Moteur de programmation multi-agents

Johan Delouvée, Patrick Milin
encadrés par Vincent Rodin

21 Février 2008

Table des matières

1 Les systèmes multi-agents **4**
1.1 Introduction . . . 4
1.2 Systèmes multi-agents à l'UBO . . . 4

2 Présentation du projet **6**
2.1 Rappel du sujet . . . 6
2.2 Fonctionnalités attendues . . . 6

3 Analyse et conception **7**
3.1 Structures de données . . . 7
3.1.1 Sets et vecteurs . . . 7
3.1.2 Stockage des sets d'agents . . . 7
3.2 Ordonnancement . . . 7
3.3 Messages . . . 7

4 Mise en oeuvre **9**
4.1 Première version . . . 9
4.1.1 Implémentation . . . 9
4.1.1.1 Dictionnaires . . . 9
4.1.1.2 Classe SMA . . . 9
4.1.1.3 Ordonnancement . . . 10
4.1.1.4 Messages . . . 10
4.1.2 Prototypes des méthodes à utiliser . . . 10
4.1.2.1 Classe SMA . . . 10
4.1.2.2 Classe Agent . . . 11
4.1.3 Prototypes des méthodes à redéfinir . . . 11
4.1.3.1 Nouvelle classe Agent : MaClasseAgent . . . 11
4.1.3.2 Nouvelle classe Message : MaClasseMessage . . . 12
4.2 Améliorations . . . 12
4.2.1 Implémentation . . . 12
4.2.1.1 Arbres . . . 12
4.2.1.2 Impacts de la structuration en arbre . . . 12
4.2.2 Prototypes des méthodes à utiliser . . . 13
4.2.2.1 Scheduler . . . 13
4.2.2.2 Agent . . . 13
4.2.2.3 TypeTreeNode . . . 13
4.2.3 Prototypes des méthodes à redéfinir . . . 13
4.2.3.1 Nouvelle classe Agent : MaClasseAgent . . . 13
4.2.3.2 Nouvelle classe Message : MaClasseMessage . . . 13
4.2.3.3 Générateur de classes . . . 14
4.3 Comparatif des performances . . . 14
4.3.1 Gains espérés . . . 14
4.3.2 Gains constatés . . . 14
4.3.2.1 Avec la première version . . . 14
4.3.2.2 Avec la seconde version . . . 14
4.3.2.3 Conclusion . . . 15

5 Application : l'atelier flexible **16**
5.1 Introduction . . . 16
5.2 Machines à états . . . 16
5.2.1 Pièce (voir Figure 5.1 page 17) . . . 16
5.2.2 Machine (voir Figure 5.2 page 17) . . . 16
5.2.3 Fonctionnement . . . 17

6 Conclusion **19**
6.1 Bilan . . . 19
6.2 Critiques . . . 19
6.3 Améliorations possibles . . . 19

Table des figures

1.1 BioDyn, le logiciel développé par le projet EBV - Accueil 5
1.2 BioDyn, le logiciel développé par le projet EBV - Simulation 5

3.1 Ordonnancement aléatoire 8

4.1 AgentMap 9
4.2 TypeMap 10
4.3 TypeTreeNode 12

5.1 Automate à états pour une Pièce 17
5.2 Automate à états pour une Machine 17
5.3 Capture d'écran de l'Atelier flexible 18

Chapitre 1

Les systèmes multi-agents

1.1 Introduction

Pour Weiss (1999), un agent est une "entité computationnelle", comme un programme informatique ou un robot, qui peut être vue comme percevant et agissant de façon autonome sur son environnement. Un système multi-agents peut-être défini comme un ensemble d'agents situés dans un certain environnement, interagissant selon une certaine organisation, partageant des ressources communes et communicant entre eux. Le point clé des systèmes multi-agents réside dans la formalisation de la coordination entre les agents. La recherche sur les agents est ainsi une recherche sur la décision, le contrôle et la communication. Les systèmes multi-agents forment un type intéressant de modélisation de sociétés, et ont à ce titre des champs d'application larges, allant jusqu'aux sciences humaines. Ils ont notamment des applications dans le domaine de l'intelligence artificielle où leur structure peut permettre de réduire la complexité de la résolution d'un problème en divisant le savoir nécessaire en sous-ensembles, par l'association d'un agent à chacun de ces sous-ensembles et la coordination de l'activité de ces agents (Ferber, 1995). Ce qui importe c'est le comportement d'ensemble et non pas le comportement individuel. Des applications existent en physique des particules, en chimie, en biologie cellulaire, en éthologie, en sociologie et en ethnologie. Pour résoudre un problème complexe, il est en effet parfois plus simple de concevoir des programmes relativement petits (les agents) en interaction qu'un seul gros programme monolithique. L'autonomie des agents permet ici de simuler le comportement exact d'une entité et permet au système de s'adapter dynamiquement aux changement imprévus qui interviennent dans l'environnement.

1.2 Systèmes multi-agents à l'UBO

L'Université de Bretagne Occidentale (UBO) est très active dans le domaine des systèmes multi-agents. Le Projet «Écosystémique et Biologie Virtuelles» (EBV), mené en collaboration avec l'Ecole Nationale d'Ingenieurs de Brest (ENIB), a pour but de développer une recherche interdisciplinaire dans les domaines de la bioinformatique et de la modélisation d'écosystèmes en utilisant des modélisations centrées sur des individus, c'est à dire des systèmes multi-agents. Cette approche permet de simuler des phénomènes biologiques complexes de manière plus naturelle que les simulations numériques traditionnelles, et surtout d'apporter des éléments de réponses là où les approches classiques peuvent échouer.

Les travaux de l'équipe s'articulent autour de quatre axes principaux : modélisation de systèmes physiologiques humains, modélisation d'écosystèmes, exploration du fonctionnement du vivant par la modélisation, la simulation et par des approches algorithmiques et heuristiques, et validation mathématique de modèles individu-centrés. Ils permettent de simuler des phénomènes biologiques dans les domaines de l'allergologie, de la cancérologie ou de l'hématologie, ainsi que des écosystèmes incluant des modèles écologiques et/ou économiques.

Ces travaux ont abouti notamment à la réalisation d'un logiciel de modélisation et de simulation nommé SimBioDyn. Ce logiciel permet de modéliser et de simuler des processus biologiques complexes, essentiellement des systèmes microscopiques, et a par exemple été utilisé dans le cadre de la modélisation de réactions enzymatiques, de migration de fibroblastes, d'intéraction entre phages ou de réponses immunitaires. SimBioDyn connait quelques évolutions comme BioDyn_NET, NetBioDyn, eBioDyn, Reaxels, ...

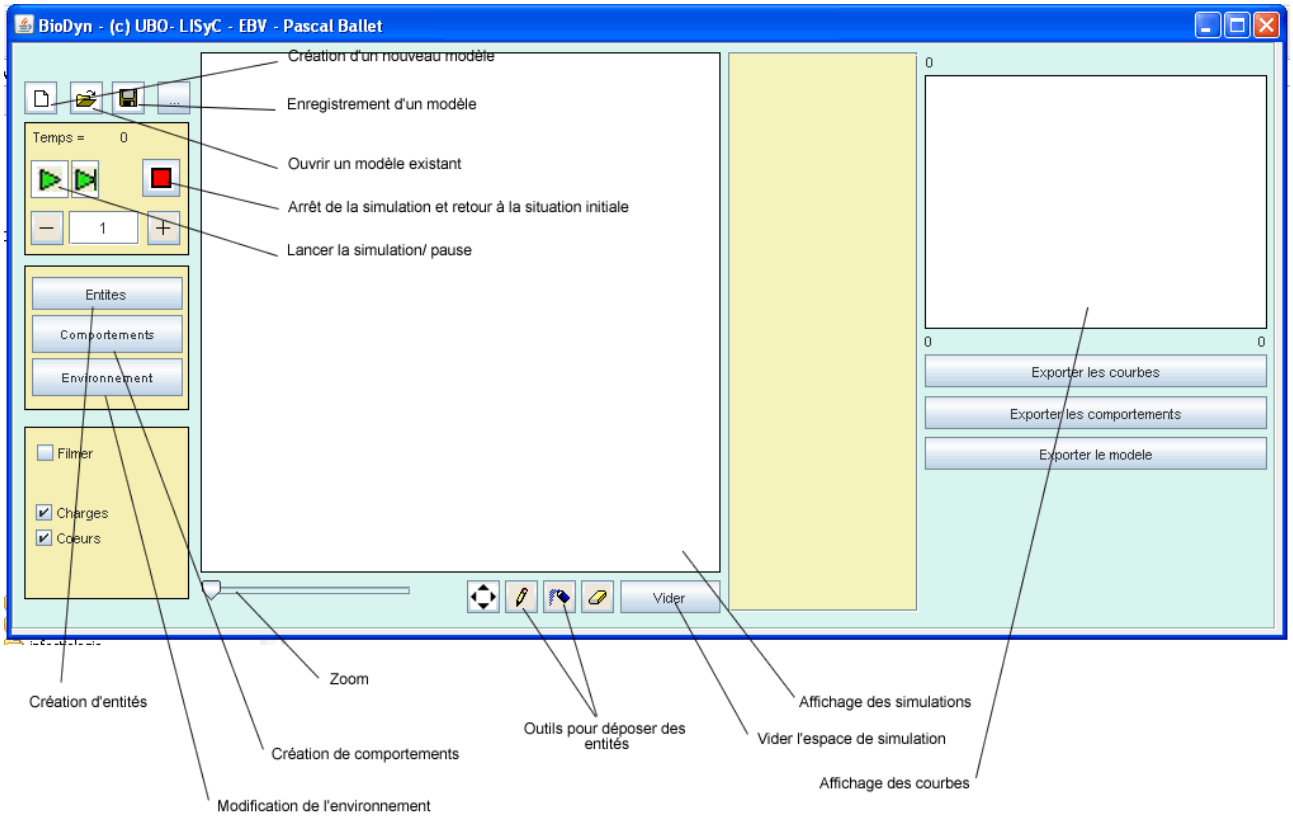

FIG. 1.1 – BioDyn, le logiciel développé par le projet EBV - Accueil

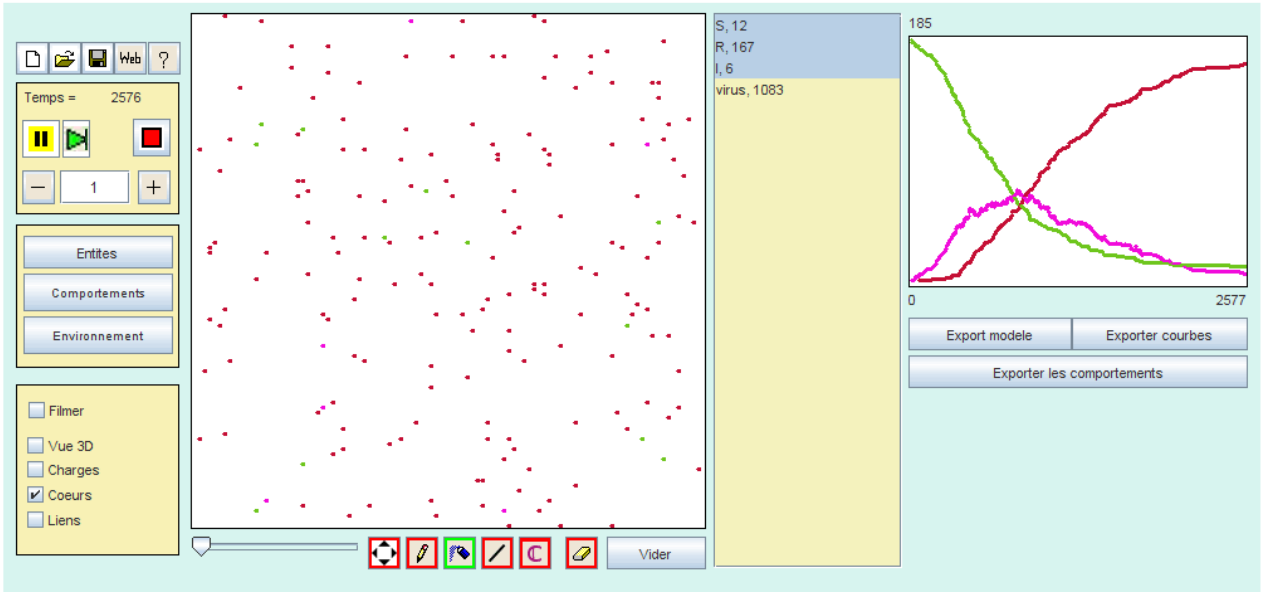

FIG. 1.2 – BioDyn, le logiciel développé par le projet EBV - Simulation

Chapitre 2

Présentation du projet

2.1 Rappel du sujet

Le but de ce projet est de réaliser un ensemble de classes C++ permettant de programmer facilement une application multi-agents. Il est encadré par Vincent Rodin (Vincent.Rodin@univ-brest.fr) et se déroule au département informatique de l'UFR Sciences.

Dans un premier temps, le travail consiste à programmer un ordonnanceur équitable permettant d'exécuter automatiquement le comportement d'agents. Ensuite, lors d'une deuxième étape, un certain nombre de classes permettant la communication par messages entre agents sont à programmer.

Plusieurs domaine d'applications peuvent être envisagés comme, par exemple, le traitement d'images, la simulation d'un atelier flexible, ou toute autre application résolvable par l'utilisation de systèmes multi-agents.

2.2 Fonctionnalités attendues

Nous devons permettre à un programmeur de créer une application multi-agents le plus simplement possible. Il doit donc être possible de :

- Créer et détruire un agent à tout moment.
  Les agents ont un cycle de vie, allant de leur naissance à leur mort, plus ou moins violente.
- Nommer automatiquement les agents.
  Afin d'avoir des agents nommés selon leur classe et leur ordre de création (ex : Piece1, Machine4, ...)
- Récupérer le nom et la classe de l'agent.
  Afin de les identifier facilement parmi tous les autres agents.
- Tester si un agent est une instance d'une classe donnée.
  Afin de l'identifier facilement également.
- Récupérer tous les agents d'une classe donnée.
  Afin de faire un traitement spécifique à une classe d'agents.
- S'assurer de l'existence d'un agent.
  Afin d'éviter de lancer des traitements sur un agent inexistant.
- Ordonnancer les agents séquentiellement ou aléatoirement.
  Afin qu'ils vivent dans un ordre séquentiel ou aléatoire.
- Envoyer un message d'un agent à un autre.
  Afin de leur permettre de communiquer.
- Envoyer un message en "broadcast" d'un agent à une liste de diffusion.
  Afin d'envoyer un message à plusieurs agents à la fois.
- Abonner / Désabonner un agent d'une liste de diffusion.
  Afin de recevoir ou non certains types de message envoyés en broadcast.
- Tester si un message est une instance d'une classe de message donnée.
  Afin de tester le type de message reçu et appliquer le traitement associé.

Chapitre 3

Analyse et conception

3.1 Structures de données

3.1.1 Sets et vecteurs

Les fonctionnalités demandées préconisent l'emploi de certaines structures de données par rapport à d'autres. Il serait par exemple maladroit d'utiliser des vecteurs (collection ordonnée d'éléments), dont le principal avantage est la manipulation du dernier élément, alors que l'on peut avoir besoin de détruire n'importe quel élément à tout moment. Le set (collection ordonnée sans doublons) semble donc plus approprié, car il dispose de méthodes d'accès et de destruction sur n'importe quel élément moins coûteuses, en moyenne, que celles des vecteurs.

3.1.2 Stockage des sets d'agents

Pour manipuler les agents et faciliter notamment les tests d'appartenance à une classe, chaque classe d'agent se voit associée à un set de ses instances. Il nous faut alors pouvoir accéder rapidement à ces sets d'agents. Des structures déjà définies comme le map (dictionnaire associatif - une clé -> une valeur) semblent, à priori, être des solutions satisfaisantes. En effet, dans un map, une clé, unique, permet d'identifier une valeur lui étant associée. Ici, la clé sera le nom de la classe, et la valeur sera le set d'agents de cette classe. Comme ce sont des structures de données associatives, elles sont conçues pour être très efficace pour l'accès par clé.

3.2 Ordonnancement

Nous devons élaborer une classe ordonnanceur qui permet à l'utilisateur de faire vivre ses agents dans un ordre prédéfini ou aléatoire (et modifié à chaque cycle de vie des agents).

Pour l'ordonnancement aléatoire (voir Figure 3.1 page suivante), on considère deux tableaux d'agents, le premier contenant au départ tous les agents à ordonnancer et le second ne contenant rien. On tire aléatoirement un agent du premier tableau, et on le place dans le second à une position arbitraire (en première ou dernière position). Ceci fait, on supprime l'agent du premier tableau et on réitère l'opération jusqu'à ce que celui-ci soit vide. Ensuite, le second tableau, rempli de tous les agents, est rendu afin de pouvoir faire vivre les agents dans l'ordre défini.

3.3 Messages

Chacun des agents doit être capable d'émettre et de recevoir des messages normalisés. Un message sera donc caractérisé, au minimum, par l'agent qui l'envoit (sender). Pour l'émission, deux choix s'offre aux agents : ils peuvent soit envoyer un message à un agent donné, soit envoyer un message en broadcast à tous les agents abonnés à recevoir le type de message envoyé. Chaque classe de message s'occupe donc de stocker un tableau d'agents s'étant abonné à cette classe, et lors d'un broadcast, s'occupe d'aller récupérer tous ces agents afin de leur envoyer le message.

Les messages étant susceptibles d'être stockés par le receveur pour une utilisation ultérieure, il est nécessaire de ne leur envoyer qu'une copie du message, qu'ils détruiront une fois celui-ci devenu inutile, afin d'assurer que tous les messages créés soient détruits et éviter une fuite de mémoire.

Enfin, afin de permettre aux agents de recevoir plusieurs messages à la fois, chacun d'entre eux aura à sa disposition une boite aux lettres, c'est à dire un tableau de messages reçus mais non lus.

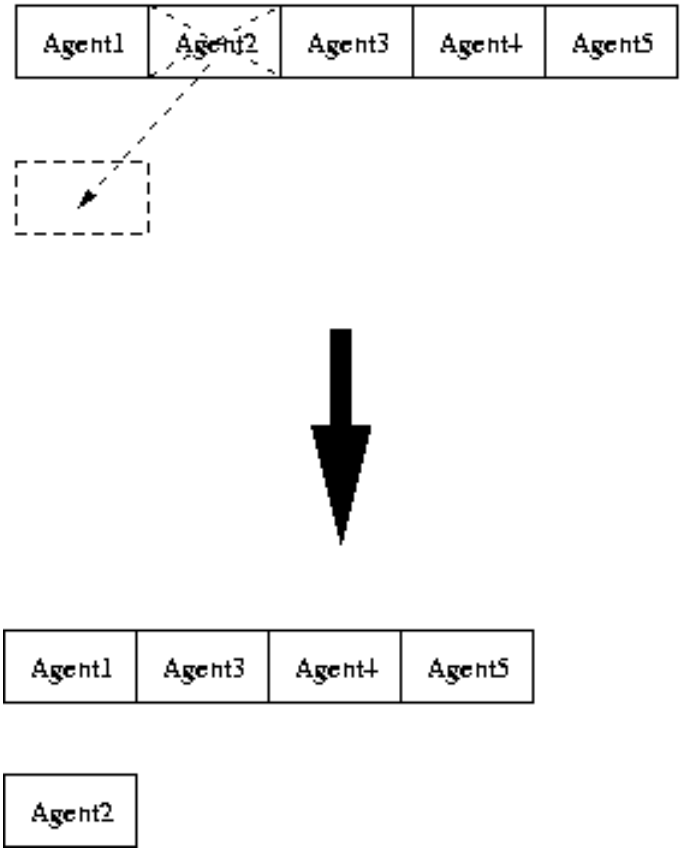

FIG. 3.1 – Ordonnancement aléatoire

Chapitre 4

Mise en oeuvre

4.1 Première version

4.1.1 Implémentation

4.1.1.1 Dictionnaires

Les agents créés sont stockés créés dans un map particulier implémenté spécialement pour ce projet (classe AgentMap - voir Figure 4.1). Il s'agit d'une association entre les noms des classes dérivées d'Agent (les clés), et le set d'instances de ces classes (les valeurs). Cette instance d'AgentMap est déclarée statiquement dans la classe Agent. Agent elle-même n'y figure pas car il s'agit d'une classe abstraite et n'a donc aucune instance. L'utilisateur doit spécialiser la classe Agent pour créer ses propres agents.

La hiérarchie de classe a été représentée par un map particulier également (classe TypeMap - voir Figure 4.2 page suivante). Ce map associe le nom d'une classe (les clés) au nom de sa classe mère (les valeurs).

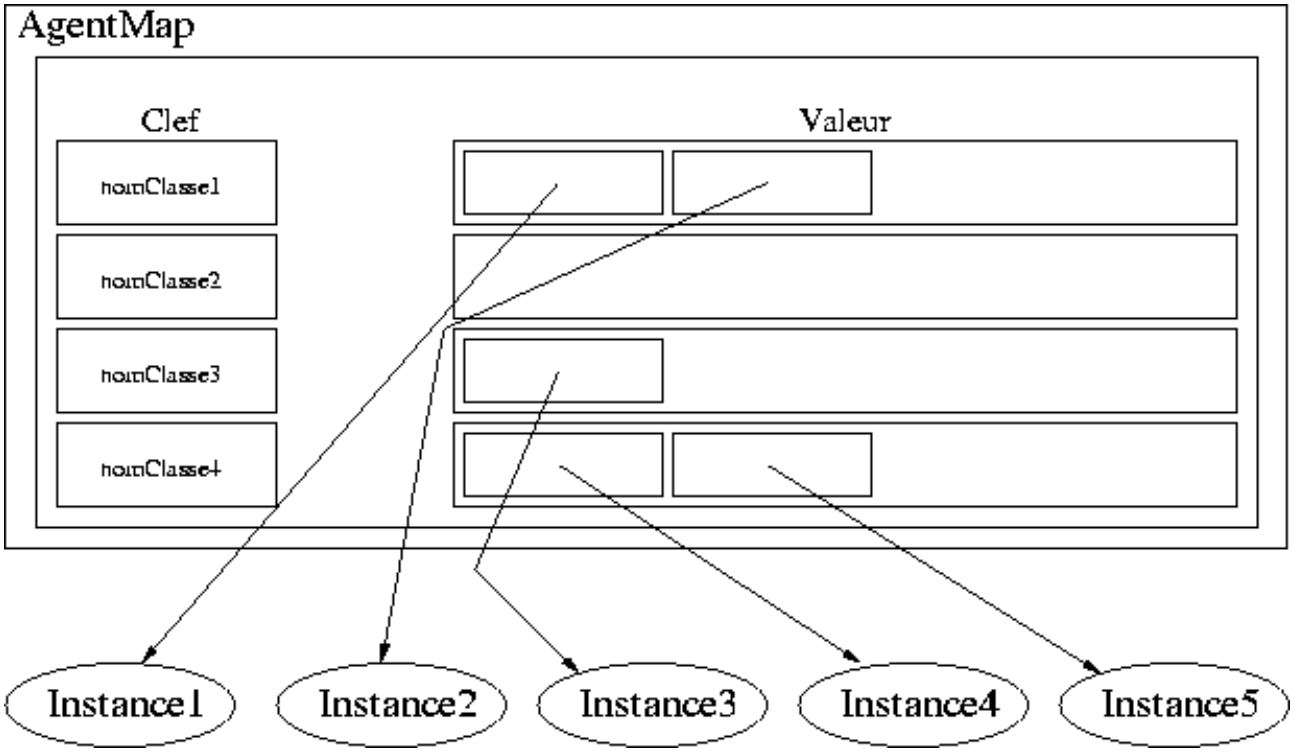

FIG. 4.1 – AgentMap

4.1.1.2 Classe SMA

Cette classe est un peu particulière dans le sens où elle ne joue aucun rôle dans le système multi-agents lui-même. Son utilité est de cacher les mécanismes et les composants de notre moteur. Excepté pour la création des objets, on ne manipule que cette classe dans le Main du programme.

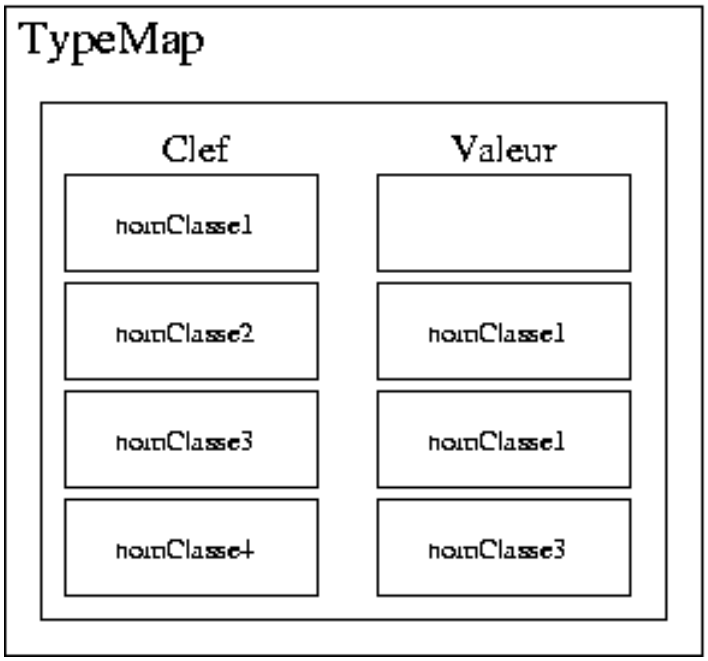

FIG. 4.2 – TypeMap

4.1.1.3 Ordonnancement

Pour l'ordonnancement aléatoire, une sous-classe RandomScheduler est définie. Comme nous avons besoin de supprimer un élément situé à une position aléatoire, il est nécessaire d'utiliser des sets et non des vecteurs. Il aurait également été possible d'utiliser un set pour le tableau initial et un vecteur pour le tableau ordonnancé, ce qui nous aurait permis d'utiliser la méthode `push_back`, très efficace pour insérer en dernière position, mais ce tableau étant retourné par la fonction, des problèmes de compatibilité avec le reste du programme, qui n'utilise que des sets, seraient apparus.

Le problème majeur posé par les sets est que les éléments sont automatiquement triés (dans notre situation, par nom d'agent). C'est une chose qu'il faut absolument éviter ici puisque l'on veut conserver l'ordre aléatoire. Le tri est donc neutralisé en redéfinissant la structure `classcomp`.

4.1.1.4 Messages

Les messages sont définis grâce à la classe abstraite Message. Chaque message est donc normalisé, car il s'agit d'une instance d'une sous-classe de Message. La hiérarchie des messages est, comme pour les agents, retranscrite via un TypeMap.

Lors de l'émission d'un message, simple ou par broadcast, celui-ci a pour mission de se copier (donc copier l'émetteur) autant de fois que nécessaire et de s'envoyer aux agents désignés à le recevoir.

Pour stocker les souscriptions aux différents broadcast, on utilise un map ayant pour clé le type de message, et pour valeur un set d'agents inscrits à ce type de message.

Enfin, afin de stocker les messages entrants, chaque Agent dispose d'une queue, semblable à une pile FIFO (First-in, First-out), permettant d'assurer, à chaque récupération d'un nouveau message, de récupérer celui arrivé en premier dans la boite aux lettres.

4.1.2 Prototypes des méthodes à utiliser

Dans cette version, l'utilisateur aura à utiliser des méthodes ne provenant que de deux classes : SMA et Agent.

4.1.2.1 Classe SMA

- `vector<string> getSubclasses(string type)`
  - Retourne le set contenant le nom des sous-classes de la classe `type`.
  - Retourne `NULL` si la classe `type` n'existe pas.
- `set<Agent *> getType(string type)`
  - Retourne les agents instances de la classe `type`.
  - Retourne `NULL` si la classe `type` n'existe pas.

- `set<Agent *> getAllAgents()`
  - Retourne tous les agents.
- `void cycling(int n, Scheduler * scheduler)`
  - Ordonnance les agents selon le scheduler spécialisé choisi.
  - Fait vivre les agents dans l'ordre imposé par le scheduler.
  - S'exécute `n` fois.

4.1.2.2 Classe Agent

- `void create()`
  - Créé une instance de la classe Agent.
- `void create(int n)`
  - Créé `n` instance(s) de la classe Agent.
- `string getName()`
  - Retourne le nom de l'agent.
- `string getClassName()`
  - Retourne la classe de l'agent.
- `string getParent()`
  - Retourne la superclasse de l'agent.
- `bool isA(string type)`
  - Teste si l'agent est une instance de la classe `type`, d'une de ses classes filles ou mères.
- `static bool exist(Agent * agent)`
  - Teste si `agent` existe bien dans le sytème.
- `static string getSMA()`
  - Retourne le SMA associé à la classe Agent (afin d'utiliser toutes les méthodes définies ci-dessus)
- `static void deleteAgent(string name)`
  - Supprime l'agent ayant le nom `name`.
- `static void deleteAgentType(string type)`
  - Supprime tous les agents instances de la classe `type`.
- `static void deleteAllAgents()`
  - Supprime tous les agents.
- `void sendTo(Agent * agent, Message * msg)`
  - Envoit le message `msg` à l'agent `agent`.
- `void broadcast(Message * msg)`
  - Envoit le message `msg` en broadcast.
- `int getNbMessage()`
  - Renvoit le nombre de messages présents dans la boite aux lettres.
- `Message * getNextMessage()`
  - Renvoit le premier message présent dans la boite aux lettres.
- `void setSentivity(string messageType)`
  - Rend l'agent sensible aux broadcast de type `messageType`.
- `void removeSentivity(string messageType)`
  - Rend l'agent insensible aux broadcast de type `messageType`.

4.1.3 Prototypes des méthodes à redéfinir

Afin de créer une nouvelle classe d'Agent ou de Message, quelques méthodes sont à redéfinir dans ces sous-classes afin d'assurer le bon fonctionnement du moteur. Ces redéfinitions ne sont généralement que de simples copier/coller, en changeant juste le nom de la classe concernée.

4.1.3.1 Nouvelle classe Agent : MaClasseAgent

- `MaClasseAgent()`
  - Constructeur de la nouvelle classe.
- `static void create()`
- `static void create(int n)`
- `string getClassName()`
- `string getParent()`
- `void live()`
  - Méthode appelée à chaque cycle pour faire vivre un agent.

4.1.3.2 Nouvelle classe Message : MaClasseMessage

- `MaClasseMessage()`
  - Constructeur de la nouvelle classe.
- `void sendTo(Agent * receiver)`
  - Envoyer le message courant à l'agent `receiver`.
- `void getType()`
- `void getParent()`

4.2 Améliorations

4.2.1 Implémentation

4.2.1.1 Arbres

La représentation de la hiérarchie de classes est peu satisfaisante dans notre première implémentation. En effet, le map n'est pas fait pour représenter une structure d'arbre et il est difficile de remonter vers le noeud parent. La deuxième version du moteur voit donc les dictionnaires disparaître au profit d'une classe TypeTreeNode (voir Figure 4.3).

Cette classe regroupe les préoccupations de hiérarchisation des classes et de stockage des instances. On associe statiquement un noeud unique (on s'en assure par l'application d'un pattern Singleton) à chaque classe. Ce noeud "contient" le set d'instances, le set de classes dérivées mais aussi la classe parente.

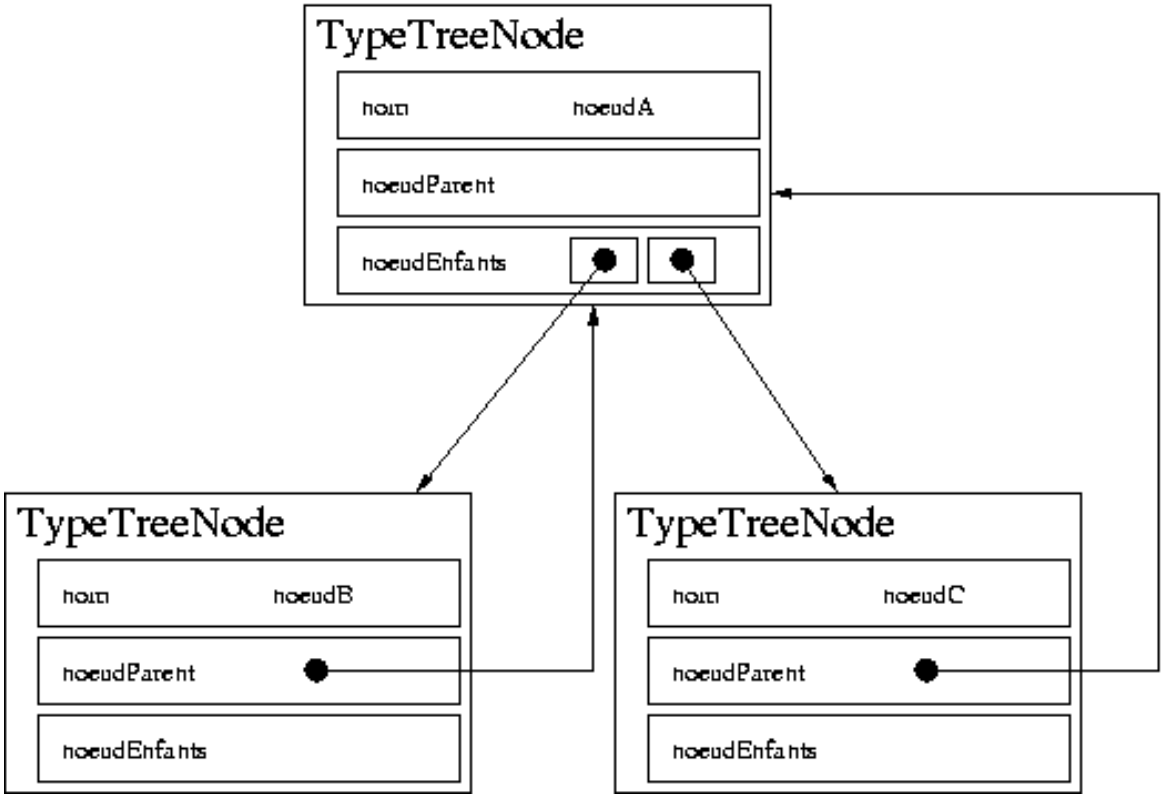

FIG. 4.3 – TypeTreeNode

4.2.1.2 Impacts de la structuration en arbre

Il est beaucoup plus facile de se déplacer dans la hiérarchie de classe avec cette version. Cependant, beaucoup de méthodes sont dépendantes du TypeTreeNode associé à la classe et plus de code doit donc être réécrit pour pouvoir y accéder.

Chaque classe d'Agent et de Message disposera d'un TypeTreeNode propre à elle-même, en remplacement du TypeTree de la première version.

Cette structuration nous permet également de faire disparaitre la classe SMA, dont le rôle est dorénavant assuré par les TypeTreeNode.

4.2.2 Prototypes des méthodes à utiliser

Dans cette version, l'utilisateur aura à utiliser des méthodes provenant des classes Agent et TypeTreeNode, et une méthode provenant de la classe Scheduler.

4.2.2.1 Scheduler

- `void cycle(set<Agent *> agents, int n)`
  - Ordonnance les `agents` selon le scheduler spécialisé choisi.
  - Fais vivre les `agents` dans l'ordre imposé par le scheduler.
  - S'exécute `n` fois.

4.2.2.2 Agent

- `void create()`
- `void create(int n)`
- `string getName()`
- `string getType()`
- `static TypeTreeNode * getTypetreenode()`
  - Retourne le TypeTreeNode associé à la classe.
- `static set<Agent *> getAgents()`
  - Retourne les agents instances de la classe et de ses classes filles.
- `static void eraseAgents()`
  - Supprime tous les agents.
- `bool isA(string type)`
- `static bool exist(Agent * agent)`
- `void sendTo(Agent * agent, Message * msg)`
- `void broadcast(Message * msg)`
- `int getNbMessage()`
- `Message * getNextMessage()`
- `void subscribe(TypeTreeNode *)`
- `void unsubscribe(TypeTreeNode *)`

4.2.2.3 TypeTreeNode

- `static void eraseAgent(Agent* agent)`
  - Supprime l'agent `agent`.
- `static void eraseAgent(string name)`
  - Supprime l'agent ayant le nom `name`.

4.2.3 Prototypes des méthodes à redéfinir

Afin de créer une nouvelle classe d'Agent ou de Message, plus de méthodes que précédemment sont à redéfinir dans ces sous-classes afin d'assurer le bon fonctionnement du moteur. Ces redéfinitions ne sont généralement que de simples copier/coller, en changeant juste le nom de la classe concernée.

4.2.3.1 Nouvelle classe Agent : MaClasseAgent

- `MaClasseAgent()`
- `static set<Agent *> getAgents()`
- `static set<Agent *> getTypetreenode()`
- `string getType()`
- `void create()`
- `void create(int n)`
- `void eraseAgents()`
- `void isA(string type)`
- `void live()`

4.2.3.2 Nouvelle classe Message : MaClasseMessage

- `MaClasseMessage()`
- `static set<Agent *> getSubscribers()`
  - Retourne les agents inscrits pour recevoir ce type de messages.

- `string getType()`
- `string getTypetreenode()`
- `void broadcast()`
- `void sendTo()`
- `void isA(string type)`

4.2.3.3 Générateur de classes

Afin de faciliter le travail de l'utilisateur, nous avons développé une petite application C++ permettant de générer automatiquement les fichiers .h et .cpp avec les méthodes nécessaires, et définissant une nouvelle classe d'Agent ou de Message selon son nom et le nom de sa classe mère.

Il reste toutefois à compléter les méthodes "live" de chaque classe ainsi générée., avec le comportement propre à leurs instances.

4.3 Comparatif des performances

4.3.1 Gains espérés

La facilité pour remonter dans l'arbre laisse présager des gains sur la méthode isA. Ce gain peut sembler minime au premier abord. Il ne faut pas oublier que c'est l'utilisateur qui implémente ses propres agents et les méthodes live correspondantes. Par conséquent, si un utilisateur venait à utiliser isA dans une méthode live, isA serait alors appelée autant de fois que le nombre d'instances de cette classe multiplié par le nombre de cycles à effectuer. L'écart de performance peut donc atteindre des proportions non négligeables.

4.3.2 Gains constatés

Sur un exemple contenant trois classes : des Machines et des Pièces, sous-classes d'Agent, et des Piecettes, sous-classe de Pièce. A chaque cycle, une Pièce (et une seule) envoit un message en broadcast, qui est reçu et lu par toutes les Machines. Après le dernier cycle, toutes les Pièces sont détruites. Avec 250 Machines, 750 Pièces, et 1000 Piècettes (soit 2000 agents en tout), et un ordonnancement aléatoire à chaque cycle, voici les résultats obtenus.

4.3.2.1 Avec la première version

1. Sur 100 cycles.
   Creation des objets : 0 seconde
   Duree totale des cycles : 12 secondes
   Destruction des objets : 0 seconde

2. Sur 1000 cycles.
   Creation des objets : 0 seconde
   Duree totale des cycles : 120 secondes
   Destruction des objets : 0 seconde

3. Sur 10000 cycles.
   Creation des objets : 0 seconde
   Duree totale des cycles : 1240 secondes
   Destruction des objets : 0 seconde

4.3.2.2 Avec la seconde version

1. Sur 100 cycles.
   Creation des objets : 0 seconde
   Duree totale des cycles : 4 secondes
   Destruction des objets : 0 seconde
2. Sur 1000 cycles.
   Creation des objets : 0 seconde
   Duree totale des cycles : 42 secondes
   Destruction des objets : 0 seconde

3. Sur 10000 cycles.
   Creation des objets : 0 seconde
   Duree totale des cycles : 429 secondes
   Destruction des objets : 0 seconde

4.3.2.3 Conclusion

On peut remarquer que la seconde version, pour cet exemple donné, est environ trois fois plus rapide que la précédente, le gain est donc conséquent et remarquable. Malgré le plus grand travail demandé à l'utilisateur, nous avons donc décidé d'opter pour cette version pour le développement de l'application présentée dans le chapitre suivant.

Il est également clairement visible que les temps nécessaires à la création et à la destruction d'objets sont non significatifs, et n'auront pas à être éventuellement testés ultérieurement.

Chapitre 5

Application : l'atelier flexible

5.1 Introduction

Dans un atelier flexible, des pièces doivent être traitées par une séquence de machines de divers types. Comme il existe généralement plusieurs machines de chaque type, le trajet de deux pièces ayant une séquence identique peut être totalement différent.

> Exemple : Considérons 3 machines de type A (A1, A2, A3), 2 machines de type B (B1, B2) et 5 machines de type C (C1, C2 , C3, C4, C5). Une pièce ayant une séquence A, B, C peut très bien suivre les deux parcours suivants : A1, B1, C1 et A2, B2, C2 (ou tout autre combinaison A, B, C...).

La séquence de chaque pièce est déterminée aléatoirement à la création de celle-ci. Le premier travail consistait à spécialiser la classe Obj2D fournie, permettant de représenter un objet en deux dimensions, afin d'y ajouter des notions de cinétique dans le but de permettre aux objets de se déplacer facilement dans le plan. Ensuite, une classe héritant à la fois de notre classe Agent et de cette spécialisation d'Obj2D (Or2DEntity) est définie (Holon), et sera ensuite spécialisée pour créer les Pièces et les Machines nécessaires. Nous obtenons alors des agents representés en 2D et pouvant se déplacer dans le plan.

5.2 Machines à états

5.2.1 Pièce (voir Figure 5.1 page suivante)

Lors de son initialisation (INIT), une pièce P1 entre par la porte d'entrée de l'atelier, puis si il reste des types de machine à visiter dans sa séquence, envoit en broadcast un message indiquant de quel type de machine elle a besoin et se met en attente (CHOICE). Sinon, elle quitte l'atelier (EXIT). Les machines s'étant abonnées au type de message en question, si elle sont en fonctionnement et non réservées par une autre pièce, répondent favorablement à la demande de P1. Ensuite, P1 récupère toutes les réponses favorables reçues, et calcule quelle machine ayant répondu est la plus proche d'elle (CHOICE). Après confirmation de la réservation mutuelle (CONFIRM), P1 commence alors à se diriger vers la machine, en continuant à chaque cycle d'envoyer des messages pour vérifier que la machine choisie est toujours en fonctionnement (TRANSIT). Arrivée à destination, la pièce le signale à la machine (READY), puis est traitée par celle-ci (BUSY). Enfin, P1 enlève de sa séquence le type de machine courant (END) et retourne à son état d'initialisation (INIT).

5.2.2 Machine (voir Figure 5.2 page suivante)

Concernant les machines, une machine M1 commence par s'initialiser en, notamment, se rendant disponible à recevoir les requêtes liées à son type (INIT). Elle se met ensuite en attente de recevoir une de ces requêtes (CHOICE). Une fois reçue, elle envoit une confirmation, et se rend insensible à toute autre requête (CHOICE). Après confirmation de la réservation mutuelle (CONFIRM), M1 continue à chaque cycle de répondre aux messages envoyés par la Pièce choisie afin de vérifier son bon fonctionnement (TRANSIT). Quand la Pièce arrive à proximité, M1 reçoit un message spécial, initialise le temps de travail nécessaire (READY), puis traite la pièce (BUSY) avant de revenir à son état initial (INIT).

Tout au long de ces deux comportements, des timeout permettent de renvoyer les agents à leur état d'initialisation, afin d'éviter tout blocage.

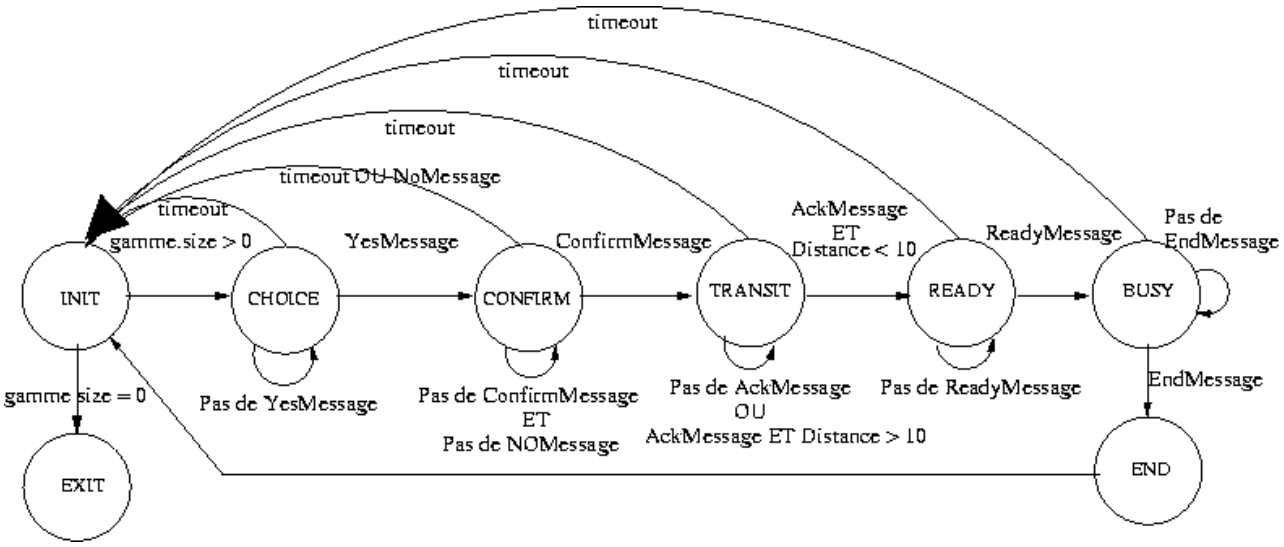

FIG. 5.1 – Automate à états pour une Pièce

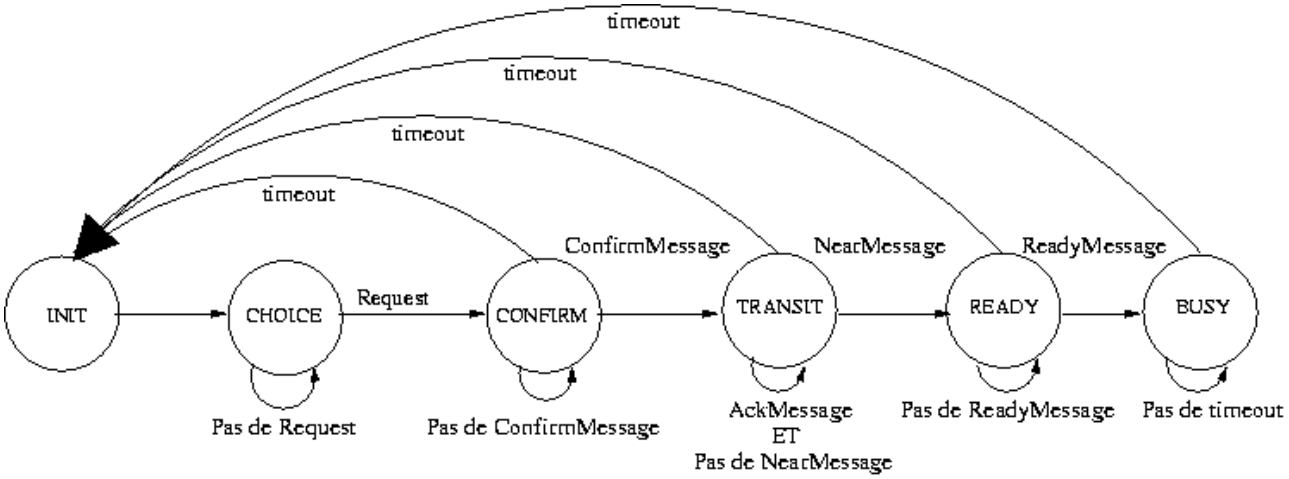

FIG. 5.2 – Automate à états pour une Machine

5.2.3 Fonctionnement

Pour faire fonctionner l'application, il suffit, à chaque cycle de vie de l'application, d'ordonnancer et de faire vivre tous les agents.

Des événements peuvent également être déclenchés par pression sur certains boutons :

- Sur l'Atelier :
  - "A" permet de créer une Machine de type A (placée aléatoirement dans l'atelier).
  - "B" permet de créer une Machine de type B (placée aléatoirement dans l'atelier).
  - "C" permet de créer une Machine de type C (placée aléatoirement dans l'atelier).
  - "P" permet de créer une Pièce.
  - "H" permet d'afficher l'aide dans le terminal.
- Sur une Machine :
  - "F1" permet d'afficher le nom et l'état (OK / HS) d'une machine dans le terminal.
  - "F2" permet de changer l'état d'une machine (OK / HS).
  - "H" permet d'afficher l'aide dans le terminal.

Sur la figure 5.3 page suivante, une capture d'écran de l'Atelier flexible, avec trois pièces et trois machines. On constate que deux machines sont réservées, et que la troisième pièce est en attente de la libération d'une de ces machines.

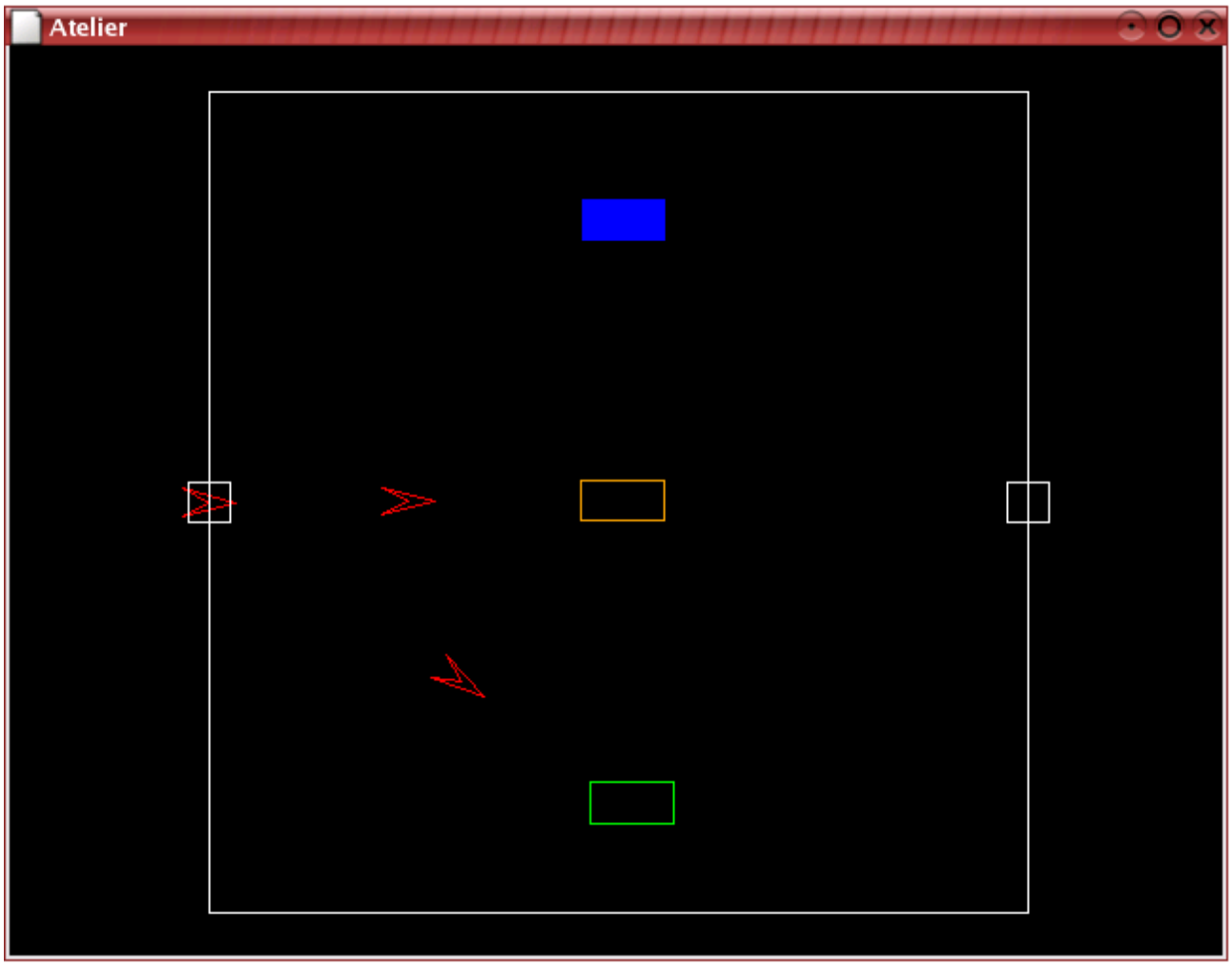

FIG. 5.3 – Capture d'écran de l'Atelier flexible

Chapitre 6

Conclusion

6.1 Bilan

Les fonctionnalités attendues ont toutes été implémentées et un exemple d'application, l'atelier flexible, a été réalisé. La simplicité d'utilisation est restée une préoccupation constante tout au long des développements. L'aspect performance a également été mis en avant lors d'un deuxième cycle de développement durant lequel le moteur de programmation a subi de profonds changements. Les gains apportés sont conséquents.

6.2 Critiques

Etant "novices" en C++, divers aspects techniques ont accaparé notre attention au détriment de l'aspect méthodologique. Une meilleure maîtrise du langage nous aurait permis d'être plus "agiles" dans le développement avec une approche basée sur des tests unitaires et/ou fonctionnels.

Cette absence de tests est des plus problématiques. En effet, il est impossible de garantir à 100% le fonctionnement de l'application sans des jeux de tests unitaires et/ou de tests fonctionnels basés sur des scénarios d'utilisation.

6.3 Améliorations possibles

Dans la deuxième version, la classe Agent devrait être enrichie afin que l'utilisateur n'ait plus du tout à utiliser la classe TypeTreeNode. Concrètement, il faut ajouter des méthodes supplémentaires s'interfaçant elles-mêmes avec celles de TypeTreeNode.

Quelques fonctionnalités supplémentaires sont également envisageables, comme le fait de renommer un agent. Il faut alors vérifier au préalable que ce nom n'est pas déjà attribué car un agent peut être identifié par son nom. Un "doublon", même s'il ne provoquerait sans doute pas un crash de l'application, entraînerait des dysfonctionnements (notamment si une suppression par nom d'Agent est demandée). Cela ne fait que confirmer la nécessité des tests. D'autres types d'ordonnanceurs peuvent également être envisagés, notamment avec un système de priorités.

L'idée de développer un outil (ressemblant à celui permettant à un utilisateur de créer très facilement ses propres classes d'agent) pour créer des interfaces graphiques 2D a également été évoquée. Plus clairement, les outils que nous avons développés permettent de créer les classes du modèle. Il pourrait alors être intéressant d'en faire de même pour les aspects vue et contrôleur. Cela reste cependant plus difficile à réaliser car les options disponibles sont largement plus nombreuses.

Bibliographie

[STL1] C++ Standard Template Library - http://www.cppreference.com/index.html

[STL2] STL Containers - http://www.cplusplus.com/reference/stl/

[SMA1] Système multi-agents sur Wikipedia - http://fr.wikipedia.org/wiki/Syst%C3%A8me_multi-agents

[SMA2] Les Systèmes multi-agents - http://cormas.cirad.fr/fr/demarch/sma.htm

[EBV] Projet «Écosystémique et Biologie Virtuelles» - http://www.lisyc.univ-brest.fr/projets_recherche/ebv.php

[BioDyn] BioDyn - Pascal Ballet - http://pagesperso.univ-brest.fr/~ballet/pages/0.html

[NetBioDyn1] NetBioDyn - Système Multiagent pour les Systèmes Complexes - http://netbiodyn.tuxfamily.org/

[NetBioDyn2] Didacticiel pour NetBioDyn - Alain Pothet - http://www.lisyc.univ-brest.fr/projets_recherche/ebv.ph

[Oris] Documentation Oris 2.1 - http://www.enib.fr/~harrouet/data/oRisDocHtml/index.html