4

Intégration opérationnelle

Résumé

Nous avons noté en introduction que l'intégration de modèles hétérogènes implique l'interopérabilité des modèles. Cette dernière existe de fait pour les modèles issus d'un même formalisme et pour lesquels des simulateurs compatibles existent. Les choses deviennent plus compliquées lorsqu'il s'agit d'intégrer des formalismes ou des paradigmes hétérogènes.
Dans cette partie, nous proposons un cadre logique et logiciel, un *framework*, qui prend en compte les avancées de ces dernières années en terme de couplage de modèles hétérogènes. Notre *framework* s'articule autour de quatre couches qui définissent différents niveaux pour appréhender l'hétérogénéité. Nous proposons des solutions d'intégration pour chaque couche. De plus, nous proposons de mieux définir le concept d'expériences virtuelles pour la simulation des systèmes complexes. Dans ce cadre, nous proposons notamment deux applications XML pour le couplage de modèles et la définition de plans d'expériences.

Sommaire

4.1 Introduction 96
4.2 Un *framework* pour l'intégration de modèles hétérogènes . . 97
4.2.1 La couche opérationnelle 98
4.2.2 La couche simulation 100
4.2.3 La couche modèle 103
4.2.4 La couche sémantique 105
4.3 Description des modèles 106
4.3.1 Description de l'espace 107
4.3.2 Description du temps 108
4.3.3 Description d'un modèle 110
4.3.4 Description des données 113
4.4 Description des expériences 116
4.4.1 Laboratoire virtuel 116
4.4.2 Spécification des expériences 118
4.5 Discussion et conclusion 121

4.1 Introduction

Nous reprenons ici le problème de l'intégration de modèles hétérogènes au niveau opérationnel. Nous commençons par énumérer les différentes questions que nous nous sommes posées. Ces questions présentent le cadre général de ce chapitre.

- comment coupler deux modèles issus de paradigmes différents ? Comment en gérer la cohérence ?
- dispose-t-on d'un formalisme opérationnel ?
- comment faire communiquer des modèles de simulation conçus séparément ?
- peut-on s'affranchir des problèmes techniques liés à la simulation (donc à l'exécution de modèles) ?
- comment réutiliser et intégrer des modèles déjà existants ?

Comme nous l'avons dit en introduction de cette thèse, l'intégration de modèles hétérogènes est en fait un problème d'interopérabilité aux différents niveaux d'abstraction décrits par la figure 1.2 page 15. Nous donnons notre propre cadre conceptuel et opérationnel (que nous appelons *framework*) à l'intérieur duquel nous faisons différentes propositions pour résoudre les problèmes d'interopérabilité.

En génie logiciel, plusieurs types de *frameworks* existent : les *frameworks* projet (il faut cadrer le projet dans ses grandes lignes et se référer à des projets-types), les *frameworks* de conception et les *frameworks* de développement. Les *frameworks* de conception et de développement sont plus proches de ce que nous cherchons à mettre en place dans le domaine de la modélisation et de la simulation de systèmes complexes. Un *framework* de conception est une sorte de livre de recettes. La première chose à mettre en place dans un *framework* de conception est un vocabulaire commun, c'est-à-dire une ontologie servant à décrire les choses (en génie logiciel, on parlera de briques logicielles ou matérielles). Ce point est essentiel. Le deuxième aspect des *frameworks* de conception est regroupé sous le terme de *design patterns*. Face à des problèmes récurrents, des *design patterns* sont définis afin de répondre plus rapidement aux problèmes. Le *design pattern Model-View-Controler* (MVC) est un exemple. Un ensemble de données est encapsulé dans un *Model*, le *Controler* permet d'y accéder et la vue est une représentation du *Model*. Pour un même *Model*, plusieurs vues différentes peuvent être définies sans toucher à la partie *Model*. Ce *design pattern* est très utilisé (par exemple, dans la notion d'interface multi-documents - MDI). La troisième forme de *framework* (de développement) est également une source d'inspiration. En effet, ce dernier doit structurer le développement, et non pas simplement offrir des bibliothèques de classes ou de fonctions. Ainsi, un *framework* de développement doit guider le développeur et homogénéiser (en partie) la façon de coder les modèles. En nous inspirant des définitions de *frameworks* de conception et de développement, nous donnons une définition liée à notre approche.

> Un *framework* pour l'intégration de modèles hétérogènes consiste à définir un cadre logiciel et des règles minimales de construction de modèles afin de permettre l'interopérabilité des modèles.

Le cadre logiciel définit l'ensemble des applications et des techniques nécessaires pour la mise en œuvre pratique de l'intégration. Les règles minimales de construction de modèles définissent l'ensemble des règles à suivre pour qu'un modèle soit compatible avec le *framework*.

Nous allons commençer par présenter notre *framework* d'intégration et proposer une syntaxe pour la description des modèles. De plus, l'utilisation des modèles dans le cadre d'expériences

nous a amené à définir une autre syntaxe pour la description des expériences de simulation. Nous concluons par une discussion à propos de notre approche.

4.2 Un *framework* pour l'intégration de modèles hétérogènes

Nous entrons dans cette section avec le dilemme suivant : conserver la diversité des modèles et en permettre l'unification, c'est-à-dire le couplage. Nous abordons cette question sous un angle opérationnel. La création de modèles et de simulateurs est une activité complexe que nous n'abordons pas ici ; nous nous focalisons sur l'intégration de modèles préexistants.

Le couplage de modèles dans un objectif de simulations réparties et comunicantes nous a conduits à organiser notre *framework* selon quatre niveaux d'abstraction (voir figure 4.1) : le niveau opérationnel, le niveau simulation, le niveau modèle et le niveau sémantique.

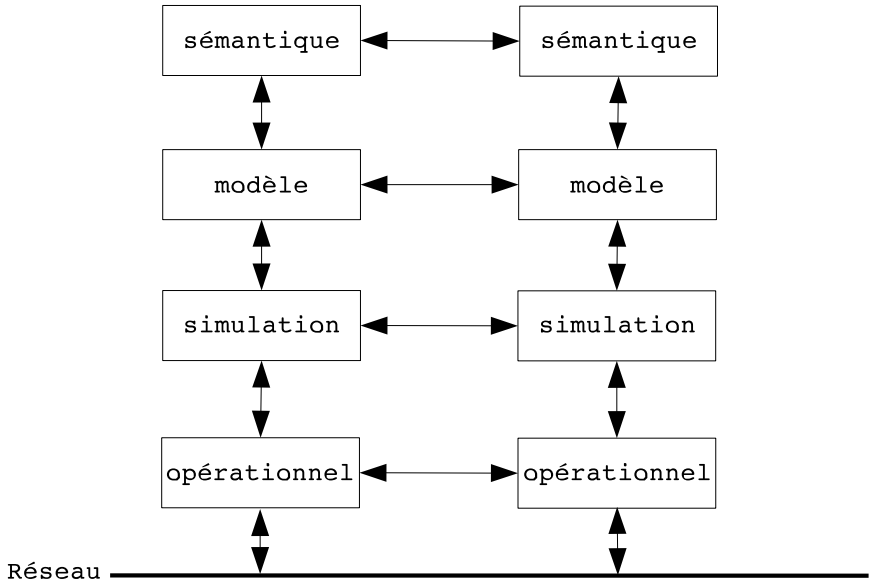

FIG. 4.1 – Hiérarchisation en quatre niveaux d'abstraction (ou couches) de notre *framework* pour l'intégration de modèles hétérogènes. Nous supposons ici deux modèles qui échangent des informations *via* le réseau. La communication entre couches (flèches horizontales) passe forcément par les niveaux inférieurs (flèches verticales).

Nous ne prétendons pas répondre totalement aux différents problèmes qui se posent pour chaque niveau (en particulier les deux derniers), mais allons néanmoins proposer un ensemble de solutions techniques et conceptuelles qui rendent opérationnel notre framework. Pour cela, le niveau opérationnel doit assurer la mise en œuvre de l'échange d'informations entre les éléments de la simulation dans un environnement réparti. Peu d'hypothèses doivent être faites sur la nature de cet environnement réparti. Nous nous sommes donc basés sur un environnement hétérogène au sens matériel, système d'exploitation, protocole et langage de programmation. Le niveau simulation a pour objectif de proposer une base algorithmique pour le couplage de simulateurs

sans spécificité propre à un type de simulateur. La couche modèle doit assurer la spécification des modèles quels que soient le formalisme et le paradigme. Une fois encore, cette couche doit autant que possible s'abstraire des spécificités propres aux formalismes et aux paradigmes afin d'offrir une notation unifiée des modèles. La dernière couche, la couche sémantique, doit permettre la spécification des éléments sémantiques d'un modèle (nature discrète ou continue de l'espace ou du temps manipulé dans un modèle par exemple).

L'idée retenue consiste à définir des bus de communication pour chaque couche. Un bus est un canal à travers lequel les niveaux identiques de notre *framework* communiquent (les flèches horizontales de la figure 4.1). La notion de communication est vue ici du point de vue général. Par exemple, pour la couche modèle, deux modèles communiquent dès lors qu'ils doivent échanger des données. Chaque bus est indépendant l'un de l'autre. Le choix d'un protocole de communication à un certain niveau ne doit pas imposer de contraintes aux niveaux inférieurs et supérieurs. Néanmoins, chaque niveau est en interaction avec ses niveaux voisins (flèches verticales de la figure 4.1). Les interactions entre niveaux peuvent être des traductions ou des encapsulations à l'image du modèle OSI. Conceptuellement, le *framework* se structure donc autour de quatre bus (un bus par couche) : le bus opérationnel, le bus de simulation, le bus des modèles et le bus sémantique. Nous allons maintenant passer en revue les différentes couches en exposant les solutions possibles pour la mise en œuvre de chaque bus.

4.2.1 La couche opérationnelle

Nous l'avons dit en introduction de cette thèse, cette couche est très étudiée et très prolifique en matière de solutions adoptées pour régler le problème de l'hétérogénéité. Les problèmes à ce niveau sont principalement techniques. Nous nous contentons donc ici de présenter les différentes possibilités qui s'inscrivent dans notre *framework*.
Notre couche opérationnelle se présente sous la forme d'un bus où viennent se connecter les éléments de la couche supérieure c'est-à-dire les simulateurs. Le bus opérationnel doit offrir les services liés à la communication, à la distribution des simulateurs indépendamment des protocoles et des langages. Nous pouvons schématiser la couche opérationnelle par la figure 4.2.

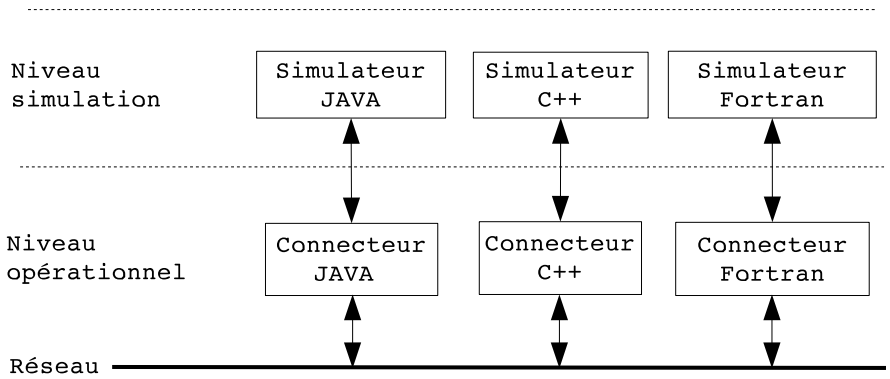

FIG. 4.2 – Couche opérationnelle de notre *framework*. Elle se situe entre le réseau (qui symbolise la couche matérielle) et le niveau simulation.

L'infrastructure repose principalement sur la notion de connecteur. Un connecteur est un

composant logiciel encapsulant l'accès au réseau, le transport d'informations *via* le réseau et l'indépendance par rapport au langage de programmation. Les connecteurs font le lien entre les simulateurs et l'infrastructure physique de communication. Comme nous l'avons vu en introduction de cette thèse avec HLA, les connecteurs doivent proposer une interface universelle (le RTI). Il existe plusieurs technologies possibles pour l'implémentation de ces connecteurs (voir figure 4.3).

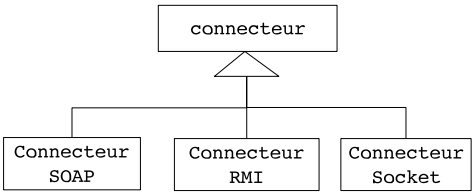

FIG. 4.3 – Exemples de connecteurs possibles pour la couche opérationnelle. Les connecteurs sont liés au langage utilisé pour la couche simulation.

Ci-dessous, nous présentons brièvement quatre cas de figure ayant été étudiés dans le cadre de projets de fin d'années de DESS et du stage de DEA de G. Quesnel [Que03] [QDNR03] :

- tous les simulateurs sont implémentés par des langages différents qui proposent une API[60] SOAP[61],
- certains simulateurs ne disposent pas de l'API SOAP mais proposent les sockets,
- tous les simulateurs sont programmés dans le même langage,
- tous les simulateurs sont en C/C++ et sont exécutés sur un cluster Linux.

Dans tous les cas, le connecteur a pour rôle de transformer les invocations de méthodes et/ou les objets SOAP en invocations et objets (ou structures) correspondants dans les langages utilisés par les simulateurs. L'idée est la même dans le deuxième cas de figure avec les sockets. Ce connecteur est mis à disposition dès lors que le langage utilisé par le simulateur ne dispose pas de l'API SOAP. De plus, SOAP n'est pas obligatoirement la technologie à utiliser. En effet, si la fréquence des échanges entre les simulateurs devient importante, le temps passé à la communication devient prohibitif par rapport au temps de simulation. Il est alors intéressant d'opter pour des moyens de communication plus rapides mais plus difficiles à mettre en place. Avec SOAP, les objets peuvent être transmis *via* le média de communication (après une phase de traduction en XML assurée par SOAP) ce qui n'est pas le cas avec les sockets : il faut que le programmeur prenne en charge les phases de traduction. Le troisième cas de figure est le plus simple. Si les simulateurs sont homogènes en terme de langage et en particulier en Java, l'échange de données et l'invocation de méthodes distantes peuvent être pris en charge par une implémentation Java de la technologie RPC[62] et en particulier RMI[63]. Nous avons pris pour exemple Java mais la remarque reste vraie pour les autres langages (ONC/RPC[64][65] pour C++, par exemple).

Les trois premiers cas de figures utilisent des technologies orientées vers les réseaux de type

[60] Application Programming Interface
[61] Il existe une alternative à SOAP avec XML/RPC disponible en Java, C++...
[62] Remote Procedure Call
[63] Remote Method Invocation
[64] Il existe aussi une version pour Java
[65] http ://www.plt.rwth-aachen.de/ks/english/oncrpc.html

LAN[66] et WAN[67] connectant un ensemble de machines « classiques » sous Linux, Windows, MacOS X, ... Le dernier cas de figure est particulier en spécialisant les technologies vers un type d'architecture précis. Celle-ci consiste en un ensemble de processeurs munis de mémoire non partagée. Les processeurs sont aussi accompagnés d'une interface réseau afin d'être connectés entre eux. Le type de réseau est LAN avec des débits plus élevés que dans un LAN « classique ». Les débits fluctuent entre la centaine de MegaBits et le GigaBit. Le système d'exploitation qui est embarqué est spécifique (très souvent une distribution de Linux légèrement adaptée) et il est augmenté de bibliothèques de communications spécifiques ; dans notre cas il s'agit de MPI[68]. Cette bibliothèque est très proche des sockets et permet de mettre en œuvre les algorithmes nécessaires à la communication entre simulateurs. Un type de connecteurs doit donc être disponible pour ce type d'architectures. Il faut noter que le langage de prédilection est le langage C ce qui n'exclut pas les autres langages mais interdit leur couplage avec MPI [69].

En conclusion, la couche opérationnelle permet à la couche simulation d'être indépendante de l'infrastructure matérielle et logicielle pour la communication entre processus. Elle n'a pas pour rôle de garantir la bonne synchronisation des processus et ne connaît rien des simulateurs. Ceci est confié à la couche simulation.

4.2.2 La couche simulation

La couche simulation a pour objectif d'assurer l'exécution correcte[70] des modèles couplés. L'algorithmique des simulateurs est, dans la majorité des cas, spécifique au formalisme adopté dans la phase de modélisation. Ce constat peut s'avérer être un problème. En effet, il faut alors répondre à la question :

« comment coupler deux simulateurs dont l'algorithmique est très différente ? »

Considérons, par exemple, une partie de système modélisée à l'aide d'un système d'équations différentielles et une autre partie par un réseau de Petri. Nous choisissons pour les équations différentielles une simulation programmée à l'aide d'un algorithme d'intégration numérique de type Runge-Kutta. Pour le réseau de Petri, nous utilisons tout simplement l'algorithme d'évolution du marquage synchrone[71]. Il paraît assez évident que le couplage de deux simulateurs aussi différents n'est pas simple.
Pour résoudre ce type de problèmes de couplage de simulateurs, nous nous sommes orientés vers les travaux de Zeigler [Zei76] [ZKP00] et le formalisme DEVS. Nous avons déjà présenté DEVS en introduction et dans le chapitre précédent. Rappelons simplement ici que DEVS est un formalisme abstrait pour la modélisation à évènements discrets et que ce formalisme a la prétention d'offrir à la fois l'encapsulation d'autres formalismes mais aussi les simulateurs associés. Nous donnons les algorithmes de base des simulateurs abstraits DEVS en annexe D[72]. D'autres algo-

[66] Local Area Network
[67] Wide Area Network
[68] Message Passing Interface
[69] Il existe quelques implémentations Java
[70] Le lecteur intéressé peut se reporter au livre de Zeigler *et al.* [ZKP00] pour une introduction au concept de morphisme qui permet de mettre en relation les modèles formels et leurs simulateurs, et ainsi de vérifier si une simulation est « correcte ».
[71] Toutes les transitions franchissables sont validées en même temps. Si des conflits existent un tirage aléatoire est effectué.
[72] Nous conseillons au lecteur n'ayant pas lu le chapitre précédent de se référer à la partie 3.2 page 50 pour une présentation générale de DEVS.

rithmes peuvent être trouvés dans [ZKP00] notamment pour les simulations distribuées. DEVS contient de façon intrinsèque les notions de hiérarchisation et de couplage. Ainsi, les algorithmes des simulateurs abstraits permettent la simulation de tout modèle DEVS ou compatible avec DEVS (nous approfondissons cette notion un peu plus loin). Le formalisme DEVS nous offre non seulement un cadre formel de spécification de modèles mais aussi des mécanismes opérationnels de simulation. C'est ce point qui nous a fait choisir DEVS pour la couche simulation.

Si nous voulons utiliser les simulateurs abstraits pour la couche simulation, quelles sont les contraintes pour un concepteur de simulateurs ? Les algorithmes des simulateurs abstraits reposent pleinement sur une spécification DEVS, ce qui implique *a priori* qu'il soit nécessaire de réaliser une spécification complète du modèle en DEVS. Deux approches sont en fait possibles : le *mapping* DEVS ou le *wrapping* DEVS.
Le *mapping* consiste à spécifier totalement le modèle en DEVS et ce quel que soit le formalisme utilisé pour la couche modèle. C'est ce que nous avons fait dans le chapitre précédent en spécifiant un modèle d'agents réactifs situés totalement en DEVS. Les travaux de Jacques et Wainer sont un autre exemple de *mapping* [JW02]. Ces travaux s'intéressent à la modélisation à base de réseaux de Petri et Jacques et Wainer proposent une spécification DEVS du formalisme. Il est alors possible de transformer les modèles à base de réseaux de Petri en modèles DEVS. Cette approche permet de garantir une solution DEVS pour la couche simulation et pour la couche modèle.
L'autre approche (le *wrapping*) est un compromis entre la volonté de disposer d'un noyau unifié de simulation qui couple des simulateurs éventuellement non DEVS. Cette idée est initialement apparue au cours de nos réunions de travail avec les membres du LISC[73] de Clermont-Ferrand. Le simulateur doit mettre en œuvre une liste de fonctions qui entrent dans la logique algorithmique des simulateurs abstraits. Le *wrapper* est donc une sorte d'interface au simulateur adapté au formalisme utilisé (voir figure 4.4). Le travail du concepteur de simulateurs « compatible DEVS » se résume alors à la construction de ce *wrapper*[74] ou à l'utilisation du *wrapper* existant pour le formalisme qu'il a choisi.

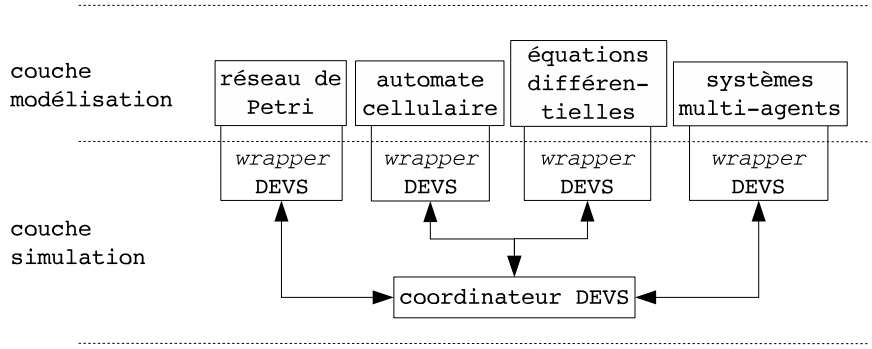

FIG. 4.4 – Schématisation des *wrappers* DEVS. Ce sont des interfaces fonctionnelles basées sur l'algorithmique des simulateur abstraits (voir le texte pour les détails).

L'étude des algorithmes des simulateurs abstraits permet d'identifier six fonctions utiles (voir figure 4.5). L'idée repose sur le fait que DEVS n'impose pas de formalisme pour les fonctions de

[73]Laboratoire d'Igénieurie des Systèmes Complexes.
[74]Nous parlons aussi d'interface DEVS

transition ou de sortie. Ainsi, nous pouvons « cacher » derrière ces fonctions un simulateur écrit pour un formalisme quelconque.

```
public abstract EventList getOutputFunction(Time p_currentTime);
public abstract Time      getTimeAdvance();
public abstract void      init();
public abstract void      processInternalEvent(InternalEvent p_event);
public abstract void      processExternalEvent(ExternalEvent p_event);
public abstract void      finish();
```

FIG. 4.5 – Définition des fonctions du *wrapper* en Java. Cette interface est définie par six fonctions issues des simulateurs abstraits de DEVS. L'ensemble des états S n'apparaît pas ici ; il fait partie des attributs du modèle (au sens de la programmation objet).

Rappelons que la spécification DEVS est une structure composée par un vecteur d'états S, un ensemble de ports d'entrée X et de ports de sortie Y, de deux fonctions de transitions (l'une interne δ_{int} et l'autre externe δ_{ext}), d'une fonction de sortie $\lambda(S)$ et d'une fonction d'avancement du temps $ta(S)$. Nous retrouvons dans l'interface DEVS (le *wrapper*) les deux fonctions de transitions sous forme de fonctions de traitement des évènements externes (*processExternalEvent*) et de fin d'état (*processInternalEvent*). Ces deux fonctions respectent la spécification DEVS telle que : $\delta_{int} : S \rightarrow S$ et $\delta_{ext} : S \times X \rightarrow S$. Le paramètre de la fonction *processInternalEvent* ne fait pas partie de la spécification DEVS mais permet dans notre cas de transporter certaines informations telles que la date d'occurrence de l'évènement, par exemple.

Le traitement de l'évènement d'initialisation est représenté par une simple fonction (*init*). La fonction de sortie se traduit par une fonction admettant en paramètre la date courante afin d'estampiller les évènements qui seront générés et retourne une liste d'évènements. La notion d'évènement est mise en œuvre par la classe *Event*. Deux sous-classes sont disponibles *InternalEvent* et *ExternalEvent*. Seule la classe *ExternalEvent* est instantiable par le concepteur d'un *wrapper*. Les instances de la classe *InternalEvent* sont créées par les simulateurs et représentent l'évènement de fin d'état. L'opération de construction d'un évènement externe par la fonction de sortie est très simple : cette fonction connaît seulement le port de sortie sur lequel l'évènement doit être émis et la date d'occurence de cet évènement.

La dernière fonction à implémenter est la fonction d'avancement du temps. Elle est invoquée par le simulateur abstrait pour déterminer la date de fin de l'état courant. L'en-tête de la fonction fait donc apparaître tout logiquement une date (*Time*) en retour d'appel de cette fonction. Si ces six fonctions sont implémentées, alors nous disposons d'un *wrapper* DEVS. Celui-ci peut être intégré dans une simulation distribuée et le modèle qu'il met en œuvre peut être couplé de manière totalement transparente.

Comme le montre la figure 4.4, un simulateur s'intègre à la simulation globale en se connectant sur un bus logiciel défini par l'interface DEVS. L'architecture du bus repose sur la relation entre un coordinateur et des simulateurs. Telle que nous la définissons ici, la couche simulation est une couche logicielle qui a pour principal objectif d'exécuter les modèles. Elle doit donc intégrer les algorithmes nécessaires pour les bases de la simulation distribuée et le contrôle du cycle d'exécution de la simulation. Ces algorithmes ont été développés notamment par Kim [KK96]

qui introduit la notion de DEVS-bus. Il a poursuivi ses travaux par une intégration de DEVS-bus dans HLA [KK98]. La figure 4.6 schématise l'architecture que nous avons choisie pour notre couche simulation. Cette figure reprend celle présentée précédemment pour la couche simulation et celle présentée pour la couche opérationnelle. De plus, nous nous sommes fortement inspirés de l'architecture proposée par Kim et Zeigler [ZKP00].

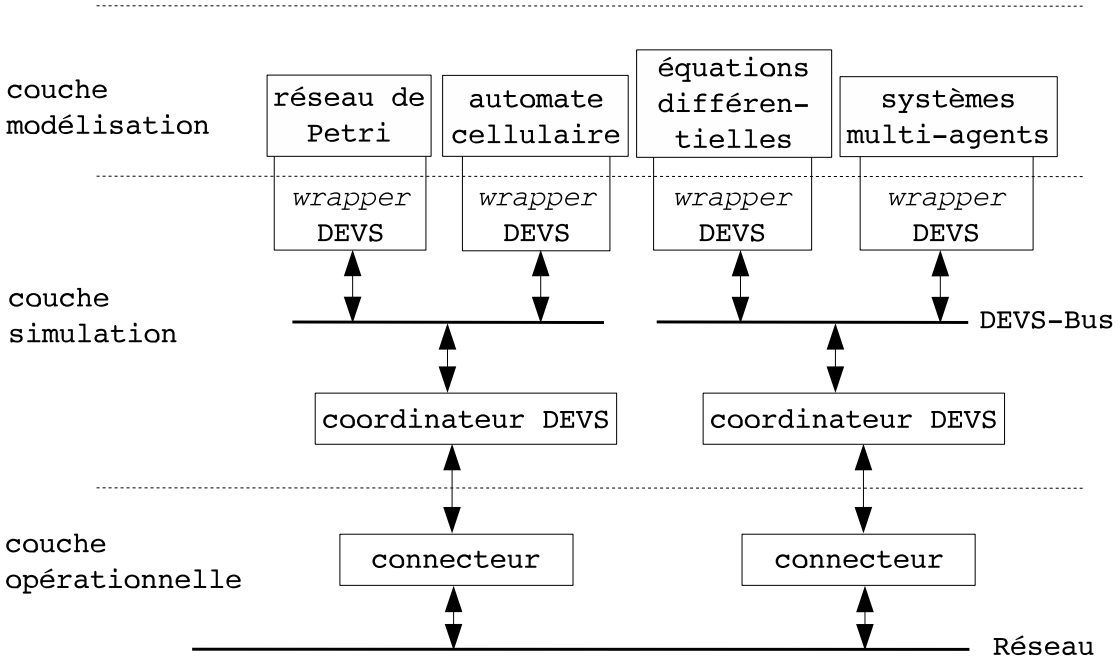

FIG. 4.6 – Intégration de DEVS-bus dans notre *framework*. Les coordinateurs DEVS peuvent être exécutés sur une même machine ou être distants. Dans ce cas, ils échangent les données *via* les connecteurs au réseau.

Les choix conceptuels qui ont été opérés pour la couche simulation permettent au concepteur de simulateurs de s'affranchir des mécanismes liés au couplage. En contrepartie, il doit développer une interface minimale de communication externe qui s'exprime en terme d'évènements. Si son modèle est implémenté en terme d'évènements discrets ou de temps discret, alors l'interfaçage ne pose pas de problème majeur. Par contre, si le temps n'est pas explicite dans son modèle, des choix doivent être faits en ce qui concerne le traitement des évènements en entrée ou en sortie qui eux, doivent être obligatoirement datés. Ce type de questions concerne plus particulièrement la couche modèle que nous allons maintenant décrire.

4.2.3 La couche modèle

Que signifie s'intéresser seulement au couplage au niveau modèle ? Comme nous l'avons dit en introduction de cette thèse, cette problématique est identifiée, notamment par Fishwick, sous le terme de modélisation multiple (*multi-modeling*) [Fis95]. La modélisation multiple considère qu'un modèle peut être composé de plusieurs autres modèles sous la forme d'un graphe où chaque nœud représente un modèle et les arcs les connexions entre ces modèles. Le couplage de modèles est alors vu comme la connexion de sorties à des entrées de modèles. La modélisation multiple

peut être vue comme l'agrégation de modèles. Fishwick parle également de raffinement lorsqu'un élément du comportement d'un modèle est exprimé plus précisément à l'aide d'un autre modèle. Cette opération définit *de facto* un nouveau niveau d'abstraction. La notion de raffinement intervient au niveau de la spécification de la dynamique. Nous pouvons par exemple définir un système d'équations différentielles pour expliquer les conditions de changements d'états d'un autre modèle (ce que nous avons fait dans le chapitre précédent avec le contrôle de l'état de satiété du copépode). Le système d'équations est alors un raffinement d'un état du modèle.
Cette approche du couplage de modèles est intéressante au niveau conceptuel mais ne répond pas à la question du couplage au niveau formel ou opérationnel. Le modélisateur est contraint de repenser ses simulateurs en fonction du multi-modèle construit.

Comme les autres couches, la couche modèle doit apporter un *medium* de communication entre les éléments de la même couche. Il est donc nécessaire de définir un bus modèle, où les formalismes et paradigmes utilisés peuvent être couplés. Il est possible d'adopter deux approches à ce niveau :

- un couplage à la fois formel et opérationel lorsque c'est possible,
- offrir un mode de représentation opérationnel des modèles pour permettre leur couplage.

Dans les deux cas, nous devons disposer d'un système d'écriture des modèles et de leur couplage.

Le premier cas correspond à la formalisation du couplage des modèles. Le concept de DEVS-bus est, là encore, utile. Il est possible de le considérer au niveau de la couche modèle. En effet, le choix de DEVS au niveau simulation est essentiellement lié au fait que DEVS est un formalisme indépendant des simulateurs qui offre également une sémantique opérationnelle. Ainsi, le lien entre la couche simulation et la couche modèle est établi de fait. Il faut pour cela formaliser le modèle en DEVS.

Comme nous l'avons dit dans le chapitre précédent, de nombreux auteurs proposent des spécifications DEVS de formalismes qui mènent parfois à des extensions de DEVS. Vangheluwe en fait état dans [Van00] et développe son approche $ATomM^3$ [LV02] de méta-modélisation et de couplage de différents formalismes[75]. Citons ici quelques exemples de spécifications DEVS de formalismes ou paradigmes connus (nous en avons déjà cité certains dans le chapitre précédent) :

- DEV&DESS [ZKP00] [BV02] pour une spécification des équations différentielles,
- Quantized Systems [Kof02], une méthode pour la résolution d'équations différentielles basée sur DEVS,
- G-DEVS pour une généralisation aux systèmes continus [GEG00],
- Cell-DEVS [WG01]) et DEVS et les automates cellulaires [VV00],
- DS-DEVS [Bar96],
- ... mais aussi des propositions concernant les state-charts [BV03], ...

Contrairement à la couche simulation, pour unifier conceptuellement les modèles, il est nécessaire de proposer les spécifications DEVS citées précédemment. Nous ne pouvons pas nous contenter d'une simple encapsulation du modèle par un *wrapper*. Le modélisateur doit faire l'effort de spécifier son modèle en DEVS soit en se référant à des travaux existants (si le formalisme a fait l'objet d'une spécification DEVS) soit en proposant sa propre spécification DEVS.

La deuxième approche, que nous pouvons adopter dans la couche modèle, est celle d'une

[75] http ://atom3.cs.mcgill.ca

description non formelle mais opérationnelle des modèles. En effet, certain modèles peuvent ne pas avoir d'équivalent en DEVS[76]. Si nous voulons communiquer ce type de modèles, nous devons mettre au point un vocabulaire commun pour leur description et une syntaxe pour leur représentation.

Ces réflexions, menées dans le cadre général du couplage de modèles hétérogènes, nous amènent à proposer le langage XML (eXtensible Markup Language) [HM01] pour la représentation des modèles de simulation des systèmes complexes. XML permet de définir une syntaxe de langage particulière à l'aide de balises du type de celles qui existent en HTML. La définition d'une telle syntaxe s'appelle une application XML. Dans ce qui suit, nous définissons une telle application appelée MLMC (Markup Language for Model Coupling). Un des grands avantages d'XML est sa flexibilité et son extensibilité, ce qui permet au format de fichier d'être étendu *ad infinitum*, tout en maintenant une compatibilité ascendante entre versions successives des formats de fichiers. De plus, un nombre croissant d'applications utilisent XML pour la sauvegarde des données, ce qui entraîne le développement rapide d'outils permettant le parcours des fichiers XML (on parle de *parser* XML). Ces *parsers* sont intégrés sous forme d'API (comme la « libxml » du langage C).
MLMC définit l'ensemble des modèles couplés et les connexions entre modèles. Il possède également des éléments de sémantique. Nous le décrivons donc après avoir présenté la dernière couche de notre *framework*.

4.2.4 La couche sémantique

Nous avons commencé à nous intéresser à la couche sémantique dans un but de couplage contrôlé des simulateurs. Plus précisément, nous voulons disposer d'une méthode et d'un outil qui rendent compte de la compatibilité des données échangées par les modèles au niveau du sens de ces données. Si un modèle calcule une vitesse, il est évident qu'il ne doit pas fournir ces données à un modèle qui a besoin d'une température en entrée. L'exemple paraît trivial mais il s'avère que l'automatisation d'un tel contrôle de cohérence au niveau sémantique est en réalité très complexe.
C'est dans le domaine des bases de données que l'on peut trouver les avancées les plus significatives en ce qui concerne l'intégration de données hétérogènes au niveau du sens [Jou01]. Cette discipline nous apprend que la fusion de deux ontologies (un vocabulaire et ses définitions) différentes est difficile. Néanmoins des outils existent, avec par exemple la mise au point de calcul de distance sémantique permettant de dire si deux termes sont proches ou différents [Jou01]. Il existe également des avancées dans le domaine de l'intégration d'applications hétérogènes qui peuvent servir de base à des outils d'intégration sémantique de modèles [XSDJ03].

Cette question nous a animés et a donné lieu à une publication qui propose XML pour la représentation de la sémantique des données pour le couplage de modèle [Dub02]. Néanmoins, nous pensons que cette question est trop vaste pour être traitée dans le cadre de cette thèse. Nous allons toutefois proposer dans l'application MLMC des éléments de sémantique et donc le support opérationnel pour relier données et sens des données.

De plus, la dernière couche de notre *framework* doit rendre compte de certains des choix du modélisateur pour la représentation des données. Dans l'état actuel de la réflexion, nous consi-

[76]Nous pensons ici à des SMAs très spécialisés ou même des plateformes de simulation existantes...

dérons quatre aspects de modélisation qui nous semblent les plus importants pour les systèmes dynamiques :

- la représentation du temps,
- la représentation de l'espace,
- la catégorisation des variables en terme d'unités,
- la classe d'appartenance des entités du système et le lien avec les variables du système.

Ce travail s'inscrit en partie dans un ensemble de réflexions menées au sein du groupe MIMOSA (Méthodes Informatiques pour la MOdélisation et la Simulation à base d'Agents[77]). Les membres du groupe MIMOSA réfléchissent sur les concepts manipulés en modélisation comme le temps, l'espace ou les entités d'un modèle. Ces réflexions permettent notamment d'adopter « un langage » commun pour la description des modèles. Ce « langage » peut être décrit par XML, ce qui rend opérationnel la comparaison, l'échange ou le couplage de modèles différents.

4.3 Description des modèles

Des travaux récents se sont orientés vers l'utilisation d'XML pour la représentation des modèles de simulation. Par exemple, F. Villa [Vil01] propose une syntaxe pour la description et l'échange des modèles. Seulement cette syntaxe s'applique principalement aux modèles formalisés avec des équations différentielles. P.A. Fishwick [Fis02] introduit MXL comme un langage basé sur XML pour la représentation des modèles. Néanmoins, ce langage reste pour l'instant limité et peu adapté à nos besoins de couplage. Les travaux les plus proches de nos préoccupations sont certainement ceux de l'équipe allemande du projet Man Model Measurement (M3) [Hoh02]. Ce projet a pour but de créer un monde virtuel dans lequel sont modélisées les interactions complexes entre l'homme et son environnement. M3 est un logiciel d'intégration ouvert dans le sens où n'importe quel modèle peut y être intégré du moment qu'il implémente les protocoles et interfaces définis pour les composants M3. Les concepteurs du projet ont décidé d'utiliser XML pour la représentation des données échangées par les différents modèles qui composent la plateforme. Néanmoins, les données et les entités réelles qu'elles représentent sont confondues dans la même application XML, ce qui oblige tous les modèles à « parler la même langue ». Nous avons préféré séparer les données des entités qu'elles représentent de manière à offrir plus de souplesse dans le couplage de modèles hétérogènes au niveau sémantique.
Pour la construction de la syntaxe XML, nous nous sommes encore une fois basés sur les travaux de Zeigler *et.al.* [ZKP00]. Nous reprenons ici les principales définitions apportées par Zeigler, comme les notions de ports d'entrées et ports de sorties auxquels sont attachées les données échangées entre modèles couplés. Nous introduisons d'autres types de ports, à savoir les ports d'états et d'initialisation. Les premiers permettent d'identifier les variables sur lesquelles le modèle permet d'effectuer des mesures. Les deuxièmes permettent d'initialiser certains paramètres comme les constantes du modèle. Zeigler définit également plusieurs niveaux hiérarchiques qui correspondent aux niveaux de connaissances que l'on possède d'un système. Dans notre approche, nous définissons un niveau hiérarchique minimal en dessous duquel nous n'avons pas connaissance du fonctionnement interne du modèle. À ce niveau, nous ne connaissons que la nature des données en entrée et en sortie et les dates associées (c'est le niveau 1 défini par Zeigler, voir figure 1.1 page 17). Le couplage des modèles décrits au niveau 1 s'effectue par la description des connexions entre les ports de sorties et les ports d'entrées des différents modèles. Ainsi, nous

[77] Site Web MIMOSA : `http://www-lil.univ-littoral.fr/Mimosa/`

décrivons un modèle couplé comme un « graphe » reliant des modèles décrits au niveau 1 ce qui engendre un modèle couplé au niveau 4.

Comme nous l'avons dit dans la section précédente, dans une perspective plus large que celle de la description du couplage qui est celle de la validation de ce couplage à différents niveaux (opérationnel et sémantique), nous avons besoin de connaître les choix de représentation du temps et de l'espace, les unités des données ou encore ce qu'elles représentent. Nous offrons donc une syntaxe de description des données ainsi qu'une description de l'espace et du temps à laquelle peuvent se référer les données.

4.3.1 Description de l'espace

L'espace peut être de trois types :
- un ensemble de lieux sans aucune relation de position des uns par rapport aux autres ;
- un espace topologique, où l'ensemble des lieux sont mis en relation par des liens de voisinage ;
- un espace métrique défini par un référentiel, une unité de mesure dans cet espace, sa caractéristique discrète ou continue, et dans le cas discret, le pas de discrétisation.

L'espace peut être décrit par un ensemble de places (*i.e.* de lieux), de relations de voisinage ou un référentiel. Il est décrit par la syntaxe XML ci-dessous (nous expliquons plus en détail cette syntaxe juste après l'avoir présentée) :

```
<SPACE name="space_name" type="topological | metric | set">

  <PLACES>

    <PLACE name="place_name1"/>

    <PLACE name="* | []" begin="n" end="m"/>

  </PLACES>

  <NEIGHBOURHOOD>

    <NEIGHBOUR link="place_name1,place_name2"></NEIGHBOUR>

    <NEIGHBOUR type="von_neumman | moore | none">
                    0 1 0 0 1 1 1 ...
    </NEIGHBOUR>

  </NEIGHBOURHOOD>

  <REFERENTIAL type="discrete | continuous" dimension="n">

    <AXIS id="x" min="val_min" max="val_max" step="val_step"/>

  </REFERENTIAL>

  <DISTANCE type="euclidian"/>

</SPACE>
```

Il est possible de définir l'ensemble des places (contenues dans la balise <PLACES>) en extension. Dans ce cas, il y aura autant de balises <PLACE> que de places dans le modèle. La première balise <PLACE> définit ce cas. Si les places sont définies en compréhension, alors il faut utiliser la syntaxe définie par la deuxième balise <PLACE> où l'attribut name="*" désigne un ensemble quelconque de places numérotées de 0 à n (n étant inconnu). Si name="[]" alors les attributs begin et end sont renseignés avec n et m désignant un ensemble de places numérotées de n à m, où n et m sont des entiers positifs, n < m et le nombre de places est égal à m-n.

L'attribut link de la balise <NEIGHBOUR> définit les relations de voisinage entre places. Si nous définissons ces relations en extension, nous utilisons la syntaxe donnée par la première balise <NEIGHBOUR>. Dans ce cas, il y aura autant de balises <RELATION> que de relations de voisinage. Si nous voulons définir ces relations en compréhension, l'ensemble des places doit impérativement avoir été défini en compréhension et nous devons utiliser la syntaxe de la deuxième balise <NEIGHBOUR>.
Le type de voisinage peut être connu ou inconnu. Dans ce dernier cas, la valeur de l'attribut type est égale à none. Dans ce cas, la suite de 0 et de 1 à l'intérieur de la balise <NEIGHBOUR> est le contenu de la matrice d'adjacence définie par l'ensemble des palces. La valeur 1 indique des voisins et 0 des non-voisins. La valeur de m-n est le nombre de lignes et de colonnes de cette matrice.

L'attribut type de la balise <REFERENTIAL> nous informe sur la nature discrète ou continue de l'espace. La valeur n de l'attribut dimension de cette balise est le nombre de dimensions de l'espace.

Un référentiel est défini à l'aide d'axes : c'est le rôle de la balise <AXIS>. Il y a autant de balises <AXIS> que de dimensions de l'espace. Si l'espace est discret, alors l'attribut val_step est renseigné. Sinon, nous définissons les bornes inférieures et supérieures val_min et val_max dans l'intervalle $]-\infty, +\infty[$. La balise <DISTANCE> a un rôle sémantique et nous renseigne sur la nature des distances mesurées (euclidiennes, d'ordre topologiques, etc.).

C'est le type de l'espace qui définit les balises contenues dans la balise <SPACE>. Si le type est « set », alors seule la balise <PLACES> et ses filles seront obligatoirement présentes. Si le type est « topological » alors les balises <PLACE>, <NEIGHBOURHOOD> et leurs filles seront obligatoirement présentes. Enfin, si le type est « metric », alors seule la balise <REFERENTIAL> et ses filles seront obligatoirement présentes. La balise <DISTANCE> est optionnelle. La balise <SPACE> peut apparaître plusieurs fois dans la définition d'un modèle, ce qui permet de représenter un modèle manipulant plusieurs types d'espace.

4.3.2 Description du temps

Le temps peut être de trois types :
- un ensemble de lieux temporels sans aucune relation d'ordre les uns par rapport aux autres ;
- un temps ordinal, où l'ensemble des instants sont mis en relation par une ou plusieurs relations d'ordre ;
- un temps cardinal défini par une base de temps, une unité de mesure dans ce temps, sa caractéristique discrète ou continue, et dans le cas discret, le pas de discrétisation.

Il y a une similitude assez évidente entre le temps et l'espace. En effet, pour les deux représentations il s'agit toujours de disposer d'un repère (ou référentiel). Le temps du modèle est

décrit par la syntaxe XML ci-dessous. Comme précédemment, nous expliquons plus en détails cette syntaxe juste après l'avoir présentée :

```
<TIME type="set | ordinal | cardinal">

    <TIME_SPANS>

       <TIME_SPAN name="time_spam_name"/>

       <TIME_SPAN name="* | []" begin="n" end="m"/>

    </TIME_SPANS>

    <ORDER>

       <RELATION sequence="name1,name2..."/>

       <RELATION sequence="Ti"/>

    </ORDER>

    <TIME_BASE type="discrete | continuous" unit="time_unit"
               begin="n" end="m" step="time_step"/>

</TIME>
```

La balise `<TIME_SPANS>` représente l'ensemble des lieux temporels utilisés dans le modèle (chez le voisin, le village, etc.). Ces lieux peuvent être définis en extension (première balise `<TIME_SPAN>`) ou en compréhension (deuxième balise `<TIME_SPAN>`), de la même façon que pour les lieux géographiques définis précédemment. La définition en compréhension implique une relation d'ordre de `n` à `m`.
Dans le cas d'une définition en extension, la balise `<ORDER>` contient les relations d'ordre (définies par la balise `<RELATION>`) entre les différents lieux temporels. Plusieurs syntaxes sont possibles pour la balise `<RELATION>`. Tout d'abord la définition en extension, où l'attribut séquence ordonne l'ensemble des lieux temporels, du plus récent au plus ancien (cas de la première balise `<RELATION>`). Il est également possible de définir les relations en compréhension (dernière balise `<RELATION>`).

La représentation du temps nécessite également de décrire le « type » de base de temps (discret ou continu). Dans le premier cas, l'attribut `step` de la balise `<TIME_BASE>` est renseigné. Les valeurs `n` et `m` sont des réels strictement positifs qui désignent la date de début et la date de fin du temps simulé par le modèle. Si la base de temps est continue (modèle à évènements discrets), alors nous ne connaissons pas à l'avance les dates d'occurrences des évènements ; nous ne spécifions donc pas de valeur pour l'attribut `step`.

Comme pour la définition de l'espace, le contenu de la balise `<TIME>` est conditionné par la valeur de son attribut `type`. Si le type de temps est `set`, alors seule la balise `<TIME_SPAN>` est présente, ainsi que ses filles. Si le type de temps est `ordinal`, alors les balises `<TIME_SPAN>` et `<ORDER>` sont présentes, ainsi que leurs filles. Si le type de temps est `cardinal`, alors seule la balise `<TIME_BASE>` est présente.
Une remarque importante s'impose : une donnée peut avoir une certaine représentation du temps

dans le modèle et être émise en sortie avec une autre représentation. Nous considérons que la représentation du temps attachée aux données correspond à leur représentation en sortie ou en entrée du modèle.

4.3.3 Description d'un modèle

La balise `<MODEL>` contient les deux balises `<TIME>` et `<SPACE>` plus d'autres balises qui spécifient les ports auxquels sont attachées les données. Chaque port est susceptible d'émettre ou de recevoir des évènements selon sa nature (ports d'entrée ou de sortie). À ces évènements sont attachés des variables structurées définies par la balise `<DATA>` (voir section 4.3.4). C'est la référence de la balise `<DATA>` au type de temps représenté par le modèle qui nous renseigne sur la fréquence ou les dates de sortie des évènements. Si le modèle ne manipule pas de temps (au sens courant), il émet une succession de résultats ordonnés. Cet ordre peut également être représenté dans la balise `<TIME>`.
Nous adoptons les notions de modularité et de hiérarchie des modèles définies dans [ZKP00]. Ainsi, un modèle peut être de type atomique (ou isolé) ou de type couplé. Ceci permet la construction d'un « graphe » de modèles, avec des modèles qui peuvent en contenir d'autres. Nous définissons d'abord la syntaxe pour les modèle atomiques, puis pour les modèles couplés.

4.3.3.1 Les modèles atomiques

La figure 4.7 montre les différents types de ports attachés à un modèle.

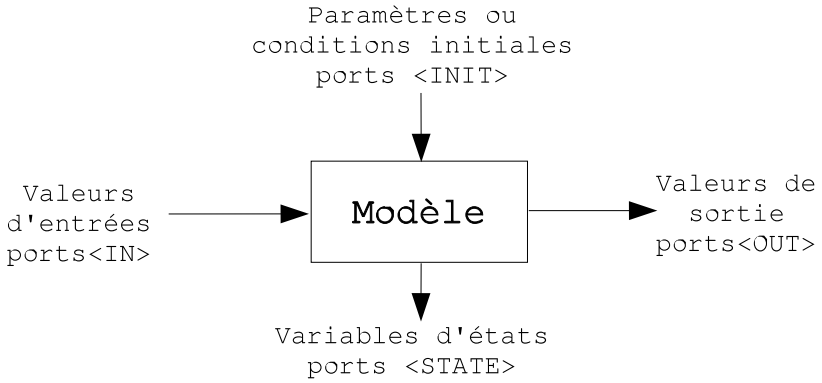

FIG. 4.7 – Représentation des différents types de ports attachés à un modèle. Par rapport à la définition des simulateurs abstraits DEVS (voir annexe D page 183), nous avons ajouté les ports d'états, qui permettent de définir les variables observables pour un modèle donné.

La syntaxe XML suivante définit la balise `<MODEL>` :

```
<MODEL name="model_name" type="atomic" autonomous="yes or no"
            xmlnx:xlink="http://www.w3.org/1999/xlink">

   <DESCRIPTION>

   </DESCRIPTION>
```

```
   <TIME>  </TIME>
   <SPACE> </SPACE>

   <INIT>

      <PORT name="port_name">  </PORT>

   </INIT>

   <IN>

      <PORT name="port_name">

   </IN>

   <OUT>

      <PORT name="port_name">  </PORT>

   </OUT>

   <STATE>

      <PORT name="port_name">  </PORT>

   </STATE>

</MODEL>
```

La balise `<MODEL>` contient les balises `<TIME>` et `<SPACE>` définies précédemment. De cette façon, nous unissons la description des modèles et leurs représentations du temps et de l'espace. L'attribut `autonomous` de la balise `<MODEL>` nous renseigne si le modèle peut être utilisé seul (*i.e.* sans être couplé à aucun autre). L'attribut `xmlns :xlink` définit un espace de nom. Ainsi tous les attributs définis peuvent prendre un attribut ayant comme préfixe `xlink` correspondant à la norme de définition des liens du *world wide web consortium* (w3c[78]).

La balise `<DESCRIPTION>` contient du texte libre. Cette balise permet de décrire le modèle par exemple.
Les balises `<INIT>`, `<IN>`, `<OUT>`, et `<STATE>` peuvent contenir plusieurs balises `<PORTS>`. Chaque port contient une ou plusieurs balises `<DATA>` de description des données qui lui sont attachées (nous décrivons la balise `<DATA>` un peu plus loin). Les ports d'initialisation ne peuvent pas être des ports d'entrée, et réciproquement.
La Balise `<IN>` permet de spécifier les données « réceptionnées » en entrée par le modèle. La balise `<OUT>` permet de spécifier les données « envoyées » par un modèle.

4.3.3.2 Les modèles couplés

Afin de spécifier le couplage de modèles, nous devons identifier les connexions externes (les ports d'entrée et de sortie du modèle couplé), ainsi que les différentes connexions internes qui

[78] URL du w3c : `http://www.w3c.org/`

relient les modèles composants. La figure 4.8 montre les connexions internes et externes sur un cas simple.

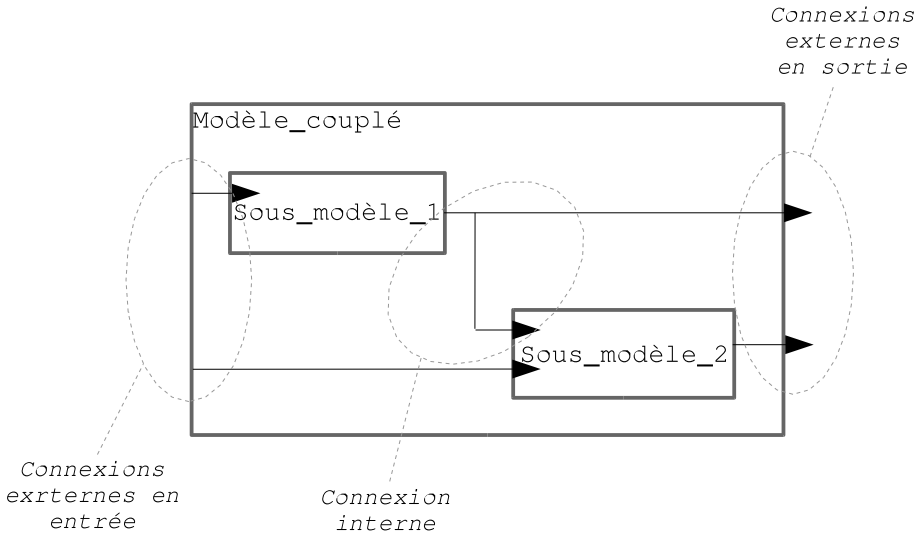

FIG. 4.8 – Exemple de modèle couplé composé de deux sous-modèles. Noter le fait qu'un port peut être connecté à plusieurs autres.

La syntaxe XML des modèles couplés diffère peu de celle des modèles atomiques. L'attribut `type` de la balise `<MODEL>` prend la valeur `coupled`, ce qui impose la définition des connexions internes et externes. Pour cela, nous introduisons de nouvelles balises définies dans la syntaxe XML ci-dessous.

```
<MODEL type="coupled" autonomous="yes | no"
       xmlns:xlink="http://www.w3.org/1999/xlink">

   <DESCRIPTION> </DESCRIPTION>

   <IN>          </IN>
   <OUT>         </OUT>
   <SPACE>       </SPACE>
   <TIME>        </TIME>

   <SUBMODELS>

      <LI xlink:type="simple"
          xlink:href="model_path#xpointer(XPath)"/>

   </SUBMODELS>

   <CONNECTIONS xlink:type="extended">

      <PORT_REF xlink:type="locator" xlink:label="port_label"
                xlink:href="model_path#xpointer(XPath)"/>
```

```
      <CONNECTION type="connection_type"
             xlink:type="arc"
             xlink:from="origin_port_label"
             xlink:to="destination_port_label"/>

   </CONNECTIONS>

</MODEL>
```

Nous voyons ici que la notion de décomposition hiérarchique est représentée par l'encapsulation des modèles dans la balise `<SUBMODELS>`. Les connexions externes du modèle couplé sont représentées par les mêmes balises que pour le modèle atomique.
Il y a autant de balise `<LI>` que de modèles composants. L'attribut `xlink :href` designe un lien qui pointe sur la racine d'un fichier de description d'un modèle. Ainsi, nous pouvons connaître la description des modèles composants.
La balise `<PORT_REF>` identifie un port qui entre en jeux dans la connexion des modèles. Le nombre de balises `<PORT_REF>` est égal à la somme des ports d'entrée et de sortie des modèles composants et du modèle composé.
La balise `<CONNECTION>` précise les connexions entre ports. Ainsi, le type des connexions peut être (voir figure 4.8) : `EIC` (connexion externes en entrée), `IOC` (connexion internes en sortie) ou `IC` (connexion interne). Chaque arc est orienté d'un port de sortie vers un port d'entrée.

Notons que dans cette syntaxe XML, les balises `<INIT>`, `<STATE>`, `<TIME>` et `<SPACE>` peuvent ne pas apparaître. En effet, il est possible de les déduire des modèles composants comme suit :

- l'ensemble des ports `<INIT>` correspond à l'union de tous les ports `<INIT>` des modèles composants,
- l'ensemble des ports `<STATE>` correspond à l'union de tous les ports `<STATE>` des modèles composants,
- les balises `<TIME>` et `<SPACE>` sont définies pour chaque modèle atomique.

Le modèle composé peut également spécifier ses propres ports différents des ports du modèle qui le compose. Les ports contenus dans les balises `<IN>` et `<OUT>` sont respectivement certains des ports `<IN>` et `<OUT>` des modèles composants. Il est nécessaire de les spécifier du fait qu'un même port peut être connecté à plusieurs autres.

4.3.4 Description des données

Nous décrivons ici l'ensemble des données en entrée ou en sortie des modèles. Trois niveaux de signification sont attachés aux données :

- le niveau informatique (entier, réel, booléen...),
- le niveau unitaire (l'unité de la donnée, une cardinalité ou sans unité...),
- le niveau sémantique (ce que représente la donnée : une distance, un nombre d'individus etc.).

La donnée représente plus que son unité et sa valeur aux yeux du modélisateur. C'est pourquoi nous ajoutons un niveau sémantique attaché à la donnée. Ce niveau est divisé en deux sous-niveaux :

- le sens propre de la donnée (une distance, une concentration, un objet...),
- sa représentation dans le temps et dans l'espace.

La représentation du temps ou de l'espace est très généralement indépendante de la donnée mais plutôt liée aux choix adoptés pour le modèle. La donnée fait référence à une certaine représentation. Par exemple, si la donnée est une matrice à deux dimensions représentant un espace topologique quadrillé, c'est la balise `<SPACE>` qui permettra de savoir que chaque élément de la matrice est situé sur une place particulière. Pour ce qui concerne le niveau du « sens », la donnée devra faire référence à une application XML particulière qui permet de représenter une « ontologie » du domaine scientifique dans lequel le modèle a été conçu. Une telle application reste à définir mais nous discuterons (brièvement) d'une première approche possible en ce qui concerne la modélisation en écologie.

La syntaxe XML pour la définition des données est la suivante :

```
<DATA name="data_name" xlink:type="extended">

        <TYPE class="data_type"/>

        <UNIT class="unit in the MKSA system"

        <TIME_REF xlink:type="locator" xlink:href="#xpointer(Xpath)"/>

        <SPACE_REF xlink:type="locator" xlink:href="#xpointer(XPath)"/>

        <METADATA xlink:type="locator"
                  xlink:href="metadata_path#xpointer(XPath)"/>

        <CONTENT dimension="matrix dimensions"
                 size="size of each dimension separated by ,">

        </CONTENT>

</DATA>
```

En ce qui concerne la balise `<UNIT>`, l'expression des unités se fait dans le système MKSA (Mètre, Kilogramme, Seconde, Ampère, etc.) pour les unités classiques. La donnée peut être également une simple cardinalité ou même sans unité.
La balise `<TIME_REF>` permet de faire référence à la représentation du temps définie précédemment et renseigne donc, par exemple, sur la fréquence de sortie ou d'entrée dans un modèle, d'une donnée particulière. La syntaxe de `path` est définie par le langage XPath[79]. Il en est de même pour la balise `<SPACE_REF>` qui concerne la représentation de l'espace.

Il est possible de décrire le contenu la donnée elle-même à l'intérieur de la balise `<CONTENT>`. Ce contenu sera toujours traité comme une chaîne de caractères. Des blancs séparent les contenus. Il est possible de traiter les matrices grâce aux attributs `dimension` et `size`.
Nous donnons également la possibilité de décrire des structures en incluant dans la balise `<DATA>` la balise `<AGREGATE>`. Cette inclusion récursive permet de décrire des données complexes comme l'ensemble des valeurs d'attributs d'un objet par exemple. Dans ce cas, l'argument `class` de le balise `<TYPE>` prend la valeur `agregate` et seules les balises `<UNIT>` et `<AGREGATE>` sont présentes comme filles directes de la balise `<DATA>`. Les balises `<UNIT>`, `<TIME_REF>` et `<SPACE_REF>` sont

[79] Voir par exemple [HM01] pour une présentation précise de ce langage.

toujours présentes au niveau des données agrégées, jamais au niveau de l'agrégat.

```
<DATA name="data_name" xlink:type="extended">

        <TYPE class="agregate"/>

        <AGREGATE>

           <DATA name="data_name" xlink:type="extended">
           </DATA>

        </AGREGATE>

</DATA>
```

La balise `<METADATA>` se réfère à une application XML qui reste à déterminer. Cette application doit permettre de dire à quelle entité s'applique la donnée et de quel point de vue. Néanmoins, une première piste existe. Elle consiste en la spécification, sous forme de diagramme de classes UML, des concepts de point de vue et d'entité dans un modèle, de représentation du temps ou de l'espace. Cette spécification est déjà bien avancée à l'intérieur du projet MIMOSA. À partir de cette dernière, il est possible de générer un fichier « .xmi » (XML meta-data interchange) avec des outils comme ARGOUML par exemple. Ce format XML permet de représenter les diagrammes UML. Ce fichier « .xmi » peut être transformé en un format XML propre à MIMOSA *via* une transformation XSLT[80]. Une fois que les concepts manipulés dans le domaine de la modélisation sont représentés dans une application XML, ils peuvent être mis en correspondance avec les entités de simulation manipulées dans des domaines particuliers. La figure 4.9, tirée d'un de nos articles [Dub02] et complétée, nous montre une association possible dans le domaine de la modélisation des écosystèmes.

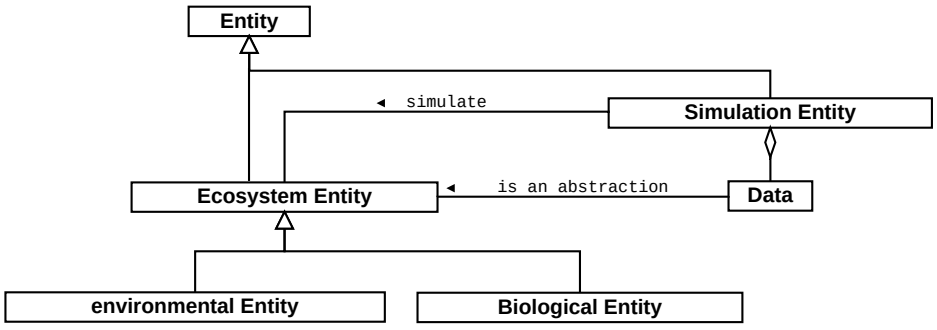

FIG. 4.9 – Diagramme de classes UML montrant les associations entre les entités de simulations et les entités simulées du point de vue des écosystèmes

Chaque donnée peut « pointer » sur une entité biologique ou environnementale, ce qui permet de la situer dans le contexte particulier du domaine de modélisation des écosystèmes. Dans la perspective de pouvoir valider, au moins partiellement, le couplage entre deux modèles au

[80]Se référer à l'ouvrage de référence sur XML [HM01] et le site de l'OMG sur le Model Driven Architecture (MDA) pour les définitions attachées à ces termes : `http://www.omg.org/mda/`

niveau sémantique, il apparaît nécessaire de définir une « ontologie » propre au domaine de modélisation. Ceci peut être fait en considérant le vocabulaire propre à une discipline. Il paraît difficile de construire un tel « dictionnaire » de manière exhaustive. Nous devrons donc mettre en place les outils qui permettront de la construire de façon empirique. Chaque modèle apportant des termes nouveaux construira l'arbre sémantique de sa discipline. Un effort particulier doit également être mené pour définir des termes communs entre disciplines pour permettre la validation du couplage de modèles issus de disciplines différentes. Ce travail reste à faire dans un cadre concerté.

Dans cette partie, nous avons défini une syntaxe XML particulière pour la définition des données, des modèles et des modèles couplés. Nous avons appelé cette application XML Meta Language for Model Coupling (MLMC). La DTD de ce langage est donnée en annexe A.1.

Jusqu'ici, nous avons élaboré un *framework* d'intégration de modèles hétérogènes en nous souciant des problèmes propres au couplage (opérationnel ou sémantique). Les modèles couplés que nous pouvons alors construire sont utilisés dans un cadre expérimental. Nous avons voulu définir une application XML qui permet la conservation, la modification et l'échange des expériences de simulation. Nous présentons cette application dans la section suivante.

4.4 Description des expériences

Comme nous l'avons dit précédemment, il y a un besoin important pour l'échange et la communication des modèles complexes. Nous sommes persuadés que ce besoin existe aussi pour les expériences. Traditionnellement, une expérience consiste à placer le système en cours d'étude dans des conditions expérimentales bien définies et à observer son évolution au cours du temps par l'intermédiaire de mesures. Deux approches peuvent être menées :

- on change les conditions expérimentales et on observe les variations de comportements du système ;
- on considère plusieurs systèmes placés dans les mêmes conditions expérimentales et on compare leur comportement.

La comparaison des comportements d'un ou plusieurs systèmes conduit à déployer de multiples stratégies statistiques. Ces stratégies correspondent aux études de sensibilité par exemple. L'avantage des expériences virtuelles (sur les modèles) par rapport aux expériences faites sur des systèmes réels est la maîtrise totale des conditions expérimentales et le fait que les mesures ne perturbent pas le système. Ainsi, nous pouvons construire des plans d'expériences rigoureux en vue de tests statistiques appropriés. Par analogie avec les expériences classiques menées en laboratoire ou sur le terrain, un plan d'expériences définit l'ensemble des conditions expérimentales, le nombre de réplicats (combien de fois on effectue la même simulation), éventuellement un blanc de simulation (une simulation de référence), une politique d'échantillonnage des résultats, etc. Pour ce qui concerne les expériences virtuelles, nous considérons que ce sont les valeurs des paramètres qui définissent les conditions expérimentales. Les paramètres sont les constantes du modèle ainsi que les valeurs d'initialisation des variables.

4.4.1 Laboratoire virtuel

Dans les sciences expérimentales classiques, il est nécessaire de décrire comment une expérience a été réalisée, le matériel utilisé et la démarche suivie. La description du protocole

expérimental prend une place importante dans les articles de journaux scientifiques sous le titre « *materials and methods* », le but évident de cette partie étant de permettre la reproductibilité de l'expérience. Dans cette partie, nous proposons une syntaxe XML (MLVE pour Markup Language for Virtual Experiments) pour la description des expériences virtuelles. Elle a pour but de décrire et stocker les plans d'expériences de simulations effectuées sur un modèle. L'adoption d'une syntaxe commune et portable permet l'échange et la validation des plans d'expériences sur plusieurs plateformes de modélisation ainsi que la reproductibilité des expériences. Indépendante de la plateforme d'exécution, cette syntaxe pourrait également permettre la mise au point d'une expérience standard sur un modèle standard pour la validation d'un outil de simulation par exemple. Dans la partie précédente, en définissant une syntaxe pour les modèles, nous avons introduit les termes de « port d'initialisation », « port d'entrée », « port de sortie » et « port d'état ». Ces définitions restent valables ici. Nous y ajoutons les notions de conditions expérimentales et de mesures.

Les conditions expérimentales fixent les valeurs à appliquer aux ports d'initialisation. Seulement une partie des ports peut être renseignée. L'autre partie sera renseignée par la valeur par défaut définie par le modèle. Si une valeur par défaut n'est pas définie, alors les conditions expérimentales doivent impérativement spécifier une valeur. Selon la politique d'exploration du modèle, les conditions expérimentales doivent être définies comme des ensembles de valeurs à parcourir pour toute ou partie des ports d'initialisation. La description des ensembles de valeurs peut être :

- simple : le paramètre prendra successivement les valeurs d'un ensemble de valeurs ou les valeurs comprises dans un intervalle selon un certain pas ;
- complexe : le paramètre prendra des valeurs qui sont fonction de la valeur prise par d'autres paramètres (on parlera de paramètres contraints).

Les modèles offrent la lecture des variables d'états par les ports d'états. La définition d'une expérience inclut la liste des points de mesures effectuées sur les variables d'états. La syntaxe indique tout simplement quelles sont les variables d'états que l'on désire observer. La lecture des mesures fait l'objet d'une politique d'échantillonnage. Cette politique définit le type d'acquisition des données. L'échantillonnage peut être de deux types :

- selon une certaine fréquence définie par le plan d'expériences,
- selon l'ordre d'arrivée des mesures défini par le modèle (évènements de sortie du modèle).

Nous parlons dans le premier cas de mesure active : le modèle est capable de générer les sorties en accord avec le plan d'expériences. Dans le deuxième cas, on parlera de mesure passive : chaque mesure est datée par la date d'occurrence de l'évènement de sortie de sa valeur. Pour ce qui concerne les mesures actives, on peut imaginer vouloir ne conserver qu'une valeur synthétisant un ensemble de valeurs en sortie d'un modèle. Jusqu'à maintenant, nous avons toujours supposé que le modèle n'effectuait pas lui-même de traitement sur les données. Cela implique de pouvoir appliquer des opérateurs statistiques (moyenne, par exemple) afin de synthétiser des indicateurs de comportement du modèle. Nous offrons la possibilité d'exprimer un tel type de mesure active, étant entendu que le traitement à la volée des mesures en sortie du modèle se fait par une application particulière.

Si le modèle ou le système intègre une partie stochastique (autrement dit, si deux simulations dans les mêmes conditions expérimentales conduisent à des réponses différentes), il est nécessaire d'effectuer des réplicats. Un ensemble de réplicats est un ensemble de simulations dans les même conditions expérimentales. Voyons maintenant l'utilisation d'XML pour la spécification des

expériences.

4.4.2 Spécification des expériences

Nous posons :
- qu'un modèle possède son simulateur ;
- qu'une simulation correspond à une exécution du simulateur ;
- que les mesures sont attachées à des ports d'états du modèle ;
- que l'initialisation se fait *via* les ports d'initialisation ;
- que les sorties d'un modèle se font dans un ou plusieurs fichiers.

Nous donnons dans ce qui suit l'ensemble des éléments de notre application XML. Ce document est bien formé (au sens d'XML), ce qui donne une idée de l'arborescence décrite par notre application. Nous décrirons ensuite de façon plus précise les balises qui définissent application.

```
<EXPERIMENT name="experience_plan_name" date="creation_date">

   <NOTES>

   </NOTES>

   <MODEL xmlns:xlink="http://www.w3.org/1999/xlink"
            xlink:type="extended">

      <DESCRITPION xlink:type="locator"
                   xlink:href="model_description_path#xpointer(/)"/>


      <EXECUTABLE  xlink:type="locator"
                   xlink:href="model_executable_path"/>


      <OUTPUT_STREAM xlink:type="locator"
                   xlink:href="output_model_path"/>


      <EXECUTION type="mono | distributed | parallel">
         <EXECUTION_NODE xlink:type="locator"
                         xlink:href ="node_adress"/>

      </EXECUTION>


      <EXPERIMENTAL_CONDITIONS replicat="number_of_replicat"
                               apply="experimental policy on parameter">



         <CONDITION  xmlns:pathtoport="http://www.w3.org/1999/xlink"
                     pathtoport:type="simple"

            <SET>
               <ITEM value="a number"/>
```

```
            </SET>
         </CONDITION>


         <CONDITION xmlns:pathtoport="http://www.w3.org/1999/xlink"
                    pathtoport:type="simple"
              pathtoport:href="model_description_path#xpointer(XPath)">

            <INTERVAL begin="begin_value"
                      end="end_value"
                      step="step_value"/>

            <RANDOM function="random_function" average="value"
                    standard_deviation="value" nget="value"/>


         </CONDITION>


         <CONDITION xmlns:pathtoport="http://www.w3.org/1999/xlink"
                    pathtoport:type="simple"
              pathtoport:href="model_description_path#xpointer(XPath)">
            <CONSTRAINST>
               <EQUAL_TO>
                  <MULT>
                     <CONST value="a numbre"/>
                     <VAR pathtoport:type="simple"
               pathtoport:href="model_description_path#xpointer(XPath)"/>
                  </MULT>
               </EQUAL_TO>
            </CONSTRAINST>
         </CONDITION>

      </EXPERIMENTAL_CONDITIONS>

      <MEASURES xmlns:pathtoport="http://www.w3.org/1999/xlink"
                pathtoport:type="extended">

         <MEASURE pathtoport:type="locator"
               pathtoport:href="model_description_path#xpointer(XPath)"
                  type="active | passive"
                  frequence="x" apply="average"/>

      </MEASURES>

   </MODEL>

</EXPERIMENT>
```

La balise <NOTES> permet d'inscrire du texte libre (des remarques générale sur l'expérience). <DESCRIPTION> est un pointeur vers un fichier de description d'un modèle, ce qui permet de naviguer vers les modèles impliqués dans le plan d'expériences. La balise <EXECUTABLE> pointe vers le fichier exécutable du modèle. <OUTPUT_STREAM> désigne le répertoire de sortie des don-

nées.
La balise `<EXECUTION_NODE>` permet spécifier la distribution les simulations sur un ensemble de nœuds d'exécution (autant de balises `<EXECUTION_NODE>` que de nœuds). Si le modèle est distribué (parallèle), alors nous spécifions l'adresse de base de la machine parallèle. Si le modèle s'exécute sur la machine où est traité le fichier XML, alors l'attribut `xlink :href` prend la valeur `this`.

La balise `<EXPERIMENTAL_CONDITIONS>` contient l'ensemble des valeurs possibles prises par les ports d'initialisation. Ainsi, l'ensemble des conditions s'appliquent sur les ports d'initialisation du modèle. L'attribut `replicat` nous indique le nombre de simulation que nous voulons faire avec le même jeu de paramètre en entrée (cas d'une simulation stochastique par exemple). L'attribut `apply` définit la politique appliquée pour définir le nombre de simulations faites avec des jeux de paramètres différents. Cet attribut peut avoir deux valeurs : `exhaustive` ou `sequential`. Dans le premier cas, le nombre de simulations sera égal au produit cartésien des ensembles définis sur chaque port d'initialisation. Dans le deuxième cas, on prend les valeurs de ces ensembles dans l'ordre. Tous les ensembles doivent donc avoir le même nombre de valeurs et il y aura autant de simulations que de valeurs dans ces ensembles.

L'ensemble des valeurs successives que prend un port est décrit par la balise `<CONDITION>`. Trois cas sont possibles :

- dans le premier cas, les valeurs sont prises dans un ensemble décrit par l'ensemble des balises `ITEM`,
- dans le deuxième cas, les valeurs sont prises dans un intervalle avec un certain pas (défini dans la balise `<INTERVAL>`). Il est possible de définir un intervalle à l'intérieur duquel nous tirons un nombre aléatoire. Pour cela nous utilisons la balise `<RANDOM>`. Les types possibles de fonctions utilisées sont donnés avec la définition du type de document XML en annexe A.2. Il est bien entendu possible d'étendre leur nombre. La balise `<RANDOM>` peut également aparaître dans une condition avec la balise `<SET>`,
- dans le troisième cas, les valeurs possibles sont contraintes par rapport à d'autres. Par exemple, dans la syntaxe présentée précédemment, les valeurs du port d'initialisation doivent toujours être deux fois supérieures à celles prises par un port de référence (troisième balise `<CONDITION>`). L'application qui traite le fichier d'expérience peut avoir deux politiques : soit elle génère elle-même les paramètres contraints, soit elle vérifie la validité de la contrainte.

La balise `<MEASURE>` contient des références sur l'ensemble des ports d'états du modèle affectés par une mesure. L'attribut `fréquence` est défini si l'attribut `type` a la valeur `active`. C'est un entier positif qui définit le nombre de sortie que l'on désire prendre en compte dans le calcul d'une valeur de synthèse de type moyenne par exemple, en affectant la valeur `average` à l'attribut `apply`.

L'application qui lit un tel fichier doit être capable de déterminer le nombre de simulations qui devront être réalisées en analysant l'ensemble des conditions attachées aux ports d'initialisation. Cette application est libre de mener la politique de son choix lorsqu'elle trouve par exemple dix valeurs possibles en entrée pour le port A et cinq pour le B. Faut-il faire dix simulations ou seulement cinq ? Nous offrons seulement la possibilité d'exprimer cette politique dans les cas simples décrit dans la syntaxe.
De manière générale, l'utilisation des contraintes conduit à la résolution de systèmes d'inéqua-

tions. Cette résolution peut s'avérer difficile. La partie expression d'une contrainte peut faire intervenir :

- tout opérateur arithmétique (+,−,×,/,%),
- toute fonction exponentielle, logarithmique et trigonométrique,
- toute fonction aléatoire (tirage aléatoire dans un ensemble discret ou continu de valeurs selon une certaine loi).

Pour ce qui concerne le nommage de fichiers de sortie, le choix est laissé à l'application qui lit le fichier de description des expériences. Ce fichier contient l'ensemble des simulations à effectuer. Pour savoir quel paramétrage de simulation particulière correspond à quels résultats particuliers, il faut établir une correspondance entre les noms des fichiers, l'ordre des simulation et les nœuds d'exécution par exemple. À partir du fichier d'expériences, on peut générer des fichiers de paramétrage correspondant à une simulation particulière, dans un format que le modèle sait lire (XML ou non).
Dans cette partie, nous avons défini une syntaxe XML particulière pour la définition des expériences. Nous avons appelé cette application XML Meta Language for Virtual Experiment (MLVE). La DTD de ce langage est donnée en annexe A.2.

4.5 Discussion et conclusion

Dans cette partie, nous avons proposé un *framework* pour l'intégration de modèles hétérogènes. Ce *framework* se compose de quatre couches (ou niveaux) : la couche opérationnelle, la couche simulation, la couche modèle et la couche sémantique. Ce cadre conceptuel nous a permis d'intégrer différentes propositions existantes dans notre propre architecture.

Pour la couche opérationnelle, nous proposons des connecteurs réseaux pour différents langages de programmation qui reposent sur des technologies standards allant des *sockets* jusqu'à SOAP. Ces connecteurs permettent à des coordinateurs DEVS de communiquer *via* le réseau (voir figure 4.6 page 103).

Pour la couche simulation, nous proposons l'intégration des simulateurs abstraits et du principe de DEVS-BUS pour disposer d'une base solide en matière de simulation distribuée. Ces algorithmes sont basés sur les notions d'évènements discrets et de fonctions de transition « classiques » de DEVS. DEVS nous permet de bénéficier des notions de couplage et de décomposition hiérarchique inhérentes à ce formalisme. Ainsi, nous pouvons composer des modèles par couplage au sens de DEVS. L'adoption de DEVS au niveau simulation nous oblige à entrer dans une certaine logique. Ainsi, nous considérons deux types de simulateurs :

- soit des simulateurs DEVS, auquel cas aucun problème de couplage n'est posé,
- soit des interfaces fonctionnelles, que nous avons appelées *wrappers*, qui permettent de rendre les simulateurs compatibles avec DEVS.

La deuxième proposition permet d'intégrer des simulateurs préexistants sans forcément se ramener dans une logique DEVS. Une difficulté apparaît avec le *wrapping* lorsque le temps n'est pas explicite dans les modèles que l'on veut coupler.
En effet, tous les formalismes n'intégrent pas le temps (les automates à états ou les réseaux de Petri, par exemple). Or, le temps et la base de temps utilisés sont fondamentaux pour le couplage de modèles dans notre *framework*. Dans le cas des réseaux de Petri, le comportement du réseau fait évoluer le marquage uniquement en fonction des transitions franchissables. Si une transition est franchissable, alors les jetons des places amonts sont retirés et des jetons sont injectés dans les

places avales. À aucun moment, la notion de temps n'a été évoquée. Pour résoudre le problème, nous pouvons considérer que le temps est discret et prend ses valeurs parmi les entiers positifs. À chaque transition ou lorsque l'on a épuisé l'ensemble des transitions franchissables à l'instant t, le temps avance. Cette approche n'a aucune conséquence sur le modèle lui-même, mais quelles sont les conséquences pour les modèles couplés ? Il faut en effet que la base de temps soit compatible. La réponse à cette question se situe aussi bien dans le modèle à base de réseaux de Petri que dans les modèles couplés. Dans certains cas, le réseau de Petri représente un processus non temporel ce qui implique que le choix d'une base de temps dépendra des modèles auxquels il est couplé. Si le réseau de Petri représente un processus temporel, il faut alors adopter une base de temps conforme au processus modélisé.
Une autre difficulté des *wrappers* concerne la définition des fonctions de transitions externes et de sorties pour des modèles pré-existants ? Là encore, la réponse est directement liée au modèle que l'on désire coupler. Le modélisateur doit fournir un effort minimal de « traduction » de la réception ou de l'envoi d'évènements dans son formalisme ou son paradigme.

En ce qui concerne la couche modèle, nous avons proposé une application XML (MLMC) qui permet de décrire le couplage de simulateurs. En effet, cette syntaxe décrit les interfaces entre modèles en terme de ports d'entrée-sortie interconnectés. De plus, cette syntaxe intègre des notions de sémantique attachée aux données qui transitent par ces ports. Par définition, XML permet d'encapsuler les données dans une hiérarchie qui peut représenter les différents niveaux sémantiques d'une donnée (de l'unité utilisée jusqu'à sa représentation pour le modélisateur). Une application qui permet de valider le couplage de modèle au niveau sémantique reste à définir mais nous pensons avoir posé des bases intéressantes dans ce sens. Dans un contexte scientifique où les ontologies sont assez précises, nous pensons que ce type d'applications pourrait être développé sans problèmes majeurs.
MLMC travaille au niveau 1 et 4 des niveaux de spécification donnés par Zeigler (voir tableau 1.1 page 17). Ceci implique que nous n'avons pas de représentation de la dynamique des modèles dans notre syntaxe. Là encore, nous pensons qu'il est possible d'arriver à une application XML augmentée de cette description. Comme nous l'avons dit, XML a une compatibilité ascendante ce qui nous assure de pouvoir étendre notre syntaxe. Nous avons commencé des travaux sur la définition XML des équations différentielles et des réseaux de Petri. Il existe également des spécifications XML en cours d'élaboration pour les modèles DEVS[81].

Devant la complexité des modèles qu'il est envisageable de construire avec une telle approche, nous pensons, en accord avec V. Grimm [Gri02], qu'il peut être utile d'adopter l'attitude du naturaliste devant son sujet. Ici, la notion d'expériences virtuelles prend tout son sens. Dans bien des cas, le modélisateur doit faire des plans d'expériences pour valider une hypothèse ou simplement faire une calibration. Notre proposition de spécification d'expériences virtuelles avec XML peut permettre de mieux appréhender la problématique des plans d'expériences bien connue des naturalistes. De plus, cette spécification peut aider à communiquer nos expériences. Il est évident que nous aurions pu établir nous-mêmes un format de fichier, mais ce choix ne permet pas de s'ouvrir aux outils qui se développent actuellement pour XML, et notamment ceux orientés vers internet. Dans ce cadre, nous avons pour objectif à court terme de mettre en place un laboratoire virtuel.
Par analogie avec un laboratoire réel, un laboratoire virtuel doit fournir l'ensemble des outils nécessaires pour la préparation, la réalisation et l'analyse des résultats d'expériences virtuelles.

[81] M.K. Traoré au LIMOS à Clermont-Ferrrand.

Une expérience virtuelle est la simulation d'un modèle informatique et l'observation de son comportement *via* les résultats (ou traces) de simulations. Ces traces peuvent être considérées comme un résultat émergent [SPTD98]. Nous pouvons fixer trois étapes dans la réalisation d'une expérience virtuelle :

- la préparation du système correspond à la définition de l'ensemble des valeurs de paramètres et conditions initiales affectées au modèle (ce sont les conditions expérimentales),
- la réalisation se fait *via* une ou plusieurs simulations sur un ou plusieurs nœuds d'exécution (simulation distribuée),
- l'analyse des résultats se fait *via* des outils de visualisation ou de traitement des données.

La logique et la problématique liées à l'expérimentation restent les même que pour une expérience classique : étudier le comportement d'un système sous certaines conditions et étudier la variabilité de ce comportement. L'étude de la validité du modèle par rapport au système réel fait partie de l'« analyse des résultats ». À ce niveau, il existe beaucoup de techniques statistiques (tests d'ajustements, analyse de variance, etc.) qui peuvent servir d'outils. Nous voulons proposer un service *web* qui assure non seulement la fonction de laboratoire virtuel, mais également de plateforme de couplage dynamique de modèles hétérogènes. Ce travail a commencé cette année dans le cadre de la thèse de G. Quesnel au Laboratoire d'Informatique du Littoral à Calais.

Notre démarche est assez proche de celle adoptée par l'OMG et connue sous le nom de MDA (pour *Model Driven Architecture*). En effet, nous voulons intégrer des logiciels (ici des modèles) pré-existants. C'est le problème bien connu en génie logiciel qui concerne les architectures légataires (*legacy architectures*), c'est-à-dire la maintenance, l'évolution, la réutilisation ou l'adaptation de l'existant. De plus, nous utilisons les mêmes technologies, comme UML et XML, pour les ontologies en cours de définition dans le groupe de travail MIMOSA du GdR I3. Ces technologies nous garantissent de bénéficier des dernières avancées en terme d'évolution du logiciel. Seulement, l'approche MDA, bien que basée sur la notion de modèle, s'adresse essentiellement au logiciel en tant qu'outil effectuant une tâche pour une entreprise ou une organisation, pas en tant que modèle de systèmes dynamiques avec lequel nous nous interrogons et nous expérimentons sur la réalité. Nous avons donc choisi une approche intégrative qui est relativement ouverte et centrée sur les modèles comme outils de connaissances et d'expériementations.

Maintenant que nous avons présenté une intégration formelle (chapitre 3) permettant de spécifier un modèle d'agents réactifs situés comme un modèle DEVS et un *framework* pour l'intégration de modèles hétérogènes, nous allons présenter l'application d'une intégration en écologie marine. Cette application illustre non seulement notre méthode de transfert d'échelles définie au chapitre 2.2, mais aussi le concept d'expériences virtuelles introduit dans ce même chapitre et complété ici d'une spécification opérationnelle.