DEA Systèmes
Informatiques
Répartis
2002-2003

Mémoire de DEA

Interopérabilité des mondes virtuels

Nicolas Roard

– Remerciements –

Je voudrais d'abord remercier M.Michel Soto, qui a toujours gardé son calme et sa bonne humeur, et m'a permis de faire un stage sur un sujet de recherche original et intéressant.

Je voudrais aussi remercier tous ceux qui ont su me remonter le moral par moment cette année : mes parents, sébastien, amandine, magalie, brice, doud, emric, kim, tian, marie, max, yean et les autres !

– Résumé –

Ce rapport présente des solutions permettant d'améliorer l'interopérabilité entre des mondes virtuels, et en particulier en ce qui concerne le problème de la migration d'entitées virtuelles dans des mondes virtuels étrangers.

Pour cela, deux grands axes sont utilisés :

– utilisation d'un paradigme de modélisation des actions et réactions original, le modèle d'*influence/réaction* qui vient de la recherche sur les agents logiciels
– utilisation d'algorithmes de classification conceptuelle, qui permettent de travailler à un niveau sémantique pour résoudre les problèmes d'interopérabilité

Les principes de fonctionnement et l'architecture d'un moteur de monde virtuel qui implémente ces solutions sont présentés, ainsi que les algorithmes utilisés pour la classification et les différentes étapes de la migration d'une entitée virtuelle.

Mots-clés

Interoperabilité, Réutilisabilité, Ontologies, Mondes Virtuels, Réalité Virtuelle, Simulations, Apprentissage

– Abstract –

This paper describes several means to achieve interoperability between virtual worlds, and particularly for the problem of virtual entities migrating in foreign virtual worlds.

For that, two important methods are used :

– utilization of an original paradigm to manage actions/reactions, the influences/reactions model, which came from research on software agents
– utilization of conceptual classification algorithms, which permits to work on a semantic level in order to resolve the interoperabilities problems

The principes used and the design of a virtual world engine which implements those solutions are presented, as the algorithms used for automatic classification and differents states of virtual entity migration.

Keywords

Interoperability, Reusability, Ontologies, Virtual Worlds, Virtual Reality, Simulations, Knowledge acquisition

– Table des matières –

– Table des figures –

– Introduction –

De nos jours, il y a de plus en plus de demandes vers l'interopérabilité des logiciels et des matériels. Un minimum d'interopérabilité devient en effet indispensable, alors que la Société de l'Information prends ses marques, et que nombre de personnes veulent tirer partie des gigantesques possibilitées de communication et de connaissances apportées par les grands réseaux informatiques.

Ainsi, de nombreux protocoles ont été inventés, testés, utilisés, abandonnés ou plébiscités...Pour autant, des protocoles permettant une parfaite interopérabilité n'ont pas émergés dans tous les domaines ou ne sont pas parvenus au consensus. En particulier, en ce qui concerne le domaine de la Réalité Virtuelle (RV) et des mondes virtuels, aucun protocole n'a réellement émergé, bien que plusieurs tentatives furent intéressantes (VRML, SCOL, ...).

L'interopérabilité, du point de vue des mondes virtuels, permettrait pourtant une réutilisation de l'existant, donc une création plus rapide, plus efficace, de nouveaux mondes ou simulations.

Mais le problème s'est en fait révélé plus complexe que mettre au point un énième protocole de communication. En effet, les mondes virtuels sont des simulations, et ce faisant, ont une sémantique attachée à eux – ce n'est pas la réalisation de telle ou telle action dans un monde virtuel qui est fondamentalement importante, mais le sens qu'on y attache. L'action "marcher" ou l'action "chanter" pourraient ainsi être strictement identique d'un monde à l'autre, mais n'avoir absolument pas le même *sens* suivant le monde virtuel – un peu comme dans nos sociétés humaines. Le problème se place donc à un autre niveau, plus sémantique.

De nombreux travaux récents explorent l'utilisation de la sémantique (par exemple au travers des ontologies) dans l'informatique [1], et des avancées par exemple dans les algo-

[1] voir ainsi les travaux liés aux Web Sémantique ou aux agents logiciels

rithmes de classification conceptuelle permettent maintenant d'envisager une approche plus orienté sémantique au problème de l'interopérabilité dans les mondes virtuels.

Ce mémoire présente donc les travaux réalisés dans le cadre d'un stage de D.E.A au Laboratoire d'Informatique de Paris VI, dans l'équipe Réalité Virtuelle du thème Réseaux et Protocoles, sous la direction de M.Michel Soto.

Le mémoire se découpe en trois parties, une première partie décrivant le contexte du stage et les principes utilisés, une deuxième partie expliquant le fonctionnement d'un moteur de monde basé sur le concept d'*influence/réaction*, et une troisième partie décrivant les algorithmes utilisés pour améliorer le degré d'interopérabilité d'entitées virtuelles dans des mondes étrangers.

Contexte du Stage *1*

1. Le Laboratoire d'Informatique de Paris VI

Ce stage a été effectué au Laboratoire d'Informatique de Paris VI (lip6), au sein de l'équipe Réalité Virtuelle du thème Réseaux et Performances (RP).

Le Laboratoire d'Informatique de Paris VI a été créé le 1^{er} janvier 1997, lors de la restructuration des activités de recherche de l'université ; Il s'agit d'une Unité Mixte de Recherche [1] qui englobe l'ensemble des activités de recherche de l'UFR d'Informatique.

L'effectif est d'environ 320 permanents, dont 150 doctorants. Les permanents poursuivent des activités de recherche et d'enseignement. Le laboratoire est notamment responsable de la gestion de plusieurs D.E.A et D.E.S.S, dont le D.E.A Systèmes Informatiques Répartis (SIR), organisé par le thème Systèmes Répartis et Coopératifs (SRC).

Les «Thèmes» fédèrent des spécialistes autour de différentes directions de recherches liées à l'informatique. La liste des thèmes est disponible sur le site du laboratoire [2]. Les activités qui se déroulent dans un thème en dépassent parfois le cadre ; de tels projets «inter-thèmes» sont alors l'occasion de construire des liens entre les différentes spécialités. Les thèmes ne constituent donc pas un cloisonnement des recherches.

Le thème «Systèmes Répartis et Coopératifs» traite des applications réparties et coopératives, à un niveau théorique (permettant d'aborder les aspects d'algorithmique distribuée et de vérification formelle) et à un niveau méthodologique (pour ce qui concerne les aspects de spécification et de prototypage). Enfin il relève également de la programmation système de façon à mettre en œuvre des ateliers de génie logiciel et réaliser des mécanismes de placement dynamique et de tolérance aux fautes.

[1] UMR7606 du département Science Pour l'Ingénieur (SPI) du Centre National de la Recherche Scientifique (CNRS)

[2] http://www.lip6.fr

À l'instar des autres équipes du thème Réseaux et Performances, l'équipe Réalité Virtuelle traite les maux qui sont inhérents à l'utilisation des réseaux, mais agit au niveau des couches supérieures. Il ne s'agit pas ici d'assurer le cheminement d'un paquet ou de partager des ressources au mieux selon certains critères ; le travail porte plutôt sur la sémantique attachée à l'information et les problèmes qui apparaissent lorsque cette information est massive et distribuée. L'organisation, la distribution, la recherche, la présentation, la préservation de l'information et sa valeur sémantique sont ainsi dans la ligne de mire de cette équipe. Il y est donc fait un usage intensif des techniques issues de l'intelligence artificielle (ontologies, algorithmes d'acquisition de la connaissance, agents logiciels) et des systèmes de stockages de type bases de données.

2. Sujet du Stage

2.1. Contexte

L'interopérabilité est une propriété qui permet, d'une part, à des utilisateurs de mondes virtuels distincts d'accomplir des tâches communes (collaboration, coopération, jeux) et, d'autre part, de migrer des entités virtuelles dun monde virtuel à lautre. Cette propriété est importante car elle permet d'ouvrir les mondes virtuels qui sont, pour l'heure, des mondes clos. Les enjeux de l'interopérabilité sont la «généralisation» de la Réalité Virtuelle (RV) comme outil collaboration dans tous les domaines applicables et la réduction des coûts de développement des mondes virtuels.

L'interopérabilité doit faire face aux problèmes engendrés par l'hétérogénéité des mondes virtuels et en particulier l'hétérogénéité sémantique à laquelle les utilisateurs sont les plus sensibles. Il n'existe aujourd'hui aucune méthodologie de conception des mondes RV même si on laisse de coté la problématique de l'interopérabilité. Une telle méthodologie est pourtant nécessaire pour diminuer les coûts actuels de développement des mondes RV et les doter de capacités d'évolution afin d'amortir ces mêmes coûts. Dès que la problématique de l'interopérabilité doit être prise en compte dans la conception d'un monde RV, une méthodologie s'avère alors indispensable.

L'approche choisie dans le projet NOVÆ [3] est une approche architecturale à plat, par opposition à l'approche top-down de type HLA [4]. Cela a pour conséquence qu'il n'est plus nécessaire de définir au préalable un ensemble de concepts fédérateurs : les mondes RV peuvent ainsi être conçus de façon totalement indépendante, ce qui correspond d'avantage

[3] Networked Open VirtuAl Environment

[4] High Level Architecture, utilisé dans les simulations informatiques du DOD

à la réalité pratique. Autre conséquence, l'interopérabilité n'est plus envisagée de manière statique mais au besoin et de façon totalement dynamique.

Dans ce contexte la méthodologie de conception des mondes RV et les paradigmes sur lesquels elle s'appuie deviennent des points cruciaux. Le point de départ de la démarche du projet NOVÆ repose sur la constatation que l'action est le concept central des mondes RV. En effet, les mondes RV sont peuplés d'utilisateurs et d'objets appelés entités virtuelles dans le vocabulaire NOVÆ. La nature des entités est fonction du domaine d'application du monde. Certaines de ces entités ne possèdent pas de comportement propre comme par exemple, une table, un bureau ou un fichier. D'autres, pour la plupart, possèdent un comportement qui peut être purement réactif (sans pour autant être simple) comme par exemple une voiture, un avion, un train, une molécule. D'autres, enfin, possèdent un comportement autonome, comme par exemple, un robot, un avatar humain. Ces comportements se déroulent dans le parallélisme le plus total. Les comportements ne sont rien d'autre que des ensembles d'actions. Dans un objectif d'interopérabilité et plus encore dans l'approche NOVÆ, il importe que les actions puissent être décrites de façon la plus indépendante possible du monde RV où elles se réalisent. En effet, interopérer signifie, d'une part, que les entités virtuelles auront à se comporter dans ou avec des mondes RV étrangers autres que celui pour lequel elles ont été initialement conçues et que, d'autre part, il n'est pas possible de faire d'hypothèse sur la nature de ces mondes. Cela a pour conséquence que :

- Certaines actions du comportement d'une entité ne seront plus réalisables
- Les résultats des actions du comportement d'une entité encore réalisables dans le monde RV étranger pourront être différents des résultats de ces mêmes actions dans le monde natif de l'entité.

Il faut donc se doter d'un modèle exécutable de l'action permettant d'identifier les actions non réalisables et de calculer les résultats des actions encore réalisables dans le monde RV étranger. Le modèle de l'action choisi dans NOVÆ est le modèle influence/réaction issu des systèmes multi-agents où la problématique de l'action est aussi une problématique forte. Ce modèle propose une séparation nette entre la description de l'action et la description des conséquences de l'action qui sont en fait calculées par le monde où elles sont réalisées. Ceci est tout à fait satisfaisant du point de vue de l'interopérabilité.

2.2. Le Modèle Influence/Réactions

En effet, le modèle classique utilisé dans les simulations corresponds à un modèle *action/réaction* tel que montré sur le schéma 1.1 page suivante. Cependant, il souffre de plusieurs problèmes :

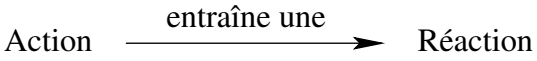


FIG. 1.1 – Action-Réaction

– Problème d'Universalité : en associant directement la réaction avec l'action, chaque objet est dépendant de son monde de départ. Prenons comme exemple un objet «guitare» ; si on associe à l'action «touche les cordes» la réaction «produit du son», la guitare fonctionnera bien dans son monde d'origine disposant d'une atmosphère. Mais si on veut utiliser la guitare sur la lune, sans atmosphère (et donc sans son), la guitare continuera à avoir la réaction «produit du son»! l'entité «guitare» n'a pas un comportement lui permettant d'être adapté à n'importe quelle situation et est donc dépendante de son monde de d'origine.
– Problème de Séquentialité : on ne peut représenter le résultat d'actions concurrentes avec le modèle *action/réaction*. En effet, chaque séquence *action/réaction* est exécuté séquentiellement. Imaginons deux actions provoquant des réactions également contraires sur un objet : l'utilisateur verra l'objet subir successivement les deux actions au lieu de ne rien constater (puisque les actions s'annulent en définitive)

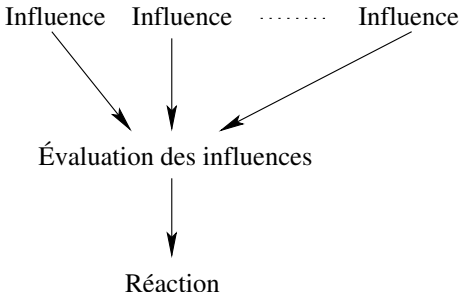


FIG. 1.2 – Influence-Réaction

Le modèle utilisé ici d'*influence/réaction* induit une étape supplémentaire (voir le schéma 1.2) de sommation des influences. Au lieu de générer directement des actions, les objets ou le monde émettent d'abord des *influences*, c'est à dire, des «souhaits» d'actions. Le moteur de monde va ensuite passer par une phase d'analyse des différentes influences, et générer la réaction correspondante.

Le modèle *influence/réaction* résout ainsi à la foi le problème d'Universalité et le problème de Séquentialité. En effet, les réactions sont ici nettement découplées des influences, et chaque monde peut alors définir des lois de réactions adéquates répondants à un type d'influence. Si l'on reprends notre exemple de guitare, l'influence «touche les cordes» déclenchera une loi de réaction correspondante. Cette loi générera la réaction «produit du son» dans un monde disposant d'une atmosphère, et ne générera rien dans un monde

qui en est dépourvu. Ainsi, notre entité «guitare» est portable d'un monde à l'autre : la réaction étant dépendante du monde et non encodé directement dans l'objet. Le problème de Séquentialité est lui résolu par l'étape de sommation des influences.

En pratique les actions des entités virtuelles sont décrites sous forme de lois dites de comportement, qui génèrent des influences. La réaction du monde RV permettant le calcul du résultat effectif des actions des entités virtuelles est, elle aussi, décrite sous forme de lois dites de réaction (déclenchées par la réception des influences).

2.3. Interopérabilité à travers le modèle influence/réaction

Le choix du modèle influence/réaction permet donc d'obtenir une première interopérabilité : notre guitare pourra effectivement passer sans difficultés et en gardant un comportement cohérent d'un monde terrien à un monde lunaire. Toutefois, le modèle influence/-réaction ne résout pas tous les problèmes :

- les symboles utilisés peuvent ne pas être identiques (par exemple, un symbole *vent* et un symbole *wind*, sont différents et pourtant désignent la même chose ; à l'inverse on peut avoir deux symboles identiques mais désignants deux choses différentes)
- il est également possible qu'une propriétée du monde d'origine d'une entité virtuelle, vitale pour son fonctionnement, ne soit plus disponible. Et à l'inverse, une entitée migrée dans un monde étranger peut manquer de certaines caractéristiques, l'empêchant alors de fonctionner correctement dans son nouveau monde

2.4. Le gérant d'interopérabilité et les mécanismes d'apprentissage

Afin de prendre en compte ces types de problèmes, NOVÆ complète le gérant d'interopérabilité par des mécanismes d'apprentissage. Ces mécanismes sont, dans l'ordre d'application, le déguisement, l'imitation et l'apprentissage.

Le *déguisement* est un mécanisme qui construit la hiérarchie des entités virtuelles du monde RV étranger et de leurs propriétés afin de rendre l'entité virtuelle du monde RV natif «semblable» à une entité virtuelle du monde RV étranger. Ceci permet à une entité virtuelle de compléter ses attributs. Il s'agit de la phase qualitative du déguisement.

Exemple : si toutes les entités virtuelles du monde possèdent un attribut *masse*, on le rajoutera aux entitées migrées qui ne l'ont pas.

Il faut aussi se préoccuper de l'aspect quantitatif afin de fournir une valeur aux attributs issus de la phase qualitative. On pourra aussi envisager de se doter d'une valeur par défaut

pour ces attributs ou bien prendre, par exemple, la valeur moyenne de ces valeurs pour les entités virtuelles existant dans le monde RV étranger.

Le gérant d'interopérabilité utilise un algorithme de regroupement conceptuel de type COING pour organiser les connaissances disponibles sur les entités virtuelles du monde RV étranger et ensuite un ensemble de règles heuristiques de déguisement.

L'*imitation* est un mécanisme d'analyse des comportements du point de vue statique (les contextes, les entités virtuelles en jeu, etc.) pour identifier à quels états du monde ils s'intéressent et quelles influences ils produisent pour agir. Le but de cette analyse est de se comporter suivant les mêmes règles que celles du monde RV étranger.

Exemple : si toutes les entités virtuelles du monde RV étranger sont sensibles à la gravité il faut que les entités virtuelles pénétrant dans le monde RV étranger y soient sensibles aussi.

Enfin, l'*apprentissage* est un mécanisme d'analyse et d'amélioration de l'interopérabilité. Concrètement, c'est ici le comportement même de l'entité virtuelle que le gérant d'interopérabilité essaie d'adapter au monde RV étranger.

Exemple : si l'entité virtuelle ne bouge pas dans le monde étranger alors qu'elle bougeait dans le monde RV natif, le gérant d'interopérabilité analyse les comportements qui font que les autres entités virtuelles bougent et modifie la correspondance comportementale effectuée par le mécanisme d'imitation. Par exemple, mettre en correspondance l'attribut power de l'entité virtuelle arrivant avec l'attribut puissance du monde étranger afin que le comportement mouvement utilise cette puissance.

2.5. Travail déja effectué

Il existe un prototype de deux mondes RV exemples. Des différences sémantiques ont été introduites dans ces mondes afin d'illustrer comment des techniques d'apprentissage peuvent accroître leur possibilité d'interopérer. Les domaines d'application choisis sont, pour le monde étranger, la bataille navale et, pour le monde natif, l'initiation à la voile.

Ce prototype est aujourd'hui capable d'adapter la description d'une entité virtuelle du monde natif (ici, bateaux à voile) pour accroître son interopérabilité avec un monde étranger (ici, bateaux de guerre). Le bilan de ce travail peut être décliné selon les 3 mécanismes d'apprentissage implémentés.

Déguisement : le prototype donne de bons résultats aussi bien sur le choix des attributs avec lesquels une entité est déguisée que sur les valeurs calculées pour ces attributs.

Imitation : les résultats sont assez bons mais perfectibles. Il faudrait pour cela imaginer et implanter des algorithmes prenant mieux en compte les relations existant entre les actions du comportement d'une entité.

Apprentissage : ce mécanisme est implanté mais doit encore faire l'objet d'un travail important de mise au point à la fois sur le plan scientifique et sur le plan pratique. Sur ce dernier plan, ce mécanisme doit être intégré au moteur de loi pour être testé en situation.

2.6. Objectifs du Stage

Plusieurs objectifs étaient donnés :

① Reprendre les précédents travaux ([4, 2, 7]) pour obtenir un moteur complet

② Améliorer les mécanismes d'apprentissage utilisés dans les services d'interopérabilité

③ Créer un moteur d'explication des mécanismes d'apprentissage

④ Fournir des jeux de tests pour les mécanismes d'apprentissage

Architecture du projet 2

1. Définitions et Architecture Générale

Un monde virtuel (une réalité virtuelle) est une simulation informatique, permettant de représenter des espaces ou des situations données. Dans ce projet, un monde est constitué de différents états ou attributs, de lois de réactions, de lois de comportement (lois du monde), et d'entités virtuelles.

Une entité virtuelle représente un objet du monde, actif ou passif, voire un avatar virtuel d'un utilisateur humain. Chaque entité est une instance d'une classe d'entités virtuelles.

Les classes d'entités définissent les différents attributs et comportements associés aux entités et disposent de l'héritage (une classe peut hériter des attributs et comportements d'une autre classe).

Les attributs ont un nom, une valeur et un type. Ils caractérisent des informations sémantiques de l'entité, tel que sa masse ou sa couleur.

Le projet NOVÆ est composé principalement d'un moteur de monde, chargé de l'exécution des mondes virtuels en utilisant et appliquant les différentes lois de comportements et de réaction. En supplément du moteur de monde, un gérant d'interopérabilité se charge d'insérer de nouvelles entités virtuelles étrangères (figure 2.1 page suivante).

2. Reprise de l'existant

Un moteur de monde existait ([7]) mais celui-ci n'était pas entièrement satisfaisant :
– mise en œuvre assez complexe des mondes virtuels
– relativement lent à l'exécution

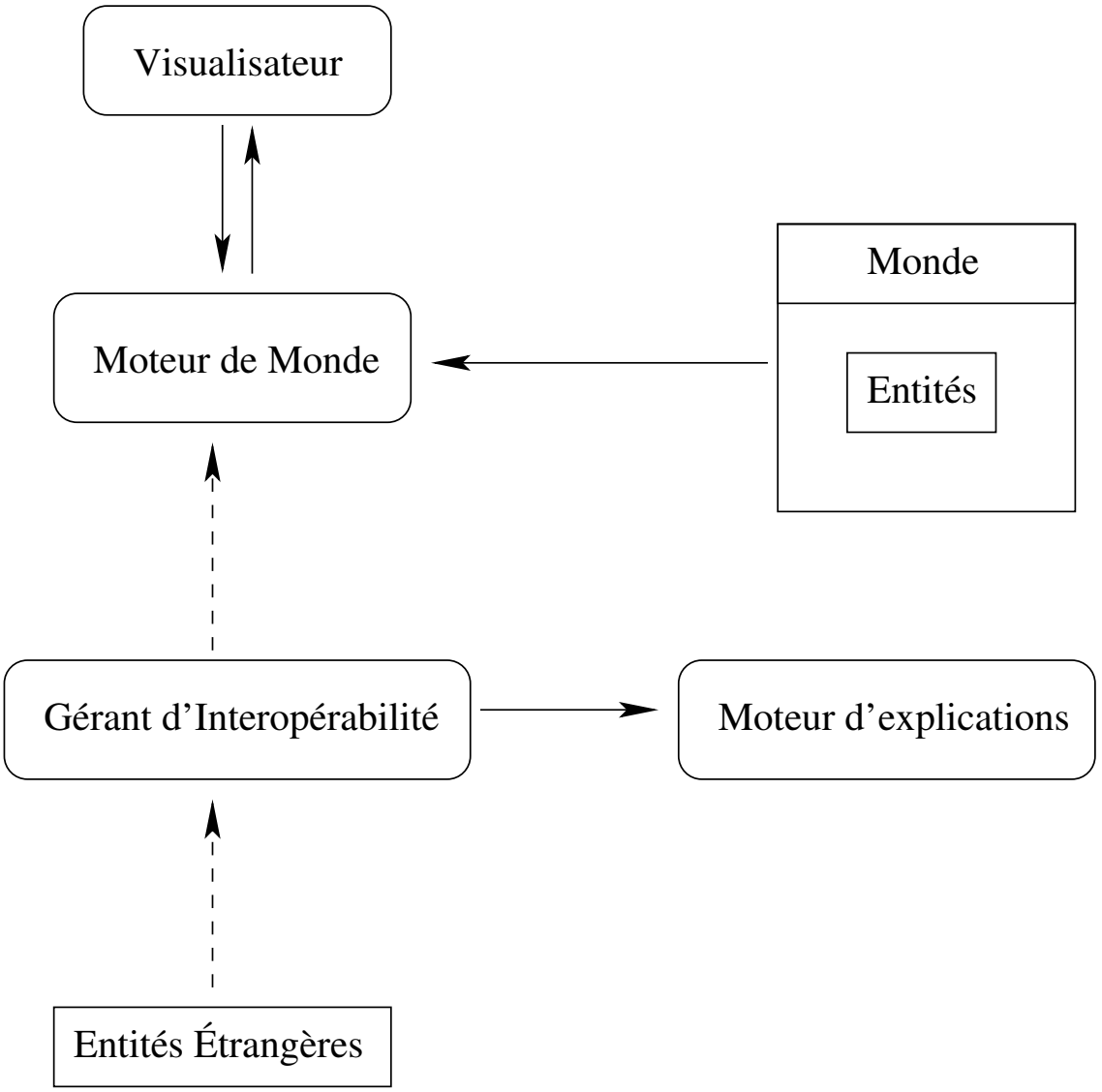


FIG. 2.1 – Architecture du projet NOVÆ

– les attributs ne disposaient que de types de données de base (entiers, flottants, chaînes de caractères)
– lors de la migration d'entitées étrangères, on modifiait directement les entitées migrées plutôt que leurs classes

L'intégration avec le gérant d'interopérabilité n'était pas idéal non plus, chacun utilisant sa propre base de donnée.

La technologie utilisée pour implémenter le moteur est Java (version 1.3.1), ce qui permet une utilisation et un portage immédiat sur le plus grand nombre de plateformes. De plus, Java est un langage orienté objet intéressant et facilitant le développement (librairies standards disponibles, utilisation d'un ramasse-miette automatique, etc.)

3. Modifications apportées

3.1. Attributs

Une des premières modifications a été de prendre en compte des types complexes en plus des types de données simples (entiers, flottants, chaînes de caractères...), permettant de gérer des structures.

En effet, bien que cela ne change que peu de choses en termes d'exécution du monde, au niveau sémantique cela est plus intéressant.

Prenons un exemple ; si on veut stocker la position d'une entité, elle disposera alors de trois attributs :

① PositionX (float)

② PositionY (float)

③ PositionZ (float)

Il est plus intéressant de pouvoir définir un type «position» comprenant trois attributs X,Y et Z. D'une part, cela permet une écriture plus simple et plus lisible des classes d'entitées, d'autre part, on améliore la sémantique en liant effectivement ces trois attributs X,Y et Z en un seul attribut de type «position».

3.2. Modèle Objet et Héritage

Le moteur est strictement conçu sur un modèle objet, chaque entité du monde étant une instance d'une classe donnée. Le précédent moteur utilisait bien des classes d'entitées,

mais rajoutait directement les attributs *dans* les entitées migrées et non dans leur classe, et ne gérait pas l'héritage.

Le gestionnaire d'interopérabilité crée donc maintenant une nouvelle classe à partir de la classe de l'entitée étrangère, et insére cette classe dans le moteur. L'entitée étrangère migrée utilisera alors cette classe. On garde ainsi un modèle orienté objet cohérent, et qui permet de plus de réutiliser très simplement le travail de migration effectué si on veut insérer une deuxième entitée de même classe.

Enfin, chaque classe peut hériter d'une autre classe de façon (ce qui n'était pas le cas auparavant), de façon à maximiser la réutilisation de classes existantes. Le modèle d'héritage choisi est un héritage simple (une classe ne peut hériter que d'une autre classe). Lors de l'héritage, la classe fille héritera alors des différents attributs et lois de la classe mère.

3.3. Structuration des fichiers

Le moteur précédent utilisait un fichier XML stockant toutes les informations liées à un monde (les différentes lois du monde, les classes et instances utilisées). Le choix du format XML a été gardé, car il permet très facilement de structurer de l'information, tout en étant lisible en clair. De plus Java dispose de librairies permettant d'accéder directement au contenu de fichiers XML. Par contre, le format a été modifié, de façon à obtenir une meilleure lisibilité. De plus, un monde n'est plus stocké dans un seul fichier, mais éclaté en différents fichiers, contenus dans un dossier suivant une hiérarchie précise (figure 2.2 page suivante).

3.4. Utilisation d'un parser pour les expressions

Les lois de réactions, prédicats, et influences, sont exprimés directement (voir également l'exemple d'une classe de monde *2.1.* page 45) :

```
<poststates>
 <poststate value="e1.position = e1.position + result"/>
</poststates>
```

Le parser permet de référencer des attributs sous la forme nomObjet.nomAttribut, de faire des opérations de base et d'utiliser les fonctions mathématiques classiques (voir le fichier cup générant le parser des prédicats page 50). De plus, les mots-clés *self* et *world*

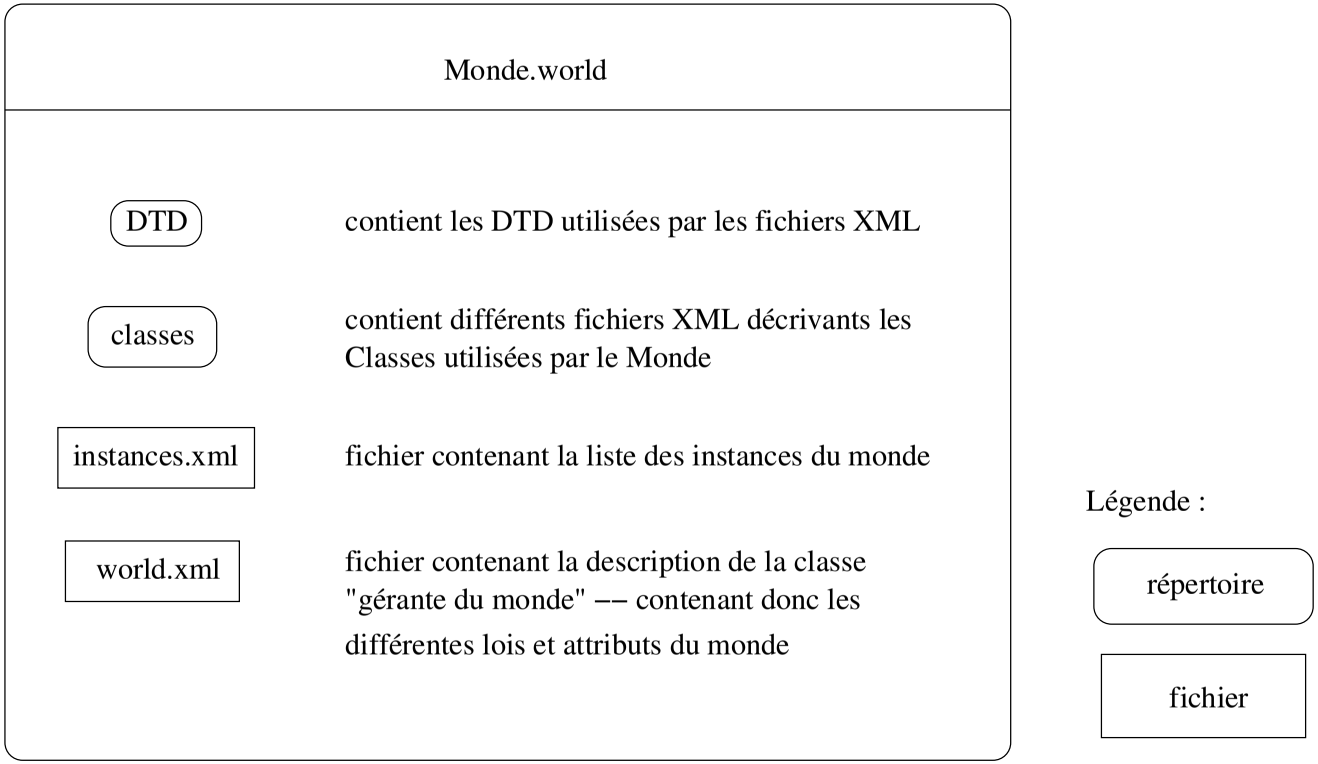


FIG. 2.2 – Structuration d'un dossier de monde

utilisables à la place de «nomObjet» permettent de référencer respectivement l'objet courant dont on évalue la loi et la classe du monde.

4. Fonctionnement du Moteur : le cycle influence-réaction

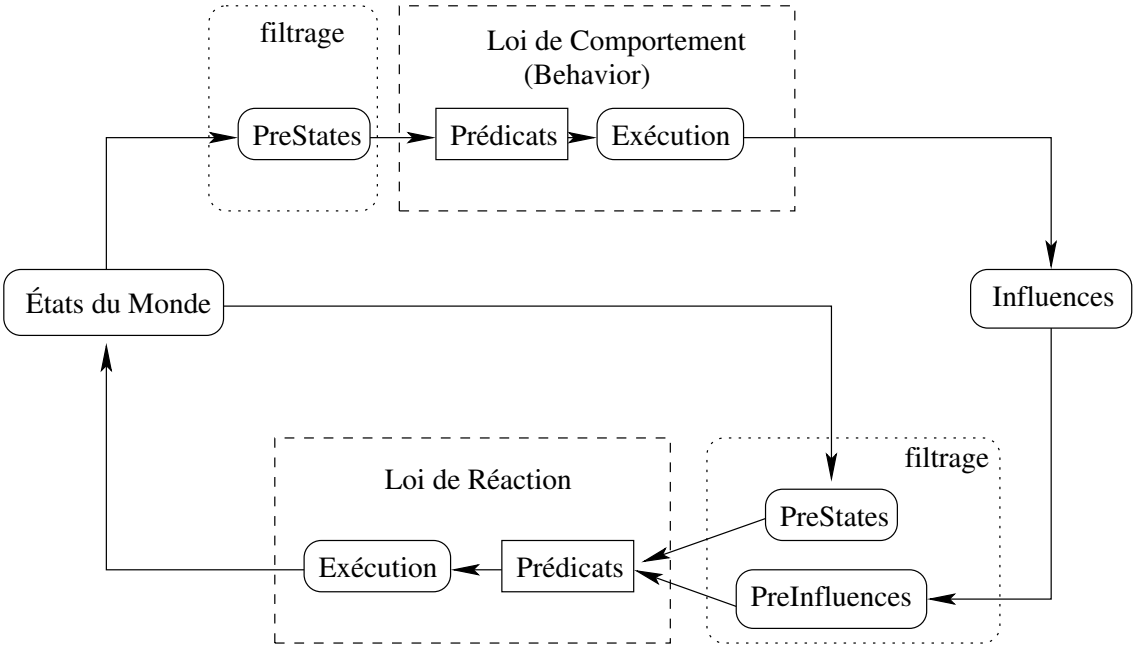


FIG. 2.3 – Cycle Influence-Réaction dans le moteur de monde

La figure 2.3 nous montre le cyle influence-réaction utilisé par le moteur.

Deux types de lois sont utilisés :
– Les lois de comportement, générant des influences
– Les lois de réactions, modifiant les états du monde
À partir des différents états du monde – c'est à dire des différents attributs des entitées du monde – le cycle consiste à appliquer respectivement les lois de comportement puis les lois de réactions. Précédent chaque loi, il y a d'abord une étape de filtrage (sur les états et/ou les influences reçues), puis on applique ou non la loi.

4.1. Lois de Comportement

Les lois de comportements vont générer des influences ; l'évaluation d'une loi se décompose de cette façon :

① on filtre d'abord les entitées du monde pour obtenir celles qui nous intéressent

② si le ou les prédicats sont validés, alors...

③ ... on exécute la loi, qui va générer une influence

D'après la DTD, une loi de comportement contient les trois éléments suivants, correspondants aux différentes parties de la loi :

① prestates

② predicates

③ postinfluences

L'élément prestates permet d'indiquer une liste d'état préliminaires au déclenchement de la règle (filtrage). L'élément predicates contient une liste de prédicat à valider ; si cet élément est absent, le prédicat est considéré comme toujours valide. Enfin l'élément postinfluences contient les différentes influences pouvant être générées si la loi est validée.

Voici un exemple de loi de comportement :

```
<behavior name="vent_ouest">
 <prestates>
  <state name="position" entity="e1"/>
 </prestates>
 <postinfluences>
  <influence name="inf_vent" entity="e1" value="self.vent_ouest"/>
 </postinfluences>
</behavior>
```

Cette loi récupère donc toutes les entitées contenant un attribut (état) «position» et leur associe le nom «e1» (utilisable dans le reste de la loi). Elle génère ensuite une influence «inf_vent» sur l'entité «e1» en cours, qui a pour valeur l'attribut «self.vent_ouest».

4.2. Lois de Réactions

Les lois de réactions fonctionnent de manière similaire, mais effectuent en plus d'un filtrage sur les états du monde un filtrage sur les influences pouvant être reçues. Ainsi une loi de réaction se décompose en cinq parties :

① prestates
② preinfluences
③ predicates
④ operations
⑤ poststates

Les éléments prestates et preinfluences se chargent d'effectuer un filtrage ; l'élément predicates permet d'indiquer des prédicats à valider. L'élément operations indique le type d'opération ensembliste (SUM,COUNT,AVG,MAX,MIN) à appliquer à l'ensemble des influences reçues. Enfin l'élément poststates permet de définir le nouvel état de l'entité.

Exemple d'une loi de réaction :

```
<reaction name="reaction_vent">
 <prestates>
  <state name="position" entity="e1"/>
 </prestates>
 <preinfluences>
  <preinfluence name="inf_vent" entity="e1"/>
 </preinfluences>
 <operations>
  <operation type="sum" using="inf_vent" name="result"/>
 </operations>
 <poststates>
  <poststate value="e1.position = e1.position + result"/>
 </poststates>
</reaction>
```

Cette loi récupère d'abord l'ensemble des entitées «e1» ayant un attribut «position» ainsi que l'ensemble des influences «inf_vent» reçues par les entitées «e1». Sur l'ensemble des influences «inf_vent» reçues, on applique une somme, et le résultat de cette opération est rangé dans une variable «result». Enfin, on modifie la valeur de l'attribut «position» de l'entité «e1» courrante, en lui rajoutant la valeur de «result».

4.3. Lois Internes et Lois du Monde

Le comportement du monde est gêré par une classe, comme n'importe quel autre entitée ; sa seule particularité étant d'être définie dans le fichier world.xml (voir la figure 2.1 page 18). Le moteur de monde se charge de créer automatiquement une instance de cette classe.

Chaque classe peut contenir des lois de comportement et des lois de réaction ; mais pour les classes hors classe du monde, les lois de réaction ne pourront concerner que les états internes de la classe.

Apprentissage 3

Nous avons vu dans la section *2.4.* page 13 que l'interopérabilité ne peut se satisfaire uniquement du modèle *influence/réaction*, et que l'on utilise alors des algorithmes d'apprentissage. En fait, [1, pp.61-67] donne une méthode de calcul du degré d'interopérabilité d'une entité, en se basant sur le nombre de comportements valides (c'est à dire, qui fonctionneront à priori de manière identique) de l'entité dans un monde étranger. Une fois connu ce degré d'interopérabilité, on peut essayer de l'augmenter, par diverses méthodes (déguisement, imitation et exploration). Ces méthodes utilisent un algorithme de classification automatique, l'algorithme de Godin (voir [5, 6]), permettant la création incrémentale d'un treillis de Galois.

1. Le Treillis de Galois

Le treillis de Galois est une figure définissant une hiérarchie de concepts à partir de deux ensembles finis E et E'. Ces deux ensembles sont reliés par une relation binaire $R \subseteq E \times E'$. Les regroupements naturels de E et E' par rapport à la relation R peuvent être représentés par un treillis (voir figure 3.1 page 31). Chaque élément du treillis est un couple (appelé concept formel), noté (X, X') et composé d'un ensemble $X \in P(E)$ et d'un ensemble $X' \in P(E')$ satisfaisant aux deux propriétés suivantes :

① $X' = f(X)$ où $f(X) = \{x' \in E' | \forall x \in X, xRx'\}$

② $X = f'(X')$ où $f'(X') = \{x \in E' | \forall x' \in X', xRx'\}$

On associe habituellement les éléments de E aux objets que l'on veut classer, et les éléments de E' à leurs attributs.

Le treillis de Galois (noté G) d'une relation R est l'ensemble de tous les couples complets (c'est à dire, répondant aux propriétés 1 et 2) dérivés de R.

La relation d'ordre partiel définissant le treillis est la suivante :

Pour deux concepts $C_1 = (X_1, X_1')$ et $C_2 = (X_2, X_2')$, $C_1 < C_2 \Leftrightarrow X_1' \subset X_2'$

Étant donné que $X_1' \subset X_2' \Leftrightarrow X2 \subset X1$, on a également $C_1 < C_2 \Leftrightarrow X_2 \subset X_1$.

Le treillis de Galois est particulièrement adapté à notre problématique, puisque nous traitons de classes et de hiérarchies de classes. Les différents concepts C_i représentent alors les ensembles de classes dotés d'attributs communs. La relation d'ordre du treillis corresponds alors à une relation de généralisation/spécialisation : les éléments supérieurs du treillis sont plus généraux que les éléments inférieurs.

Plusieurs algorithmes de construction du treillis sont disponibles (par exemple ceux de *Bordat* et de *Godin*) ; l'algorithme utilisé ici est celui de Godin (voir [5]), car il nous permet de construire un treillis de façon incrémentale. Cela nous permet alors de ne pas regénérer un nouveau treillis en partant de zéro dans le gérant d'interopérabilité à chaque ajout d'un nouvel objet...

En terme de complexité, le treillis de Galois peut croître exponentiellement avec le nombre d'objets et de propriétés – mais cette complexité est bornée par le nombre de propriétés k . Avec n le nombre d'objets, on constate que la taille du treillis est inférieure à $2^k n$. Le nombre de propriétés k étant fixé, on peut considérer que la complexité reste raisonnable.

La représentation graphique du treillis de Galois est appellée *diagramme de Hasse* (exemple de treillis, voir figure 3.1 page 31).

2. Algorithme du Treillis de Galois

Voici l'algorithme utilisé pour la construction du treillis de Galois. On ajoute un nouveau couple objet + attributs, $(\{x^*\}, f(\{x^*\}))$, avec x^* le nouvel objet et $f(\{x^*\})$ les attributs reliés à x^* par la relation R.

$sup(G)$ correspondant au concept supérieur (i.e, contenant le plus d'attributs) du treillis.

ALGORITHME DE GODIN

---

if $sup(G) = (0, 0)$ **then**
  On remplace $sup(G)$ par le couple $(\{x^*\}, f(\{x^*\}))$
else
  if not $(f^*(x^*) \subseteq X'(sup(G)))$ **then**
    if $X(sup(G)) = 0$ **then**

$X'(sup(G)) \leftarrow X'(sup(G)) \cup f(\{x^*\})$
else {On ajoute un nouveau concept H qui deviendra $sup(G^*)$}
$H : (0, X'(sup(G)) \cup f(\{x^*\}))$
On ajoute un lien $sup(G) \rightarrow H$
end if
$C[i] \leftarrow \{H : ||X'(H)|| = i\}$ {On crée un tableau $C[i]$ qui contiendra, pour i correspondant à une cardinalité, la liste des X' correspondants}
$C'[i] \leftarrow 0$ {On crée un tableau $C'[i]$ et on l'initialise à zéro}
for $i : 0$ **to** cardinalité maximale **do** {On parcours C par ordre croissant de cardinalité}
for each paire H dans $C[i]$ **do**
if $X'(H) \subseteq f(\{X*\})$ **then** {On traite les paires modifiées}
On ajoute x^* à $X(H)$
On ajoute H à $C'[i]$
if $X'(H) = f(\{X^*\})$ **then**
On quitte l'algorithme
end if
else
$int \leftarrow X'(H) \cap f(\{X^*\})$
if $\neg\exists H_1 \in C'[||int||]$ tel que $X'(H1) = int$ **then**
On crée une nouvelle paire $H_n = (X(H) \cup \{x^*\}, int)$ et on l'ajoute à $C'[||int||]$
On ajoute un lien $H_n \rightarrow H$
for $j : 0$ **to** $||int|| - 1$ **do** {On modifie les liens}
for each $H_a \in C'[j]$ **do**
if $X'(H_a) \subset int$ **then** {H_A est un parent potentiel de H_n}
$parent \leftarrow VRAI$
for each H_d fils de H_a **do**
if $X'(H_d) \subset int$ **then**
$parent \leftarrow FAUX$
break
end if
end for
if *arent* **then**
if H_a est un parent de H **then**
on élimine le lien $H_a \rightarrow H$
end if
On ajoute un lien $H_a \rightarrow H_n$
end if
end if
end for

```
                end for
                if int = f*({x*}) then
                  On quitte l'algorithme
                end if
              end if
            end if
          end for
        end for
      end if
    end if
```

3. Le Déguisement

Rappelons notre définition du déguisement (voir page 13) :

Le déguisement est un mécanisme qui construit la hiérarchie des entités virtuelles du monde RV étranger et de leurs propriétés afin de rendre l'entité virtuelle du monde RV natif «semblable» à une entité virtuelle du monde RV étranger. Ceci permet à une entité virtuelle de compléter ses attributs.

En effet, rajouter des attributs à l'entité virtuelle peut améliorer de façon importante l'interopérabilité, en ajoutant des attributs nécessaires ; imaginons un monde ou tous les objets disposent d'une masse, et ou des lois du monde (la pesanteur...) agissent ou se servent de cet attribut, si on fait migrer dans ce monde une entité dépourvue de cet attribut son comportement sera à priori complètement indéfini.

3.1. Déroulement du déguisement

Sois deux mondes, M_A et M_B, et E une entité virtuelle de M_A et de classe C. Le monde M_B contient plusieurs classes K_i et instances associées.

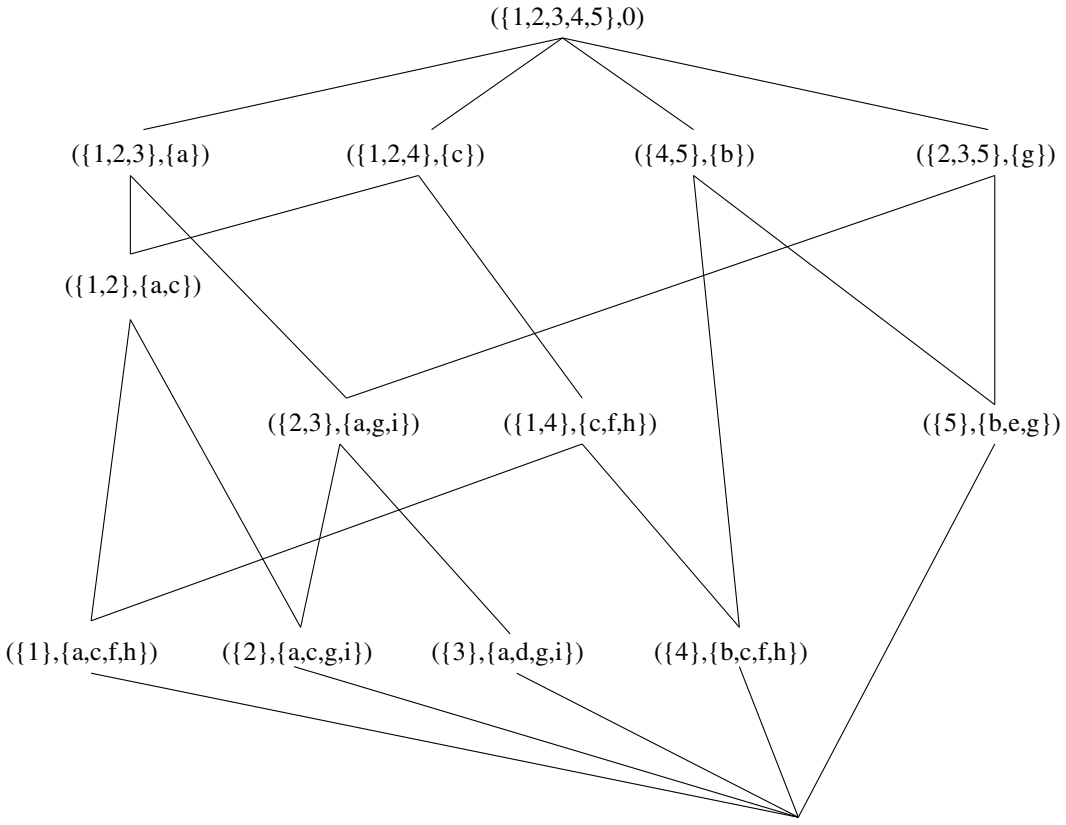
Le but est de faire migrer E de M_A vers M_B. Cette migration s'effectue en plusieurs étapes :

① migration de C, dans M_B

② création de la classe C_B, résultante de la migration de C

R	a	b	c	d	e	f	g	h	i
1	1	0	1	0	0	1	0	1	0
2	1	0	1	0	0	0	1	0	1
3	1	0	0	1	0	0	1	0	1
4	0	1	1	0	0	1	0	1	0
5	0	1	0	0	1	0	1	0	0

Représentation matricielle de la relation R



Treillis de Galois correspondant à la relation du tableau

FIG. 3.1 – Exemple de Treillis de Galois

③ envoi de E dans M_B

④ E_B est instancié dans M_B à partir de E et en utilisant C_B

⑤ les éventuels attributs supplémentaires de E_B sont valués

Le principe du déguisement est donc de migrer dans un premier temps la classe, puis ensuite de migrer l'instance. La figure 3.2 page ci-contre montre le déroulement de l'opération.

En effet, on travaille d'abord et avant tout sur les classes (ici C) – travailler directement sur les instances, ne conduirait qu'à une explosion combinatoire. De plus, on perdrait l'avantage de réutiliser le travail effectué sur une entité migrée si on veut en migrer une seconde qui soit de même classe...

3.2. Migration de C dans M_B

La migration de C va consister à bâtir un treillis de Galois à partir de ΣK_i, l'ensemble des classes de M_B. Une fois cela fait, on doit déterminer à quel sous-groupe du treillis notre classe C se rapproche le plus.

Pour cela, on détermine l'intersection I de l'ensemble des attributs des classes de M_B, ΣK_{i_a} avec l'ensemble des attributs de C, ΣC_a :

$$I = \sum K_{i_a} \cap \sum C_a$$

Les attributs sont typés.

Une fois que l'on a I, on essaie alors de trouver un concept H du treillis qui corresponde à notre intersection.

Cependant, trouver un concept adéquat n'est pas forcément simple, et plusieurs stratégies de recherche peuvent être envisagées. On cherche un concept qui comporte un maximum de propriétés communes avec I ; Il existe forcément un concept ou I apparaît [1], mais la difficulté consiste à choisir le bon parmis les nombreuses possibilitées. En effet, on trouve généralement plusieurs concepts pouvant répondre à nos critères, descendants d'un concept minimal H_{min}.

H_{min} correspondant au concept qui possède le plus de propriétées se retrouvant dans I tout en ayant le moins de classes associées possibles (soit $card(E)$ le plus petit possible).

[1] dans le pire des cas, il s'agit du concept inférieur si I est vide ou du concept supérieur si I est trop complexe

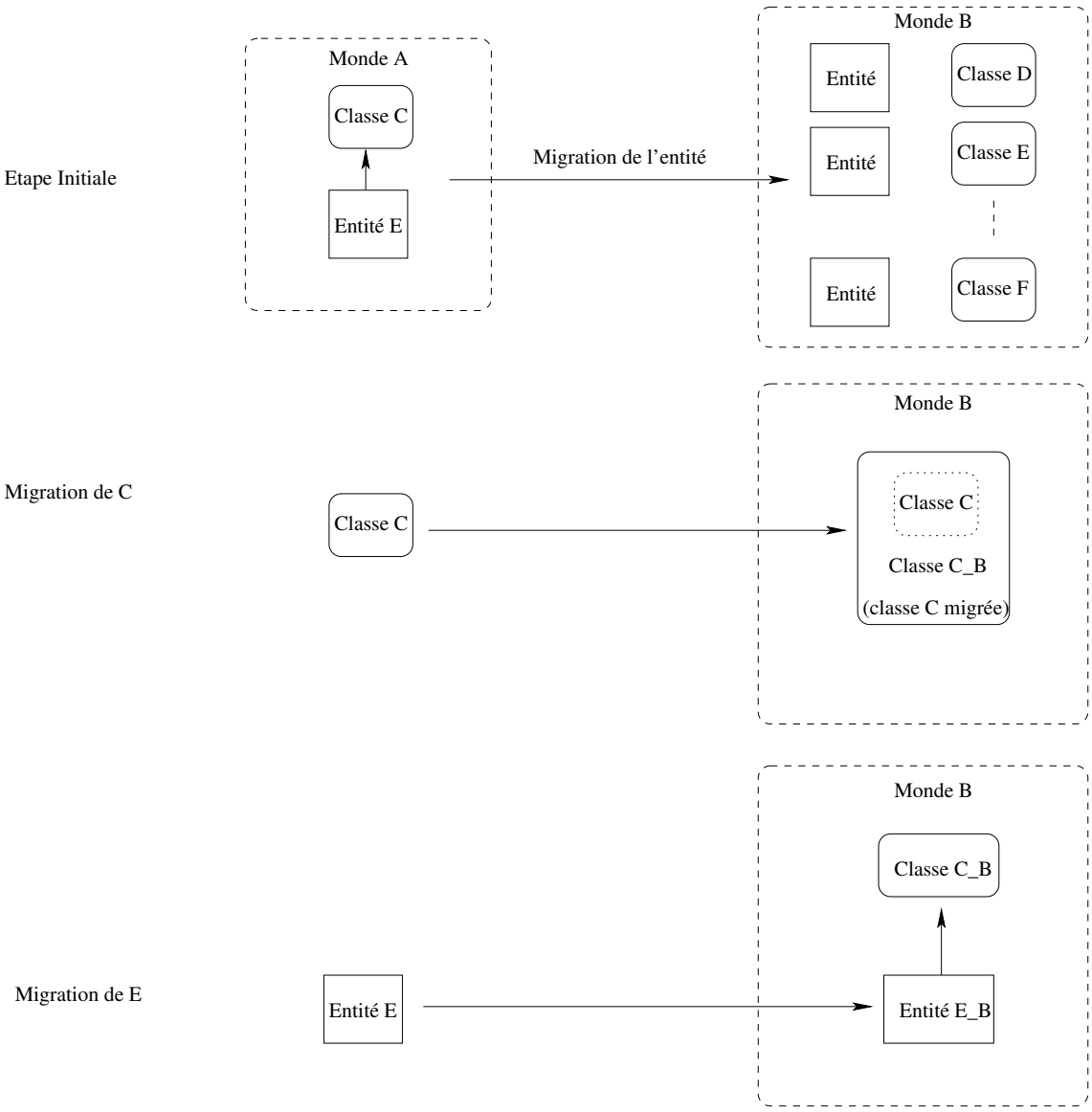


FIG. 3.2 – Déguisement

Plusieurs solutions sont possibles à partir de là, par exemple on peut utiliser ([8, p.27]) :

- Classic : utilise simplement H_{min}
- Fashion : on choisit l'entité appartenant à l'extension du nœud H_{min} ayant le moins d'attributs, et on complète C avec ces attributs
- Very Fashion : même principe que pour Fashion, mais au lieu de choisir l'entité la plus simple, on choisit l'entité la plus complète
- Excentric : cette fois-ci, on ajoute l'ensemble des attributs des entités du nœud H_{min} et on les ajoutes à C

Pour le moment, le choix est laissé à l'utilisateur de façon interactive.

3.3. Assimilation de notre classe C dans la hiérarchie des classes de M_B

Sois H_δ, le concept considéré comme le plus proche après la recherche dans le treillis (en ayant utilisé une des méthodes, Classic, Fashion, Very Fashion ou Excentric). Il peut être intéressant d'homogénéiser ce que l'on a reçu avec la hiérarchie des classes de M_B ; en effet, pour le moment on obtient juste un concept, H_δ, contenant l'ensemble des attributs à ajouter à C, mais pas de relations d'héritage.

Hors, il peut être dommage de simplement rajouter les attributs de H_δ (moins les doublons) à C. On essaie donc de déterminer, parmis les classes de H_δ, celle qui serait la plus intéressante pour un héritage – c'est à dire, celle qui apporterait le plus grand nombre d'attributs à C par elle même, de façon à minimiser le nombre d'attributs à ajouter à C. En fait, cela corresponds à «ranger» C_B le plus bas possible dans la hiérarchie des classes de M_B.

Pour chaque classe contenu dans ce concept, on demande donc au moteur de monde la liste de ses attributs. On détermine ensuite l'intersection la plus grande par rapport à notre liste totale d'attributs de C. Cela nous permet de déterminer C_h, la classe la plus intéressante à hériter.

On crée alors C_B en indiquant qu'elle dérive de C_h ; il ne reste plus alors qu'à ajouter les attributs de C qui ne seraient pas déja présent, ainsi que ceux éventuellement supplémentaires à C_h que H_δ contenait. Bien évidemment, les doublons ne sont pas pris en compte.

Note

Cette méthodologie n'est pas forcément idéale ; en effet, on peut très bien avoir des entités totalement similaires au niveau des attributs, et qui pourtant ont des comportements diamétralement opposés ; cependant, l'utilisation d'attributs typés et de l'héritage dans les classes, permet néanmoins d'obtenir des résultats intéressants. En particulier car une seule hiérarchie de classe existe dans l'exemple – une racine commune. Dans le cas d'un monde ayant plusieurs racines, on peut imaginer des problèmes plus régulièrement. L'assimilation de C_B dans la hiérarchie des classes doit donc être considéré de façon optionnelle, mais dans certains cas les résultats sont effectivement intéressants.

3.4. Exemple de migration d'une classe

On dispose d'un monde M_B contenant les classes suivantes :

classe entites :
– pos (position)
– taille (integer)
– masse (integer)
– vitesse (integer)
classe entites_guerre dérive (entites) :
– blindage (integer)
classe navires_guerre dérive (entites_guerre) :
– canons (integer)
classe sousmarins_guerre dérive (entites_guerre) :
– missiles (integer)
– profondeur (float)

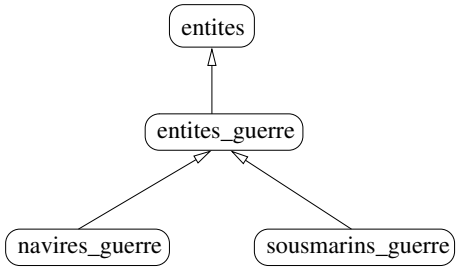


FIG. 3.3 – Hiérarchie des classes

Le schéma 3.3 montre la hiérarchie obtenue.

On veut insérer la classe C suivante :

classe avion :
– pos (position)
– passagers (integer)
– autonomie (float)
– taille (integer)
Voici les différentes classes avions_migrated que le gérant d'interopérabilité a créé :

Déguisement Fashion

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE classes SYSTEM "class.dtd">
<class name="avion_migrated" extends="entites">
 <attributes>

      <!-- Les attributs suivants sont partagés par avion et entites -->
      <!-- [<taille:integer>, <pos:position>] -->

      <!-- Les attributs suivants de la classe entites n'existaient pas dans la
          classe avion originale -->
        <!- [<vitesse:integer>, <masse:integer>] -->

        <!- Les attributs suivants n'existaient pas dans la classe entites -->
      <attribute name="passagers" type="integer"/>
      <attribute name="autonomie" type="float"/>
 </attributes>
</class>
```

Déguisement Very Fashion

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE classes SYSTEM "class.dtd">
<class name="avion_migrated" extends="sousmarins">
 <attributes>

      <!-- Les attributs suivants sont partagés par avion et sousmarins -->
      <!-- [<taille:integer>, <pos:position>] -->

      <!-- Les attributs suivants de la classe sousmarins n'existaient pas dans la
          classe avion originale -->
```

```
␣␣␣␣␣␣␣␣<!−−␣[<blindage:integer>,␣<vitesse:integer>,␣<masse:integer>,␣<
    profondeur:float>,␣<missiles:integer>]␣−−>

      <!−− Les attributs suivants n' existaient pas dans la classe sousmarins −−>
      <attribute name="passagers" type="integer"/>
      <attribute name="autonomie" type="float"/>
 </attributes>
</class>
```

Déguisement Excentric

```
<?xml version='1.0' encoding='ISO−8859−1'?>
<!DOCTYPE classes SYSTEM "class.dtd">
<class name="avion_migrated" extends="entites">
 <attributes>

      <!−− Les attributs suivants sont partagés par avion et entites −−>
      <!−− [<taille:integer>, <pos:position>] −−>

      <!−− Les attributs suivants de la classe entites n'existaient␣pas␣dans␣la␣
          classe␣avion␣originale␣−−>
␣␣␣␣␣␣␣␣<!−−␣[<vitesse:integer>,␣<masse:integer>]␣−−>

␣␣␣␣␣␣␣␣<!−−␣Les␣attributs␣suivants␣n'existaient pas dans la classe entites −−>
      <attribute name="canons" type="integer"/>
      <attribute name="passagers" type="integer"/>
      <attribute name="profondeur" type="float"/>
      <attribute name="missiles" type="integer"/>
      <attribute name="autonomie" type="float"/>
      <attribute name="blindage" type="integer"/>
 </attributes>
</class>
```

3.5. Valuation des attributs de E

Une fois C converti en C_B, il ne reste plus qu'à envoyer E au moteur de monde, en changeant son type de classe $C \rightarrow C_B$.

Cependant, il nous reste à assigner des valeurs aux nouveaux attributs qui ont été rajoutés !

Plusieurs méthodes sont là encore possibles. La méthode la plus évidente est d'utiliser les valeurs par défaut des attributs – soit les valeurs par défaut du type de donnée (0,"", etc.) soit les valeurs par défaut indiqué dans les classes. On peut en effet indiquer une valeur par défaut dans les fichiers XML de la façon suivante :

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE classes SYSTEM "class.dtd">
<class name="avion" extends="entites_guerre">
 <attributes>
     <attribute name="blindage" type="integer">20</attribute>
 </attributes>
</class>
```

D'autre possibilitées sont implémentables (classification multidimensionnelle, etc., voir [8, p.28]). Ce que le gérant d'interopérabilité fait pour le moment est :

1. instancier les attributs en utilisant la valeur par défaut de la classe
2. pour toutes les instances de C_h (la classe dont hérite C_B), on récupère les valeurs de leurs attributs, et on mets dans E_B la moyenne de ces valeurs.

On pourrait également implémenter le même principe, mais cette fois-ci en récupérant les valeurs des attributs de toutes les instances de C_h et de toutes les instances de classes *dérivées* de C_h.

4. L'imitation

L'imitation est un mécanisme d'analyse des comportements du point de vue statique (les contextes, les entités virtuelles en jeu, etc.) pour identifier à quels états du monde ils s'intéressent et quelles influences ils produisent pour agir. Le but de cette analyse est de se comporter suivant les mêmes règles que celles du monde RV étranger.

On essaie d'améliorer le degré d'interopérabilité de l'entité migrante. On peut déterminer les comportements de l'entité qui ne sont pas valides (voir [1, pp.61-67]), c'est à dire par exemple des comportements dont les influences ne sont utilisées par aucune loi du monde. Pour chaque comportement invalide de C_B, on essaie de trouver un comportement similaire dans les classes de M_B qui pourrait être utilisé à la place.

En fait, le principe de l'imitation ressemble assez fortement à ce qui a été fait pour le déguisement. On construit ainsi un treillis à partir des lois de comportement des entités du monde d'accueil M_B, en utilisant les différents paramètres de ces lois comme espace de propriétés E' du treillis.

On utilisera comme paramètres :
– la liste des *PreStates* et des *PreInfluences*
– les prédicats utilisés
– les influences générées
Soit ΣK_{i_p} l'ensemble de ces paramètres pour les comportements des classes du monde d'accueil M_B, et ΣC_p l'ensemble des constituants du comportement invalide traité. On détermine I l'intersection entre ces deux ensembles :

$$I = \sum K_{i_p} \cap \sum C_p$$

et on recherche le nœud H_{min} le plus compatible de plus faible cardinal ([1, pp.108-109],[8, p.31]) [2], on liste alors les descendants par cardinalité. Si le comportement est complet – et que le comportement n'utilise que les attributs possédés par l'entité (vérification sur self) – on peut ajouter le comportement, le déguisement est terminé pour ce comportement. Dans le cas de plusieurs comportements complets dans le nœud courant, on peut en choisir un aléatoirement, voire en analysant les buts de l'intrus.

Dans le cas contraire (comportement incomplet), on passe au descendant suivant.

[2] ce qui corresponds en fait à l'ancêtre commun des nœuds de même niveau de compatibilité

Conclusion 4

Les différents travaux menés par M.Michel Soto et l'équipe RV sur ce thème montrent bien que l'approche sémantique, couplée à l'approche *influence/réaction*, donne des résultats intéressants concernant l'interopérabilité dans les mondes de réalité virtuelle.

De plus, l'architecture originale du projet mériterait que d'autres voies soient ouvertes, par exemple pour essayer de déterminer les problèmes et solutions éventuelles dans une optique de répartition du serveur de monde sur différentes machines. Même problématique concernant les problèmatiques de visualisation du monde.

On voit également que les différents algorithmes fonctionnent bien sur des schémas classiques de programmation orientée objet (modèle actuel du moteur), et sont d'ailleurs utilisés par d'autres équipes dans cette optique (atelier logiciel, refactorisation de code).

Concernant le logiciel, des choses restent à faire (implémentation de l'exploration) mais néanmoins l'intégration des différents composants (moteur, gestionnaire d'interopérabilité) devrait faciliter le travail futur sur ce sujet.

Quelques Fichiers Sources *A*

1. DTD

Voici les formats XML utilisés et les DTD correspondantes.

1.1. DTD des classes

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!ELEMENT class (attributes?, behaviors?, reactions?)>
      <!ATTLIST class name CDATA #REQUIRED>
      <!ATTLIST class extends CDATA #IMPLIED>

<!ELEMENT attributes (attribute+) >
<!ELEMENT behaviors (behavior+) >
<!ELEMENT reactions (reaction+) >

<!ELEMENT attribute (#PCDATA)>
      <!ATTLIST attribute name CDATA #REQUIRED>
      <!ATTLIST attribute type CDATA #REQUIRED>

<!ELEMENT behavior (prestates?,predicates?,postinfluences?)>
      <!ATTLIST behavior name CDATA #REQUIRED>

<!ELEMENT prestates (state+)>

<!ELEMENT state EMPTY>
      <!ATTLIST state name CDATA #REQUIRED>
```

```
      <!ATTLIST state entity CDATA #REQUIRED>

<!ELEMENT predicates (predicate+)>
<!ELEMENT predicate EMPTY>
      <!ATTLIST predicate value CDATA #REQUIRED>

<!ELEMENT postinfluences (influence+)>

<!ELEMENT influence EMPTY>
      <!ATTLIST influence name CDATA #REQUIRED>
      <!ATTLIST influence entity CDATA #REQUIRED>
      <!ATTLIST influence value CDATA #REQUIRED>

<!ELEMENT reaction (prestates?,preinfluences?,predicates?,operations?,poststates?)>
      <!ATTLIST reaction name CDATA #REQUIRED>

<!ELEMENT preinfluences (preinfluence+)>

<!ELEMENT preinfluence EMPTY>
      <!ATTLIST preinfluence name CDATA #REQUIRED>
      <!ATTLIST preinfluence entity CDATA #REQUIRED>

<!ELEMENT operations (operation+)>

<!ELEMENT operation EMPTY>
      <!ATTLIST operation type CDATA #REQUIRED>
      <!ATTLIST operation using CDATA #REQUIRED>
      <!ATTLIST operation name CDATA #REQUIRED>

<!ELEMENT poststates (poststate+)>

<!ELEMENT poststate EMPTY>
      <!ATTLIST poststate value CDATA #REQUIRED>
```

1.2. DTD des instances

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!ELEMENT instances (instance+)>

<!ELEMENT instance (attributes?)>
      <!ATTLIST instance name CDATA #REQUIRED>
      <!ATTLIST instance class CDATA #REQUIRED>

<!ELEMENT attributes (attribute+)>

<!ELEMENT attribute (#PCDATA)>
      <!ATTLIST attribute name CDATA #REQUIRED>
```

2. Exemples de fichiers XML

2.1. Classe du monde

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE class SYSTEM "class.dtd">

<!--
 Exemple
 Ce fichier contient la definition d'une classe
 dont une instance sera utilisee comme classe du monde.
 Nicolas Roard <nicolas@roard.com>
-->

<class name="world">
<!--
 On commence par definir les attributs de la classe et leur type.
 le contenu d'un tag "attribute" peut contenir une valeur par defaut
 pour l'attribut.
-->

<attributes>
 <attribute name="vent_ouest" type="float">10</attribute>
 <attribute name="vent_est" type="float">11</attribute>
</attributes>
```

```
<!--
 On definit ensuite des comportements du monde
-->

<behaviors>

<!--
 Il s'agit de deux vents (dont la puissance est determinee a partir
 des deux attributs "vent_ouest" et "vent_est" de la classe)
 Ces comportements s'appliquent a toutes les instances disposant
 d'un attribut "position"
 Dans les valeurs on constate l'utilisation de "self", qui reference
 l'instance courante lors de l'execution. "world" est autre mot-cle
 valable (plutot donc pour des autres classes ;-)
-->

 <behavior name="vent_ouest">
  <prestates>
   <state name="position" entity="e1"/>
  </prestates>
  <postinfluences>
   <influence name="inf_vent" entity="e1" value="self.vent_ouest"/>
  </postinfluences>
 </behavior>

 <behavior name="vent_est">
  <prestates>
   <state name="position" entity="e1"/>
  </prestates>
  <postinfluences>
   <influence name="inf_vent" entity="e1" value="- self.vent_est"/>
  </postinfluences>
 </behavior>

</behaviors>

<!--
 On definit enfin une reaction a l'influence "inf_vent" generee par les
  precedents comportements
-->
```

```
<reactions>
 <reaction name="reaction_vent">

  <!--
   On recupere d'abord toutes les entites ayant un etat "position"
  -->

  <prestates>
   <state name="position" entity="e1"/>
  </prestates>

  <!--
   Sur ces entites, on selectionne l'influence "inf_vent" si elle existe
  -->

  <preinfluences>
   <preinfluence name="inf_vent" entity="e1"/>
  </preinfluences>

  <!--
   On effectue une operation ensembliste sur les influences "inf_vent"
   recues : une somme. Le resultat de l'operation aura pour nom "result"
  -->

  <operations>
   <operation type="sum" using="inf_vent" name="result"/>
  </operations>

  <!--
   Enfin on determine l'impact de la reaction sur les etats.
   Ici on ajoute simplement la somme des influences "inf_vent" a
   l'attribut position des entites e1 (selectionnees dans les prestates)
  -->
  <poststates>
   <poststate value="e1.position = e1.position + result"/>
  </poststates>

 </reaction>
</reactions>
```

```
</class>
```

2.2. Exemple de classes

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE classes SYSTEM "class.dtd">

<class name="entites">
      <attributes>
            <attribute name="positionx" type="float"/>
            <attribute name="positiony" type="float"/>
            <attribute name="taille" type="integer"/>
            <attribute name="masse" type="integer"/>
            <attribute name="vitesse" type="integer"/>
      </attributes>
</class>

<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE classes SYSTEM "class.dtd">

<class name="entites_guerre" extends="entites">
      <attributes>
            <attribute name="blindage" type="integer"/>
      </attributes>
</class>

<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE classes SYSTEM "class.dtd">

<class name="navires_guerre" extends="entites_guerre">
      <attributes>
            <attribute name="canons" type="integer"/>
      </attributes>
</class>

<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE classes SYSTEM "class.dtd">

<class name="sousmarins" extends="entites_guerre">
```

```
      <attributes>
            <attribute name="missiles" type="integer"/>
            <attribute name="profondeur" type="float"/>
      </attributes>
</class>
```

2.3. Instances

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE instances SYSTEM "instances.dtd">

<instances>
      <instance name="porteavion" class="navires_guerre">
            <attributes>
                  <attribute name="taille">300</attribute>
                  <attribute name="masse">300000</attribute>
            </attributes>
      </instance>
      <instance name="destroyer" class="navires_guerre">
            <attributes>
                  <attribute name="taille">100</attribute>
                  <attribute name="masse">100000</attribute>
            </attributes>
      </instance>
      <instance name="sousmarin" class="sousmarins">
            <attributes>
                  <attribute name="taille">150</attribute>
                  <attribute name="masse">150000</attribute>
            </attributes>
      </instance>
      <instance name="voilier" class="entites">
            <attributes>
                  <attribute name="taille">20</attribute>
                  <attribute name="masse">20000</attribute>
            </attributes>
      </instance>
</instances>
```

3. Parser

Voici un exemple de fichier cup utilisé en tant que parser pour les prédicats ; les parsers des influences et des réactions sont à peu près identiques. Pour générer les parsers, JavaCup et JFlex ont été utilisés, respectivement comme générateur de parser et comme générateur de lexer.

```
/*
 * Nicolas Roard
 * Parser gérant les prédicats
 * basé sur le précédent parser de Xavier Prat et Christopher Lopes
 *
 * Pour l' utiliser :
 *                  java java_cup.Main < ycalc-predicates.cup
 */

package scanner ;

import java_cup.runtime.*;

/*
 * Code ajouté au parser
 */

parser code
{:

      Lexer lexer ;

      // Contient tous les noms d'alias d' attribut et leur type
      InstancesList identifiers ;

      public ParserPredicates (Lexer lexer, InstancesList identifiers)
      {
            this.lexer = lexer;

            if (identifiers == null) {
                  this.identifiers = new InstancesList ();
            }
            else {
                  this.identifiers = identifiers;
```

```
            }
      }
:} ;

/*
 * On utilise le lexer créé par jflex
 */

scan with
{:
      return lexer.next_token();
:};

/* ----------Section de Déclaration des Terminaux et non Terminaux
    ----------*/

/* Terminaux (jetons retournés par le scanneur).

Les terminaux qui ne possèdent pas de valeur sont déclarés en premier, suivis
des terminaux qui ont une valeur, dans notre cas une valeur entière ou réelle .
*/
terminal INFERIOR, SUPERIOR, EQUAL, RANDOM, PLUS, MINUS, TIMES, DIVIDE, LPAREN,
    RPAREN, LN ;
terminal POWER, SQRT, EXP, ABS, MODULO, TRUE, FALSE, PI, E ;
terminal COS, SIN, TAN, ARCCOS, ARCSIN, ARCTAN, FLOOR, CEIL, ENT ;
terminal Integer INTEGER ;
terminal Double FLOAT ;
terminal String ID, STRINGLITTERAL ;


/* Non Terminaux utilisés dans la section grammaire. */
non terminal AliasType expr, factor, function, term;
non terminal AliasType expr_id, factor_id, function_id, term_id ;
non terminal Boolean result;
non terminal String expr_string ;

result ::= expr:e1 INFERIOR expr:e2 {: RESULT = e1.inferior (e2); :}
      | expr:e1 SUPERIOR expr:e2 {: RESULT = e1.superior (e2); :}
      | expr:e1 EQUAL expr:e2 {: RESULT = e1.equal (e2); :}
;
```

```
expr ::= expr:e PLUS factor:f {: RESULT = e.plus (f); :}
      | expr:e MINUS factor:f {: RESULT = e.minus (f); :}
      | factor:f {: RESULT = f; :}
;

factor ::= factor:f TIMES function:t {: RESULT = f.times (t); :}
      | factor:f DIVIDE function:t {: RESULT = f.divide (t); :}
      | factor:f MODULO function:t {: RESULT = f.modulo (t); :}
      | function:t {: RESULT = t; :}
;

function ::= COS function:f
            {: RESULT = new AliasType( new Double(Math.cos( f.doubleValue()) ), "
                float") ; :}
         | SIN function:f
            {: RESULT = new AliasType( new Double(Math.sin( f.doubleValue()) ), "
                float") ; :}
         | TAN function:f
            {: RESULT = new AliasType( new Double(Math.tan( f.doubleValue()) ), "
                float") ; :}
         | ARCCOS function:f
            {: RESULT = new AliasType( new Double(Math.acos( f.doubleValue()) ), "
                float") ; :}
         | ARCSIN function:f
            {: RESULT = new AliasType( new Double(Math.asin( f.doubleValue()) ), "
                float") ; :}
         | ARCTAN function:f
            {: RESULT = new AliasType( new Double(Math.atan( f.doubleValue()) ), "
                float") ; :}
         | SQRT function:f
            {: RESULT = new AliasType( new Double(Math.sqrt( f.doubleValue()) ), "
                float") ; :}
         | ABS function:f
            {: RESULT = new AliasType( new Double(Math.abs( f.doubleValue()) ), f.
                type ) ; :}
         | EXP function:f
            {: RESULT = new AliasType( new Double(Math.exp( f.doubleValue()) ), "
                float" ) ; :}
         | LN function:f
```

```
            {: RESULT = new AliasType( new Double(Math.log( f.doubleValue()) ), "
                float" ) ; :}
          | FLOOR function:f
            {: RESULT = new AliasType( new Double(Math.floor( f.doubleValue()) ), "
                int" ) ; :}
          | ENT function:f
            {: RESULT = new AliasType( new Double(Math.floor( f.doubleValue()) ), "
                int" ) ; :}
          | CEIL function:f
            {: RESULT = new AliasType( new Double(Math.ceil( f.doubleValue()) ), "
                int" ) ; :}
          | MINUS function:f
            {: RESULT = new AliasType( new Double(0 - f.doubleValue()), f.type)
                ; :}
          | RANDOM function:f
            {: RESULT = new AliasType( new Double( Math.floor( Math.random()*f.
                doubleValue() ) ), "int" ); :}
          | term:t
            {: RESULT = t; :}
;

term ::= LPAREN expr:e RPAREN
            {: RESULT = e; :}
          | INTEGER:i
            {: RESULT = new AliasType( i, "int"); :}
          | FLOAT:f
            {: RESULT = new AliasType( f, "float"); :}
          | PI
            {: RESULT = new AliasType( new Double(Math.PI), "float"); :}
          | E
            {: RESULT = new AliasType( new Double(Math.E), "float"); :}
          | term:t POWER LPAREN expr:e RPAREN
            {: RESULT = t.power (e); :}
          | term:t POWER INTEGER:n
            {: RESULT = t.power (new AliasType (n.doubleValue (), "int")); :}
          | term:t POWER FLOAT:n
            {: RESULT = t.power (new AliasType (n.doubleValue (), "float")); :}
          | ID:identifier
            {:
                  Object number = parser.identifiers.valueOfIdentifier (identifier)
                      ;
```

```
                Class numberClass = number.getClass ();

                if (numberClass == Class.forName ("java.lang.Integer"))
                {
                      RESULT = new AliasType ((Number) number, "int");
                }
                else if (numberClass == Class.forName ("java.lang.Float"))
                {
                      RESULT = new AliasType ((Number) number, "float");
                }
                else if (numberClass == Class.forName ("java.lang.String"))
                {
                      RESULT = new AliasType ((String) number, "string");
                }
          :}
;
```

– Bibliographie –

[1] S. Allongue, *Méthodes de modélisation pour l'interopérabilité des mondes virtuels, fondées sur les ontologies et des mécanismes d'apprentissage*, (1998). Thèse, Université Pierre et Marie Curie.

[2] C. Boucher, *Visualisation de monde virtuel*, (Septembre 2002). Rapport de Stage, Magistère d'Informatique Appliquée d'Île-de-France.

[3] M. Courtine, *Changements de représentation pour la classification conceptuelle non supervisée de données complexes*, (2002). Thèse, Université Pierre et Marie Curie.

[4] M. Courtine et G. Harmel, *L'interopérabilité des mondes virtuels*, (1998). Rapport de Stage, Université Pierre et Marie Curie.

[5] R. Godin, , R. Missaoui, et H. Alaoui, *Incremental concept formation algorithms based on galois (concept) lattice*, (1995). Computational Intelligence, Vol.11, No.2.

[6] R. Godin, G. Mineau, R. Missaoui, et H. Mili, *Méthodes de classification conceptuelle basée sur les treillis de galois et applications*, (1995). Paru dans Revue d'Intelligence Artificielle, 1995, 9(2), pp. 105-137.

[7] X. Prat et W. Ramdani, *Moteur de monde virtuel*, (Septembre 2001). Rapport de Stage, Magistère d'Informatique Appliquée d'Île-de-France.

[8] J.-J. Puig, *Mondes virtuels interopérables par apprentissage*, (2001). Rapport de Stage, Université Pierre et Marie Curie.

[9] M. Soto et S. Allongue, *Modeling methods for reusable and interoperable virtual entities in multimedia virtual worlds*, (2001). Paru dans Multimedia Tools and Applications, 16, pp. 161-177.

– Index –