclusion qu'il y a un compromis à faire entre précision des résultats et réalisme de la simulation; plus le nombre d'agents est élevé, plus la simulation représente bien l'environnement macroéconomique. Par contre, pour pouvoir exécuter des simulations ayant un nombre élevé d'agents, il faut soit distribuer la charge de calcul sur plusieurs ordinateurs, soit réduire la puissance de l'algorithme d'appren-

tissage. Étant donné que chaque agent possède son propre module d'apprentissage, les besoins en ressources informatiques (mémoire requise et puissance de calcul) de l'algorithme d'apprentissage utilisé sont critiques.

Il serait intéressant de poursuivre les simulations pour voir quelle précision pourrait être obtenue avec un algorithme d'apprentissage plus puissant que *GALCS*, en particulier l'algorithme *LCS* avec rétribution *bucket-brigade* et population de classifieurs, en utilisant l'environnement de simulation distribué *JavaSpaces* sur plusieurs ordinateurs.

Cet environnement est présentement réutilisé par Marcelin Joanis pour l'étude de l'impact des taxes sur la masse salariale des entreprises. Le modèle sera une évolution du modèle de validation et un marché du travail y sera ajouté : les entreprises auront la capacité de varier leur nombre d'employés en fonction de leur perception de l'environnement. Avec ce simple ajout, il sera possible de voir quels impacts ont les taxes sur la masse salariale des entreprises. Est-ce que le nombre d'employés d'une entreprise dépend de cette taxe? Est-ce que la taxe est assumée par le consommateur, en tout ou en partie, par le biais d'une augmentation des prix? C'est à ce genre de questionnement que l'environnement de simulation microanalytique adaptative permet de répondre.

Il est évidemment souhaité que les macroéconomistes adoptent cette approche de simulation et qu'elle devienne un outil d'usage courant en économie computationelle. Pour y parvenir, un meilleur échange de connaissances entre informaticiens et économistes sera nécessaire.

Bibliographie

[BB86] B. BERGMANN et R. BENNET. « A Microsimulated Transactions Model of the United States Economy ». *The Johns Hopkins University Press*, 1986.

[BPQA96] N. BASU, R.J. PRYOR, T. QUINT, et T. ARNOLD. « Development of Aspen: A microanalytic simulation model of the U.S. economy ». Rapport Technique SAND96-0434 UC905, Sandia National Laboratories, Février 1996.

[dB94] B. de BOER. « Classifier Systems, A useful approach to machine learning? ». Master's thesis, Leiden University, Août 1994.

[Dum95] R. DUMEUR. « Synthèse de comportements animaux individuels et collectifs par Algorithmes Génétiques ». Master's thesis, Université de Paris-8, 1995. Publié sur Internet à l'adresse : `http://www.ai.univ-paris8.fr/~renaud/publications/hthese/hthese.html`.

[Hol75] J. HOLLAND. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[Hol80] J. HOLLAND. « Adaptative Algorithms for Discovering and Using General Patterns in Growing Knowledge Bases ». *International Journal for*

Policy Analysis and Information Systems, 4(3):245–268, 1980.

[Koz90] J. KOZA. « Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems ». Rapport Technique STAN//CS-TR-90-1314, Computer Science Department, Stanford University, Juin 1990.

[Koz92] J. KOZA. *Genetic Programming*. MIT Press, 1992.

[LS97] G. LUGER et W. STUBBELFIELD. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison-Wesley, troisième édition, 1997.

[Mae97] P. MAES. Modeling Adaptive Autonomous Agents. Dans Christopher G. LANGTON, éditeur, *Artificial Life, an overview*, pages 135–162. MIT Press, 1997.

[MCM83] R MICHALSKI, J CARBONELL, et T MITCHELL. *Machine Learning: An Artificial Intelligence Approach*. Tioga Publishing Company, 1983.

[MF97] M. MITCHELL et S. FORREST. Genetic Algorithms and Artificial Life. Dans Christopher G. LANGTON, éditeur, *Artificial Life, an overview*, pages 267–289. MIT Press, 1997.

[MS92] MARIMON et SUNDER. « Indeterminacy of Equilibria in a Hyperinflationary World: Experimental Evidence ». Rapport Technique, University of Minnesota and Carnegie-Mellon University, 1992.

[OCW76] G. ORCUTT, S. CALDWELL, et R. WERTHEIMER. « Policy exploration through microanalytic simulation ». *The Urban Institute*, 1976.

[Pry98] R.J. PRYOR. « Developing Robotic Behavior Using a Genetic Programming Model ». Rapport Technique SAND98-0074 UC905, Sandia National Laboratories, Janvier 1998.

[Sar93] T. SARGENT. *Bounded Rationality in Macroeconomics*. Clarendon Press, Oxford, 1993.

[SB98] R. SUTTON et A. BASTO. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[Sun98a] Sun Microsystems. « *Infobus 1.1.1 Specification* », 1998. Disponible sur Internet à l'adresse `http://java.sun.com/beans/infobus/infobus-1.1.1.pdf`.

[Sun98b] Sun Microsystems. « *JavaSpaces Specification* », 1998. Disponible sur Internet à l'adresse `http://java.sun.com/products/javaspaces/specs/js.pdf`.

[Sun98c] Sun Microsystems. « *Transaction Specification* », 1998. Disponible sur Internet à l'adresse `http://java.sun.com/products/javaspaces/specs/txn-spec.pdf`.

[WWWK96] J. WALDO, G. WYANT, A. WOLLRATH, et S. KENDALL. « A Note On Distributed Computing ». Rapport Technique, Sun Microsystems Laboratories, 1996.

Annexe A

Configuration du simulateur

La configuration et le lancement du simulateur se font par le biais d'un script de démarrage codé en *JPython*. Par exemple, les variables exogènes du modèle y sont définies : elles seront, au moment du lancement de la simulation, placées dans l'objet babillard public (*ecosim.space.entries.BillboardEntry*). Voici un exemple de fichier de configuration permettant d'exécuter une version réduite du modèle de simulation.

```
#
# Chargement des libraires.
#
import sys
sys.add_package("ecosim.space")


from ecosim.space import BillBoardManagerImpl
from ecosim.space import Main
from jarray import array
from java.lang import String
```

```
#
# Creation des objets essentiels.
#
m = Main()
bbmi  = BillBoardManagerImpl()


#
# Variables exogènes de l'enchere
#
bbmi.putValue("q", "5")
bbmi.putValue("v", "0")
bbmi.putValue("t", "0.0")
bbmi.putValue("u", "0.16")


#
# Nombre d'individus "travailleurs"
#
bbmi.putValue("n_t", "500")


#
# Nombre d'employes par entreprise/gouvernement
#
bbmi.putValue("e", "100")


#
# Nombre de proprietaires d'entreprises
#
```

```
bbmi.putValue("p", "4")

#
# Nombre d'entreprises
#
bbmi.putValue("m", "4")

#
# Salaire d'un employe
#
bbmi.putValue("s", "100")

#
# Cout de fabrication d'une unité du produit par une entreprise
#
bbmi.putValue("c", "100")

#
# Duree de la simulation
#
bbmi.putValue("Iterations", "15000")

#
# Variable de l'algorithme d'apprentissage
#
bbmi.putValue("DeltaUp", "1")
bbmi.putValue("DeltaDown", "1")
```

```
#
# Nom des entreprises
#
#enames = array(["Enterprise_0", "Enterprise_1", "Enterprise_2", "Enter-
prise_3"], String)
#bbmi.putValue("EnterprisesNames", enames)


#
# Paramètres de débuggage
#
bbmi.putValue("ecosim.space.Entreprise_DEBUG", "true")
bbmi.putValue("ecosim.space.Individu_DEBUG", "false")
bbmi.putValue("ecosim.space.Gouvernement_DEBUG", "true")
bbmi.putValue("ecosim.space.Synchro_DEBUG", "true")


#
# Lancement du simulateur
#
m.start(bbmi)
```

Annexe B

Documentation des classes

La documentation[1] des classes créées pour les deux modèles est disponible à cet *URL*: `http://www.cirano.umontreal.ca/~trussarv/memoire/doc/Packages.html`

1. Documentation automatisée du genre "*javadoc*"