Université du Littoral
Côte d'Opale

École doctorale de l'Université du Littoral - Côte d'Opale

Approche formelle et opérationnelle de la multi-modélisation et de la simulation des systèmes complexes

Apports pour la simulation de Systèmes Multi-Agents

Thèse

présentée et soutenue publiquement le 1er décembre 2006

pour l'obtention du grade de

Docteur
Spécialité informatique

par

Gauthier Quesnel

Composition du jury :

Président :	Philippe Mathieu	Professeur (LIFL - Lille)
Rapporteurs :	Claudia Frydman	Professeur (LSIS - Marseille)
	David Hill	Professeur (ISIMA – Clermont-Ferrand)
Examinateurs :	Edith Perrier	Directrice de Recherches (IRD - Bondy)
	Éric Ramat	Professeur - directeur de thèse (LIL - Calais)
	Jean Christophe Soulié	Maître de Conférences - encadrant (LIL - Calais)

Laboratoire d'Informatique du Littoral – UPRES EA-4029

L I L

Remerciements

Comment commencer ces remerciements par une autre personne que mon directeur de thèse, Éric Ramat. Son soutien de tous les jours depuis mon DEA et durant mes années de thèse m'ont permis de mener à bien ces travaux. L'impact qu'il a eu sur mes recherches et sur ma vie personnelle a largement dépassé le cadre de ses fonctions. Je tiens donc tout particulièrement à exprimer ma plus profonde gratitude à Éric pour avoir dirigé ces travaux et m'avoir soutenu dans cette étude. Ses conseils ont été indispensables à la concrétisation de cette recherche.

Je remercie également Jean Christophe Soulié pour m'avoir co-encadré. Sa disponibilité, ses critiques constructives et les nombreuses discussions que nous avons eu ensemble m'ont permis de profiter de son expérience, notamment dans le domaine des systèmes multi-agents et ainsi, de faire aboutir cette thèse.

Je tiens également à remercier Raphaël Duboz pour son aide précieuse tout au long de ma thèse et de la rédaction de ce manuscrit. Il m'a permis de m'orienter, de développer et de mettre au point mes idées et d'améliorer ainsi, la qualité de mon travail.

Je remercie également mes deux rapporteurs, Claudya Frydman et David Hill pour la lecture qu'ils ont fait de ce manuscrit et les remarques positives et critiques qui m'ont permis de prendre du recul sur mon travail. Je tiens aussi à remercier mes examinateurs, Edith Perrier et Philippe Mathieu, qui ont bien voulu examiner ce manuscrit.

Je tiens particulièrement à remercier l'ensemble des membres du laboratoire d'informatique du littoral pour l'ensemble des échanges que nous avons eu. Je remercie particulièrement Henri Basson, directeur du laboratoire, pour son accueil, Mourad Bouneffa pour son humour et sa culture, Dominique Verhaghe pour sa disponibilité et les échanges que nous avons eu.

Je remercie tous les chercheurs et personnels de l'université pour leur grande gentillesse et leur contribution à l'élaboration de ce travail. En écrivant ces remerciements, je pense évidement à mes compagnons d'infortune de la salle des thésards qui fut autant un lieu de vie qu'un lieu de travail durant ces quelques années passées au laboratoire.

Je termine ces remerciements par une pensée affectueuse à tous mes proches, mes parents, ma famille et mes amis. Je ne les citerai pas ici, bien qu'ils aient une part très

importante dans mon parcours pour leur soutien sans faille, leurs relectures assidues pour traquer les fautes. Ils ont été également d'un soutien indispensable dans des moments quelquefois difficiles.

Sommaire

Table des figures

Avant propos

Sommaire

Contexte de travail

Le travail présenté ci-après a été réalisé au sein du Laboratoire d'Informatique du Littoral à Calais (LIL) de l'Université du Littoral Côte d'Opale, et plus particulièrement dans l'équipe de Modélisation, Évolution des Systèmes Complexes (MESC). Cette équipe étudie les systèmes complexes selon deux points de vue. Le premier concerne les algorithmes et la programmation génétique dans une perspective d'apprentissage, d'optimisation mais aussi la construction de modèles automatiques. Le deuxième concerne la modélisation et la simulation des systèmes complexes naturels.

Cette thèse s'inscrit dans cette deuxième thématique de recherche initiée, en 1996, par Christophe Cambier et Philippe Preux avec la collaboration de la station marine de Wimereux. Ces travaux ont été ensuite repris par Éric Ramat dont les premiers résultats sont proposés dans [Ramat *et al.*, 1998]. Les problématiques soulevées par l'interaction entre les informaticiens et les écologues furent nombreuses. Par exemple, la définition d'un vocabulaire commun, l'utilisation de formalismes adéquats, la construction de plans d'expériences ou le paramétrage de modèles.

Des modélisations ont été effectuées à l'aide d'outils mathématiques et informatiques comme les systèmes multi-agents. L'ensemble des problématiques précédentes nous ont aidé à développer des compétences dans le couplage de systèmes hétérogènes avec la thèse de Raphaël Duboz [Duboz *et al.*, 2003] et dans la réalisation d'une plate-forme multi-agents [Ramat et Preux, 2003], basée sur le concept de laboratoire virtuel.

La modélisation et la simulation sont devenues des méthodes très répandues dans les différentes disciplines scientifiques pour comprendre, analyser et essayer de prédire le

comportement des systèmes complexes. Les travaux sur la modélisation et la simulation trouvent leur place dans trois groupes de recherches auxquels nous participons : Modélisation Multiple et Simulation (MMS), VERs une théorie de la SIMulation (VERSIM) et Méthodes Informatiques de la MOdélisation des Systèmes à base d'Agents (MIMOSA) du GDR I3.

Les problématiques soulevées entre nos coopérations avec les écologues et les biologistes forment la base de nos travaux de recherche : la modélisation à base de formalismes hétérogènes avec les questions de couplages soulevées par Raphaël Duboz, la construction de plans d'expériences, le paramétrage automatique des modèles, la distribution automatique des simulations des grandes tailles, la robustesse des modèles et l'étude d'impact des paramètres ou encore l'introduction d'éléments sémantiques dans les descriptions des modèles. Dans cette thèse, nous voulons montrer que la recherche en informatique peut apporter des réponses et des outils opérationnels aux problématiques de la modélisation et de la simulation des systèmes complexes.

Structure du document

Introduction générale

Nous commençons notre exposé par une introduction générale sur un ensemble de définitions de termes et de concepts employés dans ce manuscrit. Nous présentons également le formalisme DEVS [Zeigler, 1976] qui est une spécification pour le développement et le couplage de modèles hétérogènes. De cette spécification, un ensemble d'extensions ont vu le jour pour prendre en compte des paradigmes, comme Cell-DEVS [Wainer et Giambiasi, 2001] pour les automates cellulaires. Les principales extensions sont également présentées dans ce chapitre.

Fonctionnement interne du simulateur

Le deuxième chapitre expose la formalisation d'une nouvelle spécification d'un simulateur basé sur DEVS. Cette spécification s'appuie sur une amélioration possible du formalisme DEVS que nous avons identifié. Dans ce chapitre, nous proposons la formalisation de ce simulateur et de plusieurs extensions de DEVS. Nous proposons également un ensemble de perfectionnements du formalisme pour permettre une meilleure intégration des outils de simulation.

Formalisation du paradigme agent

Le troisième chapitre introduit le paradigme agent dans la théorie de la modélisation et de la simulation [Zeigler *et al.*, 2000]. Nous proposons, dans ce chapitre, une forma-

lisation des différents aspects des systèmes multi-agents [Ferber, 1995]. Cette nouvelle spécification est basée sur une extension DEVS nommée DS-DEVS [Barros, 1996], sur nos travaux de perfectionnement du formalisme DEVS et sur une définition des agents spécifiques [Soulié, 2001].

VLE, architecture logicielle

Les deuxième et troisième chapitres présentent, de manière formelle, nos travaux réalisés au cours de cette thèse. Ce quatrième chapitre expose le développement d'une nouvelle plate-forme logicielle, VLE, dans laquelle tous les concepts des précédents chapitres sont proposés et décrits de manière opérationnelle. Cette plate-forme, placée sous une licence de logiciel libre, repose sur les concepts de composants et de modularité afin de proposer aux contributeurs un ensemble de méthodes différentes pour améliorer la plate-forme.

Exemples de modélisation sous VLE

Ce cinquième chapitre illustre l'utilisation de notre plate-forme logicielle VLE et des spécifications présentées dans le deuxième chapitre à travers deux exemples traditionnels, le « Jeu de la vie » de J. Conway [Gardner, 1970] et un modèle de ségrégation de T. Schelling, [Schelling, 1971]. Un troisième exemple est proposé pour expliquer la modélisation, à l'aide de la spécification de systèmes multi-agents décrits dans le troisième chapitre, dans un modèle de brigade de pompiers en lutte contre des incendies de forêt.

Remarques générales avant lecture

Cette thèse repose sur deux apports majeurs, la définition et la spécification d'un simulateur et l'intégration d'outils formels pour la description de systèmes complexes.

Ce document propose, dans la mesure du possible, une approche graduelle dans la description des différents concepts et apports des cinq chapitres composant ce manuscrit afin d'aider à la compréhension des différents mécanismes mis en jeu.

Nous vous souhaitons une bonne lecture.

Chapitre 1

Introduction

Sommaire

L'objectif de ce premier chapitre est de poser les définitions des concepts manipulés en modélisation des systèmes complexes. Nous développons, ici, la définition que nous avons adoptée pour les concepts de système, de système complexe, de système spatio-temporel, de paradigme, de formalisme et de multi-formalismes. Ce panorama nous conduit à évoquer les problèmes sous-jacents de la modélisation et, en particulier, de la multi-modélisation. Nous montrerons les réponses apportées par la littérature au problème de couplage de modèles tant au niveau formel qu'au niveau opérationnel. Nous introduirons les approches de spécifications formelles des systèmes dynamiques de B. P. Zeigler. Cette introduction nous permettra d'évoquer la modélisation multi-agents dans ce cadre formel et nous montrerons comment certaines extensions de DEVS, *Discrete EVent system Specification*, peuvent apporter un complément de réponse à la formalisation des systèmes multi-agents.

1.1 Définitions

Dans cette partie, nous abordons et définissons les différents concepts employés tout au long de ce manuscrit : système, système complexe, système spatio-temporel, modèle, niveau de spécification formelle des modèles, paradigme et formalisme. Ces définitions sont l'occasion de préciser notre point de vue sur les concepts de l'activité de modélisation et de simulation.

Qu'est-ce qu'un système ?

La notion de système est à associer aux objectifs de l'étude que l'on mène. En effet, étudier un système, c'est se poser des questions sur son fonctionnement. Le fait même de se poser des questions définit ce qu'est le système. Un système est donc l'ensemble des éléments concernés par l'interrogation. Dès lors que nous nous intéressons à chaque chose pour en comprendre le fonctionnement, ou la dynamique, nous devons être capables d'identifier les éléments qui constituent l'objet d'étude. Cette définition, [Fishwick, 1995], intègre l'idée que nous devons être capables de nous limiter aux éléments essentiels et de prendre en compte seulement les éléments susceptibles de répondre à la question posée. Cette opération n'est pas toujours simple et le chercheur doit souvent procéder par tâtonnements.

Qu'est-ce qu'un système complexe ?

La définition précédente suggère qu'un système est un ensemble d'éléments ou d'entités. Celles-ci possèdent un comportement ou une dynamique propre qui, par composition et interactions entre elles, forment la dynamique du système. Si nous étendons cette définition, nous pouvons définir un système complexe comme un ensemble

composé d'un grand nombre d'entités en interaction où le comportement global du système complexe émerge de l'interaction des entités composant le système. En systémique [Bertalanffy, 1968], une des caractéristiques des systèmes complexes est d'être hiérarchisée. Chaque système est alors composé de sous-systèmes interconnectés et est élément du super-système. Ce point de vue forme un courant fort chez les modélisateurs. Pour comprendre cet état de fait, nous étudions dans les paragraphes suivants la philosophie des Sciences.

L'histoire des Sciences a vu naître plusieurs courants de pensées : le *globalisme*, le *totalisme*, l'*holisme*, le *réductionnisme*, etc. [Le Moigne, 1977]. Ces courants ont conduit à une perception des systèmes différente. Par exemple, le globalisme et le totalisme ne croient pas nécessaire une vision d'un Monde décomposable en éléments plus petits et surtout ne croient pas en l'idée que le comportement global émerge des interactions des parties. Le globaliste pense qu'il n'y a pas de limite entre les choses et donc, une division du système en éléments n'est pas nécessaire. Un exemple simple permet d'illustrer ce précepte. Si une personne n'ayant jamais vu une calculatrice en trouve une, pour comprendre ce que c'est, est-il nécessaire qu'elle ouvre l'objet pour découvrir qu'elle est composée d'un circuit électronique lui-même composé d'éléments plus petits ? Le globaliste affirme que non. Pour ce courant de pensées, ce qui est important c'est la fonction globale de l'objet. La connaissance des parties n'est pas nécessaire voire néfaste à la compréhension. Le holisme [Smuts, 1926], autre courant de pensées, postule que le *Tout* précède ou transcende ses parties avec l'expression consacrée :

> « *le Tout est plus que la somme de ses parties* »

Cette vision du Monde implique que le Tout n'est pas seulement la réunion d'éléments, mais qu'il participe au comportement du Tout. La question que nous pouvons nous poser est quelle est cette chose qui donne corps au Tout ? C'est la question que se pose un réductionniste. Pour lui, toute réalité se réduit en fin de compte à des constituants élémentaires. En biologie, on dira que le vivant se réduit à des molécules et à leurs interactions. Le monde est vu comme une hiérarchie de niveaux ordonnés suivant des échelles de complexité, d'espace et de temps, qui s'emboîtent. Le réductionnisme ontologique est compatible avec l'idée que les systèmes complexes puissent posséder des propriétés spécifiques qui n'existeraient pas à un niveau inférieur. Du point de vue de la connaissance, le réductionnisme épistémologique affirme que la connaissance d'un phénomène ne peut se faire qu'en réduisant ses multiples descriptions à un nombre de plus en plus restreint de principes, lois, théories ou concepts. Le scientifique élimine ce qui ne paraît pas essentiel à sa compréhension, fait un choix de paramètres ou de variables afin de déterminer la structure du système qu'il étudie. Il est difficile de procéder autrement sous peine de se voir dépasser par la quantité d'informations. Par l'introduction de la notion d'interactions entre les parties, les réductionnistes prétendent assimiler le précepte des holistes : « le Tout est plus que la somme des parties ». En effet, les interactions auraient la propriété de faire émerger des propriétés au niveau du Tout qui ne sont pas décrites en tant que telles au niveau des parties. Cette idée est

la pierre angulaire des approches multi-agents réactifs.

Afin de conclure sur la notion de systèmes, de systèmes complexes et d'approches de modélisation du Monde, J. L. Le Moigne dans [Le Moigne, 1977] propose le nouveau discours de la Méthode et érige quatre préceptes :

– la pertinence :

> « *[...] convenir que tout objet que nous considérerons se définit par rapport aux intentions implicites ou explicites du modélisateur. Ne jamais s'interdire de mettre en doute cette définition si, nos intentions se modifiant, la perception que nous avions de cet objet se modifie [...]* »

– le globalisme :

> « *[...] considérer toujours l'objet à connaître par notre intelligence comme une partie immergée et active au sein d'un plus grand Tout. Le percevoir d'abord globalement, dans sa relation fonctionnelle avec son environnement sans se soucier outre mesure d'établir une image fidèle de sa structure interne, dont l'existence et l'unicité ne seront jamais tenues pour acquises [...]* »

– la télélogique :

> « *[...] interpréter l'objet non pas en lui-même, mais par son comportement, sans chercher à expliquer a priori ce comportement par quelque loi impliquée dans une éventuelle structure. Comprendre en revanche ce comportement et les ressources qu'il mobilise par rapport aux projets que, librement, le modélisateur attribue à l'objet [...]"* »

– l'agrégativité :

> « *[...] convenir que toute représentation est partisane, non pas par oubli du modélisateur, mais délibérément. Chercher en conséquence quelques recettes susceptibles de guider la sélection d'agrégats tenus pour pertinents et exclure l'illusoire objectivité d'un recensement exhaustif des éléments à considérer [...]* »

Ces quatre préceptes sont très importants pour la modélisation de systèmes complexes car elles peuvent aider et guider la modélisation. Nous ajoutons également le précepte dicté par les réductionnistes où toute réalité se réduit à des constituants élémentaires. Ce dernier précepte ouvre la porte à l'approche par décomposition qui aujourd'hui est l'approche dominante.

Qu'est-ce qu'un système dynamique et un système spatio-temporel ?

Les travaux que nous menons, au sein de notre équipe MESC, s'inscrivent principalement dans le cadre de systèmes spatio-temporels. La définition générale d'un système ne fait pas apparaître explicitement la notion de temps et encore moins la notion d'espace. La théorie des systèmes distingue deux informations : la structure interne du système en terme d'éléments qui le composent et le comportement. Le comportement

peut être vu comme le résultat de la dynamique interne du système. Le système peut alors être représenté comme une boîte noire composée uniquement d'entrées et de sorties. Dans ce cadre, nous ne nous intéressons pas à la compréhension de la dynamique interne, mais uniquement aux observations et ainsi aux manifestations externes. Si le temps n'est pas exprimé, alors les sorties sont mises en correspondance avec les entrées. Par exemple, en plaçant X en entrée, nous obtenons Y en sortie. Nous disposons alors d'une table de correspondance comme dans le cas d'un circuit logique ou d'une fonction du type $y = f(x)$. Nous reviendrons sur ce point de vue dans la hiérarchisation des niveaux de spécifications des systèmes.

Un système dynamique est un système où la dynamique est définie par rapport à une base de temps, un état courant et une fonction d'évolution des états en fonction des entrées du système. Dans ce cas, le temps est une composante forte des systèmes dynamiques. Les changements d'états du système s'inscrivent par rapport au temps. Nous pouvons identifier sur un axe temporel l'évolution de l'état du système. La fonction d'évolution est appelée *fonction de transition*.

Un système spatio-temporel est, par définition, un système dynamique dans lequel les éléments et les processus sont définis relativement à un espace. L'espace est ici entendu comme l'espace dans lequel des entités peuvent évoluer. Les entités possèdent alors des coordonnées relatives à un repère absolu. De même, les processus sont décrits dans l'espace des entités. Les écosystèmes sont de parfaits exemples de systèmes spatio-temporels puisqu'ils désignent l'ensemble formé par une association d'êtres vivants et ses environnements géologique et atmosphérique.

Qu'est-ce qu'un modèle ?

L'étude d'un système implique son observation [Fishwick, 1995] or, observer un système, c'est avant tout construire un modèle du système observé [Pavé, 1994]. Les outils et les connaissances utilisés pour l'observation impliquent un filtrage de la réalité par un modèle [Ferber, 1995]. Par exemple, lorsque nous regardons une chaise, notre œil en capte une image que nous considérons comme la réalité. Cette réalité est une construction physico-chimique d'une image mentale d'un objet réel. Une caméra infra-rouge et un insecte auront sûrement une autre vision de la même chaise.

Un modèle est donc un filtre conditionné par nos connaissances, nos vérités et nos capteurs. En restreignant le discours à l'activité de modélisation, le modèle est le résultat de l'activité de modélisation. Cette dernière implique un schéma expérimental, un ou des paradigmes et une méthodologie. La notion de paradigme se substitue ici à la notion de vérité dans le cadre général. La figure 1.1 représente l'activité de modélisation.

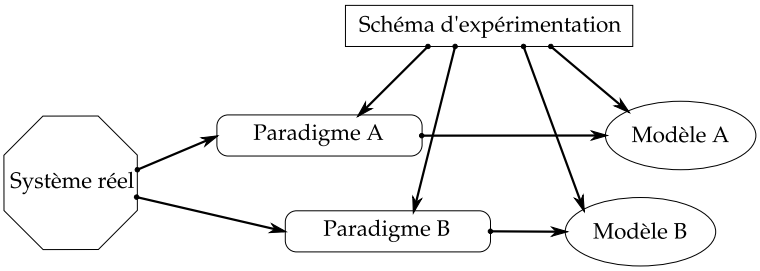


FIG. 1.1 – L'activité de modélisation.

Les niveaux de spécifications formelles des modèles

Avant d'aborder les notions de paradigme et de formalisme, attardons-nous sur la proposition de B. P. Zeigler dans [Zeigler, 1976], où un système peut être spécifié formellement et cette spécification dépend d'un certain niveau de description. Cette approche, purement formelle, permet de décrire un système sans aucune hypothèse liée à un paradigme ou à un formalisme.

Le premier niveau de description d'un système diffère peu de la définition de boîtes noires telle que nous l'avons présentée précédemment. Les entrées du système sont liées à un intervalle de temps que l'on nomme segment, noté ω. Nous définissons la notion de segment de sortie ρ, ou de trajectoire de sortie, qui met en relation un intervalle de temps et la sortie observée. B. P. Zeigler nomme ce type de spécifications des schémas d'observations. Aucune relation n'est établie entre les entrées et les sorties. Nous sommes ici en présence d'une simple observation d'un phénomène, sans aucune interprétation de celui-ci.

Si nous établissons une relation entre les segments d'entrée et les segments de sortie, alors nous définissons le deuxième niveau de spécification formelle d'un système. Ce niveau, est appelé observation de la relation d'entrée-sortie. La relation a pour but de construire un ensemble de couples (ω, ρ), sachant que ω et ρ sont observés sur le même intervalle de temps. L'inconvénient majeur de cette spécification est d'être non-déterministe. En effet, pour un même segment d'entrée ω, il peut exister plusieurs segments de sortie ρ. Nous ne sommes donc pas capables de prévoir la sortie par simple observation de l'entrée du système.

Cependant, il manque effectivement une notion essentielle pour les systèmes dynamiques dans les deux premiers niveaux de spécification : la notion d'état. Il suffit, en effet, d'ajouter la spécification de l'état initial du système pour rendre déterministe la relation d'entrée-sortie. On parle alors d'observation de la fonction d'entrée-sortie.

Le quatrième niveau d'abstraction s'attache à définir la spécification formelle d'un système dynamique. À ce niveau, nous cherchons à spécifier le comportement interne du système. Pour cela, nous introduisons la notion d'état interne et de fonctions de transition.

$$S = \langle T, X, Y, \Omega, Q, \Delta, \Lambda \rangle$$

où Q est l'ensemble des états du système, Δ la fonction de transition ($\Delta : Q \times \Omega \rightarrow Q$) et Λ la fonction de sortie ($\Lambda : Q \rightarrow Y$) où, Y est l'ensemble des valeurs de sortie. L'ensemble Ω est l'ensemble des segments d'entrée admissibles tandis que X est l'ensemble des valeurs d'entrée possibles. La différence réside dans le fait que Ω définit des couples $(\langle t_1, t_2 \rangle, x)$ où $x \in X$.

Cette formalisation des systèmes dynamiques permet de décrire le comportement du système en fonction de son état interne et des événements d'entrée. Or, dans cette vision, nous considérons que le système n'est pas autonome et qu'il n'a donc pas de dynamique propre puisque seuls des événements externes peuvent modifier l'état interne du système. Nous verrons par la suite que B. P. Zeigler propose un cadre formel de spécification de systèmes dynamiques généralisant ce niveau d'abstraction en ajoutant une fonction de transition interne. Cette dernière a pour objectif de modéliser la dynamique interne.

Le cinquième et dernier niveau introduit le concept de couplage de modèles. Le modèle global est alors un ensemble de modèles interconnectés. Ces connexions déterminent les relations entre les modèles et les événements échangés entre les modèles. À ce niveau, on considère que le modélisateur connaît la structure interne de son système et qu'il est donc capable de le décomposer en sous-modèles.

Cette hiérarchisation des niveaux de spécifications formelles des systèmes sera par la suite notre référence dans la plupart de nos choix. Il est intéressant de retenir les idées d'interaction entre modèles à l'aide d'événements, de fonctions de transitions, de fonctions de sortie et de couplage de modèles.

Qu'est-ce qu'un paradigme ?

Lorsque B. P. Zeigler propose ses niveaux de spécifications, il exploite le paradigme de la modélisation à événements discrets. Un paradigme est un cadre de pensée composé par un ensemble d'hypothèses fondamentales, de lois et de moyens sur la base desquels les modèles peuvent se développer [Kuhn, 1972].

En règle générale, le paradigme est utilisé au niveau sémantique tandis que le formalisme est l'outil opérationnel du paradigme. À un paradigme est associé un ou plusieurs formalismes. De plus, nous pouvons concevoir des modèles en s'appuyant sur les concepts d'un paradigme et spécifier le modèle à l'aide d'un formalisme qui n'est pas obligatoirement issu du paradigme. Si nous prenons l'exemple du paradigme des systèmes multi-agents, la spécification de la structure du système peut être réalisée à l'aide d'UML (*Unified Modeling Language* [Jacobson et Booch, 1999]), tandis que la dynamique des agents peut être spécifiée à l'aide de réseaux de Petri ou de règles d'inférences.

Qu'est-ce qu'un formalisme ?

Les définitions précédentes ont posé les bases de la modélisation en définissant ce qu'est un modèle. Nous avons vu également qu'il était possible de formaliser un système selon différents niveaux d'abstraction, niveaux qui dépendent la plupart du temps du niveau de connaissances sur le système.

Revenons, par exemple, sur la notion de fonction de transition. Cette fonction a pour objectif de déterminer le nouvel état du système en fonction des événements d'entrée. La question qui reste en suspens est comment spécifier cette fonction. Le formalisme est l'outil dédié à cette opération. C'est l'outil d'expression des paradigmes. On peut utiliser, par exemple, les automates à états finis pour spécifier les changements d'états, mais ce n'est pas la seule possibilité.

Qu'est ce que le multi-formalisme ?

Modéliser un système, c'est construire un objet abstrait représentant la structure et la dynamique du système observé. Pour un même système et pour des modélisateurs différents, l'activité de modélisation va donner naissance obligatoirement à des modèles différents. Comment peut-on qualifier cette différence ?

Le premier cas de figure est le plus simple : les deux modélisateurs ont le même objectif, ils veulent comprendre la même chose et ils en ont le même point de vue. La seule différence réside dans les outils utilisés. Le choix d'outils différents est très souvent gouverné soit par une méconnaissance des outils existants, soit par la capacité de l'outil choisi pour exprimer la structure ou la dynamique. Dans ce cas, on cherche à vérifier que les deux modélisations sont équivalentes et on parle alors de *mapping*. Le *mapping* cherche à construire une bijection entre les éléments des deux modèles.

Le deuxième cas de figure met en jeu une démarche de hiérarchisation des modèles. Le système est décomposable en sous-systèmes et chaque sous-système fait l'objet d'une modélisation. La décomposition peut aussi s'appliquer sur les éléments de la dynamique du système. Dans ce cas, on peut, par exemple, décomposer un état du système en sous-états. Les formalismes et les paradigmes utilisés pour le système global et les sous-systèmes peuvent être différents. Ce point de vue est développé par P. A. Fishwick dans [Fishwick, 1995, Fishwick, 1993], où il parle alors de raffinement ou d'abstraction dans un cadre de multi-formalisme. Le prérequis à un tel type de multi-formalisme est que les paradigmes et les formalismes utilisés soient compatibles.

Dans l'exemple suivant, représenté sur la figure 1.2, extrait de [Fishwick, 1993], le système modélisé est une casserole avec de l'eau en train de chauffer, posée sur un système de chauffage. Pour le premier niveau de description de la dynamique du système, on peut identifier deux super-états : chaud et froid. Le passage d'un état à un autre est gouverné par l'activation du système de chauffage et la température de l'eau. Le for-

malisme utilisé pour ce niveau peut être les automates à états finis. L'auteur décompose ensuite ces super-états en sous-états afin de décrire plus précisément les différents états de l'état « chaud ». Le formalisme adopté ne change pas. Nous restons donc dans le cas d'une décomposition hiérarchique. En revanche, l'un des états de l'état « chaud » est à son tour décomposé. Cet état représente l'état « en cours de montée en température » et les conditions de transition sont fonction de la variable température. Il devient alors intéressant d'utiliser un autre formalisme, ici les équations différentielles, pour représenter le processus de chauffage et les conséquences sur la variable température. Quelles sont les implications d'une telle décomposition ? Il faut que le formalisme du sous-état prenne en compte les contraintes du formalisme utilisé au niveau d'abstraction supérieur. Dans l'exemple développé ici, les transitions entrantes sur l'état « en cours de montée en température » doivent être traduites dans le formalisme des équations différentielles. Cette traduction détermine les conditions initiales de l'équation différentielle. Réciproquement, les transitions sortantes de l'état sont fonction de la variable décrite par l'équation différentielle.

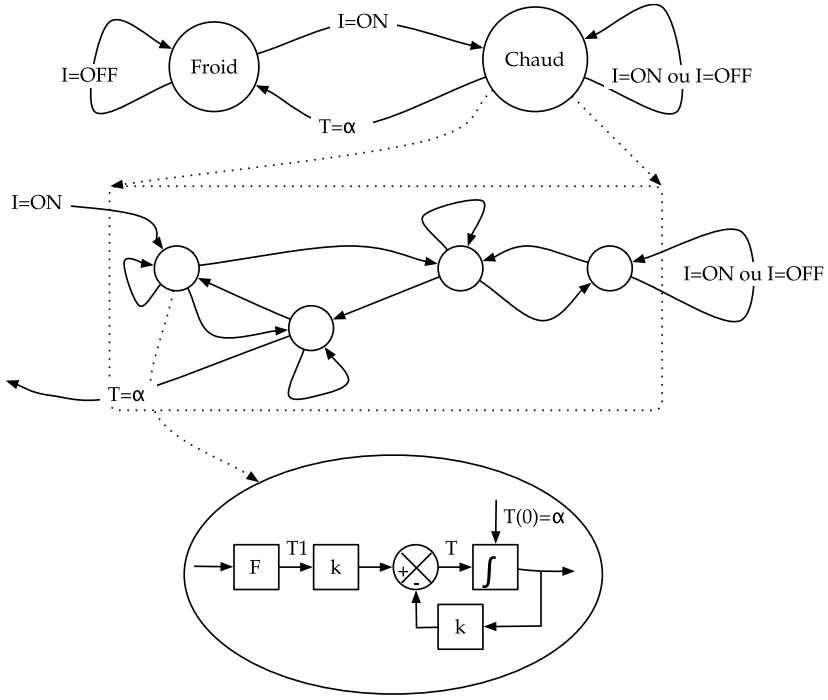


FIG. 1.2 – Exemple de multi-modélisation par décomposition présenté dans [Fishwick, 1993].

Les possibilités de couplage sont nombreuses : pilotage d'un état par un réseau de Petri, décomposition d'une transition d'un réseau de Petri en un automate à états finis, etc. La décomposition peut aussi s'appliquer à des modélisations structurelles. L'exemple précédent traite d'un modèle de comportements. Les règles restent les mêmes : on décompose un élément de la structure en sous-éléments. Si nous nous intéressons aux

écosystèmes forestiers, nous pouvons identifier l'entité *Forêt* comme un tout à un certain niveau d'abstraction. Une forêt peut être vue comme une composition d'arbres qui eux-mêmes sont composés de troncs, de branches, etc.

Quelle que soit la forme de modélisation, il faut observer certaines précautions :

- déterminer les contraintes d'un élément d'un formalisme sur les éléments de l'autre formalisme ;
- vérifier la cohérence sémantique.

Nous avons déjà traité de la première question à travers l'exemple proposé. Le concept de cohérence sémantique est tout aussi important. Il faut, en effet, vérifier la compatibilité sémantique liée, par exemple, aux échelles de temps et d'espace, quand le système est spatialisé, ou tout simplement entre les éléments du modèle.

P. A. Fishwick centre essentiellement son discours sur la décomposition, ou l'agrégation, lorsque l'on modélise le système par une approche *bottom-up*, sans proposer un cadre général. Chaque couple de formalismes voire chaque modèle doit faire l'objet d'une étude ayant pour objectif de vérifier l'adéquation des formalismes. D'autre part, B. P. Zeigler dans [Zeigler, 1976] propose une vision que nous qualifions de couplage de modèles où les formalismes adoptés pour les différents modèles peuvent être différents. Dans cette vision, B. P. Zeigler n'intègre pas l'opération de couplage dans une quelconque démarche (*top-down* ou *bottom-up*). Le couplage fait abstraction de la démarche mais propose, en revanche, un cadre formel du couplage de modèles. Notre point de vue est donc à la croisée de ces deux courants de pensées auxquels nous ajoutons la notion de cohérence sémantique.

1.2 Formalismes

Traditionnellement, nous distinguons d'un côté les outils de modélisation de la structure du système et d'un autre côté les outils de modélisation de la dynamique. Le point de vue structurel décrit les relations statiques qui existent entre les entités du système. Le point de vue dynamique décrit, pour sa part, le comportement des entités du système, c'est-à-dire, l'évolution temporelle de l'état du système et de ses parties. Ces deux points de vue d'un même système ont donné naissance à une multitude d'outils. Dans cette partie, nous réalisons une synthèse des différents outils à partir desquels nous essaierons de dégager la philosophie générale de chaque outil en mettant l'accent sur les éléments fondamentaux et originaux. Nous conclurons par une proposition de classification en fonction des caractéristiques du système.

Cette présentation nous paraît essentielle afin de mieux comprendre les interactions possibles entre les différents formalismes dans une optique de multi-modélisation. Il est nécessaire de comprendre les éléments fondamentaux des formalismes pour assurer la cohérence sémantique lors du couplage de modèles. Si nous nous plaçons dans une

approche plus abstraite telle que DEVS, définie dans la section 1.3, il est nécessaire d'être conscient des capacités et des limites des formalismes. Comme nous allons le voir dans la partie 1.3, DEVS et ses extensions auront pour objectif d'encapsuler les formalismes. Or, il est nécessaire d'observer certaines précautions et cette encapsulation implique dans la majorité des cas des choix conceptuels. Ces choix ne peuvent pas être faits si les aspects conceptuels des différents formalismes ne sont pas clairement exposés.

1.2.1 Formalismes pour la modélisation structurelle

La modélisation structurelle s'attache à représenter les entités du système et leurs relations. Nous allons développer, dans les paragraphes suivants, les formalismes qui se proposent de spécifier uniquement la structure du système. Nous verrons, par la suite, qu'il existe des formalismes où la séparation entre les aspects structurels et comportementaux est difficile. Nous devons dès maintenant donner une définition des concepts d'entité et de relation qui sont à la base de la modélisation structurelle.

Qu'est ce qu'une entité ?

Une entité est un élément d'un système qui possède un état et un comportement. Cette définition est très proche de celle de système, mais la différence réside dans le fait qu'une entité est un système, mais tout système n'est pas une entité. Un système peut être représenté à l'aide des processus caractéristiques. Dans ce cas, aucune entité n'est décrite. Le système est appréhendé au travers de ces variables caractéristiques et de la dynamique de ces dernières. Pour conclure, nous proposons de définir une entité comme un élément discret d'un système. Par exemple, un fluide n'est pas naturellement discret, on choisira donc de le décrire à l'aide d'un système de variables continues. En revanche, un poisson se déplaçant dans le fluide est une entité discrète.

Qu'est ce qu'une relation ?

Une relation est un lien sémantique entre plusieurs entités. La notion de lien sémantique couvre principalement trois aspects :

- un lien d'accointances ou d'interaction : deux entités se connaissent, elles pourront alors s'échanger des informations ;
- un lien d'agrégation ou de composition : une entité est formée de l'assemblage de plusieurs entités, une relation hiérarchique est alors établie ;
- un lien d'héritage : une entité est une généralisation d'une autre entité, les propriétés de l'entité sont formées par l'union des propriétés de l'entité plus abstraite et de ses propres propriétés.

Qu'est ce que la modélisation structurelle ?

La modélisation structurelle est soit la description des entités formant le système et leur mise en relation soit la description des variables caractéristiques des processus du système. Cette définition à double sens va donner naissance à deux courants de modélisation : la modélisation fonctionnelle ou à contraintes et la modélisation déclarative.

La première forme identifie les processus ou les grandes lois gouvernant le système à partir desquels elle identifie les variables caractéristiques. La modélisation comportementale décrit ensuite les relations fonctionnelles entre les variables, par exemple, à l'aide d'équations différentielles. Nous pouvons alors nous demander si ces relations fonctionnelles ne sont pas elles aussi des relations au sens de liens sémantiques. La question est difficile car les relations fonctionnelles entre variables modélisent des processus qui eux sont typiquement des aspects comportementaux et dynamiques des systèmes.

La deuxième forme a pour objectif de décrire les entités et leurs relations. Nous allons donc nous intéresser à cette deuxième forme en raison de l'approche de modélisation que nous avons choisie. Les formalismes dédiés uniquement à la spécification de la structure sont nombreux et souvent très spécialisés au domaine d'applications. Par exemple, en tant qu'électronicien, on peut utiliser des formalismes à base de composants logiques pour décrire des circuits numériques. Les formalismes généraux de cette famille sont rares. Pour notre part, nous allons focaliser sur la modélisation à base d'objets et d'agents.

Modélisation par objets et par agents

Depuis une dizaine d'années, la modélisation objets a normalisé son vocabulaire, ses concepts et ses formalismes. Nous disposons aujourd'hui du langage UML, qui est devenu une norme dans la spécification objets [Booch *et al.*, 1997, Jacobson *et al.*, 1997, Rumbaugh *et al.*, 1997]. UML est un langage graphique qui nous propose plusieurs types de diagrammes. On peut classifier ces diagrammes en trois catégories : les diagrammes statiques pour la structure, les diagrammes dynamiques et les diagrammes de déploiement. Ces derniers n'ont pas d'intérêt pour les objectifs que nous nous sommes fixés ici. Les diagrammes statiques, ou diagrammes de classes et d'objets, représentent la structure statique du système. Ils permettent d'identifier les entités et les relations entre les entités. Deux points de vue sont disponibles : le point de vue « objet », où ce sont les entités du système qui sont mises en relation dans une situation donnée et le point de vue « classe », où ce sont les classes d'objets qui sont mises en relation dans un cadre plus général. Les objets sont des réalisations particulières de classes. Une classe est une abstraction d'un ensemble d'objets qui possèdent des caractéristiques identiques en terme d'informations et de comportements.

Le point de vue de la modélisation objets et de l'UML est conforme aux définitions proposées précédemment. Les entités sont décrites à l'aide des attributs qui les caractérisent et les relations peuvent être typées afin de respecter les différents types de relations mis en évidence précédemment. L'avantage de ce type de modélisation est qu'il est applicable quel que soit le domaine d'applications.

Le second avantage réside dans le fait qu'il est possible d'avoir plusieurs niveaux d'abstractions : le niveau objet, le niveau classe, mais aussi des niveaux dits « meta », où les classes représentent des schémas de classes.

La notion d'agent est proche du concept objet mais elle ajoute des notions d'autonomie, de réactivité et d'activité. Une synthèse de ces travaux peut être trouvée dans [Ferber, 1995] et [Grimm, 1999]. Ainsi, contrairement aux objets, un agent possède un comportement autonome. Il est capable de prendre des décisions et d'établir des plans d'actions pour accomplir des activités complexes. Tous les agents n'ont pas ce degré d'autonomie. Il existe un gradient entre les agents réactifs et les agents cognitifs. Les agents réactifs sont des agents capables de réagir à des stimuli externes en déclenchant des actions. L'approche agents est, comme pour l'approche objets, à la fois un paradigme et un formalisme. Cette affirmation est particulièrement vraie pour l'approche objets comme nous venons de le voir, mais reste partiellement acceptable pour l'approche agents.

1.2.2 Formalismes pour la modélisation dynamique

La modélisation dynamique a pour objectif de décrire le comportement et les interactions des entités du système ou les contraintes sur les variables caractéristiques du système. Ces deux points de vue s'inscrivent dans deux logiques complémentaires : la description des processus ou la description des lois. Cette partition des approches dessine deux familles de formalismes : les formalismes apparentés aux équations différentielles et les autres.

Les systèmes d'équations différentielles

Les équations différentielles sont l'outil privilégié des scientifiques. Les raisons principales sont la puissance d'expressions des mathématiques et des outils formels qui les accompagnent. En effet, à partir d'un système d'équations différentielles, on peut, si ce dernier possède de bonnes propriétés, déduire des propriétés de convergence ou de stabilité à l'infini sous certaines conditions. On peut également manipuler le système avec tous ces paramètres et en déduire des propriétés en fonction des paramètres.

Comme outil de modélisation, les équations différentielles permettent de mettre en équations les grandes lois de la Nature telles que les équations à équilibre, équations de bilan, etc. et d'en déduire la dynamique temporelle. Il est, en effet, facile si les équations

ont de bonnes propriétés de déduire l'évolution temporelle des variables composant le système.

Le reproche que nous pouvons formuler envers les équations différentielles est qu'il n'existe pas toujours le bon théorème ou le bon outil pour le système que l'on construit. Dans ce cas, il n'est plus possible d'en déduire quoi que ce soit sans passer par des outils d'analyse numérique qui procèdent par simulation numérique. On rejoint alors les méthodes par simulation. Cet inconvénient n'ôte pas les capacités de formalisation des Mathématiques.

Le deuxième reproche concerne la représentation d'entités discrètes. Il est très difficile, voire impossible, dans l'état actuel des outils mathématiques courants, de représenter des entités discrètes et de représenter les processus les mettant en jeu. Les équations différentielles sont, dans la grande majorité des cas, utilisées pour représenter des lois ou des contraintes entre variables du système. On ne peut pas faire de description directe des processus. Le modélisateur décrit des équations de bilan, par exemple. Ces bilans sont les conséquences de processus qui ne sont pas représentés. Naturellement, cette approche est aussi intéressante. Il n'est pas toujours possible de décrire les processus car ils ne sont, tout simplement, pas connus. Les équations différentielles accompagnent souvent une approche phénoménologique où l'on s'attache à décrire les conséquences des phénomènes.

Les automates à états finis et à événements finis

Les équations différentielles, décrites dans la partie précédente, sont très bien adaptées à la description des conséquences de processus. Or, si on s'inscrit dans le courant de pensées de B. P. Zeigler et de la spécification formelle de systèmes dynamiques, les notions d'états, de fonctions de transitions et de sortie doivent faire partie du formalisme. Les équations différentielles intègrent la notion d'états mais n'explicitent pas les notions de transitions. De plus, l'approche système dynamique de B. P. Zeigler se focalise sur la description des processus. Les bilans émergent de la dynamique du système. Le prérequis d'une telle approche est que l'on doit posséder la connaissance des processus mis en jeu pour répondre aux problèmes posés.

Les automates à états finis constituent l'outil le plus simple mis à la disposition du modélisateur. Les automates permettent de décrire la dynamique à l'aide des notions d'état et de transition. La structure d'un automate est proche de la structure d'un système dynamique définie dans la section 1.1 :

$$A = \langle X, S, \Delta \rangle$$

où X et S sont respectivement les entrées du modèle et les états du système. La fonction Δ décrit la fonction de transition en indiquant l'état suivant en fonction de l'état courant et de l'entrée du système. Les entrées sont des symboles, mais il est facile de passer de

la notion de symbole à celle d'événement et ainsi de disposer d'un outil de spécification à événements discrets. Nous pouvons représenter graphiquement un automate à états finis à l'aide d'un graphe où les sommets représentent les états et les arcs les transitions. Les automates à événements finis constituent un point de vue complémentaire où les sommets représentent les événements et les arcs la succession possible des événements. Les arcs portent, en général, une durée. Cette durée représente le temps qui s'est écoulé entre les deux événements.

Les automates à états finis, ou à événements finis, sont des outils simplifiés de spécification tels que DEVS que nous présenterons par la suite. Ils permettent de spécifier formellement la dynamique du système en prenant en compte, ou non, explicitement les aspects temporels. La description de la dynamique peut faire partie, ou non, d'une description structurelle, c'est-à-dire, que la dynamique modélisée peut être celle d'une entité ou non. Nous pouvons, en effet, décrire les états du système global sans pour autant décrire le système en terme d'entités.

Il existe de multiples variantes dont les automates non déterministes, où les transitions sont exprimées à l'aide de probabilités. Dans ce cas, les changements d'états suivent un modèle stochastique. Cette variante est utilisée dès lors que l'on ne connaît pas les processus sous-jacents aux changements d'états.

Les diagrammes dynamiques UML : les *state charts*

La modélisation par objets et UML propose une extension objets des automates à états finis : les *state charts*. Comme pour la majorité des diagrammes d'UML, il est possible d'utiliser le formalisme selon différents niveaux de détails. Les *state charts* peuvent être utilisés comme un automate à états finis. Dans sa version complète, le formalisme s'intègre dans une démarche de modélisation objets. Par exemple, l'expression des transitions peut se construire autour des attributs de la classe des entités dont on modélise la dynamique.

Le formalisme des *state charts* est une combinaison des formalismes à base d'états et de transitions. Nous retrouvons :

- des transitions déclenchées sur des événements : si le modèle reçoit un événement, un message dans le vocabulaire objets, provenant d'un autre *state chart* alors la transition est franchie ;
- des transitions temporisées : la transition est franchie au bout d'un certain temps ;
- des transitions conditionnelles : si la condition exprimée en fonction du vecteur d'états est vraie alors la transition est franchie.

L'ensemble de ces types de transitions est présent dans DEVS. Les *state charts* proposent, aussi, des formalisations de la décomposition d'états en sous-états. Cette idée est défendue dans les travaux de P. A. Fishwick. Naturellement, ici la décomposition d'un état donne naissance à un *state chart*. Il n'y a pas d'idée de multi-formalismes.

Les *state charts* proposent une syntaxe relativement riche pour la modélisation de la dynamique d'un système, mais la formalisation proposée n'est pas intégrée à un cadre formel de spécifications de systèmes dynamiques. Néanmoins, on peut montrer que si on prend quelques précautions, les *state charts* peuvent offrir un cadre formel. En effet, il est possible de définir une spécification DEVS d'une partie des *state charts* [Borland et Vangheluwe, 2003]. Cela démontre que les *state charts* sont un candidat précieux à la description de comportement et surtout leurs capacités d'expression sont parfaitement intégrées à une démarche de modélisation objets du système.

1.2.3 Formalismes hybrides ou mixtes

La dernière famille de formalismes est constituée de formalismes où la séparation entre structure et comportement est difficile. Nous ne traiterons, ici, que deux exemples : les réseaux de Petri et les automates cellulaires.

Les réseaux de Petri

Les réseaux de Petri [Peterson, 1977] sont, d'une certaine manière, une extension des automates à états finis. Nous y retrouvons les notions d'état, rôle joué par le marquage, et de transition. Si les réseaux de Petri sont utilisés dans leur forme primaire, ils ressemblent aux automates à états finis. La seule différence réside dans le fait que le franchissement des transitions est fonction du marquage, l'état du modèle et que la description de ces franchissements se fait à l'aide du réseau d'interconnexions des places. La logique de changement d'états est alors guidée par l'algorithme de franchissement des transitions. Les réseaux de Petri offrent, ainsi, une autre manière d'exprimer les changements d'états.

Nous avons classé les réseaux de Petri dans les formalismes hybrides du fait que les modélisateurs utilisant les réseaux de Petri affectent une sémantique aux places du réseau. La sémantique peut être liée à l'état d'une entité du système. Si l'on adopte les réseaux de Petri colorés, alors l'état du système devient beaucoup plus complexe puisque les jetons transportent des informations. Ces informations vont conditionner les transitions. Beaucoup de modélisateurs assimilent les jetons à des entités circulant dans le système.

Les automates cellulaires

Le formalisme des automates cellulaires [von Neumann et Burks, 1966] est un exemple à part. Il décrit des systèmes où l'espace est une composante importante. La notion d'espace est prise ici au sens large : espace de variations d'une variable, ou espace topologique où des entités sont localisées. Ce formalisme hybride décrit la structure

spatiale du système à l'aide d'une grille multidimensionnelle de cellules interconnectées et décrit le comportement local de chaque cellule de l'automate, en fonction de son voisinage.

Les automates cellulaires sont un des formalismes les plus utilisés en modélisation d'écosystèmes. Néanmoins, ce formalisme part du postulat que l'espace est discrétisé et non continu, contrairement aux équations aux dérivées partielles. Ces dernières sont des équations différentielles où les processus sont explicités en fonction du temps et de l'espace, l'espace étant considéré comme continu.

L'espace est une composante présentée dans les modèles de systèmes spatio-temporels. Les automates cellulaires et les équations aux dérivées partielles ne sont pas les seules réponses mais restent les plus couramment utilisées et surtout ces formalismes décrivent explicitement l'espace. On peut construire un modèle de type objet où les entités « embarquent » la description de l'espace. Il est très facile d'introduire, dans les caractéristiques de l'objet, ses coordonnées. La définition de l'espace ne fait pas alors l'objet d'une description propre. Néanmoins, la dynamique des objets ou des agents tient compte de certains aspects de l'espace. Si l'espace est fermé ou torique, lors du déplacement d'une entité dans l'espace, le processus de déplacement doit garantir le respect de la cohérence des coordonnées.

1.2.4 Une classification des formalismes

Nous venons de présenter quelques exemples de formalismes que l'on peut rencontrer en modélisation de systèmes complexes. Chaque formalisme est spécialisé dans la spécification d'un aspect du système : la description de la structure ou du comportement, la description de l'espace, etc. Ainsi, nous pouvons établir une classification des formalismes en fonction des aspects à modéliser et des propriétés du système ou d'une partie du système. Cette classification a un seul objectif : montrer qu'il est fondamental de bien choisir son formalisme en fonction de son système. Cette démarche s'inscrit dans une approche par décomposition de son système en sous-systèmes et il est important d'adopter le formalisme le plus adéquat pour la représentation de chaque sous-système.

Cette classification a aussi pour objectif de montrer que certains aspects ne sont pas pris en compte de la même façon selon les formalismes. Cette différence nous conduira à nous poser certaines questions dans le cadre de la multi-modélisation. Un modèle qui manipule des changements d'états continu et un autre de manière discrète sont-ils couplages ? Si oui, quelles sont les précautions à prendre ?

Afin d'établir une classification, il est nécessaire de déterminer les propriétés discriminantes des systèmes ou de leur modélisation :

– discret ou continu : cette propriété est à considérer selon plusieurs points de vue, changement d'états, espace, temps, etc.

– les processus modélisés sont-ils déterministes ou stochastiques ?
– l'espace est-il à prendre en compte ?
– le temps a-t-il une importance ?

La première propriété est probablement la plus importante. Pour les systèmes spatio-temporels où l'espace et le temps font partie des modèles, le modélisateur doit faire un choix de représentation de ces derniers. Il est bien évident que définir le temps comme une variable discrète ou continue n'implique pas les mêmes outils de modélisation. Typiquement, les automates cellulaires manipulent un temps discret. À chaque pas de temps, l'état de l'automate est calculé en fonction de l'automate à l'instant précédent ; et les équations différentielles ordinaires manipulent un temps continu. Par exemple, la température évolue de manière continue dans le temps alors que le nombre de produits en fin de chaîne de production évolue de manière discrète. Les modèles construits autour de ces variables ne sont pas de la même nature et ne mettent pas en jeu les mêmes formalismes.

Comme le proposent les automates à états finis, il est possible de spécifier les transitions d'états soit de manière déterministe soit de manière stochastique. Ce choix n'est pas obligatoirement guidé par une propriété de stochasticité du système mais par l'approche adoptée pour la modélisation des processus. Si le niveau de connaissances sur le système n'est pas suffisant, les probabilités modélisent la proportion d'apparition de l'état suivant en fonction de l'état courant. Cette probabilité est déterminée par l'observation du phénomène.

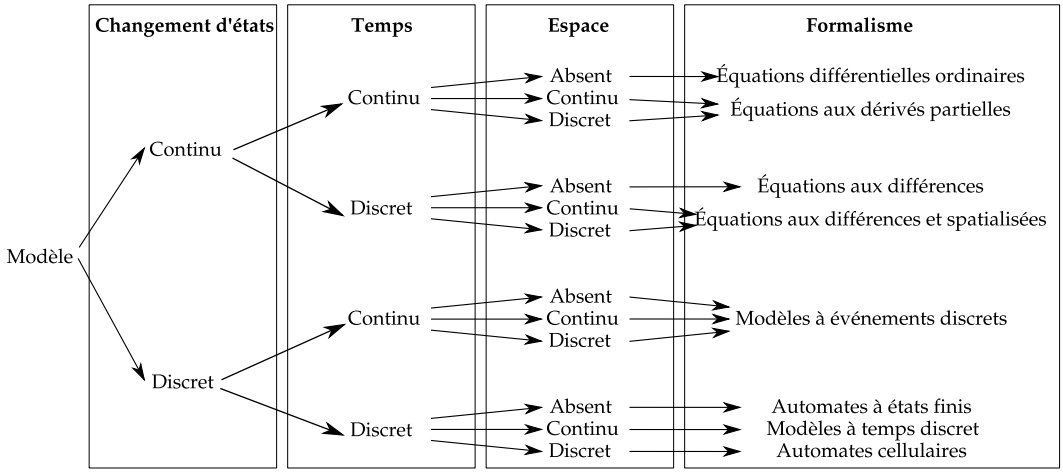


FIG. 1.3 – Classification des formalismes selon les aspects continus ou discrets des variables du temps et de l'espace (Source : [Ramat, 2003]).

La figure 1.3, proposée dans [Ramat, 2003], dresse une classification des formalismes dédiés à la spécification de la dynamique des systèmes en fonction de trois critères : la manipulation de variables continues ou discrètes, la représentation de l'espace continu ou discret et la présence du temps continu ou discret. Elle nous montre, également,

la multitude des formalismes et l'adéquation aux hypothèses retenues pour le modèle. Dans le cadre de la multi-modélisation, cette diversité de formalismes va poser la question du couplage de modèles : comment coupler deux modèles exprimés dans des formalismes différents ? La présentation de DEVS va nous apporter un éclairage unificateur des formalismes et nous allons montrer comment s'affranchir des formalismes.

1.3 DEVS : intégration de modèles

Comme nous l'avons vu précédemment, nous pouvons définir cinq niveaux de spécifications formelles d'un système dynamique [Zeigler, 1976]. Cependant, pour répondre au problème d'intégration de modèles ou de couplage de modèles, il faut répondre à la question suivante : quel est le niveau le plus adéquat pour l'intégration de modèles ? La question sous-jacente est : quel est le niveau d'intégration que l'on désire ? Si le degré d'intégration est faible, nous pouvons nous contenter de spécifier tout modèle avec un niveau 0, la « boîte noire ». Mais quel est alors le potentiel d'une telle intégration ?

Depuis les années 1970, des travaux formels ont été menés pour développer les fondements théoriques de la modélisation et la simulation des systèmes dynamiques à événements discrets [Zeigler, 1976]. B. P. Zeigler s'attache alors à travailler au niveau systèmes dynamiques et systèmes couplés. DEVS, *Discrete EVent system Specification*, a été introduit comme un formalisme abstrait pour la modélisation à événements discrets et est un formalisme universel.

1.3.1 DEVS atomique

Un modèle DEVS atomique possède la structure suivante :

$$DEVS = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$$

où :

$$\begin{cases} X \text{ est l'ensemble des ports et des valeurs d'entrée,} \\ Y \text{ l'ensemble des ports et des valeurs de sortie,} \\ S \text{ l'ensemble des états du système,} \\ \delta_{ext} \text{ la fonction de transition externe,} \\ \delta_{int} \text{ la fonction de transition interne,} \\ \delta_{con} \text{ la fonction de transition conflit,} \\ \lambda \text{ la fonction de sortie,} \\ ta \text{ la fonction d'avancement du temps.} \end{cases}$$

Cette structure est très proche de celle proposée au niveau systèmes dynamiques définie dans la section 1.1. La seule différence réside dans la séparation de la fonction de transition en trois fonctions distinctes.

Un modèle DEVS avec port[1] peut-être représenté graphiquement. La figure 1.4 représente un modèle atomique par un rectangle avec un ensemble d'entrées appelées ports d'entrées et de sorties appelées ports de sortie, représentés par des triangles remplis. Les vecteurs d'entrée et sortie sont l'union de tous les ports du modèle.

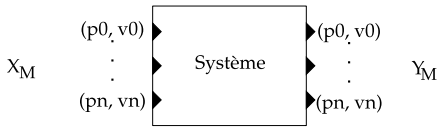


FIG. 1.4 – Représentation graphique d'un modèle atomique DEVS avec port.

La valeur v_i est la valeur prise par un port d'entrée ou de sortie. Cette valeur appartient à l'ensemble des valeurs possibles du port p_i. Un port d'entrée prend une valeur lors de l'émission d'un événement attaché à ce port. Un port de sortie prend une valeur lorsque la fonction de sortie prend une valeur pour ce port.

S est l'ensemble des états du système. L'ensemble des valeurs pris par le vecteur d'états à un instant donné e est appelé état du système. L'ensemble S_T des états totaux du système est :

$$S_T = \{(s,e)|s \in S, 0 < e < ta(s)\}$$

où e représente le temps écoulé dans l'état s. Ce concept d'état total (s,e) est fondamental car il permet de spécifier un état futur en fonction du temps écoulé dans l'état présent.

$ta(s_k)$ est le temps pendant lequel le modèle restera dans l'état s_k, si aucun événement externe ne survient. Un état s de S est dit passif si et seulement si sa durée de vie est infinie. Dans ce cas, la fonction de transition interne n'est pas définie.

La fonction classique de transition est décomposée en deux fonctions, représentant respectivement les évolutions autonomes et celles dues à des événements externes. La fonction de transition interne, partie autonome de la fonction de transition, est définie par :

$$\delta_{int} : S \rightarrow S$$

[1] B. P. Zeigler propose dans [Zeigler *et al.*, 2000] principalement deux formes de DEVS, classiques ou avec ports. Par la suite, nous emploierons uniquement la définition de DEVS avec ports qui permet de mieux découper la sémantique des modèles.

Elle spécifie les états futurs des états actifs, états dont la durée de vie n'est pas infinie. Ces états correspondent à des régimes transitoires ou des états instables du système. La fonction de transition externe est :

$$\delta_{ext} : S \times X \rightarrow S$$

La fonction de transition externe δ_{ext} représente la réponse du système aux événements d'entrée. Le système est dans un état (s, e) à un instant t. Lorsqu'un événement externe survient, la fonction δ_{ext} indique quel est le nouvel état du système en fonction de s.

Cette décomposition de la fonction de transition en deux fonctions constitue l'un des points forts du formalisme DEVS. En effet, elle autorise une spécification des évolutions autonomes du modèle, évolutions autonomes représentant des régimes transitoires ou des états instables du système réel.

La spécification des changements d'états est complétée par la fonction de conflit, notée δ_{con}, qui permet de spécifier l'état futur dans le cas d'événements simultanés dans le monde du modèle. Cette fonction est présente seulement dans une variante de DEVS qui se nomme *Parallel DEVS* [Zeigler *et al.*, 2000]. Nous présenterons cette variante de DEVS dans le chapitre 2.

$$\delta_{con} : S \times X \rightarrow S$$

La fonction de sortie est une application de l'ensemble des états S dans l'ensemble des sorties Y. Elle est définie par :

$$\lambda : S \rightarrow Y$$

Cette fonction sera activée lorsque le temps écoulé dans un état donné sera égal à sa durée de vie. Par suite, λ n'est définie que pour des états actifs.

Un système peut être autonome et donc ne recevoir aucun événement extérieur. La dynamique du système est alors le seul fait de la fonction de transition interne. Cette fonction de transition est définie pour spécifier les changements d'états dûs exclusivement à l'état interne du système et au temps.

Le système est entré à l'instant t dans l'état s_t. Si aucun événement extérieur ne survient alors le système changera d'état à $t + ta(s_t)$. La fonction ta donne la durée pendant laquelle le système sera dans un certain état. Si cette durée est nulle alors on l'appellera état transitoire. À l'inverse, si la durée est infinie alors on l'appellera état passif. Un événement extérieur peut survenir exactement à $t + ta(s_t)$. Le nouvel état est alors défini par la fonction de transition δ_{con}. Cette fonction règle les problèmes de conflit entre les transitions dues aux événements extérieurs et les transitions internes.

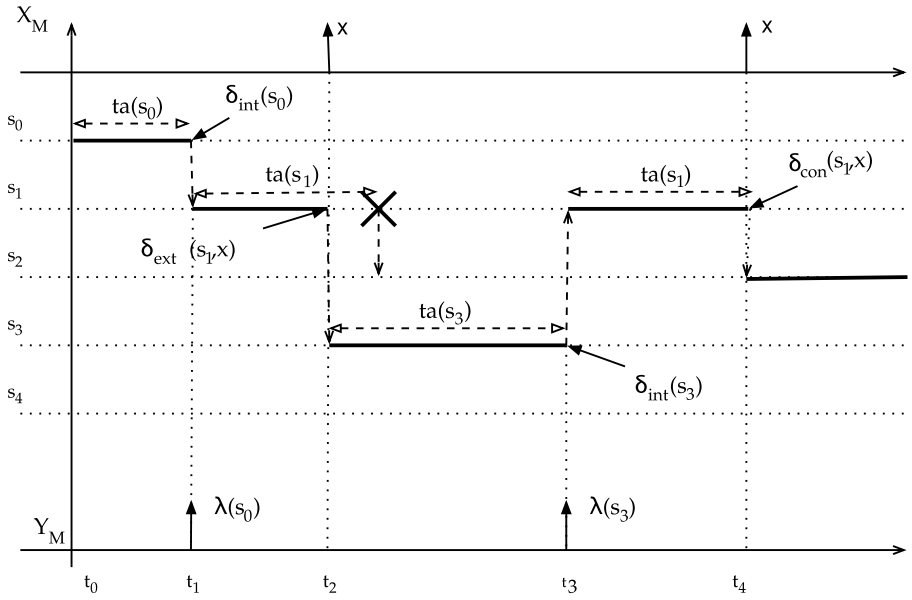


FIG. 1.5 – Exemple de graphe de transitions.

La figure 1.5 illustre l'évolution d'un modèle DEVS sur un exemple. À l'état initial $t = 0$, le système est dans l'état s_0. La fonction ta nous indique que pour l'état s_0, le système changera d'état à $t + ta(s_0)$ si aucun événement extérieur ne survient. À $t_1 = t + ta(s_0)$, aucune entrée n'a eu lieu. La fonction de sortie est donc activée et Y prend pour valeur la valeur produite par la fonction λ pour l'état s_0. Après avoir affecté les ports de sortie, la fonction de la transition interne δ_{int} est appliquée. Le système passe dans l'état $s_1 = \delta_{int}(s_0)$ et changera d'état à $t_1 + ta(s_1)$. À l'instant t_2, qui est inférieur à $t_1 + ta(s_1)$, un événement extérieur est placé en entrée. On fait alors appel à la fonction de transition externe pour déterminer le nouvel état. Dans ce cas, la fonction de sortie n'est pas appliquée. Elle est appliquée exclusivement lors de transition interne. À l'instant t_2, le système passe dans l'état $s_3 = \delta_{ext}(s_1)$. Si aucun événement externe n'avait eu lieu, le système serait à $t_1 + ta(s_1)$ dans l'état $s_2 = \delta_{int}(s_1)$. Le système évolue ensuite jusqu'à l'instant t_4 par transition interne. À l'instant t_4, la fonction de transition interne doit être activée mais au même moment un événement extérieur survient. La fonction δ_{con} règle le conflit en indiquant le nouvel état. Ce nouvel état est fonction de l'état courant et de l'événement d'entrée. La fonction de sortie ne prend pas de valeur.

1.3.2 DEVS couplé

Un modèle couplé définit comment sont couplés des modèles entre eux pour former un nouveau modèle. Il peut lui-même faire partie d'un modèle couplé. On définit alors une construction hiérarchique de modèles. Un modèle couplé comprend les informations suivantes :

– l'ensemble des modèles qui le composent ;
– l'ensemble des ports d'entrée qui recevront les événements externes ;
– l'ensemble des ports de sortie qui émettront les événements ;
– les couplages en ports d'entrée et ports de sortie des modèles composant le modèle couplés.

Un modèle couplé, aussi appelé réseau de modèles, possède la structure suivante :

$$N = \langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC \rangle$$

La définition de X et Y est identique à celle de X_M et Y_M d'un modèle atomique. Les entrées et sorties sont composées de ports, chaque port peut prendre des valeurs, chaque port possède son propre domaine de valeurs.

D est l'ensemble des identifiants des modèles intervenant dans le modèle couplé. M_d est un modèle DEVS appartenant au réseau. Les variables représentant les entrées et les sorties du modèle seront indexées par l'identifiant du modèle. D'où la notation suivante :

$$M_d = \langle X_d, Y_d, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$$

Les entrées et les sorties du modèle couplé noté N sont connectées aux entrées et sorties des modèles composant le modèle couplé. EIC représente la liste de ports d'entrée du modèle couplé, $IPorts$, qui sont connectés aux ports d'entrée des sous-modèles $IPorts_d$ avec $d \in D$.

$$EIC = \{((N, a), (d, b)) | a \in IPorts, b \in IPorts_d\}$$

La même situation se présente pour les ports de sortie où EOC représente la liste de ports de sortie du modèle couplé, $OPorts$, qui sont connectés aux ports de sortie des sous-modèles $OPorts_d$ avec $d \in D$.

$$EOC = \{((N, a), (d, b)) | a \in OPorts, b \in OPorts_d\}$$

À l'intérieur du modèle couplé, les sorties d'un modèle peuvent être couplées aux entrées des autres modèles. Une sortie d'un modèle ne peut pas être couplée à l'une de ses entrées. Nous verrons par la suite que cette contrainte pourra être supprimée lors de l'utilisation de l'extension DEVS Parallèle [Zeigler *et al.*, 2000] que nous introduirons dans le chapitre 2 section 2.3.

$$IC = \{((i, a), (j, b)) | i, j \in D, i \neq j, a \in OPorts_i, b \in IPorts_j\}$$

Pour illustrer le modèle formel, nous développons un exemple simple de couplage de modèle dont la représentation graphique est fournie sur la figure 1.6.

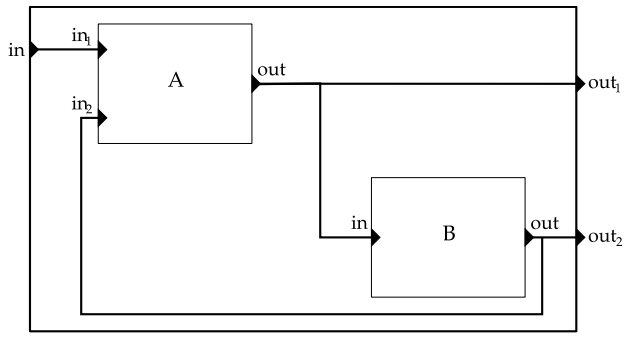


FIG. 1.6 – Représentation graphique d'un modèle couplé composé de deux modèles composants.

Les caractéristiques de ce modèle couplé sont :

$$\begin{cases} IPortsN = \{in\}, \\ PortsN = \{out_1, out_2\}, \\ D = \{A, B\}, \\ EIC = \{((N, in), (A, in_1))\}, \\ EOC = \{((A, out), (N, out_1)), ((B, out), (N, out_2))\}, \\ IC = \{((A, out), (B, in)), ((B, out), (A, in_2))\}, \end{cases}$$

Si le modèle couplé fait partie lui-même d'un modèle couplé, alors il faut définir les paramètres standards d'un modèle DEVS. Le vecteur d'états du modèle couplé est le n-uplet composé des vecteurs d'états des modèles composants le modèle couplé.

$$S_N = \{S_d | d \in D\}$$

L'ensemble des valeurs de sortie du modèle couplé est défini par :

$$Y = \{(p, v) | p \in OPorts, v \in Y_p\}$$

Nous pouvons les exprimer en fonction des valeurs de sortie des modèles composant le modèle couplé en précisant Y_p. Y_p est l'ensemble des valeurs prises par le port de sortie p du modèle couplé.

$$Y_p = \{v | (p', v) \in Y_p, p' \in OPorts_d, d \in D, ((d, p'), (N, p)) \in EOC\}$$

Les valeurs d'un port de sortie sont les valeurs du port qui lui sont connectées soit un port de sortie d'un modèle le composant.

$$\delta_{ext_S}(S_N,(p,v)) = \{S_{d'}/d' \in D, d \neq d'\} \cup \delta_{ext_d}(S_d,(p',v))$$

Tel que :

$$((N,p),(d,p')) \in EIC$$

Les deux sections suivantes vont nous permettre d'illustrer le formalisme DEVS à partir d'extensions souvent implémentées dans les modèles tels que les automates cellulaires et le changement de structures.

1.3.3 Cell-DEVS

L'extension Cell-DEVS est née d'une constatation selon laquelle de nombreux modèles font intervenir des espaces discrets et utilisent des formalismes tels que les automates cellulaires. En effet, dès lors que l'on doit représenter l'espace deux possibilités sont offertes :

- l'espace est continu : on définit une origine et un repère par rapport auquel toute entité doit se repérer ;
- l'espace est discret : on divise l'espace en régions ; dans la majorité des cas, toute entité sera localisée sur une et une seule région.

G. Wainer et N. Giambiasi, dans [Wainer et Giambiasi, 2001], développent l'extension Cell-DEVS. Cette extension permet de décrire et de simuler des modèles à base d'automates cellulaires multi-dimensionnels et à événements discrets. La dynamique des cellules est temporisée, c'est-à-dire, l'état d'une cellule sera modifié en fonction de l'état de son voisinage mais il ne sera connu des cellules voisines qu'après un certain délai. L'idée de base est de fournir un mécanisme simple de définition de la synchronisation des cellules. Comme toute proposition d'extensions, les auteurs offrent à la fois l'extension du formalisme qui se résume à l'ajout de variables supplémentaires et de leur sémantique et le simulateur abstrait. Deux spécifications sont proposées : l'une pour la dynamique des cellules et l'autre pour la dynamique de l'automate complet. Un modèle Cell-DEVS est défini comme un espace composé de cellules qui peuvent être couplées afin de former un espace complet. La sémantique liée aux cellules n'est pas précisée, seule la dynamique du modèle couplé fait l'objet d'une description détaillée. Il est donc possible d'utiliser ce formalisme pour représenter un espace réel, un lieu de déplacement pour des entités ; ou un espace plus abstrait, un espace comme un ensemble de lieux abstraits.

Un modèle Cell-DEVS est basé sur la même structure qu'un modèle DEVS classique. Nous retrouvons les modèles atomiques pour les cellules, les éléments de base d'un réseau et les modèles couplés pour les réseaux eux-mêmes. Néanmoins, la structure

proposée par B. P. Zeigler [Zeigler *et al.*, 2000] se voit augmentée d'attributs. Le modèle DEVS d'une cellule est défini par la structure suivante :

$$TDC = \langle X, Y, I, S, N, delay, d, \delta_{int}, \delta_{ext}, \tau, \lambda, ta \rangle$$

où :

$$\begin{cases} X \text{ est l'ensemble des ports et des valeurs d'entrée,} \\ Y \text{ l'ensemble des ports et des valeurs de sortie,} \\ I \text{ l'interface de la cellule,} \\ S \text{ l'ensemble des états de la cellule,} \\ N \text{ l'ensemble des états des cellules voisines,} \\ delay \text{ le type de délai utilisé,} \\ d \text{ la durée du délai,} \\ \delta_{ext} \text{ la fonction de transition externe,} \\ \delta_{int} \text{ la fonction de transition interne,} \\ \tau \text{ la fonction locale de calcul,} \\ \lambda \text{ la fonction de sortie,} \\ ta \text{ la fonction d'avancement du temps.} \end{cases}$$

L'interface I de la cellule définit le voisinage de la cellule ainsi que les connexions en terme de ports d'entrée et de sortie entre la cellule et ses voisines. Il y a autant de ports d'entrée que de voisins. La fonction de calcul τ modélise la fonction de calcul de l'état de la cellule en fonction de l'état du voisinage. C'est une fonction booléenne et elle utilise I et N pour effectuer son calcul. L'état de la cellule n'est effective pour les cellules voisines qu'au bout du délai d'attente d. Ce délai peut être de trois types différents :

- transport : le délai représente tout le temps de propagation du signal dans la cellule ;
- inertiel : le délai est ici relié à l'inertie du système ; il faut un certain niveau d'énergie pour que le système change d'état ;
- signal : le délai est dû au temps de changement d'état ; pour passer de signal haut à signal bas, certains systèmes ont besoin d'un certain temps.

Le type de délai influence la fonction de sortie. Comme nous l'avons préalablement évoqué, l'état de la cellule est calculé par la fonction τ. Cette dernière met à jour une partie de l'état S de la cellule. En effet, S se décompose en un attribut représentant l'état booléen de la cellule, un attribut indique la période d'activité et deux autres attributs stockent les états successifs de la cellule, la figure 1.7 représente ce fonctionnement.

La période d'activité de la cellule est proposée sous la forme d'une variable prenant la valeur *active* ou *passive*. Une cellule est dite active si la cellule doit communiquer des états aux cellules voisines. La communication des états s'effectue par l'intermédiaire

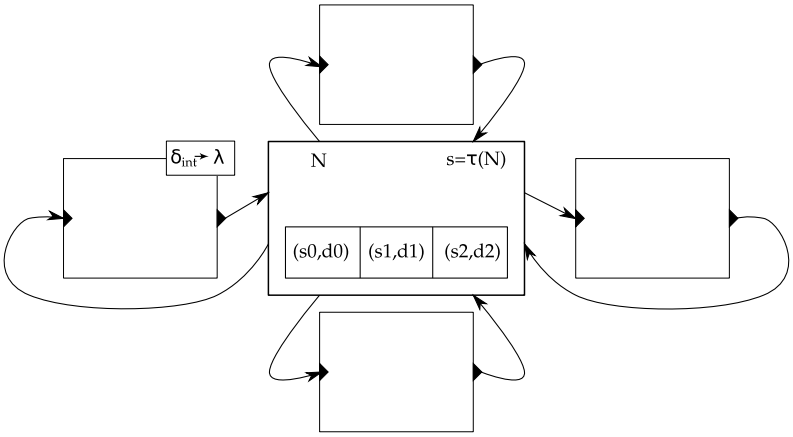


FIG. 1.7 – Une cellule du réseau avec ses connexions aux cellules voisines.

des ports de sortie et d'entrée des modèles, comme pour un modèle DEVS classique. En revanche, l'introduction d'un délai de transmission implique une dynamique bien particulière et nécessite l'introduction d'une file d'attente. Lorsqu'une cellule voisine informe de son changement d'état, la fonction τ est calculée et le nouvel état est alors stocké dans la variable s de S. De plus et conformément au délai de transmission, ce nouvel état n'est pas transmis immédiatement aux voisins. Il est alors stocké dans une file d'attente avec la valeur du délai. Si rien ne se produit avant la fin du délai, la cellule informe ses voisines qu'elle a changé d'état. En revanche, si une nouvelle cellule voisine informe de son changement d'état avant le délai, la cellule change d'état et stocke dans la file d'attente le nouvel état avec le délai. Dès que le premier délai est écoulé, l'état est communiqué, ainsi que le second lorsque le second délai est écoulé. Cette dynamique devient complexe dès lors que les délais des différentes cellules ne sont pas identiques. La description qui vient d'être faite est dite de transport. Dans le cas de délai inertiel, l'état transmis au voisinage peut ne pas suivre l'état de la cellule. Si la cellule revient dans le même état avant le délai inertiel, alors le changement d'état n'est pas signalé.

La définition des cellules est complétée par la définition d'un modèle couplé. Le modèle d'une cellule définit l'interface qu'elle offre à l'extérieur mais ne précise pas la forme des connexions. Le rôle du modèle couplé est donc de définir les connexions entre les cellules, à l'aide d'un *pattern* - N, C et B, la taille $\{t_1, ..., t_n\}$ et le nombre de dimension n du réseau et l'interface I avec d'éventuels modèles DEVS de type Cell-DEVS ou non.

$$GCC = \langle X_{list}, Y_{list}, I, X, Y, n, \{t_1, ..., t_n\}, N, C, B, Z) \rangle$$

X_{list} et Y_{list} définissent la liste des cellules du réseau possédant des ports d'entrée et des ports de sortie non connectés en interne et par conséquent, disponibles pour une

connexion avec un autre modèle. En terminologie DEVS, cet ensemble est l'ensemble des connexions entre les ports d'entrée et des ports de sortie avec les ports du modèle couplé. L'interface I du réseau complète la définition en réunissant au sein d'une même structure les éléments de définition de l'interface du réseau vers l'extérieur. Cette définition intègre l'ensemble Z qui met en relation les ports de sortie appartenant à Y_{list} d'un réseau et les ports d'entrée appartenant à X_{list} d'un autre réseau. X et Y représentent l'ensemble des événements d'entrée et de sortie. De manière à simplifier la définition des connexions entre les cellules du réseau, communément noté connexion interne dans les modèles couplés, un *pattern* de voisinage, noté N, est défini. Ce *pattern* spécifie pour toute cellule n'appartenant pas à la bordure B la position relative de ses voisins.

Avec Cell-DEVS, nous disposons d'un outil de spécification DEVS pour les automates cellulaires. Il faut noter que Cell-DEVS ne s'arrête pas à un langage de spécification mais offre aussi les simulateurs abstraits afin de préciser le comportement d'un modèle Cell-DEVS. Cette remarque est de nouveau valable comme pour la totalité des extensions de DEVS.

En conclusion, si l'une des parties d'un modèle nécessite une description de type automate cellulaire, il suffit de la spécifier en Cell-DEVS et elle aura la capacité d'être couplée à d'autres modèles compatibles DEVS.

1.3.4 DS-DEVS

L'un des reproches de DEVS, mais aussi de bien d'autres formalismes, est leur incapacité à changer dynamiquement de structure. Les formalismes peuvent, en général, seulement représenter les changements d'états en fonction des événements d'entrée et de la dynamique interne. Les changements de structures sont alors possibles en les intégrant dans les variables descriptives du système. On mélange alors les aspects purement comportementaux avec des aspects de structure. Les exemples de changements de structures sont nombreux :

- un nouveau serveur qui apparaît dans un réseau complexe ;
- le remplacement d'un composant défectueux par un autre ;
- la transformation d'une particule en une autre particule :
- l'apparition et la disparition, de produits dans un système de production ;
- le changement de modèles dans une simulation en cours d'exécution ;
- une entité avec un comportement dans un environnement d'entités communicantes (de type système multi-agents).

Dynamic Structure DEVS a été défini par F. J. Barros dans [Barros, 1995] afin de pallier à cette insuffisance. Ce formalisme est basé sur DEVS et fournit tous les mécanismes pour le changement dynamique de la structure d'un modèle DEVS. DS-DEVS introduit une nouvelle spécification pour les modèles couplés mais n'effectue aucune modification de spécification des modèles atomiques. Dans les paragraphes suivants, nous décrivons les

différences afin de comprendre quels sont les apports potentiels pour la formalisation des systèmes complexes avec une structure dynamique.

Le changement dynamique de structures est introduit à l'aide de la définition d'un réseau de modèles DEVS atomiques. Les différences par rapport aux modèles couplés de DEVS sont que la liste des connexions et la liste des modèles de ce réseau peuvent changer au cours du temps. Ce réseau est défini à l'aide de l'ensemble des modèles atomiques activables Δ et le réseau des modèles actifs χ. Ce réseau est représenté par la structure suivante :

$$DSDEVSN_{\Delta} = \langle X_{\Delta}, Y_{\Delta}, \chi, M_{\chi} \rangle$$

Les attributs X_{Δ} et Y_{Δ} représentent l'ensemble des ports d'entrée et des ports de sortie du modèle global. Tous les ports ne sont pas obligatoirement actifs. Le modèle M_{χ} précise de quelle manière la structure du modèle actif change au cours du temps. Par conséquent, ce modèle définit les ports actifs. Le modèle M_{χ} est représenté par la structure suivante :

$$M_{\chi} = \langle X_{\chi}, S_{\chi}, Y_{\chi}, \delta_{int_{\chi}}, \delta_{ext_{\chi}}, \lambda_{\chi}, \tau_{\chi} \rangle$$

Cette structure est compatible avec celle des modèles atomiques classiques afin de respecter la spécification DEVS. Pour comprendre comment est décrit le changement dynamique de la structure, il faut détailler le vecteur d'états S_{χ}. Ce vecteur contient les informations de l'état de la structure du réseau et est défini par le n-uplet suivant :

$$s_{\chi} = (X_{\Delta}^{\chi}, Y_{\Delta}^{\chi}, D^{\chi}, \{M_i^{\chi}\}, \{I_i^{\chi}\}, \{Z_{i,j}^{\chi}\}, \Xi^{\chi}, \theta^{\chi})$$

Tout changement de ce n-uplet traduit un changement de la structure du réseau des modèles. Par exemple, la valeur X_{Δ}^{χ} définit l'ensemble de ports d'entrée actifs. Cet ensemble est un sous-ensemble de X_{Δ} et s'il est modifié cela signifie que les événements admissibles par le modèle changent. Cette modification est, en général, la conséquence de l'activation, ou de la désactivation, d'un modèle composant le modèle global actif.

D^{χ} est l'ensemble des identifiants des modèles actifs et $\{M_i^{\chi}\}$ l'ensemble des modèles actifs où i appartient à D^{χ}. Nous pouvons généraliser la définition en considérant les modèles actifs soit comme des modèles atomiques, soit comme des modèles couplés. Les ensembles $\{I_i^{\chi}\}$ et $\{Z_{i,j}^{\chi}\}$ définissent, quant à eux, les connexions entre les modèles actifs du réseau. Nous ne détaillerons pas ces deux ensembles qui jouent le même rôle que les ensembles EIC, EOC et IC des modèles couplés. La structure est complétée par une fonction de sélection Ξ^{χ}. Cette fonction gère les problèmes de conflits lorsque plusieurs événements se produisent à la même date et que les destinataires appartiennent au réseau. Comme toute variable d'états, la variable s_{χ} inclue l'attribut θ_{χ} qui permet de

définir des variables d'états propres au réseau. Il faut noter que cet attribut n'existe pas dans le cas des modèles couplés. Les modèles couplés définissent seulement un réseau de connexions entre modèles, mais n'intègrent pas de dynamique propre.

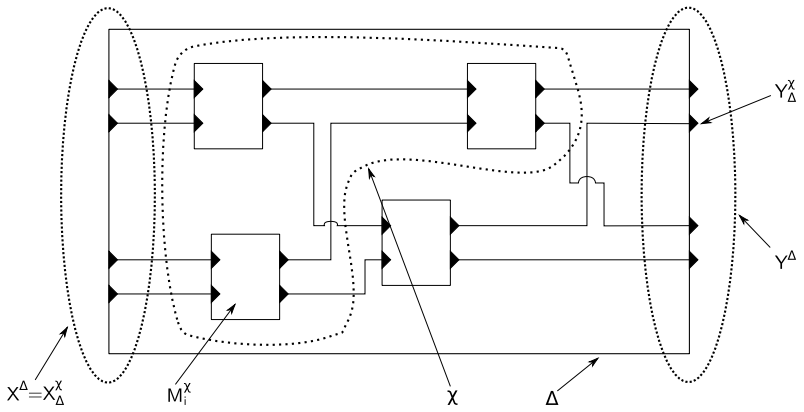


FIG. 1.8 – Représentation graphique d'un modèle DS-DEVS.

F. J. Barros démontre dans son article [Barros, 1995] que le formalisme DS-DEVS est conforme aux spécifications DEVS et possède donc la capacité d'être couplé avec tout modèle DEVS. De plus, deux simulateurs abstraits sont proposés. Ces deux simulateurs décrivent les opérations réalisées lors de l'exécution des fonctions de transitions internes et externes du réseau ainsi que l'initialisation des modèles. Quant aux modèles actifs du réseau, ils disposent des simulateurs abstraits des modèles atomiques DEVS. L'algorithme ci-dessous représente le traitement des événements externes pour le réseau. Cet algorithme nous permet d'illustrer la prise en compte du changement de structures.

```
Quand réception d'un événement externe : (x,t)
  Si t_l ≤ t ≤ t_n Alors
    D^α = D^χ
    Envoyer (x,t) à tous les modèles dont une entrée est connectée à
      l'entrée du modèle globale sollicité par (x,t)
    Attendre jusqu'à ce que tous les modèles aient acquitté du traitement
      de (x,t)
    Envoyer un message d'initialisation à tous les modèles appartenant à
      D^α − D^χ
    Attendre jusqu'à ce que tous les modèles aient acquitté du traitement
      de l'événement d'initialisation
    t_l = t
    t_n = min{t_{n,i} | i ∈ D^χ}
    Envoyer au simulateur père l'événement (fait, t_n)
  Sinon
    "Erreur"
  Fin si
Fin quand
```

t_l et t_n représentent respectivement la date du dernier événement et la date de fin de l'état courant $t_n = t_l + ta(s)$. L'algorithme ci-dessus ne met pas directement en évidence le changement de structures. Lorsque l'événement (x, t) est envoyé à tous les modèles connectés à l'entrée sollicitée, le réseau attend que tous les modèles aient traité le message. Suite à ce traitement, la structure peut alors changer. Dans ce cas, l'algorithme envoie un événement d'initialisation à tous les nouveaux modèles. Si, au contraire, des modèles ont été désactivés lors du calcul de la date de fin d'états, seules les dates de fin d'états des modèles actifs sont prises en compte. Les mêmes remarques auraient pu être faites pour le traitement des événements internes ou de fin d'états.

En conclusion, DS-DEVS nous offre un formalisme et une algorithmique pour les structures dynamiques de modèles couplés. Nous citons F. Barros qui écrit, au sujet des systèmes à structure dynamique en général :

> « *[...] structural changes can better reflect dynamic of real systems we want to model in which drastic changes of structure and behaviors can be observed.*[2] »
>
> F. Barros [Barros, 1995]

1.4 Conclusion

Ce premier chapitre nous a permis de définir les concepts de base de la modélisation, de la multi-modélisation et de la spécification formelle de systèmes dynamiques avec les notions de système, de système complexe, de schéma expérimental, de paradigme et de formalisme. Nous avons replacé ces termes dans leur contexte épistémologique avec un rappel sur l'histoire des Sciences.

Au travers du discours, qui nous dit que nous sommes conditionnés par notre vision des choses et par nos connaissances, nous avons mis en évidence les problématiques sous-jacentes : l'adéquation du schéma expérimental par rapport aux interrogations sur le système, l'adéquation du formalisme au système et au paradigme, l'intégration d'une démarche au sein de l'activité de modélisation, la cohérence sémantique du couplage de modèles, etc.

Nous avons abordé ces problématiques au travers de diverses définitions pour aboutir à une proposition. Cette proposition s'inscrit à la fois dans les idées de P. A. Fishwick et de multi-modélisation et dans les idées de B. P. Zeigler et de spécifications formelles de systèmes. La multi-modélisation est vue comme un processus de modélisation mariant des idées de décomposition ou d'agrégation et des idées de couplage de paradigmes et de formalismes. Les processus de décomposition et d'agrégation sont inhérents à l'approche de modélisation adoptée : *top-down* ou *bottom-up*. Ces approches sont conformes à la vision dominante : le réductionnisme. Le fait d'adopter au sein des processus de

[2] « *[...] les changements de structures peuvent mieux refléter la dynamique des systèmes réels que nous modélisons pour lesquels des changements drastiques de structures et de comportements peuvent être observés.* »

décomposition et d'agrégation le postulat du multi-formalisme, nous obtenons alors la multi-modélisation. Pour être plus juste, il faut également ajouter tous les problèmes liés à la cohérence du couplage de formalismes par décomposition et le point de vue du *mapping*. L'approche de B. P. Zeigler est alors complémentaire puisqu'elle s'abstrait des processus de décomposition et d'agrégation et propose la spécification formelle de systèmes dynamiques et de systèmes couplés avec DEVS.

L'activité de modélisation, développée dans notre équipe [Ramat, 2003], s'inscrit dans une démarche où à chaque niveau d'abstractions ou chaque élément du multi-modèles, il faut adopter un paradigme et un formalisme conformes aux propriétés du système ou du sous-système. De plus, celui-ci doit être compatible formellement et sémantiquement avec les autres éléments du modèle. Nos travaux, dans cette thèse, s'inscrivent dans la continuité de cette pensée.

À partir de ces travaux [Ramat *et al.*, 1998], des collaborations se sont développées avec des laboratoire de biologie. Celles-ci ont fait naître une problématique majeure sur la formalisation des Systèmes Multi-Agents appliquée aux systèmes vivants et leurs couplages avec d'autres formalismes. Ces problèmes sont développés dans les travaux de R. Duboz [Duboz *et al.*, 2003] pour l'utilisation du formalisme DEVS et la définition d'une spécification SMA. La nécessité du développement d'un simulateur DEVS est ainsi survenue au regard des différentes plates-formes DEVS disponibles. Nous avons ainsi développé une nouvelle spécification d'un simulateur DEVS. Cette spécification s'attache à ajouter à DEVS différentes possibilités, dont les plus importantes sont le couplage de modèles, via les capsules ou *wrapper* et la simplification des comportements des modèles, en utilisant de nouvelles fonctions de transitions. Nous avons intégré également un ensemble de fonctions améliorant l'utilisation de DEVS en tant que simulateur, avec la gestion des entrées et des sorties des simulations ainsi que la définition et la distribution des expériences. L'ensemble de ces travaux est décrits dans le deuxième chapitre. À partir de ce nouveau simulateur et des ajouts réalisés, nous avons développé une nouvelle spécification des SMA. Cette spécification, orientée principalement sur les agents réactifs, utilise les aspects modulaires de DEVS et les concepts de J. C. Soulié des environnements multiples. Cette spécification est exposée au troisième chapitre. L'ensemble de ces travaux ont été développé dans une plate-forme de multi-modélisation nommée VLE, *Virtual Laboratory Environment*, dont la description est développée dans le quatrième chapitre. Quelques exemples d'utilisation de VLE sont proposés dans le cinquième chapitre.

Chapitre 2

Fonctionnement interne du simulateur

Sommaire

Dans le cadre de cette thèse, nous avons développé une plate-forme de modélisation et de simulation nommée VLE, *Virtual Laboratory Environment*, dont les principales caractéristiques sont de répondre à un besoin d'un *framework* ou cadriciel[1] opérationnel de modélisation et de simulation. Cette plate-forme se base sur le formalisme DEVS, les simulateurs abstraits définis par B. P. Zeigler [Zeigler *et al.*, 2000] et ses extensions pour fournir les outils nécessaires à la réalisation du cycle de développement de modèles, c'est-à-dire, la modélisation, l'expérimentation via la notion de plans d'expériences, ainsi que l'analyse de résultats obtenus de la simulation. Pour accomplir ce travail nous proposons d'ajouter et de modifier certaines parties du formalisme DEVS.

Dans ce chapitre, nous abordons en première partie, le fonctionnement interne du noyau de simulation de VLE et de sa principale caractéristique, la mise à plat de la hiérarchie de modèles. La deuxième partie introduit la notion de plans d'expériences avec en particulier, l'observation où nous développons une méthode formelle de capture de l'état des modèles sans perturbation. Nous exposons également une généralisation de l'initialisation des modèles atomiques via l'utilisation d'événements d'initialisation. La troisième partie se sert d'un exemple de modèle d'économie, pour traiter un problème de gestion du temps dans les modèles DEVS classiques. Cette partie permet en outre de présenter une méthode de simplification des modèles DEVS. Enfin, en dernier point, nous abordons une extension dont le rôle est d'accroître les possibilités de modélisation du formalisme DEVS, nous montrons son rôle dans l'optimisation des simulations en terme de temps de simulation.

2.1 Mise à plat de la hiérarchie de modèles

B. P. Zeigler définit le formalisme DEVS [Zeigler, 1976] comme une hiérarchie de composants, où les modèles couplés forment les nœuds, les modèles atomiques les feuilles. À chaque modèle couplé est associé un coordinateur dont le rôle est de gérer les événements attachés aux modèles qu'il contient. Si un modèle interne est connecté à un port de sortie du modèle couplé, les messages sont envoyés au coordinateur parent du modèle couplé. Ainsi, les messages, entre deux sous-arbres de modèles, remontent dans la hiérarchie jusqu'à atteindre le premier modèle couplé commun entre les deux hiérarchies et redescendent ensuite, dans la hiérarchie de modèles destinataires du message. Le coût engendré par ce routage de l'événement est important et est dû principalement, à la duplication des événements entre les coordinateurs. Ce coût dépend du nombre de coordinateur à traverser pour atteindre la destination. Dans cette section, nous introduisons le concept de mise à plat de la hiérarchie de modèles où seulement un coordinateur gère les événements de toute la hiérarchie de composants évitant ainsi le routage des événements avec un surcoût de gestion d'événements. Nous développons ce point dans la première partie de cette section.

[1] Un cadriciel est un ensemble de bibliothèques permettant le développement d'applications. Dans le reste du manuscrit nous emploierons également le terme plate-forme logicielle pour désigner un cadriciel.

La mise à plat de la hiérarchie de modèles apporte des avantages, principalement en terme de vitesse et de simplification des algorithmiques des simulateurs abstraits. Cependant, des problèmes surviennent lorsque l'on souhaite utiliser les techniques de parallélisation et de distribution des calculs de la simulation, où les algorithmes de B. P. Zeigler ne peuvent être utilisés en l'état. Nous présentons, dans les deuxième et troisième parties, une introduction à la parallélisation et à la simulation ainsi que des méthodes pour les utiliser dans un environnement à un seul coordinateur. Dans la dernière section, nous développons le fonctionnement interne de l'unique échéancier dans un système de mise à plat de la hiérarchie de modèles DEVS. Celui-ci doit, en effet, contenir les événements de tous les modèles atomiques de la hiérarchie à la différence du DEVS classique où chaque coordinateur possède un échéancier local et le coordinateur racine, l'échéancier global.

À partir de ces travaux sur la théorie de la modélisation et de la simulation ainsi que sur DEVS, B. P. Zeigler propose un ensemble d'algorithmes afin de fournir un passage du formalisme vers une version opérationnelle [Zeigler *et al.*, 2000]. Ces algorithmes sont nommés « simulateurs abstraits » et sont disponibles pour DEVS et ses variantes comme DS-DEVS ou DEVS parallèle [Chow et Barros, 1994]. Dans la suite de ce manuscrit, nous développons les fonctionnalités sous les formes algébriques ou algorithmiques afin d'améliorer la lisibilité des fonctions étudiées.

2.1.1 Comparatif des plates-formes DEVS

Dans cette partie, nous développons un comparatif entre les principales plates-formes de multi-modélisation basées sur le formalisme DEVS et ses extensions telle que DEVS parallèle que nous décrirons dans la partie 2.3. Nous explorons les principales différences entre les plates-formes et décrivons les avantages et inconvénients. Les projets utilisant DEVS comme base de simulation et modélisation sont nombreux, mais très peu ont atteint un stade utilisable. Nous nous focaliserons donc sur les plates-formes récentes ou encore développées ainsi que celles ayant une grande activité. Ce comparatif se base sur les plates-fomes DEVS déclarées auprès de G. Wainer [Wainer, 2006] ainsi que sur les codes sources des API, lorsqu'ils sont disponibles.

2.1.1.1 Adevs

Adevs, *A Discrete EVent System simulator*, développé par J. Nutaro [Nutaro, 2003] à l'Université d'Arizona, d'après sa documentation officielle, est une bibliothèque pour la construction de simulation basée sur DEVS parallèle et dynDEVS, une extension pour le support de la gestion de structures dynamiques de modèles basée sur le *Dynamic DEVS* de A. M. Uhrmacher [Uhrmacher, 2001]. Cette bibliothèque requiert aujourd'hui des compétences dans DEVS et dans le formalisme DEVS, les seules méthodes de développement de nouveaux modèles.

2.1.1.2 CD++

Cet ensemble de bibliothèques est développé en 2002 par G. Wainer [Wainer, 2002] et les Universités de Carleton, au Canada et Buenos Aires, en Argentine. Elle permet la définition de modèles DEVS et Cell-DEVS, en utilisant un langage de spécification de haut niveau sous la forme de manipulation de graphes d'états. Des versions introduisent les extensions DEVS parallèle et DEVS temps réel [Glinsky et Wainer, 2002]. De plus, les modèles peuvent se baser sur les formalismes de type réseaux de Petri, les automates cellulaires, ou les graphes d'états DEVS. Le programme DEVSView, développé récemment, permet aux utilisateurs de créer des visualisations à partir des sorties de CD++ avec des rendus des états et des événements sous forme d'animation.

2.1.1.3 DEVS/C++

Cette bibliothèque est l'une des premières plates-formes basée sur le formalisme DEVS. Elle a pour but d'être une implémentation efficace des simulateurs abstraits en proposant différents mécanismes d'optimisation [Zeigler *et al.*, 1996]. Elle est développée à l'Université d'Arizona par Hyup. J. Cho et Young K. Cho avec la possibilité d'utiliser les extensions de structures dynamiques et d'automates cellulaires en laissant la possibilité d'être distribuable sur plusieurs calculateurs. La dernière version de DEVS/C++ date de 2002 et semble, ainsi, être un projet stoppé.

2.1.1.4 DEVS/Java

DEVS/Java est développé à l'Université de l'Arizona [Sarjoughian et Zeigler, 1998], dont la dernière version date de janvier 2004 à la date d'écriture de ce manuscrit, est une plate-forme de modélisation et de simulation incluant l'extension de modification de la structure des modèles en ajoutant ou supprimant des modèles, couplés ou non, ainsi que les ports. Deux autres extensions sont également incluses comme l'intégration de systèmes d'équations différentielles et des automates cellulaires. Le développement de modèle permet le développement collaboratif par composition de manière synchrone ou asynchrone. Cette plate-forme propose également le support de la norme High Level Architecture, HLA, pour la distribution de simulation [Zeigler *et al.*, 1999].

2.1.1.5 JDEVS

Ce logiciel, développé par le laboratoire Systèmes Physiques de l'Environnement de l'Université de Corse principalement, par J. B. Filipi [Filippi et Bisgambiglia, 2003], est un travail de thèse sur le développement d'une plate-forme sur un environnement de modélisation et de simulation de systèmes naturels complexes. Il propose des solutions pour la communication avec des données fournies par des composants externes comme

les systèmes d'informations géographiques. Les modèles intégrés et proposés par la plate-forme sont les automates cellulaires et les réseaux de neurones [Filippi *et al.*, 2002].

2.1.1.6 Atom3

Atom3, *A TOol for Multi-formalism and Meta-Modeling*, est une plate-forme de modélisation et de simulation développée par Juan De Lara à l'Université Autonome de Madrid, UAM et Hans Vangheluwe à l'Université Mc. Gill au Canada, au laboratoire Modelling, Simulation and Designa. Dans Atom3, les modèles et les formalismes, sont décrits sous forme de graphes [de Lara et Vangheluwe, 2002]. À partir d'une description des formalismes, Atom3 propose des outils de manipulation des modèles décrits dans ces formalismes. Les méta-modèles fournis sont, actuellement, les réseaux de Petri, les automates à états finis et les diagrammes de flux. Les modèles peuvent alors être traduits vers le formalisme DEVS automatiquement par des routines de traduction. DEVS est alors utilisé comme formalisme unificateur en transformant les modèles sous forme DEVS. Le simulateur DEVS attaché se nomme PyDEVS.

2.1.1.7 Conclusion

La principale conclusion que nous pouvons formuler pour cette étude de comparaison de simulateurs et de plates-formes est l'impossibilité de la reprise d'un projet récent en début de cette thèse, due principalement aux choix réalisés en terme de licences des logiciels mises à disposition, ainsi que la spécificité des plates-formes pour des domaines particuliers. On remarque aussi que le développement de modèles requiert souvent des connaissances dans le langage de programmation de la plate-forme, la seule méthode pour développer des modèles. C'est un choix peu réaliste en terme d'ouverture vers d'autres utilisateurs et rend la plate-forme trop dépendante des connaissances des outils de développement.

Après cette étude, nous avons considéré le développement d'une nouvelle plate-forme de modélisation et de simulation basée sur DEVS dont les principales spécificités sont les suivantes :

– être libre : en utilisant une licence permettant l'utilisation, l'étude, la modification et la redistribution du code et des spécifications de la plate-forme, afin de laisser toute personne la liberté d'utiliser le cadriciel ;
– être utilisable par les non-informaticiens : en proposant des outils de haut niveau pour la conception de modèles et en proposant différentes possibilités d'interventions dans la plate-forme ;
– améliorer les capacités des simulateurs DEVS : en proposant des techniques d'optimisations et d'améliorations des capacités de DEVS en tant que plate-forme, c'est-à-dire, en proposant une meilleure malléabilité ;

– proposer les principales extensions de DEVS : en proposant, comme la plupart des plates-formes, les automates cellulaires et l'accès aux modifications des graphes de modèle en cours de simulation.

2.1.2 Principe de fonctionnement

Dans les systèmes modélisés avec une hiérarchie de modèles importante, où un grand nombre de messages ou d'événements circulent, le coût engendré par la gestion des messages entre coordinateurs est très important puisqu'il est équivalent au parcours d'un arbre et est directement dépendant de la hauteur à parcourir dans la hiérarchie. Nous avons pris en compte cette problématique pour la réalisation d'un simulateur efficace lorsque la taille de la hiérarchie est importante.

La solution que nous avons adoptée est une méthode de mise à plat de la hiérarchie de modèles, où seul subsiste le coordinateur racine, un coordinateur ou un simulateur, suivant l'existence d'une hiérarchie de modèles ou d'un seul modèle atomique. Tous les modèles couplés, contenus dans le premier modèle couplé, sont alors supprimés de la hiérarchie. Une représentation graphique de cette mise à plat est fournie sur la figure 2.1. Cette figure montre sur sa partie gauche, figure proposée par B. P. Zeigler [Zeigler, 1976], la hiérarchie de modèles DEVS et sa traduction en simulateurs au centre. Sur la partie droite, nous représentons la mise à plat de la hiérarchie de modèles.

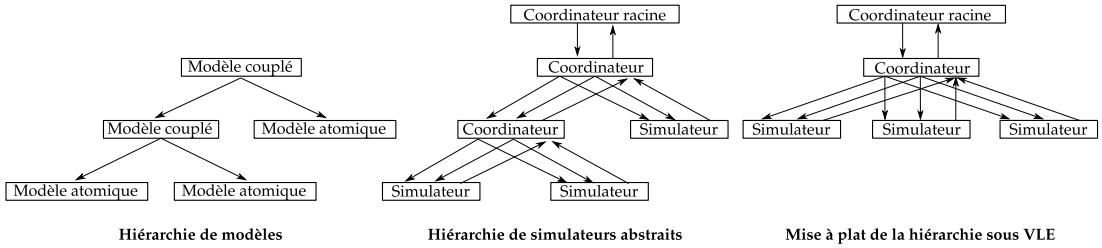


FIG. 2.1 – Réorganisation de la hiérarchie de modèles.

Notre méthode de mise à plat du graphe simplifie la hiérarchie de graphe en supprimant tous les modèles couplés de la hiérarchie de modèles, à l'exception du premier. Comme pour les simulateurs abstraits, un coordinateur est attaché au modèle couplé et à chaque modèle atomique est attaché un simulateur. Si le modèle ne possède qu'un seul modèle atomique, notre simulateur fonctionne alors de la même manière que les simulateurs abstraits, le coordinateur racine, étant connecté à un simulateur qui traite les événements du modèle atomique.

La formalisation DEVS n'est pas modifiée par cette mise à plat des modèles puisque nous utilisons les mêmes composants. Les simulateurs et le coordinateur racine sont identiques à ceux définis par B. P. Zeigler. Le comportement du coordinateur d'un mo-

dèle couplé reste le même dans notre mise à plat. La différence réside dans le nombre de coordinateurs puisqu'un seul est conservé. Les autres sont simplement supprimés et les modèles atomiques qu'ils manipulaient sont accrochés à l'unique coordinateur. Les connexions entre modèles couplés sont simplifiées pour connecter directement les modèles atomiques entre eux.

Cette technique de mise à plat de la hiérarchie de modèles n'est pas appliquée à la modélisation d'un système mais à sa simulation. Ainsi le modélisateur conserve tous les avantages de DEVS, la modularité et la décomposition hiérarchique principalement. À l'initialisation de la simulation, la hiérarchie est simplement « oubliée » par les simulateurs, afin de fournir de meilleures performances. Pour atteindre un meilleur niveau d'efficacité, des algorithmes de routages statiques peuvent être appliqués lors de la connexion des modèles atomiques entre eux.

Dans les paragraphes précédents, nous avons expliqué le fonctionnement de la mise à plat de la hiérarchie. Cette technique a pour principal avantage de conserver les algorithmes des simulateurs abstraits fournis par B. P. Zeigler, tout en apportant une économie d'envois de messages entre les hiérarchies de coordinateurs. Cependant, cette technique apporte un problème lors de la parallélisation et la distribution de la simulation puisque ces concepts ont été prévus pour une hiérarchie de modèles et donc plusieurs coordinateurs et un coordinateur racine gèrent la simulation complète. Ces techniques permettent d'améliorer la vitesse de simulation, d'augmenter la taille des modèles et d'exploiter des ressources informatiques plus importantes. B. P. Zeigler aborde ces techniques dans le chapitre 11 de son livre [Zeigler *et al.*, 2000]. Les deux sections suivantes montrent l'intégration du concept de mise à plat de la hiérarchie de modèles dans la parallélisation et la distribution de simulation.

2.1.3 Parallélisation des calculs

Les simulations ont des besoins en ressources informatiques de plus en plus importantes en augmentant la précision des modèles et les durées des simulations. Le rôle de la parallélisation de calculs est de diminuer le temps de simulation en parallélisant les calculs sur plusieurs processus. Les plates-formes, répondant à cette demande, migrent de systèmes mono-processeur aux systèmes multi-processeurs. Les architectures multi-processeurs consistent en l'utilisation de plusieurs processeurs interconnectés via des réseaux de communication interne ou externe. Dans cette partie, nous discuterons sur la gestion de la parallélisation dans les événements discrets et plus particulièrement dans le formalisme DEVS.

Les systèmes physiques obéissent toujours au *principe de la causalité*. Il peut être énoncé de la façon suivante : le futur n'influence jamais le passé. La causalité impose donc un ordre partiel des transitions d'états dans les systèmes physiques. L. Lamport a défini [Lamport, 1978] une méthode basée sur l'utilisation d'estampilles pour permettre aux processus de respecter la causalité. Cet ordre partiel imposé par les systèmes phy-

siques induit un ordre partiel entre les événements du modèle de simulation. En particulier, l'ordre dans lequel le simulateur produit les événements doit être cohérent avec cet ordre partiel, pour que le simulateur soit un reflet juste du système physique simulé. Par exemple, si la transition A du système physique intervient à la date 3 et a une influence sur la transition B intervenant à la date 5, le simulateur doit générer l'événement modélisant la transition A avant celui modélisant la transition B.

La parallélisation de la simulation sur un système distribué pose alors un problème pour le maintien de ce principe de causalité. Les problèmes sont en plus accentués lorsque les événements sont distribués sur différents processus. En effet, l'échéancier est distribué et chacun des processus évalue localement chaque événement. Il faut alors utiliser différentes approches pour assurer, qu'à tout instant, le principe de la causalité est assuré.

2.1.3.1 La parallélisation de calculs

Pour présenter les techniques de parallélisation de simulation basées sur la technique de l'événement discret, nous exposons un exemple étudié dans un travail de DEA [Quesnel, 2004], où nous avons été amenés à spécifier et développer un simulateur à événements discrets pour des simulations dotées d'un nombre de modèles conséquents. Ces modèles ont la particularité d'être très communiquant et d'avoir un comportement principalement réactif.

Le thème de cet exemple est la modélisation d'un grand nombre de particules, représentées par des points placés dans un cube. Celui-ci est découpé en plusieurs parallélépipèdes, chacun étant simulé par un processeur différent. Les entités du système sont positionnées, en suivant une politique de placement, dans le cube et se déplacent en utilisant une marche aléatoire[2] [Rudnick et Gaspari, 2004]. De ces déplacements résultent des franchissements des frontières des zones gérées par les processeurs, c'est ce que l'on nomme des points de synchronisation. Il faut, en effet, que la particule arrive dans la zone destinataire sans violer la règle de causalité dans celle-ci, c'est-à-dire, sans arriver dans le passé.

Ce simulateur, basé sur le formalisme DEVS, propose trois types de synchronisation de coordinateur. Le premier est la distribution de calcul, où aucun calcul n'est effectué en parallèle. Ce test sert principalement de référence de temps de simulation. Le deuxième, une méthode dite pessimiste, parallélise le calcul sans causer de problème de causalité. Enfin, la méthode optimiste parallélise les calculs en effectuant des retours dans le temps de simulation lorsqu'un problème de causalité apparaît.

– Approche synchrone forte : un simulateur peut traiter tout événement qui n'a pas de conséquence sur les simulateurs jusqu'à une certaine date appelée barrière de

[2]Une marche aléatoire consiste à réaliser un déplacement d'une entité en suivant une même loi uniforme pour chaque axe disponible.

synchronisation. Dans notre cas, nous sommes obligés de considérer que tout événement peut avoir une conséquence d'où l'algorithme adopté, celui de B. P. Zeigler où un processus central gère une horloge globale. Ce processus est appelé coordinateur et donne la main au simulateur qui possède l'événement dont la date d'occurrence est la plus petite globalement. Dans cette approche, il n'existe aucun parallélisme ;
- Approche asynchrone faible : chaque processus possède un échéancier qui fournit la date minimale (date jusqu'à laquelle un simulateur peut traiter les événements sans conséquence sur ses voisins). Celle-ci est envoyée à tous les simulateurs. Si toutes les dates reçues sont supérieures à sa date minimale, le simulateur traite son prochain événement et envoie à tous ses voisins sa date minimale. Le simulateur ne pourra recommencer à dépiler son échéancier seulement quand il aura reçu des dates supérieures à son temps courant des autres processus ;
- Approche optimiste : chaque processus décide de faire avancer la simulation, sans se préoccuper de l'état des autres. Dans ce cas, le principe de la causalité peut être corrompu. Pour préserver ce principe, Jefferson [Jefferson, 1985] propose une solution : le *Time Warp*. Lorsqu'un processus s'aperçoit que le principe de la causalité n'est plus vérifié, c'est à dire que la date courante de la simulation est postérieure à la date du dernier événement reçu, il annule tous les messages déjà expédiés. Pour cela, il faut gérer un historique des états du système, mais aussi le contenu de tous les messages reçus et expédiés, afin de pouvoir avertir les autres processus de faire des annulations. Pour cela, nous utilisons des *anti-messages*, qui possèdent exactement les mêmes caractéristiques que les messages d'origine à l'exception d'un drapeau. Ces messages lorsqu'ils sont reçus par un simulateur, demandent à celui-ci de supprimer le message normal, s'il est encore présent dans la liste de messages, sinon, il réalise un retour dans le temps, ou *Rollback*, jusqu'à la date d'envoi de ce message.

Les résultats de cette étude [Quesnel, 2004] montrent qu'il est très difficile d'améliorer la vitesse de simulation dans une approche optimiste où le nombre d'échanges d'information est très important. Le système effectuant alors un nombre conséquent de retours dans le temps pour corriger les erreurs. La méthode pessimiste a affiché de meilleurs résultats avec un gain de temps de simulation à partir du moment où chaque particule devait effectuer un calcul à chaque transition d'état.

Dans le paragraphe suivant nous développons l'introduction de la parallélisation dans le formalisme DEVS et nous exposons son intégration dans le cadre de notre simulateur où la hiérarchie de modèles est mise à plat.

2.1.3.2 Parallélisation et DEVS

B. P. Zeigler fournit des algorithmes de simulateur pour réaliser la parallélisation de code sous DEVS. Cependant, la mise en place d'une parallélisation dépend fortement du type de simulation utilisée. En effet, dans une simulation où des modèles calculent leurs nouveaux états indépendamment des autres modèles, la parallélisation peut consis-

ter à gérer les modèles indépendants sur des processeurs différents. La figure 2.2 montre un exemple en DEVS d'un tel système où deux modèles couplés, possédant chacun trois modèles, sans aucune connexion entre eux, peuvent facilement se paralléliser via l'utilisation d'un processeur par modèle couplé. Toutefois, des modèles complètement indépendants ne sont pas une règle courante.

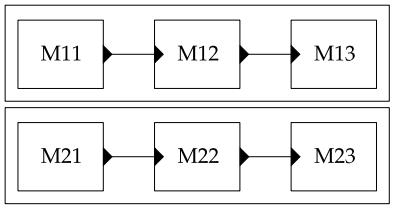


FIG. 2.2 – Cette figure montre deux modèles couplés indépendants. Chacun de ces modèles couplés possèdent trois modèles. Une parallélisation des calculs peut être effectuée en attachant les modèles couplés sur des processeurs différents.

La dépendance dans les simulations à événements discrets est posée par les événements eux-mêmes. Si un événement en sortie d'un modèle vient perturber un autre modèle, celui-ci ne doit pas se trouver à une date supérieure à la réception de l'événement, c'est le principe de causalité.

La figure 2.3 montre un exemple de problème engendré par la parallélisation. Si les modèles couplés $M1$ et $M2$ sont exécutés sur des processeurs différents, les événements externes posés sur le port d'entrée du modèle $M3$ doivent être synchronisés pour éviter au modèle $M3$ de recevoir un événement externe daté à une date inférieur.

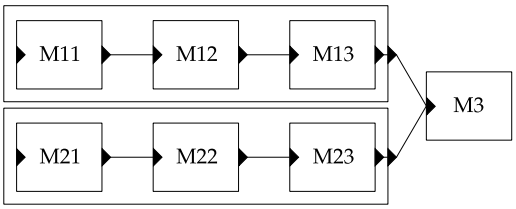


FIG. 2.3 – Ce graphique montre, par rapport à la figure précédente 2.2, un point de synchronisation engendré par les événements arrivant sur le port d'entrée du modèle $M3$.

B. P. Zeigler propose [Zeigler, 1984] des algorithmes de simulateurs abstraits pour s'assurer de la causalité dans la parallélisation de modèles DEVS. Les méthodes employées sont multiples et sont résumées par :

– Optimiste ou *Time Warp* : autorise la violation de la causalité et corrige les erreurs via l'utilisation de retours dans le temps. Cette méthode est difficile à mettre en œuvre ;
– Pessimiste ou sans risque : retient la sortie d'événement d'un modèle à l'autre tant qu'il crée un problème de causalité. C'est une méthode relativement facile à utiliser ;

– DEVS parallèle : synchronise les modèles suivant une date minimale pendant laquelle aucune sortie de modèles ne peut engendrer un problème de causalité. C'est une méthode sans risque.

Les méthodes précédemment évoquées ne peuvent être employées directement dans notre simulateur où la hiérarchie de modèles est mise à plat et où un seul coordinateur existe. Nous proposons, cependant, une solution à ce problème en modifiant la technique de mise à plat :

– Le coordinateur racine est autorisé à gérer plusieurs coordinateurs ou simulateurs via la gestion d'un échéancier contenant les dates minimales de chaque composant ;
– La hiérarchie de modèles doit être traduite en hiérarchie de modèles à un seul niveau, avec ainsi, autant de coordinateurs ou de simulateurs attachés au coordinateur racine que de modèles couplés ou de modèles atomiques. La séparation entre chaque modèle couplé représente un point de synchronisation.

L'utilisation de cette technique nous permet l'emploi de méthodes optimistes ou pessimistes en utilisant au maximum un processeur par coordinateur. Le choix du découpage de la hiérarchie est alors un point très important, puisqu'il va décider de l'utilisation des processeurs disponibles. De plus, la synchronisation des modèles passe par le coordinateur racine, celui-ci doit donc avoir une représentation des connexions entre chaque coordinateur, entre chaque modèle couplé.

D'un point de vue formalisation, seul le coordinateur racine est modifié par rapport à DEVS classique. Les modifications sont nombreuses puisque celui prend alors le même comportement que le coordinateur DEVS. Il gère les connexions des modèles couplés entre eux et possède l'échéancier global, sa définition est la suivante :

$$DEV_{N_{racine}} = \langle D, M_d, IC \rangle$$

où : $d \in D$:

$$\begin{cases} d \in D, \text{ avec } d \text{ un nom de modèle et } D \text{ l'ensemble des modèles,} \\ M_d \text{ est un modèle DEVS géré sur un processeur,} \\ IC \text{ les connexions internes entre les coordinateurs et simulateurs.} \end{cases}$$

2.1.4 La simulation distribuée

La simulation distribuée est une technique permettant d'exploiter des ressources informatiques géographiquement dispersées. B. P. Zeigler propose d'utiliser HLA, *High Level Architecture*, une norme logicielle composée d'un *framework*, d'une architecture et d'une interface de spécification de systèmes répartis. Développée sous le contrôle du Bureau pour la Modélisation et la Simulation de la Défense américaine, DMSO (*Defense*

Modelling and Simulation Office), HLA définit une spécification pour la création d'une simulation globale composée de simulations distribuées interagissant. Son rôle était, à sa création, l'intéropérabilité et la réutilisabilité pour les simulations développées au sein du Département de la Défense américain, DOD (*Department Of Defense*). Aujourd'hui, elle est adoptée par l'OMG (*Object Management Group*) et approuvée par IEEE (*Institute of Electrical and Electronical Engineers*) sous la dénomination, IEEE 1516. HLA propose une terminologie où un fédéré, *federate*, est une simulation compatible HLA, une fédération, est un ensemble de fédérés interopérants, utilisant un RTI (*RunTime Infrastructure*), pour supporter les opérations d'exécution et de coordination des simulateurs.

L'intéropérabilité de HLA se base non pas sur les langages de programmation, mais sur les simulateurs qui sont alors des boîtes noires où seule la fonction d'avancement du temps est disponible pour HLA. Le comportement des modèles sous-jacents est complètement écarté du fonctionnement du cadriciel HLA. HLA est donc une bonne initiative pour les couches externes de simulateur, c'est-à-dire, pour l'intéropérabilité et le contrôle des données via des normes IEEE. Notre approche, dans la définition de notre simulateur, est compatible avec HLA. Nous utilisons un bus logiciel pour la gestion des événements basé sur DEVS-Bus introduit par J. Y. Kim en 1998 [Kim et Kim, 1998]. Sur ce bus est connecté l'ensemble des différents simulateurs et un coordinateur qui gère le déroulement de la simulation et du temps. La distribution des comportements est alors prise en compte par ce bus et par le modèle lui-même.

Nous utilisons une méthode de distribution de la simulation basée sur une connexion réseau entre le simulateur et le modèle du développeur. Une couche supplémentaire est ajoutée afin de réaliser cette communication réseau entre le simulateur connecté au coordinateur et le simulateur connecté au modèle. La communication entre ces deux parties est alors synchrone pour s'assurer de la continuité. Nous nommerons respectivement ces deux parties :

– simulateur-distribué-serveur : la partie entre le coordinateur et le simulateur ;
– simulateur-distribué-client : la partie entre le simulateur et le modèle atomique.

La figure 2.4 montre une représentation graphique de l'éclatement du simulateur en utilisant une communication réseau, comme la couche *socket*[3].

Les deux algorithmes suivants montrent comment le découpage du simulateur est réalisé en prenant pour exemple la réception d'un message d'initialisation. La première partie concerne la version serveur.

```
1 simulateur-distribué-serveur
2 variables :
3   parent : le coordinateur
4   tl : la date du dernier événement
5   tn : la date du prochain événement
6   reseau : une connexion réseau vers le simulateur-distribué-client
```

[3]Le *socket*, ou prise en français, est apparu dans les systèmes UNIX. Un *socket* est un élément logiciel faisant une interface logicielle avec les services du système d'exploitation.

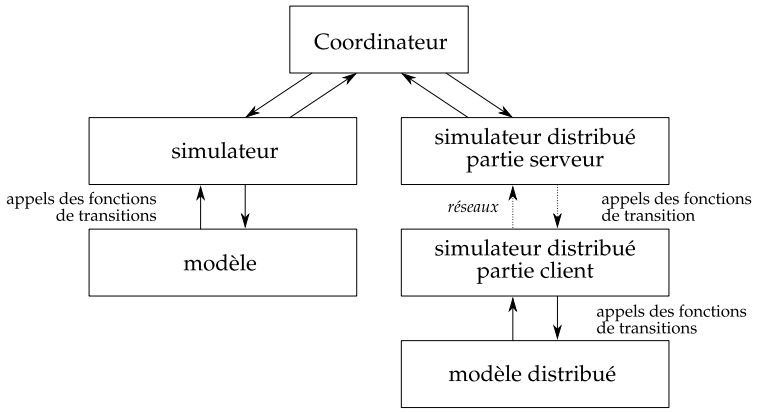


FIG. 2.4 – Cette figure représente deux modèles atomiques. Le premier est accédé directement via un simulateur DEVS. Le second utilise un simulateur distribué découpé en deux parties *simulateur-distribué-serveur* et *simulateur-distribué-client* pour s'échanger, les événements sur le réseau, via une communication par *socket*.

```
début :
  sur réception d'un i − message(i,t) au temps t
    tl = t - e
    envoie i − message(i,t) à reseau
    tn = tl + réponse de reseau
  fin sur
fin simulateur-distribué-serveur
```

La seconde partie, la version cliente :

```
simulateur-distribué-client
variables :
  parent : le coordinateur
  tl : la date du dernier événement
  tn : la date du prochain événement
  reseau : une connexion réseau vers le simulateur-distribué-serveur
début :
  sur réception d'un i − message(i,t) au temps t de reseau
    tl = t - e
    envoie ta(s) à reseau
    tn = tl + tl
  fin sur
fin simulateur-distribué-client
```

Le découpage du modèle atomique en deux parties *simulateur-client* et *simulateur-serveur* ajoute une couche supplémentaire lors de l'exécution des fonctions de transitions du modèle atomique. Cependant, cette couche trouve son utilité uniquement lors de la distribution de simulation. Nous introduisons un paramétrage particulier lors de l'initialisation des modèles pour éviter les coûts engendrés par l'utilisation de cette couche supplémentaire lors des simulations localisées sur le même processeur. Cette opération

est transparente pour le développeur de modèles puisque l'interface fonctionnelle des modèles reste la même en simulation distribuée ou locale.

L'ajout de cette couche supplémentaire dans notre spécification, c'est-à-dire, le découpage du simulateur de DEVS, apporte deux nouvelles possibilités :

1. l'utilisation du réseau comme médium de communication. Les appels de fonctions sont alors traduits sur le réseau via des technologies comme XML-RPC[4], MPI[5], ou directement via les interfaces réseaux bas niveau du système d'exploitation ;
2. l'hétérogénéité du langage de développement en utilisant des enveloppes sur des langages de programmation. La partie cliente du simulateur distribué est alors un objet de traduction vers d'autres langages. Les langages de programmations disponibles dans la plate-forme, au jour de l'écriture de ce manuscrit, pour le développement de modèles sont les langages *Python*, *C* et *Java*.

2.2 Proposition de simulateur

Dans les parties précédentes, nous avons présenté une solution pour améliorer la rapidité des modèles dans certains cas en utilisant la mise à plat de hiérarchie. Dans cette partie, nous exposons les ajouts réalisés aux modèles atomiques, principalement dans le découpage de celui-ci en deux parties distinctes. Nous complétons ensuite le formalisme DEVS avec la notion de plan d'expériences. Nous abordons les nouveaux concepts d'événements d'initialisations et de captures de l'état des modèles via les événements d'états dont nous formalisons l'écriture dans la deuxième et troisième partie.

2.2.1 Division du modèle atomique

Le formalisme DEVS repose sur un modèle atomique où sont définis les ports d'entrées, de sorties ainsi que les fonctions d'initialisation, de sortie, de transitions internes et externes. Le modèle atomique se compose ainsi des informations relatives à la hiérarchie de modèles DEVS et les informations de la dynamique du modèle.

Nous proposons une approche où le modèle atomique est divisé en deux parties distinctes. La première concerne le graphe, avec les ports d'entrées et de sorties, ainsi que les ports d'initialisations et d'états auxquels sont attachés les informations sur leurs

[4]XML-RPC : *XML Remote Procedure Call*, une spécification de codage des informations et des appels de fonctions à distance.

[5]MPI : *Message Passing Interface*, API de communication réseau sur un cluster.

noms et le type des données. La deuxième partie possède toutes les fonctions de transitions interne, externe, d'état ou d'initialisation. Les deux parties du modèle atomique sont nommées M_{graphe} et $M_{dynamique}$. Le modèle atomique M se caractérise par :

$$M = \langle M_{graphe}, M_{dynamique} \rangle$$

où M_{graphe} et $M_{dynamique}$ sont définis :

$$\begin{pmatrix} M_{graphe} : \langle X, Y \rangle \\ M_{dynamique} : \langle S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \end{pmatrix}$$

Le $M_{dynamique}$ possède une référence sur la partie graphe M_{graphe} afin de lui offrir la possibilité de connaître sa représentation dans la hiérarchie de modèle.

Ce découpage est introduit dans un but de simplifier les opérations de développement des modèles en introduisant une plus grande modularité lors de la conception de modèles en permettant la modification du graphe sans intervenir sur la partie dynamique. Cette méthode permet alors de créer des modèles atomiques possédant la même définition de $M_{dynamique}$ mais pour plusieurs M_{graphe} différents.

Le passage de la formalisation à l'implémentation est réalisé à l'aide d'un *design pattern*[6] nommé *bridge* [Gamma *et al.*, 1995]. Celui-ci consiste à découpler une abstraction de son implémentation. La mise en place de ce découpage ne modifie pas les algorithmes des simulateurs abstraits puisque les fonctions de transitions de $M_{dynamique}$ sont appelées directement depuis le simulateur.

Par la suite et dans un but de simplification de l'écriture des modèles atomiques, nous utiliserons la définition du modèle atomique décrite précédemment :

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

Le développement d'une plate-forme de simulation impose des choix techniques aux modélisateurs et analystes. Nous développons nos simulateurs avec comme principal intérêt d'ajouter des méthodes d'encapsulation au formalisme DEVS et d'augmenter les capacités de modélisation au modélisateur. Dans ce cadre de travail, nous ajoutons des éléments au formalisme pour nous permettre d'atteindre ce but. Les principales modifications touchent l'initialisation des modèles atomiques et la capture des sorties de simulation. Nous avons développé deux nouveaux types de ports : les ports d'initialisations et de captures d'états dont une représentation graphique est fournie sur la figure 2.5. Ces types de ports, accompagnés d'événements d'initialisations et d'états, font appel à deux nouvelles fonctions de transitions : δ_{init} et δ_{state}.

[6] En génie logiciel, un motif de conception ou *design pattern* est un concept destiné à résoudre les problèmes récurrents d'architecture ou de conception.

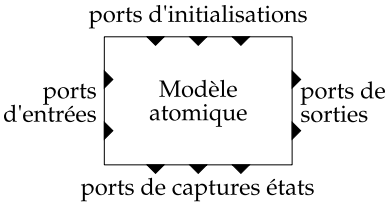


FIG. 2.5 – Cette figure montre une représentation graphique d'un modèle atomique avec les ports d'initialisations et d'états.

Dans les sections suivantes, nous proposons une formalisation de ces deux nouvelles fonctionnalités dans le formalisme DEVS.

2.2.2 Initialisations des modèles

Le formalisme DEVS propose une méthode d'initialisation des modèles atomiques via l'utilisation des $i - message(i, t)$ qui permettent d'initialiser les variables tl et tn des modèles atomiques, respectivement la date du dernier et du prochain événement. Le modélisateur peut inclure dans cette fonction, des méthodes d'initialisations de l'état de son modèle.

Toutefois, cette méthode s'avère inefficace dans le cadre de la réutilisation de modèles ou lorsque plusieurs modèles atomiques, possédant le même comportement, doivent être initialisés avec des données différentes. L'exemple typique est l'automate cellulaire où chaque cellule peut être initialisée avec une valeur différente. Dans ce cas de figure, le développeur doit changer le comportement de sa fonction d'initialisation pour chaque modèle atomique. Enfin, la gestion de l'initialisation des modèles via la fonction $i - message(i, t)$ n'est plus suffisante lors de l'utilisation de plans d'expériences automatisés, c'est-à-dire, lorsque le développeur souhaite réaliser plusieurs simulations avec les mêmes modèles mais des initialisations différentes et souvent des initialisations générées par d'autres modules.

Nous proposons dans cette section une méthode d'initialisation des états d'un modèle en intégrant un nouveau type d'événements au formalisme DEVS. La modification de la structure pour l'utilisation des événements d'initialisation est simple, puisqu'elle consiste en l'ajout d'une fonction de transition δ_{int} et du type d'événement d'initialisation. Notre définition du modèle atomique est la suivante :

$$M = \langle X, Y, X_{init}, S, \delta_{ext}, \delta_{init}, \delta_{int}, \lambda, ta \rangle$$

où :

$X_{init} : \{(p_{init}, v) | p_{init} \in InitPorts, v \in V_{X_{init}}\}$
les valeurs d'initialisation des variables d'états du modèles attachées aux ports,

$InitPorts$
est l'ensemble des ports d'initialisation relié aux variables d'états du modèle,

$\delta_{init} : S \rightarrow S'$ où $((\cdots, s'_i, \cdots)) = ((\cdots, v_i, \cdots))$
la fonction d'initialisation des états du modèle.

Les événements d'initialisation permettent aux modélisateurs d'initialiser les variables d'états de leurs modèles atomiques de manière dynamique. Les algorithmes précédents sont légèrement modifiés pour tenir compte de la primitive du traitement de ces événements. Le fonctionnement du simulateur est légèrement modifié :

```
simulateur
variables :
  parent : le coordinateur
  tl : la date du dernier événement
  tn : la date du prochain événement
début :
  sur réception d'un x_init − message(x_init, t) au temps t
    s_init = δ_init(S, t, x_init)
  fin sur
simulateur
```

La modification du coordinateur :

```
coordinateur
variables :
  parent : le coordinateur
  tl : la date du dernier événement
  tn : la date du prochain événement
début :
  sur réception d'un x_init − message(i, t) au temps t
    envoie x_init − message(i, t)
  fin sur
fin coordinateur
```

Le coordinateur racine :

```
coordinateur-racine
variables :
  t : la date courante
  t_max : le temps de la simulation
  fils : un simulateur ou un coordinateur
début :
  t = t_0
  envoie x_init − message(i, t) au fils
  envoie i − message(i, t) au fils
```

```
10   t = tn du fils
11   boucle
12     envoie (*,t) − message au fils
13     t = tn du fils
14   fin boucle quand t ≥ t_max
15 fin coordinateur-racine
```

2.2.3 Observations des modèles

La gestion des sorties d'une simulation est une partie très importante dans le cadre de l'utilisation d'une plate-forme de modélisation et de simulation complète. Cependant, le formalisme DEVS ne propose pas de méthode de capture d'information d'un modèle lors de la simulation. Les développeurs de plate-forme doivent alors faire des choix sur la récupération des informations au cours de la simulation.

- Le premier choix est un choix de modélisation, la plate-forme autorise- t-elle la perturbation de la simulation pour la récupération des informations, ou comment observer une simulation sans perturber le système ?
- Le second choix est d'ordre technique. Que faut-il récupérer comme information et comment la traiter : stockage ou envoi dans un flux d'information et comment fournir une méthode simple pour le faire ?

Les développeurs de plate-forme DEVS utilisent ainsi différentes techniques. De notre étude précédente en section 2.1.1, deux techniques sont développées :

- La première consiste à utiliser les extensions du langage de programmation pour capturer les états du système directement depuis le code source de modèle. Cette méthode, bien que très facile à mettre en œuvre, ne permet pas une gestion aisée des données recueillies de simulations de modèles importants et l'utilisation de méthodes génériques ne peut s'appliquer.
- La deuxième utilise le système d'événement DEVS pour capturer les états du modèle lors de l'arrivée d'un événement interne ou externe. Cette méthode pose toutefois un problème puisque le système sur la réception d'un événement externe doit calculer un nouvel état et donc, perturber le système.

Dans le cadre du développement de nos simulateurs, nous avons choisi d'utiliser cette deuxième technique en lui apportant des modifications importantes dont la principale est de fournir un mécanisme empêchant la modification de l'état du modèle observé et ainsi de ne pas perturber la simulation.

Pour réaliser cette gestion de la capture de l'état d'un modèle, nous introduisons un nouveau type d'événement nommé *événements d'états*. Le rôle de ces événements est de venir interroger un modèle sur un de ses ports d'états Y_{state} via l'appel à la fonction de transition δ_{state}. Cette fonction est suivie de l'appel à la fonction de sortie d'états, λ_{state} qui récupère les données attachées à l'état et les pose sur le port d'état.

La définition de notre structure d'un modèle atomique s'en trouve modifiée par l'ajout des ports Y_{state} et des fonctions δ_{state} et λ_{state} :

$$M = \langle X, Y, X_{init}, Y_{state}, S, \delta_{ext}, \delta_{int}, \delta_{init}, \delta_{state}, \lambda, \lambda_{state}, ta \rangle$$

où :

$Y_{state} : \{(p, v) | p \in StatePorts, v \in Y_{state}\}$
l'ensemble des ports d'états et des valeurs de sortie,

$\delta_{state} : S \longrightarrow S$
la fonction d'information de capture de l'états d'une variable,

$\lambda_{state} : S \longrightarrow Y_{state}$
la fonction de sortie, posant sur le port Y_{state} la valeur de la variable s.

Nous pouvons remarquer, dans le fonctionnement de ce système, que la fonction de transition δ_{state} ne modifie pas l'état du modèle, elle informe simplement celui-ci d'une prochaine capture d'événement et prépare la donnée qui sera fournie lors de l'appel de la fonction λ_{state}. Les événements sont gérés par une partie du simulateur nommé *observateur* dont la fonction est de créer les événements d'états pour le simulateur et de récupérer les valeurs posées par les modèles sur leurs ports d'états. Dans les implémentations effectuées, les deux fonctions, δ_{state} et λ_{state}, sont fusionnées afin d'alléger l'écriture du comportement des modèles atomiques.

Les observations de modèles sont de deux types, à pas de temps constants ou événementiels, et dépendent de l'observateur :

- La première permet la capture de l'état d'un modèle suivant un pas de temps discret Δ_t défini pour chaque observateur. L'échéancier est donc modifié pour prendre en compte ces nouveaux événements ;
- La deuxième permet de capturer l'état du système lors de l'arrivée d'un événement interne ou externe. Ce type d'événement est le moins coûteux en terme de gestion puisqu'il est déclenché uniquement sur la réception d'un événement interne ou externe.

La gestion de l'événement d'état est intégrée au simulateur :

```
1 Simulateur
2 variables :
3   parent : le coordinateur
4   tl : la date du dernier événement
5   tn : la date du prochain événement
6   observeur : fichier de sortie
7 début :
8   sur réception d'un i − message(i,t) au temps t
9     tl = t - e
```

```
10      tn = tl + ta(s)
11      si d observé par temps Δt
12        envoie y_state − message(i, t + Δt) à parent
13      fin si
14    fin sur
15
16    sur réception d'un y_state − message(y, t) au temps t
17      estate = t - tl
18      observeur = δ_state(s, estate, y)
19      si d observé par temps Δt
20        envoie y_state − message(y, t + Δt) à parent
21      fin si
22    fin sur
23  fin Simulateur
```

Les modifications du coordinateur :

```
1  Coordinateur
2  variables :
3    tl : la date du dernier événement.
4    tn : la date du prochain événement.
5    event − list : liste d'élément (d, tn_d) trié par tn_d
6    state − event − list : liste d'élément (d, tn_d) trié par tn_d
7  début :
8    sur réception d'un i − message(i, t) au temps t
9      pour chaque d ∈ D
10       envoie i − message(i, t) à d
11     fin pour
12     trie event − list suivant tn_d
13     tl = max {tl_d | d ∈ D}
14     tn = min {tn_d | d ∈ D}
15   fin sur
16
17   sur réception d'un ∗ − message(∗, t)
18     d∗ = premier(event − list)
19     si d observé événementiellement
20       envoie y_state − message(∗, t) à d∗
21     fin si
22     envoie ∗ − message(∗, t) à d∗
23     trie event − list suivant tn_d
24     tl = t
25     tn = min {tn_d | d ∈ D}
26   fin sur
27
28   sur réception d'un x − message(x, t)
29     destinataires = {r | r ∈ D, N inI_r, Z_{N,r}(x) ≠ ∅}
30     pour chaque r de destinataires
31       envoie x − message(x_r, t)
32       si r observé événementiellement
33         envoie y_state − message(x_r, t) à d∗
34       fin si
35     fin pour
36     trie event − list suivant tn_d
37     tl = t
```

```
38      tn = min {tn_d | d ∈ D}
39    fin sur
40
41    sur réception d'un y_state − message(y_d, t) au temps t du modèle d
42      d = premier(event − list)
43      envoie y_state − message(y_state, t) à d*
44      trie (state − event − list) suivant tn_d
45    fin sur
46  fin coordinateur
```

2.2.4 Échéancier

L'échéancier, lors du développement d'un simulateur à événements discrets, est la pièce centrale puisqu'elle a pour rôle de trier les événements par date de la manière la plus efficace possible. Les sections suivantes vont permettre de présenter le fonctionnement interne de l'échéancier lors de l'utilisation d'une mise à plat de la hiérarchie de modèles ainsi que les modifications apportées pour la gestion des événements d'état.

2.2.4.1 L'échéancier pour un unique coordinateur

La mise à plat de la hiérarchie de modèle impose un échéancier différent de ceux habituellement proposés dans la spécification DEVS. Dans le cas de notre spécification de simulateur, nous avons plusieurs éléments à traiter :

- Trier les événements internes suivant leurs dates d'occurrences et pouvoir les invalider lorsqu'un modèle disparaît ou qu'un événement externe perturbe le modèle avant sa date de prochain changement d'état ;
- Stocker le, ou les, événements externes posés sur les ports d'entrées des modèles.

Dans le cas du dépôt d'un événement externe sur l'un des ports d'un modèle, deux cas de figures s'opposent :

- Si l'événement externe est déposé à une date inférieure à l'occurrence de l'événement interne, celui-ci doit être détruit de l'échéancier puisque le modèle doit réagir à l'événement externe ;
- Si l'événement externe est déposé à la même date que l'occurrence de l'événement interne, ceux-ci vont entrer en conflit. Le modélisateur doit alors faire un choix entre l'événement interne ou l'événement externe via la fonction de conflit dite « *confluent function* » [7], δ_{conf}.

Cette première version de l'échéancier se compose de plusieurs objets. Une représentation graphique de l'échéancier est fournie sur la figure 2.6.

[7] B. P. Zeigler, dans [Zeigler *et al.*, 2000], utilise le terme *confluent transition function* que nous traduisons par fonction de conflit dans ce manuscrit.

– Une liste d'événements externes par modèle ;
– Une table des événements internes triée sur les dates. Le tri est effectué à l'aide d'une structure de type tas binaire[8] stockée dans un tableau unidimensionnel [Knuth, 1975] ;
– Un tableau associatif entre le modèle, un lien vers une table d'événements externes et un lien sur l'événement interne contenu dans la table d'événements.

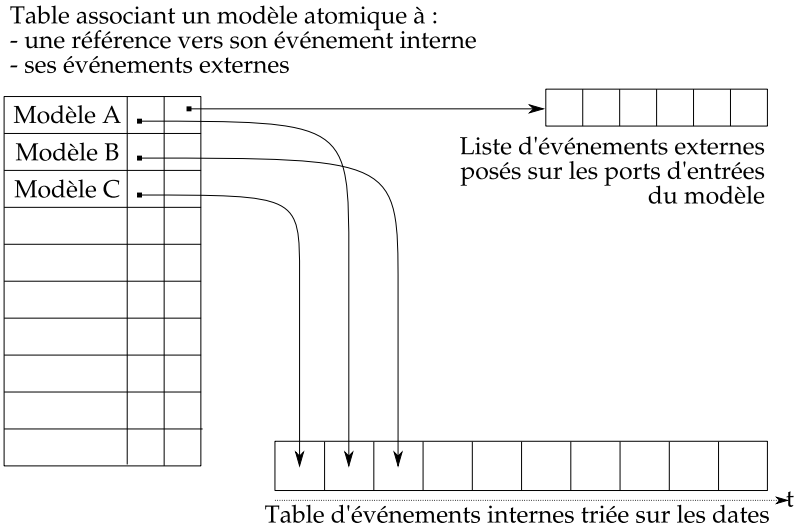


FIG. 2.6 – Représentation de première version de l'échéancier de notre proposition de simulateur.

2.2.4.2 Modification de l'échéancier

L'ajout des événements d'états requiert la modification de l'échéancier dans le coordinateur pour la gestion des événements d'états déclenchés sur un pas de temps constant. La figure 2.7 propose une représentation graphique du nouvel échéancier. Cette figure montre l'ajout d'un nouveau tableau trié. L'opération de dépilement d'un élément de l'échéancier consiste simplement à prendre l'élément le plus récent entre les tableaux d'événements internes et d'états.

Nous avons choisi de découper la table d'événements en deux parties distinctes suite au constat que les événements d'états sont souvent regroupés à la même date. En effet, si nous prenons l'exemple de l'observation à pas de temps discret Δ_t d'un automate cellulaire, lors de la capture de l'état de chaque cellule au temps t, tous les événements d'états de chaque cellule doivent être dépilés de l'échéancier et traités par les fonctions de captures δ_{state}. À la fin du traitement de chaque événement, un nouvel événement d'états doit être créé au temps $t + \Delta_t$ et stocké dans l'échéancier.

[8] Un tas binaire est une structure de données dont les caractéristiques sont d'être un arbre binaire complet et d'être ordonné à l'aide de clés : pour tous nœuds A et B tel que B soit un fils de A, clé$(A) \geq$ clé(B)

Le coût de l'enregistrement dans l'échéancier de chaque événement sera de l'ordre $O(log(n))$ ainsi que toutes les manipulations mémoires nécessaires pour la gestion du tas. Avec le système d'échéancier de listes d'événements d'états, le coût reste de l'ordre $O(log(n))$, mais il n'y a plus de manipulation mémoire, puisque les événements d'états possédant la même date d'occurrence sont stockés dans une liste d'événements d'états du tas.

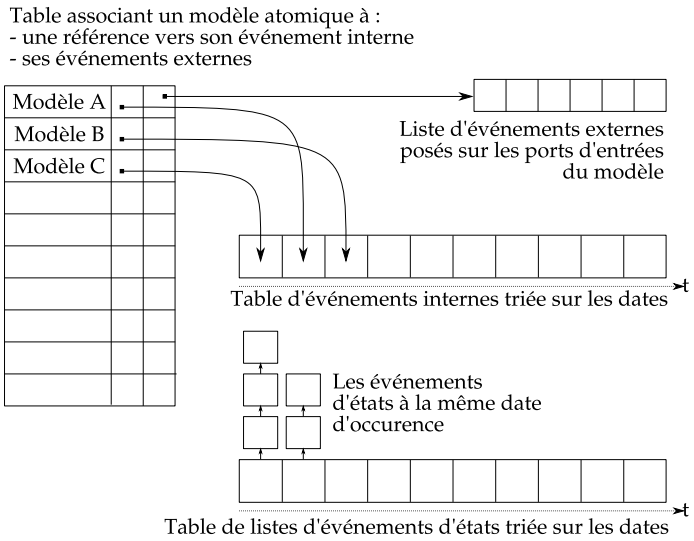


FIG. 2.7 – Ce graphique montre le fonctionnement de l'échéancier avec la gestion des événements d'états. Par rapport à la figure 2.6, on remarque l'ajout d'un nouveau tas pour la gestion des événements d'états. Ce tas est un tableau trié de tableaux d'événements pour optimiser les calculs d'ajouts et de suppressions dans l'échéancier.

2.3 Simulateur DEVS parallèle

Les deux premières sections ont permis de développer le fonctionnement interne de notre simulateur DEVS. Nous avons ajouté des événements d'initialisations et d'observations des états d'un modèle atomique lors de l'utilisation d'une mise à plat de la hiérarchie de modèles. Ces techniques permettent de faciliter l'utilisation du formalisme DEVS lors de la planification de plans d'expériences de systèmes complexes en simplifiant la gestion des simulations. Cependant, DEVS classique apporte un problème fondamental pouvant rompre la notion de causalité des modèles : les événements simultanés.

Pour introduire ce concept d'événements simultanés, nous développons un exemple particulier, où ce type de problème apparaît. Dans les deux premières parties, nous exposons ce problème et les solutions pour le résoudre. Enfin, en dernière partie, nous

fournissons une technique de simplification des modèles DEVS basée sur la correction des événements simultanés.

2.3.1 Présentation du problème des événements simultanés

Les événements simultanés peuvent créer des perturbations dans le formalisme DEVS classique et ne peuvent être résolus que par l'adjonction de fonctions dans les coordinateurs ou par l'utilisation de sacs d'événements. Ces concepts, et leur mise en place dans le simulateur, sont décrits dans les prochaines sections.

2.3.1.1 Le modèle d'économie de T. Schelling

Le modèle étudié pour l'illustration du problème des événements simultanés est le modèle développé par Thomas Schelling en 1971 dans son article « *Dynamic Models of Segregation* » [Schelling, 1971]. Cet article traite de la dynamique du partage de l'espace entre les « races ». Il a pour but de montrer que des structures résidentielles ségrégationnistes pouvaient apparaître, même si les préférences des habitants étaient compatibles avec une structure intégrée des populations.

Dans un quartier résidentiel, chaque habitant admet, ou souhaite, un voisinage différent de lui mais à partir d'un seuil, que nous noterons Δ, il décide de quitter le quartier. T. Schelling montre en appliquant la théorie des jeux que ce seuil peut modifier le quartier en deux situations stables : une ségrégation pure ou un mélange complet. Le résultat dépend alors de l'initialisation et du seuil de chaque habitant.

Le système est modélisé à l'aide d'un automate cellulaire où chaque cellule est un agent qui interagit avec ses voisins en utilisant un voisinage de Moore [Wolfram, 1986], c'est-à-dire, avec ses huit voisins immédiats. D'après le modèle de Schelling, les agents ont deux états :

– être inactif : tant qu'un agent voit son nombre de voisins inférieur au seuil Δ, il reste inactif et attend le départ ou l'arrivée d'un nouveau voisin pour calculer son état ;
– déménager : si un agent a un nombre de voisins supérieur au seuil Δ, il décide de déménager. Le déménagement se traduit par un déplacement aléatoire sur une cellule vide.

La figure 2.8 montre un exemple de simulation, à l'aide de VLE, de l'évolution du modèle d'économie de T. Schelling. Nous utilisons trois types d'habitants représentés graphiquement par les couleurs : *vertes*, *jaunes* et *bleues*. Les cases *noires* représentent des zones sans résident. L'environnement physique est modélisé par un automate cellulaire de 30×30 cellules. L'initialisation du modèle est réalisée à l'aide d'une loi uniforme et le seuil Δ est fixé à $\frac{1}{2}$. Les observations de la simulation montrent la formation de groupe de voisins de couleurs à partir d'un mélange complet.

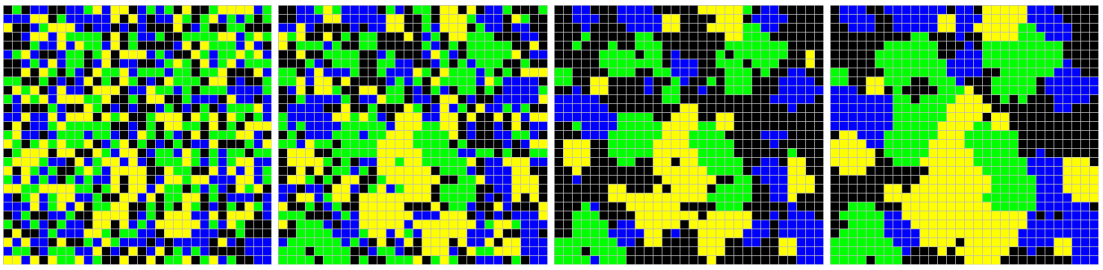


FIG. 2.8 – Représentation graphique de l'évolution d'une simulation du modèle d'économie de Schelling sur la plate-forme VLE.

2.3.1.2 Les événements simultanés

B. P. Zeigler expose le problème des événements simultanés dans [Zeigler *et al.*, 2000] en utilisant un modèle du jeu de la vie. Dans les paragraphes suivants, nous développons la même problématique en nous basant sur le modèle d'économie vu précédemment (cf. paragraphe 2.3.1.1).

Les événements simultanés surviennent lorsque plusieurs modèles réagissent à des événements internes ou externes au même instant t, c'est-à-dire, l'échéancier du coordinateur possède plusieurs événements à la date t. L'ordre dans lequel ils seront dépilés dépend alors entièrement de l'algorithme de tri de l'échéancier, *fifo*[9], *lifo*[10], aléatoire etc. Or, ce choix peut avoir des conséquences sur les actions des modèles lors du dépilement d'un nouvel événement.

Dans une traduction du modèle de T. Schelling, présenté dans le paragraphe 2.3.1.1, en simulation à événements discrets, le temps peut ne pas exister et l'avancée de la simulation dépend du déplacement des habitants. Les événements externes sont traduits par des informations envoyées par les habitants à leurs voisins pour leur indiquer un départ ou une arrivée. La figure 2.9 propose une représentation graphique du problème des événements simultanés à l'aide d'un automate cellulaire de 7×7 cellules habitées par des habitants *bleus* et *jaunes* et un seuil de tolérance Δ de $\frac{3}{4}$.

À l'initialisation du système, *temps 0*, deux habitants, positionnés aux coordonnées représentées par les couples $(-1, 0)$ et $(1, 0)$, vont quitter leurs cellules car leur seuil de voisins différents est dépassé. Comme les deux événements sont à la même date, les deux actions ont lieu au même instant, *temps 1*. Deux choix sont donc possibles suivant l'ordre dans lequel ils seront dépilés de l'échéancier. Le premier voit le déplacement de la cellule $(-1, 0)$ en $(0, 0)$, le deuxième de $(1, 0)$ vers $(0, 0)$ en raison de l'utilisation du même générateur aléatoire, la destination est la même. On remarque que le déplacement de la cellule en $(0, 0)$ bloque le départ de la deuxième cellule. Au *temps 2*, les cellules $(-2, -1)$ et $(2, -1)$ se sont déplacées à la coordonnée $(-1, -2)$. Le système est

[9] *fifo* : *first in first out*, le premier événement empilé est le premier à sortir.
[10] *lifo* : *last in first out*, le dernier événement empilé est le premier à sortir.

maintenant stable, les cellules ne peuvent plus avoir un nombre de voisins plus important que le seuil.

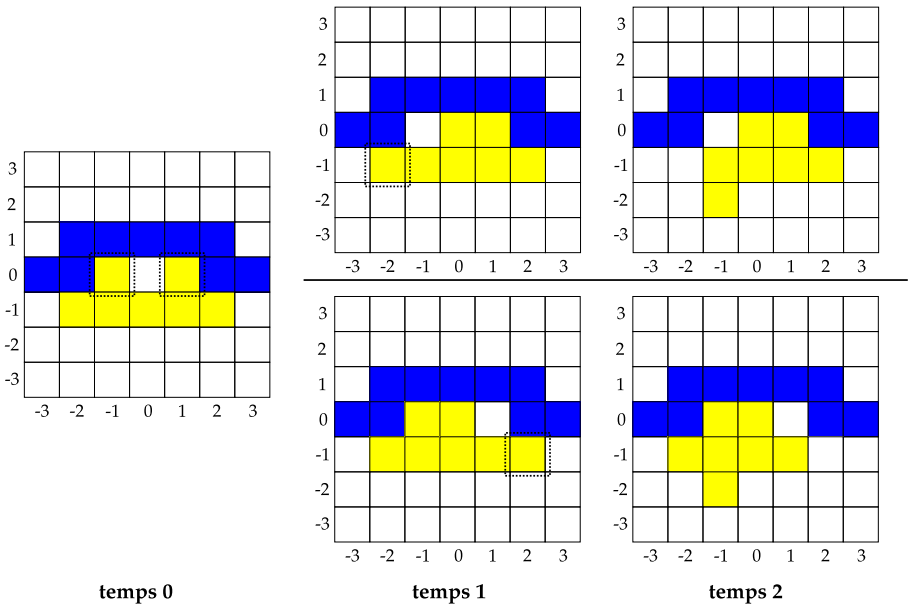


FIG. 2.9 – Représentation de l'évolution du modèle de T. Schelling dans un environnement à événements discrets. Au temps $t = 0$, deux actions différentes peuvent être prises d'où vont résulter deux états finaux différents.

Le système se trouve dans deux configurations complètement différentes à la fin de la simulation. Le problème réside au *temps 0*, lorsque les cellules ont décidé de se déplacer. La cellule qui se déplaçait en premier bloquait l'autre lors du dépilement suivant. Ce problème est nommé « événements simultanés » par B. P. Zeigler [Zeigler *et al.*, 2000].

Le problème des événements simultanés peut être corrigé de plusieurs manières différentes. La première consiste à introduire une notion de temps au modèle à chaque déplacement de cellules, mais cette technique pose le même problème si les pas de temps sont constants. La deuxième, la plus simple à mettre en œuvre et la plus implémentée, fournit une fonction de conflit dans le coordinateur. Cette fonction est appelée lorsque plusieurs événements ont la même date d'échéance. Lorsque cette fonction est surchargée par le modélisateur, elle lui offre la possibilité de privilégier un événement sur un autre en utilisant le nom des modèles destinataires par exemple.

Cette solution, simple à intégrer dans un environnement hiérarchique, ne peut l'être lors de l'utilisation d'une mise à plat de cette hiérarchie puisque la fonction de conflit, développée dans l'unique coordinateur, doit gérer les conflits d'événements de tous les modèles atomiques. La complexité de cette fonction dépend alors du nombre de modèles atomiques du système modélisé, ainsi que des choix du modélisateur.

La solution que nous employons pour résoudre le problème des événements simulta-

nés est d'utiliser une nouvelle classe de simulateurs abstraits : *DEVS parallèle* dont les paragraphes suivants décrivent le fonctionnement.

2.3.2 Le simulateur DEVS parallèle

L'extension DEVS parallèle, développée par B. P. Zeigler a pour rôle de fournir une technique simple de parallélisation des calculs. Les extensions traditionnelles utilisent des techniques où des branches de la hiérarchie de modèles sont réparties sur des calculateurs. Un point de synchronisation est alors nécessaire pour effectuer un test sur la causalité des événements. Plusieurs techniques de synchronisation existent : optimiste, de type *Time Warp* et pessimiste. Elles sont décrites dans [Zeigler *et al.*, 2000]. Un rappel est proposé dans la section 2.1.3.2.

2.3.2.1 DEVS parallèle

La méthode DEVS parallèle utilise une méthode simple de parallélisation des calculs en utilisant une parallélisation des modèles lorsqu'au dépilement d'une date dans l'échéancier, toutes les fonctions de transition internes ou externes des modèles sont effectuées en parallèle.

B. P. Zeigler définit DEVS parallèle à partir de DEVS classique. Dans le cadre de cette thèse, nous utilisons DEVS classique avec ports. Le modèle atomique, utilisant DEVS parallèle avec ports, est le suivant :

$$DEVS = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{cond}, \lambda, ta \rangle$$

où :

$$\begin{cases} X = \{(p,v) | p\ IPorts, v \in V_X\} \text{ est l'ensemble des valeurs d'entrée avec :} \\ \qquad IPorts \text{ l'ensemble des ports d'entrée,} \\ \qquad V_X \text{ l'ensemble des valeurs possibles sur les ports d'éntrée,} \\ Y = \{(p,v) | p\ OPorts, v \in V_X\} \text{ est l'ensemble des valeurs de sortie avec :} \\ \qquad OPorts \text{ l'ensemble des ports de sortie,} \\ \qquad V_Y \text{ l'ensemble des valeurs possibles sur les ports de sortie,} \\ S \text{ est l'ensemble des états.} \end{cases}$$

Les fonctions de transitions, de sorties et d'avancement du temps sont définies par :

$$\begin{cases} \delta_{int} : S \rightarrow S \text{ la fonction de transition interne,} \\ \delta_{ext} : Q \times X^b \rightarrow \text{ la fonction de transition externe où :} \\ \qquad X^b \text{ est un ensemble du sac d'éléments finis de } X, \\ \delta_{con} : S \times X^b \rightarrow \text{ la fonction de conflit où :} \\ \qquad \delta_{con}(s, \emptyset) = \delta_{int}(s), \\ \lambda : S \rightarrow Y^b \text{ la fonction de sortie,} \\ ta : S \rightarrow \mathbb{R}_0^+ \cup \infty \text{ la fonction d'avancée du temps où :} \\ \qquad Q = (s, e) | s \in S, 0 < e < ta(s) \\ \qquad e \text{ est le temps écoulé depuis le dernier changement d'état.} \end{cases}$$

La différence entre DEVS parallèle et le DEVS classique est l'utilisation de sac d'événements en entrée sur la fonction de transition externe. Un sac, ou *bag* dans la terminologie de B. P. Zeigler, est un ensemble d'événements non triés provenant d'une, ou plusieurs, sources différentes.

La fonction de conflit, δ_{con}, introduite par DEVS parallèle, donne la possibilité au modélisateur de choisir, pour un modèle, entre ses événements externes et internes s'ils ont lieu à la même date. Cette fonction remplace la fonction de sélection de DEVS classique.

La structure d'un modèle couplé dans DEVS parallèle est identique à celle définie dans le premier chapitre. Les ensembles EIC et EOC, respectivement les ensembles de connexions entre le modèle couplé et les sous-modèles ; et les sous-modèles vers le modèle couplé, n'existent plus en raison de la mise à plat de la hiérarchie de modèle DEVS. De la même manière, la fonction de sortie, λ, du modèle couplé n'a plus lieu d'exister. La définition du modèle couplé est la suivante :

$$N = \langle D, \{M_d\}, IC \rangle$$

où pour chaque $d \in D$:

$$M_d \text{ est un modèle atomique DEVS parallèle}$$

La différence entre DEVS parallèle et DEVS classique intervient surtout dans la définition des fonctions de transitions. Pour simplifier la définition de ces fonctions, nous définissons les ensembles suivants :

– Les modèles imminents, $IMM(s)$ dont des sorties vont être générées juste avant la prochaine transition :
$$IMM(s) = \{d | \sigma_d = ta(s)\}$$

– L'ensemble des modèles recevant au moins un message en entrée :
$$INF(s) = \{d | i \in IC_{i,d}, i \in IMM(s) \wedge x_d^b \notin \emptyset\}$$

avec x^b un sac d'événements en entrée :

$$x_d^b = \{IC_{i,d}(\lambda_i(s_i))|i \in IMM(s) \cap IC_{i,d}\}$$

– Le sous-ensemble des modèles de $IMM(s)$ recevant au moins une entrée et effectuant une transition interne :

$$CONF(s) = IMM(s) \cap INF(s)$$

– Le sous-ensemble des modèles de $IMM(s)$ qui n'ont pas de messages d'entrée :

$$INT(s) = IMM(s) - INF(s)$$

– Le sous-ensemble des modèles de $IMM(s)$ recevant un événement en entrée mais n'effectuant de transition interne :

$$EXT(s) = INF(s) - IMM(s)$$

La fonction d'avancée du temps se traduit alors :

$$ta(s) = minimum\{\sigma_d|d \in D\}$$

où :

$$\begin{cases} s \in S \\ \sigma_d = ta(s_d) - e_d \end{cases}$$

La fonction de transition interne résultante est le compromis des quatre types de transition, transition internes $INT(s)$, externes $EXT(s)$ et en conflit $CONF(s)$. Nous définissons :

$$\delta_{int}(s) = (\ldots, (s'_d, e'_d), \ldots)$$

où :

$$\begin{cases} (s'_d, e'_d) = (\delta_{int,d}(s_d), 0) & \text{pour } d \in INT(s) \\ (s'_d, e'_d) = (\delta_{ext,d}(s_d, e_d + ta(s), x_d^b), 0) & \text{pour } d \in EXT(s) \\ (s'_d, e'_d) = (\delta_{con,d}(s_d, x_d^b), 0) & \text{pour } d \in CONF(s) \\ (s'_d, e'_d) = (s_d, e_d + ta(s)) & \text{dans les autres cas} \end{cases}$$

La fonction de transition externe découpe le sac d'événement x^b vers les modèles influencés par l'événement x_d^b. Elle est définie comme :

$$\delta_{ext}(s, e, x^b) = (\ldots, (s'_d, e'_d), \ldots)$$

où :

$$\begin{cases} 0 < e < ta(s) & \\ (s'_d, e'_d) = (\delta_{ext,d}(s_d, e_d + e, x_d^b), 0) & \text{pour } d \in EIC \\ (s'_d, e'_d) = (s_d, e_d + e)(s)) & \text{dans les autres cas} \end{cases}$$

Enfin, la fonction de conflit est appelée quant un modèle possède un événement interne et au moins un événement externe à la même date. À la différence de DEVS classique qui doit prendre en compte les événements des connexions de l'ensemble EIC, la fonction de conflit, dans un environnement à plat de la hiérarchie, se traduit de la même manière que la fonction de transition interne :

$$\delta_{inf}(s, x^b) = (\ldots, (s'_d, e'_d), \ldots)$$

où :

$$\begin{cases} (s'_d, e'_d) = (\delta_{int,d}(s_d), 0) & \text{pour } d \in INT(s) \\ (s'_d, e'_d) = (\delta_{ext,d}(s_d, e_d + ta(s), x^b_d), 0) & \text{pour } d \in EXT(s) \\ (s'_d, e'_d) = (\delta_{con,d}s_d, x^b_d), 0) & \text{pour } d \in CONF(s) \\ (s'_d, e'_d) = (s_d, e_d + ta(s)) & \text{dans les autres cas} \end{cases}$$

La traduction de l'extension DEVS parallèle, vers les simulateurs abstraits DEVS est proposée dans [Chow et Barros, 1994]. Nous ne développerons pas ces algorithmes dans notre simulateur puisqu'ils fusionnent simplement les algorithmes des précédentes sections avec ceux de A. C. Chow et F. J. Barros.

2.3.3 Les événements instantanés

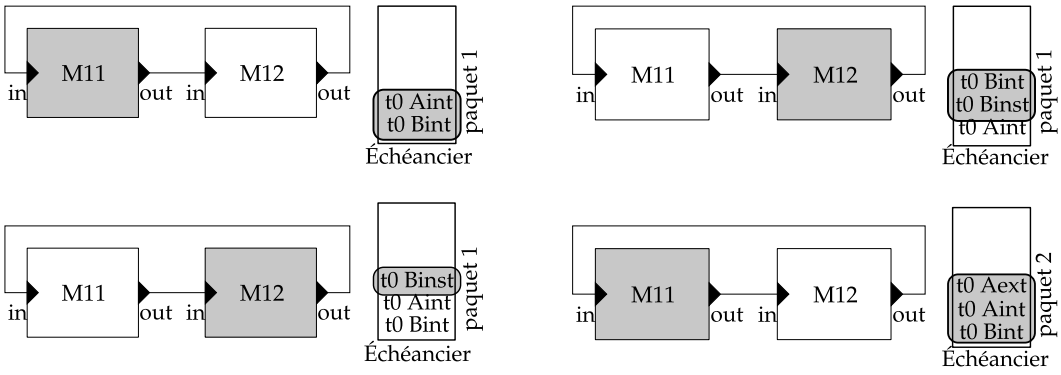
Aujourd'hui, la complexité atteinte par les modèles nous oblige à poursuivre plus en avant la simplification du processus de leurs développements. Dans ce cadre, nous avons étudié une particularité des modèles DEVS lors de la génération d'événements de type question-réponse. Ce type de relation se traduit par un événement envoyé d'un modèle A vers un modèle B afin de fournir au modèle A la valeur d'une des variables d'états du modèle B sans que celui-ci ne soit modifié. Ce type de relation peut se mettre en place dans un graphe d'états de modèle DEVS, cependant, celui-ci voit sa complexité augmentée par l'ajout d'un état supplémentaire pour chaque état du système.

Afin de simplifier l'écriture des modèles, R. Duboz [Duboz, 2004], pour l'écriture du modèle copépode, utilise un appel direct d'une fonction du modèle B par le modèle A, la manière la plus naturelle, mais aussi, une technique posant des problèmes d'ordre opérationnel et en contradiction avec les événements :

- Le problème technique engendré par ce type d'écriture est que le modèle A doit avoir une référence sur le modèle B et le modèle B une fonction d'interrogation et de réponse connue du modèle A. Ainsi, la mise en place d'une technique homogène permettant à tous les modèles de connaître les modèles à questionner peut-être compliquée à mettre en œuvre, surtout, avec l'ajout dynamique de modèle décrit dans la partie 2.4 ;
- La technique d'appel direct d'une fonction pose un problème d'ordonnancement des événements. En effet, si le modèle A effectue une transition et demande une valeur

d'état du modèle, via un appel de fonction ; et si le modèle B doit effectuer une transition à la même date, alors A doit attendre que B effectue sa transition avant d'effectuer la sienne. Ceci dans le but de ne pas rompre la causalité, or le choix n'étant pas fourni au modélisateur, cette technique n'est pas exploitable.

Afin de répondre à ce problème, nous proposons d'ajouter une nouvelle fonction de transition, nommée transition instantanée δ_{inst} et d'un nouveau type d'événement, l'événement instantané. Cette fonction est appelée par le simulateur lors de la réception d'un événement instantané sur une des entrées du modèle. La fonction interdit la modification de l'état du modèle de la même manière que la transition d'état et permet, de sortir un événement externe sur un des ports de sortie du modèle. Les événements instantanés sont placés dans le paquet d'événements courants (cf. section 2.3 sur l'intégration du simulateur parallèle DEVS). Tout comme les événements d'états, ils sont dépilés à la fin du paquet, c'est-à-dire après les événements internes et externes. Comme les événements instantanés et d'états ne peuvent modifier l'état du modèle, leur ordonnancement à peu d'importance. Le but est ici de forcer la capture de l'état du modèle à la même date et lors de la même transition que le modèle questionnant afin d'éviter de récupérer un état futur.



t0 Binst : Événement instantané sur le modèle B.
t0 Bint : Événement interne sur le modèle B.
t0 Aint : Événement interne sur le modèle A.
t0 Aext : Événement externe sur le modèle A.

FIG. 2.10 – Exemple de simulation avec la gestion des événements instantanés.

La figure 2.10 propose un scénario simplifié du fonctionnement du simulateur avec l'intégration des événements instantanés. Elle montre la progression de la simulation, l'état des modèles et l'échéancier, lors de l'utilisation d'un événement instantané :

1. Le modèle A effectue une transition interne et exécute sa fonction de sortie. Cette fonction pose un événement instantané sur le port *out* ;
2. Le modèle B effectue sa transition interne et aucun événement de sortie n'est créé ;
3. Le sac d'événement, à la date $t0$, ne possède plus d'événement interne ou externe. La fonction de transition instantanée du modèle B est appelée ;

4. La fonction de transition instantanée crée un événement externe qu'elle pose sur son port *out*. Cet événement trouve sa place dans le 2^{e} paquet d'événements et possède la valeur de l'état du modèle B lors du traitement du 1er paquet ;
5. Le modèle A doit évaluer sa fonction de conflit, δ_{con}, pour sélectionner la priorité entre l'événement interne et l'événement externe.

La structure du modèle atomique évolue :

$$M = \langle X, Y, X_{init}, Y_{state}, S, \delta_{ext}, \delta_{int}, \delta_{init}, \delta_{state}, \delta_{inst}, \delta_{con}, \lambda, ta \rangle$$

où la fonction de transition d'événements instantanés est définie :

$$\begin{cases} \delta_{inst} : Q \times X^b \rightarrow Y^b \\ \qquad\qquad Q = \{(s,e) | s \in S, 0 < e < ta(s)\} \\ \qquad\qquad e \text{ le temps écoulé depuis la dernière transition d'état.} \end{cases}$$

Cette fonction, déclenchée à la réception d'un événement instantané du sac X^b, pose un événement externe sur un de ces ports, récupéré dans le sac d'événements courants Y^b avec comme particularité de ne pas modifier l'état du modèle.

2.4 Changement de structures

Le formalisme DEVS utilise, par défaut, une hiérarchie de modèles statiques. Néanmoins, des changements dans cette structure permettraient de représenter la dynamique des systèmes de manière plus complète ou, de simplifier le comportement des modèles. Les premiers travaux d'introduction de la notion de structure dynamique dans la modélisation et la simulation sont donnés par G. Klir [Klir, 1985] en 1985. Depuis, A.M. Uhrmacher a proposé ses travaux [Uhrmacher, 2001] de formalisation du changement de structures dans DEVS. Néanmoins, F. Barros propose, avec son extension DS-DEVS [Barros, 1996], une solution complète pour l'intégration du changement de structures dans DEVS. Nous utilisons ses travaux dans notre développement du simulateur à base de hiérarchie à plat de modèles DEVS et de fonctionnement parallèle.

L'extension DS-DEVS conserve toutes les propriétés de DEVS : la modularité, la décomposition hiérarchique et la fermeture sous couplage. Néanmoins, pour son intégration dans notre travail, nous devons modifier son fonctionnement. Ce travail est effectué dans la première partie de cette section. En deuxième partie, nous développons les possibilités apportées par cette extension en terme de manipulations de structures. Enfin, la dernière section est une étude d'optimisation, en terme de vitesse, des simulations grâce à l'emploi d'une technique basée sur DS-DEVS.

2.4.1 Présentation de DS-DEVS

L'extension DS-DEVS, introduite par F. J. Barros [Barros, 1996], a été présentée dans le chapitre précédent dans la partie 1.3.4. Pour rappel, un modèle couplé, dit *réseau de structure dynamique*, est formalisé par :

$$DSDEVSN_{\Delta} = \langle X_{\Delta}, Y_{\Delta}, \chi, M_{\chi} \rangle$$

où :

$$\begin{cases} X_{\Delta} \text{ et } Y_{\Delta} \text{ sont les ports d'entrée et de sortie,} \\ \Delta \text{ est le nom du modèle DS-DEVS,} \\ \chi \text{ le nom du modèle exécutif,} \\ M_{\chi} \text{ est le modèle de l'exécutif.} \end{cases}$$

Le modèle de couplé DS-DEVS, $DSDEVS_{\Delta}$, possède un modèle atomique spécial nommé l'*exécutif* et noté M_{χ}. Ce modèle possède, comme état, toutes les informations sur la structure du modèle couplé dans lequel il se trouve. La modification de la structure des modèles peut alors survenir lors des transitions de fonctions interne ou externe. Le modèle exécutif est défini comme :

$$M_{\chi} = \langle X_{\chi}, S_{\chi}, Y_{\chi}, \delta_{int_{\chi}}, \delta_{ext_{\chi}}, \lambda_{\chi}, \tau_{\chi} \rangle$$

La définition d'un état, s_{χ}, de M_{χ} est :

$$s_{\chi} = (X_{\Delta}^{\chi}, Y_{\Delta}^{\chi}, D^{\chi}, \{M_i^{\chi}\}, \{IC_i^{\chi}\}, \{EIC_i^{\chi}\}, \{EOC_i^{chi}\}\{Z_{i,j}^{\chi}\}, \Xi^{\chi}, \theta^{\chi})$$

où :

$$\begin{cases} D^{\chi} \text{ est l'ensemble des modèles atomiques de } DSDEVS_{\Delta}, \\ M_i^{\chi} \text{ est le modèle } i \text{ avec } i \in D^{\chi}, \\ IC_i^{\chi}, EIC_i^{\chi} \text{ et } EOC_i^{\chi} \text{ est l'ensemble des connexions,} \\ \Xi^{\chi} \text{ la fonction de sélection des conflits,} \\ \theta^{\chi} \text{ un ensemble de variables attachées au réseau.} \end{cases}$$

Après ce rappel sur les structures des entités apportées par DS-DEVS, nous développons dans la prochaine section, son intégration dans un environnement DEVS à hiérarchie mise à plat.

2.4.2 DS-DEVS dans un environnement DEVS mis à plat

L'utilisation des algorithmes de F. J. Barros [Barros, 1996] ne peut se faire directement dans notre simulateur. En effet, il propose de modifier l'algorithme principal du coordinateur afin de lui donner les capacités de modifications du graphe du modèle couplé

lorsqu'un modèle *exécutif* est présent dans celui-ci. Toutefois, cette technique ne peut être introduite telle quelle en raison du choix de mise à plat de la hiérarchie de modèles lors de la simulation mais pas lors de la modélisation.

L'un des premiers problèmes soulevés par la mise à plat de la hiérarchie provient du fait que le fonctionnement du coordinateur est modifié par l'extension DS-DEVS pour gérer les événements spécifiques aux changements de structures. Or, si celui-ci est amené à être changé, il modifie le comportement pour les simulations sans modèle *exécutif* de DS-DEVS.

Le second problème, décrit sur la figure 2.11, survient lors de l'ajout ou la suppression de connexions externes du modèle couplé. Dans une approche de mise à plat de la hiérarchie, celles-ci n'existent plus et sont remplacées par des connexions internes de l'unique modèle.

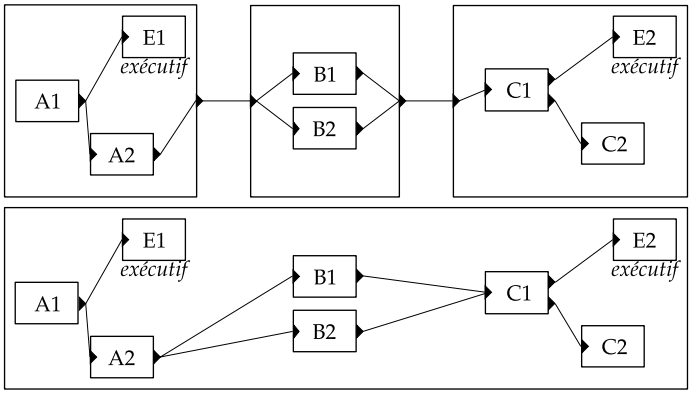


FIG. 2.11 – Correspondance, dans un système dynamique, entre une hiérarchie de modèles avec deux modèles exécutifs, $E1$ et $E2$ dans une mise à plat de modèles.

Sur la figure, si le modèle $C1$ décide de supprimer la connexion entre lui même et le modèle couplé $M3$, la traduction de ce message se traduit dans un environnement DEVS mis à plat, comme la suppression des connexions de $B1$ et $B2$ vers $C1$. Si après cette demande, le modèle $C1$ nécessite la remise en place de cette connexion, via une demande de connexion entre $M3$ et $C1$, alors le système doit se souvenir des connexions entre $B1$ et $B2$ vers $M3$. Ce problème impose la conservation de l'ensemble de la hiérarchie de modèles et de connexions afin de fournir cette information.

Pour résoudre ce problème, nous proposons une solution mixte où la mise à plat de la hiérarchie de modèle est effectuée et un modèle exécutif est ajouté au coordinateur. Ce modèle possède un attribut qui conserve l'ensemble de la hiérarchie de modèles et les graphes de connexions associées.

La figure 2.12 propose la traduction du système présenté sur la figure 2.11 en utilisant cette spécification. Les modèles $E1$ et $E2$, définis par le modélisateur, manipulent les modèles couplés A et C. $E1$ et $E2$ sont des relais pour le modèle exécutif global de la

hiérarchie mise à plat E. Ce modèle est noté formellement par M_{χ_G} et sa structure est la suivante :

$$M_{\chi_G} = \langle X_{\chi_G}, S_{\chi_G}, Y_{\chi_G}, \delta_{int_{\chi_G}}, \delta_{ext_{\chi_G}}, \lambda_{\chi_G}, \tau_{\chi_G} \rangle$$

La définition d'un état, s_{χ_G}, de M_{χ_G} est :

$$s_\chi = (X^\chi_\Delta, Y^\chi_\Delta, D^G, \{M^G_i\}, \{IC^G_i\}, \{EIC^G_i\}, \{EOC^G_i\}\{Z^G_{i,j}\}, \Xi^\chi)$$

La différence entre la structure définie par F. J. Barros et celle de notre spécification réside dans la variable θ_χ qui disparaît de notre spécification puisqu'elle définit des variables supplémentaires au comportement du modèle exécutif, ce qui est incompatible avec une mise à plat où le modèle exécutif M_{χ_G} ne peut être modifié par un modélisateur. Les variables contenant la structure sont également modifiées. En effet, dans sa définition F. J. Barros enregistre dans ces variables, l'état du modèle couplé dynamique ; dans notre spécification, ces variables contiennent l'ensemble de la hiérarchie de modèles.

Le modèle M_{χ_G} est un modèle réactif, c'est-à-dire, qu'il se trouve dans un état passif tant qu'il ne reçoit pas de requêtes de modification du graphe d'un modèle exécutif M_χ. Il ne possède pas de comportement.

Les avantages de cette spécification de modèles exécutifs sont qu'elle permet de conserver le fonctionnement des modèles exécutifs de F. J. Barros, tout en respectant l'approche de mise à plat de la hiérarchie sans troubler le modélisateur, celui-ci conservant une approche de développement de composants. Les modèles exécutifs des modèles couplés envoient leurs messages au modèle exécutif global, le seul ayant les accès aux modifications réels de graphes. Le principal inconvénient de cette formalisation est la conservation dans le modèle exécutif global M_{χ_G}, de la hiérarchie de modèles qu'il doit adapter en même temps que la version réelle, utilisée par le simulateur entraînant un surcoût de calcul pouvant être résolu par des algorithmes adaptés.

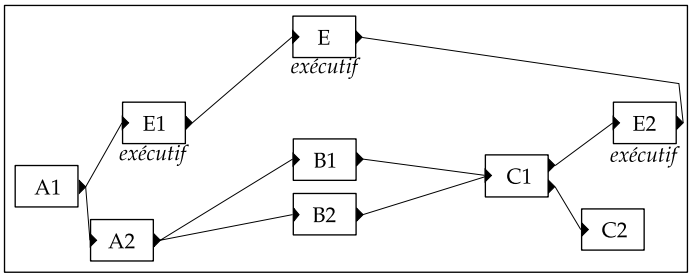


FIG. 2.12 – Traduction de la figure 2.11 dans notre spécification DEVS où la hiérarchie est mise à plat.

Les modifications apportées par l'inclusion de DS-DEVS dans le formalisme DEVS parallèle dont la hiérarchie est mise à plat sont présentées dans la section suivante.

2.4.2.1 Coordinateur global

Le modèle exécutif global M_{χ_G} apporte une légère modification dans le fonctionnement du coordinateur global et d'un point de vue technique, dans l'échéancier de celui-ci. En effet, F. J. Barros [Barros, 1997], afin d'éviter de modifier les structures du graphe pendant que des événements circulent, recommande de placer les événements de changement de structures à la fin du sac d'événements de DEVS parallèle. Ce changement dans le fonctionnement de l'échéancier vu précédemment est trivial et ne sera pas étudié dans ces paragraphes.

Les algorithmes des simulateurs abstraits basés sur DEVS Parallèle et sur DS-DEVS sont proposés par F. J. Barros dans [Barros, 1998a] et permettent de résoudre le problème des événements simultanés [Barros, 1998b]. Lors de l'utilisation d'une mise à plat de la hiérarchie de modèles, les algorithmes développés précédemment restent équivalent, en omettant la différence de tris de l'échéancier pour déplacer les événements de changement de structure en fin de sac d'événements.

Les possibilités de modification de la hiérarchie de modèle dépendent uniquement des possibilités du modèle M_{χ_G}. Elles sont présentées dans la section suivante, après une présentation du comportement du modèle.

2.4.3 Modèle exécutif global

Comportement

Le comportement du modèle M_{χ_G} est traduisible par un modèle réactif. L'état du modèle est défini par la représentation de la hiérarchie de modèle et par une variable *phase* qui peut prendre l'état *idle* lorsqu'il est dans un mode passif, *sauve* pour la sauvegarde et $modif$ pour la modification de l'état. Une variable l est nécessaire pour la sauvegarde des différentes demandes de modification du graphe d'états du sac d'événements :

$$S = \{phase, S_{\chi_G}, L\}$$

L'état initial du modèle est inactif, la fonction de conflit place les événements internes en priorité afin d'effectuer les opérations de modification de structures les unes après les autres :

$$\begin{cases} S_{init} = \{idle, S_{\chi_G}, \emptyset\} \\ \delta_{con} = \delta_{int}(\delta_{ext}) \\ ta(S_{init}) = \infty \end{cases}$$

À la réception des événements externes des modèles exécutifs du modélisateur, tous les événements externes sont stockés dans une liste triée par la fonction Ξ_{χ_G} si deux modèles exécutifs modifient la même connexion :

$$\begin{cases} \delta_{ext}(idle, S_{\chi_G}, L) \times (type) \rightarrow (sauve, S_{\chi_G}, L \cup type) \\ \delta_{ext}(sauve, S_{\chi_G}, L) \times (type) \rightarrow (sauve, S_{\chi_G}, L \cup \Xi_{\chi_G}(\{L\}, type) \\ ta(sauve, S_{\chi_G}, L) = 0 \end{cases}$$

Lorsque le sac d'événement externe est vidé, le modèle exécutif réalise les opérations demandées et repasse en mode inactif :

$$\begin{cases} \delta_{int}(sauve, S_{\chi_G}, L) = (sauve, modif(S_{\chi_G}, L_1), L = L_{(2..n)}) \\ ta(sauve, S_{\chi_G}, L) = 0 \\ \delta_{int}(sauve, S_{\chi_G}, \emptyset) = (idle, S_{\chi_G}, \emptyset) \\ ta(idle, S_{\chi_G}, 0) = \infty \end{cases}$$

Dans ce fonctionnement, les messages envoyés des modèles ont la capacité d'ajouter ou de supprimer des connexions, des ports ou des modèles. Pour les modèles atomiques, on peut également changer leurs comportements, ou créer de nouveaux comportements. La section suivante décrit la fonction de suppression de connexion de la figure 2.12.

Exemple de message : suppression d'une connexion

Avant de décrire la fonction de suppression d'une connexion dans un modèle couplé dynamique, il faut décrire le contenu du message de suppression d'une connexion externe d'entrée de modèle :

$$Msg = (M_\chi, (M_o, P_o), (M_d, P_d))$$

où M_χ est le modèle exécutif qui envoie le message, (M_o, P_o) l'origine de la connexion, un modèle couplé et (M_d, P_d) la destination de la connexion.

La suppression de cette connexion nécessite deux étapes, la première consiste à supprimer la connexion qui existe dans le simulateur, la deuxième, à supprimer celle stockée dans le modèle exécutif global.

La suppression de cette connexion du simulateur nécessite de connaître le, ou les, modèles atomiques pouvant émettre des messages d'origine de la suppression et non les interfaces des modèles couplés intermédiaires. Pour cela, il est nécessaire d'obtenir des références sur le modèle parent et grand- parent du modèle exécutif puis, via un processus récursif, de trouver l'ensemble des modèles atomiques.

La suppression de la connexion dans le modèle exécutif global est plus simple, celui-ci possédant l'ensemble des connexions, il suffit de retirer cette connexion de l'ensemble des connexions EOC_i^G :

$$EOC_i^G = EOC_i^G - EOC_{((M_o,P_o),(M_d,P_d))}$$

2.4.4 Optimisations des échanges de messages

Dans les paragraphes précédents nous avons décrit le fonctionnement de l'extension DS-DEVS et nous l'avons intégré dans notre proposition de simulateur. Nous montrons, dans cette partie, l'utilisation de cette extension pour accélérer la simulation des modèles.

Une des caractéristiques du formalisme DEVS est sa modularité. Les systèmes sont découpés et hiérarchisés en modèles atomiques ou couplés et ils sont reliés entre eux via des connexions. Les modèles atomiques utilisent la fonction de sortie λ pour poser un événement externe sur un port de sortie du modèle. Le simulateur récupère ces événements, calcule les destinations en suivant les connexions entre le port de sortie du modèle et les ports d'autres modèles. Un problème d'efficacité survient lorsqu'un modèle pose un événement destiné à un autre modèle sur un de ses ports de sortie, alors que ce port est connecté à plusieurs modèles. L'événement est dupliqué en suivant le nombre de modèles connectés au port de sortie et ceux-ci voient leurs fonctions de transition externe, δ_{ext}, appelées pour traiter l'événement, même si leur leur comportement est d'ignorer ce message.

À l'aide de l'extension DS-DEVS, nous développons deux techniques pour pallier au problème d'efficacité lors de l'utilisation des événements dupliqués et non utiles pour les modèles. La première utilise un modèle atomique nommé répartiteur connecté au modèle *exécutif* de DS-DEVS qui, en recevant un message particulier, manipule le graphe DEVS en supprimant les connexions superflues au port de sortie des modèles. La deuxième modifie les ports du modèle source et crée une connexion entre ce nouveau port et le modèle de sortie. Ces deux techniques ont chacune des avantages et des inconvénients que nous aborderons dans les deux prochaines sections.

2.4.4.1 Répartiteur

Le rôle du modèle répartiteur, *dispatcher*, est de diminuer le nombre de messages inutiles lors des communications entre les modèles. Si un modèle A est connecté à deux modèles B et C via son port de sortie out, lorsque A pose un événement sur ce port, les deux modèles B et C recevront ce message. Il y a une duplication de l'événement à cet instant. Or, si le message n'est adressé qu'à un seul modèle, les deux modèles vont devoir faire un test si le message leur est adressé.

Le modèle répartiteur corrige ce problème en utilisant l'extension DS-DEVS pour créer les connexions entre modèles à la demande. Pour cela, il a besoin d'une information supplémentaire : le ou les destinataires. Cette information encapsule le message réel. À la réception d'un message de ce type, le répartiteur crée les connexions vers les destinataires et enlève la capsule du message.

Le modèle répartiteur est un modèle atomique dont la structure est :

$$M_{répartiteur} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$$

Les entrées sont structurées par deux ports, le premier reçoit les messages des modèles voulant envoyer un message, le deuxième est un port pour la réception d'un accusé de réception. Les deux ports de sorties définissent une sortie vers les modèles destinataires et une autre pour envoyer des demandes de modification au modèle exécutif du modèle couplé dynamique $DSDEVSN$.

$$\begin{cases} X = \{(in, ((dest, port), msg), (exe, resultat)\} \\ Y = \{(exe, (op, (dest, port)), (out, msg)\} \end{cases}$$

L'état du modèle est caractérisé par deux valeurs, *phase* représente l'état courant et L une file de message l définie par un ensemble de $(nom\ de\ modèle, port)$ et d'un message (msg).

$$S = \{phase, L\}$$

Les différentes étapes du modèle répartiteur sont :

- $idle$: le modèle est dans un état passif ;
- $modification$: un modèle demande l'envoi d'un message ;
- $attente_{modif}$: le répartiteur est en attente d'un accusé de réception ;
- $message$: l'envoi du message à tous les modèles connectés au port Y_{out}.

Le modèle répartiteur est dans un état passif en début de simulation, il est en attente d'une demande d'envoi de message :

$$\begin{cases} S_{init} = \{idle, \emptyset\} \\ ta(idle, L) = \infty \end{cases}$$

Dès que le modèle est perturbé par un événement externe, il stocke les caractéristiques de l'événement dans sa file L et attend de pouvoir demander de créer les connexions au modèle exécutif M_χ :

$$\begin{cases} \delta_{ext}(idle, L) \times (in, ((dest, port), msg)) \rightarrow (modification, L \cup l(((dest, port), msg)) \\ ta(modification, L) = 0 \end{cases}$$

Si le modèle reçoit encore des demandes d'envoi de message, via les sacs de messages DS-DEVS, elles sont stockées dans la file des messages :

$$\begin{cases} \delta_{ext}(modification, L) \times (in, (dest, port, msg)) \rightarrow (modification, L \cup l(dest, port, msg)) \\ ta(modification, L) = 0 \end{cases}$$

Quand l'événement interne est activé, le modèle envoie la demande de modification du graphe d'état au modèle M_χ et se met en attente de l'accusé de réception :

$$\begin{cases} \lambda(modification, L) : l_{top} \rightarrow Y_{exe} \\ \delta_{int}(modification, L) \rightarrow (attente_{modif}, L) \\ ta(attente_{modif}, L) = \infty \end{cases}$$

De la même manière que précédemment, si des messages de communication de modèles perturbent le fonctionnement, ils sont simplement ajoutés à la file de message :

$$\begin{cases} \delta_{ext}(attente_{modif}, L) \times (in, (dest, port, msg)) \rightarrow (attente_{modif}, L \cup l(dest, port, msg)) \\ ta(attente_{modif}, L) = 0 \end{cases}$$

À la réception de l'accusé de réception du modèle M_χ, le modèle répartiteur pose directement le message du premier événement de la file sur son port de sortie :

$$\begin{cases} \delta_{ext}(attente_{modif}, L) \times (exe, resultat) \rightarrow (message, L) \\ ta(messsage, L) = 0 \\ \lambda(message, L) : l(dest, port, msg) \rightarrow (out, msg) \end{cases}$$

Si la liste de messages est vide, le modèle se met dans un état passif après avoir vidé son dernier élément, sinon il fait une nouvelle demande de changement de graphe au modèle M_χ.

$$\begin{cases} \delta_{int}(message, \emptyset) \rightarrow (idle, \emptyset) \\ ta(idle, \emptyset) = 0 \\ \\ \delta_{int}(message, L) \rightarrow (modification, L) \\ ta(modification, L) = 0 \end{cases}$$

Le modèle répartiteur que nous venons d'étudier est une réponse pour limiter les duplications d'événements qui ont lieu lorsque plusieurs modèles sont connectés à un même port, sans être obligatoirement le destinataire de tous les messages. Le répartiteur utilise DS-DEVS pour relier les modèles, uniquement lorsqu'ils sont les destinataires du message. Ce modèle peut être amélioré en ajoutant une technique de port dynamique. Ces ports servent alors de cache de connexions afin de limiter la création ou la suppression de connexions entre modèles. Ce fonctionnement ne fait pas l'objet d'une étude formelle dans cette section afin de limiter la complexité de la description du comportement du modèle vu précédemment.

2.5 Conclusion

Dans ce chapitre, nous avons proposé et développé une étude d'une nouvelle spécification DEVS dont la principale caractéristique est la mise à plat de la hiérarchie de modèles DEVS. Cette technique consiste à supprimer les modèles couplés lors de la simulation afin de ne laisser apparaître qu'un seul coordinateur qui prend en charge la globalité de la simulation. L'utilisation d'un seul coordinateur dans une simulation DEVS permet d'éviter les échanges d'événements entre coordinateurs ainsi que le coût supplémentaire introduit par ce déplacement. Cette modification ne perturbe pas la spécification formelle de DEVS puisqu'elle s'appuie sur une de ses propriétés, la fermeture sous couplage. Celle-ci annonce qu'un modèle couplé est équivalent à un modèle atomique. De plus, l'utilisation de cette technique concerne uniquement la partie simulation. Le modélisateur continue d'utiliser les concepts de base de DEVS : la composition et la hiérarchie de modèles. La mise à plat est alors transparente pour le modélisateur tout en diminuant les coûts liés à la hiérarchie de coordinateur de DEVS classique. La mise à plat de la hiérarchie de modèles DEVS engendre cependant certains problèmes. Notamment lors de la parallélisation de simulation et lors de l'utilisation de l'extension DS-DEVS pour le changement dynamique de structures. Des solutions à ces problèmes sont proposées même si ces méthodes sont perfectibles et nécessitent un approfondissement.

Cette spécification DEVS propose, également, l'ajout au formalisme DEVS de caractéristiques permettant de simplifier le développement des comportements de modèles en utilisant des ajouts de types d'événements ou de fonctions de transitions. Ainsi, les événements instantanés fusionnent les fonctions de transitions externes et des sorties dans une fonction ne permettant pas de modifier l'état du modèle. Les événements

d'initialisation sont, quant à eux, une réponse formelle à l'initialisation dynamique des modèles. Nous proposons également une méthode pour l'observation des états des modèles sans perturber leurs fonctionnements grâce à l'utilisation d'une fonction de transition qui peut être appelée à chaque événement interne ou externe, ou bien de manière datée.

L'ensemble des travaux de cette spécification DEVS va nous permettre de développer les chapitres suivants. En effet, ce travail est la base de notre spécification agent et du cadriciel de multi-modélisation et de simulation VLE. La spécification utilise les extensions DEVS parallèles et des événements instantanés. La plate-forme logicielle propose une implémentation de l'ensemble de ces techniques.

Chapitre 3

Formalisation du paradigme agent

Sommaire

Les premiers travaux réalisés par É. Ramat [Ramat et Preux, 2003] sur VLE avaient pour but d'introduire une plate-forme pour les Systèmes Multi-Agents, SMA, où le temps était discrétisé et le comportement des agents modélisé à l'aide d'un langage de haut niveau reposant sur un ensemble de structures algorithmiques. Dans ce chapitre, nous allons développer, à partir des travaux de R. Duboz [Duboz, 2004], une formalisation du concept agent qui se base sur les spécifications introduites dans le chapitre 2.

Le but de cette formalisation des SMA est de fournir une spécification complète pour la modélisation de SMA intégrant les principes-même de DEVS, c'est-à-dire, la modularité et le couplage de modèles hétérogènes. Cette spécification fait l'objet d'un développement opérationnel dans la plate-forme de multi-modélisation VLE présentée dans le chapitre 4 et dont les simulateurs sont étudiés au chapitre 2.

La finalité attendue de ce travail est de fournir un environnement complet de développement de SMA basé sur les principes de la théorie de la modélisation et de la simulation. De plus, nous essayons de fournir des services pour remédier à la complexité atteinte par les modèles et des SMA en général. Pour cela, nous nous basons sur les concepts principaux de DEVS que sont la modularité et la hiérarchie de modèles et des choix réalisés dans la conception du simulateur présentée au chapitre 2.

3.1 Systèmes Multi-Agents

Les écosystèmes marins et en particulier les interactions de type proie-prédateur d'un groupe majeur du zooplancton, les copépodes, sont notre objet d'étude depuis 1998 avec les travaux de É. Ramat [Ramat *et al.*, 1998]. Cette étude nous a conduit à construire des modèles d'agents réactifs [Ferber, 1995] dans un environnement discrétisé et dans un environnement continu [Duboz *et al.*, 2001]. La modélisation et la simulation, à base d'agents ou d'individus centrés, sont des avancées majeures dans le domaine des écosystèmes, [Bousquet *et al.*, 1994, Hill, 1996, Coquillard et Hill, 1997]) du fait de recentrer l'objet de la modélisation sur les entités du système et leurs interactions.

Des travaux formels ont été menés [Zeigler, 1976, Zeigler *et al.*, 1995, Zeigler *et al.*, 2000]. Ils s'appuient sur les mathématiques des systèmes, pour la mise au point du formalisme DEVS et pour la spécification de modèles à événements discrets et à temps continu. Ces travaux ont aussi conduit à la définition de nombreux formalismes ou extensions, sur la base de DEVS dont les principaux sont GDEVS [Giambiasi *et al.*, 2000], DEV&DESS [Zeigler *et al.*, 2000], Cell-DEVS [Wainer et Giambiasi, 2001] et DS-DEVS [Barros, 1995]. Ces différents formalismes ou extensions DEVS englobent notamment les équations différentielles, les systèmes à temps discret et les automates cellulaires ; ils ont pour objectif l'intégration de différents paradigmes de modélisation. La formalisation DEVS des SMA a fait l'objet d'études, mais elles restent des réponses partielles comme DEVS-RAP [Hocaoglu *et al.*, 2002], pour la définition d'agents cognitifs. Cependant, l'utilisation de l'ensemble de ces travaux semble permettre la création d'un formalisme « Agent-

DEVS ».

3.1.1 Formalisme SMA

Avant de développer la formalisation DEVS des SMA, il est important de bien définir les notions sur lesquelles se basent les SMA. Cette section introduit ces éléments à partir de la définition des termes d'agents et d'environnements.

3.1.1.1 Agents

Aujourd'hui, il est encore difficile de donner une définition précise des agents tant les domaines d'applications sont nombreux. Cependant, les auteurs s'accordent sur les principales caractéristiques de tels systèmes. Nous partons de la définition générale d'un agent donnée par G. Weiss [Weiss, 1999] :

> « *Agents are autonomous, computational entities that can be viewed as perceiving their environment through sensors and acting upon their environment through effectors.*[1] »
>
> G. Weiss [Weiss, 1999]

Les aspects importants sont ici les notions d'environnement et d'interactions. Pour compléter la définition précédente, M. N. Huhngs et M. P. Singh [Huhngs et Singh, 1998] donnent les caractéristiques majeures suivantes pour les systèmes multi-agents :

- chaque agent possède une information incomplète, il est limité dans ses capacités d'action ;
- le contrôle du système est distribué ;
- les données sont décentralisées ;
- les calculs sont asynchrones.

De plus, les deux types d'interactions majeures sont [Ferber, 1995]:

- les interactions directes : les agents interagissent ou communiquent par échanges de messages ;
- les interaction indirectes : les agents interagissent ou communiquent au travers de l'environnement qui joue dans ce cas le rôle de « tableau noir ».

Pour notre part, nous nous intéressons aux systèmes d'agents réactifs situés, parfois désignés sous le terme de modèles centrés individus. Un système d'agents réactifs situés est un système dynamique où il est possible d'ordonner les actions, ou activités des agents dans le temps. Pour cette raison, il est possible d'utiliser le formalisme DEVS

[1] « Les agents sont des entités autonomes et computationelles qui peuvent être vus comme percevant leur environnement à travers des capteurs et agissant sur les environnements avec des effecteurs. »

pour la spécification des systèmes d'agents réactifs. Néanmoins, il ne faut pas oublier le terme *situé* qui est synonyme de représentation de l'espace. Or, l'espace ne faisant pas partie des fondements de DEVS, des choix de modélisation sont alors nécessaires pour son utilisation.

3.1.1.2 Environnements

Dans son manuscrit, J.C. Soulié [Soulié, 2001] classe les environnements en trois catégories, les environnements centralisés, distribués ou modélisés sous la forme d'agent.

- L'environnement centralisé est une structure unique qui définit l'ensemble des éléments de l'environnement. Les agents y accèdent par l'intermédiaire de question. L'environnement réagit à cette demande par une réponse relative à la question. Le principal défaut de cette technique provient du fait qu'elle s'adapte mal à une augmentation de sa complexité, par exemple, lorsque le nombre d'agents ou d'interactions devient important.
- L'environnement distribué est défini par un ensemble de cellules formant une grille ou une construction plus complexe. Chaque cellule peut être vue comme un environnement centralisé sur lequel les agents peuvent dialoguer et se déplacer de cellule en cellule. Le voisinage entre les cellules est alors la partie la plus importante pour l'action des agents.
- La définition d'environnement sous la forme d'agent a été initiée en 1996 sur la plate-forme SWARM [Minar *et al.*, 1996] par C. Langton et plus récemment en 1998 par la plate-forme Madkit [Ferber et Gutknecht, 1998]. Tout comme son nom le définit, les environnements sont ici décrits par des agents. Tous les objets de l'environnement sont alors des agents.

De ses travaux, J. C. Soulié [Soulié, 2001] propose une nouvelle représentation des environnements : les *environnements multiples*. Dans ce type de SMA, les agents sont plongés dans un ou plusieurs environnements simultanément ou séquentiellement. Les environnements possèdent les informations globales, comme la taille de l'espace par exemple et les informations réelles des agents, leurs positions ou leurs directions réelles, pour les environnements physiques. Les agents peuvent alors communiquer aux environnements via un ensemble de capteurs et d'effecteurs, respectivement pour l'acquisition et le changement d'information dans l'environnement. L'agent est ainsi décomposé en deux parties distinctes, la première, que l'on nommera le « corps » représente l'ensemble des capteurs et des effecteurs, la seconde, nommée « tête » contient l'ensemble du ou des comportements de l'agent.

Comme nous le développons dans la prochaine partie, ce type de décomposition est très utile pour l'utilisation d'un formalisme comme DEVS où la modularité et la hiérarchisation des informations permettent de réaliser des compositions de modèles simples. La transformation vers DEVS de ces différents concepts montre la voie vers laquelle nous développerons notre formalisation de SMA, c'est-à-dire, utiliser la modularité et

la hiérarchie introduite par le formalisme DEVS pour créer un ensemble de modèles atomiques représentant les capteurs, les effecteurs et les environnements.

Ce travail s'oriente vers le développement d'une formalisation DEVS des SMA. Nous développerons, dans la première partie, sur la nécessité d'une formalisation des SMA où nous rappelons le rôle de la modélisation et la simulation. Les parties suivantes sont la présentation de notre formalisation et les choix réalisés en terme de concepts, de modularité et de simplification.

3.1.2 Formalisation des systèmes multi-agents

Les systèmes multi-agents apparaissant comme une nouvelle méthode pour la modélisation de systèmes complexes, il est crucial pour ce paradigme d'offrir des algorithmes bien définis. Enfin, l'utilisation de SMA dans le contexte de la modélisation et de la simulation scientifique implique que les logiciels doivent être testables et échangeables. En d'autres termes, la spécification de SMA doit être basée sur une formalisation opérationnelle et sans ambiguïté.

La formalisation de SMA a donné lieu à de nombreux travaux [Duboz *et al.*, 2004], nous pouvons citer ceux rassemblés par G. Weiss qui nous permet de distinguer deux types de formalismes, l'un pour les structures, l'autre pour les comportements. Un de ces travaux, proposés par J. Ferber, est BRIC pour *Block-lite Representation for of Interactive Components* [Ferber, 1995]. Ce formalisme s'appuie sur une structure modulaire et hiérarchique avec un comportement formalisé par un réseau de Petri coloré. La principale limitation de BRIC, présentée par l'auteur lui-même, est l'incapacité de modéliser des systèmes à structures dynamiques, c'est-à-dire, où les connexions entre agents ou environnements peuvent varier en cours de simulation. Le même auteur propose une formalisation basée sur le π-calculus et le *Chemical Abstract Machine* où la priorité est focalisée sur l'exécution en parallèle des agents mais l'aspect des relations dynamiques n'apparaît pas clairement [Ferber et Gutknecht, 2000]. Nous pouvons également citer P. Gruer [Gruer *et al.*, 2002], qui propose de combiner Object-Z et les Statechars pour définir une formalisation SMA, où Object-Z est utilisé pour la structure et les Statechars pour la définition de comportement.

À partir de ces exemples, nous pouvons conclure que l'utilisation de différents formalismes semble être une solution idéale pour spécifier la structure et le comportement dans les SMA. Nous pouvons donc dégager plusieurs types de difficultés lors de l'utilisation d'un unique formalisme :

- Difficultés conceptuelles :
  - la diversité des domaines d'applications ;
  - la diversité des formalismes ;
  - la diversité des points de vue sur un même système.
- Difficultés opérationnelles :

- la diversité des outils d'implémentation ;
- la diversité des algorithmes ;
- la complexité des modèles.

Ces difficultés montrent, en réalité, le potentiel du paradigme agent et les différentes méthodes pour modéliser des systèmes complexes. Dans ce chapitre, nous proposons une vision des SMA par la composition de modèles hétérogènes aux niveaux opérationnels et conceptuels, c'est-à-dire, la multi-modélisation.

Dans les sections suivantes, nous développons une spécification agents basée sur les travaux de R. Duboz avec la formalisation des agents réactifs [Duboz *et al.*, 2002] où nous apportons plusieurs modifications majeures afin de s'appuyer sur les caractéristiques du formalisme DEVS dont nous décrivons les principales caractéristiques dans la section suivante.

3.1.3 Théorie de la modélisation et de la simulation

Les théories de la modélisation et de la simulation s'appuient sur une approche systémique où un système est considéré comme une entité traversée par la matière, l'énergie ou un flux d'information. Cette entité est spécifiée dans une formalisation mathématique sous la forme d'un ensemble d'état et de fonctions affectant cet état. Cette structure est appelée modèle atomique dans le sens où il ne peut être divisé en parties plus petites. Ces modèles atomiques peuvent être combinés par couplage pour former un modèle couplé. Les principales caractéristiques de DEVS sont décrites dans les chapitres 1 et 2.

Les premiers travaux de formalisation des SMA par l'utilisation de DEVS sont ceux de A. M. Uhrmacher [Uhrmacher et Arnold, 1994, Uhrmacher et Kullick, 2000]. Ces travaux mettent en relation les modèles atomiques avec les agents, où les fonctions internes représentent l'autonomie, les fonctions externes, la perception et les fonctions de sorties représentent l'action des agents sur l'environnement ou sur les autres agents. Les agents sont alors vus comme des automates à états finis. Ces travaux ne prennent pas en compte le haut niveau d'abstraction des SMA qui demandent une sémantique plus élaborée, par exemple: la proaction, l'action, le comportement ou les groupes d'agents. M. F. Hocaoglu propose [Hocaoglu *et al.*, 2002] une collaboration entre les SMA et un modèle DEVS qui représente l'environnement mais ne formalise pas le SMA. R. Duboz propose une analogie entre le formalisme DEVS et les agents réactifs [Duboz, 2004] et entre DS-DEVS et les SMA [Duboz *et al.*, 2004].

Dans ce chapitre, nous proposons une approche de définition des SMA sociaux et éthologiques en utilisant DEVS. Notre formalisation s'appuie sur deux concepts, le premier DS-DEVS pour la modification de graphes DEVS en cours de simulation, le deuxième pour la définition des environnements multiples et ses aspects de découpage de l'agent en deux parties. Dans le reste du chapitre, nous utiliserons la formulation DEVS des

modèles. Pour rappel, un modèle DS-DEVS, $DSDEVSN_{\Delta}$ est un modèle couplé ou réseau de modèles dont la structure est :

$$DSDEVSN_{\Delta} = \langle X_{\Delta}, Y_{\Delta}, \chi, M_{\chi} \rangle$$

Le modèle exécutif, M_{χ}, permet aux modèles inclus dans $DSDEVSN_{\Delta}$ de modifier la structure du réseau de modèles via l'envoi de messages sur ses ports d'entrées :

$$M_{\chi} = \langle X_{\chi}, Y_{\chi}, S_{\chi}, \delta_{ext_{\chi}}, \delta_{int_{\chi}}, \lambda_{\chi}, ta_{\chi} \rangle$$

Dans les paragraphes suivants, nous développons une formalisation de la structure et du comportement des agents, avec la formalisation de la perception, de l'action, de la pro-action, de la réaction et de l'autonomie. Dans la dernière partie nous formalisons le concept d'organisation, de groupe et de rôle.

3.1.4 Concepts généraux

Avant de débuter la formalisation, nous abordons les concepts sur lesquels notre spécification se base. La ligne directrice de cette formalisation suit les trois règles suivantes :

- l'hétérogénéité : l'ensemble des composants DEVS doit être utilisable dans le SMA, afin de fournir la multi-modélisation, comme les équations différentielles ou les réseaux de Petri ;
- la simplicité : la complexité grandissante des modèles et des SMA doit être simplifiée afin de permettre, idéalement, son utilisation par des non-informaticiens ;
- l'efficacité : pour être utilisable, l'ensemble de la spécification doit aider à limiter les opérations coûteuses en terme de temps d'exécution.

Le développement suit les concepts DEVS de modularité et de hiérarchie de modèles. Ceux-ci permettent de simplifier le développement de modèles en permettant de réutiliser des modèles existants et de former des modèles plus complexes en utilisant des combinaisons de modèles. Ces choix nous orientent vers l'utilisation d'environnements multiples centralisés, c'est-à-dire, un agent, au lieu de se situer et de communiquer avec un seul environnement, se situe dans plusieurs environnements physiques ou sociaux où toutes les informations sont stockées.

- Les environnements physiques permettent de décrire les informations relatives aux placements des agents dans l'environnement. Nous distinguons plusieurs types d'environnements physiques :
  - discrets : représentation sous la forme d'automates cellulaires dans les trois dimensions spatiales où les communications entre voisins peuvent suivre les règles de Van Neumann, Moore, etc. ;

  - continus : zone de l'espace dans les trois dimensions spatiales où les frontières peuvent être décrites sous forme de zones géométriques ;
  - places : relation de type graphe où les lieux sont reliés entre eux par des liens auxquels des informations peuvent être associées ;
- Les environnements sociaux permettent de réaliser des communications ou interactions directes entre les agents du SMA.

Cette approche de la dissociation entre les environnements, permet de fournir un comportement global pour les environnements de type question-réponse sur un ensemble de données. Par exemple, dans un environnement continu, les positions des agents sont des réels alors que dans un graphe, la position est désignée par le nœud du graphe où se trouve l'agent.

Les interactions des environnements avec les agents interviennent lorsque les agents demandent des informations ou des modifications aux environnements, par exemple: leurs coordonnées ou le rôle qu'ils peuvent avoir en fonction du type d'environnement. C'est une approche modulaire où les environnements sont fournis par notre spécification, mais peuvent être implémentés de plusieurs manières.

Les environnements que nous venons de décrire ne possèdent pas de comportement global complexe, ce sont simplement un ensemble de données auxquelles les agents accèdent. Ainsi, ce sont les agents qui dirigent les environnements, même si, comme nous le verrons dans la section suivante, les agents peuvent être contraints par les informations contenues dans l'environnement. Les agents ont pour rôle de se déplacer dans les environnements physiques, de communiquer dans les environnements sociaux et de gérer les interactions en modifiant les données de l'environnement. Par exemple, si deux agents doivent communiquer entre eux uniquement à partir du moment où ils sont assez proches l'un de l'autre, l'un des agents ou les deux doivent gérer la proximité de l'autre pour entamer une communication. Cependant, pour permettre la modélisation de dynamique aux environnements, nous développerons, par la suite, le modèle *perturbateur* qui a pour rôle de perturber les informations contenues dans l'environnement.

Pour simplifier la conception des modèles des agents, nous avons découpé l'agent en deux parties, le corps et la tête. Le corps réceptionne les informations des environnements ou interagit avec eux. La tête est connectée au corps et participe au comportement général de l'agent. La simplification du modèle agent intervient dans la partie « corps » de l'agent où tout peut être automatisé par notre spécification, aussi bien les connexions que les modèles. Pour cela, il faut définir trois concepts supplémentaires, les « déclencheurs », les « capteurs » et les « effecteurs ».

- le déclencheur est un modèle géré par l'environnement distribuant les messages à destination des agents. Dès qu'une information d'un environnement est modifiée, un déclencheur reçoit les informations et les propage aux agents reliés à lui via les capteurs ;
- les capteurs sont des entités entre les déclencheurs et les agents, ils permettent de

simplifier l'information fournie par l'environnement pour l'agent sur lequel il est connecté ;
– les effecteurs sont des modèles connectés en entrée aux agents et en sortie aux environnements. Ils permettent d'appliquer un changement d'information sur les environnements. Par exemple, l'effecteur *déplacement* permet de déplacer un agent dans un environnement physique.

Cette description succincte montre que la modularité fournit une simplification des modèles. Les entités laissées libres aux modélisateurs sont les comportements des agents, les déclencheurs, les capteurs et les effecteurs. Ces trois derniers sont des modèles très simples à développer dans le sens où ils sont spécialisés pour une seule opération afin de ne pas fusionner les comportements des agents et des capteurs dans une seule entité DEVS. La tête de l'agent reste cependant l'entité la plus complexe du système puisqu'elle est le point de fusion des interactions entre l'agent et les environnements.

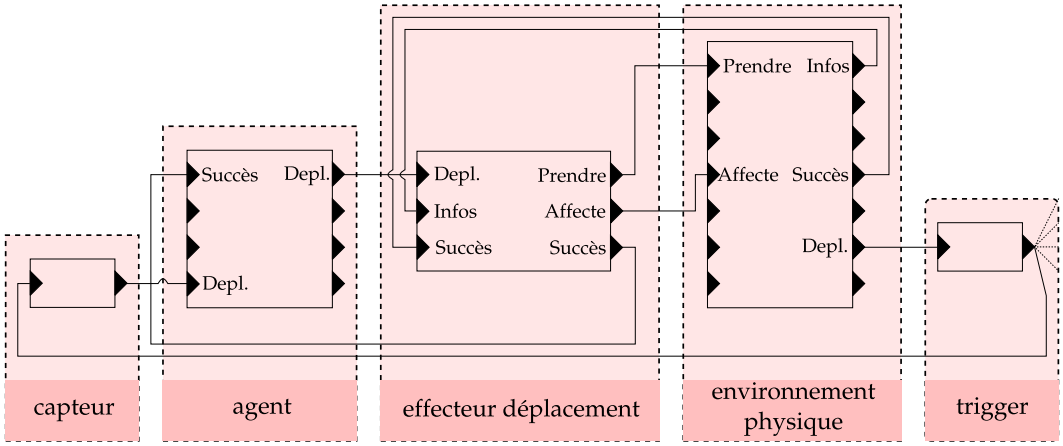


FIG. 3.1 – Cette figure montre un exemple de modélisation d'un agent se déplaçant dans un environnement.

La figure 3.1 est un exemple de déplacement d'un agent. Celui-ci pose un événement sur son port de sortie *move*. L'effecteur déplacement, connecté sur ce port, reçoit l'information et génère le déplacement en communiquant avec l'environnement, via la récupération des positions, vitesse et orientation de l'agent. C'est cet effecteur qui effectue le calcul de déplacement. Dès que l'environnement physique autorise ce déplacement, il pose sur un port de sortie qu'un déplacement vient d'avoir lieu. L'information est récupérée par un modèle *déclencheur* qui informe tous les capteurs connectés, eux-mêmes connectés directement sur le corps de l'agent.

Dans les parties suivantes, nous décrivons notre spécification de manière formelle en traitant les agents et les environnements.

3.2 Agents

Ce travail de spécification présente les bases de notre formalisation de SMA dans DEVS. Elle s'appuie sur les apports de notre simulateur développé dans le chapitre 2 et principalement sur les fonctions de simplifications de modèles comme les événements instantanés.

Cette spécification est le résultat de nos travaux sur les systèmes multi-agents débutés par É. Ramat [Ramat et Preux, 2003] sur la première version de VLE et R. Duboz [Duboz, 2004] sur les agents réactifs. Elle s'articule principalement autour de l'extension DS-DEVS présentée dans le chapitre 2 et de son modèle exécutif qui permet de modifier dynamiquement et formellement le graphe de connexions, la structure et l'apparition ou la destruction des modèles en cours de simulation.

3.2.1 Concepts autour des agents

Le comportement des agents est l'entité laissée libre au modélisateur, celui-ci, doit faire le choix du type de modèle DEVS à attacher. Il faut remarquer que les ports de ce modèle sont entièrement gérés par le SMA puisque les ports sont créés en fonction des corps et donc, des environnements dans lesquels l'agent se situe. Les ports sont nommés par les effecteurs et capteurs disponibles dans les environnements, par exemple, dans un environnement physique représentant un espace continu, plusieurs effecteurs sont fournis comme le déplacement, le changement d'orientation, l'observation des agents proches. Chacun de ces effecteurs, enregistrés auprès de l'environnement, génère un port en entrée pour les réponses et un port de sortie pour l'envoi des requêtes à l'environnement.

La figure 3.2 propose un exemple où un agent interagit avec deux environnements différents. La tête de l'agent, représentée par un modèle DEVS, voit son interface découpée en deux parties distinctes pour les communications avec les environnements. Le développement du comportement de l'agent doit alors se faire en relation avec les capteurs et effecteurs qu'il possède au travers de ses ports.

3.2.2 La tête, le comportement

Le comportement des agents est le travail principal pour le modélisateur dans cette spécification de SMA. En effet, c'est ce comportement qui a pour rôle d'agir avec les effecteurs des environnements et de réagir aux informations des capteurs et des déclencheurs. Le modélisateur a trois possibilités pour la modélisation du comportement de son agent :

– modèle couplé : le réseau de modèles est la méthode à utiliser lorsque le comportement est complexe et nécessite le couplage de formalismes comme les équations

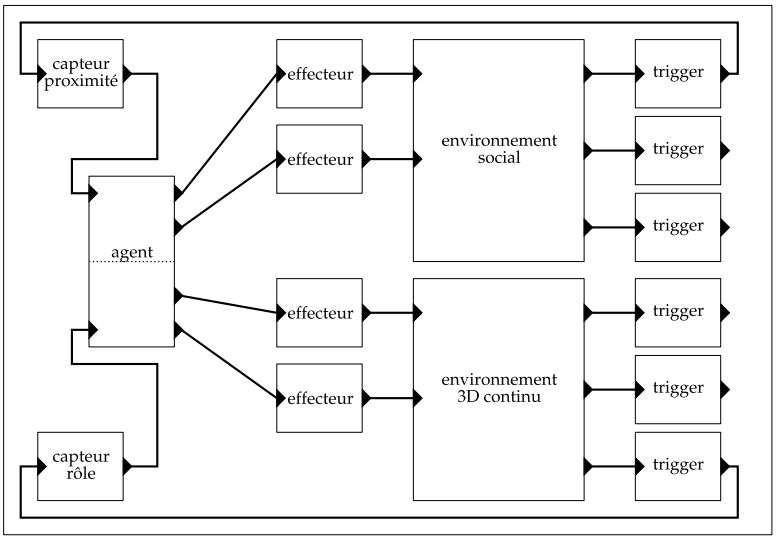


FIG. 3.2 – Exemple de modélisation où un agent interagit avec deux environnements différents.

différentielles, les réseaux de Petri, etc. Cette méthode permet la réutilisation de modèles, mais nécessite de connaître les environnements dans lesquels l'agent se trouve et elle ne peut être utilisée lorsque les agents changent d'environnement, le graphe de modèles étant figé.

– modèle atomique : le plus simple à développer, mais le moins flexible pour l'utilisation des différents formalismes. Ce type de comportement est principalement réservé aux agents simples modélisables sous forme d'automates à états finis. Cette méthode, à la différence des modèles couplés, permet le changement d'environnement mais complexifie le comportement du modèle atomique.
– modèle couplé dynamique : ces modèles sont à privilégier lorsque le changement de comportement ou d'environnement est nécessaire. En effet, lorsque les agents sont plongés dans des environnements sociaux, ils doivent pouvoir changer de rôle, c'est-à-dire, avoir la possibilité d'utiliser un nouveau comportement en manipulant les variables d'états des modèles, ou en modifiant la structure interne du graphe de modèle.

Dans le reste du document nous nous baserons sur un comportement modélisé à l'aide d'un *DSDEVSN*, les autres types de modélisation étant modélisables à l'aide d'un réseau dynamique.

Le comportement de l'agent est le résultat de ses décisions en fonction de ses perceptions, de son autonomie et des actions qu'il peut avoir dans les environnements dans lesquels il se trouve. Les quatre sections suivantes ont pour but de formaliser ces concepts.

3.2.3 Formalisation de l'entité agent

Dans un SMA, les agents ne sont pas isolés mais communiquent avec les autres agents et interagissent avec l'environnement via la manipulation de messages. Ces messages se traduisent dans le formalisme DEVS sous la forme d'événements, lesquels peuvent contenir les informations issues du message. Cette spécification propose d'utiliser le réseau de modèles dynamiques pour la représentation de la tête des agents. Nous formalisons cette entité sous la forme :

$$DSDEVSN_A = \langle X_A, Y_A, \chi_A, M_{\chi_A} \rangle$$

où X_A est l'union des ports pour les messages arrivant respectivement des autres agents et des environnements sous forme de capteurs ou de communications directes :

$$X_A = X_{A_{agent}} \cup X_{A_{env}}$$

Et Y_A, l'union des ports pour les messages de sorties à destination des effecteurs pour l'envoi de messages vers les autres agents, les effecteurs ou les environnements présents dans le SMA :

$$Y_A = Y_{A_{agent}} \cup Y_{A_{env}}$$

Le modèle exécutif M_{χ_A}, nommé χ_A, contenu dans le réseau de modèles dynamiques de l'agent $DSDEVSN_A$, permet de modifier la structure du réseau en ajoutant, supprimant des modèles ou des connexions.

3.2.3.1 La perception

La perception est spécifiée par l'arrivée d'événements externes sur les ports d'un modèle dont la conséquence est le changement d'état d'un ou plusieurs modèles de l'ensemble M_A. Par conséquent, nous considérons la perception en deux parties :

La première est définie par les événements arrivant sur $X_{A_{agent}}$ ou $X_{A_{env}}$ que nous appellerons « récepteurs ». La définition de cet ensemble spécifie le type des messages échangés par les agents. C'est l'aspect structurel de la perception.

La seconde considère l'aspect fonctionnel de la perception impliquant un changement d'état dans au moins un des modèles du modèle couplé.

Soit un agent A formalisé par un $DSDEVS$, la perception P_A de A est définie comme l'ensemble des fonctions de transitions externes telles que :

$$P_A = \{\delta_{ext_d} | \delta_{ext_d}(Q_d, (v, ip_d))\}$$

Avec :

$$\begin{cases} (d, ip_d) \in Z^A_{DSDEVSN_A,d}, \\ ip_d \text{ un port d'entrée du modèle } d \in D_A, \\ D_A \text{ l'ensemble des noms du réseau de modèles du modèle exécutif.} \end{cases}$$

L'évaluation d'une fonction de transition externe implique le changement de l'état de l'agent. L'état interne de l'agent peut être formalisé comme :

$$S_A = \overset{d \in \{D_A\}}{\times} S_d$$

En prenant en compte le fait que le domaine de définition des valeurs contenues dans les messages et donc les ports, n'est pas restreint, nous pouvons assimiler que tout type de message arrivant sur le modèle agent via les « récepteurs » sont des stimulus.

3.2.3.2 Action, pro-action et réaction

Dans les SMA, les actions des agents peuvent être de différents types comme le déplacement, la destruction, la modification d'un objet de l'environnement, le dépôt d'information etc. Ces actions se représentent, une nouvelle fois, sous forme d'événements générés par un agent, c'est-à-dire, l'émission d'un événement externe depuis la fonction de sortie λ des modèles atomiques. Comme pour la perception, l'action se distingue suivant les aspects structurels ou fonctionnels, où $Y_{A_{agent}}$ et $Y_{A_{env}}$ sont nommés « effecteurs ».

En considérant A un agent formalisé par un $DSDEVSN$, l'action F de A est définie comme l'ensemble des fonctions de sorties telles que :

$$F_A = \{\lambda_d | \lambda_d(S_d) = (v, op_d)\}$$

Avec :

$$\begin{cases} (d, op_d) \in Z^A_{d,DSDEVSN_A}, \\ op_d \text{ est un port de sortie du modèle } d \in D_A. \end{cases}$$

Le concept d'action peut être divisé en deux types, la pro-action et la réaction. Le concept de modèle pro-actif existe en DEVS, lorsqu'un modèle atomique, ou couplé, possède uniquement des ports de sortie, c'est-à-dire, $X = \emptyset$ et $Y \neq \emptyset$. De cette façon, le comportement de modèle est conduit par les fonctions de transitions internes. La pro-activité peut être vue comme l'ensemble des fonctions de sortie qui ont pour origine un modèle pro-actif.

Soit M_p l'ensemble des modèles pro-actifs composant un agent A, nous pouvons formaliser la pro-activité W_A comme :

$$W_A = \{\lambda_{d_p} | \lambda_{d_{p(S_{d_p})}} = (v, op_{d_p}))\}$$

Avec :

$\left(\begin{array}{l} (d_p, op_d) \in Z^A_{d,DSDEVSN_A}, \\ d_p \in M_p \text{ avec } M_p \in D_A \text{ l'ensemble des composants pro-actifs du modèle agent.} \end{array} \right.$

À ce niveau, la définition est incomplète. En effet, un modèle pro-actif peut englober un changement d'état dans un autre modèle composant l'agent. Une conséquence peut être que les derniers calculs d'une fonction de sortie sont reliés à l'arrivée d'un événement externe venant du modèle pro-actif. Ainsi, il peut y avoir un certain nombre de transition et de modèles, entre l'émission d'une valeur de sortie par un modèle pro-actif et l'émission d'une valeur de sortie de l'agent ; sans compter l'utilisation d'un graphe dynamique pour définir la pro-activité.

Le concept de réaction encourt les mêmes difficultés, néanmoins, comme pour la pro-activité, nous donnons une définition minimale de la réactivité des agents.

Soit A un agent de type $DSDEVSN$, une partie de la réaction de A peut être formalisée par l'ensemble des fonctions de sorties associées au modèle A et des états transitoires des modèles composants $d \in D_A$. En effet, les états transitoires autorisent de simuler l'émission d'une sortie sur une transition externe. Cette réaction à un événement d'entrée peut définir la réactivité R_A de A telle que :

$$R_A = \{\lambda_d | \lambda_d(s_d) = (v, op_d), ta(s_d) = 0\}$$

Avec :

$\left(\begin{array}{l} (s_d \in S_d | D \in D_A - \{M_p\}) \\ \text{l'ensemble des états des modèles composants qui ne sont pas pro-actifs,} \\ (d, op) \in Z^A_{d,\Delta_A} \\ \text{où } op_d \text{ est l'ensemble des ports de sortie du modèle } d_p \in D_A - M_p. \end{array} \right.$

L'action et la réaction sont considérées ici dans leurs définitions minimales.

3.2.3.3 Autonomie

Le concept d'action et de réaction est relié à celui d'autonomie, qui est plus que l'union de l'ensemble des transitions internes d'un modèle $DSDEVS$. En effet, le comportement

pro-actif d'un agent peut être la conséquence d'une décision d'un modèle pro-actif qui envoie des événements à d'autres modèles du $DSDEVSN$. En conclusion, l'ensemble des fonctions des modèles de $DSDEVS$ qui ne sont pas connectées aux entrées ou sorties du modèle agent définissent l'autonomie d'agent.

Soit un agent A de type $DSDEVS$, toutes les fonctions de transitions externes qui ne reçoivent pas d'événements internes du modèle couplé agent et toutes les fonctions de transitions internes définissent l'autonomie O de l'agent A :

$$O_A = \{\delta_{int_d} \cup \delta_{ext_d} | \delta_{ext_d}(S_d, (v, ip))\}$$

où :

$$\begin{cases} (v, ip_d) \notin Z^A_{\Delta_A, d} \text{ où } ip_d \text{ est un port d'entrée et } v \text{ sa valeur,} \\ S_d \text{ est l'ensemble des états des modèles de } DSDEVS_A \text{ avec } d \in D_A \end{cases}$$

3.2.3.4 Comportement

Les fonctions de transitions internes ou externes du modèle exécutif de modèle agent permettent de modifier la structures du réseau de modèles et ainsi participe à la définition du comportement de l'agent en modifiant les interactions entre les différents composants du modèle couplé et en spécifiant les sous-modèles. Nous considérons alors un changement de structure dans un modèle agent, comme un changement de comportement. Pour rappel, le modèle exécutif du réseau de modèles dynamiques agent $DSDEVSN_A$ est défini par :

$$M_{\chi_A} = \langle X_A, Y_A, S_A, \delta_{ext_A}, \delta_{int_A}, \lambda_A, ta_A \rangle$$

Un agent peut changer sa structure suivant les valeurs des événements internes ou externes qui modifie son état S_A. La spécification de ces changements peut être donnée par δ_{ext_A} et δ_{int_A}. Ceci implique qu'un agent peut effectuer des changements autonomes de son comportement. Néanmoins, le comportement des agents ne se définit pas uniquement par le changement de structure. Il se définit aussi par l'ensemble des fonctions de perceptions et d'actions. C'est la partie fonctionnelle du comportement des agents.

La perception, l'action, la pro-action et l'autonomie formalisent le comportement C_A de l'agent A qui peut être écrit comme :

$$C_A = F_A \cup P_A \cup O_A$$

Cette spécification de comportement signifie que toutes fonctions définies dans le réseau de modèles de l'agent spécifient son comportement.

3.2.4 Le corps, les capteurs et les effecteurs

Comme présenté dans les précédents paragraphes, les environnements sont les entités qui stockent toutes les informations relatives aux environnements eux-mêmes ainsi qu'aux agents. Lorsqu'un changement d'état intervient dans un environnement, par l'action d'un agent ou d'un phénomène global sur l'environnement par exemple, celui-ci émet des événements sur des ports de sorties attachés à des modèles de type déclencheur. Ces modèles ont pour rôle de fournir les informations reçues des événements aux agents connectés. Pour être connecté aux modèles déclencheurs, les modèles doivent en faire la demande auprès de l'environnement qui transmet la demande au modèle déclencheur.

Les capteurs utilisent les informations provenant d'un ou de plusieurs modèles déclencheurs pour générer des informations complexes à destination de la tête de l'agent afin de l'informer de changements apparus dans l'environnement. Les capteurs sont donc des entités à développer en fonction du SMA à modéliser. Dans les paragraphes suivants, nous nous attacherons a décrire, à l'aide d'un exemple, le comportement d'un capteur.

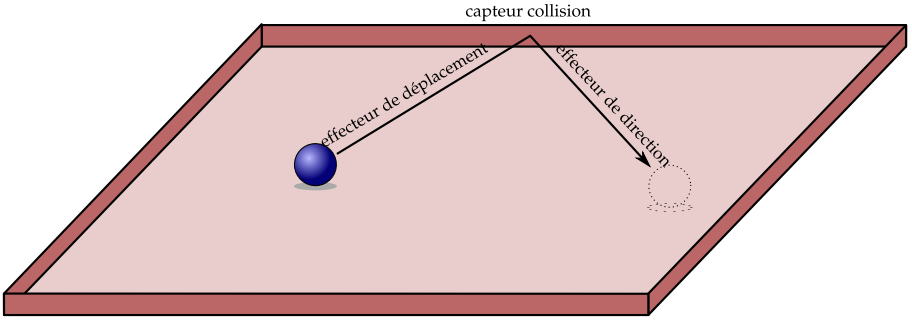


FIG. 3.3 – Cette figure illustre la présentation des modèles capteurs et effecteurs. Un agent boule, lancé sur un plateau par un effecteur, percute un bord par la détection d'un capteur, après demande, un effecteur change la direction.

La figure 3.3 représente un environnement en deux dimensions continues sur lequel une balle rebondit sur les frontières. Dans cette représentation, il n'existe qu'une seule classe d'agent et un seul environnement. Le but du modèle capteur est d'envoyer un événement à la date à laquelle la boule rebondira sur un côté. La boule est caractérisée par une position, une vitesse et une direction. Ce sont ces données dont le capteur a besoin pour effectuer son calcul avec la taille de l'environnement.

Dans notre spécification agent, ce système est modélisé par l'apparition d'un environnement, d'un agent, d'un déclencheur sur modification d'une des données de l'agent, d'un capteur qui calcule la date à laquelle la boule sort de l'espace. L'agent possède deux effecteurs, le premier pour se déplacer, le deuxième pour changer de direction lors de l'arrivée sur un bord. Une représentation simplifiée des modèles est fournie

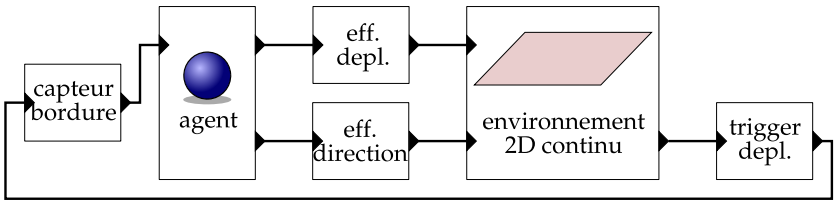


FIG. 3.4 – Modélisation du système précédent dans notre spécification agent avec un capteur de bordure et deux effecteurs pour le changement de direction et le déplacement.

sur la figure 3.4. Les sections suivantes montrent une définition des effecteurs et des capteurs pour ce système.

3.2.4.1 Les capteurs

Le capteur de détection de sortie a besoin de ces informations à plusieurs moments, lorsque le modèle agent lui en fait la demande, ou lorsque le modèle agent a décidé de modifier ses paramètres de vitesse ou de direction auprès de l'environnement. Le modèle capteur doit ainsi s'enregistrer à l'initialisation auprès du déclencheur de changement d'état du modèle boule. Le modèle capteur de détection de bord est défini par :

$$M_{capteur} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

où les entrées et sorties de ce modèle sont :

$$\begin{cases} X = (declencheur, ((x, y), \vec{d}, v, t)) \\ Y = (agent, t) \end{cases}$$

X définit le port connecté au déclencheur de modification de la boule. Le message contient les informations relatives à son changement d'état. Y caractérise la réponse du capteur à l'agent afin de lui fournir la date à laquelle il rencontre l'obstacle formé par la bordure de l'espace.

L'état du modèle est caractérisé par la taille de l'environnement, les variables P_1 et P_2 de type (x, y), et la phase dans laquelle il se trouve. La phase se décompose en deux parties, la première *idle* est l'état passif du modèle où il attend la venue d'un message du déclencheur. La deuxième phase, *calcul*, a lieu lors de la réception de ce message, c'est un état pour le calcul de la date de sortie de la boule de l'environnement.

$$S = (phase, P_1, P_2, t_r)$$

La taille de l'espace est, ici, initialisée dans la fonction *init* du modèle puisque la taille et la forme de l'espace ne changent pas dans cet exemple. Le comportement du modèle est, pour ce capteur, très simple puisqu'il n'effectue qu'un calcul lors de la réception d'un message externe :

$$ta(idle, P_1, P_2, t_r) = \infty$$

Lorsqu'un message est reçu, le calcul est effectué et la réponse envoyée immédiatement :

$$\begin{cases} \delta_{ext}(idle, P_1, P_2, t_r) \times (declencheur, ((x,y), \vec{d}, v, t) \rightarrow (calcul, P_1, P_2, t_r = f((x,y), \vec{d}, v, t)) \\ ta(calcul, P_1, P_2, t_r) = 0 \\ \lambda(calcul, P_1, P_2, t_r) : (t_r) \rightarrow Y \\ \delta_{int}(calcul, P_1, P_2, t_r) \rightarrow (idle, P_1, P_2, t_r) \end{cases}$$

où $f((x,y), \vec{d}, v, t))$ est la fonction de calcul de la sortie de l'agent de l'espace. Elle retourne la date à laquelle se produit la collision.

3.2.4.2 Les effecteurs

Un effecteur est un modèle dont le rôle est de modifier les informations contenues dans l'environnement à partir d'ordre qu'il reçoit de l'agent auquel il est attaché. De la même manière que pour le modèle capteur, un effecteur peut être interrompu si une des informations de l'agent a changé.

Comme précédemment, nous utilisons l'exemple de la balle sur une table pour décrire le fonctionnement des effecteurs. Nous ajoutons ainsi à l'agent un comportement qui simule un joueur par le biais d'un effecteur qui modifie la vitesse et l'angle de direction de la balle.

Les effecteurs sont des modèles DEVS dont le rôle est de modifier les attributs de l'environnement suivant les requêtes envoyées par les agents. Les effecteurs sont attachés au type d'environnement car ils ont besoin des informations relatives à l'environnement pour effectuer un changement. Dans cette section, nous nous attachons à développer l'effecteur de changement de direction. Le modèle de déplacement possédant le même fonctionnement, il n'est pas étudié ici. La structure de l'effecteur de changement de direction est un modèle atomique :

$$M_{direction} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

où l'interface du modèle est définie par les ports suivants :

$$\begin{cases} X = (agent, \vec{d_n}), (env, \emptyset) \\ Y = (env, \emptyset), (env?, agent) \end{cases}$$

Les ports d'entrées sont caractérisés par l'arrivée d'un ordre de changement d'orientation et pour la récupération des informations de l'environnement. Les deux sorties sont connectées à l'environnement, la première pour changer les informations de l'agent, la deuxième pour récupérer la direction de l'agent.

L'état du modèle $M_{direction}$ est défini par l'état courant de l'agent et par la *phase* dans laquelle il se trouve : *idle* lorsque le modèle est passif, *env*? pour l'envoi d'un événement instantané à l'environnement pour capturer l'orientation courante de la boule, *wait* quand le modèle est en attente de la réponse et *env* lorsque l'effecteur envoie les informations au modèle environnement.

$$S = (phase, \vec{d})$$

L'état initial du modèle est le mode passif dans lequel il attend un message de la tête de l'agent :

$$\begin{cases} S_{init} = (idle, \vec{d}) \\ ta(idle, \vec{d}) = \infty \end{cases}$$

Lorsque l'effecteur reçoit un ordre de changement de direction il se prépare à récupérer l'orientation réelle de l'agent via l'envoi d'un événement instantané à l'environnement :

$$\begin{cases} \delta_{ext}(idle, \vec{d}) \times (agent, \vec{d_n}) \rightarrow (env?, \vec{d} = \vec{d_n}) \\ ta(env?, \vec{d}) = 0 \end{cases}$$

L'événement instantané est envoyé à l'environnement, l'attente de la réponse est infinie même si, avec un événement instantané, le message de réponse est dans le même sac d'événements (cf. DEVS parallèle et les sacs d'événements décrits dans la partie 2.3).

$$\begin{cases} \lambda(env?, \vec{d}) : (agent) \rightarrow Y_{env?} \\ \delta_{int}(env?, \vec{d}) \rightarrow (wait, \vec{d}) \\ ta(wait, \vec{d}) = \infty \end{cases}$$

À la réception de la réponse de l'environnement, l'effecteur calcule la nouvelle orientation de l'agent et envoie la mise à jour à l'environnement. L'effecteur retourne alors dans son état passif :

$$\begin{cases} \delta_{ext}(wait, \vec{d}) \times (env, \vec{d_n}) \rightarrow (env, \vec{d} = \vec{d} + \vec{d_n}) \\ ta(env, \vec{d}) = 0 \\ \lambda(env, \vec{d}) : \vec{d} \rightarrow Y_{env} \\ \delta_{int}(env, \vec{d}) \rightarrow (idle, \vec{d}) \\ ta(idle, \vec{d}) = \infty \end{cases}$$

3.2.4.3 Comportement

Le cas de la boule sur un plateau présente un exemple très simple de notre spécification. Nous avons choisi cet exemple pour montrer un modèle agent avec un comportement simple avec ses modèles capteurs et effecteurs. L'agent boule est ici un modèle purement réactif où le comportement de l'agent ne change qu'à la réception d'un message d'un capteur.

Nous représentons le comportement du modèle agent par un simple modèle atomique dont les caractéristiques sont :

$$M_{boule} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

Les entrées sont définies par trois ports, le premier est relié au capteur de sortie de l'espace, les deux autres ports sont destinés aux résultats des actions des effecteurs de déplacement et de changement d'orientation. Les sorties du modèle M_{boule} sont connectées aux effecteurs et déclenchent des changements d'informations contenues dans l'environnement.

$$\begin{cases} X = (capteur, t_{bord}), (deplace), (oriente) \\ Y = (deplace), (change, \vec{d} \end{cases}$$

L'état de l'agent se caractérise uniquement par la phase dans lequel il se trouve et par une date t utilisée pour sauvegarder une date. Les phases du modèles sont :

- $init$: l'état initial de la boule, en attente de la réception d'une information du capteur ;
- $init_{mvt}$: un état transitoire entre l'initialisation et le mouvement ;
- $mouvement$: un état d'attente de la boule pendant lequel la boule se déplace ;
- $attente_{mvt}$: une demande de changement des informations contenues dans l'environnement ;
- $oriente$: la boule est sur un bord, elle change de direction et attend ;
- $attente_{ori}$: l'agent attend une réponse de l'effecteur pour reprendre son mouvement.

$$S = (phase, t)$$

L'état initial se traduit par une attente indéfinie de la boule, même si le capteur doit répondre immédiatement à t_0 d'après son modèle.

$$\begin{cases} S_{init} = (init, 0) \\ ta(init, t) = \infty \end{cases}$$

Lors de la réception de l'événement externe du capteur fournissant la date à laquelle la boule arrivera sur un bord, l'agent annonce via un événement interne que son prochain changement d'état aura lieu à cette date afin de prendre la décision d'une nouvelle direction.

$$\begin{cases} \delta_{ext}(init, t) \times (capteur, t_{bord}) \rightarrow (init_{mvt}, t_{bord}) \\ ta(init_{mvt}, t) = 0 \\ \delta_{int}(init_{mvt}, t) \rightarrow (mouvement, t) \\ ta(mouvement, t) = t \end{cases}$$

Dans ce système, aucun modèle ne peut envoyer d'information à cette étape. Le système, à l'arrivée sur le bord, doit demander le déplacement de l'agent à l'effecteur déplacement et se met en attente d'un accusé de déplacement.

$$\begin{cases} \lambda(mouvement, t) : (deplace) \rightarrow (deplace) \\ \delta_{int}(mouvement, t) \rightarrow (attente_{mvt}, t) \\ ta(attente_{mvt}, t) = \infty \end{cases}$$

Le déplacement peut être annulé si les informations ne sont pas correctes, auquel cas le modèle agent doit choisir une nouvelle action. Cet exemple étant relativement simple, les accusés de réception sont toujours positifs.

Une fois le déplacement effectué, la boule demande le changement d'orientation à l'effecteur prévu à cet effet. Une nouvelle fois, il se met en attente d'un accusé de changement d'orientation.

$$\begin{cases} \delta_{ext}(attente_{mvt}, t) \times (deplace) : (oriente, t) \\ ta(oritent, t) = 0 \\ \lambda(ortiente, t) : (oriente) \rightarrow (oriente, \vec{d}) \\ \delta_{int}(oriente, t) \rightarrow (attente_{ori}, t) \\ ta(attente_{ori}, t) = \infty \end{cases}$$

Une fois le changement d'orientation effectué, le modèle agent se met en attente d'une information du capteur pour effectuer son prochain déplacement.

$$\begin{cases} \delta_{ext}(attente_{ori}, t) \times (oriente) \rightarrow (init, t) \\ ta(init, t) = \infty \end{cases}$$

Le comportement de l'agent boule présenté dans cette partie est simple et ne montre pas la totalité des concepts de notre spécification agent. Cependant, il illustre le découpage des comportements de l'agent entre tête et corps simplifiant la complexité des modèles en leur définissant un comportement relatif à leurs fonctions. Enfin, la réutilisabilité des modèles s'en trouve augmentée puisque les modèles effecteurs et capteurs peuvent être employés dans d'autres SMA.

Lors de la traduction de cette spécification agent en version opérationnelle, un grand nombre de composants pourra être développé et fourni par la plate-forme, comme par exemple les effecteurs de déplacement, d'orientation, de changement de données publiques ainsi que les capteurs de sortie d'espace ou de perception d'entités. Le but est ici de laisser le modélisateur se focaliser sur le comportement de l'agent.

3.2.5 Le rôle

Comme mentionné précédemment, les méthodes comme GAIA [Zambonelli *et al.*, 2003] et AALAADIN [Ferber et Gutknecht, 1998] utilisent un haut niveau de métaphores pour la définition de systèmes multi-agents sociaux. Cependant, ces métaphores doivent trouver une correspondance dans les couches de la spécification afin de définir une spécification complète.

Le rôle est un des aspects fondamentaux des SMA [Odell *et al.*, 2003]. Le rôle d'un agent peut être vu comme une représentation abstraite des fonctions de l'agent, des services ou comme une identification dans un groupe [Gutknecht *et al.*, 2001]. Il fournit à l'agent une place au sein de l'organisation du SMA et il est associé à un ensemble de comportements [Wooldridge *et al.*, 2000]. Le rôle contraint ou manipule le comportement de l'agent en ajoutant ou soustrayant des buts, des désirs, des croyances ou des intentions. Les interactions sont alors vues comme un moyen pour les agents d'accomplir leurs rôles dans le système.

Le rôle est ainsi la conjonction d'un comportement particulier et la topologie d'interaction entre les agents d'un groupe. La spécification du rôle ne peut être associée seulement à l'agent mais aussi au groupe à l'intérieur duquel l'agent joue un rôle.

Dans notre spécification, un rôle est associé à un comportement de l'agent. Par exemple, si C_A est le comportement de l'agent A, C_A peut être modifié en ajoutant ou en supprimant des modèles ou des connexions dans la tête de l'agent, formée par un réseau de modèles dynamiques $DSDEVN$ et son modèle exécutif M_A.

L'état S_A du modèle M_A est défini par :

$$S_A = (X^A_{DSDEVSN_A}, Y^A_{DSDEVSN_A}, D^A, \{M^A_i\}, \{I^A_i\}, \{Z^A_{i,j}\}, V^A)$$

où D^A est l'ensemble des noms de modèles du réseau, M^A_i l'ensemble des modèles

composant le réseau $\forall i \in D^A$. Nous proposons de définir un rôle comme une des structures possibles du réseau de modèles, ainsi, un nom de rôle définit alors un état du modèle couplé de la tête de l'agent. Si Ω_A est l'ensemble des rôles possibles pour un agent A alors :

$$\Omega_A = \{\omega_i | 1 \leq i \leq n\}$$

où n est le nombre de rôle de Ω_A. La décision de changer de rôle peut être réalisée de manière autonome ou externe, suite à la demande d'un autre agent. Ceci implique qu'une transition interne ou externe permet de changer le rôle :

$$\begin{cases} \delta_{int_A} = (\{\ldots, \omega_i\}) \rightarrow (\{\cdots, \omega_j\}) \\ \delta_{ext_A} = (\{\ldots, \omega_i\}) \times (\omega_j, p) \rightarrow (\{\ldots, \omega_j\}) \end{cases}$$

Cependant, un changement de rôle n'est pas forcément attaché à un changement dans la composition du modèle couplé de la tête de l'agent. Il peut affecter simplement la topologie des modèles, ou simplement les états des modèles atomiques. Ainsi, la réalisation de méthodologie pour simplifier cette partie reste à développer puisqu'elle dépend des systèmes à modéliser.

La notion de groupe à laquelle est attachée la notion de rôle est prise en compte dans notre spécification par les environnements sociaux et sont étudiés dans les sections suivantes.

3.3 Environnements

Notre définition de l'entité environnement provient d'un des principaux problèmes des modèles atomiques DEVS : la complexité des modèles lorsque le graphe d'état du modèle est important. Une des solutions est d'utiliser la modularité en construisant des modèles à base de modèles simples. Nous définissons ainsi le modèle environnement comme une structure simple de type questions-réponses dont le graphe d'état est très simple afin de permettre la construction facile de modèles environnements. Cette coupure déporte les comportements complexes comme la détection des intersections, le déplacement d'agent dans d'autres modèles spécifique à ces actions.

Les travaux de J. C. Soulié avec les environnements multiples [Soulié, 2001] sont aussi une des approches retenues pour la conception de l'entité environnement. En effet, cette approche consiste à utiliser plusieurs environnements pour la représentation des différentes situations où l'agent peut évoluer, c'est-à-dire, les lieux physiques et sociaux où l'agent intervient de manière simultanée ou séquentielle. Ce concept est une notion que nous avons prise en compte pour définir notre entité environnement afin de simplifier

la construction d'environnement en fournissant uniquement les données dont les environnements possèdent les informations. Par exemple, les environnements physiques contiennent les informations géographiques des agents ; alors que les environnements sociaux vont contenir les graphes d'accointances des agents.

Les environnements, dans notre spécification, sont donc des environnements centralisés. Toutes les informations sont regroupées au sein d'une seule entité, mais aussi distribuées, puisque le modèle environnement peut être attaché à un automate cellulaire de type Cell-DEVS. Ainsi, nous avons décidé de définir deux classes d'environnements : physique et sociale. La première pourra être dissociée en plusieurs environnements, continus, discrets et sous forme de graphe. Le deuxième représente le groupe du concept AGR (Agent, Groupe, Rôle, [Ferber, 1995]). Dans la description des environnements réalisée dans la section 3.1.1.2 nous remarquons que l'autonomie de l'environnement est une des problématiques pour réaliser une formalisation complète. Or, dans notre définition d'environnement sans autonomie, nous allons définir des modèles « perturbateurs » qui, à l'instar des agents, vont permettre de modifier les caractéristiques des agents contenues dans le modèle environnement.

Dans cette section de chapitre, nous abordons le fonctionnement interne des environnements. Nous développons, en première partie, les éléments communs aux environnements physiques et sociaux avant d'entrer dans les détails de chacun. Enfin, nous proposons deux entités nécessaires aux environnements, les déclencheurs et les perturbateurs.

3.3.1 Formalisation générique

Les deux types d'environnements présentés dans la section précédente fonctionnent de la même manière : ce sont des modèles à comportement de type question-réponse, où seul le type d'information différencie les deux environnements. Les environnements physiques contiendront les informations de types géographiques comme les positions et vitesses des agents, alors que l'environnement social contient lui, des informations comme les rôles disponibles et les comportements associés.

Les agents disposent d'informations que l'on nomme « publiques », dans le sens où ces informations ne sont disponibles que depuis l'environnement pour tous les agents. Ces informations sont utiles pour proposer des informations globales à l'environnement et permettent de minimiser les échanges d'informations entre les agents.

Les environnements sont des structures dont le rôle est de stocker les informations des agents et de fournir un mécanisme de type questions-réponses aux requêtes des autres modèles. Ces modèles sont représentés sous forme de modèles couplés ou modèles atomiques. Leurs rôles sont simplement de stocker les informations des agents et des informations publiques de l'environnement lui-même. Dans la suite de ce document, nous formalisons l'environnement sous forme d'un modèle atomique :

$$M_{env} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{inst}, \delta_{con}, \lambda, ta \rangle$$

L'état de ce modèle générique, commun aux deux classes d'environnement, est défini par le couple de variables :

$$S = \{(phase, M)\}$$

où *phase* peut prendre la valeur *Idle* ou $(Q_{X_i} | i = 1 \dots n)$ avec n le nombre de ports d'entrées du modèle. *Idle* désigne l'état passif, les ports d'entrées X_i sont des questions ou requêtes potentielles sur l'environnement. M est l'ensemble d'informations publiques, la taille de l'espace, par exemple, pour les environnements physiques.

$$\begin{cases} X_{env} = X_{env_{eff}} \cup X_{env_{dyn}} \\ Y_{env} = Y_{env_{eff}} \cup Y_{env_{dclencheur}} \end{cases}$$

X_{env} est l'union des ports pour les messages arrivant des effecteurs, des agents et des modèles de comportements de l'environnement. Y_{env} est l'union des ports de sorties pour les messages de réponse à destination des effecteurs, ainsi que les messages vers les déclencheurs.

Le modèle $DEVS_{env}$ fonctionne en mode requête-réponse. Les requêtes peuvent être de deux types : une simple question ou une modification d'une des variables d'états du modèle. La formalisation de la question simple est réalisée à l'aide des événements instantanés sous la forme suivante :

$$\begin{cases} \delta_{inst} : (S_{eff_i}, M) \times X_{eff_j} \rightarrow (S_{eff_i}, M) \\ \delta_{inst} : (Idle, M) \times X_{eff_i} \rightarrow (Idle, M) \\ \lambda_{inst} : (S_j, m) : (S_j, M) \rightarrow Y_{eff_j} \end{cases}$$

On remarque, dans cette formalisation, que quel que soit l'état dans lequel se trouve le modèle, celui-ci peut répondre à la question demandée sans changer son état précédent, simplifiant ainsi le graphe d'état du modèle DEVS.

La formalisation de la modification d'une des variables du système est réalisée par l'arrivée, sur un port d'entrée, d'un événement provenant d'un effecteur, la réponse à la modification est instantanée :

$$\begin{cases} \delta_{ext} : (Idle, M) \times X_i \rightarrow (S_i, M) \\ ta(S_i, m) = 0 \end{cases}$$

Le message d'accusé est envoyé au port de sortie du même nom si la modification est réussie et une copie du message d'origine est posée sur le port de sortie à destination

du *déclencheur* affecté à l'action réalisée. L'état du modèle repasse alors en mode inactif, formalisé par les fonctions suivantes :

$$\begin{cases} \lambda(s_i, m) : (S_i, M) \longrightarrow Y_{env_{eff_i}} \\ \lambda(s_i, m) : (S_i, M) \longrightarrow Y_{env_{dclencheur_i}} \\ \delta_{int} : (S_i, M) \longrightarrow (Idle, M) \\ ta(Idle, m) = \infty \end{cases}$$

3.3.2 Environnements physiques

L'environnement physique est une entité dont le rôle est de contenir toutes les informations des entités sous forme de vecteurs positions et vitesses. Les informations que les entités laissent libres aux autres entités, ainsi que les informations relatives à l'environnement lui-même, comme la taille de l'espace et son nombre de dimension. L'environnement physique se représente comme un modèle atomique :

$$DEVS_{env.physique} = \langle X, Y, X_{init}, Y_{state}, S, \delta_{ext}, \delta_{int}, \delta_{init}, \delta_{state}, \delta_{inst}, \delta_{con}\lambda, ta \rangle$$

L'état de l'environnement physique est défini :

$$S = \{(phase, E, M)\}$$

où *phase* possède la même définition que celle de l'environnement générique. La variable M définit les informations de l'environnement :

$$M = \{C_M, P_M\}$$

$$\begin{cases} C_M \text{ un ensemble d'informations relatives à la représentation de l'espace,} \\ P_M \text{ l'ensemble de valeurs publiques de l'environnement.} \end{cases}$$

À ce niveau, la formalisation n'est pas complète et nécessite un approfondissement. En effet, la gestion des espaces physiques n'est décrite que via les variables C_M et P_M. Cependant, aucune information n'est fournie sur le type d'environnement et les spécificités de celui-ci. Nous distinguons trois types d'environnements physiques suivant la qualification de l'espace de l'environnement, discret, continu ou graphe dont les variables E et C_M dépendent.

3.3.2.1 Environnements discrets

Notre formalisation s'appuie, principalement, sur la réutilisabilité des modèles et leurs simplicités. Ainsi, nous utilisons l'extension CellDEVS de DEVS spécialisée dans la modélisation d'automate cellulaire.

E définit les informations des agents par rapport à l'environnement physique, c'est-à-dire, les coordonnées, directions, vitesses et les informations publiques :

$$E_{env.discret} = \{(C_E, D_E, V_E, P_E)\}$$

où :

$$\begin{cases} C_{E_i} \in \{C_E\} \text{ et } C_{E_i} \in \mathbb{Z}^n \text{ avec } n = \{1,2,3\} \\ \qquad \text{l'ensemble des coordonnées des entités dans un espace à 1,2 ou 3 dimensions ;} \\ \vec{D_{E_i}} \in \{\vec{D_E}\} \text{ et } \vec{D_{E_i}} \in \mathbb{Z}^n \text{ avec } n = \{1,2,3\} \\ \qquad \text{l'ensemble des vecteurs directions des entités ;} \\ V_{E_i} \in \{V_E\} \text{ et } V_{E_i} \in \mathbb{Z} \\ \qquad \text{l'ensemble des vitesses des entités ;} \\ \{P_{E_i}\} \in \{P_E\} \text{ et } \{P_{E_i}\} \in \mathbb{R}, \mathbb{N}, \mathbb{C}, \ldots \\ \qquad \text{les valeurs publiques des entités proviennent de divers domaines.} \end{cases}$$

L'espace C_M est formalisé, pour un exemple simple, par la taille, le nombre de cellules et leurs indices pour chaque axe de l'espace.

L'environnement discret possède une légère différence due à l'utilisation de l'extension Cell-DEVS pour représenter l'environnement. Certaines parties du corps de l'agent sont connectées directement aux cellules de l'automate cellulaire, comme l'effecteur de perturbation de la cellule. Cependant, l'autre partie de l'agent reste connectée à l'environnement pour la gestion des informations publiques comme, par exemple, l'effecteur de déplacement.

3.3.2.2 Environnements continus

Dans les environnements continus, les informations de position, vecteur vitesse se trouvent dans $\mathbb{R}^n$ avec $n \in \{1,2,3\}$ suivant la dimension de l'espace. Les caractéristiques des agents, publiées dans l'environnement, définissent la variable E :

$$E_{env.continus} = \{(C_E, D_E, V_E, P_E)\}$$

où :

$$\begin{cases} C_{E_i} \in \{C_E\} \text{ et } C_{E_i} \in \mathbb{R}^n \text{ avec } n = \{1,2,3\} \\ \quad \text{l'ensemble des coordonnées des entités dans un espace à 1,2 ou 3 dimensions ;} \\ \vec{D_{E_i}} \in \{\vec{D_E}\} \text{ et } \vec{D_{E_i}} \in \mathbb{R}^n \text{ avec } n = \{1,2,3\} \\ \quad \text{l'ensemble des vecteurs directions des entités ;} \\ V_{E_i} \in \{V_E\} \text{ et } V_{E_i} \in \mathbb{R} \\ \quad \text{l'ensemble des vitesses des entités ;} \\ \{P_{E_i}\} \in \{P_E\} \text{ et } \{P_{E_i}\} \in \mathbb{R}, \mathbb{N}, \mathbb{C}, \ldots \\ \quad \text{les valeurs publiques des entités proviennent de divers domaines.} \end{cases}$$

L'espace C_M est formalisé, dans un exemple très simple, par deux points définissant un cube, un carré ou une ligne suivant le nombre de dimension de l'espace. Cet exemple est le plus simple mais il permet de simplifier les calculs attachés aux modèles effecteurs et capteurs.

3.3.2.3 Environnements graphes

Dans un environnement de relation nœuds – connexions, les caractéristiques des entités sont en relation avec les nœuds. Ainsi, E est défini comme :

$$C_{M_{env.graphe}} = \langle M, C \rangle$$

où:

$$\begin{cases} M \text{ l'ensemble des places de l'environnement physique,} \\ C = \{(i,j) | i,j \in M, i \neq j\} \text{ l'ensemble des connexions reliant deux places.} \end{cases}$$

$$E_{env.graphe} = \{(C_E, D_E, P_E)\}$$

où :

$$\begin{cases} C_{E_i} \in \{C_E\} \text{ et } C_{E_i} \in C_M \\ \quad \text{l'ensemble des places sur lesquelles se trouve l'entité,} \\ D_{E_i} \in \{D_E\} \text{ et } D_{E_i} \in C_M \\ \quad \text{l'ensemble des directions vers les places distances,} \\ \{P_{E_i}\} \in \{P_E\} \text{ et } \{P_{E_i}\} \in \mathbb{R}, \mathbb{N}, \mathbb{C}, \ldots \\ \quad \text{les valeurs publiques peuvent prendre des valeurs dans divers domaines.} \end{cases}$$

Les modèles que nous venons de décrire sont limités à de simples structures géométriques sous la forme de lignes, rectangles ou de cubes suivant le nombre de dimensions de l'espace des environnements discrets ou continus. Cependant, des structures

plus complexes peuvent être implémentées puisque ce sont les effecteurs qui manipulent réellement les données. L'environnement ne sert que de structure de stockage de données. Une autre remarque peut porter sur les SMA où les agents doivent interagir directement avec les données comme, par exemple, la pose de phéromones des fourmis sur un sol. Lorsque l'environnement est de type automate cellulaire, le processus peut se représenter simplement. Dans le cadre d'environnements continus, le processus est plus complexe et nécessite des modèles de types *perturbateurs* modifiant les données publiques de l'environnement. Bien que nécessaire pour la spécification complète de SMA, nous limiterons nos exemples de formalisation aux modèles plus simples.

3.3.3 Environnements sociaux

Notre spécification agent propose d'associer les groupes et les environnements sociaux. Avant d'aborder les caractéristiques des environnements sociaux, nous développons la notion de groupe dans la SMA et leur relation avec les rôles des agents.

3.3.3.1 Groupe

Le terme de groupe définit un ensemble d'agent qui, en conjonction avec la définition de rôle, définit la topologie des interactions entre les rôles [Gutknecht *et al.*, 2001]. La description formelle d'un groupe nécessite la prise en compte des interactions entre les rôles. La formalisation du groupe se base sur celle des rôles proposés dans la section 3.2.5. Nous définissons alors g_{ω_i} comme l'ensemble des rôles qui interagissent avec le rôle ω_i :

$$g_{\omega_i} = \{\omega_j, \ldots, \omega_k\} | 1 \leq (i, j, k) \leq n$$

où ω_i est un rôle dans le groupe et n est le nombre de rôle du groupe. L'ensemble de tuple $g = \{(\omega_i, g_{\omega_i})\}$ définit les interactions entre agents. De plus comme les rôles sont joués par les agents, nous pouvons associer un agent avec un rôle sous la forme (A_i, ω_j). En conséquence, nous définissons $g' = \{(A_i, \omega_j)\}$ comme l'ensemble des types qui associe un agent à un rôle. Ainsi, nous pouvons définir une partie de l'état du modèle d'environnement physique comme :

$$S_G = \{g, g'\}$$

L'état de l'environnement social n'est pas complet puisqu'il manque les informations publiques des agents ainsi que ceux de l'environnement, s'il en possède.

Dans notre spécification un agent peut être membre de plusieurs groupes ou environnements physiques au même moment et peut donc créer des rôles ou des groupes. Pour

permettre la création de groupe, une structure nommée « organisation » est nécessaire. Elle sera étudiée dans la prochaine section.

Dans la section suivante, nous abordons la spécification des environnements sociaux.

3.3.3.2 Spécification

L'environnement social ou le groupe possède les mêmes caractéristiques que les environnements physiques en localisant, dans une seule entité, les informations des agents ainsi que les informations de rôles disponibles et les interactions entre ces rôles. Les informations stockées dans cet environnement sont les informations relatives aux graphes d'accointances entre les agents. La structure DEVS se représente sous la forme d'un modèle atomique qui communique avec un modèle exécutif pour gérer les communications entre agents. Le modèle environnement social est défini comme un modèle atomique dont la structure est :

$$M_{env.social} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{inst}, \lambda, ta \rangle$$

La définition de l'état S est :

$$S = \{phase, S_G, M\}$$

où *phase* utilise les mêmes valeurs définies pour le fonctionnement de l'environnement générique, $S_G = \{g, g'\}$ les rôles et les interactions, M les attributs publiques des agents et du groupe.

Le groupe ou l'environnement social est un modèle atomique de type question-réponse instantané. Il est défini comme un modèle passif et réagit lors de questions ou de modifications demandées par les effecteurs des agents. Les actions spécifiques de cet environnement sont de pouvoir assigner un rôle à un agent ou de lui changer son rôle. Nous formalisons l'association d'un agent à un rôle lors de l'arrivée d'un événement externe :

$$\delta_{ext}(phase, S_G, M) \times X_{ajouteragent}(A_i, \omega_j) \rightarrow (phase, S'_G, M)$$

où le nouvel état de S'_G, ajoute l'agent et le rôle à l'ensemble. Il est défini par :

$$S'_G = \{g, g' + (A_i, \omega_j)\}$$

L'événement externe transporte le nom du rôle ω_j et le nom de l'agent A_i. Si $\omega_j = \emptyset$, alors il faut que l'environnement fournisse un rôle à l'agent. Une politique doit alors être prise au sein du groupe.

La suppression d'un agent d'un groupe suit la même logique, c'est-à-dire, à la réception d'une demande de suppression du groupe, l'agent est simplement déconnecté des connexions vers les autres agents.

$$\delta_{ext}(phase, S_G, M) \times X_{retireragent}(A_i, \omega_j) \rightarrow (phase, S'_G, M)$$

où le nouvel état de S'_G, retire l'agent de l'ensemble. Il est défini par :

$$S'_G = \{g, g' - (A_i, \omega_j)\}$$

Lorsqu'un agent quitte un groupe, il n'a plus d'influence sur les rôles. Il est déconnecté des autres agents, mais peut, en fonction du comportement implémenté, s'ajouter à un autre groupe. Si un groupe perd son dernier agent, il doit simplement disparaître et c'est au modèle d'organisation qu'incombe la gestion des groupes.

Le modèle du groupe doit gérer l'ajout et la suppression du rôle. La formalisation de ces changements suit la même procédure, c'est-à-dire, à la réception d'un événement externe d'un agent ou de l'organisation, le groupe doit ajouter le rôle et sa topologie avec les autres rôles.

$$\begin{cases} \delta_{ext}(phase, S_G, M) \times X_{ajouterrole}(\omega_i, g_{\omega_i}) \rightarrow (phase, S'_G, M) \\ S'_G = \{g + (\omega_i, g_{\omega_i}), g')\} \end{cases}$$

La suppression de rôle nécessite une opération plus complexe, puisque les interactions entre les rôles doivent elles aussi être supprimées.

$$\begin{cases} \delta_{ext}(phase, S_G, M) \times X_{retirerrole}(\omega_i, g_{\omega_i}) \rightarrow (phase, S'_G, M) \\ S'_G = \{g - (\omega_i, g_{\omega_i}), g')\} \\ g_{\omega_j} = g_{\omega_j} - \omega_i \forall (\omega_j, g_{\omega j}) \in g \end{cases}$$

Dans cette partie, nous nous sommes attachés à l'état et au comportement du groupe. La section suivante aborde le modèle d'organisation qui gère et manipule les groupes et les rôles des groupes via l'utilisation du modèle exécutif M_χ du modèle couplé SMA.

3.3.3.3 Organisation

Dans les SMA, l'organisation se réfère souvent au niveau social, où, comme pour les groupes, l'organisation régule, contraint et guide le comportement des agents. J. J. Odell définit l'organisation comme un groupe dans lequel les rôles et les interactions apparaissent stables dans le temps [Odell *et al.*, 2003]. V. T. Da Silva et C. J. P. De Lucena proposent de définir l'organisation comme un ensemble de sous-organisations et

d'axiomes pour lesquels les agents et les sous-organisations doivent obéir. Un axiome établit les règles, les principes ou les lois et caractérise les contraintes globales du SMA, décrit dans [Silva et Lucena, 2004]. Ces auteurs attribuent un comportement à l'organisation caractérisé par les buts, les croyances et les actions. De notre point de vue, une organisation est un ensemble de règles et de relations entre les groupes au niveau du SMA. Comme pour la définition des groupes, nous n'attribuons pas de comportement autonome à l'organisation, laissant cette tâche aux effecteurs et perturbateurs.

Une organisation suit alors le même comportement de question-réponse que les environnements, elle est définie par :

$$M_{organisation} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{inst}, \lambda, ta \rangle$$

Le modèle d'organisation repose exactement sur le comportement des groupes en ajoutant la possibilité aux agents, via leurs effecteurs, de créer ou de retirer des groupes en posant des événements sur son port de sortie Y_χ connecté au modèle exécutif du SMA, en considérant celui-ci comme un réseau de modèle dynamique $DSDEVSN$.

Lors d'une demande d'ajout d'un agent dans un groupe, deux niveaux de contrôles sont alors établis. Le premier par le groupe, celui-ci accepte la création de l'ajout de la part de l'effecteur et envoie, via son déclencheur d'ajout d'agent, un message à l'organisation pour l'ajout de l'agent. Celui-ci peut alors accepter ou refuser la connexion, suivant le perturbateur qui lui est connecté. Le deuxième niveau de contrôle est donc réalisé par l'organisation. Si les deux entités, l'organisation et le groupe, ont des choix différents, c'est au modélisateur de faire le choix.

Nous ne spécifions pas le fonctionnement et l'état du modèle de l'organisation, la formalisation est similaire à ceux du groupe. La différence entre les fonctionnements des modèles provient des modèles externes de type perturbateur ou les effecteurs des agents. La communication entre les groupes et les organisations passe par l'utilisation des déclencheurs, dont nous proposons une définition dans la section suivante.

3.3.4 Déclencheur

Lorsqu'un environnement reçoit un message de changement d'une de ces variables d'états, par exemple, pour le déplacement d'une entité, celui-ci réalise la modification et génère deux événements de sortie. Le premier est à destination du modèle qui a émis le message, afin de lui confirmer le déplacement. Le deuxième événement est posé sur un port de sortie de l'environnement, afin que les entités à l'écoute de ce port aient accès aux changements effectués.

Ces informations ne sont pas disponibles directement, elles passent par un modèle nommé déclencheur, ou *trigger* en anglais, dont le comportement est similaire aux modèles *dispatchers* étudiés dans le chapitre précédent. Son principe est de relayer les évé-

nements de changements d'états de l'environnement aux agents. Pour être informés, les agents doivent en faire la demande à l'environnement par un message sur un de ces ports. Celui-ci transfère la demande aux déclencheurs associés à la demande. Le déclencheur prend alors en charge la modification du graphe de connexions du SMA pour attacher ou détacher le modèle agent sur un de ces ports de sorties.

Nous formalisons le modèle déclencheur sous la forme d'un modèle atomique :

$$M_{déclencheur} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$$

où les ports d'entrées X sont composés par : un port vers l'environnement lors de l'ajout de modèle en sortie de déclencheur, un port de données lorsque l'environnement envoie une information de changement d'état et un port vers le modèle exécutif pour la réception des modifications du graphe de connexions. Les ports de sorties reposent sur le même fonctionnement avec un port vers les modèles agents, un pour les acceptations de modèles et le dernier vers le modèle exécutif. Nous formalisons ces informations par :

$$\begin{cases} X = X_{data} \cup X_{env} \cup X_{exe} \\ Y = Y_{data} \cup Y_{env} \cup Y_{exe} \end{cases}$$

Pour simplifier l'écriture du modèle, nous décrivons ci-dessous les caractéristiques des événements, où L est la donnée de l'environnement vers les modèles à l'écoute du déclencheur et G l'information pour l'ajout ou la suppression de connexions du nouveau modèle à connecter ou à déconnecter.

$$\begin{cases} X = \{(data, L), (env, G), (exe, succes?)\} \\ Y = \{(data, L), (env, succes?), (exe, connexion)\} \end{cases}$$

L'état du modèle est défini par :

$$S = \{phase, L_D, L_G, L_{Ln}, L_{Ln}\}$$

où *phase* peut prendre la valeur *idle*, *donnee*, *graph* ou *ackgraph*. Le modèle est passif, *idle*, tant qu'il n'est pas perturbé par un événement externe. Lorsqu'il reçoit un événement sur X_{data}, celui-ci est envoyé au port Y_{data}. Les événements arrivant sur le port X_{env} sont envoyés à l'exécutif pour ajouter ou supprimer le nom du modèle passé en valeur du message. L_D et L_G représentent respectivement des listes de données reçues de X_{data} et des listes de changements de graphes de X_{env}. Le modèle *déclencheur* met en place une priorité des changements du graphe de connexions aux envois de données. Les variables L_{Dn} et L_{Gn} servent ici de tampon de messages lors de la phase d'attente de réussite de changement de graphe. L'état initial du modèle étant :

$$S_{init} = \{idle, \emptyset, \emptyset, \emptyset, \emptyset\}$$

À l'initialisation, le modèle est inactif et attend d'être perturbé par l'arrivée d'un message d'ajout ou de suppression de modèles de l'environnement sur le port X_{env} ou à l'arrivée d'un message X_{data} à envoyer aux modèles :

$$\begin{cases} \delta_{ext} : (idle, L_D, L_G, L_{Dn}, L_{Gn}) \times (data, L) \rightarrow (donne, L_D \cup L, L_G, L_{Dn}, L_{Gn}) \\ \delta_{ext} : (idle, L_D, L_G, L_{Dn}, L_{Gn}) \times (env, G) \rightarrow (graphe, L_D, L_G \cup G, L_{Dn}, L_{Gn}) \\ ta(donne, L_D, L_G, L_{Dn}, L_{Gn}) = 0 \\ ta(graphe, L_D, L_G, L_{Dn}, L_{Gn}) = 0 \end{cases}$$

Les messages sur le port environnement sont prioritaires, les changements de graphes doivent avoir lieu avant l'envoi des messages :

$$\begin{cases} \delta_{ext} : (donne, L_D, L_G, L_{Dn}, L_{Gn}) \times (data, L) \rightarrow (donne, L_D \cup L, L_G, L_{Dn}, L_{Gn}) \\ \delta_{ext} : (graphe, L_D, L_G, L_{Dn}, L_{Gn}) \times (data, L) \rightarrow (graphe, L_D \cup L, L_G, L_{Dn}, L_{Gn}) \\ \delta_{ext} : (donne, L_D, L_G, L_{Dn}, L_{Gn}) \times (env, G) \rightarrow (graphe, L_D, L_G \cup G, L_{Dn}, L_{Gn}) \\ \delta_{ext} : (graphe, L_D, L_G, L_{Dn}, L_{Gn}) \times (env, G) \rightarrow (graphe, L_D, L_G \cup G, L_{Dn}, L_{Gn}) \end{cases}$$

Une fois les messages externes réceptionnés, l'ensemble des messages de changement de graphes sont envoyés au modèle exécutif pour ajouter ou supprimer les connexions. Le modèle est alors en attente.

$$\begin{cases} \lambda(graph, L_D, L_G, L_{Dn}, L_{Gn}) : (L_G) \rightarrow Y_{exe} \\ \delta_{int} : (graph, L_D, L_G, L_{Dn}, L_{Gn}) \rightarrow (ackgraph, L_D, \emptyset, L_{Dn}, L_{Gn}) \\ ta(ackgraph, L_D, L_G, L_{Dn}, L_{Gn}) = \infty \end{cases}$$

Les messages de l'environnement reçus pendant la phase d'attente du message de réception de l'exécutif sont stockés dans les variables L_{Dn} et L_{Gn}.

$$\begin{cases} \delta_{ext} : (ackgraph, L_D, L_G, L_{Gn}, L_{Gn}) \times (data, D) \rightarrow (ackgraph, L_D, L_G, L_{Dn} \cup D, L_{Gn}) \\ \delta_{ext} : (ackgraph, L_D, L_G, L_{Dn}, L_{Gn}) \times (env, G) \rightarrow (ackgraph, L_D, L_G, L_{Dn}, L_{Gn} \cup G) \\ ta(ackgraph, L_D, L_G, L_{Dn}, L_{Gn}) = 0 \end{cases}$$

Dès que le modèle est perturbé par le message de réception de l'exécutif sur X_{exe}, le modèle envoie la liste des messages reçus :

$$\begin{cases} \delta_{ext} : (ackgraph, L_D, L_G, L_{Dn}, L_{Gn}) \times (exe, succes?) \rightarrow (ackgraph, L_D, L_G, L_{Dn}, L_{Gn}) \\ ta(ackgraph, L_D, L_G, L_{Dn}, L_{Gn}) = 0 \end{cases}$$

Les données reçues sont envoyées aux modèles à l'écoute du port Y_{data}. L'état du modèle est restauré en fonction des éléments reçus pendant l'état *ackgraph*.

$$\begin{cases} \lambda(ackgraph, L_D, L_G, L_{Dn}, L_{Gn}) : (L_D) \rightarrow Y_{data} \\ \delta_{int} : (ackgraph, L_D, L_G, \emptyset, \emptyset) \rightarrow (idle, \emptyset, \emptyset, \emptyset, \emptyset) \\ ta(idle, L_D, L_G, L_{Dn}, L_{Gn}) = \infty \\ \delta_{int} : (ackgraph, L_D, L_G, L_{Dn}, \emptyset) \rightarrow (donne, L_{Dn}, \emptyset, \emptyset, \emptyset) \\ ta(donne, L_D, L_G, L_{Dn}, L_{Gn}) = 0 \\ \delta_{int} : (ackgraph, L_D, L_G, \emptyset, L_{Gn}) \rightarrow (graphe, \emptyset, L_{Gn}, \emptyset, \emptyset) \\ ta(graphe, L_D, L_G, L_{Dn}, L_{Gn}) = 0 \\ \delta_{int} : (ackgraph, L_D, L_G, L_{Dn}, L_{Gn}) \rightarrow (graphe, L_{Dn}, L_{Gn}, \emptyset, \emptyset) \\ ta(graphe, L_D, L_G, L_{Dn}, L_{Gn}) = \infty \end{cases}$$

Si aucun événement de changement d'état n'a été reçu pendant le cycle, les messages sont envoyés aux agents :

$$\begin{cases} \lambda(donnee, L_D, L_G, L_{Dn}, L_{Gn}) : (L_D) \rightarrow Y_{data} \\ \delta_{int} : (donne, L_D, L_G, L_{Dn}, L_{Gn}) \rightarrow (idle, \emptyset, \emptyset, \emptyset, \emptyset) \\ ta(idle, L_D, L_G, L_{Dn}, L_{Gn}) = \infty \end{cases}$$

Cette section nous a permis de définir le fonctionnement interne du modèle déclencheur dont le rôle est d'avertir les modèles sur lesquels il est connecté des informations de changement d'état de l'environnement. Le comportement du modèle déclencheur est proche de celui du modèle dispatcher étudié dans le chapitre précédent puisqu'il gère le graphe de connexions de modèles pour ajouter ou supprimer des connexions entre lui et les agents.

Nous pouvons remarquer que l'environnement n'est utilisé que pour transférer les demandes des modèles d' « écoute de déclencheur » et ses changements d'états. Les déclencheurs auraient pu être fusionnés avec le fonctionnement de l'environnement. Nous avons décidé de découper le modèle environnement en deux types d'entités, le premier gère l'ensemble des données, le deuxième gère les événements de sorties afin de simplifier l'écriture de nouveaux modèles de type environnement, dans un soucis de simplification des comportements de modèles et permettre l'écriture de déclencheur plus complexe qui modifierait les données qu'il reçoit, par exemple.

Dans la sous-section suivante, nous décrivons l'entité perturbateur dont le rôle est de manipuler les informations de l'environnement, c'est-à-dire, ajouter un comportement à l'environnement.

3.3.5 Perturbateur

Dans notre spécification, seuls les agents ont les outils et modèles nécessaires pour manipuler les informations publiques des environnements ou du graphe d'interactions entre modèles. Cependant, les systèmes à modéliser demandent souvent de bénéficier d'outils de perturbation de l'environnement. Par exemple, lors de la modélisation de vent dans un environnement physique discret, la modélisation de cet aspect peut déplacer les agents de cellules en cellules suivant un pas de temps constant. Pour permettre le modélisation de ce type de processus, nous introduisons le concept de « perturbateur ».

Le perturbateur est une entité DEVS connectée au modèle environnement. Il a pour rôle de modifier les informations contenues dans l'environnement en suivant sa propre dynamique. Afin de pouvoir interagir plus facilement, il est connecté en entrée aux modèles déclencheurs dont les informations l'intéressent. La formalisation de ce modèle est, tout comme le modèle agent dépendant du comportement, à définir. Il peut être représenté sous la forme d'un modèle atomique, couplé ou dynamique. Ainsi, sa définition ne peut être définie de manière complète :

$$M_{perturbateur} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$$

De cette structure de modèle atomique, nous pouvons définir les ports d'entrée et de sortie comme l'ensemble des ports vers les modèles déclencheurs et des capteurs ainsi que les ports en direction des effecteurs ou des ports de l'environnement. Nous représentons les ports sous cette définition :

$$\begin{cases} X_{perturbateur} = X_{declencheur} \cup X_{capteurs} \cup X_{M_\chi} \\ Y_{perturbateur} = Y_{environnement} \cup Y_{effecteurs} \cup X_{M_\chi} \end{cases}$$

Les ports X_{M_χ} et Y_{M_χ} représentent les connexions du modèle scénario vers le modèle exécutif M_χ du modèle couplé SMA.

De la même manière que pour les agents, les modèles de type perturbateur peuvent se connecter aux modèles effecteurs et aux capteurs afin d'exploiter les mêmes modèles développés pour les corps des agents.

3.4 Système multi-agent

Dans notre spécification, un système multi-agent est représenté par un modèle couplé dynamique *DSDEVSN* où toutes les entités, ainsi que les interactions entre les agents et les environnements, peuvent être modifiées par le modèle exécutif. Cependant, dans certains cas, il peut paraître nécessaire de coupler le SMA avec d'autres formalismes

afin de définir des systèmes plus complexes, ou mieux adaptés aux contextes de la problématique traitée.

Le SMA étant un paradigme, il peut sembler étonnant de vouloir le coupler avec un autre formalisme tel que des équations différentielles. En effet, le formalisme à coupler doit communiquer avec le SMA. Or, le SMA doit pouvoir fournir l'interface de ports pour donner la possibilité d'être perturbé, via les environnements ou les agents. Cependant, des systèmes à échelle de temps différents peuvent permettre de réaliser un couplage SMA, équations différentielles, comme par exemple dans les travaux de R. Duboz [Duboz, 2004].

Dans les prochaines sections, nous nous attachons à la spécification du modèle couplé SMA avec ses entrées et sorties, nous formulons un exemple de couplage de SMA avec un modèle externe. En dernière partie, nous développons un modèle dont le rôle est de réaliser un scénario de SMA en perturbant le fonctionnement du modèle SMA.

3.4.1 Entrées et sorties

Le modèle SMA est le modèle couplé qui englobe la totalité des modèles présentés dans les sections précédentes, c'est-à-dire, les corps et têtes des agents, les environnements physiques et sociaux. C'est un réseau de modèle dynamique défini par :

$$DSDEVS_{N_{SMA}} = \langle X, Y, \chi, M_\chi \rangle$$

Cependant, pour permettre le couplage de SMA avec d'autres modèles dans un modèle couplé d'une hiérarchie supérieure, il faut définir les entrées et sorties du modèle SMA qui représentent l'interface fonctionnelle de modèle DEVS.

3.4.1.1 Sorties

La définition des sorties des SMA doit permettre de récupérer les informations des modèles inclus dans le SMA. Dans notre spécification, les modèles environnements possèdent toutes les informations publiques des agents et des environnements physiques ou sociaux. Nous proposons de définir comme sortie du modèle SMA l'ensemble des sorties des modèles environnements.

Dans notre spécification, ce sont les déclencheurs qui manipulent les sorties des environnements. Nous attachons les ports de sorties des déclencheurs aux ports de sortie du modèle couplé SMA :

$$Y_{sma} = \{(Y_{M_{declencheur}})\}$$

où l'ensemble $Y_{M_{declencheur}}$ est défini comme l'ensemble des ports de sorties des modèles déclencheurs tels que :

$$\begin{cases} Y_{M_{declencheur_i}} = (out, v) \text{ le port de sortie } out \text{ d'un modèle déclencheur dont la valeur est } v, \\ Y_{M_{declencheur_i}} \in M_{declencheur_i} \text{ le port appartient au modèle déclencheur,} \\ M_{declencheur_i} \in D_{sma} \text{ le modèle déclencheur appartient au modèle couplé SMA.} \end{cases}$$

où $Y_{M_{declencheur_i}}$ est un port de sortie sur lequel est connecté le port de sortie Y_{out} du modèle déclencheur $M_{declencheur_i}$ appartenant au modèle couplé SMA.

Nous pouvons remarquer que si le modèle externe a besoin de s'appuyer sur une donnée interne d'un agent, celui-ci doit obligatoirement fournir cette variable de manière publique, afin que celle-ci puisse être manipulée et transférée au modèle couplé via les déclencheurs. Cette technique peut entraîner une baisse des performances si cette variable d'état est utilisée souvent en écriture.

Dans cette version de spécification nous ne définissons pas d'autre moyen d'accéder à ces données, cependant, d'autres techniques pourront être intégrées comme des ports spécifiques à tous les modèles pour les questionner sur une variable d'état.

3.4.1.2 Entrées

Le rôle des entrées dans un modèle atomique DEVS est de perturber le fonctionnement du modèle avec l'arrivée d'un événement externe sur un de ces ports d'entrées. Le modèle peut alors soit refuser de prendre en compte l'événement, soit modifier son graphe d'état ou ses variables d'état pour prendre en compte la perturbation.

Dans le cadre d'un modèle couplé de type SMA, la question que l'on se pose est pourquoi perturber un SMA qui pourrait de lui même se créer ses propres perturbations ? Que faut-il perturber, la tête ou les corps des agents, les environnements ?

Nous avons choisi, dans cette première spécification de SMA, de permettre la perturbation des modèles via les modèles de types perturbateurs des environnements physiques ou sociaux. En effet, dans notre spécification, ce sont les seules entités laissées complètement libres de développement au modélisateur. Celui-ci peut alors ajouter, créer des entités, ou intervenir dans les variables publiques contenues dans l'environnement.

Nous définissons les entrées d'un SMA comme l'ensemble des modèles perturbateurs de tous les environnements disponibles dans ce SMA :

$$X_{sma} = \{(X_{M_{perturbateur}})\}$$

où l'ensemble $X_{M_{perturbateur}}$ est défini comme l'ensemble des ports d'entrées des modèles perturbateurs tels que :

$$\begin{cases} X_{M_{perturbateur_i}} = \{(p,v)\} \text{ l'ensemble des ports et valeurs d'entrées ;} \\ X_{M_{perturbateur_i}} \in M_{perturbateur_i} \text{ les ports appartiennent au modèle perturbateur ;} \\ M_{perturbateur_i} \in D_{sma} \text{ le modèle perturbateur appartient au modèle couplé SMA.} \end{cases}$$

où $Y_{M_{declencheur_i}}$ est un port de sortie sur lequel est connecté le port de sortie Y_{out} du modèle déclencheur $M_{declencheur_i}$ appartenant au modèle couplé SMA.

3.4.2 Couplage SMA

L'un des buts de ce manuscrit et de nos travaux au sein de l'équipe est de fournir une formalisation, les méthodes et les outils afin de faciliter le couplage de modèles ou de formalismes hétérogènes. Le formalisme DEVS apporte un début de réponse avec des traductions en DEVS de formalismes tels que les travaux de Kaufman avec QSS [Kofman et Junco, 2001] et QSS2 [Kofman, 2002] pour la résolution d'équations différentielles et K. Jensen avec les réseaux de Petri [Jensen, 1997] ainsi que les travaux effectués sur les capsules de compatibilité DEVS [Quesnel *et al.*, 2004a]. Le couplage de modèles hétérogènes permet de simplifier le développement de modèles complexes en utilisant les formalismes spécifiques lorsqu'ils sont requis et en utilisant la composition de modèles simples ou complexes afin de réaliser des modèles avec une plus grande expressivité.

Dans la première section, nous nous attachons à développer les entrées et sortie d'un SMA. En deuxième partie nous étudions le couplage de notre SMA avec un modèle d'équation différentielle à travers l'exemple de R. Duboz sur un copépode. Enfin, en dernière partie, nous développons un modèle non spécifique au SMA mais qui apporte de plus grandes possibilités en terme de modélisation.

Dans son travail de thèse, R. Duboz [Duboz, 2004] propose le couplage d'un système multi-agent avec un système d'équations différentielles sur le sujet d'un modèle proie-prédateur entre des copépodes[2] et des phytoplanctons[3]. Le SMA propose de réaliser des simulations de quelques secondes simulées d'agents copépodes dans un espace où se trouvent des phytoplanctons. La simulation permet de fournir une information de vitesse de consommation de phytoplanctons par les copépodes. Cette information est nécessaire aux équations différentielles fonctionnant sur une taille de population nettement supérieure pour de plus grandes périodes de temps. Le modèle de résolution d'équation différentielle affecte alors une valeur pour la ré-initialisation du SMA afin

[2] Les copépodes sont des animaux zooplanctoniques.
[3] Le phytoplancton est une algue microscopique.

que celui-ci ajoute ou retire le nombre de copépodes ou phytoplanctons. Le couplage introduit ici une nouvelle problématique de gestion d'espace temporel différent.

La figure 3.5 est un exemple d'implémentation du système évoqué précédemment. Il représente de manière épurée, le modèle couplé et ses modèles composants dont les types ont été présentés tout au long du chapitre. On retrouve également un modèle de résolution d'équations différentielles ayant un port d'entrée et un port de sortie pour l'initialiser et capturer le résultat de la résolution. Enfin, deux modèles atomiques, plus simples, apparaissent. Il s'agit d'un modèle de traduction d'information provenant du SMA et en direction du SMA. En effet, le moteur de résolution numérique a besoin d'une quantité et non des informations produites par les déclencheurs. Un calcul est donc nécessaire pour lui permettre de recevoir des informations compatibles. Le modèle de traduction de sortie possède le même comportement en proposant un calcul simple du nombre de copépodes et de phytoplanctons à placer dans l'environnement lors de la ré-initialisation du SMA par le modèle de perturbation.

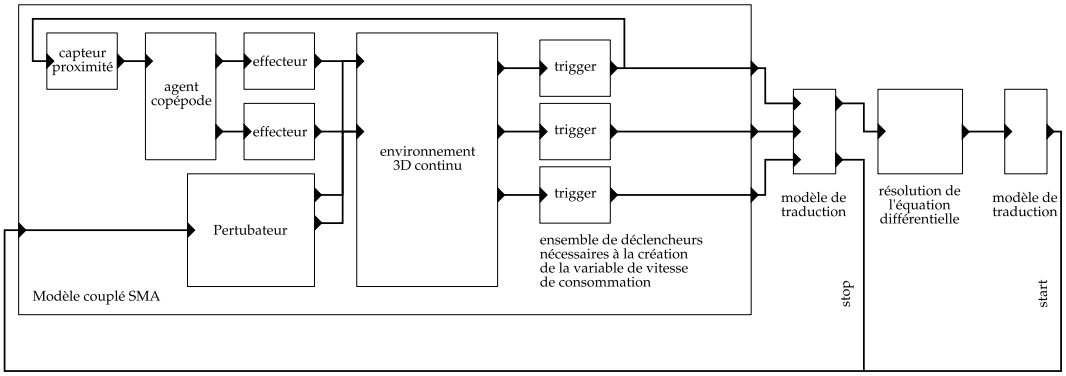


FIG. 3.5 – Exemple de traduction de l'exemple du système proie-prédateur basé sur les travaux de R. Duboz [Duboz, 2004] et traduit dans notre spécification agent.

Dans l'exemple, le modèle agent fournit les informations au moteur de résolution d'équations différentielles avec un pas de temps de l'échelle des quelques jours. La partie SMA fonctionne sur un espace de temps limité à quelques secondes. Le travail sur le temps doit être effectué par les modèles perturbateurs et les traducteurs. Le premier, à chaque ré-initialisation du SMA doit initialiser les agents à des dates égales à l'événement reçu. La partie de l'état de l'environnement physique attaché au modèle copépode A est défini par :

$$S_{env_{copepod1}} = ((x_A, y_A, z_A), \vec{d_A}, v_A, t_A = 0)$$

Pour rappel, t_A est la date à laquelle le copépode se trouvait lorsqu'il était sur la position (x_A, y_A, z_A). À l'initialisation de la simulation, la variable t_A, ainsi que toutes celles des autres agents, est initialisée à 0. Lorsque la date envoyée des déclencheurs au traducteur est supérieure au seuil de quelques secondes, celui-ci demande l'interruption

de la partie SMA, via un message *stop*, afin d'éviter de réaliser des calculs non nécessaires. Le modèle perturbateur reçoit ce signal et supprime toutes les entités actives du SMA. Un fois que le moteur de résolution d'équation a fini ses calculs, il envoie au deuxième modèle de traduction la demande de ré-initialisation du SMA, via le message *start* au déclencheur. Celui-ci doit restaurer l'ensemble du SMA avec des dates de positions initialisées au temps courant.

$$S_{env_{copepod1}} = ((x_A, y_A, z_A), \vec{d_A}, v_A, t_A = t_{courant})$$

Cette technique impose au modèle perturbateur une gestion complète de l'environnement et des entités attachées. C'est une réponse partielle au problème d'échelle de temps [Allen et Starr, 1982].

Le but de cette section n'est pas de fournir la définition complète des entités de ce SMA, mais de montrer comment exploiter le modèle couplé SMA pour le coupler avec d'autres modèles en manipulant deux échelles de temps différentes.

3.4.3 Scénarios

La classe de modèles *scénario* possède un comportement proche du modèle perturbateur étudié dans la section 3.3.5. Son rôle est de manipuler les modèles à l'échelle du SMA et non plus des environnements. Comme son nom l'indique, le modèle scénario doit permettre de perturber le SMA par la modification des environnements ou des connexions entre environnements. Une nouvelle fois, le modèle scénario peut être modélisé sous la forme d'un modèle DEVS atomique en communication avec le modèle exécutif du modèle couplé dynamique du SMA.

$$M_{scenario} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{inst}, \lambda, ta \rangle$$

De cette structure, nous pouvons définir les ports :

$$\begin{cases} X_{scenario} = X_{declencheurs} \cup X_{capteurs} \cup X_{M_\chi} \\ Y_{scenario} = Y_{environnement} \cup Y_{effecteurs} \cup Y_{M_\chi} \end{cases}$$

Le modèle scénario est connecté sur les ports d'entrées de tous les déclencheurs de tous les environnements contenus dans le SMA étudié.

$$\begin{cases} X_{declencheurs} = \{X_{declencheur_i} | X_{declencheur_i} \in M_{declencheur_i}\} \\ X_{capteurs} = \{X_{capteur_i} | X_{capteur_i} \in M_{capteur_i}\} \end{cases}$$

où les modèles $M_{declenheur_i}$ et $M_{capteur_i}$ sont des modèles déclencheurs et capteurs, contenus dans le SMA. De même pour les sorties, le modèle scénario est connecté à l'ensemble des effecteurs de l'environnement :

$$\begin{cases} Y_{environnements} = \{X_{environnemt_i} | X_{environnement_i} \in M_{environnement_i}\} \\ X_{effecteurs} = \{X_{effecteur_i} | X_{effecteur_i} \in M_{effecteur_i}\} \end{cases}$$

Les ports X_{M_χ} et Y_{M_χ} représentent les connexions du modèle scénario vers le modèle exécutif M_χ du modèle couplé SMA.

Le modèle scénario n'est pas nécessaire au fonctionnement du SMA, il permet simplement d'ajouter des possibilités de modélisation de système. Dans notre description il apparaît comme modèle afin de planifier des tâches comme l'ajout ou la suppression d'agent, le changement de comportement des perturbateurs des environnements. Le modèle scénario permet simplement d'augmenter les capacités de modélisation à travers notre spécification agent.

3.5 Conclusion

Ce chapitre a donné lieu à une proposition de formalisation des systèmes multi-agents en utilisant la théorie de la modélisation et la simulation et en exploitant, au mieux, les concepts de la systémique et de DEVS avec la composition et la hiérarchisation de modèles. Les particularités de cette spécification sont les environnements multiples avec le découpage des agents. Les environnements sont découpés en deux catégories, physiques ou sociaux, afin de mieux définir leurs rôles et de ne pas complexifier les graphes d'états de ces modèles. L'autonomie des environnements est restreinte à une simple interaction de type question-réponse avec les agents. Le découpage des agents sépare les concepts du comportement avec les entités, effecteurs et capteurs des environnements.

Cette spécification offre de multiples avantages de modélisation de SMA avec le formalisme DEVS. L'un d'eux est la simplification des environnements, permettant d'exploiter au mieux les concepts de DEVS comme les *wrapper* afin de pouvoir modéliser les environnements à l'aide de bases de données ou de systèmes d'information géographique. De plus, si la charge d'information est trop importante pour une structure informatique unique, les modèles environnements peuvent être développés sous forme de modèles couplés afin de répartir la charge sur plusieurs unités de calcul. Enfin, la réutilisabilité des modèles est aussi un point important de notre spécification. Il est exploité avec les environnements et les objets de types effecteurs, capteurs, déclencheurs. Ceux-ci forment un ensemble de modèles réutilisables par d'autres SMA.

Ce travail de recherche offre de nombreuses perspectives, en particulier lors du couplage de systèmes multi-agents avec d'autres formalismes où la notion de temps diffère

de manière importante. Par exemple, le concept du « holon », où l'idée principale est que chaque chose est non seulement un tout, mais fait également partie d'un plus grand tout, définissant ainsi un « hol-on ». Cette approche peut aider à la compréhension de systèmes complexes en définissant par exemple, le comportement des agents par des modules SMA.

L'un des intérêts de ce travail est de considérer les SMA comme des multi-modèles qui permettent l'intégration de plusieurs formalismes. Dans ce contexte les SMA peuvent interagir avec d'autres formalismes DEVS ou encapsuler dans DEVS. L'ensemble de ce travail est implémenté dans une plate-forme logicielle VLE, *Virtual Laboratory Environment*, un outil dédié à la multi-modélisation de systèmes complexes. Cette plate-forme propose un outil nommé *translator*, ou traducteur, dont le rôle est de transformer une application XML de haut niveau d'abstraction en format reconnu par notre cadriciel sous forme de hiérarchie de modèles, d'initialisation et de plan d'expérience. Nous utilisons ce concept avec notre spécification SMA pour décrire l'ensemble de cette spécification sous la forme d'un fichier XML autorisant sa simulation dans VLE. Nous abordons ces différents concepts dans les prochains chapitres.

Chapitre 4

VLE architecture logicielle

Sommaire

4.1 Plate-forme VLE

Dans le cadre de cette thèse, nous avons développé une plate-forme de simulation et de modélisation basée sur les travaux de B.P. Zeigler [Zeigler *et al.*, 2000] et des travaux de formalisation développés dans le chapitre 2. Le fil conducteur de ce chapitre est la description de l'activité de modélisation et de simulation adoptée dans notre plate-forme où chaque partie du cycle de modélisation est décrite par un composant du cadriciel VLE.

Nous estimons que l'intégration de modèles hétérogènes, ainsi que le respect de la théorie de la modélisation et de la simulation, sont les principales caractéristiques pour fournir une plate-forme complète et fiable pour l'étude de systèmes complexes. Nos travaux sont issus d'un développement multi-disciplinaire entre l'informatique et la biologie dont la première version de VLE est le résultat [Ramat et Preux, 2003]. Néanmoins, cette version ne se base pas sur la multi-modélisation ni sur le cycle de modélisation.

Dans cette section, nous exposons les bases du développement de la plate-forme VLE, avec une définition du cycle de modélisation et sa traduction dans notre plate-forme. En deuxième partie, nous fournissons les concepts de génie logiciel sur lesquels notre cadriciel repose. Enfin, nous définissons le médium de communication utilisé dans les échanges d'information entre composants.

4.1.1 Cycle de modélisation et de simulation

Avec l'augmentation de la complexité des systèmes, la modélisation et la simulation demandent des outils informatiques de plus en plus efficaces. De plus, les champs de recherches comme l'écologie ou la sociologie, utilisent des logiciels de modélisation et de simulation afin de mieux comprendre la dynamique des systèmes étudiés. La principale difficulté, pour ces disciplines, est d'intégrer des connaissances hétérogènes, du point de vue des langages ou des formalismes, dans une unique représentation.

Un autre aspect des systèmes complexes est de respecter le cycle de modélisation et de simulation présenté dans la figure 4.1. En effet, le modélisateur doit être capable de définir ses modèles, via l'utilisation de plusieurs formalismes. Il doit pouvoir définir des cadres d'expériences en manipulant les entrées et les sorties de ses simulations pour affiner les paramètres de ses modèles en fonction de ses observations, ou encore, changer le comportement de ses modèles si ceux-ci ne fournissent pas les valeurs attendues.

Notre objectif, avec le cadriciel VLE, est d'offrir aux scientifiques un environnement de travail sous forme de laboratoire virtuel, c'est-à-dire, leur permettre de réaliser des expériences via l'outil informatique. VLE propose ainsi un environnement de spécification de modèles, d'expériences, d'exécutions et de traitements statistiques.

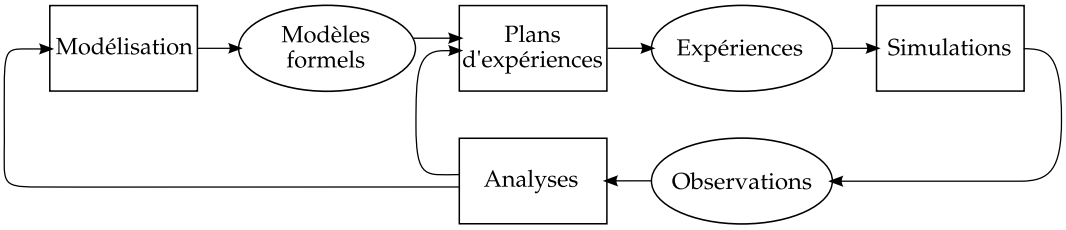


FIG. 4.1 – Cycle de modélisation et de simulation. Les boîtes représentent les actions, les ellipses, les résultats des actions. La plate-forme VLE propose, pour chaque action du cycle, un composant ou un module associé à la tâche.

Les actions, représentées par les boîtes sur la figure 4.1, sont traduites dans la plate-forme par les applications décrites ci-dessous :

- Modélisation : GVLE est un programme de modélisation utilisant des composants de modélisation qui définissent le comportement des modèles atomiques. Il permet l'utilisation des modèles couplés et respect ainsi la modularité et la hiérarchisation des modèles ;
- Plans d'expériences : GVLE possède un outil de préparation des plans d'expériences en proposant des valeurs d'initialisations, la planification de répliquas, les observations des modèles et les paramètres associés aux sorties ;
- Simulations : VLE est le moteur de simulation de la plate-forme, il fournit la traduction de la partie formelle du chapitre 2 en version opérationnelle en se basant sur les simulateurs abstraits. Il propose un module d'exécution des plans d'expériences sur plusieurs calculateurs ;
- Analyses : un ensemble d'outils d'observations en temps réel de la simulation, géré par l'application EOV, ou à posteriori via l'outil AVLE basé sur la plate-forme d'outils statistiques R [R Development Core Team, 2006][1].

Les résultats des actions, représentés par des ellipses sur la figure, ont les traductions suivantes dans notre plate-forme :

- Modèles formels : des modèles atomiques, ou des compositions de modèles fournis par la plate-forme ou développés par le modélisateur ;
- Expériences : une instance d'un plan d'expériences. Celles-ci sont générées automatiquement à partir du plan et sont simulées par le programme VLE ;
- Observations : les observations des sorties de la simulation sont de deux types : en cours de simulation ou à posteriori. Elles sont obtenues via des composants d'observations et des flux de données.

Nous avons choisi de développer la plate-forme VLE par une approche modulaire, c'est-à-dire, chaque programme du cadriciel utilise un ensemble de bibliothèques communes. Par exemple, nous avons développé des bibliothèques pour la lecture des fi-

[1] R est un langage et un environnement d'études statistiques proposant des sorties graphiques.

chiers du simulateur, pour la représentation de la hiérarchie de modèles DEVS et pour les simulateurs. Afin de permettre d'étendre facilement les possibilités de la plate-forme, nous utilisons le concept de composants et de composants légers. Les communications entre ces modules se font via l'emploi de formats de fichiers XML[2].

4.1.2 Concept de composants

La plate-forme VLE est construite autour du concept de composants du génie logiciel dont notamment : les composants logiciels, *software components*, et les méthodes de construction, *design patterns*.

Le concept de composant informatique est présenté pour la première fois par M. D. McIlroy [McIlroy, 1968], en 1968, où il définit un composant logiciel comme un élément d'un système rendant un service prédéfini et capable de communiquer avec d'autres composants. Depuis la naissance de la programmation orientée objets, la définition de composant s'est enrichie de notions complémentaires comme la réutilisabilité, les interfaces et le déploiement indépendant du composant [Szyperski, 1998].

Dans le reste du document, nous utilisons cette définition du composant, c'est-à-dire, celle où un composant :

- fournit un service précis et unique ;
- regroupe des fonctionnalités qui peuvent être appelées depuis un programme ou un autre composant externe nommé client ;
- propose une interface, c'est-à-dire un ensemble de fonctions lui permettant de communiquer avec le client ;
- doit être réutilisable, c'est-à-dire, que son utilisation ne doit pas se limiter uniquement au cadre du projet.

La Programmation Orientée Composants, POC, consiste à utiliser une approche modulaire au niveau de l'architecture d'un projet informatique. Cette méthode permet de transformer une application monolithique en briques, ou composants réutilisables. Elle permet aussi d'améliorer la lisibilité et la maintenance du projet. Cette technique de programmation est proche du développement orienté objets, mais au lieu d'utiliser une méthode de composition proche du code, elle utilise une méthode de composition sur l'architecture générale du logiciel.

Dans le reste de ce manuscrit, nous employons le concept de composants légers. Ce type de composant se définit comme un cas particulier de la définition précédente. En effet, ce composant hérite, en général, d'une seule interface fonctionnelle et ne peut remplir qu'une seule tâche. Les composants légers sont souvent appelés *greffons* ou *plugins* en anglais. Nous utiliserons souvent ce type de module dans le reste de l'application pour étendre les possibilités de la plate-forme avec des modules simples à

[2] *eXtensible Markup Langage*, XML, est un format à balises recommandé par le *World Wide Web Consortium*. Il est abordé plus amplement dans la partie 4.1.5

développer et pour sa capacité d'être aisément portable vers d'autres langages de programmation.

4.1.3 Langage de programmation et dépendances

Le développement d'une plate-forme informatique nécessite l'emploi d'un langage programmation standardisé et compatible avec les systèmes d'exploitations ciblés. Il doit, de plus, proposer un grand nombre de bibliothèques afin de fournir aux utilisateurs les outils nécessaires à leurs développements.

Après une étude des principaux langages disponibles, nous avons choisi le C++ de B. Stroustrup [Stroustrup, 1986] comme langage de programmation principal pour le développement de notre plate-forme VLE. Le C++ est un langage de programmation standardisé par plusieurs organismes[3], il est porté sur la grande majorité des systèmes d'exploitation et est interopérable avec la plus grande partie des langages de programmation comme, par exemple, le Fortran ou le Java. Les principales caractéristiques de ce langage orienté objets sont :

- la classification ;
- l'encapsulation où l'implémentation est cachée à l'utilisateur et accessible de l'extérieur par une interface ;
- la composition de classes ;
- l'association de classes ;
- l'héritage, éventuellement multiple et le polymorphisme ;
- l'abstraction où certaines parties d'une classe ne sont pas définies ;
- la généricité, l'utilisation d'algorithmes identiques pour des informations différentes ;
- la méta-programmation.

La plate-forme VLE se repose sur un grand nombre de bibliothèques dont certaines dépendent du projet GNU [Free Software Fundation, 1984] et principalement du projet Gnome d'environnement graphique [Free Sofware Fundation, 1997]. Ces bibliothèques sont :

- glib$_{mm}$: une bibliothèque qui étend la portabilité du langage C++ en ajoutant des fonctions de haut niveau, encapsulant celle du système d'exploitation, par exemple la gestion des chemins de fichiers ;
- gtk$_{mm}$: une bibliothèque C++ encapsulant gtk+ dit, *the Gimp ToolKit*, une bibliothèque de développement d'interfaces graphiques C ;
- libxml++ : une bibliothèque de manipulation de fichiers XML, via l'utilisation des interfaces de programmation DOM ou SAX. Elle encapsule la bibliothèque C, libxml2.

[3] Le langage C++ est standardisé par ANSI (The American National Standards Institute), ISO (The International Standards Organization) sous les normes (ISO/CEI 14882:1998) et sa dernière de 2003 (ISO/CEI 14882:2003) et plusieurs entités nationales.

Les principaux avantages de ces bibliothèques sont la portabilité et l'efficacité tant au niveau rapidité de développement qu'en terme de temps d'exécution. De plus, ces bibliothèques ont l'avantage de reposer sur les derniers concepts introduits dans le C++ et donc de fournir une plus grande simplicité de développement.

4.1.4 L'API de VLE

Le développement de notre plate-forme utilise un découpage modulaire strict entre les besoins des différentes applications. Chaque composant de la plate-forme ne fait qu'un seul type de traitement. Nous avons choisi ce mode de développement pour permettre une meilleure intégration des tests unitaires et de proposer aux développeurs de modèles une simplification de l'utilisation des composants et une meilleure représentation du projet.

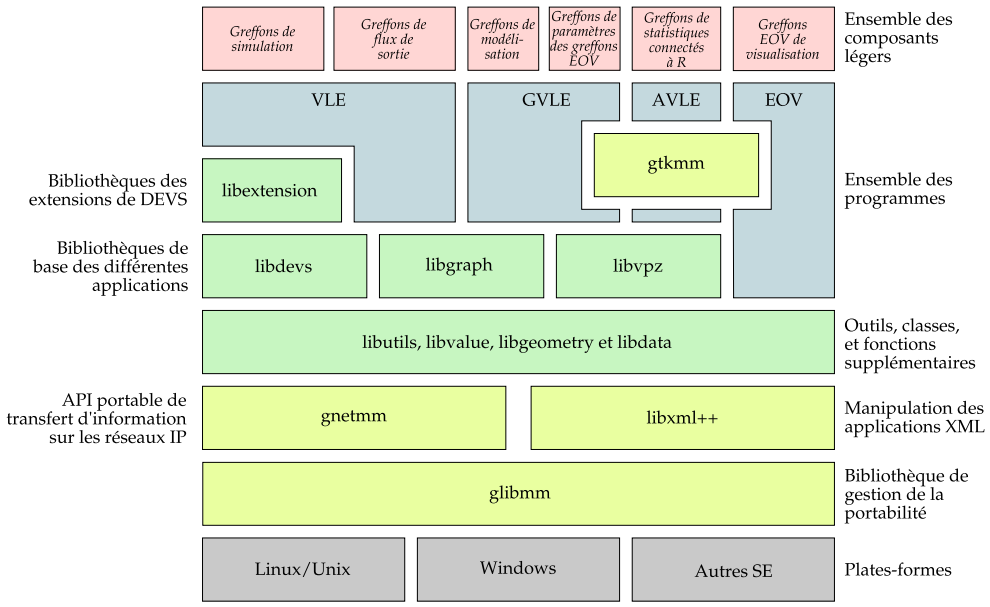


FIG. 4.2 – Graphe de dépendances entre les bibliothèques internes, représentées par des rectangles blancs et externes, les rectangles noirs, de la plate-forme VLE.

La figure 4.2 montre les liens de dépendances entre les différents programmes, bibliothèques partagées et composants dont les principaux sont :

– libutils : est une bibliothèque qui fournit des outils supplémentaires à la glib$_{mm}$:
  – libvalue : un ensemble de classes encapsulant des objets de type primitif, entier, réel ou complexe comme les listes, tableaux, tables associatives. Chaque classe possède une méthode pour se transformer en chaîne de caractères simple ou au format XML ;

  - libgeometry : un ensemble de classes génériques encapsulant des objets de définition d'espace continu comme les vecteurs qui fournit les opérations associées pour le calcul de vitesse dans l'espace etc ;
  - libdata : un ensemble de classes abstraites pour l'accès aux bases de données et sur les fichiers de données standard.
- libgraph : est un ensemble de classes permettant de manipuler une hiérarchie de modèles DEVS. Les principales entités sont les modèles atomiques et couplés, les connexions et les ports ;
- libvpz : est une bibliothèque dont le rôle est de gérer les entrées et sorties du fichier du cadriciel d'extension vpz ;
- libdevs : sont les classes de gestion de la simulation, dont font partie les objets coordinateur, coordinateur racine et simulateur, ainsi que l'échéancier, les événements ou la définition du temps ;
- libvle : est un module contenant les classes de manipulation de plans d'expériences, de la gestion des répliquas et de différentes fonctions d'exécution de simulation ;
- libextension : est la bibliothèque qui fournit les extensions du formalisme DEVS présentées dans le chapitre 2 via les modèles Cell-DEVS, DS-DEVS ou bien encore QSS.

4.1.5 Les applications XML

Les communications entre les différents composants, les modules et exécutables, ainsi qu'entre le modélisateur et la plate-forme, sont primordiales pour simplifier les échanges de modèles, permettre une écriture facile et proposer des informations interopérables. Pour apporter une réponse à toutes ces questions, nous avons décidé d'utiliser des applications XML pour les formats de sauvegarde des modèles. XML est un format permettant de définir des langages à balises via la définition d'une grammaire et d'un vocabulaire représentés par un fichier de type DTD ou XSchema. Si un fichier XML est bien formé, c'est-à-dire, sans faute de syntaxe et conforme à sa DTD, ou à son schéma, on dit que le fichier est valide. Depuis sa création, des technologies diverses sont apparues comme les feuilles de styles XSLT qui permettent de transformer une application XML vers d'autres types de documents.

Le but des applications XML est de faciliter le transport de données entre différents systèmes. De plus, elle sont décrites d'une manière formelle, permettant ainsi au programme de modifier et de valider les documents sans avoir une connaissance du document. Une description complète du langage XML peut être trouvée sur le site internet du W3C ou dans un grand nombre de livres, par exemple, *XML in a Nutchell* [Harold et Means, 2002] de E. R. Harold.

4.1.5.1 Formats utilisés dans VLE

Dans la plate-forme VLE, deux applications XML ont été créées à partir des travaux de R. Duboz [Duboz, 2002] avec les applications XML nommées *XML Meta Language for Model Coupling*, MLMC et *XML Meta Language for Virtual Experiment*, MLVE. À partir de ces travaux, nous introduirons une nouvelle application XML nommée VPZ pour *VLE file Project gZiped*. Ce format condense les formats développés par R. Duboz dans une seule application afin de simplifier l'écriture des modules des manipulations de ces données. Ce format regroupe alors la définition d'un plan d'expériences au sein de la plate-forme avec toutes les informations nécessaires, c'est-à-dire, la définition de la hiérarchie de modèles, l'initialisation des modèles atomiques et les formats des flux de sorties. Le deuxième est utilisé comme médium de communication entre le simulateur et les composants de visualisations. Ces formats sont décrits plus amplement dans les deux sections suivantes.

4.1.5.2 Application XML VPZ - définition de plans d'expériences

Cette première application XML fournit toutes les informations pour la définition d'un plan d'expériences. Elle est découpée en quatre parties distinctes :

- structure : définit la structure de la hiérarchie de modèles, c'est-à-dire, la définition de chaque modèle atomique avec son nom et sa description, une définition de ses ports d'entrées, de sorties, d'initialisations et d'états avec une description des données acceptées. Les modèles couplés définissent les modèles internes et les connexions internes et externes ;
- dynamique : définit la dynamique de chaque modèle atomique, à savoir, la définition du comportement du modèle. Le contenu de la balise est laissé libre au modélisateur. Par exemple, pour le *wrapper*, ou capsule, de résolution d'équations différentielles [Quesnel *et al.*, 2004a], nous proposons une syntaxe pour décrire les équations. Cette méthode permet de changer le comportement d'un modèle en changeant uniquement les informations contenues dans la balise dynamique ;
- expérience : détermine le plan d'expériences, c'est-à-dire, la durée de la simulation, une liste de valeurs d'initialisation des modèles atomiques, le paramétrage des observations et les composants de visualisation associés, le nombre et le type de répliquas ainsi que des informations sur l'auteur de l'expérience et sur la description de celle-ci ;
- graphique : définit la représentation graphique des modèles, par exemple, la taille, la position, les couleurs, les fontes utilisées, etc. Elle n'est nécessaire que pour le composant GVLE, cette partie n'est pas obligatoire.

La gestion des fichiers XML définissant les plans d'expériences est réalisée avec l'approche XML nommée DOM, *Document Object Model*, où le fichier XML est analysé et vérifié, via les fichiers DTD, ou XSchema, par la bibliothèque XML. Le développeur

peut ensuite manipuler la représentation objet du fichier XML directement via les interfaces fournies par la bibliothèque.

4.1.5.3 XML - médium de communication

La communication entre programmes est un des points très importants dans la conception d'une plate-forme moderne. Elle nécessite, pour son utilisation et son évolution, un langage simplement extensible évitant, ainsi, de modifier trop profondément les algorithmes de lecture et d'écriture. Ainsi, de nombreuses structures informatiques utilisent le format XML comme médium de communication.

Dans notre plate-forme, la communication entre les principaux composants utilise les applications XML suivantes :

- l'application XML VPZ, qui définit les plans et les instances d'expériences, est échangée entre les programmes, VLE, GVLE, EOV et AVLE ;
- la communication entre les modèles atomiques, lorsque la simulation est distribuée, utilise le format XML RPC ;
- la communication entre les observations et les modules de visualisations utilise une version simplifiée d'XML RPC ;
- les observations, sous forme de flux, sont réalisées dans les formats SDML ou dans un XML de représentation de données.

Dans le cadre de la communication nécessitant une lecture par le réseau, les algorithmes de lecture se font via la technologie XML, SAX : *Simple API for XML*. SAX est une interface de programmation permettant de lire et de traiter des documents XML. Contrairement à DOM, SAX traite les documents, balise par balise, laissant le traitement au développeur. Avec une telle lecture, l'utilisateur peut commencer à lire ses données sans avoir à attendre la fin du flux. De plus, en terme de gestion mémoire, SAX ne contient que la ligne sur laquelle le pointeur de lecture se trouve à la différence de DOM qui charge complètement l'arborescence XML en mémoire sous forme d'objets. Cette technique permet un gain sensible en terme de vitesse ainsi que sur l'occupation mémoire du programme, mais elle ne permet pas de modifier le fichier XML.

4.2 Modélisation

La modélisation d'un système dans la plate-forme VLE est en relation avec le programme GVLE. GVLE est une application graphique permettant de préciser le comportement des modèles atomiques, de composer de nouveaux modèles via les modèles couplés et de définir des hiérarchies de modèles. Elle possède, en outre, la possibilité de paramétrer les plans d'expériences avec les initialisations et les sorties des simulations.

Dans les prochaines sections, nous décrivons, en première partie, le développement du comportement d'un modèle atomique via l'utilisation des interfaces fonctionnelles. En deuxième partie, nous montrerons l'utilisation de GVLE pour composer des modèles à partir de modèles couplés. En troisième et quatrième partie, nous exposerons les concepts de *translators* et de classes de modèles. La cinquième partie sera consacrée à un exemple d'utilisation de la plate-forme VLE pour le développement d'un exemple de modélisation du modèle de T. Schelling introduit dans le chapitre 2 section 2.3.1.1.

4.2.1 Développement de comportements

Le formalisme DEVS, tel qu'il est employé dans le cadriciel VLE et sa mise à plat de la hiérarchie de modèles, n'attache de comportements qu'aux modèles atomiques. GVLE et VLE proposent plusieurs techniques pour simplifier le développement des comportements de ces modèles. Les sections suivantes abordent la construction de comportements à partir d'une interface fonctionnelle du modèle atomique ainsi que des différentes classes de haut niveau d'encapsulation des extensions de DEVS. En dernière partie, nous développerons la composition de modèles couplés à partir de modèles atomiques.

4.2.1.1 Interface fonctionnelle

Le développement du comportement des modèles atomiques utilise le concept de programmation orientée objets. En effet, nous proposons la classe `devs::Dynamics` comme interface fonctionnelle pour le développement de comportement de modèles atomiques. Cette classe est issue, pour une grande partie, des travaux de B. P. Zeigler avec les simulateurs abstraits [Zeigler *et al.*, 2000]. Nous ajoutons cependant, dans notre interface fonctionnelle, les fonctions d'initialisation et de capture d'états définies dans le chapitre 2 ainsi que la fonction de transitions pour la gestion des événements instantanés, la fonction de conflit.

Cette classe définit les fonctions virtuelles, ou abstraites, appelées par le noyau de simulation de VLE et conformément à l'algorithmique des simulateurs abstraits. Cette interface fonctionnelle doit être étendue par les classes filles pour apporter le comportement voulu par le modélisateur. La classe obtenue est enregistrée dans une bibliothèque dynamique qui est chargée en cours de simulation par le simulateur VLE.

L'interface fonctionnelle, dans le langage de programmation du simulateur VLE, est la suivante :

```
1 class Dynamics
2 {
3 public:
4     /*
```

```
5       * fonction de parcours de la balise DYNAMICS de l'application
6       * XML. Cette balise est laissée libre au modélisateur.
7       */
8      virtual bool parseXML(xmlpp::Element*);
9
10     /*
11      * la fonction de gestion de conflit apportée par DEVS parallèle.
12      */
13     virtual bool processConflict(
14           const InternalEvent& internal,
15           const ExternalEventList& extEventlist);
16
17     /*
18      * les fonctions traditionnelles de DEVS définies dans les
19      * simulateurs abstraits.
20      */
21     virtual Time init();
22     virtual Time getTimeAdvance();
23     virtual ExternalEventList* getOutputFunction(const Time& time);
24     virtual void processInternalEvent(const InternalEvent& event);
25     virtual void processExternalEvent(const ExternalEvent& event);
26     virtual void finish();
27
28     /*
29      * les fonctions du plan d'expériences : initialisation et capture
30      * d'une variable d'états du modèle.
31      */
32     virtual void processInitEvent(const InitEvent& event);
33     virtual Value* processStateEvent(const StateEvent& event) const;
34
35     /*
36      * la fonction de simplification du comportement des modèles en
37      * réalisant une réponse instantanée à une question. Cette
38      * fonction est constante, c'est-à-dire, elle ne peut modifier
39      * l'état du modèle.
40      */
41     virtual ExternalEventList* processInstantaneousEvent(
42           InstantaneousEvent* evt) const;
43
44 };
```

Nous pouvons remarquer, dans cette implémentation, que les fonctions de traitements des événements d'états et des événements instantanés voient leurs fonctions de transitions et de sorties, fusionnées. Cette technique permet de simplifier l'écriture des modèles sans changer le comportement formel de ces traitements définis dans les sections 2.2.3 et 2.3.3. Nous pouvons alors écrire :

$$\begin{pmatrix} processInstantaneousEvent & \Longleftrightarrow & \{\delta_{inst}, \lambda_{inst}\} \\ processStateEvent & \Longleftrightarrow & \{\delta_{state}, \lambda_{state}\} \end{pmatrix}$$

Le découpage en modules, ou greffons, du comportement des modèles permet, facile-

ment, de créer des ports vers d'autres langages de programmation. Ce portage est alors réalisé par deux méthodes distinctes: la première, lorsque le langage de programmation est compatible avec le C++, une classe intermédiaire est écrite pour effectuer les appels d'une classe à l'autre, c'est le principe du motif de conception, ou *design pattern* en anglais, nommé *proxy* [Gamma *et al.*, 1995]. La deuxième, dans le cas de langage non compatible, principalement les langages de programmation à machine virtuelle, nous utilisons des outils externes, principalement SWIG[4], qui génère des interfaces fonctionnelles et les fonctions de communications entre les deux langages de programmation à partir d'un langage de description d'interfaces fonctionnelles.

4.2.1.2 Les couches d'abstraction

Nous avons choisi, dans notre projet, d'utiliser les concepts de programmation orienté objets où l'héritage et le polymorphisme permettent de simplifier le développement. Nous fournissons ainsi un ensemble de classes héritant de `devs::Dynamics` avec un comportement pré-défini, par exemple, la classe `extension::QSS` est adaptée à la résolution d'équations différentielles [Kofman, 2002]. Ces classes surchargent et protègent les méthodes virtuelles de `devs::Dynamics` et apportent une interface fonctionnelle différente mais adaptée au modèle à développer.

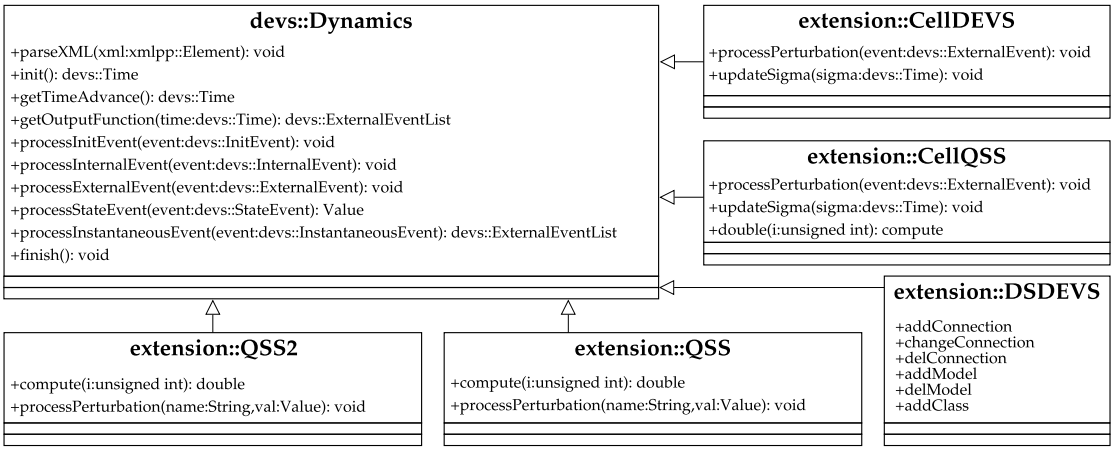


FIG. 4.3 – Diagramme de classe UML simplifié de l'arborescence d'héritage pour le développement du comportement des modèles atomiques.

Le diagramme UML de la figure 4.3 représente les différentes classes pour le développement du comportement de modèles atomiques. Ces classes fournissent un comportement générique et proposent au développeur une nouvelle interface fonctionnelle adaptée.

[4]SWIG, *Simplified Wrapper and Interface Generator*, est un outil pour réaliser des interfaces fonctionnelles de langage C ou C++ vers d'autres langages de programmation, `http://www.swig.org`.

– `extension::CellDEVS` : *Cellular automata DEVS*, l'implémentation des systèmes spatialisés dans DEVS [Wainer et Giambiasi, 2001] ;
– `extension::QSS` et `extension::QSS2`, *Quantified State System* [Kofman, 2002], pour l'intégration des systèmes continus dans DEVS. Les deux niveaux d'intégration sont proposés sous la forme des classes QSS et QSS2 ;
– `extension::CellQSS` : *Cellular-Quantified State System* [Versmisse et Ramat, 2005], une intégration de CellDEVS et QSS pour l'utilisation de système d'équations différentielles spatialisées ;
– `extension::DSDEVS` : *Dynamic Structure DEVS* [Barros, 1996], formalisation du changement de structure dans la hiérarchie de modèles dans DEVS.

L'ensemble de ces composants propose un comportement par défaut. Celui-ci correspond à ne rien faire, c'est-à-dire, oublier les perturbations et fournir une date d'avancée du temps infini. Afin d'étendre les modèles de comportement ou tout simplement de définir un modèle à partir de la classe `devs::Dynamics`, le cadriciel VLE apporte un composant de modélisation, attaché à GVLE, nommé editor.

Ce composant est issu de travaux de projets de Master Informatique de l'ULCO. Il offre une interface graphique pour la saisie de codes sources des comportements des modèles atomiques. Sa principale caractéristique est de permettre l'écriture de modèles à partir des interfaces fonctionnelles définies précédemment et ce, dans les langages de programmation compatibles avec la plate-forme.

Le fonctionnement de ce composant de modélisation est double puisqu'il propose de :

– sauvegarder, de manière traditionnelle, les fichiers sources dans un dossier. Ces sources sont compilées à la demande et installées afin de créer des composants légers de simulations pour l'exécution de VLE ;
– inclure le code source des modèles développés dans la partie comportement du fichier de sauvegarde du plan d'expériences VPZ. Lors de la simulation, VLE utilise un modèle spécifique pour compiler le code du programme et utiliser le modèle.

Cette dernière technique permet à la plate-forme VLE d'offrir la possibilité d'écriture ou de ré-écriture de modèles en cours de simulation afin d'améliorer un modèle ou étendre ces possibilités.

4.2.1.3 Développement par composition

La modélisation par composition de modèles est la partie gérée par le programme GVLE. Ce programme graphique a pour rôle de manipuler le graphe de connexions, d'ajouter et de supprimer des modèles. La définition de l'interface graphique de GVLE suit les règles développées dans le livre [Benson *et al.*, 2004] qui propose des méthodes et des recommandations pour l'écriture et la réalisation d'interfaces graphiques. La principale caractéristique de ces interfaces graphiques est de fournir des outils simples

à comprendre pour les utilisateurs et apportant de la souplesse lors de l'élaboration de celle-ci par le développeur.

Cette définition recommande de séparer les tâches lors de la création d'interfaces graphiques. Le projet GNOME conseille l'utilisation des composants du projet Glade[5] qui sont utilisés pour développer les interfaces d'un programme et de les enregistrer dans un format XML. Le développeur doit alors charger cette interface graphique et gérer les événements qui découlent des interventions de l'utilisateur. Ce découpage permet de faire développer des interfaces graphiques à des personnes ayant les qualités dans ce domaine sans en avoir dans la programmation d'interface homme-machine. De plus, cette technique possède l'avantage de permettre la modification de l'interface graphique, via le fichier XML, sans avoir à modifier le programme.

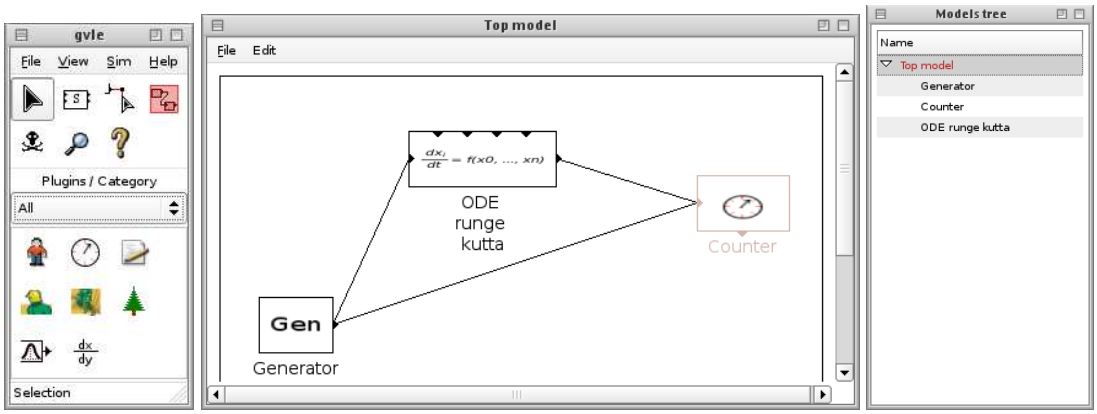


FIG. 4.4 – GVLE est utilisé pour la modélisation et pour la définition graphique des plans d'expériences. Il fournit une interface graphique pour manipuler les modèles atomiques et couplés. Les icônes sur la fenêtre de gauche représentent les composants de modélisation fournis avec la plate-forme. La fenêtre du centre est une représentation d'un modèle couplé composé de trois modèles. Enfin, l'image de droite montre la hiérarchie de modèles.

L'interface graphique de VLE se compose de trois fenêtres principales dont une représentation graphique est disponible sur la figure 4.4 :

– représentation des modèles couplés : une fenêtre est affectée à la visualisation des modèles fils du modèle couplé représenté et non de sa hiérarchie. Les modèles atomiques, de classes ou de type *translator*, ou traducteur, sont représentés par une image décrivant leurs actions ;
– outils de manipulation du graphe : les actions sont la création ou la suppression de modèles couplés, la définition de connexions entre modèles, ainsi que la visualisation

[5]Glade est disponible sur le site internet http://glade.gnome.org. Celui-ci propose un programme de définition d'interfaces graphiques, et de sauvegarder ces interfaces dans des applications XML. Il dispose également d'une bibliothèque, disponible dans la majorité des langages de programmation actuels, pour lire le fichier XML et créer l'interface graphique en mémoire.

des propriétés des modèles ;
– hiérarchie de modèles : la hiérarchie complète du système est représentée dans cette fenêtre ou les feuilles sont les modèles de type atomiques, classes ou traducteurs.

GVLE repose sur deux types de composants légers pour manipuler les modèles et paramétrer les composants de visualisation. Une représentation graphique du fonctionnement de GVLE est fournie sur la figure 4.5. La création, ou la modification, du comportement des modèles feuilles de la hiérarchie, c'est-à-dire, les modèles atomiques, les classes de modèles, ou encore les modèles de traductions, utilise un composant léger nommé *composant de modélisation*. Ce composant définit une interface fonctionnelle très simple avec simplement des échanges des applications XML fournissant la structure et le comportement des modèles feuilles. Le deuxième type de composant léger a pour rôle de paramétrer les composants de visualisation du simulateur VLE, par exemple, pour un composant d'affichage de courbes, il permet de définir les couleurs, les styles de lignes, les polices etc.

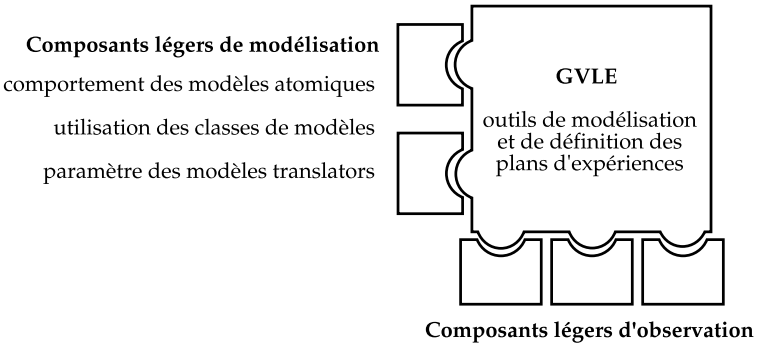


FIG. 4.5 – Présentation de GVLE et de ses deux types de composants légers de modélisation et d'observation.

4.2.2 Les traducteurs

La plate-forme VLE utilise un format de fichier basé sur une application XML dans le cadre de l'échange d'information entre les composants du projet. Ce fichier, ayant pour extension VPZ, est le fichier de sortie de l'outil de modélisation et de définition d'expériences GVLE, mais il peut, comme tout fichier XML, être modifié dans un simple éditeur de texte.

Néanmoins, l'activité de modélisation peut devenir une tâche très compliquée à mettre en œuvre dans GVLE, ou même dans un éditeur de fichiers XML, dès lors que la hiérarchie de modèles à manipuler est importante. En effet, dans le cas de la modélisation d'un automate cellulaire d'une taille relativement importante, la définition de la structure du modèle couplé génère alors un arbre XML conséquent. Enfin, sa manipulation dans l'interface graphique de GVLE devient impraticable puisque le modélisa-

teur doit, pour simplement modifier la taille de l'automate, ajouter ou supprimer les modèles et les connexions entre eux. C'est une opération fastidieuse et très peu ergonomique.

Pour résoudre ce problème, nous introduisons le concept de *translator*. Un *translator* est un composant léger qui permet de transformer, ou de traduire, une application XML en arbres XML compatibles VPZ. Dans le fichier VPZ, un *translator* prend la forme d'un modèle atomique, mais il possède un lien vers une application XML et le nom du *translator* permet de traduire cette application XML. Le résultat de la traduction par le *translator* est :

– une structure contenant des modèles atomiques ou couplés ;
– une dynamique pour les modèles atomiques en ayant besoin ;
– une liste de valeurs d'initialisation des modèles ;
– une liste des observations à effectuer.

Ces sous-arbres, résultats de l'application du *translator*, sont fusionnés avec l'application XML VPZ générale. La figure 4.6 montre un exemple de l'utilisation d'un *translator* pour la création simplifiée d'un automate cellulaire de six cellules. Le modélisateur a écrit le module *translator* qui traduit une application XML en application XML VPZ. Suivant le niveau de complexité du *translator*, la modification de la taille ou d'une cellule est fortement simplifiée puisqu'il suffit simplement de modifier les balises `row`, `columns` et `init` du modèle B.

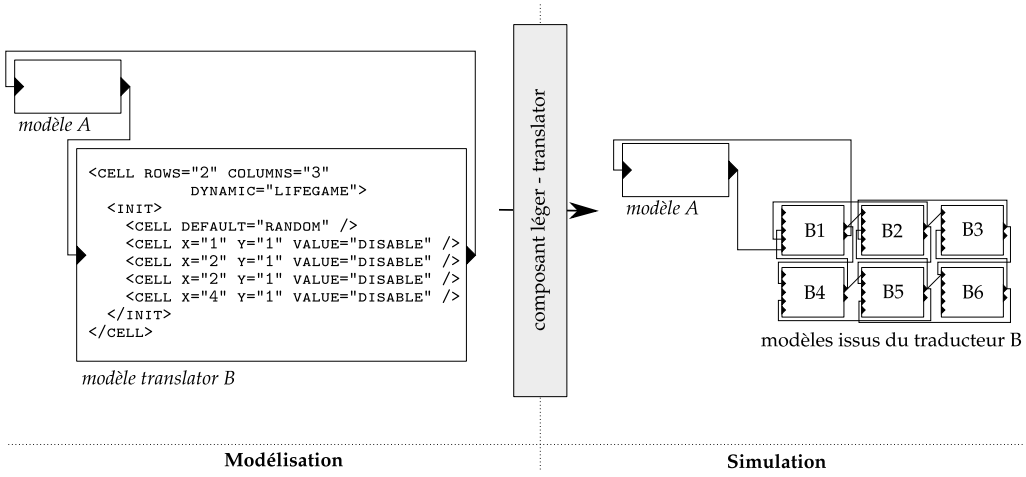


FIG. 4.6 – Représentation graphique de la transformation d'une hiérarchie de modèles VLE en utilisant un *translator*. La partie gauche est la partie modélisateur, celle de droite la vue du simulateur.

Les propriétés des traducteurs sont :

– la traduction d'une application XML en hiérarchie de modèles DEVS avec la définition des comportements des modèles atomiques et une définition pour l'initialisation

et la capture des états des modèles ;
– la conservation des connexions vers ou depuis le modèle traducteur; une fois la traduction réalisée, le modélisateur peut les affecter directement ;
– la possibilité de contenir des modèles de type traducteur pour former une récursivité d'appels de *translator* afin de réutiliser les composants légers existants.

Le concept de *translator* est une notion très utilisée dans les exemples fournis avec cette plate-forme puisqu'ils permettent de simplifier l'écriture de modèles en fournissant une méthode de généralisation de la création de modèles, ou de hiérarchies de modèles, en cachant la syntaxe XML VPZ. De plus, le composant de traduction d'applications XML étant libre, le développeur peut inclure des notions de calculs ou de boucles afin d'améliorer encore l'abstraction de la syntaxe DEVS.

4.2.3 Classes de modèles

Dans la section précédente, nous avons introduit le concept de *translator* et son utilisation pour simplifier le développement de modèles en utilisant des langages XML de plus haut niveau d'abstraction. Dans cette partie, nous introduisons le concept de *classes* de modèles dont le rôle est le même, c'est-à-dire, la simplification de l'action de modélisation.

Le concept de classes de modèles s'inspire du principe de la conception de classes et d'instances de classes de la programmation orientée objets. Une classe dite `Class` dans l'API VLE, définit un modèle ou plusieurs modèles. Elle propose une définition complète d'une hiérarchie de modèles DEVS avec sa structure, le comportement, l'initialisation et l'observation des modèles atomiques. La finalité de ce concept est de définir une classe de modèles pouvant être instanciée plusieurs fois dans une hiérarchie de modèles DEVS.

Un modèle de type classe se compose de :

– une hiérarchie de modèles DEVS couplés et atomiques ;
– une liste de dynamiques définissant le comportement des modèles atomiques ;
– une liste d'initialisations et d'observations des modèles atomiques.

Le fichier VPZ se compose alors des informations suivantes :

– une hiérarchie de modèles DEVS couplés et atomiques ainsi que des modèles de types traducteurs et de classes ;
– une liste de dynamiques définissant le comportement des modèles atomiques ;
– une liste d'initialisations et d'observations des modèles atomiques ;
– une liste des XML spécifiques aux modèles traducteurs de la hiérarchie de modèles ;
– une liste de classes de modèles.

Les classes de modèles ressemblent, dans le principe, à l'utilisation des modèles couplés. Cependant, elles permettent, lorsque la simulation utilise l'extension DS-DEVS,

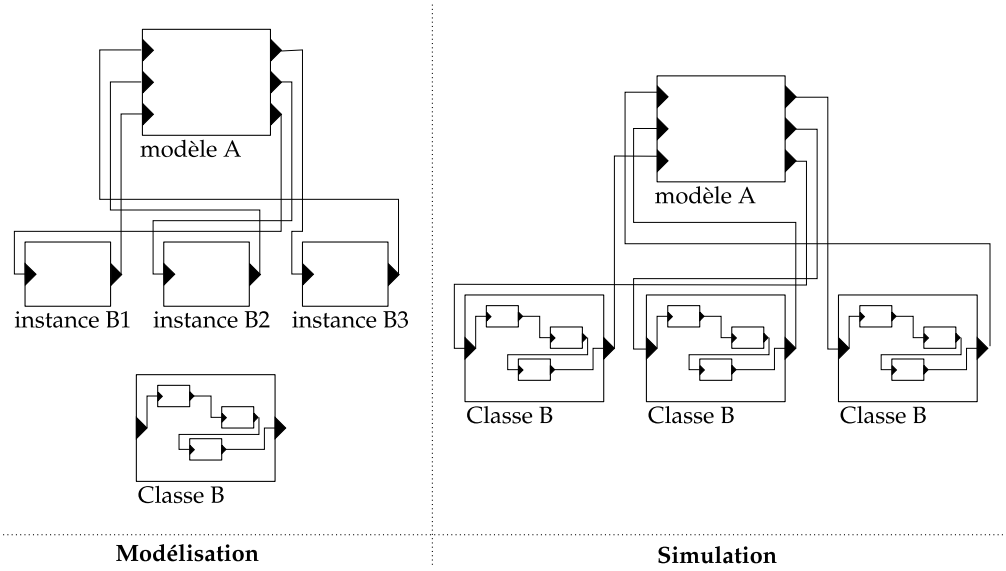


FIG. 4.7 – Utilisation d'une classe de modèles. La partie gauche est la partie modélisateur, celle de droite la vue du simulateur.

d'instancier une hiérarchie de modèles, avec toutes les informations sur les comportements, les initialisations et les observations, en lieu et place d'un unique modèle.

4.3 Plan d'expériences

Le but des plans d'expériences est de diminuer le nombre de simulation d'expériences en ajustant les initialisations, mais aussi de permettre d'interpréter des résultats facilement en limitant les données de sorties des expériences. Ils permettent également d'étudier un très grand nombre de facteurs comme la détection des interactions éventuelles.

La réalisation de plans d'expériences est une tâche importante du cycle de modélisation et de simulation puisqu'elle permet de développer et de réaliser des analyses statistiques de jeux de données multiples créés à partir de plusieurs jeux d'initialisations de modèles et via l'utilisation des répliquas.

Il existe de nombreux travaux sur la notion de plans d'expériences dont les premiers remontent à 1957 avec le livre de W.G. Cochran et G.M. Cox [Cochran et Cox, 1957]. Différentes méthodes, [Neter *et al.*, 1990, Box *et al.*, 1978], ont été formées à partir de ces travaux comme les plans factoriels. Un plan factoriel complet étudie toutes les combinaisons des différents facteurs, par exemple pour l'étude de k facteurs à deux niveaux, 2^k instances sont nécessaires.

La plate-forme VLE, en l'état actuel, propose l'exécution de deux types de plans, les plans complets et les plans factoriels. Le réseau de recherche, « exploration numérique », dont notre équipe de recherche fait partie, réalise des travaux, des créations d'outils d'analyses et des créations de plans d'expériences dans le langage de programme fourni par la plate-forme R. Dans un soucis de réutilisation et de maintenance de code, nous préférons pour le moment utiliser un module de génération de plans d'expériences.

4.3.1 Initialisation de modèles

Nous proposons dans notre plate-forme trois méthodes distinctes pour initialiser les modèles atomiques. Chacune de ces méthodes est utilisée dans un cas clairement défini par l'interface fonctionnelle des greffons de comportement. Les méthodes d'initialisation sont les suivantes :

- parseXML : cette fonction est appelée après le chargement de la bibliothèque dynamique par le constructeur de modèles. Elle reçoit en paramètre une référence sur un document XML qui contient des informations pour l'initialisation du comportement du modèle ou l'initialisation de variables globales du programme.
- init : cette fonction est laissée libre au développeur de modèle pour initialiser des variables de son modèle. Cette méthode est unique et elle n'est pas paramétrable. Elle permet de définir des valeurs par défaut de certains paramètres non initialisés par la fonction parseXML. Ainsi, si plusieurs comportements de modèles utilisent ce même code, la seule possibilité de différencier les modèles est de changer la structure du modèle atomique, c'est-à-dire, la définition des ports.
- processInitEvent : cette fonction est appelée juste avant l'exécution de la simulation par les événements d'initialisation reçus du simulateur. Ces événements sont un couple : (port d'initialisation, valeur). Cette fonction a pour rôle d'initialiser les variables d'états du modèle, mais c'est au modélisateur qu'incombe la tâche d'affecter ces variables afin de lui laisser la possibilité d'initialiser ces variables à l'aide de calculs.

Nous définissons les plans d'expériences à partir de ces différents types d'initialisation. Cependant, seule l'utilisation des événements d'initialisations permet de générer des jeux d'initialisations, les autres paramètres sont alors fixés, pour chaque instance, du plan d'expériences. Le but est de proposer de concentrer l'initialisation des modèles dans un seul endroit, représenté par l'application XML VPZ, afin de différencier le code des modèles de leurs initialisations.

4.3.2 Définition de plans

Les plans d'expériences sont définis dans l'application XML VPZ basée sur les travaux de R. Duboz de définitions d'expériences. Nous ajoutons une partie sur les conditions

expérimentales dans la définition de l'expérience. Cette balise met en relation un modèle atomique et l'un de ces ports d'initialisations à une valeur d'initialisation de type Value dans l'API VLE, c'est-à-dire, des types simples, entiers, réels, ou de type composé comme les listes chaînées ou les tableaux associatifs. L'application XML VPZ suivante est un exemple de définition de plan d'expériences :

```
<xml version="1.0">
<vle_project>
 [...]
  <experiment date="" author="">
   [...]
   <experimental_condition>
    <condition modelname="a" portname="i">
     <integer>1</integer>
     <integer>2</integer>
     <integer>3</integer>
    </condition>
    <condition modelname="a" portname="j">
     <list><integer>5</integer><integer>6</integer></list>
     <list><integer>2</integer><integer>3</integer></list>
    </condition>
   </experimental_condition>
  </experiment>
 [...]
</vle_project>
```

Dans cet exemple de définition d'expériences, nous réalisons un plan d'expériences à l'aide d'un modèle atomique et de ses deux ports d'initialisation i et j. Sur le premier port sont affectés trois entiers successifs 1, 2 et 3. Sur le deuxième, nous posons deux listes d'entiers, $(5,6)$ et $(2,3)$. Le plan d'expériences complet génère, dans ce cas, six simulations qui sont les $(1,(5,6))$, $(1,(2,3))$, $(2,(5,6))$, $(2,(2,3))$, $(3,(5,6))$ et $(3,(2,3))$. Le plan complet est alors défini comme la combinaison de toutes les valeurs d'initialisation disponibles dans le plan d'expériences.

Les plans factoriels sont alors définis via l'utilisation des combinaisons et des énumérations de valeurs pouvant se combiner. L'exemple suivant permet de réaliser ce type de plan :

```
<xml version="1.0">
<vle_project>
 [...]
  <experiment date="" author="">
   [...]
   <experimental_condition>
    <condition modelname="a" portname="i">
     <enumerate number="1">
      <integer>1</integer>
      <integer>2</integer>
     </enumerate>
    </condition>
    <condition modelname="a" portname="j">
```

```
14      <enumerate number="1">
15       <list><integer>5</integer><integer>6</integer></list>
16       <list><integer>2</integer><integer>3</integer></list>
17      </enumerate>
18     </condition>
19    </experimental_condition>
20   </experiment>
21  [...]
22 </vle_project>
```

La traduction de ce plan d'expériences produit uniquement deux simulations, $(1,(5,6))$ et $(2,(2,3))$. Les énumérations, lorsqu'elles possèdent le même identifiant et le même nombre de valeurs, sont exécutées élément par élément plutôt que de réaliser une combinaison globale. Les énumérations peuvent être combinées aux autres conditions expérimentales. Par exemple, si nous ajoutons la définition suivante dans les conditions :

```
1   <condition modelname="b" portname="i">
2    <string>a</string>
3    <string>b</string>
4   </condition>
```

Le plan d'expériences engendre avec ce modèle supplémentaire et ses deux initialisations, deux simulations supplémentaires, $(1,(5,6),"a")$, $(2,(2,3),"a")$, $(1,(5,6),"b")$ et $(2,(2,3),"b")$ qui sont la combinaison de l'énumération avec la combinaison. Ces deux types d'initialisations, combinaisons et énumérations permettent de produire des plans factoriels et complets.

4.3.3 Générateurs de nombres pseudo-aléatoires

Les générateurs de nombres aléatoires ont pris une très grande importance dans les simulations numériques aujourd'hui. Leurs qualités dépendent de la régularité des tirages aléatoires, d'une bonne répartition des tirages dans l'espace de valeurs et d'une période assez grande, c'est-à-dire, le nombre de tirages pendant lequel aucune similarité ne peut être trouvée entre les tirages.

Le générateur de nombres aléatoires utilisé dans le simulateur est celui fourni par la bibliothèque glib$_{mm}$. Ce générateur est basé sur le *Mersene Twister* développé par M. Matsumoto et T. Nishimura [Matsumoto et Nishimura, 1998] qui est présenté par P. L'Ecuyer comme un bon générateur de nombres aléatoires [L'Ecuyer, 2001]. Son principal avantage est une très longue période $2^{19937}-1$ comparée à la période du C ANSI, par exemple, qui est de $2^{48}-1$. De plus, il permet de découper les périodes en plusieurs séquences via l'utilisation d'une méthode pour se déplacer dans les tirages aléatoires sans générer de nombres aléatoires.

Son intégration, dans la plate-forme, est réalisée via une méthode de conception nommée *singleton* [Gamma *et al.*, 1995]. Celle-ci permet de ne fournir qu'une seule instance

d'un objet pour une simulation. Tous les modèles sont alors conviés à utiliser ce générateur de nombres pseudo-aléatoires afin de permettre la reproduction des simulations. La réplication des expériences possédant un caractère stochastique est un facteur essentiel. L'application XML VPZ propose différentes méthodes pour initialiser les générateurs aléatoires.

D'autre part, l'utilisation d'un générateur dans les simulations distribuées requiert une attention particulière [L'Ecuyer, 1998] afin d'éviter des corrélations entre les tirages aléatoires . L'idée générale pour la distribution d'un générateur de nombres aléatoires est de prendre les éléments d'une séquence et de les distribuer au simulateur. Trois techniques majeures existent [Traore et Hill, 2001] :

- Leap Frog (LF) : crée n sous-séquences de nombres aléatoires en utilisant une seule séquence. Les nombres sont fournis un par un au simulateur, où n correspond au nombre de simulateurs distribués ;
- Sequence Splitting (SS) : découpe une séquence de nombres en n sous-séquences (n équivaut au nombre de simulateurs), contenant m valeurs. La difficulté majeure est de déterminer une bonne valeur pour m qui doit être choisie pour chaque sous-séquence afin d'éviter les chevauchements ;
- Independent Sequences (IS) : construction de n sous-séquences en utilisant le même générateur mais avec n différentes graines. IS n'assure en aucun cas qu'il n'y aura pas de corrélation entre les tirages. Cette solution n'est pas bonne mais reste la plus simple à mettre en œuvre.

Dans notre spécification de simulateur, seul le coordinateur possède le générateur aléatoire. Les modèles ayant un besoin de nombres aléatoires utilisent l'appel d'une méthode de classe qui lui retourne une référence sur le générateur. C'est le rôle de la classe singleton. Lors de simulations distribuées, le modèle atomique distant fait appel à une méthode distante pour obtenir un nombre aléatoire. La technique est la même que lors d'appel de fonctions de transitions DEVS par exemple.

Cette technique doit évoluer vers la méthode *Leap Frog* afin d'économiser les coûts importants dûs aux transferts entre les calculateurs distants et le coordinateur. Dans ce cas, plusieurs techniques peuvent être employées, où chaque simulateur distant gère une sous-séquence de nombres aléatoires qu'elle reçoit du coordinateur racine ou qu'elle génère automatiquement, avec une copie du générateur aléatoire ou en utilisant la même graine d'initialisation.

4.3.4 Réalisation de plans

La réalisation des instances du plan d'expériences est à la charge du programme VLE, via le module *Directeur*. Celui-ci prend en entrée une définition du plan d'expériences, via l'application XML VPZ. La réalisation du plan, précisée précédemment, utilise les combinaisons et les énumérations des conditions expérimentales. Toutes les instances

du plan sont alors combinées avec les différentes valeurs d'initialisations du générateur aléatoire pour réaliser les répliquas.

4.4 Simulation

Dans le chapitre 2, nous décrivons de manière formelle le fonctionnement du noyau de simulation de VLE avec ses particularités que sont la mise à plat de la hiérarchie de modèles, la gestion des initialisations et des observations. La première partie de cette section se base sur cette description pour définir la traduction du temps et l'échéancier chargé de gérer les événements. La deuxième partie concerne les greffons simulation définis dans la partie 4.2.1.1 avec le découpage des simulateurs de modèles atomiques pour définir le fonctionnement de la distribution de modèles et le portage dans d'autres langages de programmation. Enfin, en troisième partie, nous développons une partie importante du simulateur qu'est la répartition de la distribution des instances, ou répliquas, de simulation.

4.4.1 Présentation du simulateur VLE

Le simulateur est le module principal de la plate-forme puisqu'il a pour objectif de simuler un modèle basé sur le formalisme DEVS. Dans cette partie, nous décrivons le fonctionnement interne du simulateur et principalement les méthodes associées pour la gestion des greffons de simulation développées par le modélisateur.

4.4.1.1 Gestion du temps et des événements

Dans un système à événements discrets, la gestion du temps est l'une des parties les plus importantes et les plus sollicitées. Dans le formalisme DEVS, employé par VLE, la fonction d'avancement du temps est laissée au modélisateur qui doit alors fournir une date à laquelle le prochain événement interne arrivera, via la fonction *ta*. Les événements estampillés sont alors stockés dans une structure de données nommée échéancier. Cette structure est alors fondamentale dans les performances du simulateur final.

Avant de rentrer dans les détails du fonctionnement et de l'implémentation de l'échéancier, nous développons dans les paragraphes suivants, les objets manipulés par cet échéancier : le temps et les événements.

La représentation du temps est assurée par une classe abstraite nommée simplement `Time`. Celle-ci propose par défaut d'encapsuler le temps sous forme d'un réel comme il est couramment employé dans les plates-formes DEVS. Cependant, il est possible d'intégrer de nouvelles représentations du temps comme proposé par [Fianyo *et al.*, 2001].

Pour intégrer ces changements, il est possible d'hériter et d'étendre la classe *Time*. Pour réaliser une classe fonctionnelle, il faut alors pouvoir fournir une relation d'ordre entre les dates et de pouvoir définir l'infini. Il est même possible de faire concourir les différentes représentations de date en fournissant une relation d'ordre entre ces différentes représentations.

Les lignes suivantes montrent l'interface fonctionnelle minimale de la classe *Time* qui peut être surchargée pour intégrer ces nouvelles représentations.

```
1  class Time
2  {
3  public:
4    Time(double time) :
5      value(time)
6    { }
7
8    /* Surcharge de l'opérateur d'infériorité utilisé dans
9       l'échéancier pour établir l'ordre des événements. */
10   inline virtual bool operator<(const Time& time) const
11   {
12     return (value < time.value() and not isInfinity()) or
13             time.isInfinity();
14   }
15
16   /* test si l'objet courant est égal à l'infini. */
17   inline virtual isInfinity() const
18   {
19     return value == Time::infinity().value();
20   }
21
22   /* la valeur de l'infini dont dépend la fonction isInfinity. */
23   static const Time infinity(-1.0);
24
25   /* récupère la valeur réelle du temps. */
26   inline virtual double value() const
27   {
28     return value;
29   }
30
31 private:
32   double value;
33 };
```

Les événements gérés par le noyau de simulation de VLE sont classés en cinq classes distinctes. Nous avons choisi, pour les gérer dans le noyau, d'utiliser un arbre d'héritage et le polymorphisme, présenté sur la figure 4.8, ayant pour source une classe mère nommée simplement *Event*. Celle-ci spécifie les informations globales à tous les types d'événements, comme le modèle source ayant généré l'événement et la date à laquelle il a été géré et enregistré dans l'échéancier. Il possède aussi une référence vers une donnée ou un ensemble de données attachées à l'événement pour être communiqué au destinataire.

Sur ce diagramme, nous pouvons remarquer que les événements internes et d'initialisation ne possèdent aucune autre information par rapport à la classe initiale. Au contraire, les événements externes possèdent des informations supplémentaires sur le destinataire du message et son port d'arrivée. Ces informations sont remplies par le simulateur lors de la répartition des événements externes sur le ou les modèles destinataires. La classe d'événement instantanée hérite directement de l'événement externe sans apporter de nouvelles informations. Son rôle est simplement de pouvoir être identifié par le simulateur pour que l'échéancier gère cet événement directement. Enfin, les événements d'états, qui héritent de la classe *Event*, apportent une information supplémentaire sur le port d'état et le nom de l'observateur du modèle.

La gestion des événements externes pose cependant un problème d'efficacité lors de la répartition des événements externes. En effet, lorsqu'un modèle pose un nouvel événement sur un de ces ports de sortie, si celui-ci est connecté à n modèles, l'événement est dupliqué n fois, un pour chaque modèle distant puisque le simulateur doit ajouter à l'événement le port d'arrivée de l'événement. Ce coût est d'autant plus important si une donnée est attachée à cet événement. Afin d'apporter un début de solution, nous utilisons un concept de modèle de conception appelé compteur de références, son nom est *pattern proxy* dans les motifs de conception ou *Design Patterns* [Gamma *et al.*, 1995]. Cette technique, souvent employée dans les langages de programmation où la gestion de la mémoire n'est pas automatique, est une technique qui permet de manipuler un objet et d'utiliser un compteur. Lorsque ce compteur est nul, c'est-à-dire aucune référence ne pointe sur cet objet, il est détruit. Ce concept permet de limiter la copie des objets en diminuant le coût des copies.

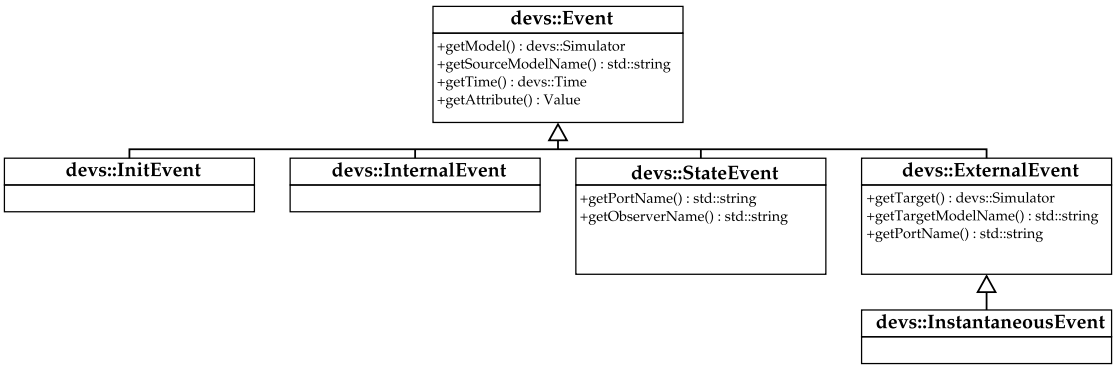


FIG. 4.8 – Diagramme de classe UML des événements disponibles du noyau de simulation.

4.4.1.2 Échéancier

Pour le développement de l'échéancier de VLE, nous nous sommes attachés à utiliser des algorithmes efficaces en terme d'utilisation mémoire et de temps d'exécution. Nous

avons opté pour des patrons de constructions, ou *templates*, de la bibliothèque standard du C++. Nous utilisons principalement les fonctions de gestion de tas, `std::make_heap` et les conteneurs standard de données, sous forme de tableaux ou de listes de tableaux, respectivement, `std::vector` et `std::deque`, pour la gestion des ensembles d'événements. La figure 4.9 est une représentation du fonctionnement interne de l'échéancier.

L'échéancier développé pour VLE est différent de celui développé pour un simulateur DEVS classique. En effet, il faut prendre en compte les caractéristiques de notre cadriciel :

- la mise à plat du graphe de modèles, qui force l'utilisation d'un échéancier unique pour la gestion des événements internes et externes ;
- l'utilisation de DEVS parallèle, qui évite le problème d'ordonnancement d'exécution des modèles, en utilisant des sacs de messages ;
- les événements instantanés, qui ont pour rôle de simplifier les graphes d'états des modèles et de rajouter un nouveau type d'événements ;
- les événements d'états, pour l'observation des modèles en cours de simulation de manière événementielle ou à l'aide d'une temporisation ;
- les événements d'initialisation, employés pour initialiser les variables d'états d'un modèle fourni du plan d'expériences à sa création.

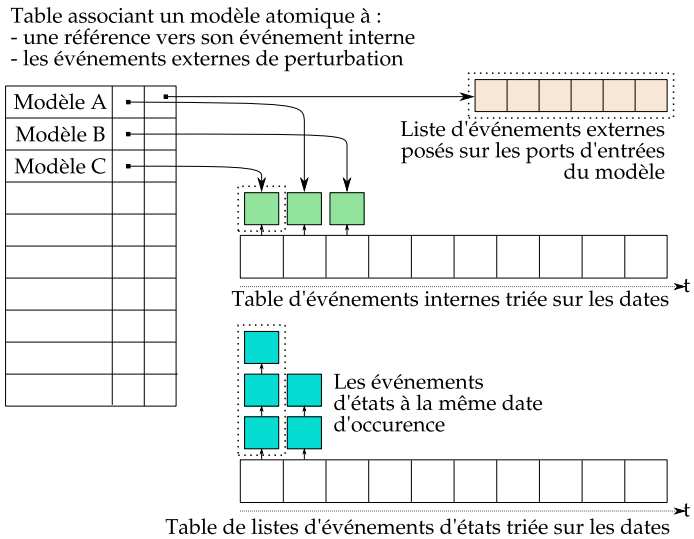


FIG. 4.9 – Structure de données de l'échéancier. Les tables d'événements sont des listes de tableaux triées par un tas, les autres conteneurs, de simples listes chaînées ou des tableaux.

Un sac d'événements est donc un ensemble d'événements internes, d'événements externes, d'événements d'états à la même date, représenté sur la figure 4.9 par les rectangles pointillés. Le simulateur reçoit un sac d'événements de l'échéancier, dépile les

événements internes et externes. Si un modèle reçoit un événement interne et un ou plusieurs externes, à la même date, sa fonction de conflit δ_{conf} est appelée. Si le modèle pose un événement externe sur un de ces ports de sortie, le simulateur ajoute cet événement à l'échéancier et non au sac. C'est le fonctionnement de DEVS parallèle. La différence de gestion des événements instantanés provient du fait que lorsque l'événement est traité par le modèle, via la fonction δ_{inst}, celui-ci génère un événement externe qui est ajouté dans le sac d'événements courant pour être traité immédiatement. Les événements d'états sont traités à la fin lorsque le sac d'événements ne contient plus d'événements internes ou externes.

Les événements d'états sont traités différemment dans la plate-forme. En effet, puisqu'ils n'ont pas d'incidence sur l'état des modèles mais qu'ils doivent récupérer une donnée « stable », ces événements sont gérés à la fin du sac des événements. Cette méthode assure que l'observation de l'état du modèle n'a lieu qu'après tout changement d'état du modèle par les fonctions de transition. Pour simplifier sa gestion, l'observation de modèles est entièrement gérée par le simulateur qui prend en charge la création et l'appel de la fonction δ_{state} du modèle ainsi que le rapatriement des données dans le flux de sortie.

La capture d'information est gérée par une classe abstraite nommée `Observateur` qui possède deux classes filles :

– *Observateur événementiel* : la méthode la plus simple, lorsqu'un modèle atomique, dont l'un des ports d'états est connecté à un observateur effectué une transition δ_{int} ou δ_{ext}, l'observateur prépare un événement d'états qui est envoyé à la fin de la gestion du sac. L'échéancier n'est alors pas utilisé, les événements d'états sont stockés dans le sac courant.
– *Observateur à temporisation* : une temporisation Δ_t est fournie à la création de l'observateur. Celui-ci crée alors un événement d'états pour chaque couple (port, modèle) sur lequel il est connecté. Dès qu'un événement de ce type est dépilé, le suivant est directement créé. L'échéancier sert alors à sauvegarder les différents événements d'états.

4.4.1.3 Greffons de comportement

Nous avons développé dans la partie 4.2.1.1 la définition de l'interface fonctionnelle du comportement des modèles atomiques basée sur les simulateurs abstraits de DEVS. Ce module joue alors le rôle de greffons de simulation. Dans les paragraphes suivants, nous développons le fonctionnement du chargement de ces composants à travers le concept de fabrique de modèles ou *Abstract Factory* dans les motifs de conception [Gamma *et al.*, 1995]. Celui-ci apporte des possibilités à VLE comme l'ajout et la création de modèles en cours de simulation.

Le concept de fabrique de modèles est une méthode de génie logiciel de patron de

modélisation, *Design Pattern*, qui permet, selon sa définition, de créer des modèles et de gérer leur vie tout au long de l'application. Les premières étapes de ce module est de parcourir le fichier d'expériences afin de lister les comportements des modèles, c'est-à-dire :

– le nom du comportement, représenté simplement par le nom de la classe du modèle héritant de l'interface fonctionnelle ;
– le langage de programmation définissant le nom et le type de bibliothèque dynamique, différent suivant les langages et les systèmes d'exploitations ;
– la localisation, si le modèle se trouve sur un calculateur local ou distant et sa méthode d'accès, une simple adresse IP par exemple.

L'exemple suivant montre l'utilisation d'une partie de l'application XML de VLE, VPZ, lors de la définition du comportement de trois modèles atomiques dans le cadre d'un calculateur de type GNU/Linux :

```
1  <?xml version="1.0"?>
2  <vle_project>
3   [...]
4    <dynamics>
5      <dynamic formalism="model::mesmodeles::counter" language="c++" />
6      <dynamic formalism="digestion" package="copepod" language="c++" />
7      <dynamic formalism="generator" type="vle.univ-littoral.fr:8080" />
8    </dynamics>
9   [...]
10 </vle_project>
```

– en cinquième ligne, le modèle *counter* de l'espace de nom `model::mesmodeles`, de la bibliothèque dynamique `libcounter.so` est chargée ;
– la sixième ligne, le modèle *digestion* du fichier `libcopepod.so` contenant plusieurs modèles est chargée ;
– la septième ligne définit un modèle distant nommé *generator* dont les fonctions peuvent être appelées via le port 8 080.

Les remarques d'ordre technique sont :

– un modèle peut faire partie d'un paquet aussi bien dans le langage de programmation C++ que pour les ports vers les autres langages ;
– si le paquet n'est pas défini, les modèles portent le même nom que la bibliothèque dynamique hors décoration du système d'exploitation ;
– lorsqu'un modèle est distant, le langage de programmation n'est plus nécessaire puisque les requêtes sont réalisées via des appels de méthodes distantes.

Une fois la liste des comportements extraite du fichier d'expériences, la fabrique de modèles réalise le chargement de la hiérarchie des modèles afin d'obtenir la liste des modèles atomiques, le nombre et le type de comportement à charger. Les comportements sont alors associés à la classe simulateur du noyau de simulation nommée `devs::Simulator`. La figure 4.10 montre graphiquement les liens entre les objets char-

gés des fichiers de bibliothèques dynamiques et le noyau de simulation avec le lien entre les simulateurs et la représentation hiérarchique de modèles.

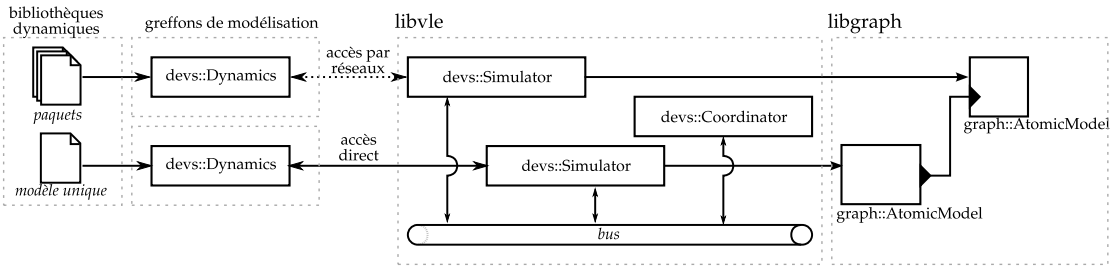


FIG. 4.10 – Représentation du couplage entre les bibliothèques dynamiques contenant le comportement des modèles atomiques et le noyau de simulation de VLE.

Les comportements, représentés par les greffons des simulations, sont chargés des bibliothèques dynamiques. À chaque modèle atomique `graph::AtomicModel` sont attachés un simulateur et un comportement `devs::Dynamics`. Les simulateurs sont attachés au bus DEVS auquel est connecté le coordinateur global qui anime la simulation via l'appel des fonctions de transition. Lorsque l'extension DS-DEVS est activée, la création de modèles est alors disponible aux modélisateurs sous forme de connexions sur les ports d'entrée du modèle *Exécutif*. Lorsque le modèle atomique *Exécutif* reçoit ce message, il peut influer sur le coordinateur pour ajouter un modèle en deux étapes. Le message contient :

– la structure du modèle et sa position dans la hiérarchie de modèles DEVS ;
– les informations sur le comportement de modèle vues précédemment ;
– les informations d'initialisation du modèle ;
– les événements d'initialisations.

Le modèle *Exécutif* demande au constructeur de modèles de créer le comportement du modèle de la même manière que lors de l'initialisation de la simulation.

4.4.2 Simulation distribuée

La simulation distribuée a pour rôle d'augmenter les capacités de simulation en terme de taille de simulation, ou plus simplement, d'exploiter des ressources informatiques éparses. Cette technique ne permet, en général, pas de diminuer les temps de simulation comme le propose la parallélisation de calculs, mais permet de tirer parti d'une meilleure architecture informatique. Par exemple, la mémoire nécessaire pour un modèle peut être importante et nécessiter un calculateur qui en propose suffisamment. De même, lors de l'utilisation de programmes externes, comme une base de données, il peut être intéressant de placer le modèle faisant des requêtes sur cette base de manière la moins coûteuse en communication. La répartition géographique des modèles est alors une fonction importante de la simulation distribuée.

Dans les paragraphes suivants, nous explorons la mise en place de la simulation distribuée dans l'architecture de VLE. Nous abordons la décomposition des modèles atomiques en une partie structure du modèle et une partie comportement qui introduit le concept de multi-langages et des enveloppes sur des formalismes ou des programmes, nommés *wrappers* ou capsules.

4.4.2.1 Décomposition de modèles

Nous avons développé dans la partie 2.2.1 de ce manuscrit le découpage formel du modèle atomique en deux parties distinctes. La première partie conserve les informations sur la structure du modèle, c'est-à-dire, ses ports d'entrée, de sortie, d'états et d'initialisation. La deuxième partie prend en charge la gestion du comportement du modèle avec les fonctions de l'interface fonctionnelle décrite dans la section 4.2.1.1. Ce découpage permet de proposer plusieurs avantages :

- le comportement d'un modèle atomique n'est pas attaché à une structure, c'est-à-dire, un comportement du modèle peut être réutilisé pour plusieurs modèles atomiques et la possibilité de créer des modèles génériques ;
- lorsqu'un comportement a besoin de connaître sa structure, il peut la connaître en utilisant une méthode spécifique de la classe `dynamics` ;
- la réalisation de portage vers d'autres langages de programmation est facilitée par le fait que seule cette classe sert d'intermédiaire entre les deux langages ;
- dans le même ordre d'idée, la réalisation d'une communication réseau entre le simulateur et le greffon de simulation est simplifiée puisque seule l'interface fonctionnelle est à développer.

Ce découpage intervient, dans la partie opérationnelle, en ajoutant une entité faisant le lien entre la structure du modèle et la partie comportement. Cette classe est nommée, `devs::Simulator`. La figure 4.11 est une représentation du fonctionnement de la communication entre le simulateur VLE et les comportements de modèles atomiques, via l'utilisation d'un bus logiciel nommé DEVS-Bus. En effet, nous avons adopté, pour la réalisation de ce simulateur, le concept de DEVS-Bus introduit par J.Y. Kim en 1998 [Kim et Kim, 1998]. DEVS-Bus permet l'interopérabilité avec différents formalismes qui doivent alors être encapsulés par des modèles atomiques nommés *wrappers*, ou enveloppes. Cette technique est décrite plus amplement dans la section 4.4.2.3.

Le fonctionnement de cette partie du noyau de VLE est relativement simple puisque chaque modèle atomique possède un comportement représenté sur la figure par les greffons de modélisation. Le bus logiciel possède, pour chaque modèle atomique, un objet *Simulator* qui fait la relation entre les composants légers et la hiérarchie de modèles DEVS. Les communications entre le bus logiciel et les *Simulators* sont, en réalité, les appels de l'interface fonctionnelle présentés précédemment. Sur le bus est connecté le coordinateur de VLE qui prend en charge la gestion du temps avec son échéancier global. Cette technique apporte plus de souplesse dans la gestion de la multi-modélisation

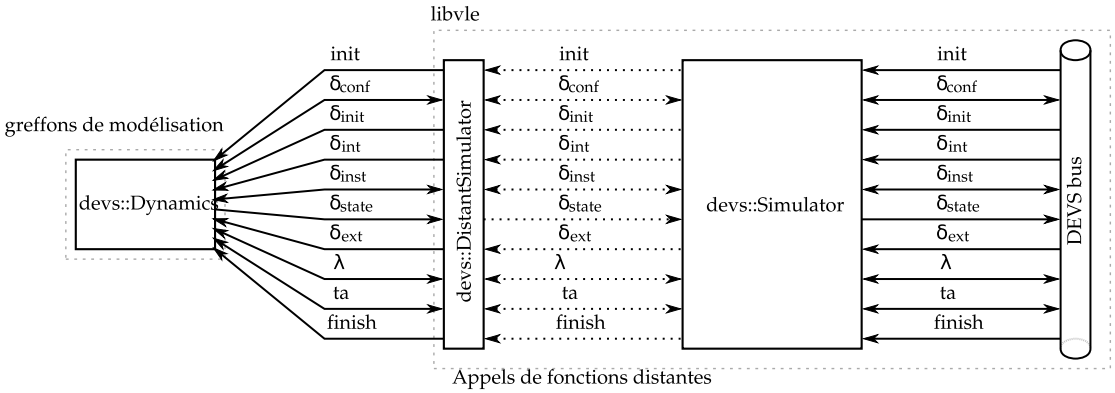


FIG. 4.11 – Représentation de l'utilisation de modèle distant au sein du simulateur. Le bus DEVS est connecté à tous les simulateurs qui sont eux-mêmes connectés aux comportements des modèles atomiques. Dans le cas présent, le comportement du modèle atomique est déporté sur une machine distante via l'utilisation d'un modèle de simulateur distant `devs::DistantSimulator`.

avec les enveloppes de formalisme et pour la gestion multi-langages ou simulation distribuée. De plus, lors de l'utilisation de modèles atomiques situés sur le même calculateur que le simulateur, les appels à la classe *Simulator* se font via le concept de fonctions *inline*[6] qui permet de réaliser un appel direct du bus logiciel vers la fonction du greffons de simulation. Cette technique permet d'éviter l'appel de la fonction de l'entité *Simulator* et ainsi, d'améliorer les performances.

4.4.2.2 Simulation muti-langages

La portabilité est une problématique du génie logiciel depuis la création des premiers langages de programmation. Aujourd'hui, la portabilité d'un programme peut se faire à plusieurs niveaux. Les bibliothèques logicielles comme POSIX ou Gtk limitent la portabilité du programme à son code source en fournissant des fonctions ou objets encapsulant les appels aux systèmes d'exploitations compatibles. Ainsi, le même code source, après compilation, peut fonctionner sur les différents systèmes d'exploitations. Il existe d'autres niveaux comme, par exemple, celui des langages de programmation possédant des machines virtuelles qui encapsulent les appels systèmes et proposent une API de haut niveau. Ces langages peuvent être classés en deux parties, ceux qui exécutent directement le code source, comme les langages Python ou Ruby et les langages utilisant des objets précompilés, comme Java ou C#, pour améliorer les performances des machines virtuelles.

L'une des principales raisons du choix du langage de programmation C++ est la possi-

[6] En informatique, une fonction *inline* dans un langage de programmation comme le C++, suggère au compilateur d'insérer le corps de la fonction à la place de l'appel de cette fonction.

bilité de communication avec les autres langages de programmation de manière simple et efficace. Pour les composants de simulation, nous proposons deux méthodes d'utilisation :

– La première technique utilise un couplage dur entre les deux langages de programmation : le simulateur VLE appelle directement les méthodes de l'interface fonctionnelle du comportement des modèles atomiques du langage de programmation cible. Les deux langages sont alors liés statiquement ou dynamiquement. Par exemple, pour le langage de programmation Java, la communication utilise les JNI (*Java Native Interface*), un ensemble de méthodes permettant d'appeler des fonctions du langage C ou C++.
– La seconde méthode, décrite formellement dans la partie 2.1.4, utilise une communication réseau entre la partie du simulateur attachée à VLE et à celle attachée au langage de programmation cible. Les appels aux fonctions de l'interface fonctionnelle du comportement des modèles atomiques sont alors envoyés sur le réseau vers le client. Cette technique a pour principal avantage de simplifier les outils de communication entre les deux langages de programmation malgré l'inconvénient d'une légère perte de vitesse d'exécution due à l'utilisation de la pile TCP/IP[7] du système d'exploitation.

Cette deuxième technique est principalement utilisée dans le cadre de la simulation distribuée puisqu'elle permet d'exécuter le comportement des modèles atomiques sur différents calculateurs. Ceux-ci peuvent alors être spécialisés. Par exemple, pour un modèle atomique requérant une quantité de mémoire importante, il peut être placé sur un nœud de calculateur ayant une quantité de mémoire suffisante.

Aujourd'hui, la plate-forme VLE propose deux portages complets, c'est-à-dire, tous les objets nécessaires à la communication C++/Java ou C++/Python, mais aussi les extensions du formalisme DEVS : Cell-DEVS, QSS, Cell-QSS et DSDEVS.

4.4.2.3 Utilisation de *Wrappers* ou capsules

La décomposition des modèles atomiques et le concept de *wrapping*, définis dans le chapitre 2, offrent une grande opportunité de communication entre la plate-forme VLE et des programmes issus de formalismes non-DEVS. Le concept de *wrapper* qui encapsule un formalisme pour le rendre compatible avec le comportement normal des modèles atomiques DEVS, comme par exemple les modèles ODE (*Ordinary Differential Equation*), système de résolutions d'équations différentielles et Petrinet, une encapsulation d'un réseau de Pétri [Quesnel *et al.*, 2004a] trouvent une nouvelle utilité lors du développement de modèles de simulation dans VLE.

Cette technique d'encapsulation d'un formalisme particulier peut aussi s'appliquer à

[7] TCP/IP, TCP (Transmission Control Protocol) et IP (Internet Protocol), les deux premiers protocoles utilisés par le réseau Internet. Le document de référence sur ce sujet est la RFC 1122.

des programmes informatiques pour les rendre compatibles avec la notion de simulation et à la modélisation sous DEVS. Ainsi, dans les paragraphes suivants, nous développons le *wrapper* autour d'un programme de gestion de base de données. Une base de données est un ensemble structuré et organisé pour le stockage de grandes quantités d'informations et manipulable via des opérations d'ajout, de mise à jour et de recherche de données. Les bases de données les plus connues sont pour les logiciels libres, PostgreSQL, MySQL et pour les propriétaires Oracle.

Le fonctionnement de l'interface fonctionnelle de la capsule ou *wrapper* sur un système de base de données peut être le suivant :

- *init* : initialisation de la communication avec la base de données, par exemple, le nom et le mot de passe de l'utilisateur ;
- *getTimeAdvance* : deux valeurs possibles, 0 lorsque le modèle doit répondre à un événement externe et ∞ le reste du temps ;
- *processInternalEvent* : envoi de la requète SQL à la base de données, reçue de l'événement externe, attente et stockage du résultat pour l'appel à la fonction de sortie ;
- *processExternalEvent* : réception d'un événement avec, comme données transportées, la requête SQL envoyée par un modèle externe ;
- *getOutputFunction* : une transformation du résultat de la réquête SQL effectuée par la fonction de transition interne ;
- *processStateEvent* : une simple interrogation de la base de données ou des informations plus globales sur la base de données elle-même ;
- *finish* : fermeture de la communication avec la base de données.

Nous remarquons que les fonctions λ, δ_{int} et δ_{ext} sont les manipulateurs de la requête SQL et du résultat vers le programme de gestion de bases de données. Ainsi, pour spécialiser le *wrapper*, par exemple pour les SIG[8], les informations véhiculées par les événements peuvent être des données issues du SIG, une position ou une zone sur une carte ou le résultat d'une requête.

Cette technique de *wrapper* permet de faire communiquer les modèles DEVS avec des informations issues d'autres formalismes amenant l'ouverture de la plate-forme VLE vers des modèles complexes ou manipulant une grande quantité d'informations. De plus, cette technique permet d'éviter le développement de modèles reproduisant les mêmes fonctionnalités que des programmes ou formalismes extérieurs.

[8]Un Système d'Information Géographique, SIG, permet de gérer des données alpha-numériques spatialement localisées pour le traitement et la diffusion de l'information géographique. Son rôle est de proposer une représentation de l'environnement spatial en se basant sur des primitives géographiques telles que des points, des arcs, des vecteurs ou des maillages. À ces primitives sont associées des informations telles que la nature de la primitive ou toute autre information.

4.4.3 Distribution de simulations

Le but de la distribution de simulations est d'augmenter la vitesse d'exécution d'un plan d'expériences. Le plan d'expériences dans VLE, présenté dans la section 4.3, fournit des informations sur l'initialisation et la capture des états d'un modèle. Ce plan d'expériences génère des instances d'expériences à partir des initialisations et des répliquas qui doivent être combinés.

L'exécution d'un plan d'expériences est une opération longue et coûteuse en temps de simulation suivant le nombre de répliquas et de combinaisons d'initialisations voulues. Cependant, un plan est facilement distribuable puisque seules changent les initialisations et les sorties des modèles qui sont, dans notre cadriciel, définies en dehors du comportement des modèles.

Dans VLE, la distribution de simulations consiste à répartir la charge de calcul des simulations des instances du plan d'expériences sur plusieurs calculateurs. Dans la section suivante, nous développons le fonctionnement interne de cette distribution et les différentes ouvertures de cette technique.

4.4.3.1 Fonctionnement

Nous proposons dans notre plate-forme un ensemble de programmes et d'API pour la réalisation de la distribution de simulations dans la logique du développement de VLE, c'est-à-dire, en découpant chaque tâche en composants. Les trois principaux composants exécutables de VLE sont :

Directeur ou *manager* :
: Son rôle, après la transformation du plan, VPZ, en multiples instances d'expériences, notées VPZi, est d'envoyer ces dernières aux répartiteurs. Il lit, pour cela, le fichier de paramétrisation globale de la plate-forme, la liste des répartiteurs disponibles pour l'utilisateur sur Internet. Il possède la capacité de demander aux répartiteurs, le nombre total de processeurs et ceux déjà utilisés. À la fin d'une simulation, le répartiteur envoie les résultats au manager afin de centraliser les résultats dans une même arborescence. Enfin, il possède, comme le répartiteur, la capacité de fonctionner en démon[9], c'est-à-dire, qu'il se met en attente de réception d'un fichier VPZ du réseau.

Simulateur ou *simulator* :
: L'unique composant de cette partie du cadriciel à posséder des liens vers les objets du formalisme DEVS, le coordinateur racine et la représentation de la hiérarchie des modèles. Son rôle est, simplement, d'exécuter une instance d'expériences

[9] Le terme démon (ou *daemon* en anglais, de l'acronyme *Disk And Execution MONitor*) désigne un type de programmes informatiques qui s'exécute en arrière-plan plutôt que sous le contrôle direct d'un utilisateur.

VPZi. Comme les formats des documents VPZ et VPZi sont les mêmes, le simulateur est capable d'exécuter la première instance d'un plan d'expériences si celui-ci est passé en paramètre.

Répartiteur ou *distributor* :
Le répartiteur est un composant dont le rôle est de répartir les documents d'instances d'expériences qu'il reçoit du directeur sur les processeurs dont il possède la charge. Il possède la capacité de récupérer les résultats des simulateurs et de les envoyer au directeur qui a la charge de centraliser les résultats. Il possède, en outre, la capacité de fonctionner en processus démon sur le calculateur sur lequel il se trouve. Il est alors en attente d'une connexion d'un directeur et de la transmission d'un ensemble d'instances VPZi.

La figure 4.12 représente le fonctionnement de la distribution de simulations dans la plate-forme VLE à l'aide des programmes présentés précédemment. Si nous prenons l'exemple du laboratoire LIL, nous avons à disposition :

– un calculateur Bull de seize processeurs Intel Itanium 2 Ghz ;
– un serveur à double processeur Intel Xeon 2,8 Ghz ;
– un nœud de cluster à double processeurs Athlon MP 2400+ 1,8 Ghz.

Dans une telle architecture, trois répartiteurs paramétrés à seize, quatre et deux processeurs sont nécessaires. Le module directeur peut alors demander l'exécution de 22 instances du plan d'expériences en même temps. Dès qu'un processeur finit sa simulation, ses données résultats sont rapatriées au directeur qui fournit une nouvelle instance du plan à simuler.

4.5 Analyses

Le rôle de l'analyse, dans le cycle de modélisation et de simulation, est d'interpréter les sorties des répliquas de simulations. L'analyse des observations et des sorties des modèles est donc une partie très importante du cycle de modélisation, puisque l'analyse permet de remettre en cause la modélisation, les modèles ou le paramétrage des modèles. Dans cette partie, nous développerons le fonctionnement de l'observation des modèles et les deux techniques d'analyse proposées par notre plate-forme, la visualisation des données en temps réel avec le programme EOV et l'analyse des données avec AVLE.

4.5.1 Captures des informations

Dans la plate-forme VLE, les observations sont effectuées grâce aux événements d'états. Ce type d'événement, décrit formellement dans la partie 2.2.3, permet de capturer la valeur des variables d'états des modèles en cours de simulation, sans modifier l'état des

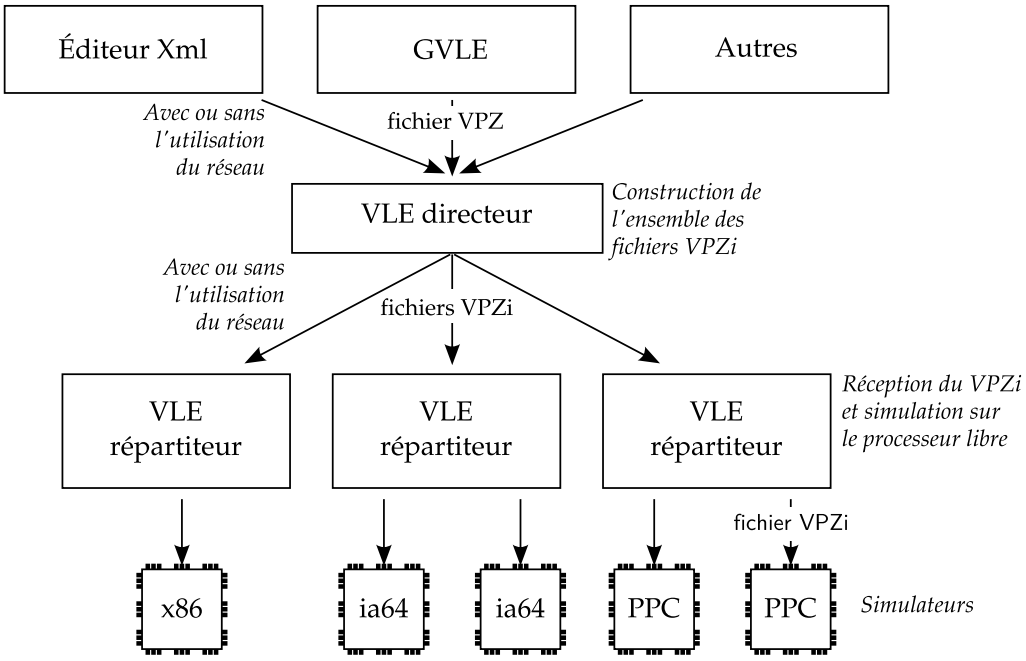


FIG. 4.12 – Procédure de distributions des instances d'expériences. Le fichier VPZ est envoyé au programme *manager* qui construit les instances d'expériences VPZi en accord avec le plan d'expériences. Les fichiers produits sont déployés sur les simulateurs dès qu'un processeur est libéré de sa précédente simulation.

modèles. Les événements d'états sont créés directement par le simulateur DEVS pour interroger les modèles. Deux types de captures peuvent être employés: une capture lors d'un changement d'état, c'est-à-dire, lors d'une transition interne ou externe, une autre utilise un pas de temps δ_t pour venir interroger le modèle.

D'un point de vue fonctionnel, l'interrogation des modèles suit un processus de traitement de l'information complexe. La figure 4.13 montre un exemple de capture d'un événement. Les événements d'états sont capturés par des objets de type *observateurs*. Ceux-ci ont pour rôle de récupérer les données d'un modèle suivant leur type de captures, événementielles ou à pas de temps constants. Les modules de type *mesures* ont pour rôle de récupérer les informations d'un, ou de plusieurs, observateurs et de trier ces informations suivant des règles d'associations de groupes, comme le numéro dans le groupe. Par exemple, lors d'une sortie texte d'un modèle de type automate cellulaire, les données doivent être triées afin de former l'automate cellulaire dans le fichier texte. Les données doivent être triées puisque les captures d'états des modèles, si elles se font de manière événementielle, ne permettent pas de capturer l'état de chaque cellule de manière linéaire. Les données collectées par une mesure sont envoyées à un flux de sortie. Il existe plusieurs types de flux, les fichiers et les flux TCP/IP.

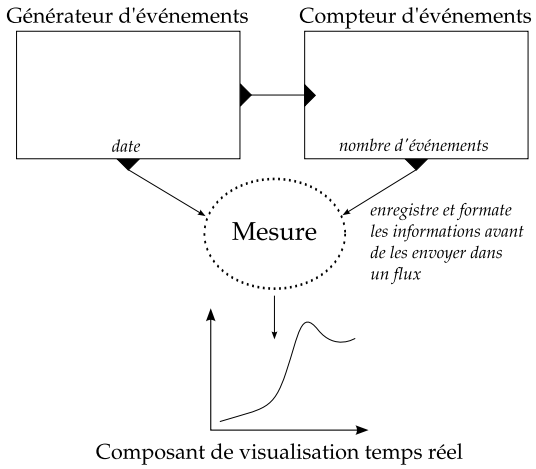


FIG. 4.13 – Représentation de la gestion des sorties dans VLE. Ce diagramme montre le couplage entre un modèle générateur d'événements, à gauche, et un compteur d'événement, à droite. Ces deux modèles ont un port d'état sur lequel est connecté un objet mesure, qui a pour rôle d'envoyer le résultat dans un flux, par exemple un composant de visualisation.

Les flux de sortie peuvent être connectés à des bases de données, des SIG ou des outils spécialisés. Cependant, une des méthodes les plus simples est de relier les flux de sorties des modèles à de simples fichiers. Dans ce cadre nous proposons, dans notre cadriciel, plusieurs formats de flux de type fichier :

– StatDataML : un format de type XML défini par le langage StatDataML[10] lisible par les programmes de statistiques comme le projet R ;
– CSV : sortie textuelle de données sous forme de matrice pour une importation dans les logiciels de type tableurs[11] ;
– Texte : simple sortie textuelle des informations reçues des mesures. Les informations sont simplement traduites sous forme de chaînes de caractères ;
– XML-RPC : utilise la définition des données du format XML-RPC pour les échanges avec les greffons de visualisation du programme EOV.

Les flux réseaux utilisent une communication TCP/IP. Ce type de flux permet d'envoyer les informations sur des ordinateurs distants. Nous utilisons le format XML-RPC pour les échanges d'informations entre les différents composants. Dans la plate-forme VLE, plusieurs composants sont fournis pour aider l'analyste. Nous les classons dans deux catégories distinctes :

– Les composants de visualisations qui ont pour but de montrer graphiquement l'évolution des variables d'états de modèles pendant la simulation ;
– Le programme AVLE, *Analysis for VLE*, est un programme qui utilise des greffons d'analyse connectés au programme R [R Development Core Team, 2006] afin de réaliser des études statistiques sur les résultats des observations.

Les sections suivantes décrivent le fonctionnement des deux types d'analyses proposés par la plate-forme.

4.5.2 Visualisation temps réel

La capture des états des modèles en cours de simulation est une méthode très simple d'observation des résultats des simulations. Il permet d'obtenir une visualisation simple, par exemple l'évolution d'une variable, ou plus complexe comme des moyennes, écarts-type etc. Il existe deux types de composants de visualisation temps réel dans la plate-forme suivant l'utilisation du programme EOV, *Eyes of VLE*.

4.5.2.1 Fonctionnement

Comme expliqué précédemment dans la partie 4.5.1, les composants de visualisation temps réel utilisent une communication réseau pour le transfert des données dans un format XML-RPC où les données sont représentées sous format de code XML encapsulé dans des balises d'informations sur les modèles et ports observés.

[10] SDML, *Statistical Data Markup Language* est un langage à balises pour la représentation de données statistiques.
[11] CSV, *comma-separated values*, est un format de fichier texte représentant les données sous forme de matrice.

Le programme EOV est un programme conçu pour écouter les requêtes d'un flux de sortie réseau du simulateur VLE. La communication se déroule en quelques étapes, où le simulateur est amené à envoyer les paramètres d'initialisations et de types de données aux greffons de visualisation. Chaque greffon se voit attribuer un port réseau sur lequel il va attendre le flux d'information du simulateur. Lorsqu'il reçoit ces informations, elles sont transmises aux fonctions de lecture de données XML-RPC et représentées graphiquement par le composant léger.

4.5.2.2 Composants légers

La plate-forme VLE fournit plusieurs composants légers au programme EOV. Ces modules héritent du composant graphique *conteneur binaire* de l'API de gtk_{mm} afin de pouvoir être assemblés dans le programme EOV. L'interface fonctionnelle, pour le développement des greffons de EOV, est très souple. Les principales lignes sont les suivantes :

```
1  class NetObserver : public Gtk::Bin
2  {
3  public:
4    /* récupère la première trame reçue et stockée dans la file. */
5    std::string top();
6
7    /* supprime la trame en sommet de file. */
8    void pop();
9
10   /* appelée lorsqu'une balise paramètre est reçue du programme EOV. */
11   virtual void parseParameters() =0;
12
13   /* appelée lorsqu'une balise data se trouve dans la file de message
14      reçu de VLE. */
15   virtual void parseData() =0;
16
17   /* appelée lorsque la simulation est finie. */
18   virtual void finished() =0;
19 };
```

La première fonction est utilisée pour paramétrer le composant avec des informations lues du fichier VPZ et envoyée sur le réseau. La deuxième fonction est utilisée pour lire une trame de données du réseau, celle-ci est un arbre XML contenant les informations d'une trame. Enfin, la dernière fonction est utilisée lorsque la simulation est finie.

Les composants légers fournis dans VLE sont :

– *Cell* : composant dont le rôle est d'afficher une grille carrée ou hexagonale. Chaque cellule possède une couleur calculée à partir de gradient ou traduite d'une table associant une couleur à une valeur. Des données supplémentaires peuvent être représentées sur les cellules de l'automate, des cercles, rectangles ou images ;

– *Plot* : inspiré des outils GNU Plot, ce module a pour but d'afficher l'évolution d'une ou plusieurs variables, suivant des droites, des points ou suivant un histogramme. Chaque variable est associée à une couleur ;
– *Gauge* et *Level* : deux modules très simples pour la visualisation de la valeur numérique d'une variable entre deux bornes fixées par le plan d'expériences ;
– *Particule* : composant de représentation d'un espace en trois dimensions continu dans lequel des objets statiques et des objets en mouvement peuvent être utilisés. Ce module utilise les bibliothèques OpenGL[12] proposées sur le système d'exploitation ;
– *Text* : principalement utilisé pour la découverte de problème dans les protocoles de communication. Ce composant montre directement, dans une zone de texte graphique, les données qu'il reçoit du flux.

4.5.3 Analyses post-simulation

Le composant AVLE est un programme d'analyses statistiques de données basé sur le projet R. Il utilise en entrée un fichier de description d'un plan d'expériences VPZ et par une simple manipulation des chaînes de caractères, il trouve les fichiers SDML, XML, ou texte. À partir de ces fichiers, l'analyste peut utiliser les greffons de statistiques fournis par R et AVLE pour analyser les résultats des modèles.

4.5.3.1 Projet R

S est un langage et un environnement d'études statistiques proposant des sorties graphiques. Il existe deux implémentations de S, la première, le projet libre R, la deuxième, le programme commercial Insightful's S-PLUS [Insightful, 1988]. R fait partie du projet GNU [Free Software Fundation, 1984] et est parfois appelé GNU S. Le langage R est similaire au langage et à l'environnement S, développé par J. Chambers et pour les dernières versions, R. Becker et A. Wilks [Becker *et al.*, 1988] des laboratoires Bell (maintenant Lucent Technologies). Le rôle de ce langage est de proposer un environnement mathématique pour l'analyse statistique de données.

R est considéré aujourd'hui comme une implémentation différente de S puisqu'il repose sur des différences importantes. Cependant, une grande partie du code écrit en S fonctionne de manière identique sous R. Ce projet propose une grande variété de méthodes statistiques, modélisation linéaire et non linéaire, les tests statistiques classiques, des méthodes de classification, etc. Les sorties de R peuvent être graphiques en utilisant des méthodes d'affichage évoluées.

D'un point de vue technique, R est un logiciel libre sous licence libre GPL et peut ainsi être lié, avec un couplage fort, c'est-à-dire, lié statiquement ou dynamiquement, à notre

[12] OpenGL, *Open Graphics Library*, est une spécification qui définit une API multi plate-forme pour la conception d'applications générant des images en deux ou trois dimensions.

plate-forme. De plus, il fonctionne sur la plupart des environnements Unix (incluant GNU/Linux et les BSD) et il est simplement extensible via le langage de programme R et ses API en langage C.

4.5.3.2 Principe de fonctionnement

Le fonctionnement du programme AVLE repose sur un ensemble de composants légers qui communiquent avec l'environnement R afin de produire les tests statistiques voulus. La figure 4.14 montre une représentation du fonctionnement du programme AVLE et de ses composants légers.

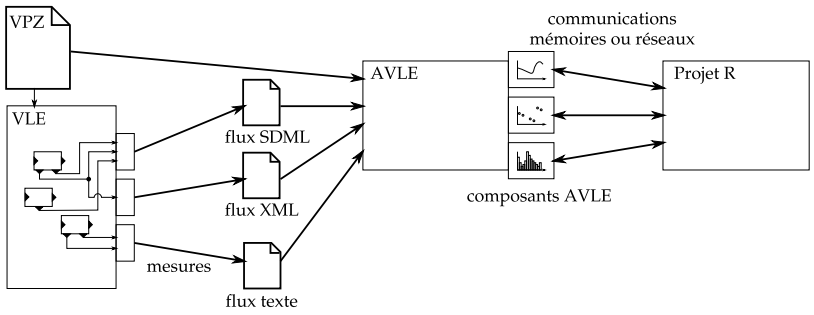


FIG. 4.14 – Représentation simplifiée du fonctionnement du programme AVLE, L'application XML VPZ est lue par le simulateur et par le programme AVLE. Lors de la simulation, les observations des modèles sont transmises aux mesures qui organisent les données et les envoient dans les flux de sorties. Le programme AVLE travaille indirectement sur ces fichiers résultats, via l'utilisation de composants légers d'analyse qui communiquent les calculs au programme R présenté dans la partie 4.5.3.1.

AVLE utilise en entrée deux types de fichiers : le premier, la définition du plan d'expériences VPZ et lLe second représente les résultats de la simulation, ou des simulations dans le cas de l'utilisation de répliquas ou des combinaisons d'initialisation fournies par le simulateur VLE. Les formats de données reconnus par R sont XML, SDML ou encore de simples fichiers textes. AVLE communique avec l'environnement de statistiques via une communication réseau. Cette communication permet de déporter les calculs de statistiques sur des calculateurs spécialisés afin de déporter le problème de logistique, mémoire, vitesse du, ou des, processeurs etc. Néanmoins, une communication locale peut-être employée si l'analyste veut utiliser le même ordinateur.

Les composants légers ont un rôle très simple de création de scripts R paramétrés par les informations du fichier VPZ et des choix de l'analyse. L'interface fonctionnelle repose sur deux méthodes :

– Récupérer une image représentant le rôle du composant. Cette image est insérée dans l'application AVLE pour permettre la saisie de ce module :

```
virtual Glib::RefPtr < Gdk::Pixbuf >& getImage();
```

– À partir des informations sur les mesures à effectuer et les fichiers sources employés, le script est généré et est retourné au programme VLE qui se charge de l'envoi à l'environnement R :

```
virtual std::vector < std::string >
run(const std::list < std::string >& measures, /* les mesures */
    const std::list < std::string >& filenames, /* les fichiers */
    const std::string& outputtype, /* le format de sortie */
    int width, /* la hauteur et la largeur */
    int height);
```

4.6 Conclusion

Dans ce chapitre, nous décrivons le fonctionnement de la plate-forme VLE que nous avons développée au sein de l'équipe. Nous nous inspirons des formalisations réalisées dans le chapitre 2, sur le fonctionnement d'un simulateur DEVS avec un graphe de modèle mis à plat et l'utilisation de plusieurs ajouts aux formalismes pour aider à simplifier l'intégration de modèles hétérogènes pour les modélisateurs. Dans ce cadre, nous utilisons le cycle de la modélisation et de la simulation pour définir les différents composants de la plate-forme. Nous avons fait le choix de réaliser un projet informatique entièrement géré sous forme de composants et de composants légers. Cette technique de modularité, une interface de programmation simple, ainsi que le choix d'une licence de type logiciel libre offrent à la plate-forme une grande possibilité d'évolution, d'extension et de pérennité.

Le choix de la modularité du code permet aux développeurs d'utiliser le simulateur VLE de plusieurs manières. L'une d'elle consiste à manipuler le module directeur. Par exemple, lors de la définition du devenir de la plate-forme VLE, R. Duboz [Duboz, 2004] propose de réaliser, à partir du simulateur VLE, un service web, c'est-à-dire, un composant informatique disponible sur Internet pour répondre à des requêtes d'utilisateurs et de fournir les informations des résultats sous forme de page html ou xml. La méthode décrite dans la partie 4.4.3 où on illustre le découpage en plusieurs modules de la plate-forme, permet de réaliser ce type de programme informatique où les entrées, d'un formulaire par exemple, sont les descriptions de plans d'expériences ; les sorties, de simple flux ou des représentations graphiques. Un script appelle alors le module directeur pour réaliser les plans d'expériences. La plate-forme décrite par R. Duboz sous la forme d'un simulateur accessible depuis un site Internet peut aisément être réalisée en utilisant la modularité de la plate-forme.

L'utilisation directe du module directeur dans une capsule logicielle ou dans un programme permet au développeur d'utiliser VLE comme une simple fonction. Ainsi, par exemple, un programme de paramétrage de modèles DEVS a été développé à l'aide

d'algorithmes génétiques pour paramétrer automatiquement un modèle via ces sorties. Pour ce faire, deux fonctions sont nécessaires, la première doit modifier l'application XML VPZ en entrée du simulateur et traduire les sorties de simulation en données à envoyer aux algorithmes génétiques. La modification de l'application XML VPZ se fait simplement via la bibliothèque *libvpz*. L'encapsulation du module directeur permet alors de bénéficier, directement des répliquas nécessaires aux algorithmes génétiques.

Une évolution possible de la plate-forme est de modifier le format complet de l'application XML VPZ vers un format de type GXL [Holt et Winter, 2000, Holt *et al.*, 2000, Winter *et al.*, 2002] : *Graph eXchange Language*. GXL est une application XML orientée vers l'échange de données entre outils informatiques. GXL définit des graphes orientés, typés, dirigés avec des attributs étendus pour représenter des hypergraphes et des graphes hiérarchiques. Cette dernière capacité nous permet d'inclure les graphes DEVS sous le format GXL. Un des avantages de GXL est de permettre l'échange d'instances de graphes en utilisant les informations d'un schéma de la même manière qu'une DTD, *Document Type Definition*, pour un fichier XML. Ce schéma permettrait à VLE de s'affranchir de certaines parties de codes de vérification des graphes. Par exemple, les connexions sur des modèles inconnus sont directement vérifiables avec le schéma GXL de VLE.

L'étude du format GXL fait l'objet d'un travail commun au sein du laboratoire via la création de la bibliothèque de manipulation de graphe GXL, libgxl++[13]. Une partie de cette bibliothèque, issue d'un projet étudiant de Master 1re année d'informatique à l'Université du Littoral - Côte d'Opale, permet, par exemple, de réaliser des sauvegardes de représentations de graphes sous forme de fichiers graphiques via l'utilisation de la bibliothèque de placement automatique de nœuds de graphes, Graphviz[14].

L'étude des plates-formes DEVS existantes, réalisée dans la section 2.1.1, montre que seuls Atom3 et VLE proposent une solution logicielle complète de multi-modélisation et de simulation en tant que telle, c'est-à-dire : avec un environnement de modélisation à partir de plusieurs paradigmes, de simulation, d'analyses et de gestion des expériences. Cependant, il semble que dans certains domaines comme la gestion des plans d'expériences, VLE soit plus avancée. De plus, l'ensemble des programmes de ce cadriciel utilise les concepts de modularité et de simplification des modèles afin d'ouvrir le développement à différents types de compétences. Nous apportons des concepts de simplification de développement de modèles et d'ouverture vers d'autres langages mais aussi, des capsules de modèles afin d'ouvrir VLE à d'autres formalismes. Un autre ajout concerne l'utilisation de traducteur qui prend une place très importante pour la définition d'expériences, par exemple, pour notre spécification SMA, dont l'implémentation utilise l'ensemble des outils et concepts définis précédemment.

[13] libgxl++ est une bibliothèque de lecture et de manipulation de format GXL disponible sur `http://sourceforge.net/projects/libgxlplusplus`.

[14] Graphviz, *Graph Visualization Software*, est une plate-forme de représentation de graphes, diagrammes, avec un placement automatique des informations, `http://www.graphviz.org/`.

Chapitre 5

Exemples de modélisation sous VLE

Sommaire

Ce chapitre clôt ce manuscrit en présentant des exemples de modélisation utilisant DEVS et les extensions que nous avons développées ou intégrées formellement dans le simulateur étudié au chapitre 2 et développé de manière opérationnelle dans le chapitre 4. Nous utilisons également la spécification agent développée au chapitre 3 et sa version opérationnelle intégrée dans notre plate-forme VLE pour développer un exemple de systèmes multi-agents.

Nous développons le premier exemple dans le but de montrer l'API de VLE et les outils associés pour simplifier l'écriture de modèle et d'expérience. Nous illustrons ces principes avec le modèle du jeu de la vie de J. Conway. Dans la deuxième partie de ce

chapitre, nous prenons l'exemple étudié dans le chapitre 2 sur le modèle de ségrégation de T. Schelling [Schelling, 1971]. Nous revenons sur l'utilisation du modèle répartiteur pour montrer une méthode de réduction du nombre de messages transitant sur le graphe du simulateur. En troisième partie nous utilisons la spécification du système multi-agents pour développer un système où une brigade de pompiers lutte contre un incendie de forêt.

5.1 Jeu de la vie

Le premier exemple permet d'introduire l'interface fonctionnelle des modèles atomiques. Nous illustrons ce composant sur le modèle du jeu de la vie présenté en 1970 par J. Conway [Gardner, 1970]. Pour cet exercice, nous utilisons l'extension Cell-DEVS dont le rôle est de décrire des automates cellulaires [Wainer et Giambiasi, 2001] dans le formalisme DEVS.

5.1.1 Présentation

Avant de développer le fonctionnement du modèle du jeu de la vie, nous décrivons dans les paragraphes qui suivent les règles qui définissent le jeu de la vie. Nous décrivons ensuite, le modèle *CellDevs* de l'API de VLE pour l'écriture de modèles de type automate cellulaire.

5.1.1.1 Le jeu de la vie

Le jeu de la vie est né d'une analogie entre les règles présentées ci-dessous et certains critères d'évolution de population de bactéries. L'état de l'automate à l'étape n est uniquement fonction de son état à l'étape $n-1$. L'évolution de l'état d'une cellule dépend de l'état de son voisinage direct représenté par les huit plus proches voisins.

Les règles de voisinage de J. Conway sont très simples :

- *naissance* : si une case est vide et que trois de ses voisines sont occupées, alors une naissance s'y produit ;
- *stase* : si une case est occupée, la survie n'y est possible que si deux ou trois cases voisines sont occupées ;
- *mort* : si une case n'est entourée d'aucune ou d'une seule cellule voisine occupée, la case est vide à la génération suivante, c'est la mort par isolement, ou, si une case est entourée de 3 voisines occupées et plus, la case est vide à la génération suivante, c'est la mort par surpopulation.

5.1.1.2 Cell-DEVS

L'implémentation effectuée de l'extension Cell-DEVS utilise les capacités de DEVS parallèle pour résoudre les problèmes de temps comme décrit dans le chapitre 2. *CellDevs* est une classe de notre API qui encapsule l'interface fonctionnelle du modèle atomique, *Dynamics*, en apportant un comportement par défaut. En effet, le modèle *CellDEVS* gère en interne un cache des cellules voisines. Le cache contient un ensemble de variables d'états que les cellules peuvent s'échanger. La création d'une de ces variables utilise les fonctions d'initialisation. Par exemple, pour les états de types booléens, la fonction membre suivante est proposée :

```
void initBooleanState(const std::string& name,
                      bool value,
                      bool visible = true);
```

Les accès en lecture et écriture sur ces variables d'états sont réalisés à l'aide de primitives, par exemple, pour les booléens :

```
void setBooleanState(const std::string& name, bool value);
bool getBooleanState(const std::string& name) const;
```

Les données aux informations stockées dans le cache pour les modèles voisins utilisent des primitives spécifiques. Par exemple, pour les booléens, les fonctions d'initialisation du cache et de récupération d'une variable du cache sont les suivantes :

```
void initBooleanNeighbourhood(const std::string& stateName,
                              bool value);
bool getBooleanNeighbourState(const std::string& neighbourName,
                              const std::string& stateName) const;
```

Les informations relatives aux voisins sont gérées en interne par le modèle *CellDevs* lors de la réception de mises à jours de modèles voisins où lorsque le modèle héritant de la classe *CellDevs* force cette mise à jour à l'aide d'une des primitives suivantes :

```
void modify();
void neighbourModify();
void resetNeighbourModified();
```

Cette encapsulation permet de simplifier l'écriture de modèle Cell-DEVS en gérant automatiquement les informations du voisinage. La section suivante décrit l'écriture du modèle pour la réalisation du jeu de la vie.

5.1.2 Modélisation

Nous présentons le fonctionnement du modèle avec les exemples de code C++ issus des modèles d'exemples fournis avec la plate-forme VLE. Le modèle présenté ci-dessous hérite du modèle `extension::CellDevs` qui est une interface vers un modèle atomique

auquel sont ajoutées les fonctions de gestion des perturbations provenant du modèle Cell DEVS. La classe « jeu de la vie » est définie par :

```
class LifeGame : public extension::CellDevs
{
public:
  LifeGame(devs::sAtomicModel* model) :
    extension::CellDevs(model)
  { }

  virtual ~LifeGame()
  { }

private:
  enum state { INIT, IDLE, NEWSTATE };

  state  m_state;
};
```

L'état courant des instances de cellules de *LifeGame* repose sur la variable *m_state* représentant la phase dans laquelle se trouve l'état du modèle. Pour définir le jeu de la vie, deux fonctions sont nécessaires :

```
#include <model/lifegame/LifeGame.hpp>

using namespace devs;
using namespace extension;

namespace model { namespace lifegame {

  devs::Time LifeGame::init()
  {
    initBooleanNeighbourhood("s", false); /* initialise le cache */
    if (not existState("s")) {
      initBooleanState("s", false);       /* initialise l'état courant */
    }

    m_state = INIT;
    neighbourModify();                    /* informe les voisins */
    return devs::Time(0);
  }

  void LifeGame::processInternalEvent(InternalEvent* ie)
  {
    switch (m_state) {
      case INIT:
        m_state = IDLE;
        break;
      case IDLE:
        setLastTime(ie->getTime());
        m_state = NEWSTATE;
        break;
      case NEWSTATE:
        /* récupère l'état de la cellule */
```

```
32          bool state = getBooleanState("s");
33
34          /* récupère les voisins dont l'état de "s" est vrai */
35          unsigned int n = getBooleanNeighbourStateNumber("s", true);
36
37          if (state and  (n < 2 or n == 4)) {
38            setBooleanState("s", false);
39          }
40
41          if (not state and (n == 3)) {
42            setBooleanState("s", true);
43          }
44
45          resetNeighbourModified();
46          m_state = INIT;
47          break;
48        }
49    }
50
51 }}
```

La fonction *init*, présentée à partir de la ligne 8, est la primitive traditionnelle de DEVS décrite dans les simulateurs abstraits [Zeigler *et al.*, 2000] pour l'initialisation des modèles atomiques. La première étape consiste à initialiser la cellule et ses huit voisins directs. Le modèle entre alors dans une phase dite d'initialisation, ligne 15, afin de passer immédiatement dans la fonction de transition interne. La fonction suivante est la fonction de transition interne, ligne 21. Dans cette fonction, trois grandes étapes sont développées :

– INIT : cet état consiste simplement à forcer le passage dans l'état IDLE immédiatement ;
– IDLE : état transitoire pour passer dans l'état NEWSTATE ;
– NEWSTATE : état dans lequel le modèle teste ses voisins et prend une décision de changer l'état de la cellule.

L'état initial « s », des cellules de l'automate, est fourni par des événements d'initialisations lus par les cellules de l'automate cellulaire. Ces conditions expérimentales sont fournies par le « translator » ou traducteur.

5.1.3 Expérience

Le fichier de définition de l'expérience utilise un *translator* pour la génération complète de l'expérience. Le fichier d'origine, présenté ci-dessous, montre que la structure n'est remplie que par un modèle de type « no-vle », lequel traduit les balises contenues dans `NO_VLE` par une structure DEVS réelle avec les 900 modèles atomiques, les connexions associées, les dynamiques et les initialisations.

```
1 <?xml version="1.0"?>
```

```
<VLE_PROJECT DATE="" NAME="system">
  <STRUCTURES>
    <MODEL NAME="system" TYPE="no-vle">
    </MODEL>
  </STRUCTURES>
  <EXPERIMENTS>
    <EXPERIMENT NAME="exp1" DURATION="100" >
    </EXPERIMENT>
  </EXPERIMENTS>
  <NO_VLES>
    <NO_VLE MODEL_NAME="system" TRANSLATOR="lifegame-tr">
      <SYSTEM NAME="c" LANGUAGE="c++">
        <SIZE L="30" C="30" />
        <TIME_STEP VALUE="1" />
        <SEED VALUE="6352" />
      </SYSTEM>
    </NO_VLE>
  </NO_VLES>
</VLE_PROJECT>
```

Comme nous pouvons le remarquer, ce type de document est plus compréhensible et permet de s'abstraire des concepts de DEVS. L'un des avantages des modèles de types « no-vle » est la possibilité de les utiliser dans des hiérarchies de modèles afin de simplifier l'écriture et la relecture des expériences.

FIG. 5.1 – Représentation graphique de l'évolution d'une simulation du modèle du jeu de la vie de J. Conway. Ces captures montrent le composant de visualisation *eov* restituant les états des modèles en cours de simulation.

La figure 5.1 montre l'évolution du jeu de la vie développé dans cette partie sous le cadriciel VLE. L'initialisation de l'état « s » des modèles *CellDevs*, fournie par le plan d'expérience, est aléatoire.

5.1.4 Conclusion

Dans cette section, nous avons développé le modèle du jeu de la vie en nous basant sur les classes *CellDevs* et *Dynamics* de l'API VLE. Le code pour réaliser cet exemple,

bien que très simple, montre la facilité de développement de modèle dans VLE. Cette exemple fait l'objet d'un portage vers le langage de programmation Python[1]. Les résultats obtenus sont d'une plus grande simplicité d'écriture, due en grande partie au langage, pour une perte de l'ordre de 50% d'efficacité par rapport à la version C++. Ce faible écart est dû au fait que le modèle développé ne réalise que très peu d'opérations, la plus grande partie étant gérée par le noyau avec la gestion des événements.

5.2 Modèle de ségrégation

Dans cette section nous développons le modèle de ségrégation T. Schelling utilisé dans le chapitre 2 pour l'illustration du problème des événements simultanés. Nous utilisons ici ce même exemple avec le modèle dispatcher étudié dans la partie 2.4.4.1 pour exposer l'efficacité de ce modèle dans la réduction du nombre d'événements échangés dans le noyau.

5.2.1 Présentation

Le modèle de ségrégation présenté dans cette partie est celui de l'économiste T. Schelling que nous avons développé dans le chapitre 2 pour illustrer le problème des événements simultanés. Pour rappel, ce modèle se place dans un quartier résidentiel où chaque habitant admet ou souhaite un voisinage différent de lui mais à partir d'un seuil, il décide de quitter le quartier. T. Schelling montre en appliquant la théorie des jeux que ce seuil peut modifier le quartier en deux situations stables : une ségrégation pure ou un mélange complet. Le résultat dépend alors de l'initialisation et du seuil de chaque habitant.

Le système est modélisé à l'aide d'un automate cellulaire où chaque cellule est un agent qui interagit avec ses voisins en utilisant un voisinage de Moore, c'est-à-dire, avec ses huit voisins immédiats. D'après le modèle de T. Schelling, les agents ont deux états :

– *inactif*, tant qu'un agent voit son nombre de voisins inférieur au seuil Δ, il reste inactif et attend le départ ou l'arrivée d'un nouveau voisin pour calculer son état ;
– *déménager*, si un agent a un nombre de voisins supérieur au seuil Δ, il décide de déménager. Le déménagement se traduit par un déplacement aléatoire sur une cellule vide.

[1] Python, [van Rossum, 1989], est un langage de scripts pouvant s'intégrer à différents niveaux dans un logiciel, comme outils de développement, de prototypage, d'intégration de scripts, infrastructure logicielle, etc. Python est un langage orienté objets, réflexif, portable et possède une syntaxe très souple pour un coût de l'ordre de 20× plus lent qu'un programme C.

5.2.2 Modélisation

Pour cet exemple, nous utilisons uniquement le simulateur de VLE et les extensions associées, telles que DS-DEVS et Cell-DEVS. Nous modélisons ce système à l'aide de trois types de modèles différents. Le premier est l'automate cellulaire, qui peut prendre trois valeurs différentes que nous associons à des couleurs :

– *bleues*, cellules occupées par les habitants de couleur bleue ;
– *rouges*, cellules occupées par les habitants de couleur rouge ;
– *noires*, cellules inhabitées.

Le modèle suivant, le contrôleur, est un modèle qui a pour but de connaître les emplacements vides de ceux qui sont occupés afin de proposer des emplacements aux modèles demandant un déplacement aléatoire.

Enfin, le dernier est le modèle *dispatcher* ou répartiteur, qui, connecté à l'exécutif du modèle couplé, modifie les connexions pour optimiser le nombre de messages circulant sur le graphe.

Dans ce système, tous les modèles atomiques de l'automate cellulaire sont connectés au modèle contrôleur sur son port d'entrée et l'un de ces ports de sortie. Ainsi, dès qu'une cellule décide de changer de position, elle envoie une requête au contrôleur qui envoie une requête à toutes les cellules afin de recevoir l'ensemble des couleurs de l'automate cellulaire et ainsi de connaître les emplacements vides. Une fois l'ensemble des valeurs obtenues, le contrôleur prend une position vide aléatoirement et déplace la couleur dessus en rendant son ancienne position inoccupée.

La figure 5.2 représente graphiquement le fonctionnement du système évoqué précédemment. Il montre que les cellules de l'automate cellulaire sont connectées au modèle contrôleur, en entrée comme en sortie. Le modèle contrôleur possède une sortie vers le modèle répartiteur pour les demandes de changement de position.

L'algorithme suivant, issu de l'exemple fourni avec le cadriciel VLE, représente la fonction de sortie du modèle contrôleur. Nous remarquons qu'il existe deux états pour lesquels le modèle effectue une sortie :

– REQUEST : dans cet état, le répartiteur envoie une requête à toutes les cellules et leur demande leurs états. Cette partie est réalisée en utilisant les événements instantanés afin de simplifier l'écriture du modèle de la cellule.
– RESPONSE : cet état correspond à la création de l'ensemble des requêtes à destination du modèle dispatcher. Celui-ci, un peu plus évolué que le dispatcher développé dans le chapitre 2, reçoit une combinaison de couleur et de positions qu'il doit envoyer à la cellule.

```
1 ExternalEventList*
2 controleur::getOutputFunction(const Time& currentTime)
3 {
4   if (m_state == REQUEST) {
```

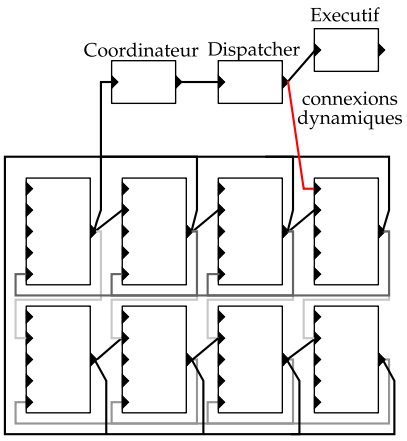


FIG. 5.2 – Représentation graphique du modèle de T. Schelling. Par commodité, l'ensemble des connexions entre les cellules de l'automate Cell-DEVS n'est pas représentée.

```
5     /* Envoie la requête à toutes les cellules. */
6     return new ExternalEventList(
7         new InstantaneousEvent("free?", currentTime, getModel()));
8
9   } else if (m_state == RESPONSE) {
10    /* Création du nouvel événement. */
11    ExternalEventList* l = new ExternalEventList();
12    std::vector < cell >::const_iterator it, found;
13
14    /* Boucle sur les listes de demandes de changements. */
15    for (it = m_changeList.begin(); it != m_changeList.end(); ++it) {
16
17      /* Création de l'événement à destination du dispatcher. */
18      ExternalEvent* e = new ExternalEvent("change", currentTime,
19                                           getModel());
20
21      /* Recherche une cellule vide aléatoirement. */
22      std::pair < int, int > p;
23      do {
24        int i = Rand::rand().get_int_range_excluded(
25                        0, m_candidateList.size() + 1) - 1;
26        p = m_candidateList[i];
27        found = m_candidateList[i];
28      } while (found->first == it->x and found->second == it->y and
29                      m_candidateList.size() > 1);
30
31      /* Ajout les attributs à l'événement. */
32      e << attribute("x", found->first)
33        << attribute("y", found.second)
34        << attribute("colour", it->colour);
35
36      /* Ajout le nouvel événement à la lise. */
37      l->addEvent(e);
```

```
38
39        /* Supprime l'espace libre utilisé. */
40        m_candidateList.erase(found);
41      }
42      return l;
43    }
44    return noEvent();
45  }
```

Dans le code C++ présenté ci-dessous, de la fonction de sortie du modèle dispatcher, nous avons fusionné le comportement traditionnel du *dispatcher* avec celui du modèle exécutif. Ainsi, les deux primitives *addConnection* et *delConnexion* se retrouvent directement dans le code du modèle *dispatcher* aux lignes 17 et 32.

```
1  devs::ExternalEventList*
2  Dispatcher::getOutputFunction(const devs::Time& currentTime)
3  {
4    devs::ExternalEventList* lst = DSDevs::getOutputFunction(currentTime);
5
6    /* Si la liste de demande de création n'est pas vide. */
7    if (not m_nameList.empty()) {
8      std::list < std::pair < std::string , int> >::iterator it;
9
10     /* Nous récupérons le premier élément, le nom et la couleur. */
11     it = m_nameList.begin();
12     std::string name = it->first;
13     int colour = it->second;
14
15     /* Dans l'état NEXT, on supprime l'ancienne connexion. */
16     if (m_state == NEXT) {
17       removeConnection(getModelName(), "out", name, "change");
18       m_nameList.pop_front();
19       it = m_nameList.begin();
20       if (it != m_nameList.end()) {
21         name = it->first;
22         colour = it->second;
23       }
24     }
25
26     if (it != m_nameList.end()) {
27       /* Création de l'événement de sortie. */
28       devs::ExternalEvent* ee;
29       ee = new devs::ExternalEvent("out", currentTime, getModel());
30
31       /* Ajoute la connexion entre le dispatcher et le modèle. */
32       addConnection(getModelName(), "out", name, "change");
33
34       /* Changement de l'attribut couleur. */
35       ee << devs::attribute("colour",colour);
36       lst->addEvent(ee);
37     }
38   }
39   return lst;
```

```
40 }
```

Les figures 5.3 montrent le résultat de la simulation du modèle de T. Schelling.

FIG. 5.3 – Ces captures d'écrans du composant EOV montrent l'évolution d'une simulation du modèle d'économie de T. Schelling sur la plate-forme VLE où trois familles de population évoluent.

5.2.3 Conclusion

L'exemple que nous venons de définir montre l'utilisation du dispatcher pour réduire les émissions d'événements entre les modèles au prix de calculs supplémentaires pour la gestion des connexions mouvantes. Si le modèle dispatcher n'était pas utilisé, chaque demande au contrôleur pour un changement de couleur, nécessiterait autant de messages supplémentaires que de cellules de l'automate cellulaire. Le nombre global de création d'événements évités est donc équivalent au nombre de cellules de l'automate et représente une économie de 30% de messages et ce, sans modifier le comportement des modèles du système.

Cet exemple a pour but de montrer une des optimisations que l'on peut réaliser avec l'extension DS-DEVS via le modèle répartiteur. Le modèle aurait pu être optimisé par d'autres manières comme par exemple en utilisant un cache, dans le modèle contrôleur, des réponses des modèles et de faire évoluer ce cache. Cependant, ce n'était pas le but de cet exercice.

Le prochain exemple a pour but de montrer le développement d'un système multi-agents en employant la spécification de SMA développée au chapitre 3. Cette démonstration s'appuie sur plusieurs aspects de cette spécification.

5.3 Modélisation d'une lutte contre un incendie de forêt

Le sujet abordé est la lutte contre les incendies de forêt où une brigade de pompiers s'évertue à combattre des feux déclenchés aléatoirement. Cet exemple n'est fourni qu'à

titre démonstratif de notre spécification SMA et ne se base pas sur un système réel. Cet exemple montre les aspects importants de notre spécification mais tous les aspects ne seront pas traités.

Nous aborderons cet exemple étape par étape à l'instar de ce manuscrit. Dans une première partie, nous décrivons les environnements employés sur le modèle. Nous développons ensuite les corps des agents avant de définir leurs comportements. Enfin, en dernière partie, nous développons un modèle scénario afin de réaliser le plan d'expérience de la simulation.

5.3.1 Description des modèles

Cette démonstration porte sur les environnements multiples. Dans cet exemple, nous utilisons deux environnements. Le premier, physique, a pour but de représenter la forêt et le feu sous forme de processus continus. Le deuxième environnement de type social existe pour que les pompiers puissent communiquer entre eux les endroits où ils découvrent des incendies.

Parallèlement aux méthodes classiques de résolution numérique d'équations différentielles, de nouvelles techniques sont apparues basées sur la quantification des valeurs de sortie plutôt que sur la discrétisation du temps. Ainsi par exemple, Kofman propose, à travers ses méthodes QSS1 et QSS2 [Kofman et Junco, 2001], des méthodes de résolution numérique d'équations différentielles du premier ordre. Pour la présentation de ces deux méthodes, considérons le système suivant :

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t)) \\ y(t) &= g(x(t), u(t)) \end{aligned}$$

Nous pouvons remarquer la dépendance en t de f qui se fait via une fonction u, ce qui est une façon d'intégrer la gestion du temps ou d'une fonction extérieure dans le formalisme DEVS. La fonction y est, quant à elle, la sortie désirée. Le modèle QSS1 repose sur deux principes :

– la quantification avec hystérésis de la fonction x ;
– un calcul du pas de temps en cohérence avec $\dot{x}(t)$ et la quantification.

La quantification avec hystérésis consiste à remplacer la fonction x par une fonction constante par morceaux $\hat{x}$. Cette fonction va prendre ses valeurs dans : $\ldots, d_{i-1}, d_i, d_{i+1}, \ldots$ en accord avec une discrétisation de $\mathbb{R}$. La fonction $\hat{x}$ prend la valeur d_i quand x augmente et $d_i \leq x < d_{i+1}$ ou quand x décroît et que $d_i - \varepsilon \leq x < d_{i+1} - \varepsilon$. ε est la largeur de la fenêtre d'hystérésis.

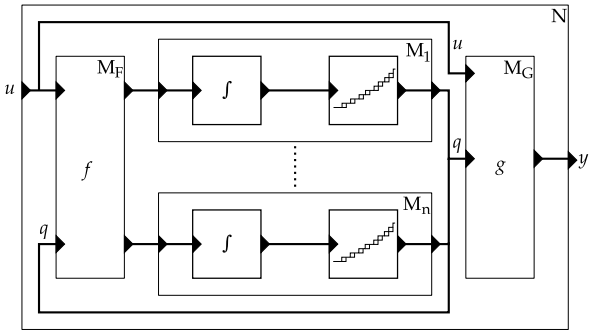


FIG. 5.4 – Modèle couplé DEVS d'un intégrateur QSS. Chaque boîte est un modèle DEVS atomique. (Source Kofman et Junco [Kofman et Junco, 2001])

5.3.1.1 Environnement physique

L'environnement physique utilisé dans cet exemple est un automate cellulaire de type Cell-Devs sur lequel nous faisons évoluer des équations de diffusion de la chaleur modélisée à l'aide de modèle QSS [Kofman et Junco, 2001] couplé à Cell-DEVS.

Le modèle *QSS* hérite du modèle *Dynamics* et apporte le comportement de résolution d'équation différentielle. L'interface fonctionnelle est équivalente au modèle *Dynamics* mais nécessite de développer la fonction virtuelle pure *compute* qui doit permettre de calculer la n^e équation du système d'équations différentielles :

```
virtual double compute(unsigned int n) =0;
```

Le modèle *CellQss* est un modèle qui hérite de *CellDevs* pour le développement d'automates cellulaires manipulant des processus continus modélisés par un système d'équations différentielles. L'interface et le fonctionnement de *CellQss* utilisent la même interface que *CellDevs* avec les données échangeables entre cellules, tout en apportant la fonction *compute* du modèle *Qss*.

5.3.1.2 Environnement social

L'environnement social ne décrit que deux rôles pour cet exemple, le pompier ou le pyromane. Les pompiers tentent de lutter contre les feux de forêt déclenchés par les pyromanes. Seuls les pompiers vont dialoguer entre eux pour s'informer de la découverte d'un feu ou lors de son extinction. Les pyromanes ne dialoguent pas entre eux. Les types de messages échangés entre les pompiers sont :

« $+(x, y)$ », découverte d'un feu sur la cellule en position (x, y), les pompiers doivent aider. Si le chemin pour aller aider le pompier signalant le feu est trop important, le pompier ne suit pas l'ordre et continue sa recherche ;

« $-(x,y)$ », ce message indique l'extinction du dernier feu en position (x,y), aucun voisinage n'est enflammé. Les pompiers en cours de déplacement peuvent arrêter leur déplacement vers cette cellule et reprendre une marche aléatoire.

5.3.1.3 Effecteurs et capteurs

Les effecteurs et les capteurs sont les entités de notre spécification qui permettent aux agents d'interagir avec les environnements ou les autres agents. Dans l'exemple de la brigade de pompiers, nous pouvons définir quelques effecteurs et capteurs :

- lutter contre le feu, en faisant baisser la température de la cellule de l'automate cellulaire sur laquelle le pompier se trouve ;
- s'orienter vers une cellule, afin de se diriger vers un appel à l'aide d'un pompier de la brigade ;
- s'orienter aléatoirement, quand il n'y a pas d'appel à l'aide, le pompier se déplace aléatoirement à la recherche d'incendie ;
- changer de vitesse, lorsqu'il est sur une cellule, le pompier arrête de se déplacer, quand il finit d'éteindre un incendie, il accélère ;
- envoyer un message aux pompiers, afin de les alerter d'une découverte d'incendie ou lorsqu'un incendie est maîtrisé.

Les capteurs forment les informations complexes à destination des agents afin de leur permettre de réagir ou de prendre des choix sur les prochaines actions à effectuer :

- indiquer la date de sortie de l'agent de l'espace, en calculant le déplacement du pompier avec sa vitesse et son orientation ;
- indiquer l'arrivée dans une cellule, afin de laisser au pompier le temps de vérifier qu'un incendie ne s'est pas déclaré dans les zones autour de lui ;
- indiquer les voisins où le feu est déclaré, détecte l'incendie sur les cellules voisines de l'automate cellulaire.

5.3.2 Description de l'implémentation

Tout au long de cette section nous allons introduire les concepts de notre spécification SMA. Nous développons, en premier, la mise en place de la marche aléatoire des pompiers. Nous abordons ensuite l'environnement physique et les modèles effecteurs et capteurs en relation avec les cellules de l'automate. En dernière partie, nous examinons l'environnement social et les modèles en relation.

5.3.2.1 Marche aléatoire des agents

Nous illustrons la marche aléatoire des agents pompiers par le graphe de structures DEVS mis en place pour le SMA. La figure 5.5 représente la marche aléatoire d'un

pompier. Pour le moment, l'automate cellulaire n'est pas requis puisque le pompier n'interagit pas avec. Nous pouvons décrire le comportement de cette partie très simplement, l'agent pompier, lorsqu'il change de direction ou de vitesse à l'initialisation, a pour conséquence de demander aux capteurs de calculer les dates auxquelles il doit s'attendre à réaliser une action. Lorsque le capteur de détection d'arrivée dans une cellule déclenche sa fonction de sortie, il informe le modèle agent qu'il vient d'arriver sur une cellule. Le comportement de l'agent est simplement de choisir une direction aléatoire et d'envoyer celle-ci à l'effecteur. Celui-ci par les biais des déclencheurs va envoyer cette information aux capteurs réalisant ainsi une boucle infinie.

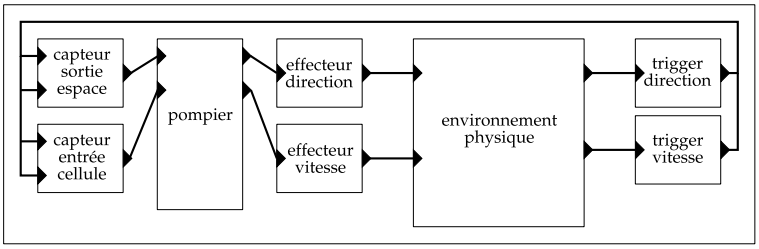


FIG. 5.5 – Représentation de la marche aléatoire des agents pompiers.

5.3.2.2 Interaction avec l'automate cellulaire

Les interactions entre les cellules de l'automate cellulaire et le pompier nécessitent l'ajout de trois nouveaux modèles. La figure 5.5 évolue pour représenter l'automate cellulaire et les modèles qui accompagnent son interaction avec les agents.

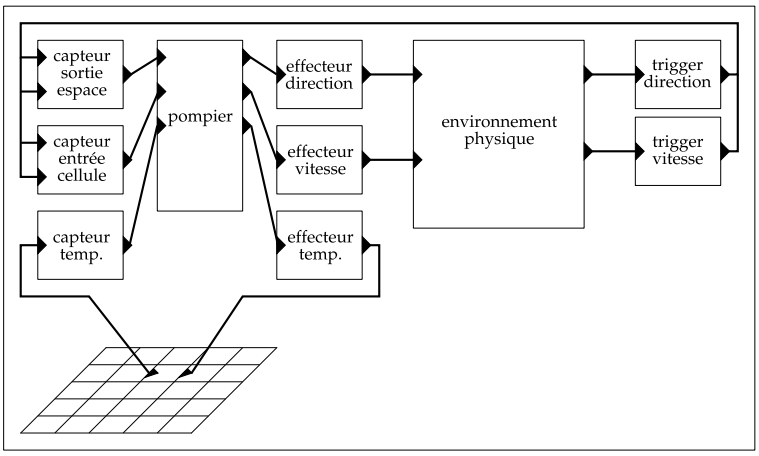


FIG. 5.6 – Illustration de l'automate cellulaire avec les modèles de perturbations.

La figure 5.6 intègre un automate cellulaire Cell-DEVS [Wainer et Giambiasi, 2001] dans lequel chaque cellule est remplacée par un modèle QSS [Kofman et Junco, 2001]. Pour

des raisons de commodité visuelle, nous avons simplifié sa représentation.

Deux modèles de type capteur et effecteur sont ajoutés au SMA.

– « effecteur de température » : il reçoit de l'agent une température qu'il va appliquer à la cellule. La température peut être négative si l'agent est un pompier et positive dans le cas d'un pyromane ;
– « capteur de température » : ce modèle est informé en continu par les informations reçues par le modèle QSS de l'automate cellulaire et dirige ces informations à l'agent lorsque la température est inférieure ou supérieure à un seuil, indiquant ainsi le déclenchement d'un feu ou son extinction. La figure 5.7 montre la capture des neuf cellules en même temps afin de capturer toutes les informations autour de l'agent.

Les modèles reliant les modèles d'effecteurs et de capteurs sont des connexions mouvantes, lorsque l'agent demande un déplacement et que celui-ci est validé par l'entité environnement physique, les connexions sont supprimées avec l'ancienne cellule et recréées avec la nouvelle cellule où se place l'agent.

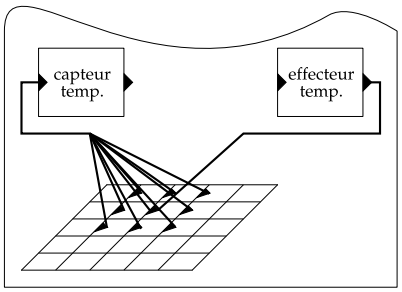


FIG. 5.7 – Représentation de la capture de plusieurs cellules de l'automate cellulaire.

L'ajout de ces modèles n'affecte que très peu le comportement du modèle agent. En effet, celui-ci n'a à gérer uniquement que les hausses de température relevées par le capteur de température. Lorsqu'il découvre un feu, il s'arrête en envoyant une vitesse nulle à l'environnement et commande à son effecteur d'éteindre le feu en refroidissant la cellule.

À partir de ce système, le pompier peut se déplacer et éteindre des feux en suivant une marche aléatoire. Nous pouvons remarquer que l'emploi de cette spécification nécessite un certain nombre de modèles DEVS. Cependant, la plupart sont très simples à développer puisqu'ils ne réalisent qu'une seule action par opération. De plus, ils peuvent être fournis par l'implémentation de la plate-forme.

Avant de décrire le comportement de l'agent, il faut mettre en place le réseau de communication. Celui-ci consiste, dans cet exemple, à ouvrir des communications entre les agents pompiers.

5.3.2.3 Interaction entre les agents

L'environnement social décrit un groupe d'agents dans lequel des rôles associés à des comportements se distinguent. Dans notre exemple, deux rôles existent, les pompiers et les pyromanes. La seule différence pour le moment entre ces deux entités survient lors de l'envoi de la température à l'effecteur, positive pour les pyromanes, négative pour les pompiers. Les pompiers ont cependant un avantage par rapport aux pyromanes puisqu'ils ont la capacité de communiquer les départs de feu ainsi que leurs extinctions. La figure suivante 5.8 représente l'infrastructure complète d'interactions entre les agents et les environnements.

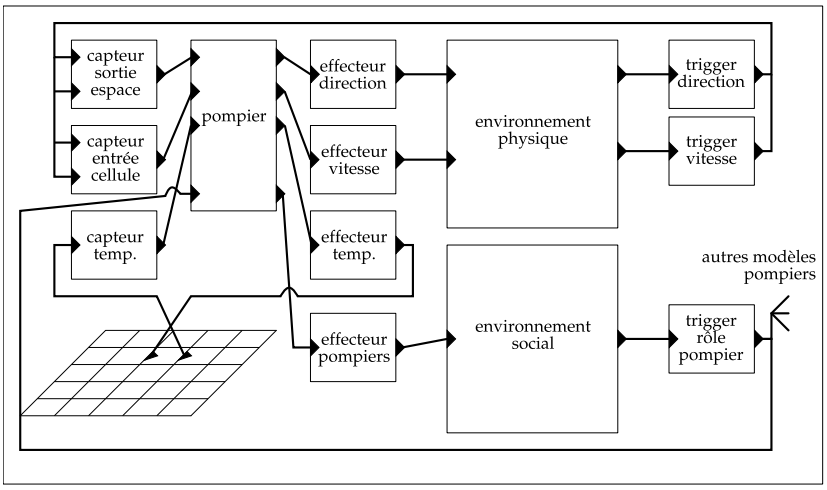


FIG. 5.8 – Représentation complète de l'interaction entre un agent pompier et les environnements physiques et sociaux.

Nous distinguons, sur cette figure 5.8, l'environnement social et deux modèles atomiques. Le premier, l'effecteur, envoie les messages du modèle agent à l'environnement à destination des agents du même rôle. Le second modèle, de type déclencheur, récupère ces messages et les envoie à tous les agents connectés à son port de sortie. Le comportement des agents doit être modifié pour prendre en compte ces modifications.

5.3.2.4 Comportement de l'agent

Le comportement de l'agent est le développement principal du modèle, nous pouvons le décrire, pour le pompier, par un ensemble de règles en fonction de l'état de l'agent :

– l'agent est positionné sur une cellule vide → déplacement aléatoire afin de trouver un incendie ou de répondre à un appel à l'aide d'un autre pompier ;

– l'agent est positionné sur une cellule en feu → refroidir la cellule ;
– l'agent est positionné sur une cellule vide et réceptionne un appel en place (x, y) → si position pas trop éloignée[2], on s'oriente vers cette cellule ;
– détection d'un feu dans les cellules voisines → appel à l'aide et début de l'extinction.

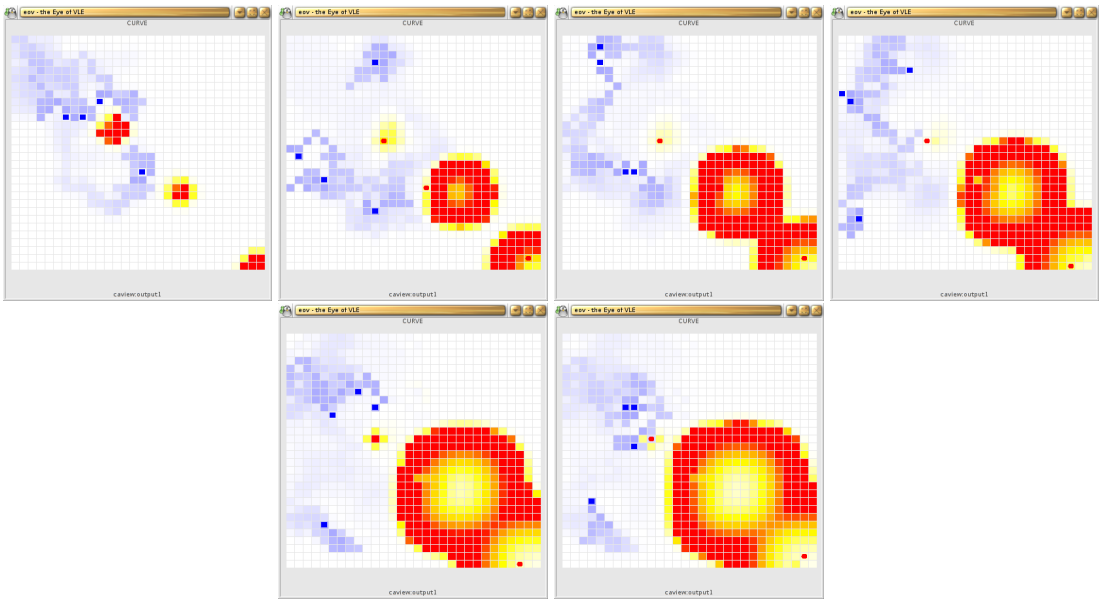


FIG. 5.9 – Cet ensemble de captures d'écrans du composant EOV du cadriciel VLE montre la simulation du modèle de la brigade de pompiers qui lutte contre un incendie déclenché par trois pyromanes.

5.3.3 Conclusion

Dans ces sections, nous avons présenté l'application de notre spécification de systèmes multi-agents développée au chapitre 3 sur un exemple historique de SMA d'une brigade de pompiers qui lutte contre un incendie de forêt. Dans le modèle réalisé, le feu est modélisé par un automate cellulaire où chaque cellule embarque un modèle QSS pour la résolution d'équations différentielles modélisant la diffusion du feu. Les pompiers et les pyromanes ont pour rôle de perturber ce système d'équations différentielles en modifiant directement les données des modèles QSS.

Cet exemple permet de suivre l'évolution de la modélisation du SMA étape par étape en utilisant des modèles pré-développés sur la plate-forme VLE. Le seul modèle à développer est la tête de l'agent. Celle-ci peut être modélisée dans les langages de programmation de l'API de VLE ou par couplage de modèles dans un modèle couplé dynamique par exemple. La seule contrainte est de respecter l'interface fournie par les

[2]La distance peut être, par exemple, calculée par : $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

ports d'entrées et de sorties du modèle.

L'une des forces de notre spécification agents développée dans VLE est l'utilisation d'un modèle *translator* étudié dans la partie 4.2.2. Ces composants simplifient énormément le travail de modélisation et de paramétrage des expériences puisqu'ils prennent en charge la création de l'ensemble de la structure DEVS du SMA ainsi que l'initialisation des modèles et la capture des états des modèles. Le modélisateur spécifie alors uniquement les relations entre les agents, les environnements et les modèles d'interactions.

5.4 Conclusion et perspectives

Ce chapitre est une illustration des capacités de développement de modèles complexes. Ces trois exemples montrent les différents outils de modélisation de la plate-forme VLE. Le premier exemple décrit l'API de l'extension Cell-DEVS [Wainer et Giambiasi, 2001] pour la génération d'automates cellulaires en DEVS. Cet exemple montre la simplicité de développement du jeu de la vie de J. Conway dans notre cadriciel. Le deuxième exemple, présenté dans le chapitre 2, est le modèle de ségrégation de T. Schelling. Cet exemple illustre l'API de VLE et de son extension DS-DEVS [Barros, 1995] pour l'optimisation du nombre d'événements, un des facteurs de limitation de DEVS en terme de vitesse que la mise à plat de la hiérarchie de modèles, présentée dans le chapitre 2, essaie de combler. Le dernier exemple montre principalement deux aspects de nos travaux, le couplage de processus continus, modélisés par un automate cellulaire Cell-DEVS en interaction avec des modèles QSS [Kofman et Junco, 2001] et un ensemble modèle où le fonctionnement est proche d'automates à états finis. Le deuxième aspect est l'utilisation de notre spécification SMA, étudiée au chapitre 3, pour utiliser le paradigme SMA dans DEVS et de manière la plus simple possible.

L'ensemble de ces travaux utilise le concept de *translator*, exposé dans la section 4.2.2, qui consiste à simplifier l'écriture et la relecture de définitions d'expériences en prenant en charge des branches de structures, de comportements, d'initialisations ou de captures d'informations de la hiérarchie de modèles en transformant une information en un XML compatible VLE. Ce concept, bien que très simple, apporte à notre plate-forme une ouverture vers d'autres programmes ou logiciels. Ainsi, une application spécifique SMA pourrait utiliser le noyau de simulation de VLE pour la réalisation des expériences tout en conservant son propre format. Le principal travail est alors le développement du composant *translator*.

Chapitre 6

Conclusion

Sommaire

Nous présentons, dans ce chapitre, un résumé des travaux effectués en nous attachant principalement aux apports réalisés au cours de cette thèse. Nous pouvons résumer ces contributions ainsi : la définition d'une nouvelle spécification de simulateur DEVS où la principale caractéristique est la mise à plat de la hiérarchie de modèles ; l'ajout au formalisme DEVS de fonctionnalités dont le but est de simplifier le développement de modèles ; la définition d'une spécification du paradigme des systèmes multi-agents, principalement des agents réactifs, en utilisant l'ensemble des spécifications précédentes ; en dernière étape, l'élaboration et l'implémentation d'un simulateur regroupant l'ensemble des définitions introduites dans cette thèse. Dans la deuxième partie de ce chapitre, nous développons des perspectives de nouveaux travaux à partir des études réalisées dans ce manuscrit.

6.1 Résumé et apports de nos travaux

Ce manuscrit de thèse est orienté vers les concepts de modélisation et de simulation autour de la problématique du couplage de modèles hétérogènes. En effet, nous pensons, comme P. A. Fishwick [Fishwick, 1995] et H. Vangheluwe [Vangheluwe *et al.*, 2002], que la multi-modélisation peut être bénéfique à la description des modèles. Dans ces travaux, nous réalisons le couplage de modèles hétérogènes [Quesnel *et al.*, 2005] en utilisant le formalisme DEVS de B. P. Zeigler [Zeigler, 1976] qui fait partie de la théorie de la modélisation et de la simulation.

DEVS, *Discrete EVent Specification*, est une spécification à base d'événements discrets. Il est basé sur une formalisation des systèmes qui a pour origine les mathématiques discrètes. Le formalisme DEVS manipule les concepts de structures, d'ensembles et de fonctions, qu'il met en relation. DEVS s'appuie, principalement, sur la notion de temps, où les événements déterminent l'avancée du temps. DEVS classe les structures en deux catégories : les modèles atomiques et les modèles couplés. Les premiers sont utilisés pour développer le comportement des modèles ; les deuxièmes, sont utilisés pour coupler des modèles entre eux et définir une hiérarchie de modèles. DEVS est ainsi un formalisme abstrait qui offre une vision modulaire et hiérarchique des systèmes dynamiques. B. P. Zeigler apporte également une version opérationnelle de DEVS, nommée simulateurs abstraits. Néanmoins, ces algorithmes ne sont pas optimaux lors de l'utilisation d'une grande hiérarchie de modèles ou lorsque le nombre d'événements est important. Cette remarque est le point d'entrée des premiers travaux réalisés dans cette thèse où nous proposons une solution en formalisant un simulateur DEVS dépourvu d'une représentation hiérarchique des modèles.

Le deuxième chapitre résume les premiers travaux réalisés durant cette thèse. Nous avons étudié une des caractéristiques de DEVS représentée par la hiérarchie de modèle. Dans ce chapitre, nous avons développé une spécification où cette hiérarchie est mise à plat du point de vue du simulateur. Nous démontrons que ce changement n'affecte pas la compatibilité avec DEVS puisqu'elle s'appuie sur l'une de ses caractéris-

tiques, l'équivalence entre un modèle couplé et un modèle atomique. Cette mise à plat de modèles a pour but de diminuer les coûts de gestion de la simulation lorsque la hiérarchie de modèles est importante et les échanges d'informations nombreux. Cependant, cette technique s'accompagne de modifications majeures lors de l'utilisation des extensions DEVS auxquelles nous apportons des solutions formelles et opérationnelles. Par exemple, lors de l'utilisation des extensions de parallélisation des calculs ou de changement dynamique de structures.

Dans la deuxième partie du chapitre, nous nous attachons à apporter au formalisme DEVS et à ses simulateurs abstraits, des méthodes permettant de réaliser une meilleure intégration dans une plate-forme de multi-modélisation.

Dans ce même chapitre, nous développons deux aspects supplémentaires pour simplifier l'écriture de modèles DEVS atomiques. Le premier porte sur les sorties des simulations auxquelles le formalisme DEVS ne répond pas de manière optimale. Nous avons étendu la spécification DEVS afin de prendre en compte les captures des modèles sans perturber leurs comportements c'est-à-dire, sans introduire de rupture des graphes d'états. C'est le concept d'événements d'états et d'observateurs. La deuxième étude a pour origine une observation très simple : beaucoup de modèles DEVS posent des questions à d'autres modèles et attendent une réponse immédiate. Le problème réside dans le fait où le modèle recevant la question doit interrompre son comportement pour gérer cette question, quelle que soit la complexité du traitement du changement de l'état. Nous avons introduit une fonction dans DEVS qui introduit les réponses instantanées permettant à un modèle de questionner un autre modèle sans que celui-ci n'aie à changer son état courant. C'est le concept d'événements instantanés. Ces fonctionnalités supplémentaires sont utilisées dans le troisième chapitre pour développer de manière plus souple une spécification pour les systèmes multi-agents.

La deuxième étape de cette thèse, présentée dans le troisième chapitre, considère une formalisation du paradigme des systèmes multi-agents [Duboz *et al.*, 2004]. Nous nous attachons principalement aux agents réactifs où les notions de temps et d'espace sont très importantes. Cette formalisation utilise les principes de DEVS, la modularité et la hiérarchie de modèles, pour définir les différentes entités. Ainsi, nous nous approchons des définitions de J. C. Soulié [Soulié, 2001], où deux types d'environnements sont utilisés pour décrire les systèmes, les environnements physiques où les agents se situent, les environnements sociaux, pour gérer les interactions entre les agents avec un système proche du concept d'Agents - Groupes - Rôles [Ferber, 1995]. Les agents possèdent deux parties distinctes définissant le comportement et les interactions avec l'environnement. Ces interactions sont réalisées par des entités nommées effecteurs et capteurs en interaction avec le ou les environnements physiques ou sociaux dans lequel l'agent se situe. Notre formalisation s'attache à un principe simple : affecter à chaque entité une classe de comportement. Le but étant de réaliser des modèles DEVS qui n'accomplissent qu'un seul type action afin de diminuer sa complexité. Le couplage des modèles DEVS gère alors les interactions entre ces différentes entités.

L'ensemble de ces travaux a fait l'objet d'une implémentation au sein du cadriciel VLE, exposée dans le quatrième chapitre. Cette plate-forme de multi-modélisation, dont l'implémentation à base de DEVS a débuté en 2002 [Quesnel *et al.*, 2003], nous a permise de valider les ajouts réalisés sur DEVS comme la mise à plat de la hiérarchie de modèles et les modifications des différentes extensions DEVS. La mise en œuvre de la spécification SMA fait également partie de la plate-forme VLE même si l'ensemble de la spécification n'est pas complète [Quesnel *et al.*, 2004b]. Cette plate-forme se base sur les principes de bus logiciel sur lequel les modèles communiquent. Ce bus est ouvert aux différents formalismes et langages via l'utilisation de capsules [Quesnel *et al.*, 2004a]. D'un point de vue technique, cette plate-forme logicielle s'appuie sur des concepts collaboratifs et sur la décomposition des fonctionnalités, permettant ainsi aux différents contributeurs de développer les fonctionnalités souhaitées de manière simple et précise. Par exemple, le développement de modèles atomiques de simulation peut être réalisé suivant trois formes, en codant le modèle, en réalisant des couplages de modèles ou en utilisant des techniques de capsules utilisant des langages de plus haut niveau sémantique.

Aujourd'hui, la plate-forme VLE est employée dans plusieurs projets de recherche. Le premier, nommé CHALOUPE (CHAngement gLObal, dynamiqUe de la biodiversité marine exploitée et viabilité des PEcheries), est un projet financé par L'Agence Nationale pour la Recherche. C'est un projet pluridisciplinaire, impliquant des organismes de recherche tels que l'IFREMER[1], l'IRD[2], le CNRS[3], le MNHN[4] et le LIL. Il a pour sujet de décrire les changements observés au cours des dernières décennies dans les systèmes de peuplement marins et de pêcheries du plateau continental amazonien de la Guyane française, du plateau continental tempéré du golfe de Gascogne et de la zone à *upwelling* du Maroc et d'identifier leurs principaux facteurs d'évolution bio-économique par l'analyse des corrélations entre tendances et modélisation des processus. Notre cadriciel est utilisé dans la deuxième partie de l'ANR. Il est employé pour réaliser des couplages entre des modèles biologiques et économiques sur des pêcheries.

Un autre projet, développé par l'IRD de Sète et plus particulièrement par R. Duboz, porte sur la modélisation de la dynamique des communautés de poissons qui vivent dans un estuaire du Sénégal. VLE est utilisé pour son aspect multi-modélisation, utilisant un SMA pour la représentation des poissons, un modèle QSS pour modéliser la capacité trophique du milieu et une capsule autour d'un modèle de résolution d'équation spatialisée pour prendre en compte l'évolution de la salinité dans l'estuaire sous l'influence des précipitations. R. Duboz commence également un nouveau travail pour l'ANR REMIGE (Réponses comportementales et démographiques prédateurs marins de l'Océan Indien aux changements globaux) où VLE est utilisé pour modéliser des SMA.

VLE est également employé dans le PNEC, Plan National d'Études Côtières en collabo-

[1] IFREMER, Institut Français de Recherche pour l'Exploitation de la MER
[2] IRD, Institut de Recherche pour le Développement
[3] CNRS, Centre National de la Recherche Scientifique
[4] MNHN, Muséum National d'Histoire Naturelle

ration avec le LIL. Ce plan vise à étudier la dynamique du copépode à petites échelles et les interactions avec un modèle de dynamique de population et plus particulièrement les interactions entre les zooplanctons et les phytoplanctons. Les modèles étudiés forment le couplage entre un modèle multi-agents, avec l'étude de systèmes multi-agents centrés individus dans un univers continu en trois dimensions, et des équations différentielles résolues par des modèles DEVS de type QSS-1.

6.2 Perspectives

Les travaux réalisés tout au long de cette thèse révèlent un nombre important de perspectives de recherches et d'évolutions de nos applications. Nous les classerons, dans les paragraphes suivants, dans le même ordre que les apports, c'est-à-dire :

– Simulateur et noyau de simulation ;
– Spécification des Systèmes Multi-Agents ;
– Plate-forme logicielle VLE.

Les premières perspectives couvrent nos travaux sur DEVS et principalement l'amélioration de l'intégration, dans notre spécification de simulateur, des extensions DEVS. La première extension concerne la parallélisation des exécutions des modèles atomiques. Pour ce travail, dont le but est de tenter de gagner du temps de simulation, nous nous baserons essentiellement sur les travaux de B. P. Zeigler [Zeigler *et al.*, 2000] et sur ceux que nous avons réalisé en 2003 dans un travail de DEA [Quesnel *et al.*, 2003]. Les principaux problèmes découlent de notre choix de mise à plat du graphe de modèles qui interdit la parallélisation des coordinateurs DEVS. Une solution, proposée dans ce manuscrit, est de déclarer une liste de coordinateurs, plutôt qu'une hiérarchie, où chaque coordinateur aurait la tâche de gérer les modèles qu'il possède. Cette solution pourrait être développée mais nécessiterait des modifications dans le coordinateur racine de notre spécification. De la même manière, l'extension DS-DEVS, pour la prise en charge des changements de structures, requiert une meilleure intégration que celle proposée dans ce manuscrit. Les solutions présentées sont nombreuses, mais souvent peu élégantes. Une voie de recherche sont les tables de routage. En effet, dans notre simulateur, lorsque le graphe de modèles est statique, il utilise une simple table de correspondances entre les ports de sorties et d'entrées des modèles. L'une des idées est de généraliser ce principe lors de l'utilisation des structures dynamiques en utilisant, par exemple, des travaux sur les tables de routages dynamiques de la couche réseau du modèle OSI[5].

L'étude de la sensibilité et de la robustesse des modèles DEVS représente également des perspectives d'études intéressantes. Un modèle DEVS, dans notre simulateur, est

[5]OSI est le modèle d'interconnexion des systèmes ouverts de l'ISO (Organisation internationale de normalisation). C'est un modèle de communications entre ordinateurs. Il décrit les fonctionnalités nécessaires à la communication et à l'organisation de ces fonctions.

paramétrable par plusieurs types d'initialisation. Cependant le modélisateur ne connaît pas toujours les relations entre les paramètres d'entrées et les sorties de son modèle. De plus, plus le modèle est complexe, plus les relations le sont. Il existe une multitude de méthodes dans le cadre déterministe mais, dès lors que le modèle est stochastique, les méthodes sont plus restreintes. L'étude de la sensibilité est très proche du calibrage de modèle. La différence réside, dans le cas de l'étude de sensibilité, à faire varier les valeurs des paramètres et d'observer la variation des sorties du modèle, dans le cas du calibrage, nous cherchons la valeur optimale des paramètres en fonction des sorties que nous nous fixons.

La construction automatique et le paramétrage de modèles font également partie des perspectives que nous voulons réaliser. Une réflexion sur l'intégration de ces éléments est en cours au sein de notre équipe MESC en utilisant la programmation génétique pour le développement et la paramétrisation des modèles atomiques via les plans d'expériences. Les possibilités d'un tel système sont multiples. Par exemple, nous pouvons étudier la généralisation des modèles couplés de modèles élémentaires en modèles plus simples ou paramétrer automatiquement l'initialisation des modèles, en effectuant des calculs sur les sorties réalisées des simulations et trouver des jeux de paramétrages correspondant aux sorties des simulations.

La spécification SMA, étudiée dans ce manuscrit, offre également des perspectives de recherches importantes. En effet, notre spécification et notre plate-forme logicielle VLE, permettent dès aujourd'hui de coupler un SMA avec un autre formalisme comme les équations différentielles. Ce concept permet de réaliser des changements d'échelles, proposés dans la section 3.4.2. La méthode employée dans cet exemple utilise deux modèles atomiques pour convertir les informations entre les deux formalismes. Il serait intéressant de généraliser ce principe avec d'autres formalismes. Une des solutions peut être d'utiliser l'approche HLA pour coupler notre simulateur VLE à un autre simulateur VLE et ainsi reproduire des couplages de formalismes différents.

La spécification, telle que nous l'avons décrite, repose sur les bases formelles et opérationnelles de DEVS. Cependant, elle est encore trop proche de ce formalisme pour être facilement utilisable. Des couches d'abstraction et de simplification peuvent être proposées. Par exemple, lors du dialogue entre les agents notre spécification, pour ce type de conversation, ne propose aucun langage. Les informations échangées sont laissées libres au modélisateur. Or, des travaux sur des langages entre agents existent et peuvent être intégrés dans notre spécification. Nous pouvons citer par exemple, le langage de communication pour agents de la FIPA[6], *the Foundation for Intelligent Physical Agents*. Les ACL[7], *Agent Communication Language*, sont des langages de communications d'une couche supérieure des protocoles de transfert bas niveau et adressent le niveau intentionnel et social des agents.

[6] http://www.fipa.org/
[7] http://www.fipa.org/repository/aclspecs.html

Dans le cadre du développement de la plate-forme VLE, de nombreux problèmes ont été évités par notre approche modulaire où tous les programmes peuvent être étendus par des composants légers de type greffon. Les possibilités de ces greffons sont très importants puisqu'ils permettent, à un contributeur, d'utiliser le cadriciel en fonction de ses compétences. Il reste cependant des ouvertures de développement, que nous classons en deux types :

- généraliser le couplage des différents formalismes et programmes à tous les modules de développement disponibles dans VLE ;
- corriger l'un des principaux reproches que nous pouvons faire à VLE, mais également à DEVS : la complexité des modèles.

Le couplage de modèles est l'un des principes de DEVS et donc de la plate-forme VLE dans son aspect simulation. Aujourd'hui, les couplages sont réalisés à plusieurs niveaux : dans les modules des simulations et dans les composants de modélisation. De nouveaux couplages doivent être proposés, par exemple, sur les systèmes de bases de données ou les systèmes d'informations géographiques. Ces programmes doivent pouvoir se coupler aux différents greffons ou modules de VLE. Si nous prenons l'exemple des bases de données et des interactions dans le cycle de modélisation et de simulation de la section 4.1.1 :

- modélisation : pour utiliser une représentation des données ou initialiser les modèles afin d'aider à la description du système étudié ;
- simulation : un modèle capsule pilote la base de données pour récupérer ou ajouter des informations en cours de simulation ;
- construction de modèles : rôle attribué aux traducteurs, ceux-ci peuvent être paramétrés par une base de données et former de nouvelles constructions ou initialisations ;
- analyse : le simulateur doit permettre de fournir les sorties des simulations aux bases de données afin que celles-ci puissent traiter ces lots d'informations.

Dans le cadre de la simulation, nous aborderons ultérieurement une meilleure gestion des plans d'expériences via le couplage du simulateur VLE et des modèles de statistiques pour une meilleure gestion des résultats. Nous avons commencé à travailler sur ces questions en collaboration avec le réseau « exploration numérique des modèles ». Celui-ci réalise un ensemble de programmes et de bibliothèques développés dans le langage R pour l'analyse et l'exploration des résultats des simulations. Pour gérer cette interaction, un module de création des plans d'expériences manipulant R doit être développé. De la même manière, l'exploration des modèles nécessite une intégration plus importante dans le module analyse de VLE.

Nous devrons explorer également la possibilité d'ouvrir la boucle de gestion événementielle du coordinateur racine de VLE. En effet, aujourd'hui, le simulateur ne peut être contrôlé que par la primitive de lancement de simulation. Celle-ci prend en charge l'avancée du temps en manipulant la pile événementielle, en dépilant et ajoutant des événements. L'ouverture de cette primitive pourrait, par exemple, permettre la gestion d'un historique des événements pour s'autoriser des retours dans le temps afin de choi-

sir de meilleurs paramètres pour les modèles. Elle permettrait également à d'autres programmes de prendre le contrôle de la simulation lorsqu'un système nécessite des informations à chaque avancée du temps sans utiliser les observations des modèles. Nous pouvons donner l'exemple du nombre d'événements dans l'échéancier qui ne peut être une sortie d'un modèle puisque cette information n'est pas disponible pour un modèle.

Le deuxième aspect majeur de VLE est de tenter d'améliorer la compréhension et l'écriture de modèles DEVS. Des travaux ont déjà fourni des outils d'aide à la construction des modèles: le premier, par le biais d'arbres d'héritages de modèles permettant de spécialiser les modèles par exemple, les modèles CellDevs et Qss, un deuxième, est de poursuivre le travail sur les modules traducteurs. Ceux-ci permettent de transformer des langages de haut niveau vers des informations compréhensibles par la plate-forme, c'est-à-dire, sous forme de fichiers XML. Ces outils permettent de simplifier le développement de modèles et de simulations, puisqu'ils utilisent des langages de plus haut niveau pour décrire les comportements, les structures et les plans d'expérience. Les traducteurs ouvrent la plate-forme VLE à d'autres types de programmes spécialisés dans des domaines particuliers.

Annexe A

XML VPZ

Cette annexe regroupe les différents formats d'applications XML employés dans les composants de la plate-forme VLE. Pour une meilleure lisibilité, les formats seront présentés sous forme d'exemple d'utilisation plutôt que sous la forme de DTD ou de Schéma XML.

A.1 Types de données

Les données sont échangées entre les composants du projet lorsque les communications utilisent le réseau TCP/IP. Ce flux de données peut être compressé au format Gzip[1] afin de diminuer la taille du flux.

A.1.1 Types simples

Les données de types simples sont des encapsulations des types primitifs disponibles dans la plupart des types de programmation. Les chaînes de caractères sont codées dans la norme UTF-8, les nombres à virgules flottantes, dans la norme IEEE 754.

```
1 <DATA TYPE="double">0.5</DATA>
2 <DATA TYPE="integer">32768</DATA>
3 <DATA TYPE="bool">true</DATA>
4 <DATA TYPE="string">texte</DATA>
```

[1] gzip, http://www.gzip.org/index-f.html est un programme de compression libre.

A.1.2 Les nombres aléatoires

Les données échangées peuvent être des nombres aléatoires. Dans ce cas, il faut préciser les types des générateurs et les paramètres associés :

```
<DATA TYPE="random" RANDOM="normal" INSTANCE="" AVERAGE=""
      STANDARD_DEVIATION="" />
<DATA TYPE="random" RANDOM="lognormal" INSTANCE=""
      AVERAGE="" STANDARD_DEVIATION="" />
<DATA TYPE="random" RANDOM="uniform" INSTANCE=""
      MIN="" MAX=""  />
```

A.1.3 Les types composés

Les types composés vont permettre de structurer les informations afin de mieux les représenter. Nous avons développé deux types composés, les listes chaînées et les tables associatives.

```
<DATA TYPE="list">
    <DATA TYPE="double">0.33</DATA>
    <DATA TYPE="list">
        <DATA TYPE="double">1.05</DATA>
        <DATA TYPE="double">2.05</DATA>
    </DATA>
    [...]
</DATA>
```

Le tableau associatif permet d'associer une donnée à une chaîne de caractères.

```
<DATA TYPE="map">
    <KEY NAME="x">
        <DATA TYPE="double">0.33</DATA>
    </KEY>
    <KEY NAME="y">
        <DATA TYPE="double">0.99</DATA>
    </KEY>
    [...]
</DATA>
```

A.2 Application XML : Structures

L'application XML VPZ relie trois applications XML distinctes basées sur les travaux de R. Duboz [Duboz, 2002] : une définition de la structure de couplage entre modèles, une définition des comportements de modèles atomiques, et la définition du plan d'expérience d'où découlent les instances d'expériences. Nous commencerons par définir la structure :

```
<?xml version="1.0"?>
<MODEL NAME="nom" TYPE="coupled" >
<!-- Définition d'un modèle couplé. -->

  <SUBMODELS>
  <!-- Définit la liste de ses sous-modèles. -->

    <MODEL NAME="nom1" TYPE="coupled" >
    <!-- Un modèle couplé peut contenir un ou plusieurs modèles
         couplés. -->
         [...]
    </MODEL>

    <MODEL NAME="nom2" TYPE="atomic" DYNAMICS="dyn0" INIT="init0" >
    <!-- Un modèle atomique définit ses ports ainsi que le couple
         (comportement associé, initialisation). La définition des ports
         de R.~Duboz intègre une gestion des types de données pouvant être
         reçus. Nous ne décrivons pas cette syntaxe ici. -->
      <IN>
        <PORT NAME="i1" />
        [...]
      </IN>
      <OUT>
        <PORT NAME="o1" />
        [...]
      </OUT>
      <INIT>
        <PORT NAME="i1" />
        [...]
      </INIT>
      <STATE>
        <PORT NAME="s1" />
        [...]
      </STATE>
    </MODEL>
    [...]
  </SUBMODELS>

  <CONNECTIONS>
  <!-- Définition de la liste des connexions entre les différents
       sous-modèles du modèle couplé. -->

    <CONNECTION TYPE="internal | input | output">
    <!-- Définition de la connexion : internal pour une connexion interne
         au modèle couplé, input entre une entrée du modèle couplé et un
         modèle interne, output entre un port de sortie du modèle couplé
         et un modèle interne. -->
      <ORIGIN MODEL="titi1" PORT="out" />
      <!-- Définition du modèle et du port d'origine de la connexion. -->
      <DESTINATION MODEL="titi2" PORT="in" />
      <!-- Définition du modèle et du port de destination de
           la connexion. -->
    </CONNECTION>
```

```
    [...]
  </CONNECTIONS>

  <!-- La définition des modèles du réseau est réalisée, définition des
       ports du modèle couplé. -->
  <IN>
    <PORT NAME="name" />
    [...]
  </IN>
  <OUT>
    <PORT NAME="name" />
    [...]
  </OUT>
</MODEL>
```

A.3 Application XML : Dynamiques

L'application XML dynamique permet de définir l'ensemble des comportements de tous les modèles de la simulation. Elle est représentée par une liste de comportements dans laquelle le développeur peut saisir n'importe quel type d'information.

```
<?xml version="1.0"?>
<DYNAMICS>
<!-- Définition de la liste des comportements. -->

  <DYNAMIC FORMALISM="name" TYPE="mapping | wrapping | distant">
  <!-- Définit la dynamique de modèle atomique en utilisant le composant
       de simulation name. Le contenu de la balise DYNAMIC est laissé
       libre au développeur du composant de simulation.  -->
  </DYNAMIC>
  [...]
</DYNAMICS>
```

A.4 Application XML : Expériences

L'application XML expériences permet de définir les plans d'expériences ainsi que les instances. Elle définit une graine pour le générateur de nombre pseudo- aléatoire, une durée de simulation etc.

```
<?xml version="1.0"?>
<EXPERIMENT NAME="exp1" DURATION="500" DATE="08-04-2004" >
<!-- Définit une expérience avec un nom, une durée de simulation et la
     date de génération de expérience. -->

  <REPLICAS NUMBER="number" SEED="random | enumerate | list" BEGIN="0">
  <!-- Définition de la liste des répliquas à réaliser. Le paramètre
       SEED permet de préciser le type : random, la liste est générée
```

```
9       aléatoirement, list, la liste est continue et nécessite la balise
10      BEGIN, 0 par défaut, et enumerate, la liste est énumérée par les
11      balises REPLICA. -->
12    <REPLICA NUMBER="1" SEED="53455" />
13    <REPLICA NUMBER="2" SEED="123" />
14     [...]
15  </REPLICAS>
16
17  <EXPERIMENTAL_CONDITIONS>
18  <!-- Les conditions expérimentales regroupent les informations sur
19       l'initialisation des comportements. -->
20
21    <CONDITION INIT="init0" PORT_NAME="i1">
22    <!-- Établissement des conditions initiales pour le modèle dont le
23         nom est MODEL_NAME et pour le port nommé PORT_NAME. -->
24
25      <DOUBLE VALUE="0.1" />
26      <!-- Initialisation du port avec un DOUBLE de valeur 0.1.
27           Les types de données sont : BOOLEAN, DOUBLE, INTEGER, SET. -->
28      [...]
29    </CONDITION>
30    ...
31  </EXPERIMENTAL_CONDITIONS>
32  <MEASURES>
33  <!-- Les mesures à effectuer pendant la simulation via les
34       observateurs. -->
35    <OUTPUTS>
36    <!-- Définit la liste des différents types de sorties des
37         observateurs. -->
38      <OUTPUT NAME="output1" FORMAT="text" LOCATION="">
39      <!-- Définit un type de sortie avec un nom. Deux formats sont
40           disponibles texte, pour une sortie textuel des observateurs
41           et SDML pour une sortie sous forme d'un arbres XML.
42           LOCATION définit le l'adresse où se retrouveront les
43           fichiers de sortie. -->
44      </OUTPUT>
45      [...]
46    </OUTPUTS>
47
48    <MEASURE NAME="c" TYPE="event" TIME_STEP="0.04" OUTPUT="output1">
49    <!-- Définit les observateurs avec un nom et un type connecté à une
50         sortie. Le type d'observation peut être événementiel
51         TYPE=event ou daté TYPE=timed. Dans ce dernier cas, il
52         faut renseigner un champ TIME_STEP.  -->
53
54      <OBSERVABLE MODEL_NAME="titi1" PORT_NAME="c" />
55      <!-- Définition d'une liste d'observations, c'est-à-dire un couple
56           de modèles, port d'état. -->
57       [...]
58    </MEASURE>
59    [...]
60  </MEASURES>
61 </EXPERIMENT>
```

Bibliographie

[Allen et Starr, 1982] ALLEN, T. F. H. et STARR, T. B. (1982). *Hierarchy perspectives for ecological complexity*. University of Chicago Press, Chicago, Illinois, USA.

[Barros, 1995] BARROS, F. J. (1995). Dynamic structure discrete event system specification: A new modelling and simulation formalism for dynamic structure systems. *Dans Proceedings of the 1995 Winter Simulation Conference*, pages 781–785.

[Barros, 1996] BARROS, F. J. (1996). Dynamic structure discret event system specification: Formalism, abstract simulators and applications. *Transaction of the Society for Computer Simulation*, 13(1):35–46.

[Barros, 1997] BARROS, F. J. (1997). Modelling formalisms for dynamic structure systems. *ACM Transactions on Modelling and Computer Simulation Science*, 7:501–515.

[Barros, 1998a] BARROS, F. J. (1998a). Abstract simulators for the dsde formalism. *Dans Proceedings of the 1998 Winter Simulation Conference*, pages 407–412, Washington DC, USA.

[Barros, 1998b] BARROS, F. J. (1998b). Handling simultaneous events in dynamic structure models. *Dans Proceedings of SPIE 12th Annual International Symposium on Aerospace / Defense Sensing, Simulation and Controls: Enabling Technology for Simulation Science*, pages 355–363.

[Becker *et al.*, 1988] BECKER, R. A., CHAMBERS, J. M. et WILKS, A. R. (1988). *The New S Language*. Chapman and Hall.

[Benson *et al.*, 2004] BENSON, C., ELMAN, A., NICKEL, S. et C.ROBERTSON (2004). *GNOME Human Interface Guideline 2.0 – The GNOME Usability Projet*. GNOME Documentation Project. Disponible librement sous licence FDPL sur le site `http://developer.gnome.org/projects/gup/hig/`.

[Bertalanffy, 1968] BERTALANFFY, L. V. (1968). *Théorie générale des systèmes*. Dunod.

[Booch *et al.*, 1997] BOOCH, G., RUMBAUGH, J. et JACOBSON, I. (1997). *Unified Modelling Language User Guide*. ed. Addison Wesley.

[Borland et Vangheluwe, 2003] BORLAND, S. et VANGHELUWE, H. (2003). Transforming statecharts to devs. *Dans Summer Computer Simulation Conference, Student Workshop - Society for Computer Simulation International*, pages 154 – 159, Montréal, Canada. A. Bruzzone and Mhamed Itmi editors.

[Bousquet *et al.*, 1994] BOUSQUET, F., CAMBIER, C. et MORAND, P. (1994). Distributed artificial intelligence and object-oriented modelling of a fishery. *Mathl. Comput. Modelling*, 20:97–107.

[Box *et al.*, 1978] BOX, G., HUNTER, W. et HUNTER, J. (1978). *Statistics for Experimenters*. Wiley.

[Chow et Barros, 1994] CHOW, A. C. et BARROS, F. J. (1994). Abstract simulator for the parallel devs formalism. *Dans Proceedings of the Fifth Annual Conference on AI, Simulation and Planning in High Autonomy Systems*, pages 157–163.

[Cochran et Cox, 1957] COCHRAN, W. G. et COX, G. M. (1957). *Experimental Designs*. John Wiley & Sons.

[Coquillard et Hill, 1997] COQUILLARD, P. et HILL, D. (1997). *Modélisation et simulation d'écosystèmes*. ed. Masson.

[de Lara et Vangheluwe, 2002] DE LARA, J. et VANGHELUWE, H. (2002). AToM3: A tool for multi-formalism and meta-modelling. *Dans Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering table of contents*. Springer-Verlag, London, UK.

[Duboz, 2004] DUBOZ, R. (2004). *Intégration de modèles hétérogènes pour la modélisation et la simulation de systèmes complexes : application au transfert d'échelles en écologie marine*. Thèse de doctorat, Université du Littoral Côte d'Opale.

[Duboz, 2002] DUBOZ, R. (april 7-10, 2002). XML for the representation of semantic in model coupling. *Dans* BARROS, F. J. et GIAMBIASI, N., éditeurs : *AIS'2002. AI, Simulation and Planning in High Autonomy Systems*, pages 267–270, Lisboa, Portugal.

[Duboz *et al.*, 2002] DUBOZ, R., RAMAT, É. et GIAMBIASI, N. (2002). Utilisation du formalisme DEVS pour la spécification de systèmes d'agents réactifs. *Dans Actes des Journées Francophones en Intelligence Artificielle et Systèmes Multi Agents (JFIADSMA)*, pages 99–102, Lille, France. Hermès.

[Duboz *et al.*, 2001] DUBOZ, R., RAMAT, É. et PREUX, P. (2001). Towards a coupling of continuous and discrete formalism in ecological modelling: influences of the choice of algorithms on results. *Dans Proceedings of the 13th European Symposium on Simulation*, pages 481–487, Marseille, France.

[Duboz *et al.*, 2003] DUBOZ, R., RAMAT, É. et PREUX, P. (2003). Scale transfer modelling: Using emergent computation for coupling an ordinary differential equation system with a reactive agent model. *Systems Analysis Modelling Simulation*, 43(6):793–814.

[Duboz *et al.*, 2004] DUBOZ, R., RAMAT, É. et QUESNEL, G. (2004). Systèmes multi-agents et théorie de la modélisation et de la simulation : une analogie opérationnelle. *Dans* BOISSIER, O. et GUESSOUM, Z., éditeurs : *Actes des douzièmes Journées Francophones sur les Systèmes Multi-Agents (JFSMA) - Systèmes multi-agents défis scientifiques et nouveaux usages*, Paris.

[Ferber, 1995] FERBER, J. (1995). *Les Systèmes Multi-Agents, vers une intelligence collective*. Inter-Éditions.

[Ferber et Gutknecht, 1998] FERBER, J. et GUTKNECHT, O. (1998). A meta-model for the analysis and design of organisations in multi-agent systems. *Dans 3rd International Conference on Multi-Agent Systems*, pages 128–135, Paris, France. ICS Press.

[Ferber et Gutknecht, 2000] FERBER, J. et GUTKNECHT, O. (2000). Pour une sémantique opérationnelle des systèmes multi-agents. *Dans Actes des 8eJFIADSMA, Journées Francophones pour l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents*, pages 39–55.

[Fianyo *et al.*, 2001] FIANYO, E., TREUIL, J. P. et PINSON, S. (2001). OSIRIS system : An Architecture for coupling Environmental Processes with Agents. *Dans* GIAMBIASI, N. et FRYDMAN, C., éditeurs : *13th European Simulation Symposium*, Marseille, France. SCS European Publishing House.

[Filippi *et al.*, 2002] FILIPPI, J., BISGAMBIGLIA, P. et DELHOM, M. (2002). Neuro-devs, an hybrid methodology to describe complex systems. *Dans Processings of the Society for Computer Simulation International European Symposium on Simulation 2002 conference on simulation in industry*, volume 1.

[Filippi et Bisgambiglia, 2003] FILIPPI, J.-B. et BISGAMBIGLIA, P. (2003). JDEVS: An implementation of a DEVS based formal framework for environmental modelling. citeseer.ist.psu.edu/filippi03jdevs.html.

[Fishwick, 1993] FISHWICK, P. A. (1993). A simulation environment for multimodelling. *Discrete Event Dynamic Systems: Theory and Applications*, 3:151–171.

[Fishwick, 1995] FISHWICK, P. A. (1995). *Simulation Model Design and Execution*. Prentice Hall.

[Free Software Fundation, 1984] Free Software FUNDATION (1984). GNU Operating System: GNU's Not Unix. http://www.gnu.org.

[Free Sofware Fundation, 1997] Free Sofware FUNDATION (1997). Gnome: GNU Network Object Model Environment, the Free Software Desktop Project. http://www.gnome.org.

[Gamma *et al.*, 1995] GAMMA, E., HELM, R., JOHNSON, R. et VLISSIDES, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software. Reading*. Addison-Wesley.

[Gardner, 1970] GARDNER, M. (1970). Mathematical games. the fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223:120–123.

[Giambiasi *et al.*, 2000] GIAMBIASI, N., ESCUDÉ, B. et GOSH, S. (2000). Gdevs: A generalized discrete event specification for accurate modelling of dynamic systems. *Transaction of the Society for Computer Simulation International*, 17(3):120–134.

[Glinsky et Wainer, 2002] GLINSKY, E. et WAINER, G. (2002). Definition of Real Time simulation in CD++ toolkit. *Dans Proceedings of the 2002 Summer Computer Simulation Conference*, San Diego, USA.

[Grimm, 1999] GRIMM, V. (1999). Ten years of individual-based modelling in ecology: what we have learned and what could we learn in the futur. *Ecological Modelling*, 115:129–148.

[Gruer *et al.*, 2002] GRUER, P., HILAIRE, V., KOUKAM, A. et CETNAROWICZ, K. (2002). A formal framework for multi-agent systems analysis and design. *Expert Systems with Applications*, 23(4):349–355.

[Gutknecht *et al.*, 2001] GUTKNECHT, O., FERBER, J. et MICHEL, F. (2001). Integrating tools and infrastructures for generic multi-agent systems. *Dans Processings of the Fifth International Conference on Autonomous Agent*, pages 441–448, Montreal, Canada. J. P. Müller, E. Andre, S. Sen and C. Frasson.

[Harold et Means, 2002] HAROLD, E. R. et MEANS, W. S. (2002). *XML in a Nutshell, 2nd Edition*. O'Reilly.

[Hill, 1996] HILL, D. (1996). *Object-Oriented Analysis and Simulation*. ed. Addison-Wesley.

[Hocaoglu *et al.*, 2002] HOCAOGLU, M. F., FIRAT, C. et SARJOUGHIAN, H. (2002). DEVS/RAP : Agent-based simulation. *Dans Proceedings of the Internationnal Conference on AI, Simulation and Planning in High Autonomy Systems*, pages 117–121, Lisbon, Portugal.

[Holt et Winter, 2000] HOLT, R. C. et WINTER, A. (2000). A short introduction to the GXL software exchange format. *Dans Proceedings of the Seventh Working Conference on Reverse Engineering*, page 299, Washington, DC, USA. IEEE Computer Society.

[Holt *et al.*, 2000] HOLT, R. C., WINTER, A. et SCHRR, A. (2000). GXL: Toward a standard exchange format. *Dans Proceedings of the Seventh Working Conference on Reverse Engineering*, page 162, Washington, DC, USA. IEEE Computer Society.

[Huhngs et Singh, 1998] HUHNGS, M. N. et SINGH, M. P. (1998). Agents and multi-agent systems: themes, approaches and challenges. *Reading in agents*, pages 1–23.

[Insightful, 1988] INSIGHTFUL (1988). S-PLUS is a general purpose statistics package using the S programming language. http://www.insightful.com/products/splus/default.asp.

[Jacobson et Booch, 1999] JACOBSON, I. et BOOCH, G. (1999). *The Unified Software Development Process*. Addison Wesley Professional.

[Jacobson *et al.*, 1997] JACOBSON, I., BOOCH, G. et RUMBAUGH, J. (1997). *The Objectory Software Development Process*. ed. Addison Wesley.

[Jefferson, 1985] JEFFERSON, D. (1985). Virtual time. *ACM Transactions on Programming Languages and Sytems*, 7(3):404–425.

[Jensen, 1997] JENSEN, K. (1997). Coloured petri-nets basic concepts, analysis methods and practical use. *Dans Monographs in Theoretical Computer Science*, volume 1,2,3. Springer-Verlag.

[Kim et Kim, 1998] KIM, J. Y. et KIM, T. G. (1998). A heterogeneous simulation framework based on the DEVS bus and the High Level Architecture. *Dans Winter Simulation Conference*, Washington, DC.

[Klir, 1985] KLIR, G. (1985). *Architecture of System Problem Solving*. Plenum Press.

[Knuth, 1975] KNUTH, D. E. (1975). *The Art of Computer Programming – Sorting and Searching*, volume 3. Addison-Wesley, Massachusetts.

[Kofman, 2002] KOFMAN, E. (2002). A second order approximation for devs simulation of continuous systems. *Journal of the Society for Computer Simulation International*, 78(2).

[Kofman et Junco, 2001] KOFMAN, E. et JUNCO, S. (2001). Quantized State Systems. a DEVS Approach for Continuous Systems Simulation. *Dans Transactions of SCS.*, volume 18, pages 123–132.

[Kuhn, 1972] KUHN, T. (1972). *La structure des révolutions scientifiques*. Flammarion. Nouvelle traduction par Laure Meyer 1983.

[Lamport, 1978] LAMPORT, L. (1978). Time, clocks and the ordering of events in a distributed system. *Communications of the Association of the Computing Machinery*, 21(7):558–565.

[Le Moigne, 1977] LE MOIGNE, J. L. (1977). *La théorie du système général - Théorie de la modélisation*. Presses Universitaires de France.

[L'Ecuyer, 1998] L'ECUYER, P. (1998). *Handbook on Simulation*, chapitre Random Number Generation, pages 93–137. Jerry Banks Ed.

[L'Ecuyer, 2001] L'ECUYER, P. (2001). Software for uniform random number generation: Distinguishing the good and the bad. *Dans* PRESS, I., éditeur : *Proceedings of the 2001 Winter Simulation Conference*, pages 95–105.

[Matsumoto et Nishimura, 1998] MATSUMOTO, T. et NISHIMURA, T. (1998). Mersene twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *Dans ACM Trans. on Modelling and Computer Simulation*, volume 8, pages 3–30.

[McIlroy, 1968] MCILROY, M. D. (1968). Mass produced software components. *Dans* NAUR, P. et RANDELL, B., éditeurs : *Proceedings of NATO Software Engineering Conference*, pages 138–150, Garmisch, Germany.

[Minar *et al.*, 1996] MINAR, N., BURKHART, R., LANGTON, C. et ASKENAZI, M. (1996). The swarm simulation system : a toolkit for building multi-agent simulations.

[Neter *et al.*, 1990] NETER, J., WASSERMAN, W. et KUTNER, M. H. (1990). *Applied Linear Statistical Models*. Irwin.

[Nutaro, 2003] NUTARO, J. (2003). adevs, A Discrete EVent System simulator. http://www.ornl.gov/1qn/adevs/.

[Odell *et al.*, 2003] ODELL, J., PARUNAK, H. et FLEISCHER, M. (2003). The Role of Roles in Designing Effective Agent Organizations. *Dans Sofware Engineering for Large-Scale MultiAgent Systems*. A. Garcia and al.

[Pavé, 1994] PAVÉ, A. (1994). *Modélisation en Biologie et en Écologie*. Aléas.

[Peterson, 1977] PETERSON, J. L. (1977). Petri nets. *Computing Surveys*, 9(3):223–252.

[Quesnel, 2004] QUESNEL, G. (2004). Vers la simulation distribuée et à événements discrets d'entités spatialisées. Mémoire de D.E.A., Université du Littoral Côte d'Opale.

[Quesnel *et al.*, 2004a] QUESNEL, G., DUBOZ, R. et RAMAT, É. (2004a). DEVS wrapping: A study case. *Dans Proceedings of Conference on Conceptual Modelling and Simulation 2004*, pages 374–382, Genoa, Italy.

[Quesnel *et al.*, 2004b] QUESNEL, G., DUBOZ, R. et RAMAT, É. (2004b). Environnement de modélisation et de simulation de systèmes hétérogènes : Applications aux systèmes multi-agents. *Dans* et ZAHIA GUESSOUM., O. B., éditeur : *Actes des douzièmes Journées Francophones sur les Systèmes Multi-Agents (JFSMA) - Systèmes multi-agents défis scientifiques et nouveaux usages*, Paris.

[Quesnel *et al.*, 2005] QUESNEL, G., DUBOZ, R., VERSMISSE, D. et RAMAT, É. (2005). DEVS coupling of spatial and ordinary differential equations: VLE framework. *Dans Proceedings of OICMS 2005 conference*, Clermont Ferrand, France.

[Quesnel *et al.*, 2003] QUESNEL, G., NOLOT, F., DUBOZ, R. et RAMAT, É. (2003). Vers la simulation distribuée et à événements discrets d'entités spatialisées. *Dans Proceedings of MajecStic 2003 conference*, Marseille, France.

[R Development Core Team, 2006] R DEVELOPMENT CORE TEAM (2006). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

[Ramat, 2003] RAMAT, É. (2003). Contributions à la modélisation et à la simulation des systèmes complexes. Habilitation à diriger des recherches.

[Ramat et Preux, 2003] RAMAT, É. et PREUX, P. (2003). Virtual Laboratory Environment (VLE): a software environment oriented agent and object for modelling and simulation of complex systems. *Dans Simulation Modelling Practice and Theory*, volume 11, pages 45–55.

[Ramat *et al.*, 1998] RAMAT, É., PREUX, P., SEURONT, L. et LAGADEUC, Y. (1998). Modélisation et simulation multi-agents en biologie marine. etude du comportement du copépode. *Dans Colloque Joint Conference on Multi-Agent Modelling for Environmental Management, Cemagref*, pages 35–49, Clermont-ferrand, France.

[Rudnick et Gaspari, 2004] RUDNICK, J. et GASPARI, G. (2004). *Elements of the Random Walk - An introduction for Advanced Students and Researchers*. Cambridge University Press.

[Rumbaugh *et al.*, 1997] RUMBAUGH, J., JACOBSON, I. et BOOCH, G. (1997). *Unified Modelling Language Reference Manual*. ed. Addison Wesley.

[Sarjoughian et Zeigler, 1998] SARJOUGHIAN, H. et ZEIGLER, B. (1998). DEVSJava : Basis for a DEVS-based collaborative ms environment. *Dans Actes de la conférence 1998 Society of Computer Simulation International Conference on Web-Based Modelling and Simulation*, volume 5, pages 29–36, San Diego. SCS The Society for Modelling and Simulation International.

[Schelling, 1971] SCHELLING, T. (1971). Dynamic Models of Segregation. *Journal of Mathematical Sociology*, 1:143–186.

[Silva et Lucena, 2004] SILVA, V. T. D. et LUCENA, C. J. P. D. (2004). From a conceptual framework for agents and objects to a multi-agent system modelling language. *Autonomous Agents and Multi-Agent Systems*, 9:145–189.

[Smuts, 1926] SMUTS, J. C. (1926). *Holism and Evolution*. Macmillan & Co Ldt London.

[Soulié, 2001] SOULIÉ, J. C. (2001). *Vers une approche multi-environnement pour les agents*. Thèse de doctorat, Université de la Réunion.

[Stroustrup, 1986] STROUSTRUP, B. (1986). *The C++ Programming Language*. Addison Wesley.

[Szyperski, 1998] SZYPERSKI, C. (1998). *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, Massachusetts, USA.

[Traore et Hill, 2001] TRAORE, M. K. et HILL, D. R. C. (2001). The use of random number generation for stochastic distributed simulation : application to ecological modelling. *Dans* GIAMBIASI, N. et FRYDMAN, C., éditeurs : *13th European Simulation Symposium*, Marseille, France. Society for Computer Simulation European Publishing House.

[Uhrmacher, 2001] UHRMACHER, A. M. (2001). Dynamics Structure in Modeling and Simulation - A Reflective Approach. *ACM Transaction on Modeling and Simulation*, 11(2):206–232.

[Uhrmacher et Arnold, 1994] UHRMACHER, A. M. et ARNOLD, R. (1994). Distributing ans maintaining knowledge: Agents in variable structure environment. *Dans Proceeding of the 5th Annual Conference on Artificial Intelligence, Simulation and Planning in High Autonomy Systems*, pages 178–184.

[Uhrmacher et Kullick, 2000] UHRMACHER, A. M. et KULLICK, B. (2000). "plug and test" - software agents in virtual environments. *Dans the 2000 Winter Simulation Conference*, pages 1722–1729. J. Joines and R. Barton and K. Kang and P. Fishwick.

[van Rossum, 1989] van ROSSUM, G. (1989). The python programming language. http://www.python.org.

[Vangheluwe *et al.*, 2002] VANGHELUWE, H., LARA, J. et MOSTERMAN, P. J. (2002). An introduction to multi-paradigm modelling and simulation. *Dans* BARROS, F. et GIAMBIASI, N., éditeurs : *AIS'2002. Simulation and Planning in High Autonomy Systems*, pages 9–20, Lisbon, Portugal. Society for Modelling and Simulation International.

[Versmisse et Ramat, 2005] VERSMISSE, D. et RAMAT, É. (2005). Management of perturbations within a spatialized differential equations system. *Dans European Simulation and Modelling Conference*, pages 520–524, Porto, Portugal. SCS.

[von Neumann et Burks, 1966] VON NEUMANN, J. et BURKS, A. W. (1966). *Theory of Self-Reproducing Automata*. University of Illinois Press.

[Wainer, 2002] WAINER, G. (2002). CD++: a toolkit to develop DEVS models. *Software – Practice and Experience*, 32:1261–1306.

[Wainer, 2006] WAINER, G. (2006). DEVS tools. http://www.sce.carleton.ca/faculty/wainer/standard/tools.htm.

[Wainer et Giambiasi, 2001] WAINER, G. A. et GIAMBIASI, N. (2001). Application of the Cell-DEVS Paradigm for Cell Spaces Modelling and Simulation. *Dans Simulation*, volume 76, pages 22–39.

[Weiss, 1999] WEISS, G. (1999). *Multiagent Systems. A modern approach to distributed artificial intelligence*. MIT Press.

[Winter *et al.*, 2002] WINTER, A., KULLBACH, B. et RIEDIGER, V. (2002). An overview of the gxl graph exchange language. *Dans Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK. Springer-Verlag.

[Wolfram, 1986] WOLFRAM, S. (1986). *Theory and Applications of Cellular Automata*. World Scientific.

[Wooldridge *et al.*, 2000] WOOLDRIDGE, M., JENNINGS, N. R. et KINNY, D. (2000). The gaia methodology for agent-oriented analysis and design. *Autonomous Agent Multi-Agent Systems*, 3(3):285–312.

[Zambonelli *et al.*, 2003] ZAMBONELLI, F., JENNINGS, N. R. et WOOLDRIDGE, M. (2003). Developing multi-agent systems: The gaia methodology. *ACM transaction on Software Enginneering and Methodology*, 12(3):317–370.

[Zeigler *et al.*, 1999] ZEIGLER, B., HALL, S. et SARJOUGHIAN, H. (1999). Exploiting HLA and DEVS to promote interoperability and reuse in lockheed's corporate environment. *Simulation, Special Issue on The High Level Architecture*, 74(4):288–295.

[Zeigler, 1976] ZEIGLER, B. P. (1976). *Theory of Modeling and Simulation*. Wiley Interscience.

[Zeigler, 1984] ZEIGLER, B. P. (1984). *Theory of Modeling and Simulation*. Krieger Publishing Compagny. 2nd Edition.

[Zeigler *et al.*, 2000] ZEIGLER, B. P., KIM, D. et PRAEHOFER, H. (2000). *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press.

[Zeigler *et al.*, 1996] ZEIGLER, B. P., MOON, Y., KIM, D. et KIM, J. G. (1996). DEVS-C++: A high performance modelling and simulation environment. *Dans Hawaii International Conference on System Sciences*, volume 1, pages 350–359.

[Zeigler *et al.*, 1995] ZEIGLER, B. P., SONG, H. S., KIM, T. G. et PRAEHOFER, H. (1995). DEVS framework for modelling, simulation, analysis, and design of hybrid systems. Lecture Notes in Computer Science.

Résumé

Dans le cadre de la recherche scientifique, nous assistons depuis quelques années à un essor de la multi-modélisation, c'est-à-dire, l'utilisation de la combinaison de différents formalismes pour l'étude des systèmes complexes où l'informatique se présente comme l'un des supports théoriques ou opérationnels pour son développement. La problématique étudiée dans cette thèse découle de ces travaux. Nous étudions, en particulier, le couplage de modèles hétérogènes en utilisant la spécification à événements discrets de B. P. Zeigler, DEVS, *Discrete EVent System specification*. Nos travaux portent principalement sur les ajouts, à ce formalisme, d'outils aussi bien formels qu'opérationnels pour simplifier le couplage ou le développement de modèles. Nous étudions également la construction de plans d'expériences, le paramétrage de modèles et la distribution de simulation. De ces travaux sont nées des collaborations avec des laboratoires de biologie où la nécessité de la création d'une spécification formelle des Systèmes Multi-Agents, principalement centré individu, est apparue. Nous proposons cette spécification en nous basant sur les travaux de J. C. Soulié et les environnements multiples et R. Duboz pour le comportement des agents. L'ensemble des objets manipulés par les SMA est défini suivant une approche modulaire où chaque entité est spécialisée dans un domaine. Afin de mettre en œuvre tous les concepts étudiés dans cette thèse, une plate-forme logicielle VLE, *Virtual Laboratory Environment*, recouvrant tous les aspects de la multi-modélisation, a été développée. Cette plate-forme est développée suivant un principe de décomposition où chaque composant a une tâche spécifique : modélisation, simulation, analyse et la définition de plan d'expériences.

Mots clés : Multi-modélisation, Simulation, DEVS, Systèmes Multi-Agents

Within the field of scientific research, we assist, since a few years, a rapid growth of the multimodeling activities - i.e the use of various formalisms combination in order to study complex systems where computer science is an operational, or theoretical, support to tackle their studies. The problematic presented in this thesis work follows from these works. We study, in particular, the coupling of heterogeneous models using a discrete event specification proposed by B. P. Ziegler: DEVS acronym of *Discrete EVent System specification*. Our works provide, mainly, tools as well as operational in order to simplify the coupling, as the models development. We also study the build of experimental design, the setup of models, and the distribution of simulations. Thanks to these works, collaborations with biology labs have started. Following their needs, we have to create a formal specification of multiagents systems, individual based models more precisely. We propose a specification that relies on the works of J. C. Soulié with the concept multiple environments; and works of R. Duboz for the agents behaviour. All the objects used in multiagents systems are defined using a modular approach where each entity is specialised into a dedicated domain. In order to develop all the concepts studied in this thesis work, a platform called VLE (acronym of *Virtual Laboratory Environment*) has been programmed. This platform covers all the fields of the multimodelling activities and relies on the decomposition principle. Each component performs a dedicated task: modelling, simulation, and experimental design definition and analysis.

Keywords: Multimodelling, Simulation, DEVS, Multi-Agents Systems