Numéro d'ordre : 9249

Université Paris-Sud XI
UFR Scientifique d'Orsay

Thèse

présentée par

Cécile HARDEBOLLE

pour obtenir le grade de Docteur en Sciences
de l'Université Paris-Sud XI Orsay
Spécialité : Informatique

Composition de modèles pour la modélisation multi-paradigme du comportement des systèmes

Soutenue publiquement le 28 Novembre 2008 devant la commission d'examen :

Rapporteur	M. Charles ANDRE	Université de Nice Sophia Antipolis
Rapporteur	M. Christian ATTIOGBE	Université de Nantes
Co-directeur de thèse	M. Frédéric BOULANGER	Supélec
Co-directeur de thèse	M. Dominique MARCADET	Supélec
Examinateur	Mme. Marie-Agnès PERALDI	Université de Nice Sophia Antipolis
Examinateur	M. Marc POUZET	Université Paris Sud
Examinateur	M. François TERRIER	INSTN, CEA Saclay
Directeur de thèse	M. Guy VIDAL-NAQUET	Université Paris Sud et Supélec

Thèse préparée au sein du
Département Informatique de Supélec,
3 rue Joliot-Curie,
91192 Gif-Sur-Yvette Cedex,
FRANCE

Document mis en page avec le logiciel LaTeX 2_ε et la classe `book`.
Numéro de révision du document : 174 (12-02-2009)
Fichier PDF généré le 9 mars 2009.

Remerciements

Je tiens à remercier :

Guy Vidal-Naquet, mon directeur de thèse, Professeur à l'Université Paris-Sud et à Supélec, pour ses conseils, sa grande culture scientifique, et pour ses apports toujours originaux à nos discussions. Merci d'avoir su me guider dans les différentes étapes de la thèse.

Frédéric Boulanger et Dominique Marcadet, mes encadrants de thèse, Enseignants-Chercheurs à Supélec, qui m'ont guidée et conseillée au quotidien. Frédéric m'a fait découvrir avec passion le domaine de la modélisation système et a su m'enthousiasmer pour les problèmes liés à l'hétérogénéité des modèles. Nos très nombreuses discussions ont été l'inspiration principale des travaux présentés dans ce mémoire. Dominique a su m'apporter son éclairage de spécialiste sur les questions liées à l'approche MDA (Model Driven Architecture) et à la méta-modélisation.

Charles André, Professeur à la Faculté des Sciences de l'Université Nice Sophia Antipolis, et Christian Attiogbé, Professeur à la Faculté des Sciences et des Techniques de l'Université de Nantes, d'avoir accepté la lourde tâche de rapporter sur ce travail. Leur relecture attentive et leurs observations judicieuses m'ont été d'une grande aide.

Marie-Agnès Péraldi, Maître de conférences à l'Université de Nice, Marc Pouzet, Professeur à l'Université Paris-Sud et François Terrier, Professeur INSTN et Directeur de laboratoire au CEA, d'avoir accepté de participer au jury de cette thèse.

Yolaine Bourda, Directrice du Département d'Informatique de Supélec, de m'avoir accueillie dans le département et de m'avoir permis de réaliser cette thèse dans des conditions excellentes.

Arnaud Cucurru, Alain Faivre, Christophe Gaston et Chrokri Mraidha, Chercheurs au CEA-LIST, pour nos nombreux échanges de points de vue sur l'hétérogénéité des modèles et pour nos collaborations fructueuses dans le cadre du projet OpenDevFactory.

Tous les membres du Département d'Informatique et du Centre de Ressources Informatiques de Supélec, pour leur accueil chaleureux. Merci en particulier à Luc pour sa bonne humeur constante et sa compagnie lors des footings pluvieux, à Evelyne et à Sylvain pour leur patience et leur disponibilité face à mes problèmes techniques et administratifs.

Tous mes collègues de bureau(x) ainsi que les doctorants du Département d'Informatique pour leur soutien amical et leurs encouragements tout au long de ces trois années.

Résumé

Dans le contexte de l'Ingénierie Dirigée par les Modèles, l'utilisation de multiples paradigmes de modélisation pour développer un système complexe est à la fois inévitable et essentielle. Les modèles qui représentent un tel système sont donc hétérogènes, ce qui rend tout raisonnement global sur le système difficile. L'objectif de la modélisation multi-paradigme est de faciliter l'utilisation conjointe de modèles hétérogènes pendant le cycle de développement. Les travaux exposés dans cette thèse concernent l'étude de l'hétérogénéité des modèles et la conception d'une approche pour la modélisation multi-paradigme des systèmes.

Nous caractérisons les causes de l'hétérogénéité des modèles par rapport au cycle de développement puis identifions différents types d'hétérogénéité. En nous basant sur ces causes d'hétérogénéité, nous proposons un cadre d'étude pour le domaine de la modélisation multi-paradigme avec différents axes de recherche.

La multidisciplinarité de la modélisation multi-paradigme rend applicables des techniques issues de différents domaines. Nous proposons un état de l'art et une classification des techniques dont nous avons étudié la pertinence par rapport à l'hétérogénéité. La gamme des techniques présentées inclut les transformations de modèles, la composition de méta-modèles, la composition de modèles, l'adaptation de composants, la co-simulation ou encore les méga-modèles.

Nous présentons ensuite ModHel'X, l'approche de composition de modèles pour la modélisation multi-paradigme que nous avons développée. Elle s'appuie sur le concept de modèle de calcul et permet :

1. de spécifier la sémantique d'un langage de modélisation de manière exécutable à travers la spécialisation opérationnelle d'une sémantique abstraite pour les modèles de calcul ;
2. de spécifier explicitement les mécanismes de composition à utiliser entre des modèles hétérogènes via une structure de modélisation appelée bloc d'interface ;
3. de simuler le comportement global de modèles hétérogènes par un algorithme générique d'exécution que nous avons défini.

Une implémentation de ModHel'X a été réalisée sous la forme d'un framework s'appuyant sur EMF (Eclipse Modeling Framework).

Mots clés : modélisation multi-paradigme, hétérogénéité des modèles, modélisation hétérogène, composition de modèles, langage de modélisation, modèle de calcul, exécution de modèles, Ingénierie Dirigée par les Modèles (IDM)

Abstract

In the context of Model Driven Engineering, the use of multiple modeling paradigms for developing complex systems is both unavoidable and essential. It results in the heterogeneity of the models representing the considered system and makes global reasoning about the system difficult. The objective of multi-paradigm modeling is to ease the joint use of heterogeneous models during the development cycle. In the work presented in this dissertation, we focus on the study of the heterogeneity of models and propose an approach to multi-paradigm modeling.

We first qualify the causes of the heterogeneity of models with respect to the development cycle and we identify several types of heterogeneity. On this basis, we propose a framework for the study of the domain of multi-paradigm modeling with several research axis.

The multidisciplinarity of multi-paradigm modeling allows the use of techniques from various fields. We propose a survey and a classification of the techniques which are relevant with respect to heterogeneity. The range of the techniques that we present includes model transformation, meta-model composition, model composition, component adaptation, co-simulation and mega-models.

Then we present ModHel'X, the approach to the composition of models for multi-paradigm modeling that we developed. It relies on the concept of model of computation and allows:

1. the specification of the semantics of a modeling language in an executable way by specializing an abstract semantics for models of computation that we developed;
2. the explicit specification of the composition mechanism between heterogeneous models through a special modeling structure called interface block;
3. the simulation of the global behavior of heterogeneous models thanks to a generic execution algorithm that we defined.

ModHel'X has been implemented in a framework based on EMF (Eclipse Modeling Framework).

Keywords: multi-paradigm modeling, heterogeneity of models, heterogeneous modeling, composition of models, modeling language, model of computation, execution of models, Model Driven Engineering (MDE)

Table des matières

1 Introduction

Les travaux présentés dans ce mémoire s'inscrivent dans le contexte de l'Ingénierie Dirigée par les Modèles. Ils ont pour objectif général l'étude de l'hétérogénéité des modèles et la conception d'une approche destinée à faciliter l'utilisation conjointe de modèles hétérogènes au cours du cycle de développement des systèmes.

1.1 Contexte

Le contexte de l'ingénierie des systèmes a beaucoup évolué récemment avec le développement de l'Ingénierie Dirigée par les Modèles (IDM). Faisant des *modèles* (informatiques) les supports prépondérants du processus de conception, l'IDM est vue comme une réponse possible aux nombreux enjeux de l'ingénierie système [BBJ07] : augmentation de la productivité, diminution des coûts et du temps de développement, augmentation de la fiabilité, etc.

D'abord utilisée principalement dans le domaine des systèmes logiciels, l'IDM est maintenant appliquée à la conception des *systèmes dits « complexes »*, dont l'une des particularités est d'être fortement hétérogènes. Cette évolution s'est accompagnée d'un développement notable des techniques de définition de langages de modélisation, entrainant ainsi la multiplication des *langages de modélisation dits « domaine-spécifiques »* à côté des langages de modélisation intégrés à des approches existantes. Capturant des connaissances et un savoir-faire adaptés à des métiers ou à des domaines, ces langages sont essentiels pour faciliter la conception et augmenter la qualité du système conçu. La conception de systèmes complexes dans le cadre de l'IDM implique donc inévitablement la mise en œuvre simultanée de plusieurs de ces langages de modélisation. L'hétérogénéité au niveau des systèmes entraîne ainsi *l'hétérogénéité au niveau des modèles.*

1.2 Contributions

Afin de mieux comprendre les causes de l'hétérogénéité des modèles, nous proposons tout d'abord d'analyser l'utilisation des techniques de modélisation en fonction des étapes classiques du cycle de développement. Pour cela, nous introduisons dans un premier temps quelques-unes des différentes approches de la modélisation des systèmes existant dans le cadre de l'IDM, dont notamment l'approche MDA (Model Driven Architecture) de l'OMG. Nous mettons alors en évidence les causes de l'hétérogénéité des modèles et nous présentons les différents types d'hétérogénéité que nous avons identifiés.

L'hétérogénéité des modèles rend le processus de développement complexe et soulève de nombreuses problématiques. La nécessité de traiter ces problématiques est aujourd'hui admise (voir par exemple [PY96] et [EJL+03]). Les recherches sur ce sujet ont d'ailleurs donné naissance récemment à un domaine de recherche, baptisé « Modélisation Multi-Paradigme ». Multi-

disciplinaire par essence, la modélisation multi-paradigme implique naturellement diverses communautés, spécialisées dans des disciplines variées. Les techniques développées pour adresser les problématiques liées à la modélisation multi-paradigme proviennent donc d'horizons très différents. En conséquence, la terminologie et l'état de l'art propre à la modélisation multi-paradigme ne font pas consensus, ce qui complique les interactions entre chercheurs et rend les travaux de recherche de ce domaine difficiles à comparer. C'est pourquoi nous proposons dans ce mémoire d'étendre et de généraliser la définition du domaine de la modélisation multi-paradigme telle qu'elle est présentée dans différents travaux existants et de décliner ses axes de recherche selon les différentes causes d'hétérogénéité des modèles que nous avons identifiées. Puis, nous proposons de passer en revue différentes techniques issues de différentes disciplines de recherche et dont l'objectif est d'adresser les problématiques liées à l'hétérogénéité des modèles. Tout au long de cette étude, nous précisons le sens de plusieurs termes employés dans le domaine de la modélisation multi-paradigme. Nous examinons par ailleurs la pertinence des techniques présentées par rapport au traitement de l'hétérogénéité des modèles. Pour conclure cette étude, nous proposons un ensemble de critères que nous avons élaborés pour comparer ces techniques.

Parmi les différents types de techniques que nous avons étudiés, les techniques de composition de modèles trouvent un intérêt particulier à nos yeux. Nous présentons dans ce mémoire les résultats de travaux réalisés sur une approche de composition de modèles dédiée à la simulation. Les problématiques de ce domaine sont liées bien évidemment à l'hétérogénéité des modèles à composer mais également aux questions liées à l'exécutabilité des modèles. Nous abordons ce sujet selon deux axes particuliers : (1) les moyens permettant de décrire la sémantique des langages de modélisation de manière exécutable et (2) les techniques permettant de composer les modèles tout en les préservant. L'approche que nous avons développée, dont nous présentons les principes ainsi qu'une implémentation, s'appuie sur le concept de modèle de calcul, initialement introduit par le professeur E. A. Lee. Elle permet de :

1. spécifier la sémantique d'un langage de modélisation de manière exécutable à travers la spécialisation opérationnelle d'une sémantique abstraite pour les modèles de calcul ;
2. spécifier explicitement les mécanismes de composition à utiliser entre des modèles hétérogènes via une structure de modélisation appelée bloc d'interface ;
3. simuler le comportement global de modèles hétérogènes par un algorithme générique d'exécution que nous avons défini.

Nous présentons les résultats de nos expérimentations sur cette approche et positionnons nos travaux par rapport aux techniques présentées dans le cadre de notre état de l'art sur la modélisation multi-paradigme.

1.3 Structure du mémoire

Ce mémoire est organisé en deux parties principales : une première partie intitulée « Conception des systèmes et hétérogénéité », dans laquelle nous présentons le contexte de nos travaux, introduisons les problématiques du domaine et proposons un ensemble de définitions ainsi qu'un état de l'art des travaux existants ; et une deuxième partie intitulée « Composition exécutable de modèles hétérogènes avec ModHel'X » dans laquelle nous proposons une approche pour la composition de modèles hétérogènes dédiée à la simulation. Nous détaillons ci-dessous le contenu de ces deux parties.

La **partie I, « Conception des systèmes et hétérogénéité »**, est composée de deux chapitres :

– Dans le **chapitre 2 intitulé « Ingeniérie système, modélisation et hétérogénéité »**, nous fixons la terminologie utilisée dans le mémoire et présentons le domaine de l'Ingénierie

Dirigée par les Modèles. Nous introduisons ensuite différentes approches de l'IDM, dont l'approche Model Driven Architecture de l'OMG. Nous terminons ce chapitre en mettant en évidence les différentes causes de l'hétérogénéité des modèles par rapport au cycle de développement des systèmes et en présentant les différents types d'hétérogénéité que nous avons identifiés.

– Dans le **chapitre 3 intitulé « Modélisation multi-paradigme »**, nous présentons le domaine de la modélisation multi-paradigme. Nous introduisons différents axes de recherche de ce domaine liés aux causes de l'hétérogénéité identifiées dans le chapitre précédent. Nous passons alors en revue un ensemble de techniques applicables à la modélisation multi-paradigme et issues de différents domaines de recherche. Nous concluons ce chapitre en présentant un ensemble de critères dédiés à la comparaison des approches de modélisation multi-paradigme.

Les contributions et résultats présentés dans cette partie ont fait l'objet d'un article soumis en 2008 au journal SIMULATION pour une édition spéciale sur la Modélisation Multi-Paradigme.

La **partie II, « Composition exécutable de modèles hétérogènes avec ModHel'X »**, est composée de deux chapitres :

– Dans le **chapitre 4 intitulé « ModHel'X : une approche de la composition de modèles hétérogènes »**, nous décrivons l'approche de composition de modèles hétérogènes que nous avons développée. Après avoir introduit les concepts fondamentaux sur lesquels s'appuie notre approche, nous détaillons les différents éléments qui la composent, à savoir un méta-modèle générique ainsi qu'un algorithme d'exécution générique. Nous exposons ensuite la méthode associée permettant de décrire un modèle de calcul dans notre approche. Une section de positionnement et de discussion de différents aspects de notre approche conclut ce chapitre.
– Dans le **chapitre 5 intitulé « Application et validation : le framework ModHel'X »**, nous présentons l'implémentation que nous avons réalisée de notre approche sous la forme d'un framework. Ce chapitre comprend tout d'abord les différents éléments de conception du framework. Nous présentons ensuite les différentes difficultés que nous avons rencontrées pendant l'implémentation du framework. Puis, nous illustrons la mise en œuvre de notre approche sur un cas d'étude dont nous détaillons les différents éléments et leur implémentation dans notre framework.

L'approche que nous présentons dans cette partie, ainsi que l'implémentation qui en a été réalisée, ont fait l'objet de différents articles, acceptés à des workshops internationaux [HBMVN07, HB08] et à une conférence internationale IEEE [BH08].

1.4 Le projet OpenDevFactory

Une partie du travail présenté dans ce mémoire a été préparée dans le cadre du projet OpenDevFactory, sous-projet du projet Usine Logicielle [Sys] du Pôle de Compétitivité System@tic Paris Région. L'objectif du projet OpenDevFactory est de créer une plate-forme d'intégration d'outils de génie logiciel permettant de supporter l'ingénierie des systèmes complexes dans une approche IDM (Ingénierie Dirigée par les Modèles).

Nous avons participé, en collaboration avec le Laboratoire d'Intégration des Systèmes et des Technologies du CEA (CEA-LIST) et Thales Research Technology (TRT), à la tâche PC-xUML (Prospective Component – Executable UML) dont l'objectif est l'obtention de modèles UML pour le temps réel qui soient exécutables. Ces travaux ont abouti à une version préliminaire du framework ModHel'X, que nous présentons dans la partie II de ce mémoire.

Première partie

Conception des systèmes et hétérogénéité

2

Ingénierie système, modélisation et hétérogénéité

2.1 Introduction

Multiplication des fonctionnalités, diminution des coûts de développement, augmentation de la fiabilité ou encore amélioration des performances sont autant d'enjeux de la conception des systèmes. Ces enjeux multiples diffèrent en fonction du type de système considéré mais ont une influence sur tout le processus de conception. En conséquence, celui-ci a fait l'objet d'évolutions fondamentales depuis plusieurs années. Après une présentation de la terminologie que nous utiliserons tout au long de ce mémoire, nous abordons quelques-unes des évolutions majeures concernant l'activité de conception dans le domaine de *l'Ingénierie Système*. Nous discutons en particulier les évolutions autour des modèles utilisés au cours du processus de conception des systèmes et présentons différentes approches d'ingénierie système centrées sur les modèles. Nous nous attachons notamment à montrer tout au long de ce chapitre que l'hétérogénéité est omniprésente à tous les niveaux de la conception des systèmes telle qu'elle se pratique aujourd'hui. Nous formalisons l'impact de cette hétérogénéité sur l'ingénierie système au travers de la notion de *modèle hétérogène* et présentons les problématiques qui en découlent.

2.2 Terminologie

2.2.1 Système

La notion de *système* a été définie de manière théorique par plusieurs travaux fondamentaux dans les domaines de la Théorie Systémique [vB82, Dur71], de la Cybernétique [Wie48] ou encore du Structuralisme [dS13]. Nous n'avons pas pour objectif ici de traiter de telles définitions mais simplement de rendre explicite cette notion que nous utiliserons tout au long de ce mémoire. Suivant la définition de l'AFIS [dSA04], nous appelons *Système* un ensemble organisé d'éléments fonctionnant de manière unitaire et en interaction permanente. Cet ensemble forme un tout cohérent, intégré pour assurer une ou plusieurs fonctions correspondant à la finalité du système par rapport à son environnement.

Un système est caractérisé par les propriétés qui résultent de l'*interaction* de ses éléments. Ces propriétés peuvent être qualifiées d'émergentes [Hol98] car ce sont des propriétés nouvelles obtenues au niveau du système du fait des synergies existant entre ses constituants. De telles propriétés émergentes peuvent être souhaitées ou indésirables. L'objectif de l'activité de conception en ingénierie système est d'obtenir les comportements émergents recherchés en maintenant les comportements émergents non intentionnels dans des limites considérées comme acceptables. Dans ce contexte, l'étude du système d'un point de vue comportemental est donc fondamentale. C'est sur les questions liées à cet aspect de la conception des systèmes que portera principalement ce mémoire.

Différents types de systèmes peuvent être distingués. Dans leur classification, D. Harel et A. Pnueli [HP85] distinguent notamment les systèmes transformationnels, les systèmes interactifs et les systèmes réactifs. Les systèmes dits transformationnels sont des systèmes qui acquièrent des données, les traitent, produisent des sorties puis terminent. Les systèmes interactifs sont des systèmes qui interagissent avec leur environnement, à une vitesse qui leur est propre. Quant aux systèmes dits réactifs, ce sont des systèmes qui interagissent en permanence avec leur environnement, mais à une vitesse imposée par l'environnement. Ce couplage avec l'environnement, qui est par essence non totalement maîtrisable et non totalement prévisible, rend les systèmes réactifs particulièrement difficiles à concevoir. Les systèmes dits embarqués (voir la section suivante) sont des systèmes réactifs.

La difficulté à concevoir un système est liée notamment à sa complexité. Celle-ci tient à au moins trois facteurs :

Le nombre et la nature de ses éléments. Le nombre d'éléments peut être élevé, il peut même être éventuellement variable au cours du temps, et la nature de ces éléments peut

être très variée. Les systèmes complexes sont généralement fondamentalement *hétérogènes*.

La nature de son organisation interne, qui est liée aux relations existant entre ses éléments. Ceux-ci peuvent former des réseaux, des hiérarchies, etc. L'organisation du système peut également varier au cours du temps.

Le couplage avec l'environnement. Plus le système est en interaction forte avec l'environnement, plus il est exposé à l'incertitude et à l'imprévisibilité de celui-ci.

Dans ce mémoire nous nous intéressons en particulier à la complexité causée par l'hétérogénéité des éléments d'un système. Nous détaillerons dans les parties suivantes les conséquences de l'hétérogénéité sur le processus de conception.

2.2.2 Systèmes embarqués

Les systèmes embarqués sont très souvent considérés à travers des sous-types particuliers tels que les systèmes temps réel, les systèmes critiques, etc. Différentes taxonomies des systèmes embarqués existent [Zur05, Tim07], mais il est difficile d'en trouver une définition générale. Nous appelons *système embarqué*, ou système enfoui, un système intégré dans un système plus large avec lequel il est interfacé et pour lequel il réalise une ou plusieurs fonctions particulières (par exemple : contrôle, surveillance, communication...). Un système embarqué comprend des logiciels et des matériels conjointement et spécifiquement conçus pour assurer ces fonctionnalités. De ce fait, les systèmes embarqués sont par essence *hétérogènes* [EJL+03].

Les systèmes embarqués sont également par nature des systèmes contraints [HS07]. Tout d'abord, ce sont des systèmes réactifs qui subissent une forte intégration avec leur environnement. De plus, du fait de leur insertion dans un système « embarquant », l'espace matériel qu'ils peuvent occuper est généralement limité (tendance à la miniaturisation) ainsi que, très souvent, les ressources énergétiques qu'ils peuvent utiliser. Ils doivent par ailleurs répondre à des spécifications exigeantes [Zur07], que ce soit en termes de performances, de robustesse, de fiabilité, de délai de mise sur le marché, d'évolutivité ou encore de coût. Du fait de ces contraintes et exigences, la conception des systèmes embarqués pose des difficultés particulières que nous étudierons dans la section 2.5.5. Nous proposons de distinguer ci-dessous les différents types de contraintes auxquelles ils peuvent être exposés et qui donnent lieu à différentes catégories de systèmes embarqués. Il est important de noter que cette classification n'est pas stricte car un système embarqué donné est souvent soumis à plusieurs contraintes et exigences de types différents en fonction de l'application à laquelle il est destiné. Nous identifions, entre autres :

Les systèmes temps réel D'après [Sta88], dans un système temps réel, la correction dépend non seulement du résultat logique du calcul mais également du temps nécessaire pour que le résultat soit produit. La contrainte de temps est donc un élément déterminant du fonctionnement des systèmes temps réel. Un système temps réel peut être décrit comme un système destiné à observer et à agir sur son environnement extérieur avec un délai de réponse fini et spécifié [Liu00]. Tout délai de réponse supérieur au délai spécifié implique un déphasage du système et des erreurs de traitement aux conséquences plus ou moins graves. Par exemple, lors de l'atterrissage d'un avion, le système d'indication d'altitude doit fournir une valeur exacte à un instant précis de manière à ce que les réactions du pilote, qui dépendent de l'altitude, aient lieu au bon moment. Tout retard de réponse du système peut conduire dans ce cas à l'écrasement de l'avion. L'interaction d'un système temps réel avec son environnement fait intervenir des capteurs et des actionneurs. La réaction du système comporte trois étapes, et doit se faire en un temps déterminé : réception des données à travers les capteurs, réaction (traitement des données, calcul), réponse à travers les actionneurs.

Les systèmes critiques Ce type de système est soumis à des contraintes de fiabilité extrêmement fortes [Sto96]. Toute erreur dans le fonctionnement d'un tel système a des consé-

quences catastrophiques telles que des pertes de vies humaines, des destructions matérielles importantes, un impact financier colossal, etc. Les systèmes critiques peuvent également être des systèmes temps réels, pour lesquels les contraintes de temps de réponse sont dite « strictes » : leur non respect cause une erreur de fonctionnement pouvant entraîner d'importants dégâts. Le contrôle de processus industriels sensibles, comme par exemple des centrales nucléaires, le contrôle de systèmes aéronautiques, la bourse ou encore la médecine assistée par ordinateur font partie des systèmes critiques.

Les systèmes distribués et mobiles Dans le cadre des systèmes embarqués, nous désignons par le terme de systèmes distribués les systèmes embarqués répartis spatialement fonctionnant généralement en association les uns avec les autres via un réseau. Les systèmes de type « réseaux de capteurs », par exemple, sont des systèmes embarqués distribués composés d'unités autonomes dotées de modules de communication (souvent sans fil). Leurs applications sont très variées, depuis les systèmes de surveillance environnementale jusqu'aux systèmes d'armement militaire. Les systèmes embarqués mobiles, quant à eux, sont caractérisés par une position spatiale qui change au cours du temps. Les puces de nos téléphones portables sont de parfaits exemples de systèmes embarqués mobiles. Qu'ils soient distribués ou mobiles, ces systèmes ont souvent pour contrainte majeure l'autonomie. Fonctionnant généralement sans intervention humaine directe, ils sont soumis à de fortes restrictions sur leurs ressources énergétiques (fonctionnement sur batterie en particulier).

Les systèmes reconfigurables Appelés également systèmes adaptatifs, les systèmes reconfigurables [Tei07] sont des systèmes capables de s'adapter de manière dynamique aux changements de leur environnement en modifiant leur configuration : modification de la disposition relative des éléments, du nombre d'éléments, de la nature des éléments, du comportement des éléments, etc. De telles notions d'adaptabilité et de flexibilité sont plutôt récentes, concernent principalement le domaine des logiciels embarqués et posent encore des problèmes techniques importants.

Les systèmes embarqués sont d'une importance stratégique pour l'économie car ils sont facteurs d'innovation, de différentiation et d'amélioration de la compétitivité : ils permettent d'offrir de nouvelles fonctionnalités et de nouveaux services, dans des produits existants ou dans de nouveaux produits. La source de la valeur ajoutée est principalement aujourd'hui dans les logiciels embarqués. Les composants embarqués sont intégrés massivement dans les systèmes communicants qui sont à la base de notre société de l'information : téléphones cellulaires, automobile, appareils médicaux, photo/vidéo/hifi, électroménager, avionique, spatial, jouets, etc.

2.2.3 Ingénierie Système

L'*Ingénierie Système* [INC, AFI] peut être vue comme un processus qui intègre l'ensemble des activités centrées autour du cycle de vie d'un système, depuis sa définition jusqu'à son retrait de service, en passant par sa conception, sa validation ou sa maintenance. L'objectif de ce processus est l'obtention d'un système apportant une solution à un besoin opérationnel, satisfaisant un ensemble d'attentes et de contraintes fixées par les parties prenantes et répondant à des critères de qualité mais également de coût sur l'ensemble de son cycle de vie. Il s'agit d'un processus coopératif, itératif et *interdisciplinaire* car il consiste à intégrer les efforts de toutes les disciplines impliquées dans le cycle de vie du système pour progresser vers la solution en prenant des décisions successives.

Il existe plusieurs méthodologies proposant des approches différentes de l'Ingénierie Système, notamment à travers des définitions différentes du *cycle de vie* du système. Nous avons choisi de nous appuyer sur le cycle de vie en V dans la suite de ce mémoire, même si celui-ci est discuté par certains auteurs. En effet, il nous semble que les concepts sous-jacents au cycle en V sont intégrés à la plupart des autres modèles de cycle. Défini initialement par Paul Rook dans les années 80,

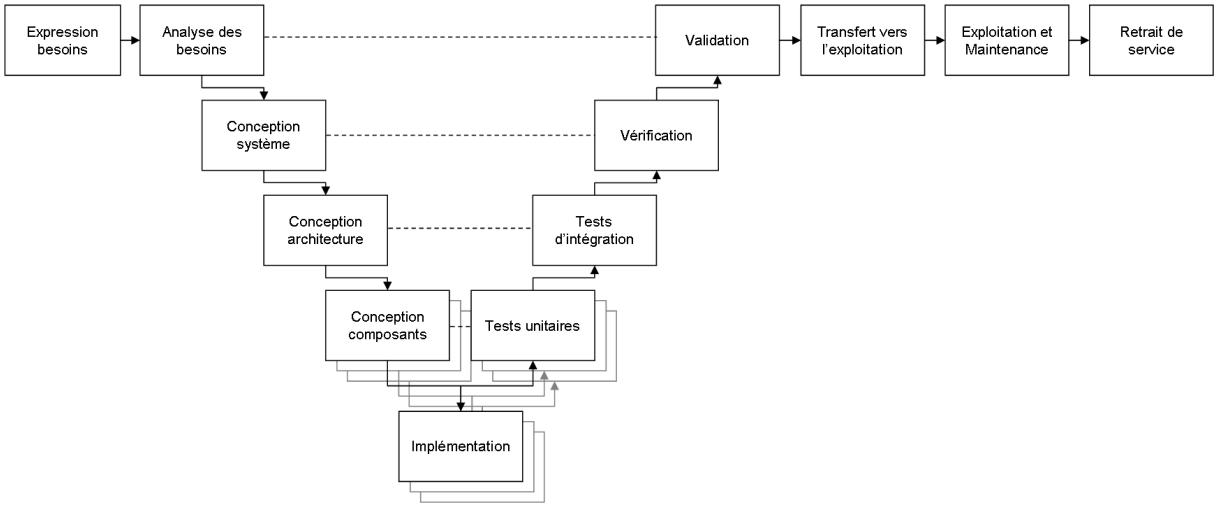


Fig. 2.1 – Cycle de vie en V

le modèle de cycle de vie en V est conçu à l'origine pour les systèmes logiciels [BD99], puis il a été adapté pour l'ensemble des systèmes en général [FMC00]. Une des versions de ce modèle est représentée sur la figure 2.1. Le modèle du cycle en V est aujourd'hui assez largement adopté dans le monde industriel, notamment au travers des processus (en particulier les processus techniques) définis par la norme ISO 15288/AFNOR Z 67-288. Dans la suite de ce mémoire, nous nous intéressons en particulier aux phases concernant la conception du système, c'est à dire l'ensemble des phases centrales du cycle, depuis l'analyse des besoins jusqu'à la validation.

Dans le cadre d'un objectif général d'amélioration de la compétitivité des entreprises, l'Ingénierie Système doit faire face à de multiples enjeux, dont notamment :

- la maîtrise de la complexité des systèmes ;
- l'amélioration de l'adéquation aux besoins et de la qualité ;
- l'anticipation des problèmes et des risques tout au long du cycle de vie ;
- le raccourcissement des temps de développement ;
- la maîtrise des coûts (avec une anticipation très en amont du coût global du cycle de vie notamment) ;
- la maîtrise de la transdisciplinarité et de la coopération de multiples acteurs.

L'Ingénierie Système est donc une discipline en évolution, faisant l'objet de nombreux travaux de recherche.

2.2.4 Modèle, modélisation

L'utilisation de modèles dans l'activité de conception des systèmes est une réalité depuis l'antiquité. Les ouvrages de Vitruve [Vit11] attestent notamment que leur usage était un standard dès la Grèce Antique. Dans le contexte de l'ingénierie système, le rôle premier des modèles est de permettre d'appréhender des systèmes dont la complexité est importante et d'en étudier des aspects particuliers avant même la mise en œuvre concrète du système [LMS03]. Les modèles permettent alors par exemple d'identifier des incertitudes quant à la spécification du système ou quant à l'adéquation d'une solution pour sa réalisation lorsque celles-ci sont trop complexes pour permettre un raisonnement immédiat. Par ailleurs, les modèles constituent également des supports pour communiquer des idées au sujet de la conception du système entre les différentes parties impliquées dans le projet. Enfin, les modèles servent également de guides lors des phases d'implémentation du système. Dans ce contexte, les modèles utilisés permettent donc d'anticiper sur le comportement qu'aura le système final. Ils sont appelés *modèles prédictifs* par opposition aux modèles dits *explicatifs* dont le rôle est de donner une représentation la plus fidèle pos-

sible du monde réel. Les modèles prédictifs peuvent être des modèles physiques, structurels ou comportementaux. Les modèles physiques donnent des informations quant aux caractéristiques physiques du futur système (largeur, hauteur, matériaux, etc.). Les modèles structurels sont utilisés pour définir l'organisation des différents éléments du futur système. Enfin, les modèles comportementaux ont pour objectif de spécifier le comportement dynamique du futur système.

Dans la suite de ce mémoire, nous appelons *modèle* une représentation prédictive du système qui met en valeur des propriétés que l'on considère intéressantes par rapport à un objectif de conception donné. Ainsi, lors de la conception, le système est vu, défini et étudié à travers différents modèles. L'activité de *modélisation* correspond à la construction d'un modèle en tant que représentation d'un aspect du système pour un objectif de conception donné (étude d'une partie du système, analyse de propriétés particulières, simulation du comportement, etc.). Nous nous intéressons tout particulièrement dans ce mémoire aux *modèles comportementaux*.

2.3 Ingénierie dirigée par les modèles (IDM) et cycle de conception des systèmes

2.3.1 Evolutions autour des modèles

Depuis Vitruve, l'ingénierie système a évolué autour de la notion de modèle. Des abstractions de puissance croissante ont été nécessaires pour représenter des systèmes d'une complexité croissante. De plus, avec l'avènement des outils informatiques, les modèles ont évolué depuis les formats papier ou les prototypes de taille réduite vers des modèles informatiques au format électronique. Des outils permettant de construire et d'exploiter ces modèles ont été développés. Les modèles informatiques ont progressivement pris un rôle prépondérant dans le cycle de développement des systèmes, à tel point que l'on peut dire aujourd'hui que l'ingénierie système est en fait « dirigée par les modèles » car ces modèles sont omniprésents depuis les phases d'expression des besoins jusqu'aux phases d'exploitation ou de maintenance. Et si le fait de travailler à partir de modèles n'est pas nouveau en ingénierie système, toutes les disciplines qui se sont développées autour des modèles informatiques dans l'objectif d'outiller et d'automatiser les différentes activités du cycle de développement des systèmes sont, elles, relativement récentes. Le terme d'*Ingénierie Dirigée par les Modèles (IDM)* [Ken02, FEBF06] recouvre l'ensemble de ces disciplines, dans lesquelles les modèles jouent un rôle de tout premier plan.

2.3.2 Modèles pour l'ingénierie système

Les rôles des modèles, dans le cadre de l'IDM, sont multiples car ils sont utilisés de bout en bout du cycle de développement. Ils peuvent permettre de capturer et décrire avec précision des besoins, de capturer des choix de conception de manière séparée des besoins, d'explorer plusieurs solutions en fonction de critères (coût, consommation d'énergie) ou encore de stocker des informations [RJB05]. Les modèles utilisés en IDM sont des modèles informatiques.

2.3.2.1 Langages et formalismes de modélisation

Chaque modèle est construit en utilisant un *formalisme de modélisation*, c'est à dire un langage. Un langage de modélisation (cette définition est générale et concerne l'ensemble des langages informatiques, notamment les langages de programmation [1]) est défini par une syntaxe

1. La distinction entre langage de modélisation et langage de programmation est, de notre point de vue, une question de niveau d'abstraction. Nous considérons que tous deux sont de même nature (langage informatique avec une syntaxe abstraite, une sémantique et une syntaxe concrète), mais qu'un langage de modélisation a un niveau d'abstraction plus élevé qu'un langage de programmation. Avec l'évolution des langages de programmation, cette distinction devient cependant de plus en plus ténue.

abstraite, une sémantique et une syntaxe concrète. La syntaxe abstraite définit les concepts de base du langage. La sémantique définit comment les concepts du langage doivent être interprétés – par l'homme mais surtout par les machines. La sémantique peut être exprimée sous différentes formes (par exemple du texte en langage naturel, des définitions mathématiques, des spécifications dans un langage informatique, etc.) et être de différents types (dénotationnelle, impérative, etc.). Remarquons ici que le terme sémantique n'est donc pas employé dans le sens « usuel » des mathématiques strictes car nous entendons par sémantique d'un langage l'ensemble des correspondances entre les éléments de la syntaxe abstraite du langage et des structures permettant de les interpréter ou de les calculer (donc pas uniquement des structures mathématiques). La sémantique d'un modèle au sens de l'IDM est obtenue sur la base de la sémantique du formalisme de modélisation utilisé : le concepteur exprime, à l'aide du langage, un ensemble de propriétés du système. Enfin, la syntaxe concrète définit le type de notation qui sera utilisé et donne à chaque concept abstrait une représentation concrète dans cette notation, qui peut être graphique, textuelle ou mixte. Dans la suite de ce mémoire nous utiliserons indifféremment les termes formalisme et langage de modélisation.

2.3.2.2 Qualités des modèles

Les qualités attendues d'un modèle dans le cadre de l'IDM sont nombreuses. Dans [LMS03], B. Selic liste cinq caractéristiques considérées comme essentielles. Selon lui, un modèle doit être :

- Abstrait : le modèle doit permettre d'omettre ou de cacher les détails que l'on ne souhaite pas considérer lors de l'étude d'une question, que ce soit pour des raisons de complexité ou de pertinence.
- Compréhensible : le modèle doit être compris par les personnes qui l'utilisent. Ce constat a des implications sur le formalisme à utiliser : d'une part, celui-ci doit être en adéquation avec l'objectif du modèle (notamment en termes de niveau d'abstraction, mais également en termes de type de représentation : graphique, textuelle, etc.) et, d'autre part, la sémantique du formalisme doit être communément admise.
- Fidèle et précis : le modèle doit représenter fidèlement les propriétés et caractéristiques bien définies du système lorsqu'il est vu dans une optique spécifique.
- Prédictif : le modèle doit fournir les informations nécessaires et suffisantes pour permettre de faire des prédictions justes au sujet de propriétés du système.
- Economique : les coûts de construction et d'utilisation du modèle doivent rester très bas, notamment par rapport aux coûts de construction ou d'utilisation d'un prototype du système par exemple.

Il faut également ajouter à cette liste d'autres caractéristiques liées à la qualité comme la maintenabilité, la traçabilité des modifications ou encore l'évolutivité.

La conciliation de toutes ces caractéristiques est un enjeu de taille pour l'IDM et est à l'origine des nombreuses problématiques du domaine.

2.3.3 Modèles pour le génie logiciel

Les modèles utilisés pour le *Génie Logiciel*, c'est-à-dire l'ingénierie des systèmes logiciels, ne sont pas fondamentalement différents des modèles présentés ci-dessus, et doivent répondre aux même contraintes. Cependant, dans le cadre de l'IDM, il est important de noter que le fossé entre un modèle informatique et son implémentation logicielle (le programme) est moins important qu'entre un modèle informatique et son implémentation matérielle. Dans le cas des systèmes matériels, les causes de ce fossé sont nombreuses : comportements propres aux matériaux, méthodes d'implémentation, effets d'échelle, différences de culture entre les parties impliquées, etc. Même si des effets similaires existent dans le monde logiciel, leur ampleur est généralement moindre et les modèles permettent alors de conduire plus directement à une implémentation.

C'est notamment pour cette raison que le génie logiciel a pris de l'avance sur l'ingénierie système dans le cadre de l'IDM, notamment au travers de l'approche Model Driven Architecture (MDA) de l'OMG (voir la section 2.4.1). Aujourd'hui, du fait de l'augmentation progressive du niveau d'abstraction des langages de programmation, la différence entre langage de modélisation et langage de programmation se fait de plus en plus ténue. Cependant, de nombreuses problématiques persistent, liées aux différentes activités de manipulation de modèles apparues dans le cadre de l'IDM.

2.3.4 Activités liées à l'IDM

Les activités liées à la manipulation des modèles dans le cadre de l'IDM sont nombreuses et apportent chacune un ensemble de problématiques spécifiques. Nous présentons ici une liste non exhaustive de ces activités et donnons un aperçu des problématiques qui leur sont liées.

Réalisation de modèles La réalisation des modèles nécessite non seulement l'expertise technique pour comprendre ou concevoir la partie du système concerné mais également une bonne connaissance du langage de modélisation utilisé. Dans le cas de systèmes complexes, les modèles deviennent également complexes et surtout de taille importante. La qualité de l'outillage devient alors essentielle car ce sont les outils qui permettent de mieux visualiser le modèle, de s'affranchir de certains détails ou encore de vérifier automatiquement la syntaxe. Les problématiques liées à l'outillage sont variées : elles concernent la visualisation des modèles, les méthodes d'assistance, le support des langages de modélisation, etc.

Stockage de modèles L'informatisation des modèles pose le problème de la gestion de leur persistance puis de leur accès par les utilisateurs. Les problématiques liées à cette activité concernent, par exemple, les formats de stockage, l'organisation du stockage ainsi que la gestion des méta-données concernant les modèles.

Echange de modèle L'échange de modèles entre différents acteurs d'un projet est une vraie nécessité car elle conditionne la bonne communication entre ces acteurs. Des problèmes de compréhension de modèles entre différents acteurs peuvent avoir des conséquences catastrophiques sur le système implémenté. L'échange de modèles pose notamment des problèmes de format (sérialisation, transport, etc.), de traduction et d'interprétation de la sémantique pour l'intéropérabilité (notamment entre les outils de modélisation).

Interrogation de modèles L'interrogation de modèles est l'activité qui permet de rechercher et récupérer de l'information dans les modèles. Les problématiques de cette activité sont liées par exemple à l'identification d'éléments ou de motifs dans les modèles.

Exécution de modèles L'exécution de modèles comprend un éventail de tâches différentes allant de la simulation à l'exécution en temps réel en passant par l'exécution symbolique ou la génération de code. Dans ce domaine, les problèmes majeurs concernent l'exécutabilité de la sémantique des langages de modélisation utilisés. En effet, cette propriété d'exécutabilité conditionne la possibilité de pouvoir calculer, à partir du modèle, un comportement du système (simulation), ou même tous les comportements possibles de ce système (model-checking).

Vérification de modèles La vérification d'un modèle consiste à vérifier les propriétés propres de ce modèle par rapport à ce que l'on attend de lui (correction syntaxique, etc.). L'activité de vérification se distingue notamment de l'activité de validation. En effet, la validation consiste à vérifier, en utilisant un ou plusieurs modèles, les propriétés du système conçu par rapport à ce que l'on attend de ce système. Le terme de vérification de modèle recouvre différents aspects allant de la vérification de la syntaxe à la vérification de la sémantique. Les problématiques les plus complexes concernent bien sûr la vérification de la sémantique. Différentes techniques de vérification existent, avec leurs problématiques particulières : la

preuve, le test ou encore le model-checking. Les techniques de preuve s'appuient sur l'utilisation de représentations formelles (à base de logique, d'automates, de contraintes par exemple) du système. Dans ce contexte, on cherche à prouver des propriétés comme la consistance ou la complétude d'un modèle. Dans le cas de systèmes complexes, ces tâches deviennent impossibles à réaliser sur un modèle préexistant et un axe important de recherche vise à obtenir ces propriétés par construction. Les techniques de model-checking visent à analyser le comportement spécifié par le modèle de manière à vérifier des propriétés comme la sûreté, l'atteignabilité ou la vivacité. Les problématiques dans ce contexte sont liées notamment à l'identification avec exhaustivité des états possibles du système (explosion des espaces d'état). Enfin, le test est utilisé en complément du model-checking, notamment dans le cas de systèmes pour lesquels le model-checking est particulièrement inefficace (lorsque les systèmes sont trop complexes par exemple). L'évaluation de la pertinence des tests est une des difficultés principales dans ce domaine, et elle conditionne la façon dont les tests sont sélectionnés. Les enjeux liés à la vérification des modèles comptent aujourd'hui parmi les enjeux les plus importants de l'IDM.

Validation La validation permet de vérifier que le système implémenté répond aux besoins initiaux qui ont amené à sa conception. Certaines techniques comme le test peuvent être utilisées à la fois pour la vérification et pour la validation. Dans ce cadre, les modèles permettent notamment de générer des scénarios et des vecteurs de test de façon automatique [BGM91]. Par ailleurs, afin de minimiser les risques d'erreur de conception le plus en amont possible, les modèles peuvent être validés les uns par rapport aux autres au cours du cycle de développement. Il s'agit alors, par exemple, de vérifier que certaines propriétés sont préservées d'un modèle à un autre. Un ensemble des problématiques de ce domaine est lié au mécanisme de raffinement de modèle, par lequel on obtient un modèle plus détaillé à partir d'un autre (nécessitant souvent un apport d'information).

Gestion de l'évolution des modèles Les modèles évoluent au cours du cycle de développement du système. Ils peuvent être modifiés dans le cadre de correction d'erreurs ou d'ajout de fonctionnalités par exemple. Les problématiques dans ce contexte concernent en particulier la répercussion automatique des modifications sur les différents modèles impliqués et sur la documentation, la traçabilité des modifications ainsi que la gestion des versions.

2.4 Approches de l'ingénierie dirigée par les modèles

Si l'ingénierie dirigée par les modèles peut être considérée comme un domaine qui a émergé avec les technologies liées à l'informatisation des modèles, il existe différentes approches concrétisant différentes façons d'utiliser les modèles dans le cadre de l'ingénierie système. L'approche la plus connue et peut-être la plus développée est l'approche appelée Model Driven Architecture (MDA). Nous présentons cette approche dans la section suivante, avant d'évoquer brièvement d'autres approches existantes dans la section 2.4.2.

2.4.1 Model Driven Architecture – MDA

L'Architecture Dirigée par les Modèles (Model Driven Architecture – MDA) [OMGg] est une approche proposée et soutenue par l'Object Management Group (OMG). Cette approche peut être considérée comme une variante de l'IDM pour le génie logiciel. Son objectif est de faire évoluer les pratiques de conception du logiciel vers une approche centrée sur le modèle et non plus sur le code (voir la figure 2.2).

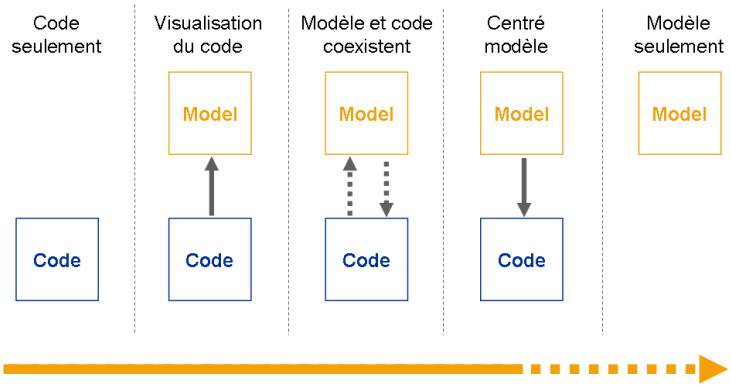


FIG. 2.2 – Spectre des pratiques de conception en génie logiciel

2.4.1.1 Standards

L'approche MDA repose sur un ensemble de standards de l'OMG (voir la figure 2.3), dont notamment UML [OMGl], le MOF [OMGf] et le CWM [OMGb].

Le Unified Modeling Language (UML), qui a largement inspiré l'approche MDA, est un langage de modélisation à vocation généraliste. De par son historique, c'est un langage orienté objet. Il comprend un ensemble de notations graphiques permettant de représenter un système sous différents points de vue.

Le MetaObject Facility (MOF) définit un langage abstrait et extensible permettant de décrire, définir et manipuler des méta-modèles (voir la section 2.4.1.5). Il a la particularité de s'autodéfinir. C'est aujourd'hui la pierre angulaire de l'approche MDA.

Le Common Warehouse Metamodel (CWM) définit un framework permettant de décrire des méta-données concernant des sources de données, des transformations de données ainsi que des processus de gestion d'entrepôts de données. Il a pour objectif de faciliter les échanges de méta-données dans le cadre d'environnements distribués et hétérogènes.

Le standard XML Metadata Interchange (XMI) [OMGm] vient compléter ces standards en définissant un format d'échange de méta-données basé sur XML. C'est sur XMI que reposent les techniques de sérialisation de modèles.

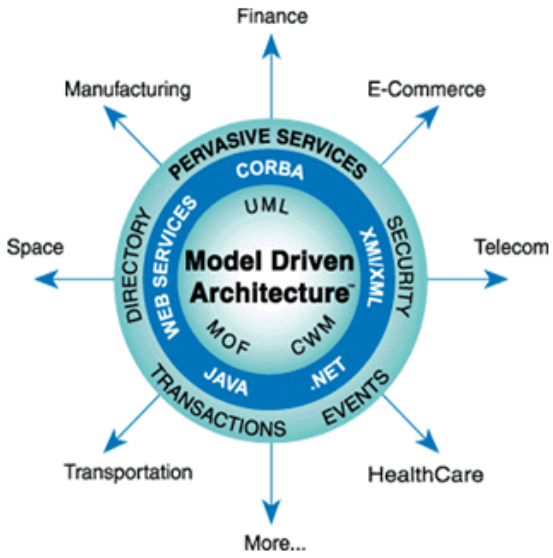


FIG. 2.3 – Les standards de l'Architecture Dirigée par les Modèles (MDA) (source : OMG)

Cette sphère de standards n'est pas fermée : au fur et à mesure que l'approche MDA évolue, l'OMG publie de nouveaux standards pour la soutenir. La spécification la plus récemment publiée [OMGe] concerne une extension du MOF définissant un cadre pour les langages de transformation de modèles (voir la section 2.4.1.4)

2.4.1.2 Principes

MDA est basée sur le principe de séparation entre la logique métier et la logique d'implémentation. Il en découle les trois règles suivantes : a) le modèle du système (le système étant un programme) et le modèle de la plate-forme (qui est une plate-forme logicielle) sur laquelle il sera exécuté sont conçus séparément, puis b) ils sont intégrés et enfin c) l'implémentation du système est générée automatiquement à partir du modèle résultant. Un objectif important de l'approche MDA est d'automatiser les processus de manipulation des modèles s'appuyant sur ces trois règles.

Le cycle de développement de l'approche MDA est vu sous la forme d'un Y [RV00] (voir figure 2.4) dont les branches représentent respectivement les spécifications fonctionnelles du système et les spécifications techniques de la plate-forme cible qui, une fois intégrées, mènent à l'implémentation. Notons que d'autres cycles ont été proposés [Béz02], dont notamment un cycle en double Y (voir figure 2.4), qui permet de séparer les spécifications non-fonctionnelles des spécifications fonctionnelles.

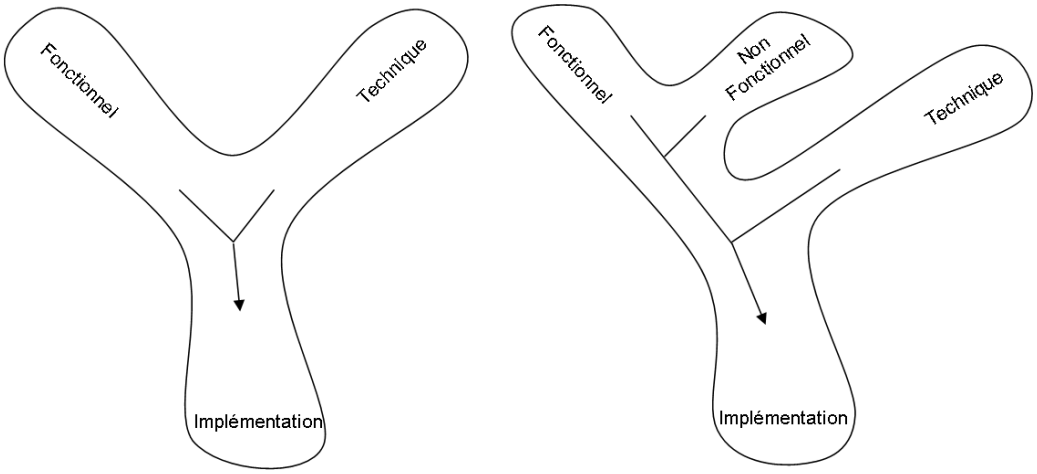


Fig. 2.4 – Les cycles de développement en Y et double Y

Dans l'approche MDA, le fossé entre le modèle et le système n'existe plus en théorie car le modèle *est* le système, ou tout du moins le système est sensé être généré automatiquement et directement à partir du modèle. En pratique cela n'est pas encore tout à fait vrai car, même si la génération de l'architecture du programme (code structurel) ne pose plus de problème à l'heure actuelle, la génération du comportement (code dynamique), par contre, est toujours source de nombreuses difficultés.

Les concepts qui constituent les fondements de l'approche MDA sont les suivants :

Vues La notion de vue utilisée dans MDA désigne une projection d'un modèle avec un point de perspective particulier. Chaque vue sert à mettre en évidence un aspect d'un système en masquant les autres. Par exemple, la vue de cas d'utilisation, la vue comportementale, ou encore la vue de déploiement sont des vues définies dans UML 2.0 [OMGl].

Abstraction/Raffinement Le mécanisme d'abstraction permet de construire, à partir d'un modèle, un modèle plus abstrait dans lequel certains détails du système sont occultés. Le mécanisme de raffinement est le mécanisme inverse, qui permet de construire, à partir d'un

modèle, un modèle plus détaillé dans lequel des informations sur le système sont ajoutées. Dans le processus d'abstraction ou dans le processus de raffinement, les deux modèles considérés sont liés l'un à l'autre par une relation dans laquelle le modèle le plus abstrait, donc aussi le moins détaillé, est appelé l'abstraction et le modèle le plus concret, donc le plus détaillé, est appelé la réalisation. Il est important de noter que cette relation impose des contraintes de préservation de propriétés entre les deux modèles. Cela la différencie notamment du mécanisme de vue. Dans la méthode B [Abr96], cette relation est appelée relation de conformité. Les mécanismes d'abstraction et de raffinement sont précisément définis par la méthode B, qui impose également des conditions strictes sur les propriétés des modèles liés par cette relation.

Méta-modélisation La méta-modélisation définit des ensembles de concepts utilisables pour réaliser des modèles. Un méta-modèle peut être utilisé pour définir un langage de modélisation. La méta-modélisation joue aujourd'hui un rôle central dans l'approche MDA, notamment à travers le standard MetaObject Facility (MOF) [OMGf] de l'OMG. Nous abordons plus en détails la méta-modélisation et les techniques d'ingénierie des langages de modélisation dans la section 2.4.1.5.

Transformation de modèles La transformation d'un modèle est le processus par lequel ce modèle est converti en un autre modèle (relatif au même système). Il existe différentes techniques de transformation, dont notamment des techniques basées sur l'utilisation des méta-modèles qui peuvent être complètement automatisées. Les transformations sont au cœur de l'approche MDA : elles permettent d'obtenir différentes vues d'un modèle, de raffiner ou d'abstraire un modèle, ou encore de réécrire un modèle dans un langage différent. Nous abordons dans les parties suivantes les différents types de modèles utilisés dans l'approche MDA et les transformations possibles entre eux.

2.4.1.3 Typologie des modèles dans l'approche MDA

L'OMG a défini une typologie de modèles, ainsi qu'un ensemble de relations de transformation qui permettent de passer de l'un à l'autre. Les quatre principaux types de modèles définis dans l'approche MDA [BB02] sont les suivants :

CIM (Computation Independant Model) Aussi appelé modèle de domaine ou modèle métier, le CIM capture les exigences en termes de besoins et décrit la situation dans laquelle le système sera utilisé. Son but est d'aider à la compréhension du problème mais aussi de fixer un vocabulaire commun pour un domaine particulier. Dans la pratique, l'appellation « CIM » est très peu utilisée.

PIM (Platform Independant Model) Le PIM décrit le système indépendamment de la plate-forme cible sur laquelle il s'exécutera. Il présente donc une vue fonctionnelle détaillée du système, sans détails techniques. Il peut être raffiné progressivement jusqu'à intégrer des détails d'architecture spécifiques à un type de plate-forme (machine virtuelle, système d'exploitation, etc.) mais il doit rester technologiquement neutre.

PDM (Platform Description Model) Le PDM est le modèle qui décrit une plate-forme d'exécution. Il fournit un ensemble de concepts techniques représentant les différentes parties de la plate-forme et/ou les services qu'elle fournit. Un PDM peut représenter, par exemple, des plates-formes à base de composants comme CCM [OMGc] ou EJB.

PSM (Platform Specific Model) Le PSM est le résultat de la combinaison du PIM et du PDM. Il représente une vue technique détaillée du système. Il peut exister avec différents niveaux de détails. Dans sa forme la plus détaillée, il sert de base à la génération de l'implémentation.

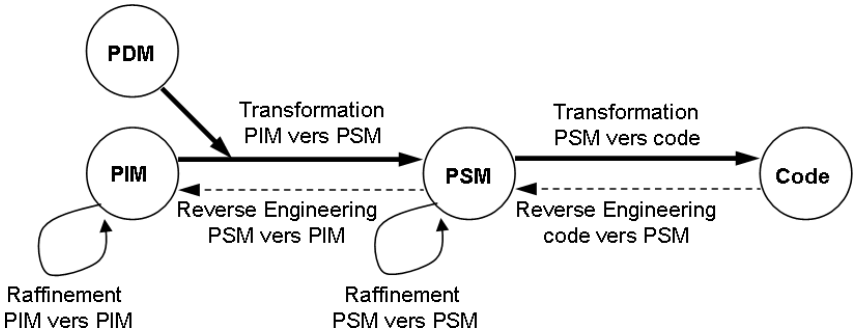


FIG. 2.5 – Modèles et transformations dans l'approche MDA

2.4.1.4 Transformations

Les transformations possibles entre ces différents types de modèles sont représentées sur la figure 2.5 :

Transformations PIM → PIM et PSM → PSM Les transformations de type PIM vers PIM ou PSM vers PSM visent à enrichir, filtrer ou spécialiser le modèle. Il s'agit de transformations de modèle à modèle [CH06]. Elles sont automatisables (ou partiellement automatisable) dans certains cas comme la traduction vers un autre langage mais les transformations de type raffinement ne le sont généralement pas.

Transformation PIM → PSM La transformation de PIM vers PSM permet de spécialiser le PIM en fonction de la plate-forme cible choisie. Elle n'est effectuée qu'une fois le PIM suffisamment raffiné. Cette transformation de modèle à modèle est réalisée en s'appuyant sur les informations fournies par le PDM.

Transformation PSM → code La transformation de PSM vers l'implémentation (le code) est une transformation de type modèle à texte [CH06]. Le code est parfois assimilé par certains à un PSM exécutable. Dans la pratique, il n'est généralement pas possible d'obtenir la totalité du code à partir du modèle et il est alors nécessaire de le compléter manuellement.

Transformations inverses PIM ← PSM et PSM ← code Ces transformations sont des opérations de rétro-ingénierie (reverse engineering). Ce type de transformation pose de nombreuses difficultés mais est essentiel pour la réutilisation de l'existant dans le cadre de l'approche MDA.

Les transformations jouent donc un rôle essentiel dans l'approche MDA. Elles n'étaient à l'origine supportées par aucun standard, c'est pourquoi l'OMG a récemment publié une spécification [OMGe] définissant un cadre basé sur le MOF pour la transformation des modèles dans le contexte de l'approche MDA.

De nombreux travaux visent à concevoir des techniques permettant de réaliser ces transformations de manière automatique, même lorsque les modèles manipulés sont complexes et tout en préservant les informations transformées. Dans [CH06], K. Czarnecki et S. Helsen ont identifié notamment les transformations de modèle à modèle dirigées par la structure, basées sur des templates (sortes de gabarits) ou encore basées sur des transformations de graphes. L'utilisation des méta-modèles est également de plus en plus courante dans ce domaine. Une taxonomie de transformations reposant sur l'identification de leurs principales caractéristiques est proposée dans [MG06]. Dans ces travaux, les auteurs proposent également un classement des langages et des outils de transformation existants.

2.4.1.5 Méta-modélisation et ingénierie des langages logiciels

Le langage UML a longtemps été au centre du processus de standardisation de l'OMG pour promouvoir l'approche MDA. Il était alors proposé comme un langage « universel » de modélisation. Cependant ses limites ont rapidement été atteintes, tout d'abord parce qu'il a été conçu spécifiquement pour concevoir des systèmes logiciels sur la base du paradigme objet mais également car il ne permet pas de modéliser des informations de type non fonctionnelles ou métier. Pour pallier la rigidité d'UML, le mécanisme de profil a alors été proposé pour permettre d'étendre le langage avec de nouvelles notions (voir la section 2.4.1.5.b). Face au succès de ce mécanisme d'ouverture, l'OMG a finalement adopté une approche permettant de définir et utiliser des langages spécifiques aux métiers ou technologies en jeu, les Domain Specific Languages (DSLs) (voir la section 2.4.1.5.a), et proposé de nouveaux standards et une nouvelle architecture pour la définition et la manipulation de langages de modélisation (voir la section 2.4.1.5.c).

2.4.1.5.a Domain Specific Languages (DSLs)

Les Domain Specific Languages (DSLs) [vDKV00], appelés également langages dédiés ou langages domaine-spécifiques, sont des langages conçus spécifiquement pour un domaine technique ou un domaine métier, comprenant généralement un petit nombre de concepts et utilisés par un nombre modeste d'utilisateurs spécialistes. Ces langages apportent une solution ciblée et efficace à un ensemble restreint et particulier de problèmes de modélisation ou de programmation. Il existe un très grand nombre de DSLs, avec des niveaux d'abstraction très différents.

Différentes études, dont celle documentée dans [KMB+96], permettent de dire que ces langages très spécifiques permettent aux spécialistes de gagner en productivité et en efficacité dans le traitement des problèmes par rapport à l'utilisation de langages généralistes (aussi appelés General Purpose Languages). Dans un contexte où la complexité des systèmes augmente, ces langages ont donc des qualités indéniables. Cependant, comme ils sont utilisés par de petits nombres de spécialistes, il est économiquement difficile de développer et maintenir l'outillage correspondant (environnements intégrés de développement, outils de génération de code, etc.).

Différentes initiatives ont vu le jour afin de permettre la génération automatique d'outils à partir de la définition du langage [AFR06]. Parmi les plus importantes, nous pouvons citer GME [Dav03], MetaEdit+ [Poh03], le Eclipse Modeling Project (incluant EMF, GEF et GMF) [Ecla], XMF-Mosaic (seule la partie XMF est maintenue à ce jour [CSW08]) ou encore l'initiative Microsoft [GSCK04] par exemple. Ces outils permettent de définir facilement des syntaxes concrètes graphiques ou textuelles et permettent même souvent de séparer syntaxe abstraite et syntaxe concrète. Aujourd'hui, l'utilisation des DSLs est activement soutenue par les communautés du domaine de la méta-modélisation (voir Paragraphe 2.4.1.5.c). Dans ce cadre, un nouveau domaine, connexe à l'IDM, est en train d'apparaître : l'Ingénierie des Langages Logiciels [DD06, Fon07]. L'influence des techniques développées dans ce domaine sur l'IDM est telle que certains auteurs parlent même de Language Driven Development (LDD) [CSW08].

2.4.1.5.b Profils UML

Le mécanisme de profil permet d'étendre ou de restreindre le méta-modèle d'UML de manière à l'adapter à un usage spécifique, pour un domaine métier ou un domaine technique. Il est possible, par exemple, d'ajouter de nouvelles constructions et de nouvelles règles d'assemblage spécifiques à la biologie.

Le mécanisme de profil permet de réaliser une ou plusieurs des modifications suivantes sur le méta-modèle UML :

- Restreindre à un sous-ensemble du méta-modèle UML.
- Spécifier de nouvelles règles de bonne formation de modèles sur la base d'ensembles de contraintes exprimées en OCL (Object Constraint Language [OMGh]).

– Spécifier de nouveaux éléments standards tels que des stéréotypes, des valeurs étiquetées ou des contraintes.
– Spécifier de nouveaux éléments de sémantique, exprimés en langage naturel.
– Spécifier des éléments de modèle communs, exprimés dans les termes du profil.

Il existe un ensemble de profils standards définis par l'OMG et réunis dans un catalogue en ligne [OMGa]. L'OMG a notamment standardisé des profils permettant de modéliser le temps pour la conception d'applications temps-réel embarquées [OMGj] ou d'exprimer des propriétés non fonctionnelles concernant la qualité de service et les performances par exemple [OMGk]. Notons qu'il est possible d'appliquer plusieurs profils en même temps dans un modèle.

2.4.1.5.c Méta-modélisation et MOF (MetaObject Facility)

La méta-modélisation prend sa source dans le besoin de représenter les concepts manipulés lors de l'utilisation des formalismes de modélisation. Dans le cadre de l'approche MDA, ce besoin est né des difficultés rencontrées dans les processus de transformations de modèles d'un langage à un autre : il s'agissait de décrire, de manière abstraite et détachée de l'instance de modèle considérée, les constructions du langage source pouvant être transformées ainsi que leurs équivalents dans le langage cible.

Dans le cadre de ses travaux concernant la méta-modélisation, l'OMG a d'une part défini la notion de méta-méta-modèle et d'autre part standardisé une architecture générale décrivant les liens entre modèles, méta-modèles et méta-méta-modèles. Cette architecture est classiquement hiérarchisée en « 3+1 » niveaux :

Niveau M3 (MMM) Dans l'approche MDA, le niveau M3 (ou méta-méta-modèle) est composé d'une unique entité qui s'appelle le MOF (MetaObject Facility [OMGf]). Le MOF permet de décrire la structure des méta-modèles, d'étendre ou de modifier les méta-modèles existants. Le MOF est réflexif, c'est-à-dire qu'il se décrit lui-même.

Niveau M2 (MM) Le niveau M2 (ou méta-modèle) définit le langage de modélisation et la grammaire de représentation des modèles M1. Le méta-modèle UML, qui est décrit dans le standard UML, et qui définit la structure interne des modèles UML, fait partie de ce niveau. Les profils UML, qui étendent le méta-modèle UML, appartiennent aussi à ce niveau. Les concepts définis par un méta-modèle sont des instances des concepts du MOF.

Niveau M1 (M) Le niveau M1 (ou modèle) est composé de modèles d'information. Il décrit les informations de M0. Les modèles UML, les PIM et les PSM appartiennent à ce niveau. Les éléments d'un modèle (M1) sont des instances des concepts décrits dans un méta-modèle (M2).

Niveau M0 Le niveau M0 (ou instance) correspond au monde réel. Il ne s'agit pas à proprement parler d'un niveau de modélisation. Ce niveau contient les informations réelles de l'utilisateur, instances du modèle du niveau M1.

La figure 2.6 présente ces « 3+1 » niveaux et illustre l'utilisation de cette hiérarchie dans le cas de modèles décrits en UML. Il est à noter que dans la version 2.0 du standard MOF, il est spécifié que le nombre de niveaux de modélisation utilisables est flexible sur la base d'au moins 2 niveaux.

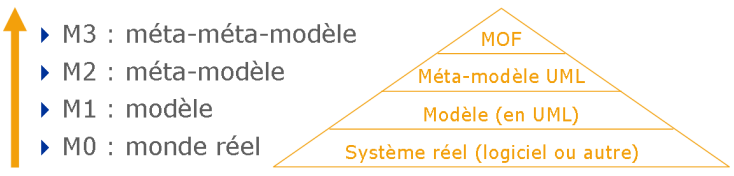


Fig. 2.6 – Exemple d'utilisation des « 3+1 » niveaux de modélisation dans l'approche MDA

Avec le domaine des transformations de modèles, la méta-modélisation est à l'origine des développements les plus récents et les plus importants de l'approche MDA. La méta-modélisation est utilisée de manière intensive dans le cadre des techniques d'ingénierie des langages de modélisation (voir Paragraphe 2.4.1.5.a). Cependant, si le traitement des syntaxes abstraites et concrètes est relativement bien adressé à l'heure actuelle, il n'existe encore que très peu de moyens d'associer une interprétation sémantique aux constructions syntaxiques. Les axes explorés pour exprimer les différents types de sémantiques comportementales possibles (opérationnelle, dénotationelle ou axiomatique) dans un cadre IDM sont, par exemple, l'extension d'un langage de méta-modélisation [MFJ05], l'utilisation de règles de réécriture (approche grammaire, graphe, QVT relationnel), etc. Il s'agit d'une problématique de première importance dans le domaine de l'IDM, car elle impacte notamment les phases de vérification et de validation du cycle de conception système.

2.4.1.6 Ouverture de l'approche MDA à l'IDM

L'OMG a récemment ouvert l'approche MDA à la modélisation de systèmes non exclusivement logiciels à travers la standardisation d'un profil appelé SysML [OMGi]. Les caractéristiques principales de ce profil sont notamment l'extension du paradigme purement objet d'UML avec des communications orientées flots de données dans les représentations structurelles du système et l'ajout de diagrammes permettant d'exprimer des exigences et des contraintes paramétriques pour l'analyse de performances. Les travaux sur ce profil ont été initiés conjointement par l'OMG et l'INCOSE (International Council on Systems Engineering, `http://www.incose.org/`) en 2003.

2.4.2 Autres approches centrées sur les modèles

L'approche MDA de l'OMG n'est pas la seule approche supportant et guidant l'utilisation des modèles en ingénierie des systèmes, logiciels notamment. Dans les parties suivantes nous passons en revue quelques-une des autres approches existantes.

2.4.2.1 Computer Aided Software Engineering – CASE

Apparue dans les années 80, l'idée initiale de l'approche CASE (Computer Aided Software Engineering) était de s'inspirer des outils de type CAD (Computer Aided Design), qui existaient déjà pour la conception des systèmes mécaniques notamment, afin d'outiller le génie logiciel. Le terme « CASE » [Cas85] a été déposé en 1982 par la Nastec Corporation of Southfield, Michigan avec leur éditeur GraphiText, qui a évolué plus tard pour devenir DesignAid, le premier outil logiciel permettant d'évaluer logiquement et sémantiquement des diagrammes de conception de systèmes et de logiciels. Sous la direction de Albert F. Case Jr. et de Vaughn Frick, la suite de produits DesignAid a été étendue pour supporter un large spectre de méthodologies de conception et d'analyse. Les outils de type CASE ont atteint leur heure de gloire au début des années 1990, époque à laquelle IBM avait proposé AD/Cycle [Mon91], un framework de développement CASE sur mainframe. Avec le déclin du mainframe, les grands outils de type CASE de l'époque ont disparu pour laisser la place aux logiciels sur ordinateurs personnels. Aujourd'hui, nous pouvons citer dans la catégorie des environnements de type CASE des suites de produits telles que celles des entreprises IBM Rational, Telelogic et Computer Associates.

Il n'y a pas de consensus sur la définition de ce qu'est un outil de type CASE. Nous retenons la définition généraliste suivante, donnée par le Software Engineering Institute de l'université Carnegie Mellon [Car, BCM+94] : "A CASE tool is a computer-based product aimed at supporting one or more software engineering activities within a software development process." [2]

2. « Un outil CASE est un produit logiciel destiné à supporter une ou plusieurs activités de génie logiciel dans

Les premières générations des outils de type CASE ciblaient l'automatisation de tâches isolées du cycle de développement telles que la production de la documentation, la gestion des versions et le support des méthodologies de conception. Le besoin de connecter ces différents d'outils a donné naissance aux environnements de type CASE tels que définis dans [BCM+94], c'est-à-dire intégrant différents outils de manière à constituer de véritables plates-formes de développement intégrées.

2.4.2.2 Model Integrated Computing – MIC

Apparue au milieu des années 90, l'approche Model-Integrated Computing (MIC) [SK97] promeut l'utilisation de modèles domaine-spécifiques à la base du développement logiciel. La motivation première de cette approche était d'apporter à l'ingénierie des systèmes logiciels complexes une méthodologie ainsi que des outils logiciels de support.

La méthodologie proposée est décomposée en deux phases. La première phase consiste à analyser le domaine d'application. L'objectif de cette phase est de trouver des paradigmes de modélisation appropriés et définir avec précision le langage de modélisation qui sera utilisé. Un outil automatique peut alors utiliser ces informations pour générer un environnement de modélisation dédié au domaine. Cet environnement est utilisé directement dans la seconde phase, qui consiste à modéliser l'application désirée.

La plate-forme GME (Generic Modeling Environment) [Dav03] est une implémentation de la méthodologie définie par l'approche MIC. Dans sa dernière version, GME est intégré à l'environnement Visual Studio .NET et propose un ensemble de langages et d'outils pour l'ingénierie des modèles et des langages. Cependant, il ne permet pas pour le moment de spécifier la sémantique des langages développés.

2.4.2.3 Software Factories

Les « Software Factories » [GSCK04] représentent la vision de Microsoft quant à l'ingénierie dirigée par les modèles. Cette approche s'inspire des principes des lignes d'assemblage dans l'industrie et repose sur les idées principales suivantes :

- Les lignes d'assemblage ne fabriquent qu'un seul type de produit avec de faibles points de variation. Ainsi les lignes de production dans l'automobile ne produisent qu'un seul type de voiture, avec des variations possibles pour la couleur ou les combinaisons d'options.
- Les ouvriers sont souvent spécialisés. Si certains ont parfois des activités relativement variées, elles ne couvrent jamais la totalité des activités de la ligne d'assemblage.
- Les outils sont très spécialisés et très automatisés. Ils ne peuvent généralement pas être réutilisés sur d'autres lignes de production que celle pour laquelle ils ont été conçus.
- Les composants à assembler ou à usiner proviennent souvent de tierces parties (fournisseurs). Par exemple, les lignes de production automobile assemblent généralement des pièces qui sont produites dans d'autres usines.

Ces principes ont prouvé leur efficacité dans le domaine de la fabrication de familles de matériels. L'approche des Software Factories propose d'appliquer ces principes au développement de logiciels. Suivant les deux premiers points, les fournisseurs de logiciel et les développeurs devraient être hautement spécialisés. Le troisième point suggère que les outils (de modélisation) devraient également spécialisés, c'est-à-dire être domaine-spécifiques. Ce principe inclut les langages de modélisation, les logiciels d'assistance ainsi que les transformations. Le dernier point pousse à la réutilisation de composants sur étagère. Ainsi, en résumé, une « Software Factory » est un environnement de développement configuré pour produire rapidement un type spécifique d'applications. L'environnement de développement intégré Microsoft Visual Studio

un processus de développement de logiciel »

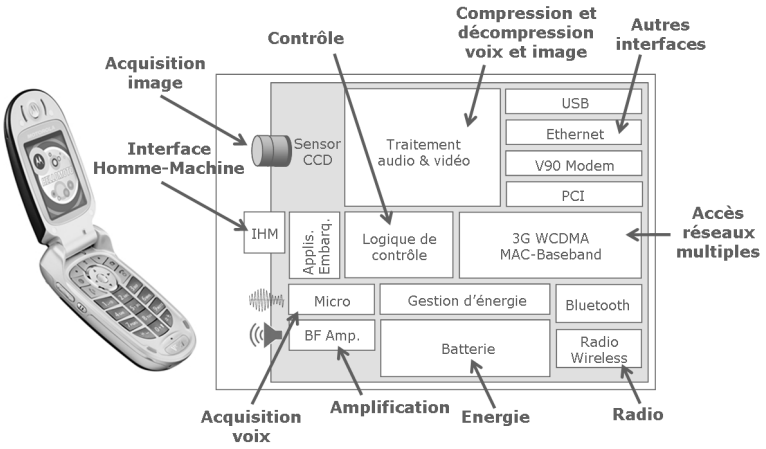


FIG. 2.7 – Un téléphone portable multimédia

.NET 2008 met en application ces idées et propose un framework de développement qui peut être configuré pour cibler un domaine particulier.

2.5 IDM appliquée aux systèmes complexes : notre vision du problème de l'hétérogénéité des modèles

L'application de l'ingénierie dirigée par les modèles aux systèmes complexes pose en fait de nombreux problèmes. Dans cette section nous allons introduire et illustrer sur un exemple le problème qui nous intéresse dans ce mémoire : le problème de l'hétérogénéité des modèles dans le cadre de l'IDM. Nous discuterons alors de ses conséquences.

2.5.1 Exemple introductif

Prenons l'exemple d'un téléphone multimédia tel que celui représenté sur la figure 2.7. Ce téléphone est un système relativement complexe car, même si il n'est pas composé d'un grand nombre d'éléments, il met en jeu différents domaines métiers tels que le traitement du signal, le logiciel, l'opto-électronique, l'électronique, la gestion de l'énergie, les réseaux, etc. Nous allons voir quels sont les types de modèles nécessaires pour concevoir ce système dans un processus de conception IDM.

Tout d'abord, pour la conception, ce système sera typiquement divisé en sous-systèmes qui seront conçus par des experts. Ainsi une équipe travaillera sur la chaîne de traitement du signal et une autre travaillera sur les applications logicielles embarquées par exemple. Pour le traitement du signal, les outils de modélisation à flots de données synchrones tels que Simulink (The MathWorks) sont couramment utilisés. Pour la conception des applications logicielles embarquées il existe de très nombreux langages de modélisation, parmi lesquels UML (qui peut être utilisé au travers d'un profil tel que MARTE [3] par exemple), Statecharts, les Communicating Sequential Processes ou les langages synchrones tels que Lustre et Esterel (qui sont supportés par l'environnement SCADE de Esterel Technologies).

Un autre aspect à ne pas négliger est que la conception de chaque sous-système progresse via différents niveaux de détail : les concepteurs commencent par formaliser des spécifications avec un haut niveau d'abstraction puis raffinent et transforment les modèles du sous-système considéré jusqu'à ce qu'ils soient assez détaillés pour passer à l'implémentation. Dans le cadre de notre exemple, si des machines à états finis sont utilisées pour modéliser la logique du contrôleur

3. MARTE est un profil UML pour la modelisation et l'analyse des systèmes embarqués et temps-réel

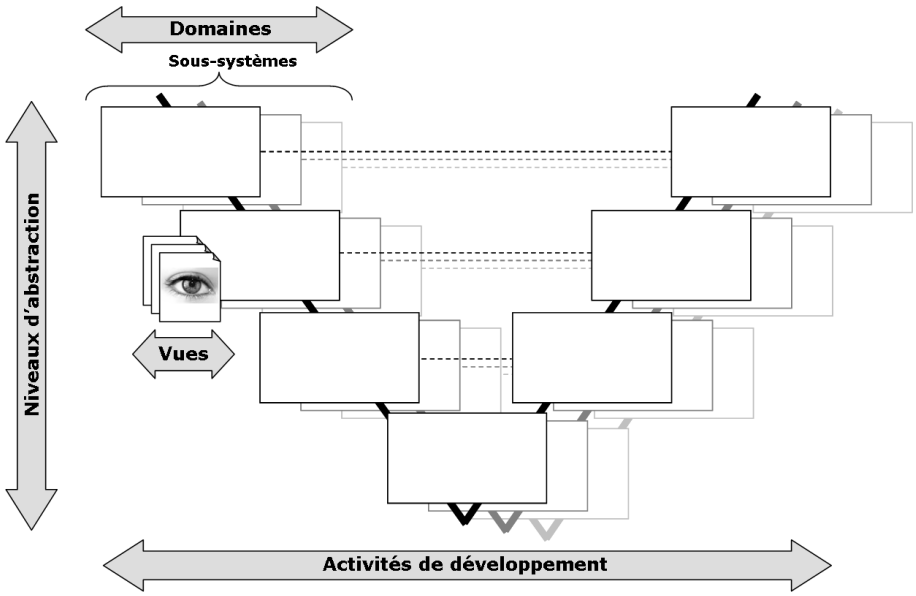


Fig. 2.8 – Quatre axes pour l'hétérogénéité des modèles

logiciel du téléphone, une version temporisée des machines à états finis pourra être utilisée pour raffiner le modèle et prendre en compte des informations telles que les délais d'expiration et les temps de réponse.

Par ailleurs, le processus de conception passe par différentes phases, notamment afin d'analyser le comportement modélisé pour vérifier sa cohérence et détecter d'éventuels problèmes. Différentes techniques telles que la simulation, le test ou le model-checking sont alors utilisées. Sur l'exemple de notre téléphone, si l'on souhaite s'assurer de l'absence d'interblocages (deadlocks) au niveau du comportement du contrôleur logiciel, une représentation de son comportement et de celui de son environnement sous la forme de réseaux de Pétri sera plus adéquate que sous la forme de machines à état fini.

Enfin, outre l'aspect fonctionnel, chaque sous-système pourra également être étudié selon d'autres points de vue afin de déterminer des propriétés non fonctionnelles sur le système telles que les performances, la consommation d'énergie, etc. Par exemple, si l'on considère l'un des objectifs de la conception de notre téléphone qui est la détermination de la capacité optimum de la batterie d'alimentation, une vue orientée consommation d'énergie du contrôleur pourrait être nécessaire. Un tel modèle utiliserait probablement des équations différentielles ou des automates hybrides.

Sur cet exemple, nous remarquons que le développement des systèmes complexes dans le contexte de l'ingénierie dirigée par les modèles fait appel à différentes techniques de modélisation dépendant à la fois :

- des différents domaines techniques et métiers en jeu dans le système conçu [CHM+99] : matériel, logiciel, contrôle, traitement du signal, mécanique, etc.
- des différentes phases du cycle de développement [CHM+99] : analyse des besoins, conception, intégration, test, etc.
- des différents niveaux d'abstraction auxquels le système est étudié [PY96] : niveau système, niveau algorithme, niveau registre, etc.
- des différents aspects spécifiques à analyser lors de la conception : fonction, performance, consommation d'énergie etc.

2.5.2 Objectifs de modélisation

Nous observons sur l'exemple de la section précédente que le choix du paradigme de modélisation le plus approprié à un instant donné dans la conception dépend principalement de quatre critères, outre les préférences personnelles du concepteur : le domaine du système ou du sous-système, le niveau d'abstraction désiré, l'aspect en cours d'étude, nécessitant de disposer d'une vue spécifique du système, et l'activité de conception courante. Ces quatre critères *<domaine, abstraction, vue, activité>*, qui sont fortement liés les uns aux autres, caractérisent l'objectif que le concepteur souhaite atteindre en réalisant un modèle du système.

Proposition 1. *Nous appelons* **objectif de modélisation** *l'objectif pour lequel un modèle d'un système est réalisé. Nous représentons un objectif de modélisation par un quadruplet :*

<domaine, abstraction, vue, activité>

- **domaine** *est le domaine technique ou métier du système ou sous-système considéré ;*
- **abstraction** *désigne le niveau de détail auquel ce système ou sous-système est étudié ;*
- **vue** *correspond à l'aspect sous lequel est considéré le système ou sous-système ;*
- **activité** *est l'activité du cycle de développement que souhaite mener le concepteur.*

Pendant le cycle de développement (voir la figure 2.8), la valeur de ces différents critères change de la manière suivante :

- il y a changement de *niveau d'abstraction* lors du passage d'un modèle à un autre modèle plus détaillé (raffinement) ;
- il y a changement de *vue* lors de l'étude des différents aspects d'un système ou d'un sous-système ;
- il y a changement de *domaine* lors de l'utilisation de techniques spécifiques à un métier. Un tel changement peut survenir en particulier lors d'un changement de vue ou lors du raffinement d'un modèle (lors du passage d'un modèle à base d'équations logiques à un modèle dans lequel les équations sont réalisées par des transistors par exemple) ;
- il y a changement d'*activité* au cours des différentes phases du cycle de développement.

Chacun de ces changements requiert généralement un changement de paradigme de modélisation de manière à utiliser le plus adapté pour l'objectif de modélisation considéré.

Ceci est *inévitable* si l'on considère par exemple qu'un développeur de logiciel embarqué ne peut évidemment pas utiliser les mêmes techniques de modélisation que celles utilisées par un électronicien. C'est également *essentiel* si l'on tient compte du fait que l'utilisation de techniques de modélisation spécifiques bien maîtrisées permet à des spécialistes de mieux se concentrer sur les problèmes liés au système et leur évite de faire des erreurs liées au manque de connaissance de la technique employée.

En conséquence, à un instant donné du cycle de développement, le système est décrit par un ensemble de modèles exprimés dans différents paradigmes.

2.5.3 Paradigme versus formalisme

Il est important de souligner ici la différence entre *paradigme de modélisation* [Nor99] et *formalisme (ou langage) de modélisation*. En effet, la notion de paradigme de modélisation désigne une manière de voir les choses, une façon de représenter le monde à travers un ensemble de concepts. La notion de formalisme de modélisation est beaucoup plus précise : un formalisme est basé sur un ensemble de concepts avec une sémantique précise et également un ensemble de règles d'écritures adaptées (appelé syntaxe concrète), comme évoqué dans la section 2.3.2.1. La notion de formalisme désigne une convention de notation (il y a consensus sur la forme et sur le sens).

En ce sens, un formalisme de modélisation peut donc s'appuyer sur un ou plusieurs paradigmes de modélisation. VHDL-AMS par exemple, repose à la fois sur un paradigme de temps

discret et sur un paradigme de temps continu. Inversement, un paradigme de modélisation peut avoir différents formalismes associés. Par exemple, le paradigme réactif synchrone est à la base de plusieurs formalismes tels que Lustre [HCRP91], Esterel [BC85] ou SIGNAL [BGJ91].

En conséquence, des modèles décrits dans des paradigmes différents ne sont pas nécessairement décrits avec des formalismes différents. Des modèles situés à différents niveaux d'abstraction peuvent donc être décrits avec le même langage de modélisation (SystemC par exemple). De plus, le même langage de modélisation peut être utilisé pour différentes activités au cours du cycle de développement. Le langage B [Abr96], par exemple, est utilisé pour la conception, la validation et la génération de code.

2.5.4 Hétérogénéité des modèles et conséquences

Nous avons conclu dans la section 2.5.2 ci-dessus qu'à un instant donné du cycle de développement, le système est décrit par un ensemble de modèles exprimés dans différents paradigmes. Par rapport à cette problématique et compte tenu des nuances existant entre les notions de paradigmes et de formalisme, nous pouvons donc préciser que différents cas peuvent être distingués, ainsi que cela est représenté dans le tableau 2.1 : (1) modèles décrits dans différents formalismes avec un paradigme unique sous-jacent, (2) modèles décrits dans différents paradigmes avec un formalisme unique de représentation et enfin (3) modèles décrits dans des paradigmes et des formalismes différents.

	Paradigmes identiques	**Paradigmes différents**
Formalismes identiques	homogène	hétérogène (2)
Formalismes différents	hétérogène (1)	hétérogène (3)

Tab. 2.1 – Types d'hétérogénéité

Proposition 2. *Nous appelons* **modèles hétérogènes** *des modèles qui sont décrits dans des paradigmes ou dans des formalismes différents.*

Dans le cas d'hétérogénéité (1), différents formalismes sont utilisés mais ils reposent sur des paradigmes quasi identiques. C'est le cas, par exemple, si l'on considère un modèle décrit avec SyncCharts [And96] et un autre décrit avec Esterel. En effet, même si l'un est basé directement sur les machines à états finis et l'autre sur le paradigme réactif-synchrone, leurs sémantiques sont complètement compatibles. Les paradigmes sous-jacents étant similaires, l'hétérogénéité est plus ou moins une hétérogénéité de format car il existe généralement des moyens de réécrire les modèles dans le formalisme le plus simple lié au paradigme considéré (même si les modèles obtenus ainsi peuvent être considérablement verbeux, des « raccourcis » syntaxiques étant souvent proposés dans les différents formalismes). Ainsi, dans notre exemple, il est possible de réécrire le modèle SyncCharts en Esterel automatiquement.

Dans le cas (2), un seul formalisme est utilisé pour décrire les différents modèles mais les paradigmes sous-jacents sont différents. Deux types de situations peuvent mener à ce type d'hétérogénéité. Dans le premier type de situation, le concepteur utilise un formalisme qui supporte différents paradigmes, comme B ou VHDL-AMS par exemple. Dans ce cas, il existe un lien sémantique entre les paradigmes sous-jacents, qui est établi dans le formalisme. Les difficultés liées à l'utilisation des différents modèles pour analyser des propriétés globales sur le système sont en fait assez réduites dans ce cas. Le deuxième type de situation concerne des modèles qui ont été décrits originellement dans deux paradigmes différents avec deux formalismes différents (Statecharts et Simulink par exemple) et traduits vers un formalisme commun (tel que C). Les modèles résultants sont alors décrits dans le même formalisme mais ont des paradigmes

sous-jacents fondamentalement différents. Il est alors très difficile de « recoller » ces modèles de manière à en faire un modèle global du système car les interactions entre ces modèles sont difficiles à identifier d'une part, et qu'il est difficile d'établir un pont sémantique entre elles d'autre part.

Enfin, le cas (3) concerne des modèles qui sont décrits dans des paradigmes et dans des formalismes différents. C'est dans ce cas que les difficultés sont les plus importantes car il y a à la fois peu de concepts en commun et pas de sémantique commune (les concepts communs ayant des sémantiques différentes). Prenons par exemple un modèle SyncCharts et un modèle Simulink. Ces deux modèles sont décrits à la fois dans des paradigmes et dans des formalismes différents. Pourtant il peut être souhaitable de savoir combiner ces deux modèles (sans les traduire) pour obtenir une description globale du comportement d'un système. Ainsi, le modèle Simulink peut être considéré comme décrivant le comportement du système lorsqu'il est dans un mode particulier de fonctionnement, décrit par le modèle SyncCharts sous la forme d'un « état ». Ce type de combinaison est similaire à l'utilisation de modèles modaux [LLEL03] (une façon de représenter les systèmes hybrides, voir la section 3.3.1). De nombreuses questions doivent être traitées afin que cette combinaison ait un sens : comment les évènements discrets du modèle SyncCharts doivent-ils être interprétés dans le modèle à temps continu Simulink et inversement, que se passe-t-il à l'arrivée dans un mode ou à la sortie d'un mode, l'état du modèle Simulink est-il préservé entre les différents modes du modèle SyncCharts, ou le comportement est-il calculé à partir d'une sorte d'état initial dans chaque mode, etc. En conclusion, la combinaison de modèles décrits à la fois dans des paradigmes et dans des formalismes différents est la plus difficile des hétérogénéités.

Dans ce mémoire nous nous intéressons tout particulièrement à l'hétérogénéité des paradigmes, c'est-à-dire aux cas de la deuxième colonne du tableau. Nous présentons dans la partie II de ce mémoire une approche dans laquelle nous réduisons le cas (3) au cas (2) en nous appuyant sur une syntaxe et une sémantique abstraites.

A un instant donné du cycle de développement, le système est donc décrit par un ensemble de modèles hétérogènes. Ces modèles représentent le même système mais ne forment pas un modèle global de ce système. Or un tel modèle est nécessaire pour raisonner sur des propriétés globales qui ne peuvent être déduites facilement en raisonnant individuellement sur les éléments du système.

L'intégration manuelle de modèles hétérogènes pour obtenir une vue globale du système est une tâche à la fois fastidieuse et source d'erreurs, sans oublier les problèmes liés à la traçabilité et à la maintenabilité lorsque des modifications doivent être reportées sur les modèles intégrés. L'analyse, la vérification et la validation de propriétés sur le système dans son ensemble, en particulier les propriétés liées à son comportement, sont donc des difficultés majeures.

2.5.5 Spécificités des systèmes embarqués

Les systèmes embarqués (voir la section 2.2.2) posent des problèmes spécifiques de conception liés à deux aspects particuliers :

- Ils sont hétérogènes par essence et leurs composantes logicielles et matérielles sont particulièrement intégrées du fait de leur nature embarquée (qui restreint l'espace qu'ils peuvent occuper par exemple) ;
- Ils subissent des contraintes physiques imposées à la fois par l'environnement (ce sont des systèmes réactifs) et par le système dans lequel ils sont embarqués (aussi appelé plate-forme lorsqu'il s'agit de logiciel embarqué).

De ce fait, la séparation entre le calcul et la physique, qui est l'une des idées centrale en informatique, n'est pas applicable aux systèmes embarqués [HS06]. Modéliser de tels systèmes

requiert donc l'utilisation conjointe de paradigmes du domaine physique, du domaine matériel (hardware) et du domaine logiciel.

Par ailleurs, les contraintes non-fonctionnelles qui s'appliquent à certains types de système embarqués tels que les systèmes temps-réels ou les systèmes critiques impliquent la prise en compte de critères de qualité stricts de bout en bout du processus de développement. L'utilisation de paradigmes adaptés à la fois pour les activités liées à la conception et pour les activités de vérification et de validation est donc essentielle.

Les problématiques liées à l'hétérogénéité des modèles sont particulièrement visibles dans le cas des systèmes embarqués, c'est pourquoi nous les prendrons comme exemple tout au long de ce mémoire.

2.6 Conclusion

Dans ce chapitre nous nous sommes attachés à mettre en évidence le rôle central des modèles et de l'outillage sous-jacent dans le processus de conception des systèmes tel qu'il est vu dans le contexte de l'IDM. Nous avons présenté différentes méthodologies de conception, notamment pour la conception de systèmes logiciels. Au cours de ce chapitre, nous avons identifié et mis en valeur les problématiques actuelles liées à la manipulation des modèles et résultant de la complexité croissante des systèmes et du besoin de s'assurer de leur qualité tout en réduisant les coûts et le délai de production. Dans le cadre de l'approche MDA de l'OMG, nous avons présenté différentes techniques de traitement des modèles parmi lesquelles nous avons détaillé les techniques de transformation de modèles ainsi que la méta-modélisation. Ces techniques sont aujourd'hui des pivots majeurs pour l'innovation dans les domaines de recherche liés à la modélisation des systèmes. Enfin, nous avons montré quel impact l'hétérogénéité, omniprésente dans les systèmes et dans les méthodes de conception des systèmes, a sur l'ingénierie système. Nous avons introduit la notion fondamentale d'hétérogénéité des modèles et évoqué les différentes problématiques en découlant. Dans le chapitre suivant nous présentons le domaine de recherche attaché à l'étude de ces problématiques et proposons un état de l'art des techniques proposées pour les résoudre.

3

Modélisation multi-paradigme

3.1 Introduction

Dans le chapitre précédent nous avons vu que l'application de l'ingénierie dirigée par les modèles à la conception des systèmes complexes a pour conséquence la multiplication des modèles d'un même système et que ceux-ci sont décrits dans différents paradigmes. Nous avons vu que l'utilisation de multiples paradigmes au cours du cycle de développement est à la fois inévitable et essentielle. Dans ce chapitre, nous nous intéressons donc à la problématique suivante : comment permettre aux concepteurs d'utiliser conjointement de multiples paradigmes de modélisation au cours du cycle de développement tout en leur permettant de raisonner globalement sur le système à un instant donné du cycle ?

3.2 Modélisation multi-paradigme

3.2.1 Travaux connexes

La problématique du traitement de l'hétérogénéité des modèles dans le respect et la reconnaissance de son utilité est maintenant relativement bien identifiée. Dans [MV04], P. Mosterman et H. Vangheluwe expliquent notamment :

> "The heterogeneity of the design process, as much as of the system itself, however, requires a manifold of formalisms tailored to the specific task at hand. Efficient design approaches aim to combine different models of a system under study and maximally use the knowledge captured in them." [1]

Dans ces mêmes travaux, P. Mosterman et H. Vangheluwe définissent le champ de recherche appelé « Computer Automated Multi-Paradigm Modeling (CAMPaM) » qui adresse les problématiques issues de ce constat et repose sur trois directions de recherche : (1) la relation entre des modèles à des niveaux d'abstraction différents, (2) la modélisation multi-formalisme, c'est-à-dire le couplage et la transformation entre des modèles décrits en utilisant des formalismes différents, et (3) la méta-modélisation, c'est-à-dire la description de formalismes de modélisation.

De nombreux travaux sur le sujet de l'hétérogénéité des modèles ont également été conduits à Berkeley, sous la direction du professeur E. A. Lee. Dès 1990, E. A. Lee s'intéresse à la combinaison de différents paradigmes de modélisation (considérés au travers de la notion de « modèles de calcul ») et introduit la notion de modélisation hétérogène. La plate-forme de modélisation Ptolemy est mise au point dans ce contexte pour permettre d'expérimenter sur la combinaison de multiples modèles de calcul [BHLM91] :

> "Ptolemy is motivated by the increasingly important role of high-level system design, the increasing proliferation of simulation platforms and computational models, and the increasing need to combine these computational models. Some of the goals of Ptolemy include: [...] Specify each subsystem of an application using a domain most natural for that subsystem, and yet seamlessly combined with other subsystems to form a heterogeneous whole. [...]" [2]

1. « Cependant, l'hétérogénéité du processus de conception, tout autant que l'hétérogénéité du système lui-même, requiert une variété de formalismes adaptés à la tâche spécifique en jeu. Les approches de conception efficaces ont pour but de combiner différents modèles d'un système en cours d'étude et d'utiliser de manière optimale les connaissances qu'ils capturent. »

2. « Le projet Ptolemy est motivé par le rôle de plus en plus important de la conception système à un haut niveau d'abstraction, la prolifération croissante des plates-formes de simulation et des modèles de calcul, et le besoin grandissant de combiner ces modèles de calcul. Les objectifs de Ptolemy comprennent : [...] La spécification de chaque sous-système d'une application en utilisant le domaine métier le plus naturel pour ce sous-système, et cependant parfaitement combiné avec les autres sous-systèmes pour former un tout hétérogène. [...] »

Dans ce mémoire, nous proposons d'étendre et de généraliser les définitions proposées dans ces travaux en prenant en compte l'ensemble des axes d'hétérogénéité que nous avons évoqués dans le chapitre précédent (voir la section 2.5.2).

3.2.2 Présentation

Proposition 3. *Nous appelons* **Modélisation Multi-Paradigme***, ou modélisation hétérogène, la discipline dont l'objectif est de faciliter et automatiser l'utilisation conjointe de modèles hétérogènes pendant le cycle de développement de manière à rendre possible un raisonnement global sur un ensemble de modèles hétérogènes.*

Cette discipline traite donc des problématiques liées à l'hétérogénéité des modèles telle que nous l'avons définie dans le chapitre précédent (voir la section 2.5.4). Par rapport à la définition de CAMPaM, nous considérons notamment le problème de la relation d'abstraction/raffinement comme un type particulier d'hétérogénéité et donc comme une possible application d'une méthode de modélisation multi-paradigme. De plus, nous voyons le couplage et la transformation de modèles, ainsi que la description de formalismes de modélisation comme des techniques qu'il est possible d'utiliser pour atteindre les objectifs de la modélisation multi-paradigme. Nous nous appuyons également sur la définition de la modélisation hétérogène hiérarchique proposée par E. A. Lee [EJL+03] et implémentée dans le projet Ptolemy II. De notre point de vue, la problématique adressée par E. A. Lee est du type composition de modèles de différents domaines (voir la section 3.2.3.1). Nous détaillons ces travaux dans la section 3.3.4.4.

La réalisation des objectifs de la modélisation multi-paradigme implique l'automatisation de différentes actions sur les modèles telles que les transformations de modèles, la composition de modèles, l'exécution conjointe de modèles, etc. Il est à noter également que, par rapport au cycle de développement, la modélisation multi-paradigme est une discipline transversale : elle s'applique à différentes activités du cycle de développement telles que la spécification, la simulation, la vérification, la génération de code ou encore le test. Nous verrons dans la section 3.3 quelles sont les techniques au cœur de la modélisation multi-paradigme et comment elles peuvent être utilisées au cours des différentes activités du cycle de développement.

3.2.3 Axes de recherche

Nous avons vu dans la section 2.5.2 du chapitre précédent que les causes de l'hétérogénéité des modèles se déclinaient selon quatre axes : hétérogénéité des domaines, hétérogénéité des niveaux d'abstraction, hétérogénéité des vues et hétérogénéité des activités. En correspondance avec ces quatre causes d'hétérogénéité, sources de difficultés particulières, nous identifions quatre grands axes de recherche liés à :

- La composition de modèles de différents domaines (c'est-à-dire des modèles concernant différents métiers) ;
- Le traitement, de manière automatique et/ou formelle, de la relation d'abstraction/raffinement entre modèles hétérogènes ;
- La combinaison de vues hétérogènes d'un même modèle ;
- L'utilisation d'un même modèle pour différentes activités.

Nous passons en revue quelques-unes des principales questions sous-jacentes à ces différentes catégories de problèmes dans les sections suivantes.

3.2.3.1 Composition de modèles de différents domaines

Ce type de composition est nécessaire lorsque l'on considère les modèles des différentes parties du système et que ces différentes parties font appel à des métiers différents et donc à des domaines techniques différents (domaines du logiciel, de la mécanique, de l'optique, etc.).

Lorsque l'on veut composer des modèles de différents domaines techniques, qui utilisent donc des paradigmes de modélisation différents, le premier problème à traiter est de savoir si ces modèles peuvent être composés, c'est-à-dire si la composition d'un modèle d'un domaine avec un modèle d'un autre domaine peut avoir un sens. Cette question se pose du fait des différentes sémantiques à la base des paradigmes employés dans les modèles à composer. Ces sémantiques peuvent ne pas être « compatibles » c'est-à-dire que leur interaction peut n'avoir aucun sens ou encore donner un sens différent à la composition que celui souhaité. L'étude de ce problème nécessite l'analyse et la comparaison des différents paradigmes utilisés pour la modélisation du comportement des systèmes.

En admettant que cette question soit résolue, la question suivante est de déterminer par quel moyen réaliser la composition. Ces moyens sont multiples et incluent les mécanismes de composition de sémantique, d'adaptation logicielle ou encore de transformation de modèles. Nous proposons un état de l'art des différentes techniques liées à la composition de modèles hétérogènes dans la section 3.3.

Enfin, il reste à déterminer si, une fois la composition effectuée, la sémantique des modèles composés a bien été préservée. En effet, si cette sémantique originale est perdue ou modifiée il devient difficile de maintenir le modèle, de préserver sa cohérence lors de modifications et de déterminer l'origine de potentiels problèmes de conception.

3.2.3.2 Relation d'abstraction/raffinement entre modèles hétérogènes

Dans le cadre d'une approche descendante du cycle de conception, les modèles sont progressivement raffinés jusqu'à mener à l'implémentation. Il existe alors une relation appelée relation d'abstraction/raffinement entre deux modèles d'un même élément mais à deux niveaux de détail différents : l'un correspond à une vue plus abstraite de l'élément et l'autre à une vue plus raffinée de ce même élément. Vis-à-vis de cette relation et lors d'un processus de raffinement manuel, une problématique classique est de savoir prouver que le modèle obtenu lors du raffinement est bien un raffinement du modèle de départ, c'est-à-dire que le comportement qu'il modélise est bien conforme à celui modélisé dans le modèle plus abstrait. La problématique de la conformité est une problématique classique. Nous retrouvons cette problématique dans notre contexte, avec la difficulté supplémentaire que les deux modèles considérés sont dans des paradigmes ou des formalismes différents.

Un autre axe de recherche dans ce domaine est l'obtention de l'un des deux modèles à partir de l'autre (obtention du modèle raffiné à partir du modèle abstrait et inversement) par un processus de transformation automatique. La question est loin d'être triviale car cela nécessite dans un cas d'ajouter de l'information et dans l'autre d'en masquer, en plus de transformer les éléments d'un langage en des éléments d'un autre langage.

Enfin, le problème de la composition de modèles hétérogènes à des niveaux d'abstraction différents se pose également, avec des problématiques similaires à celles évoquées dans la section 3.2.3.1 mais compliquées par les questions de compatibilité entre niveaux d'abstraction.

3.2.3.3 Combinaison de vues hétérogènes d'un même modèle

Lorsque l'on étudie un élément d'un système sous plusieurs angles, notamment pour analyser ses propriétés non-fonctionnelles, on utilise différentes vues du même modèle. Ces différentes vues font souvent appel à des domaines métiers différents. Cependant ces vues sont généralement liées par des éléments communs qui influencent le comportement observé dans chacune des vues. Ainsi par exemple, un contrôleur logiciel qui fonctionne selon différents modes n'aura pas la même consommation d'énergie dans chacun de ces modes. La fonction a donc un impact sur la consommation d'énergie. Une première difficulté dans ce contexte est de maintenir la cohérence entre ces vues lors de modifications qui les impactent conjointement. De la même manière

la simulation conjointe de différentes vues d'un même modèle pose également de nombreuses difficultés.

3.2.3.4 Utilisation d'un même modèle pour différentes activités

Il est rare qu'un même modèle puisse être utilisé pour des activités différentes du cycle de développement. En effet, les paradigmes de modélisation utilisés sont généralement différents dans les différentes phases du cycle. Par exemple, les paradigmes utilisés lors des activités de spécification (expression des besoins par exemple) et de vérification (model-checking par exemple) sont généralement très différents. Ainsi, pour l'expression des besoins il est possible d'utiliser les diagrammes de cas d'utilisation d'UML tandis que pour du model-checking on utilisera plutôt un langage tel que Lustre. Le problème dans ce cas est que les modèles obtenus dans chacune des activités du cycle de développement sont déconnectés les uns des autres alors qu'ils concernent le même système, ce qui amène à mettre en place des techniques de validation très lourdes afin de vérifier qu'ils sont cohérents.

Un autre exemple caractéristique de ce type de situation apparaît si l'on considère les paradigmes utilisés pour faire du model-checking (activité de vérification) et de la génération de code (activité d'implémentation). Une formalisation mathématique de la sémantique du paradigme utilisé est impérative pour le model-checking. En ce qui concerne la génération du code, le paradigme utilisé doit avoir un niveau d'abstraction adéquat pour permettre l'expression de détails d'implémentation. Souvent, ces deux propriétés sont incompatibles car la formalisation de la sémantique d'un langage est d'autant plus difficile que ce langage est riche. Il arrive donc que des modèles différents soient utilisés pour ces deux activités, ce qui peut mener à l'obtention d'un code qui a un comportement différent du comportement spécifié par le modèle qui a été vérifié par model-checking. L'utilisation d'un même modèle pour différentes activités répond donc à une vraie problématique d'ingénierie.

Certains formalismes supportent parfois plusieurs activités. C'est notamment le cas de Lustre, qui permet de faire du model-checking et de la génération de code. Cependant, ces solutions sont des solutions « ad hoc », qui concernent des paradigmes spécifiques et des tâches particulières (généralement un couple de tâches avec une tâche de l'activité conception et une tâche de l'activité vérification) et qui ne peuvent donc pas couvrir l'ensemble du cycle de bout en bout. Des solutions plus générales sont nécessaires pour permettre de maintenir automatiquement la cohérence entre les différents modèles tout au long du développement du système.

3.3 Etat de l'art des techniques pour la modélisation multi-paradigme

La modélisation multi-paradigme est un champ de recherche multi-disciplinaire par essence. Il est donc naturel qu'elle implique diverses communautés de recherche, spécialisées dans des disciplines variées telles que l'automatique, le traitement du signal, le model-checking, l'ingénierie des langages de modélisation ou le développement de systèmes sur puce. Dans cette section, nous passons en revue des techniques de différents horizons qui, d'une manière où d'une autre, permettent d'adresser une des problématiques de la modélisation multi-paradigme telle que nous l'avons définie précédemment.

3.3.1 Approches spécifiques

De nombreuses approches ciblent des combinaisons de paradigmes particuliers, notamment afin de résoudre les problèmes liés à l'hétérogénéité des temps et des modes de synchronisation.

3.3.1.1 Hétérogénéité des temps

La combinaison de paradigmes basés sur le temps continu et sur le temps discret dans les modèles a été largement étudiée dans les approches ciblant la modélisation des systèmes hybrides. Des études de l'état de l'art sur ce sujet peuvent être trouvées dans [MV04] et dans [CBP+04]. Les approches citées incluent, entre autres, les automates hybrides, VHDL-AMS, Simulink/Stateflow et Modelica [FE98].

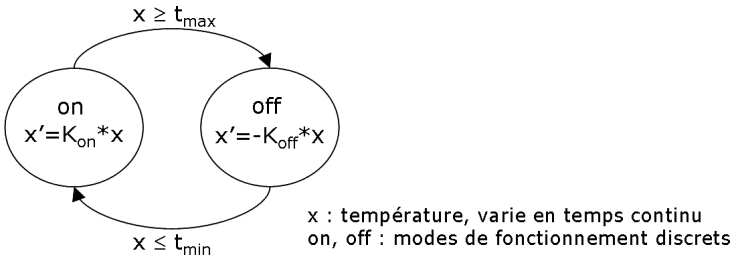


FIG. 3.1 – Représentation du fonctionnement d'un thermostat avec un automate hybride

La figure 3.1 montre un exemple d'automate hybride qui modélise le comportement d'un thermostat. Sur cet exemple, le thermostat a deux modes de fonctionnement. Lorsque la température est inférieure à $t_(min)$ le thermostat déclenche le chauffage. La montée de la température est modélisée par l'équation dans l'état « on ». Lorsque la température dépasse $t_(max)$, le thermostat arrête le chauffage. La température baisse alors, ce qui est représenté par l'équation dans l'état « off ». Cet exemple illustre bien la composition de deux aspects temporels : un aspect continu représenté par les équations et un aspect discret représenté par les différents états de l'automate.

3.3.1.2 Hétérogénéité des modes de synchronisation

En ce qui concerne l'hétérogénéité des paradigmes de synchronisation, plusieurs approches adressent les problèmes liés aux systèmes « Globalement Asynchrones – Localement Synchrones (GALS) » [Cha84]. La figure 3.2 montre un exemple de système GALS qui illustre le principe de ces systèmes : les éléments du système fonctionnent de manière synchrone en interne (c'est-à-dire que leur fonctionnement est cadencé par une horloge) mais communiquent via un medium de type bus ou tampon FIFO de manière asynchrone (c'est-à-dire à des instants non obligatoirement définis vis-à-vis des horloges respectives des éléments impliqués dans la communication). Une étude de l'état de l'art sur les techniques liées aux systèmes de type GALS (flot de conception, définition d'architecture, applications) est proposée dans [KGGV07].

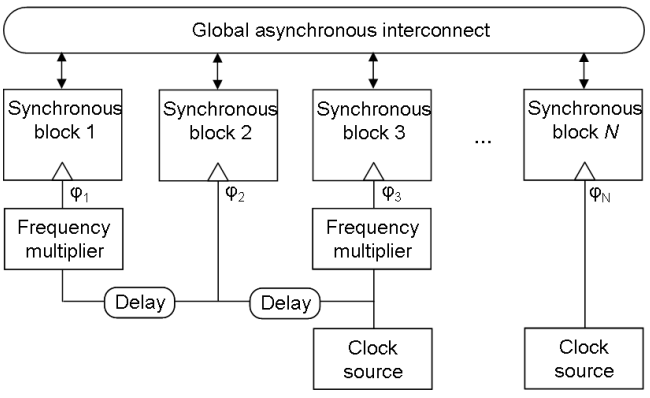


FIG. 3.2 – Principe des systèmes GALS (source : [TGL07])

Nous appelons les approches vues dans cette section « approches spécifiques »car elles permettent de résoudre le problème de l'hétérogénéité de manière ad hoc en permettant la combinaison de quelques paradigmes de modélisation particuliers. Cependant, dans le cadre de l'ingénierie dirigée par les modèles, le nombre des langages et paradigmes de modélisation différents utilisés tend à croître exponentiellement (notamment à travers l'utilisation de langages domaine-spécifiques, voir la section 2.4.1.5.a). Ces approches ne sont donc pas assez généralistes et flexibles pour répondre aux nouvelles problématiques liées à l'hétérogénéité des langages de modélisation dans le cadre de l'ingénierie dirigée par les modèles.

3.3.2 Prérequis : spécification de la syntaxe et de la sémantique des langages de modélisation

Dans le cadre d'approches plus flexibles pour le traitement de l'hétérogénéité, l'objectif est de pouvoir traiter un ensemble variable de langages de modélisation différents. Cela nécessite de pouvoir capturer facilement la syntaxe et la sémantique des langages de modélisation en jeu. Comme cela est souligné dans [MV04], la méta-modélisation est une technologie centrale lorsque l'on manipule des langages de modélisation (voir également la section 2.4.1.5.c). La méta-modélisation peut être utilisée notamment pour décrire la syntaxe abstraite d'un langage. Pour définir ensuite la sémantique des concepts ainsi spécifiés, il existe différentes approches.

3.3.2.1 Kermeta

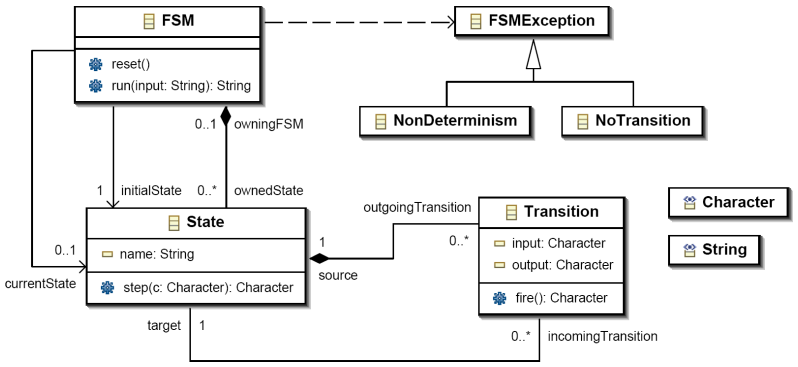
En utilisant Kermeta [MFJ05] par exemple, il est possible de donner au langage une sémantique exécutable en définissant des méthodes avec une sémantique impérative sur les éléments du méta-modèle. Ainsi, chaque élément du méta-modèle est pourvu d'une méthode d'exécution. La méthode d'exécution d'un élément peut contenir des appels aux méthodes d'exécution d'autres éléments avec lesquels il est en relation (principe de navigation sur les relations similaire à celui que l'on peut retrouver dans OCL [OMGh]). Ces opérations, définies au niveau méta, permettent alors d'exécuter n'importe quel modèle qui est conforme au méta-modèle du langage.

La figure 3.3 illustre l'utilisation de Kermeta sur une version simple du paradigme des machines à états finis (Finite State Machines — FSM). La figure 3.3a présente le méta-modèle définissant la syntaxe abstraite du langage. Les opérations associées aux éléments du méta-modèle sont décrites en utilisant le langage Kermeta afin de spécifier la sémantique d'exécution sous-jacente. La figure 3.3b montre le code de l'opération « step » de l'élément « State » du méta-modèle. Cette opération est chargée, lorsqu'elle est déclenchée, de vérifier si une des transitions sortant de l'état a été validée par l'arrivée d'un évènement, auquel cas elle déclenche la transition, faisant ainsi changer l'automate d'état.

3.3.2.2 Semantic Units (SUs)

Le principe des « Semantic Units (SUs) » est introduit dans [CSN+05]. Une Semantic Unit (littéralement une unité de sens) permet, lorsqu'elle est associée à un méta-modèle représentant la syntaxe abstraite d'un langage, de donner une sémantique opérationnelle à ce langage.

Une Semantic Unit est composée de (a) un modèle de données abstrait et (b) un ensemble de règles opérationnelles manipulant les éléments définis dans le modèle de données (a). Ces deux éléments sont décrits en AsmL (Abstract State Machine Language), un langage qui permet de représenter des machines d'état abstraites [BH98] et qui a une sémantique formelle. Une SU est associée à un méta-modèle par l'intermédiaire d'un « mapping », c'est à dire d'un ensemble de règles permettant d'associer chaque élément du méta-modèle à un élément dans le modèle de données abstrait (a). Ainsi, grâce à ce mapping, il est possible d'associer des éléments de données de la SU aux éléments de chaque modèle conforme au méta-modèle. Ceux-ci peuvent alors être



(a) Méta-modèle

```
operation step(c : Character) : Character raises FSMException is do
    // Get the valid transitions
    var validTransitions : Collection<Transition>
    validTransitions := outgoingTransition.collect { t | t.input.equals(c) }
    // Check if there is one and only one valid transition
    if validTransitions.isEmpty then raise NoTransition.new end
    if validTransitions.size > 1 then raise NonDeterminism.new end
    // fire the transition
    result := validTransitions.one.fire
end
```

(b) Code de l'opération « step » de l'élément « State » en Kermeta

FIG. 3.3 – Description du paradigme des machines à état finis avec Kermeta (source : [MFJ05])

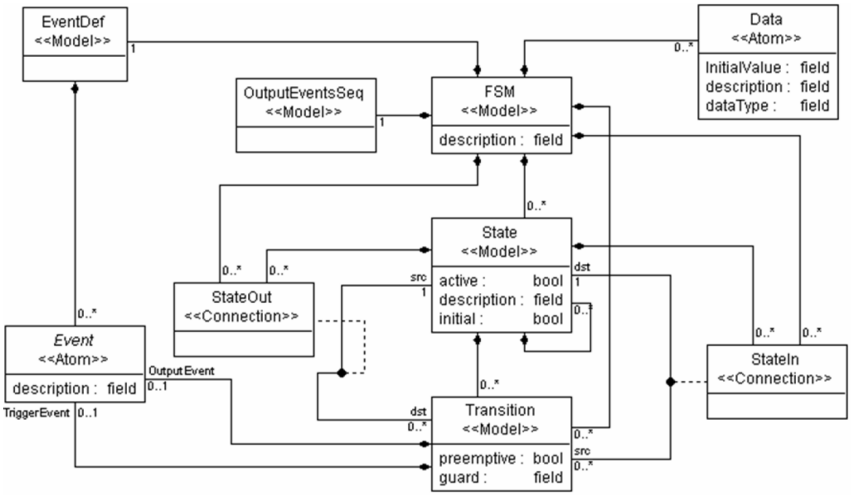
manipulés par les opérations définies dans la partie (b) de la SU. Ces opérations peuvent être considérées comme des règles d'exécution relativement similaires à celles que l'on peut définir avec Kermeta (voir le paragraphe précédent) sauf qu'elles ne sont pas directement rattachées aux éléments du méta-modèle.

La figure 3.4 illustre l'utilisation d'une Semantic Unit sur une version simple du paradigme des machines à états finis (Finite State Machines — FSM). La figure 3.4a présente le méta-modèle définissant la syntaxe abstraite du langage. La SU donnant une sémantique à ce paradigme comprend donc un modèle de données abstrait et un ensemble d'opérations. La figure 3.4b montre la définition en AsmL des éléments « FSM » et « event », qui représentent respectivement une machine à états finis et un évènement, dans le modèle abstrait de donnée. La figure 3.4c montre le code en AsmL de l'opération « fsmReact ». Cette opération prend en entrée une machine à états finis et un évènement et calcule la réaction de la machine à cet évènement. Une fois la SU définie, il reste à définir le mapping entre les éléments du méta-modèle de la figure 3.4a et les éléments du modèle de données comme ceux montrés par la figure 3.4b. La figure 3.4d montre l'une des règles de ce mapping décrite dans l'outil GReAT [fSISIVU], un outil de transformation de modèles basé sur la réécriture de graphes (voir les sections 2.4.1.4 et 3.3.3 pour plus de détails sur les transformations de modèles).

3.3.2.3 Modèles de Calcul (Models of Computation – MoCs)

Une troisième façon de définir un langage de modélisation (syntaxe et sémantique) est basée sur le concept de *Modèle de Calcul* (Model of Computation – MoC).

Le terme « modèle de calcul » étant relativement commun et général, de multiples sens lui sont associés. Cependant le concept de « modèle de calcul »qui nous intéresse dans le contexte de la modélisation multi-paradigme est celui introduit par E. A. Lee et A. L. Sangiovanni-Vincentelli dans [LSV98]. Dans ce contexte, un modèle de calcul peut être vu comme un ensemble de pri-



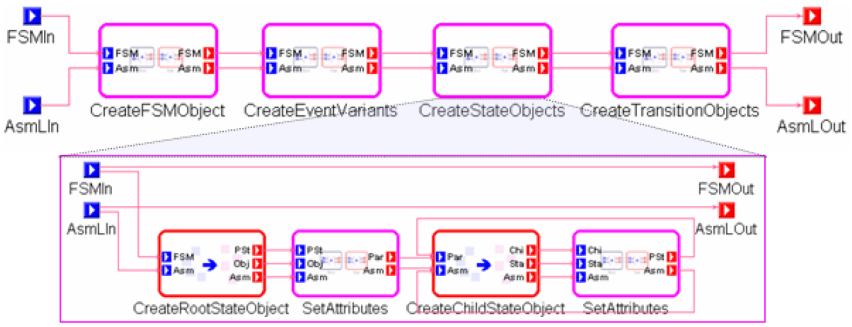
(a) Méta-modèle

```
structure Event
class FSM
  var outputEvents as Seq of Event
  var initialState as State
  var children     as Set of State
```

(b) Définition de « Event » et « FSM » dans le modèle de données abstrait en Asml

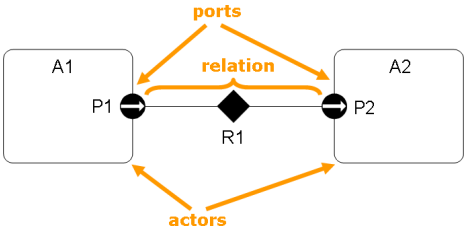
```
fsmReact (fsm as FSM, e as Event) =
  step let s as State = getCurrentState (fsm, e)
  step let pt as Transition? = getPreemptiveTrasition (fsm, s, e)
  step
    if pt <> null then doTransition (fsm, s, pt)
    else
     step
       if isHierarchicalState (s) then invokeSlaves (fsm, s, e)
       let npt as Transition? = getNonpreemptiveTransition (fsm,s,e)
     step
       if npt <> null then doTransition (fsm, s, npt)
```

(c) Code de l'opération « fsmReact » en AsmL

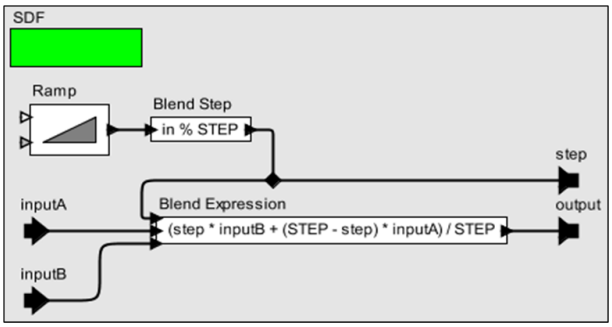


(d) Règle appartenant au mapping entre le méta-modèle de la figure 3.4a et les éléments du modèle de données abstrait

FIG. 3.4 – Description du paradigme des machines à état finis par une Semantic Unit (source : [CSAJ05])



(a) Syntaxe abstraite



(b) Modèle avec le directeur de domaine SDF (Synchronous DataFlow) (source : PtolemyII)

FIG. 3.5 – Syntaxe et sémantique avec Ptolemy

mitives sémantiques communes à plusieurs langages de modélisation. Ainsi par exemple, les langages basés sur le principe des processus séquentiels communiquant par rendez-vous font appel au même modèle de calcul (Communicating Sequential Processes – CSP), même s'ils présentent des variantes. De même, les langages basés sur le principe des flots de données synchrones font appel à un autre modèle de calcul (Synchronous DataFlow – SDF). En résumé, chaque modèle de calcul permet en fait de caractériser une classe de langages de modélisation. Dans [LSV98], les propriétés de différents modèles de calculs sont exprimées dans une théorie appelée « Tagged Signal Model » afin de pouvoir les comparer. Ces propriétés incluent le type de temps utilisé (discret, continu), le mode de communication (rendez-vous, messages, évènements, etc.) ou encore le type de synchronisation utilisé (synchrone, asynchrone). Nous présentons quelques modèles de calcul courants dans l'annexe A.

Le concept de modèle de calcul ainsi défini est mis en oeuvre dans le projet Ptolemy II [EJL+03]. L'approche proposée dans Ptolemy est basée sur : (a) l'utilisation d'une syntaxe abstraite unique à base de blocs appelés acteurs pour exprimer tous les modèles et (b) l'association à tout modèle d'un « domaine » qui définit la sémantique donnée à la syntaxe.

La syntaxe abstraite de Ptolemy repose sur un ensemble d'éléments de base très simples qui sont représentés sur la figure 3.5a : tout modèle est composé d'acteurs, les acteurs ont des ports de communication, les ports permettent de connecter des acteurs par des relations et les relations modélisent le fait que les acteurs interagissent.

Dans Ptolemy, un « domaine » définit l'ensemble des règles qui permettent d'interpréter un modèle selon un modèle de calcul donné. Ces règles définissent donc le type de temps, le mode de communication ou encore le mode de synchronisation entre acteurs. Ces règles sont implémentées par un « directeur », qui est à même de diriger l'exécution du modèle afin de réaliser des simulations par exemple. La figure 3.5b présente un exemple de modèle auquel est associé le directeur du domaine SDF (Synchronous DataFlow) représentant le modèle de calcul des flots de données synchrones. Dans ce modèle de calcul, les acteurs représentent des processus qui consomment et produisent des paquets de données de manière synchrone. Ce modèle sera donc simulé par le moteur d'exécution de Ptolemy selon la sémantique des flots de données.

L'avantage majeur de cette approche pour la modélisation multi-paradigme est que la syntaxe abstraite utilisée est unique : c'est donc la même pour tous les modèles, la sémantique d'un modèle étant donnée par le domaine qui lui est associé. Ainsi, deux modèles $M1$ et $M2$ syntaxiquement identiques pourront avoir deux sémantiques différentes s'ils sont associés aux directeurs de deux domaines différents $D1$ et $D2$. Ce principe est illustré par la figure 3.6, qui représente trois fois la même structure de modèle interprétée selon trois modèles de calcul différents. Pour chacun de ces modèles, une représentation graphique intuitive équivalente de l'interprétation

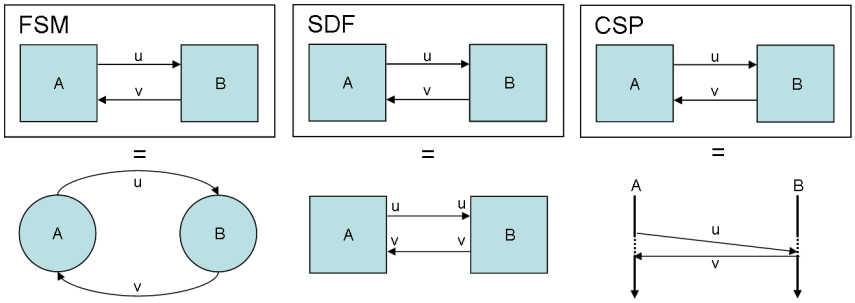


FIG. 3.6 – Interprétation d'un même modèle selon différents modèles de calcul

selon le modèle de calcul est présentée. Le modèle auquel est associé le modèle de calcul FSM (Finite State Machine) est interprété comme un automate à deux états et deux transitions déclenchées par les évènements u et v. Le modèle auquel est associé le modèle de calcul SDF (Synchronous DataFlow) est interprété comme contenant deux processus s'échangeant de façon synchrone u et v données sur leurs deux canaux de communication. Enfin, le modèle auquel est associé le modèle de calcul CSP (Communicating Sequential Processes) est interprété comme contenant deux processus se synchronisant par rendez-vous pour échanger les données u et v.

Dans les approches que nous avons vues dans les sections précédentes, un méta-modèle différent et donc une syntaxe abstraite différente peut être définie pour chaque langage de modélisation différent. Dans le contexte de la modélisation multi-paradigme, la composition de langages ainsi définis devra donc se faire à la fois au niveau de la syntaxe et de la sémantique. Le fait d'avoir une syntaxe abstraite commune pour tous les langages facilite grandement la composition, comme nous l'avons évoqué dans la section 2.5.3. Nous présentons plus en détails l'approche Ptolemy dans la section 3.3.4.4. Nous étudions notamment dans cette section la façon dont des modèles hétérogènes peuvent être composés dans Ptolemy.

3.3.3 Transformations de modèles et composition de méta-modèles

Après avoir présenté différentes manières de spécifier la syntaxe et la sémantique des langages de modélisation, nous proposons de passer en revue différentes techniques qui permettent de combiner dans un même modèle différents langages de modélisation. Dans cette section nous présentons la technique des transformations de modèles et expliquons comment cette technique peut être utilisée dans le cadre de la modélisation multi-paradigme.

3.3.3.1 Principe des transformations de modèles

Le principe des transformations de modèles est illustré par la figure 3.7. Une transformation de modèle est un ensemble de règles qui s'appliquent aux éléments d'un méta-modèle définissant les modèles sources valides (c'est-à-dire auxquels la transformation peut s'appliquer) et qui définissent pour chacun de ces éléments leur(s) équivalent(s) parmi les éléments d'un méta-modèle cible. Le moteur de transformation (aussi appelé « transformateur de modèles ») lit le modèle source, qui doit donc être conforme au méta-modèle source, et applique les règles définies dans la transformation de modèle afin de créer le modèle cible qui sera conforme au méta-modèle cible. Les méta-modèles source et cible peuvent faire partie de la définition de langages de modélisation et, dans ce sens, les transformations de modèles peuvent être considérées comme des systèmes de traduction (ou de réécriture) d'un langage vers un autre.

Le principe général présenté ici pour les transformations de modèles est indépendant de la syntaxe concrète et de la sémantique associées aux méta-modèles source et cible. Concernant les différences de syntaxe concrète, il est à noter que les transformations modèle-à-modèle et modèle-à-texte sont souvent distinguées l'une de l'autre dans la littérature.

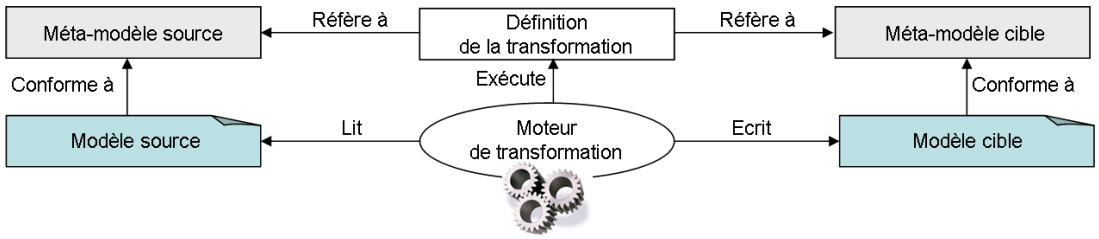


FIG. 3.7 – Principe des transformations de modèles

Les transformations de modèle peuvent s'appuyer sur différentes techniques sous-jacentes telles que les « templates » (sortes de gabarits), les transformations de graphes, les grammaires de graphes triples [Sch95], etc. Les techniques à base de graphes telles que les grammaires de graphes ou la réécriture de graphes sont notamment très utilisées. Une étude comparative de ces techniques est proposée dans [EGdL+05].

Les transformations de modèles sont à la base de nombreuses approches au coeur de l'approche MDA (voir la section 2.4.1). Certaines de ces approches offrent des avantages comme l'automatisation du mécanisme d'abstraction/raffinement (par transformation entre langages de niveaux d'abstraction différents par exemple), l'ajout automatique d'information dans un modèle par un mécanisme dit de « tissage » de modèles (ce mécanisme n'est pas formellement défini mais est évoqué notamment dans [Béz05]), ou l'obtention automatique d'une sémantique formelle pour un langage via la définition d'une transformation formellement définie établissant une correspondance entre ce langage et un langage lui-même formellement défini [KASS03].

Dans le contexte d'un ensemble de modèles hétérogènes concernant un même système, les transformations de modèles peuvent être envisagées dans deux types différents de solutions de modélisation multi-paradigme que nous présentons dans les deux sections suivantes.

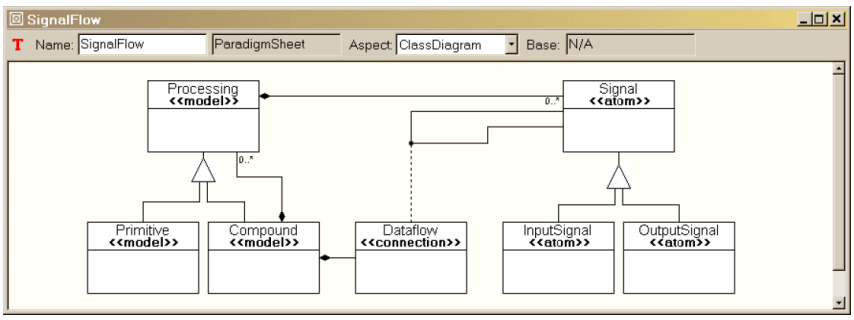
3.3.3.2 Transformation vers un méta-modèle tiers

Dans une première solution, un méta-modèle unique est choisi afin d'uniformiser les modèles hétérogènes par transformation vers ce méta-modèle. Ce méta-modèle cible peut être choisi parmi les méta-modèles auxquels sont conformes les modèles à intégrer ou être un méta-modèle tiers sans rapport avec les modèles à intégrer. Une fois transformés pour être tous conformes à ce méta-modèle, les modèles peuvent être réunis pour former un modèle unique et homogène du système.

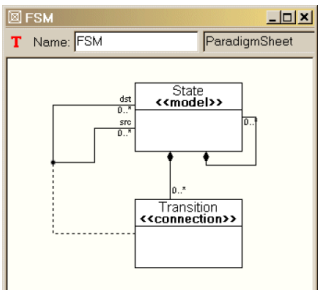
Ce type d'approche est défendu par J. De Lara et H. Vangheluwe dans [dLV02] avec l'utilisation de l'outil de transformation de modèle $ATOM^3$. L'avantage de cette méthode est que la prise en compte d'un langage additionnel est aisée et n'implique pas de modifier l'existant. De plus elle permet de choisir le langage cible de manière adaptée par rapport à un objectif de modélisation. Par contre, le nombre de transformations possibles à spécifier est très grand et croit exponentiellement avec le nombre de langages à supporter. En outre, la réunion des modèles une fois transformés n'est pas traitée par cette solution et reste donc manuelle.

3.3.3.3 Transformation vers un méta-modèle composé

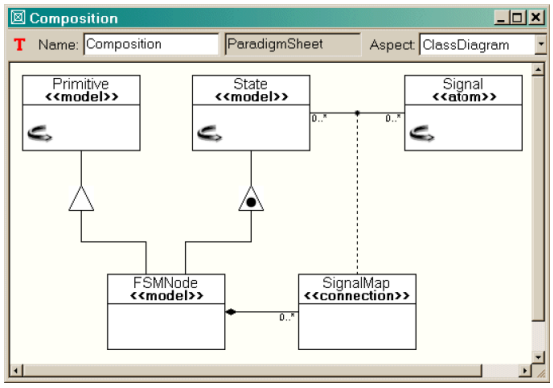
Le principe de ce deuxième type de solution est identique à celui évoqué dans la section précédente mais le méta-modèle cible des transformations est construit par composition des méta-modèles auxquels sont conformes les modèles hétérogènes originaux. L'idée est de disposer dans le méta-modèle cible de tous les éléments nécessaires pour exprimer tous les modèles hétérogènes.



(a) Méta-modèle d'un langage à base de flots de données



(b) Méta-modèle d'un langage à base de machines à états finis



(c) Composition des méta-modèles

FIG. 3.8 – Composition de méta-modèles avec GME (source : [LNK+01])

Différentes techniques permettent de composer des méta-modèles. Dans [ES06], M. Emerson et J. Sztipanovits présentent une étude de plusieurs de ces techniques incluant entre autres la fusion de méta-modèles, l'interfaçage de méta-modèles ou encore le raffinement de classes. Dans [LNK+01] par exemple, une extension d'UML est proposée afin de permettre la composition de méta-modèles exprimés en UML et en OCL. Différents opérateurs de composition sont proposés afin de permettre, par exemple, l'expression de la notion d'équivalences entre classes. L'approche proposée est implémentée dans l'outil GME (Generic Modeling Environment [Dav03]). La figure 3.8 illustre la composition de deux méta-modèles dans GME en utilisant cette approche. La figure 3.8a représente le méta-modèle d'un langage à base de flots de données hiérarchique dans lequel les éléments composant un modèle sont des unités de calcul qui traitent et produisent des signaux. Les unités de calcul peuvent être primitives, c'est-à-dire atomiques, ou composées c'est-à-dire contenir d'autres unités de calcul, elles-mêmes primitives ou composées. La figure 3.8b représente le méta-modèle d'un langage à base de machines à états finis hiérarchiques très simple. La figure 3.8c montre comment ces deux méta-modèles peuvent être composés dans l'objectif d'obtenir un langage dans lequel une unité de calcul à flots de données peut contenir une machine à états finis (cela est représenté par l'élément « FSMNode ») mais dans lequel un état ne peut pas contenir de machine à états finis ou d'unité de calcul à flots de données. Plus de détail sur les mécanismes de composition utilisés et la notation graphique employée peuvent être obtenus dans [LNK+01].

Il est important de noter que les techniques de composition de méta-modèles évoquées ici sont conçues pour permettre la composition syntaxique, c'est-à-dire la composition des syntaxes

définies par les méta-modèles, mais ne tiennent pas compte des sémantiques éventuellement sous-jacentes (qui peuvent être définies en utilisant les techniques vues dans la section 3.3.2 par exemple). Le résultat d'une telle composition de méta-modèles au niveau de la sémantique n'est donc pas garanti et dépend de l'expertise de la personne réalisant la composition. D'autres techniques permettent cependant de réaliser la composition de langages directement au niveau des sémantiques. Nous passons ces techniques en revue dans la section 3.3.4.

L'avantage de cette solution de transformation vers un méta-modèle composé est qu'elle minimise la perte d'information lors de la transformation des modèles hétérogènes puisque le méta-modèle cible contient tous les méta-modèle sources. Cependant nous avons vu que le résultat de la composition au niveau de la sémantique n'était pas évident. De plus, cette solution n'est pas satisfaisante en termes de flexibilité et d'évolutivité car, dès qu'un langage additionnel est à prendre en compte, il est nécessaire de modifier le langage « union »ainsi que potentiellement toutes les transformations des langages source au langage cible. En outre, elle ne permet pas de traiter le problème de la réunion des modèles une fois transformés, en effet, il faut alors les « recoller » de manière à ce qu'ils forment un modèle unique. Cette opération doit être faite manuellement après transformation.

Nous citons ici sans les détailler les travaux présentés dans [CM08]. Ceux-ci présentent une approche intermédiaire entre la transformation vers un langage tiers et la transformation vers un langage composé. Cette approche formelle s'appuie sur la notion de colimite et sur un graphe de logiques pour permettre la réécriture de modèles décrits dans des logiques différentes dans une logique tierce choisie automatiquement grâce au calcul de colimite. Cette approche dépasse le niveau syntaxique puisque les éléments manipulés sont des logiques, ce qui permet de travailler directement au niveau sémantique.

3.3.4 Composition de sémantiques

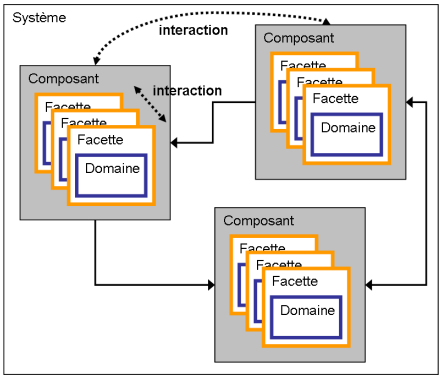
Nous avons vu dans la section précédente différentes techniques de modélisation multi-paradigme à base de transformations de modèles. Nous avons également vu qu'il était possible de composer syntaxiquement différents langages de modélisation en utilisant des techniques de composition de méta-modèles. Dans cette section, nous nous intéressons à la façon dont les sémantiques de langages de modélisation peuvent être composées afin de pouvoir former des modèles hétérogènes dont la sémantique est bien définie.

3.3.4.1 Composition d'algèbres

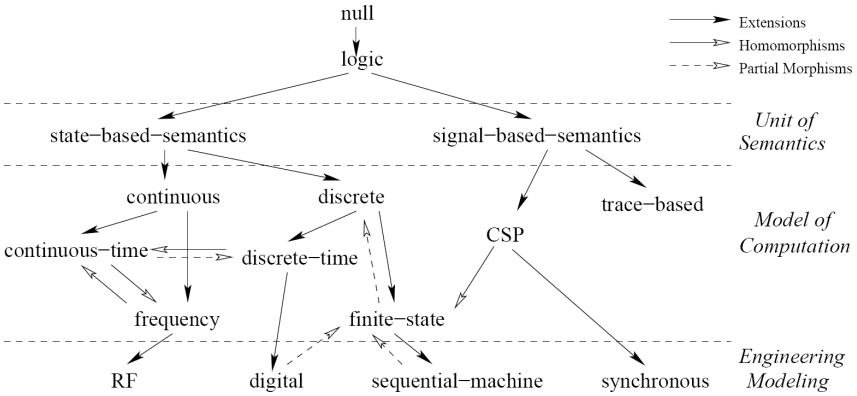
Nous abordons tout d'abord les approches dans lesquelles la sémantique des langages de modélisation est exprimée grâce à des algèbres. La composition des sémantiques est alors réalisée grâces aux opérations de composition sur les algèbres. Nous ne détaillons pas dans ce mémoire la notion d'algèbre ni les opérations qu'il est possible de réaliser sur des algèbres.

Metropolis [BLP+02] s'appuie sur les algèbres de structures de traces [BPSV01] pour exprimer la sémantique de modèles hétérogènes. Les algèbres de structures de trace, de même que les Tag Machines [BCCSV05], ont été inspirées du travail de E. A. Lee et A. Sangiovanni-Vincentelli sur le langage Tagged Signal Model [LSV98] permettant de comparer des modèles de calcul (voir la section 3.3.2.3). Dans Metropolis, une trace correspond au comportement observable d'un « agent », c'est-à-dire d'un processus, d'un acteur, d'un module, etc. Une structure de trace définit le modèle d'un agent et comprend l'ensemble des traces possibles de cet agent. Les algèbres de structures de traces sont utilisées pour définir comment les modèles de différents agents peuvent être composés pour former un modèle global.

Metropolis semble restreint aux langages à base de processus mais il fournit un ensemble assez complet de méthodes de modélisation et d'outils support. En particulier, la modélisation fonctionnelle peut être séparée de la modélisation de l'architecture et un mécanisme de corres-



(a) Composition de facettes



(b) Treillis des domaines (source : [KA03])

FIG. 3.9 – Domaines et facettes avec Rosetta

pondance est fournit pour permettre de générer des modèles spécifiques à une plate-forme. Des outils de vérification, de simulation et de synthèse sont également fournis.

3.3.4.2 Composition de coalgèbres

Dans cette section nous abordons les approches dans lesquelles la sémantique des langages de modélisation est exprimée grâce à des coalgèbres [Jac]. La notion de coalgèbre est duale de la notion d'algèbre. Ainsi, alors que les algèbres sont utilisées pour décrire comment des opérations permettent de construire des objets, les coalgèbres sont utilisées pour décrire des observations sur des objets. Tout comme il est possible de composer des algèbres, il est également possible de composer des coalgèbres avec les opérations duales de celles utilisées pour les algèbres.

Rosetta [KA03, SA06] est un langage de spécification de niveau système avec une sémantique formelle basée sur les coalgèbres. La notion de « facette » est centrale dans Rosetta. Une facette modélise un aspect observable ou une vue particulière d'un composant, comme par exemple son comportement fonctionnel observable ou sa consommation d'énergie. Rosetta permet de combiner des facettes, soit pour assembler des modèles de composants, soit pour combiner différents aspects d'un même composant. Ainsi, Rosetta permet aux concepteurs de combiner facilement différentes vues de parties d'un système. La figure 3.9a illustre les deux types de composition de facettes dans le modèle d'un système. Les flèches en pointillés montrent que les interactions entre facettes se font à la fois entre différentes vues d'un composant mais également entre composants.

Dans Rosetta, les « domaines », qui sont des facettes spéciales, encapsulent le vocabulaire et

la sémantique spécifique à un domaine métier pour permettre de réaliser des modèles domaine-spécifiques. Les différents domaines supportés par Rosetta sont reliés les uns aux autres par des relations, définissant ainsi un treillis de domaines, représenté sur la figure 3.9b. Une facette représentant un composant est définie par extension d'un domaine (cela est illustré sur la figure 3.9a).

La sémantique d'une facette est dénotée par une coalgèbre définissant des observations sur les changements d'état abstrait de la facette. Ce concept est la clé de l'approche Rosetta : puisque l'état d'une facette n'est donné que par une observation sur un état abstrait, il est possible de définir de multiples observations de cet état avec de multiples sémantiques et celles-ci peuvent être reliées les unes aux autres. Des opérations de composition ainsi que des opérations de transformation sont définies sur les facettes et s'appuient sur le treillis des domaines. Rosetta fournit donc un support formel pour la modélisation multi-paradigme qui permet de définir la sémantique d'un modèle obtenu par composition et/ou transformation.

3.3.4.3 Composition de Semantic Units

Nous avons présenté le concept de Semantic Unit [CSN+05] dans la section 3.3.2.2. Les Semantic Units (SUs) permettent d'attacher une sémantique formelle à un méta-modèle représentant la syntaxe abstraite d'un langage. Une SU comporte un modèle de donnée abstrait et un ensemble d'opérations d'exécution, tous deux définis en AsmL. Des mappings permettent d'établir des correspondances entre les éléments du modèle de données abstrait de la SU et les éléments du méta-modèle considéré.

Dans [CSN07], les auteurs proposent tout d'abord la notion de SU « primaire », qui capture les éléments sémantiques de base d'une catégorie particulière de langages. En ce sens, une SU primaire est similaire à un modèle de calcul tel que défini dans la section 3.3.2.3. Les auteurs proposent ensuite des mécanismes permettant de composer différentes Semantic Units afin de former des SUs plus complexes et ainsi pouvoir définir la sémantique de modèles hétérogènes. Une SU obtenue par composition de SUs primaires est appelée SU dérivée. Cette SU dérivée peut elle-même être composée avec une autre SU pour former une SU plus complexe.

Prenons l'exemple de deux SUs primaires que l'on souhaite composer pour définir la sémantique d'un modèle hétérogène. La composition se fait en deux étapes : (1) composition structurelle et (2) composition comportementale. La composition structurelle est réalisée en définissant un ensemble de relations de correspondances entre les modèles de données abstraits des SUs. Ces relations sont exprimées sous la forme de tables de correspondance en AsmL. Le modèle de données abstrait de la SU résultante comprend alors les modèles de données des deux SUs primaires et les correspondances entre leurs éléments. La composition comportementale est réalisée en définissant un ensemble d'opérations d'exécution additionnelles qui forment, avec les opérations des SUs primaires, l'ensemble des opérations d'exécution de la SU composée. En particulier, ces opérations additionnelles définissent comment interagissent les opérations d'exécution des SUs primaires, et orchestrent ces interactions. Elles utilisent pour cela les relations de correspondance entre les modèles de données abstraits des SUs primaires.

L'approche de composition de Semantic Units comporte donc à la fois une composition sémantique et une composition syntaxique. Elle permet une définition flexible et modulaire de compositions de langages sur la base d'unités élémentaires bien définies.

3.3.4.4 « Agrégation » de modèles et composition de Modèles de Calcul

Comme nous l'avons introduit dans la section 3.3.2.3, l'approche implémentée dans Ptolemy II [EJL+03] repose sur le concept de *modèle de calcul*. Dans Ptolemy, les modèles sont décrits en utilisant une syntaxe abstraite unique à base d'acteurs, à laquelle un domaine est associé afin de lui donner une sémantique. Un domaine définit des règles pour interpréter les

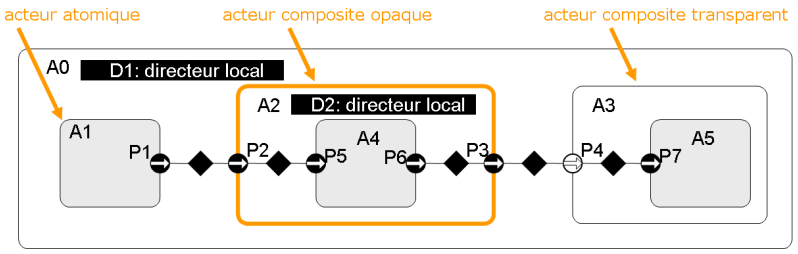
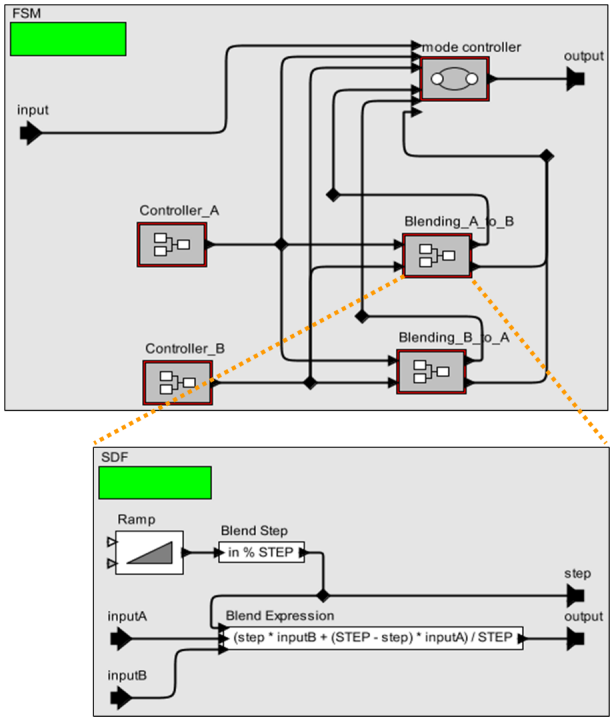
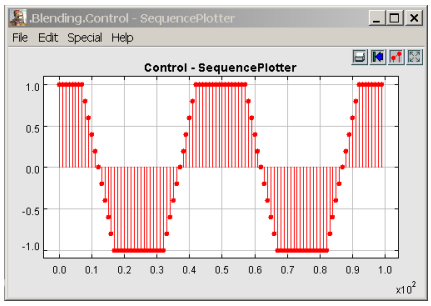


FIG. 3.10 – Acteurs composites opaques et transparents



(a) Combinaison hiérarchique des modèles de calcul FSM et SDF pour le contrôle d'un véhicule aérien non habité intelligent



(b) Adoucissement des transitions entre modes de contrôle

FIG. 3.11 – Combinaison hiérarchique de modèles de calcul avec Ptolemy (source : PtolemyII)

relations entre les acteurs selon un modèle de calcul donné. Ces règles sont implémentées par un directeur.

Ptolemy permet de combiner hiérarchiquement des modèles de calcul différents dans un même modèle. En effet, (a) un acteur faisant partie d'un modèle interprété selon un domaine $D1$ peut contenir lui-même d'autres acteurs (l'acteur est alors dit « composite ») et (b) le domaine $D2$ selon lequel sont interprétés les acteurs « internes » de l'acteur composite peut être différent du domaine $D1$ (l'acteur composite est alors dit « opaque »). Ce principe est illustré par la figure 3.10 : l'acteur $A2$ est un acteur composite opaque dont la structure interne est interprétée par un directeur $D2$ qui peut être différent du directeur $D1$. Inversement, l'acteur $E3$ est un acteur composite dit « transparent » car sa structure interne est interprétée par le directeur $D1$. Grâce à cette architecture hiérarchique dans laquelle chaque niveau d'un modèle peut faire intervenir un modèle de calcul différent, les modèles de calcul (et donc les langages de modélisation) peuvent être combinés par paire à la frontière entre deux niveaux hiérarchiques.

La figure 3.11a montre un exemple dans lequel les modèles de calcul Finite State Machine

(FSM, modèle de calcul des machines à états finis) et Synchronous DataFlow (SDF, modèle de calcul des flots de données synchrones) sont combinés dans un même modèle. Cet exemple illustre les travaux menées dans le cadre d'un projet de recherche sur l'amélioration de l'architecture de contrôle de véhicules aériens non habités intelligents [HPW01]. Les systèmes logiciels de contrôle de ce type de véhicules doivent pouvoir supporter l'ajout à la volée d'algorithmes de contrôle, la reconfiguration rapide et en ligne des algorithmes ainsi que le calcul temps-réel distribué. L'approche étudiée dans le projet est basée sur une architecture hiérarchique de contrôle dans laquelle la gestion de mission et la connaissance de la situation du véhicule sont gérées au plus haut niveau, le contrôle de vol est géré au plus bas niveau et un contrôleur de niveau intermédiaire coordonne les transitions entre les différents modes et assure la tolérance aux fautes. Sur notre exemple, c'est ce dernier contrôleur qui est représenté par le composant « mode controller » du modèle FSM sur la figure 3.11a. Ce contrôleur est chargé de piloter les changements de modes de fonctionnement du véhicule, qui a ici deux modes A et B. Dans chaque mode, le contrôle est alors réalisé respectivement par les contrôleurs « Controller_A » et « Controller_B ». Le problème de cette architecture est que la transition brutale entre les deux modes et le passage du contrôle de l'un des contrôleurs à l'autre peut provoquer des comportements non souhaités du système. La figure 3.11a illustre une technique utilisée pour adoucir les transitions entre modes à l'aide de composants qui prennent le relais des contrôleurs lors de la transition entre deux modes et effectuent une transition progressive entre les deux modes. Le signal adouci obtenu grâce à ce système est montré sur la figure 3.11b. Sans adoucissement, ce signal aurait été un signal de type créneau pur.

L'utilisation du modèle de calcul FSM (machines à états finis) est intuitive pour représenter le pilotage des changements de contrôleur par le contrôleur maître, car le concepteur manipule ici les notions de modes discrets et de transitions entre modes. De même, l'utilisation du modèle de calcul SDF (flots de données synchrones) pour représenter le comportement des composants chargés d'adoucir les transitions est également intuitive car le concepteur manipule alors les notions de signaux et de composants de traitement du signal. Cet exemple illustre ainsi l'intérêt de pouvoir utiliser conjointement dans un même modèle différents modèles de calcul.

L'avantage majeur de Ptolemy est qu'il permet « d'agréger » des modèles faisant intervenir des modèles de calcul différents grâce au mécanisme d'abstraction hiérarchique. Son principal inconvénient est que la façon dont deux modèles de calcul sont combinés lorsqu'ils sont en contact à la frontière entre deux niveaux d'un modèle est prédéfinie et codée en dur dans le noyau de la plate-forme d'exécution de Ptolemy. Ainsi, Ptolemy ne supporte qu'une façon d'embarquer un modèle SDF dans un modèle FSM. Même si ces combinaisons prédéfinies sont issues de l'étude détaillée des modèles de calcul et de leurs interactions, le fait qu'elles soient prédéfinies et implicites pose différents problèmes pour le concepteur, que nous détaillons dans la section 4.2. L'une de nos contributions dans cette thèse vise à pallier ce problème. Nous détaillons la solution proposée dans la partie II de ce mémoire.

Les travaux précurseurs de l'université de Berkeley sur les modèles de calcul et Ptolemy ont montré l'intérêt du concept de modèle de calcul, notamment pour la composition de modèles hétérogènes. D'autres approches ont été développées en s'appuyant sur ce concept. L'étude de la combinaison de différents modèles de calcul dans des modèles non hiérarchiques a notamment été étudiée dans [Mbo04] tandis que l'étude de composants pouvant obéir conjointement à deux modèles de calcul a été étudiée dans [Fer05].

L'approche « 42 » [MB07], qui repose aussi sur le concept de modèle de calcul, s'appuie sur le paradigme synchrone pour la simulation de modèles impliquant différents modèles de calcul. Dans 42, les modèles de calcul sont représentés par des « contrôleurs ». La spécificité de 42 est que le code d'un contrôleur est généré automatiquement pour chaque modèle à partir des contrats comportementaux des composants du modèle (exprimés avec des automates), des relations entre les ports des composants et d'informations additionnelles liées au type d'ordonnancement que

doit effectuer le contrôleur. La description du contrat comportemental de chaque composant est la clé de cette approche. Cependant, une telle description peut ne pas être disponible (dans le cas d'un composant fourni par un tiers par exemple) ou peut ne pas être facile à établir (dans le cas d'un composant avec un comportement continu dans le temps par exemple). Ainsi l'approche développée dans 42 semble moins générale que celle implémentée dans Ptolemy dans le sens où la gamme des modèles de calcul supportés est limitée aux modèles de calcul à sémantique discrète (ou discrétisable facilement). Cependant elle a l'avantage d'être plus automatisée.

3.3.4.5 Sémantique commune et variations sémantiques

Plutôt que de permettre l'expression libre des sémantiques des langages de modélisation et d'utiliser ensuite des opérations de composition afin de « recoller les morceaux », une autre approche consiste à définir une base sémantique commune pour l'ensemble des langages que l'on souhaite supporter et à permettre l'expression de variations sémantiques pour refléter les particularités des langages par rapport à cette base commune.

Ainsi, dans [PY96], les auteurs proposent une approche pour la modélisation multi-formalisme qui est dédiée à l'analyse d'espace d'états. Cette approche repose sur un « infra-modèle » qui capture les caractérisques sémantiques qui sont essentielles pour qu'un formalisme permette l'exploration d'espace d'états. Une étape de personnalisation permet de prendre en compte les spécificités sémantiques de chaque formalisme. Dans cette approche, les hypergraphes sont utilisés pour représenter les modèles. Des règles de transformation d'hypergraphes permettent de spécifier comment un modèle est affecté lorsqu'il est exécuté selon la sémantique d'un formalisme particulier. En ce sens, les règles de transformation d'hypergraphes embarquent la sémantique des formalismes. Cette approche permet de réaliser différents types d'analyses sur les modèles, depuis l'analyse de l'atteignabilité d'un état jusqu'à l'exécution des modèles. Cependant, il semble que la gamme des formalismes supportés par cette approche soit limitée aux formalismes à base de transitions pour les systèmes concurrents.

De manière générale, comme ce type d'approche oblige à définir une base sémantique commune, les langages supportés ont nécessairement des similitudes (ils doivent notamment être basés sur des modèles de calcul similaires).

3.3.5 Approches à bases de composants

Dans le domaine du logiciel orienté composant ou orienté service, un objectif important est d'être capable de réutiliser des composants ou des services développés par des tiers. Ceux-ci sont potentiellement hétérogènes, leurs sources ne sont généralement pas accessibles ni modifiables et leurs modes d'interaction peuvent également être hétérogènes. Lors de l'intégration de tels composants, des incompatibilités ou des discordances peuvent apparaître entre les signatures de leurs interfaces, entre leurs protocoles, entre leurs modes d'interaction ou encore entre leurs exigences en termes de qualité de service. Dans cette section nous présentons différentes techniques de modélisation développées pour résoudre de tels problèmes. Même si ces techniques ne peuvent pas être directement considérées comme des techniques de modélisation multi-paradigme, la plupart d'entre elles apportent des concepts et des solutions intéressantes pour la composition de modèles hétérogènes.

3.3.5.1 Terminologie : compatibilité et composabilité

Prenons deux composants, Composant1 et Composant2 tels que ceux représentés sur la figure 3.12. Leurs interfaces sont représentées sur la figure par des petits carrés symbolisant leurs entrées et leurs sorties. Nous ne nous intéressons pas ici au type de ces entrées et de ces sorties ni au mode de communication ou de synchronisation de ces composants. Le trait tiré entre

les deux composants symbolise simplement le fait que ceux-ci interagissent par l'intermédiaire de certaines de leurs entrées et sorties.

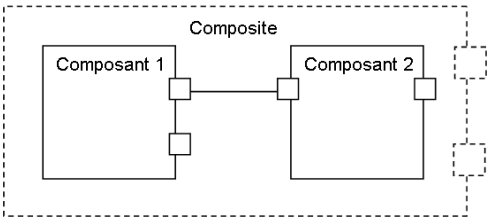


FIG. 3.12 – Composants en interaction et composition de composants

Si l'*interaction* entre les deux composants peut être considérée comme possible étant données les informations disponibles sur les contraintes de fonctionnement respectives des composants (protocole, mode de communication, comportement, qualité de service et autres contraintes non fonctionnelles) alors ils peuvent être considérés comme compatibles. La *compatibilité* de deux composants dépend donc uniquement des propriétés individuelles de chacun d'eux, sans préjuger du résultat qu'aura l'interaction de ces deux composants. La compatibilité peut être évaluée en utilisant des informations sur l'interface des composants, sur leur mode de communication, sur leur comportement, etc.

Si le comportement résultant de l'interaction entre les composants Composant1 et Composant2 peut être encapsulé pour former un nouveau composant (qui est généralement appelé « composite ») et que les propriétés individuelles des composants sont préservées (donc si l'interaction ne modifie pas les propriétés individuelles des composants) alors ces composants peuvent être considérés comme composables. La non préservation des propriétés individuelles des composants peut mener à des comportements aberrants, incohérents, imprévisibles ou encore non souhaités du composite résultant. La *composabilité* est donc profondément différente de la compatibilité car elle dépend non seulement des propriétés individuelles des composants mais également des propriétés résultant de l'interaction entre les composants. En ce sens, le problème de la composabilité est un problème particulièrement difficile et important.

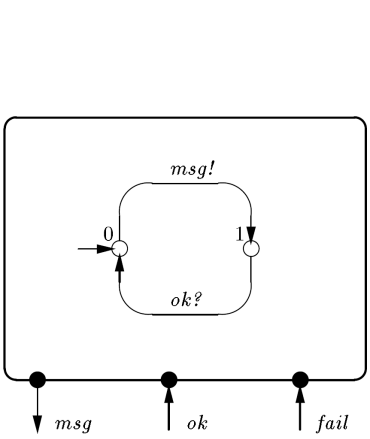
3.3.5.2 Vérification de compatibilité et de composabilité

La vérification a priori de la compatibilité entre composants s'appuie sur la description de leur comportement ou de leur interface. Ce problème a été abordé notamment dans des travaux basés sur des spécifications à base de contraintes [Jac00] ou à base d'automates [AG94] représentant le comportement des composants. Nous présentons ci-dessous deux techniques permettant de vérifier la compatibilité et la composabilité de composants.

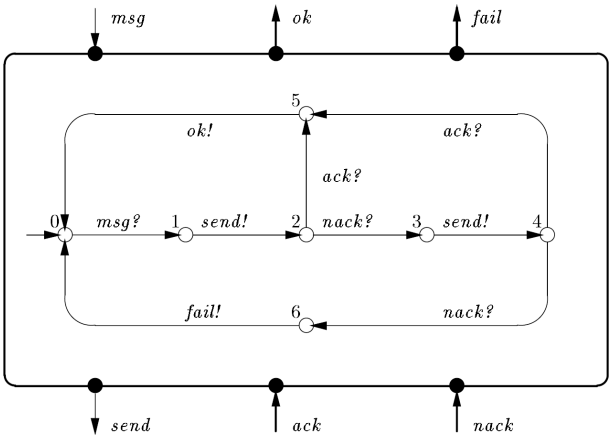
3.3.5.2.a Automates d'interface

Dans [dAH01a], des « automates d'interface » sont utilisés afin de capturer l'enchaînement dans le temps des entrées/sorties des composants. Ceux-ci formalisent en fait les hypothèses faites sur l'ordre dans lequel les méthodes du composant doivent être appelées et sur l'ordre dans lequel le composant appelle des méthodes externes (c'est-à-dire des méthodes fournies par d'autres composants). Les automates d'interface interagissent via la synchronisation des actions entrantes et sortantes. Ils permettent de vérifier automatiquement si deux composants sont compatibles : si le résultat de la composition des automates d'interface représentant les deux composants est vide, alors les composants ne sont pas compatibles entre eux.

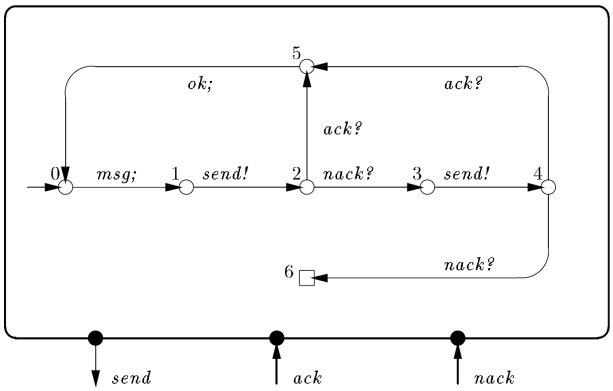
La figure 3.13 illustre l'utilisation d'automates d'interface afin de vérifier la compatibilité entre un composant logiciel « Comp » implémentant un service de transmission de messages et un composant logiciel « User » qui utilise ce composant de transmission pour envoyer des



(a) Automate d'interface du composant « User »



(b) Automate d'interface du composant de transmission de messages « Comp »



(c) Composition des automates User ⊗ Comp

Fig. 3.13 – Utilisation des automates d'interface pour la vérification de compatibilité (source : [dAH01a])

messages. L'automate d'interface du composant « Comp » est représenté par la figure 3.13b. Ce composant de transmission dialogue avec un canal de communication dont l'interface fournit la méthode « send » pour envoyer un message. Les deux valeurs de retour possibles sont « ack », qui indique une transmission réussie du message et « nack » qui indique que la transmission a échoué. Le composant de transmission effectue au moins deux tentatives de transmission du message « msg » qu'il reçoit. Si lors de la première tentative il reçoit un « nack », il tente de transmettre de nouveau le message. Si les deux tentatives échouent, il rend compte de l'échec de la transmission en retournant « fail », sinon il rend compte du succès en retournant « ok ». L'une des hypothèses faites à propos de l'environnement du composant « Comp » (hypothèse qui sont donc représentées par l'automate d'interface du composant) est que cet environnement attendra d'avoir reçu un « ok » ou un « fail » avant d'envoyer un nouveau « msg ». En effet, l'entrée « msg » n'est acceptée que sur la transition partant de l'état numéro 0 de l'automate.

Le comportement du composant « User » qui utilise le composant de transmission « Comp » est relativement simple : il tente d'envoyer un message et n'envoie d'autre message que s'il reçoit une réponse « ok » confirmant que son message a bien été envoyé. Ce composant a la particularité de ne pas tenir compte de la possibilité d'échec de la transmission. L'automate d'interface du composant « User » montré par la figure 3.13a n'acceptant pas d'entrée « fail » exprime bien

l'hypothèse faite à propos de l'environnement de ce composant qui est que la transmission d'un message ne peut pas échouer.

Le résultat de la composition des deux automates d'interface des composants « User » et « Comp » est montré par la figure 3.13c. Nous ne détaillons pas ici la méthode permettant de calculer cette composition, qui est exposée dans [dAH01a]. Le résultat de la composition n'étant pas vide, il est possible de déduire par cette méthode que les deux composants « User » et « Comp » sont compatibles. Cependant le résultat de la composition montre un état dit illégal : l'état numéro 6, représenté par un carré sur la figure 3.13c. Cet état est appelé illégal car il n'est atteint que si l'environnement ne respecte pas les hypothèses de son bon fonctionnement (= la transmission d'un message ne peut pas échouer). Les deux composants ne sont donc en fait pas totalement compatibles. Des détails sont donnés quant à la gestion des états illégaux dans la composition des automates d'interface dans [dAH01a].

3.3.5.2.b Théories des interfaces

Dans [dAH01b], les interfaces des composants sont définies par L. De Alfaro et T. Henzinger comme des ensembles de contraintes qui s'appliquent à l'environnement du composant afin de garantir son comportement. Ainsi, les interfaces expriment les hypothèses faites par le concepteur du composant sur l'environnement dans lequel le composant peut être déployé, tandis que le modèle du composant spécifie comment le composant se comporte dans un environnement quelconque. Ces définitions sont utilisées pour définir des théories d'interfaces sur la base d'algèbres d'interfaces, d'algèbres de composants et de relations d'implémentations. Avec des théories d'interfaces, il est possible de manipuler des concepts tels que l'implémentation, le raffinement, la composition et la décomposition dans le contexte de la conception et de la vérification basée sur les composants. Ainsi par exemple, si l'on décompose une interface en interfaces parcellaires, et que l'on construit des composants qui implémentent ces interfaces, alors la théorie d'interface associée garantit que les composants peuvent être connectés et composés pour former le système qui implémente l'interface initiale. Ce principe est illustré par la figure 3.14 sur laquelle le composant F est raffiné en une interconnexion de composants $F1$, $F2$ et $F3$.

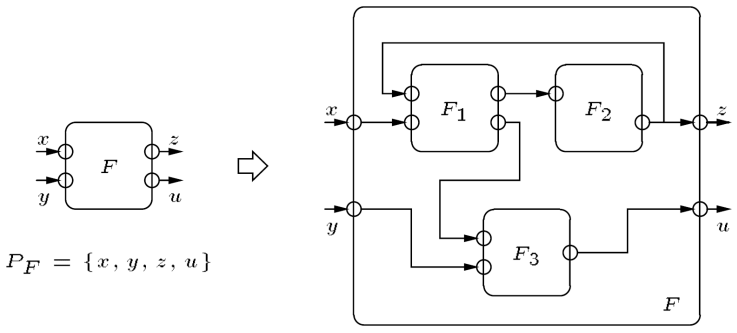


Fig. 3.14 – Raffinement d'un composant avec une théorie des interfaces (source : [dAH01b])

Les automates d'interfaces et les théories d'interfaces fournissent des mécanismes généraux et puissants pour la vérification de compatibilité et de composabilité dans le cadre de la conception basée sur des composants. Les automates d'interfaces ont été notamment appliqués à la vérification de compatibilité entre les composants et les directeurs (domaines) de Ptolemy II [LX04]. Ce travail montre comment ces concepts peuvent être appliqués à la composition de modèles hétérogènes lorsque celle-ci est basée sur le concept de modèle de calcul.

3.3.5.3 Adaptation logicielle (adaptation de composants)

L'adaptation logicielle [MA03, YS97] est une approche qui vise à corriger le comportement de composants non totalement compatibles de manière à permettre leur interaction. Deux principes

de correction différents sont généralement cités : la génération (aussi automatique que possible) d'adaptateurs statiques d'une part, et la correction dynamique à l'exécution d'autre part. La figure 3.15 illustre le principe des adaptateurs sur deux composants.

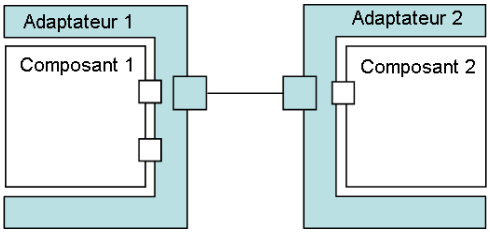


Fig. 3.15 – Composants et adaptateurs

La génération d'adaptateurs statiques est un sujet qui a déjà été largement abordé. Des définitions formelles [YS97], des méthodologies formelles [BBC05] ainsi que des méthodes pour obtenir des propriétés telles que la vivacité ou la sûreté [IMTA05] existent. Cependant, le coût en calcul de la plupart de ces approches est généralement élevé et la taille des adaptateurs générés importante. Des solutions pour permettre la génération incrémentale des adaptateurs ou leur modification à la volée (pour supprimer des interactions incorrectes ou réduire les comportements possibles) sont envisagées dans [MPS07].

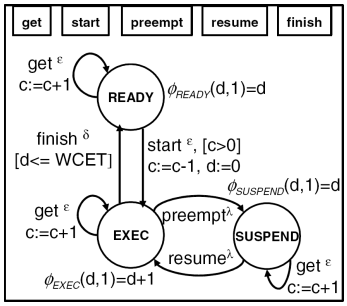
3.3.5.4 Composants avec interactions hétérogènes

Le principal avantage des approches orientées composants est que les aspects calcul et communication sont découplés [BPSV01]. La description de la partie calcul est divisée en sous-parties qui correspondent aux modèles des composants, tandis que la partie communication est décrite par un ou plusieurs modèles d'interaction. De nombreux types d'interaction existent, comme par exemple le passage de message synchrone ou asynchrone (utilisés dans les modèles de composants usuels tels que le CORBA Component Model [OMGc]) ou les variables partagées. L'interaction entre deux composants est généralement représentée graphiquement dans un modèle par un lien ou une connexion entre ces deux composants. Associer une sémantique aux liens dans un modèle peut se faire soit de manière globale en définissant une sémantique unique qui s'applique à tous les liens, soit lien par lien avec la possibilité de mélanger dans un même modèle différents types d'interactions. La première méthode est similaire à l'utilisation d'un modèle de calcul (le modèle de calcul définit une sémantique pour l'ensemble des liens, voir la section 3.3.4.4) tandis que la deuxième méthode définit des interactions hétérogènes (les sémantiques de deux liens sont indépendantes et peuvent être différentes).

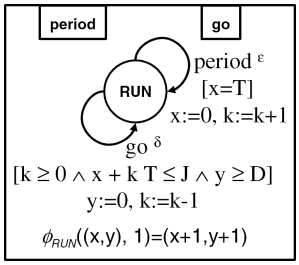
3.3.5.4.a BIP (Behavior, Interaction, Priority)

BIP (Behavior, Interaction, Priority) [BBS06, GS05] fournit des mécanismes formels pour décrire des combinaisons de composants avec des interactions hétérogènes. Dans BIP, les composants sont décrits par trois couches : une couche définissant le comportement du composant sous la forme d'un système de transitions, une couche définissant les connecteurs du composants (c'est-à-dire la façon dont il est possible d'interagir avec lui) et une troisième couche qui définit des priorités entre les interactions possibles. Dans un modèle BIP, deux liens entre deux composants peuvent dénoter des types d'interaction différents, en particulier en fonction de la nature des connecteurs. Les composants peuvent être assemblés pour former de nouveaux composants dits « composites ».

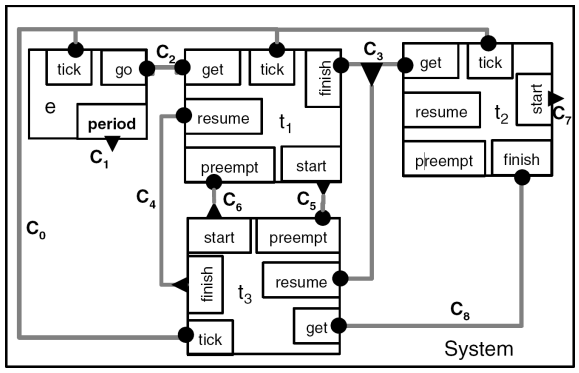
La figure 3.16 illustre l'utilisation de BIP sur un système composé de trois tâches t1, t2 et t3 mises en série et s'exécutant sur des processeurs indépendants. Une tâche ne peut s'exécuter que lorsque la tâche précédente a terminé. La figure 3.16a représente le modèle d'une tâche.



(a) Modèle d'une tâche



(b) Modèle d'un générateur d'évènements



(c) Modèle du système

FIG. 3.16 – Modélisation d'un système multi-processeur avec BIP (source : [BBS06])

Les évènements traités par les tâches sont créés par un générateur d'événements dont le modèle est représenté par la figure 3.16b. Enfin la figure 3.16c montre le système complet, qui est un composite réalisé par l'assemblage de trois modèles de tâche et d'un modèle de générateur d'événements. Les priorités sont omises dans la représentation graphique, qui ne montre que le comportement et les interactions. Sur cette représentation, les connecteurs représentés par des cercles pleins dénotent une interaction dite « incomplète » (l'interaction ne peut avoir lieu que si au moins un deuxième composant y prend part) tandis que les connecteurs représentés par des triangles pleins dénotent une interaction dite « complète » (l'interaction se fait à l'initiative du composant comportant le connecteur et aura lieu même si aucun autre composant n'y prend part). Une relation impliquant uniquement des interactions incomplètes représente la notion de rendez-vous (synchronisation forte) tandis que si l'une des interactions impliquées est complète alors la relation représente la notion de broadcast (synchronisation faible). Sur cet exemple, les deux types d'interaction sont utilisés conjointement. Dans ce modèle, les composants sont donc utilisés avec des interactions hétérogènes.

BIP offre un support pour l'exécution de modèles et le model-checking. De plus, grâce à sa représentation formelle en couches des composants et de leurs interactions, BIP permet d'obtenir par construction la correction de propriétés telles que l'absence d'interblocages ou la vivacité [GGMC+07].

3.3.5.4.b Architectural Interaction Diagrams (AIDs)

Une autre approche permettant de mixer différents modes d'interaction dans un même modèle est proposée dans [RC03]. Elle introduit les « bus »comme entités de première classe dans un paradigme de modélisation appelé « Architectural Interaction Diagrams (AIDs) ». Dans cette

approche, les bus représentent la notion de communication inter-processus. Le framework proposé est extensible, permettant l'ajout de nouvelles primitives de communication inter-processus. Une définition mathématique des AIDs est donnée comme base pour l'analyse du comportement des éléments des AIDs.

L'exemple présenté par la figure 3.17 montre deux types d'interaction basés sur les bus : le bus b modélise une interaction de type rendez-vous synchrone tandis que le bus q modélise une interaction via un canal de type FIFO. Dans cet exemple les deux types d'interaction interviennent à des niveaux hiérarchiques différents du modèle mais ils pourraient a priori être utilisés au même niveau de la hiérarchie.

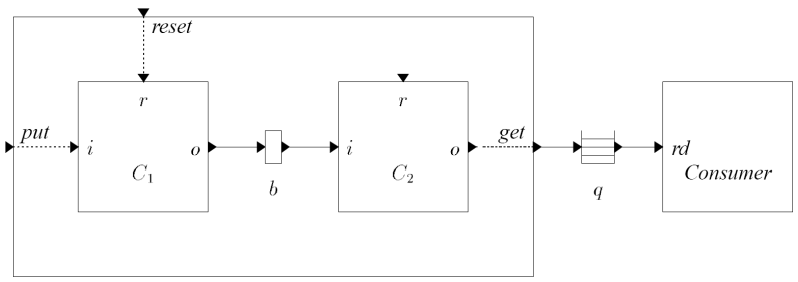


Fig. 3.17 – Interactions hétérogènes avec les AIDs (source : [RC03])

Ces deux approches, BIP et AIDs, partagent des concepts tout à fait similaires : elles permettent la représentation du comportement atomique des composants par des systèmes de transition, elles permettent la définition de types d'interactions et leur utilisation de manière hétérogène dans un modèle et enfin elles permettent l'assemblage de composants pour former des composites. Cependant, il semble que BIP permette la correction par construction alors que les AIDs sont basés sur une sémantique formelle qui ne peut être utilisée que pour l'analyse par simulation ou test et, seulement dans certains cas, pour le model-checking.

3.3.6 Autres approches

3.3.6.1 Co-simulation

Le terme « co-simulation » désigne en fait tout un ensemble de techniques permettant de co-exécuter des modèles hétérogènes, soit en utilisant un unique simulateur, soit en en interconnectant plusieurs.

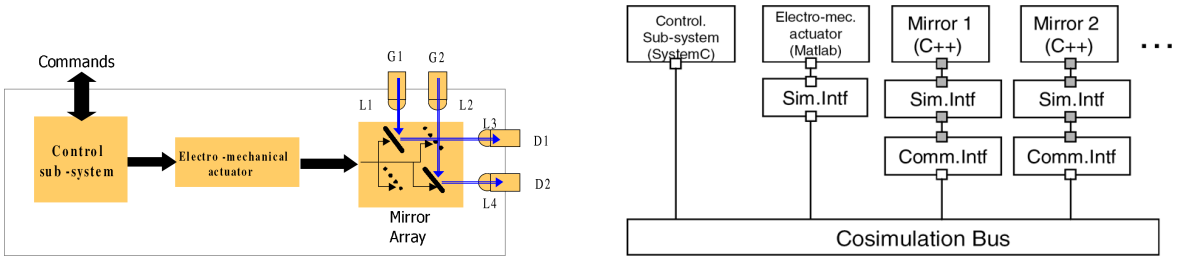
3.3.6.1.a Environnement unique

Une première catégorie d'approches repose sur l'utilisation d'un environnement de simulation unique qui permet de simuler tous les modèles. Par exemple, POLIS [BSC+97] fournit un environnement de simulation basé sur le domaine Discrete Events de Ptolemy pour la co-simulation matériel/logiciel.

3.3.6.1.b Bus de co-simulation

Dans une deuxième catégorie d'approches, plusieurs simulateurs sont connectés entre-eux via un bus de manière à ce qu'ils puissent exécuter les différents modèles de manière cohérente. Ces approches bénéficient des performances et de la précision de simulateurs qui sont optimisés pour les paradigmes qu'ils supportent. Les principales difficultés rencontrées concernent la synchronisation des différents simulateurs (qui sont alors hétérogènes) et la communication de données hétérogènes entre eux pendant la simulation. Dans [GYNJ01], les auteurs présentent un outil de co-simulation basé sur SystemC dans lequel des adaptateurs permettent de connecter des

« modules » hétérogènes, c'est-à-dire des environnements de simulation simulant des modèles de différentes parties du système. Un adaptateur est composé (a) d'une interface de simulation qui adapte un simulateur donné au bus de co-simulation et est chargé de sa synchronisation avec les autres simulateurs, et (b) d'une interface de communication dédiée à un niveau ou à un protocole de communication qui permet de gérer les données issues du simulateur de manière à les faire transiter sur le bus. Ces interfaces sont générées automatiquement à partir d'une librairie d'interfaces.



(a) Switch micro-electro-mécanique optique

(b) Bus de co-simulation

FIG. 3.18 – Bus de co-simulation appliqué à la simulation d'un switch micro-electro-mécanique optique (source : [NMK+02])

La figure 3.18, empruntée à [NMK+02], illustre l'utilisation du bus de co-simulation pour la simulation de modèles hétérogènes représentant le fonctionnement d'un switch micro-electro-mécanique optique. Les switch optiques sont généralement utilisés pour rediriger les signaux entre différentes fibres optiques. Ils contiennent des miroirs qui, selon leur position, peuvent réfléchir totalement, partiellement ou pas du tout la lumière reçue en entrée. Dans cet exemple, le switch optique considéré est un MEM c'est-à-dire un système Micro-Electro-Mécanique. La technologie à la base des MEMs combine l'électronique avec la mécanique en intégrant sur un même élément semi-conducteur le circuit électronique et des dispositifs mécaniques (capteurs, valves, engrenages, miroirs, etc.) de très petite taille. L'architecture de cet interrupteur est représentée sur la figure 3.18a. Les commandes que reçoit l'interrupteur sont traitées par le sous-système de contrôle (processeur) qui calcule les signaux électroniques nécessaire pour piloter l'actuateur electro-mécanique chargé de faire changer les miroirs de position. Les différents modèles représentant les comportements du sous-système de contrôle, de l'actuateur electro-mécanique et des miroirs sont décrits en utilisant respectivement SystemC, Matlab et C++. Ces modèles sont simulés par des simulateurs spécifiques. Comme cela est représenté sur la figue 3.18b, ces simulateurs sont connectés au bus de co-simulation grâce à des interfaces de synchronisation et de communication. Remarquons que le simulateur du sous-système de contrôle est directement connecté au bus car il s'agit d'un simulateur SystemC et SystemC est le langage utilisé par les connecteurs sur le bus.

3.3.6.1.c Co-simulation « ad-hoc »

Enfin, il existe aussi des approches permettant de connecter spécifiquement certains simulateurs. Dans l'approche MUSIC [CHM+99], la co-simulation SDL/Matlab est utilisée pour la validation de niveau système. Des environnements de co-simulation dédiés à l'hétérogénéité logiciel/matériel sont aussi proposé sous une forme commerciale dans des outils comme Seamless, de Mentor Graphics [Men].

3.3.6.2 Méga-modèles

Le concept de « méga-modèle » [BJV04, Fav06] est relativement récent. Selon J. Bézivin et J. M. Favre, les méga-modèles ont pour objectif de fournir une structure pour représenter les relations globales qui existent entre les modèles. L'introduction de ces informations a pour but de permettre notamment une meilleure interopérabilité entre les outils. L'approche s'applique non seulement aux modèles mais également aux méta-modèles, aux modèles de transformations, etc. Le type d'implémentation à utiliser pour mettre en oeuvre cette approche n'est pas fixé pour le moment.

Dans [Fav06], six types de relations sont proposées : (1) décomposition, (2) représentation, (3) appartenance (à un ensemble), (4) conformité, (5) transformation et (6) sémantique. La figure 3.19 illustre l'utilisation des trois premières de ces relations sur un méga-modèle représentant l'ensemble des modèles en jeu autour d'un logiciel écrit en PL1 [IBM] simulant le comportement du système solaire selon Ptolémée. Le logiciel est appelé « ThePSProgram ». Il est considéré comme un modèle représentant le système solaire. Il est lui-même représenté par un diagramme de classes, qui est un élément du langage UML. Notons que dans l'exemple présenté ici, un langage contient l'ensemble des éléments qui peuvent être construits selon les règles données par sa grammaire. Le logiciel « ThePSProgram » étant décrit dans un langage de programmation appelé PL1, il est donc un élément du langage PL1. Une représentation du langage PL1 est donnée par sa grammaire, décrite en Yacc [Joh] et donc élément du langage Yacc. La grammaire de Yacc pouvant elle-même être décrite en Yacc, elle est à la fois une représentation du langage Yacc et l'un de ses éléments (une structure identique serait utilisée dans le cas du MOF [OMGf] car il s'auto-décrit également).

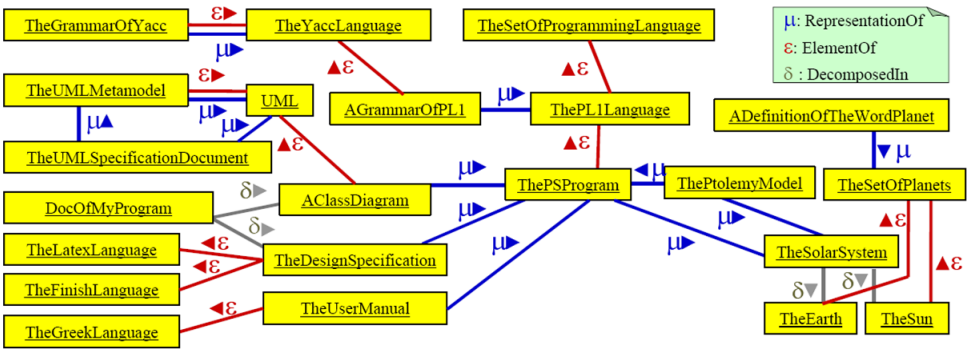


FIG. 3.19 – Exemple de méga-modèle pour un logiciel simulant le système solaire (source : [Fav06])

Les méga-modèles permettant de définir des liens sémantiques entre des éléments de type modèle (modèle, méta-modèle, etc.) potentiellement hétérogènes, ils peuvent être utilisés pour étudier les rôles que jouent les différents modèles dans la modélisation multi-paradigme. Par ailleurs, ils permettront peut-être, à terme, de mener des raisonnements à un niveau macroscopique sur un ensemble de modèles liés les uns aux autres. En ce sens, ils peuvent donc être considérés comme une technique du domaine de la modélisation multi-paradigme.

3.3.7 Particularités liées au traitement de vues multiples et de niveaux d'abstraction multiples

Les techniques que nous avons vues dans les sections précédentes s'appliquent de manière générale à la composition de modèles. Dans cette section nous nous intéressons plus particulièrement à la composition de modèles qui représentent des vues différentes d'un même élément ou qui représentent différents éléments à des niveaux d'abstractions différents car ceux-ci demandent

un traitement particulier.

3.3.7.1 Composition de vues

Une vue d'un modèle est une projection de ce modèle selon une perspective particulière. Le concept de vue permet de modéliser différents aspects d'un même élément. Différents modèles constituant des vues différentes d'un même élément interagissent via des points d'intersection dans les descriptions comportementales qu'ils contiennent. Considérons par exemple un contrôleur logiciel qui fonctionne selon différents modes et n'a pas la même consommation d'énergie dans chacun de ces modes. Le concepteur pourra vouloir réaliser deux modèles : un modèle décrivant le fonctionnement du contrôleur (probablement à base d'automates) et un modèle de la consommation d'énergie. La notion de mode est alors à l'intersection entre ces deux modèles qui représentent le même système selon deux points de vue différents. Lors de l'assemblage de modèles de ce type, la difficulté est de pouvoir établir formellement les points d'intersection entre eux de manière à permettre la répercussion des modifications de part et d'autres ou de façon à pouvoir simuler conjointement les deux modèles.

UML est probablement l'une des approches de modélisation multi-vue la plus connue. Avec UML, il est possible de réaliser différentes vues en utilisant des diagrammes différents à partir des même éléments de modélisation (classes, objets, méthodes, etc.). Cependant les relations entre les différentes vues d'UML ne sont pas formalisées et ne permettent qu'un traitement automatisé minimum (répercussion des modifications statiques). Rosetta (voir la section 3.3.4.2) définit formellement la notion de facette pour représenter une vue d'un composant et permet de composer différentes facettes à la façon d'un « feuilleté » pour obtenir une représentation multi-vue d'un composant. Dans [Att08], l'approche multi-vue proposée repose sur l'utilisation d'un modèle formel de référence dont sont dérivés les modèles spécifiques utilisés pour des analyses particulières. Un modèle dérivé est construit par traduction, réduction ou extension du modèle de référence. Les résultats des analyses effectuées sur les modèles spécifiques (preuves de propriétés etc.) sont reportés sur le modèle de référence pour peu que ces résultats concernent des points d'intersection entre ces deux modèles. Le principe de cette approche est illustré sur la figure 3.20.

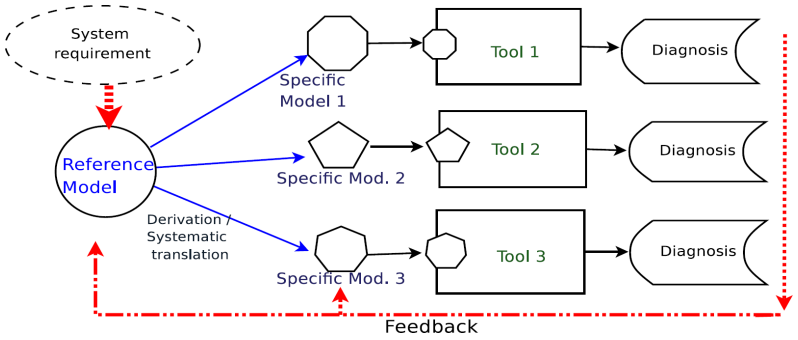


FIG. 3.20 – Dérivation de vues à partir d'un modèle de référence (source : [Att08])

3.3.7.2 Composition de modèles à différents niveaux d'abstraction

Le problème de l'hétérogénéité des niveaux d'abstraction de différents modèle se pose dans deux types de cas : (1) lorsque l'on dispose de deux modèles d'un même composant à deux niveaux d'abstraction différents et que l'on souhaite prouver que l'un est conforme à l'autre et (2) lorsque l'on dispose de modèles représentant les différents composants d'un système à des niveaux d'abstraction différents et que l'on souhaite les intégrer dans un même modèle. Le cas (1) est lié aux phases de vérification et validation et apparaît notamment lorsqu'un modèle plus précis est dérivé manuellement d'une spécification de haut niveau et qu'il est alors nécessaire de

montrer que le modèle dérivé est conforme à la spécification. Le cas (2) permet de gagner du temps au cours du cycle de développement en vérifiant, au plus tôt et quel que soit le niveau d'avancement du développement de chacun des composants du système, que ceux-ci peuvent fonctionner ensemble en utilisant leurs modèles (pour de la simulation par exemple).

Nous avons déjà évoqué le premier type de cas, qui est largement traité dans la littérature autour de la relation abstraction/raffinement. La difficulté augmente si ces modèles sont décrits dans des paradigmes différents. Des techniques issues du domaine du test de conformité ou s'appuyant sur le concept de bisimulation ou encore des techniques telles que celles évoquées dans la section 3.3.5.2.b peuvent être utilisées.

Dans le deuxième cas, différentes techniques peuvent être utilisées en fonction de l'objectif visé en intégrant ces modèles. Les transformations de modèles peuvent par exemple être utilisées pour transformer ces modèles vers un formalisme unique de manière à uniformiser leur niveau d'abstraction. Dans Ptolemy, il est possible d'intégrer dans un même modèle des composants décrits avec des niveaux de détail différents puisque tous sont considérés comme des boîtes noires. Enfin SystemC [IEE, OSC] est un langage dédié à la modélisation de systèmes sur puce dans lequel différents niveaux d'abstraction sont définis précisément et peuvent être utilisés conjointement lors de la simulation. Les niveaux définis dans SystemC sont les suivants, du plus abstraits au plus détaillé, leur utilisation au cours du processus de modélisation étant représentée sur la figure 3.21 :

- UTF (UnTimed Functional) : le modèle ne comporte aucune notion de temps, mais seulement un ordre éventuel dans l'exécution des événements.
- TF (Timed Functional) : le modèle comporte des notions de durée (temps d'exécution des processus, latence, temps de propagation, ...)
- BCA (Bus Cycle Accurate) : le comportement du système est modélisé de manière à ce que les transactions sur les interfaces de communication soient correctes au cycle près mais le modèle ne comporte pas d'information sur les signaux échangés.
- BA ou CABA (Bit Accurate, aussi appelé Cycle Accurate/Bit Accurate) : modèle BCA qui comporte en plus des informations sur les signaux, précises au bit près.
- RTL (Register Transfert Level) : chaque bit, chaque cycle, chaque registre du système est modélisé

Après le niveau BCA, le modèle permet de faire une synthèse conjointe en portes logiques de la partie logicielle et de la partie matérielle du système sur puce.

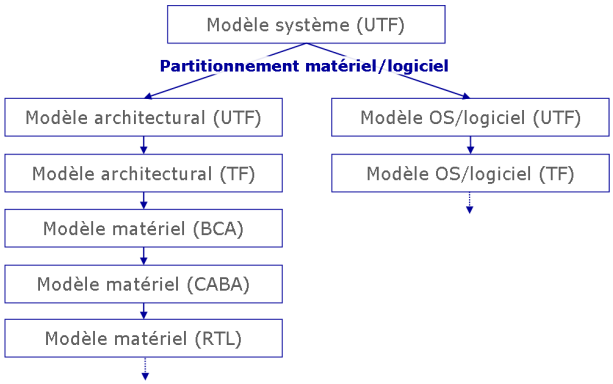


FIG. 3.21 – Niveaux d'abstraction définis pour la conception de systèmes sur puce avec SystemC

3.4 Proposition de classification des approches de modélisation multi-paradigme

Les différentes techniques que nous avons passées en revue dans la section précédente proviennent d'horizons différents et proposent des fonctionnalités très différentes. Leurs avantages respectifs sont donc très difficiles à comparer de facto. Cependant, considérant les objectifs de la modélisation multi-paradigme ainsi que les problématiques de l'ingénierie dirigée par les modèles, nous avons identifié un ensemble de caractéristiques qui qualifieraient une approche de modélisation multi-paradigme idéale. Dans cette section, nous présentons ces caractéristiques, détaillons les raisons pour lesquelles nous les avons choisies et listons les approches qui présentent respectivement ces caractéristiques.

3.4.1 Support ouvert pour de multiples paradigmes

La combinaison ad hoc d'un ensemble fini de paradigmes de modélisation est un problème bien cerné et pour lequel de nombreuses solutions existent. Nous en avons présenté quelques-unes dans la section 3.3.1 notamment. Ainsi par exemple, VHDL-AMS est un langage résultant de la combinaison de deux paradigmes de modélisation : l'un permettant de modéliser des évènements discrets et l'autre permettant de manipuler le temps continu. De la même manière, Modelica [FE98] résulte de la combinaison de paradigmes à temps discret et continu pour supporter la spécification de systèmes en utilisant des équations différentielles, algébriques ou encore discrètes. Cependant, nous avons vu dans le chapitre précédent que le développement de l'ingénierie dirigée par les modèles entraine l'utilisation de nombreux types de langages dont notamment des langages domaines spécifiques (DSLs, voir section 2.4.1.5.a). Des outils plus flexibles sont donc nécessaires. Grâce aux techniques de méta-modélisation, des outils tels que GME [Dav03] ou le Eclipse Modeling Project (EMF, GEF, GMF) [Ecla] permettent maintenant la génération automatique de l'outillage support (éditeurs graphiques, etc.) pour des langages définis par les utilisateurs. De la même manière, les outils de modélisation multi-paradigme devraient pouvoir être facilement et automatiquement étendus pour supporter des langages de modélisation additionnels définis par les utilisateurs. Cela n'est possible que si l'approche retenue est suffisamment générique et a été conçue pour supporter un ensemble ouvert de langages de modélisation.

Les approches de transformation de modèle telles que [dLV02] présentent cette caractéristique de généricité. Pour un langage de modélisation dont le méta-modèle est connu et formalisé, le coût de l'ajout du support de ce langage dans l'outil est en majeure partie dû à la conception des transformations permettant de transformer un modèle décrit dans ce langage en un modèle décrit dans le/les langage(s) cible(s).

La plupart des approches de composition de sémantique vues dans la section 3.3.4 présentent également cette caractéristique. Ptolemy II semble être l'approche qui supporte l'éventail le plus large de paradigmes de modélisation tandis que Metropolis et l'approche décrite dans [PY96] semblent restreintes à certains types de langages. Avec les approches de composition de sémantique, le coût pour ajouter un langage à une approche correspond au coût de la description de la sémantique du langage avec les outils proposés par l'approche (algèbres, coalgèbres, modèles de calcul, etc.). Cette tâche peut se révéler particulièrement difficile. Cependant la possibilité d'agréger des modèles décrits avec ce nouveau langage est généralement automatiquement dérivée des mécanismes sous-jacents à l'approche, ce qui est l'intérêt principal de ce type d'approche.

Certains types d'approches de co-simulation peuvent également être considérées comme générique si elles supportent l'ajout de simulateurs supplémentaires. Par exemple, l'approche décrite dans [GYNJ01] permet la connexion de simulateur additionnels sur le bus de co-simulation grâce à la génération automatique d'adaptateurs. Le coût de chaque ajout de simulateur provient de l'analyse nécessaire du simulateur pour ajouter les éléments correspondants à la librairie qui est utilisée pour la génération des adaptateurs.

3.4.2 Support pour de multiples activités du cycle de développement

Pendant les différentes phases du cycle de développement, les activités du concepteur ont différents objectifs : capturer les exigences, modéliser le comportement fonctionnel, analyser les performances, générer le code logiciel, valider les propriétés, etc. Comme évoqué dans le chapitre précédent (section 2.5.2), ces différentes activités sont l'une des sources de l'hétérogénéité puisque différents modèles (et très souvent différents paradigmes de modélisation) sont utilisés pour ces différents objectifs. Nous avons vu dans la section 3.2.3.4 que même si certains formalismes supportent parfois plusieurs activités de manière ad hoc, des solutions plus générales sont nécessaires afin de maintenir automatiquement la cohérence entre les différents modèles tout au long du cycle de développement.

Par ailleurs, nous remarquons que le nombre d'outils assistant le concepteur pour des tâches spécifiques durant le cycle de développement tend à croître de façon significative. Si de tels outils permettent au concepteur de bénéficier d'une meilleure adéquation et de meilleures performances, leur variété génère des difficultés supplémentaires liées aux différences dans les formats et les sémantiques supportées. Ainsi par exemple, il arrive fréquemment que différents modeleurs UML n'interprètent pas de la même manière le même modèle. De même, un modèle Statechart pourra être interprété différemment suivant les outils. En conséquence, nous estimons que le support de plusieurs activités du cycle de développement est une caractéristique clé d'un outil de modélisation multi-paradigme.

Parmi les approches que nous avons évoquées jusqu'à présent, la plupart ciblent seulement une ou deux tâches du cycle de développement (généralement une tâche de conception et une tâche liée à la vérification ou la validation). Les exceptions notables sont (a) l'approche présentée par M. Pezzè et M. Young dans [PY96], qui supporte une large gamme de techniques d'analyse (depuis l'analyse de l'atteignabilité d'état à l'exécution de modèle) et (b) Metropolis qui est supporté par des outils de vérification, de simulation et de synthèse.

3.4.3 Support pour le raisonnement formel

Les phases de vérification et de validation sont d'une importance cruciale dans le cycle de développement, en particulier lors de la conception de systèmes critiques. Lorsque l'on considère des modèles hétérogènes, les tâches de vérification et de validation, qui sont par nature hautement complexes et difficiles, requièrent des efforts extraordinaires (voire sont quasiment impossibles à mener). Les multiples problématiques dans ce domaine font l'objet de différentes initiatives de recherche. Dans cette dernière section, nous passons en revue des approches qui permettent de raisonner à propos de propriétés formelles sur des modèles hétérogènes, fournissant ainsi un support essentiel pour les activités de vérification et de validation.

Vis-à-vis de ce critère, nous avons trouvé Rosetta (voir la section 3.3.4.2) et BIP (voir la section 3.3.5.4.a) particulièrement intéressants. Rosetta se distingue par sa capacité à combiner formellement différentes vues d'un même modèle. Il ne nous semble pas que d'autres approches que Rosetta proposent cette fonctionnalité, bien que d'autres approches basées sur le concept de vue existent. UML [OMGl] est peut-être la plus connue de ces approches. Pourtant, les relations entre les différentes vues d'un modèle UML ne sont pas formalisées et ne permettent donc pas un traitement automatique. Grâce aux coalgèbres, Rosetta semble supporter un large spectre de sémantiques, même non basées sur la notion d'état. Cependant, Rosetta a l'inconvénient d'avoir été conçu comme un langage de spécification avec un haut niveau d'abstraction, ce qui implique que les modèles décrits en Rosetta ne sont pas directement exécutables. Cela signifie notamment que le concepteur ne pourra pas utiliser Rosetta dans les dernières phases du processus de développement et devra changer de langage de représentation. Par ailleurs, les outils qui supportent Rosetta ne semblent pas encore matures. Des parseurs, des interpréteurs (seulement pour un sous-ensemble du langage) ou des générateurs de vecteurs de test ont été

développés mais semblent des initiatives déconnectées les unes des autres, menant ainsi à une multiplication des outils à utiliser lorsque l'on veut exploiter des spécifications Rosetta.

En ce qui concerne BIP, son principal atout de notre point de vue est la capacité à obtenir la correction par construction vis-à-vis de plusieurs propriétés telles que l'exclusion mutuelle ou l'absence d'interblocages. Dans le contexte des systèmes complexes, une telle capacité peut être un sérieux avantage puisque les méthodes classiques de model-checking atteignent leurs limites. Cependant, cette fonctionnalité semble très expérimentale pour le moment et restreinte à un sous-ensemble du langage BIP [GGMC+07]. De façon plus traditionnelle, BIP est supporté par une plate-forme d'exécution et un outil de model-checking appelé IF. Nous remarquons que le raisonnement formel dans BIP nécessite que la hiérarchie utilisée pour structurer les modèles soit non stricte (les composants ne sont alors pas considérés comme des boîtes noires). Cela peut être considéré comme un inconvénient pour le concepteur car les modèles perdent en modularité. De plus, la façon dont le comportement des composants est exprimé (en utilisant des systèmes de transitions étiquetées) limite la gamme des sémantiques supportées par BIP.

Parmi d'autres approches intéressantes, nous avons aussi sélectionné Metropolis et les automates d'interface. Metropolis met à la disposition du concepteur une fonctionnalité de vérification de modèles qui s'appuie sur sa sémantique formelle sous-jacente (basée sur les algèbres de structures de traces). Le principal interêt de Metropolis est la large palette d'outils proposée. En contrepartie, seuls les langages orientés réseaux de processus sont supportés.

Enfin, dans le domaine de la conception orientée composants, le travail de L. de Alfaro et T. Henzinger sur les automates d'interface et les théories des interfaces définissent des bases formelles solides pour la vérification de la compatibilité et de la composabilité. Les automates d'interface ont été appliqués dans le projet Ptolemy II à la vérification de compatibilité entre des acteurs et des directeurs de domaine. En tant que tels les automates d'interface semblent prometteurs, bien que leur application à des approches autres qu'orientées composant n'a pas été envisagée pour le moment.

En conclusion, il est évident que le raisonnement formel sur les propriétés globales de modèles hétérogènes est un défi. Comme nous l'avons vu dans cette section, de nombreux travaux sont en cours dans ce domaine. Des progrès importants ont été réalisés mais se heurtent à des limitations importantes.

3.5 Conclusion

Dans ce chapitre nous avons tout d'abord proposé une définition du domaine de la modélisation multi-paradigme dans la perspective des problématiques soulevées dans le chapitre précédent concernant l'hétérogénéité des paradigmes de modélisation utilisés au cours du cycle de développement système. Après avoir passé en revue les différents axes de recherche qui nous semblent majeurs dans ce domaine, nous avons présenté différentes techniques dont nous avons montré l'intérêt pour répondre aux problématiques de la modélisation multi-paradigme. Enfin nous avons proposé des critères de caractérisation formant les bases d'un framework de comparaison pour les approches de modélisation multi-paradigme.

Ce chapitre clôt notre première partie sur le thème de la conception des systèmes et l'hétérogénéité. Les principales contributions présentées dans cette partie sont d'une part des propositions de formalisation des notions d'hétérogénéité des modèles et de modélisation multi-paradigme, et d'autre part un état de l'art des techniques pouvant être considérées dans le contexte des définitions proposées, assorti d'un ensemble de critères permettant de les caractériser. Nous proposons, dans la partie suivante, une approche pour l'exécution de modèles multi-paradigmes appelée ModHel'X.

Deuxième partie

Composition exécutable de modèles hétérogènes avec ModHel'X

4

ModHel'X : une approche de la composition de modèles hétérogènes

4.1 Introduction

Dans la partie précédente nous avons présenté le domaine de la modélisation multi-paradigme, ses problématiques ainsi qu'un état de l'art et un cadre de comparaison pour différentes techniques pouvant être utilisées dans ce domaine. Cette étude du domaine de la modélisation multi-paradigme constitue l'une des contributions de cette thèse. Nous avons vu notamment comment les techniques que nous avons passées en revue pouvaient être utilisées pour composer des modèles hétérogènes. Dans ce chapitre, nous examinons tout d'abord plus en détail les problématiques rencontrées lors de la composition de modèles hétérogènes. Puis nous proposons une approche de la composition de modèles orientée vers l'exécution. Cette approche a pour objectif de permettre de tester le résultat de la composition par simulation mais ne traite pas de la vérification de propriétés sur la composition (qui est cependant envisagée dans les perspectives de l'approche). Nous présentons les principes sur lesquels s'appuie notre approche et donnons une vue globale de son fonctionnement avant de détailler successivement ses différentes composantes.

4.2 Problématiques de la composition de modèles hétérogènes

Lors de la composition de modèles hétérogènes, le problème central est d'établir la sémantique du modèle résultant et de s'assurer que les différentes parties hétérogènes du modèle interopèrent correctement lorsque le modèle est utilisé pour analyser des propriétés du système [HS06].

Pour cela, la première difficulté est de disposer d'une spécification précise et exploitable par des outils informatiques de la sémantique des formalismes de modélisation en jeu. En effet, excepté pour certains langages fondés mathématiquement, la sémantique des langages de modélisation est généralement décrite en langage naturel. Cela est d'ailleurs source d'ambiguïtés menant à diverses interprétations par différents outils tout au long de la chaîne de conception. Lorsque l'on veut combiner différents langages dans un modèle, des ambiguïtés dans la sémantique de l'un deux rendent impossible la définition de la sémantique globale de la composition. Dans ce contexte, des variations sémantiques comme celles existant pour le langage UML sont acceptables seulement si la variation utilisée est explicitement désignée. Nous avons vu dans la section précédente différentes techniques permettant de décrire la sémantique d'un langage de modélisation : ajout de sémantique opérationnelle sur un méta-modèle, ancrage d'un méta-modèle sur une Semantic Unit, utilisation du concept de modèle de calcul, etc.

La deuxième difficulté pour obtenir un modèle multi-paradigme global qui ait du sens à partir de différents modèles hétérogènes est de définir la « glue » permettant de réunir ces différents modèles pour qu'ils n'en forment plus qu'un. Le rôle de cette « glue » est notamment d'adapter les différentes parties du modèle de manière à ce qu'elles puissent interopérer lorsque le modèle est utilisé pour analyser des propriétés du système. La définition d'un tel mécanisme d'adaptation est, à notre avis, soumise à deux contraintes majeures.

Tout d'abord, nous estimons que les parties de modèle qui sont réunies ne doivent pas être modifiées pour devenir compatibles les unes avec les autres. Cela est particulièrement important lorsque ces parties de modèle sont fournies par des tiers. En effet, celles-ci ne sont alors pas nécessairement modifiables. Et si elles le sont, les laisser intactes permet une meilleure réutilisabilité. Enfin, si elles ont été validées, toute modification rend leur validation caduque.

Par ailleurs, nous pensons que l'ajustement sémantique à réaliser entre deux parties d'un modèle décrites dans deux langages donnés ne dépend pas seulement des formalismes en jeu mais aussi du système qui est modélisé. En effet, il existe souvent des façons usuelles de « coller » ensemble des modèles décrits dans des langages donnés, mais celles-ci ne sont généralement pas uniques et peuvent représenter des adaptations par défaut qui ne sont pas appropriées pour certaines situations. Par exemple, il existe un patron d'adaptation usuel permettant d'assembler

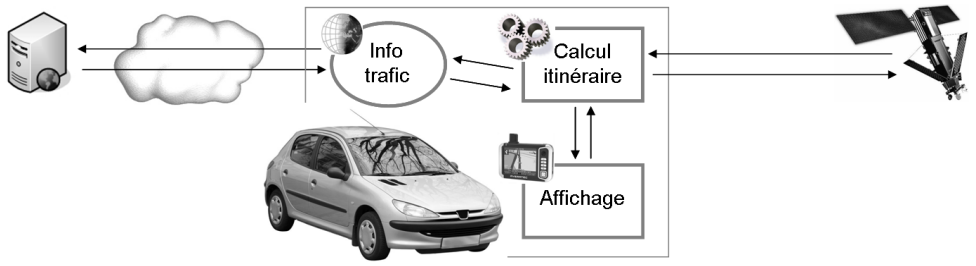


FIG. 4.1 – Système de guidage de véhicule

un modèle décrit dans un paradigme à temps continu et un modèle décrit dans un paradigme à temps discret : celui-ci consiste à échantillonner les signaux allant du continu au discret et à interpoler les signaux allant du discret au continu. Cependant le choix de la fréquence d'échantillonnage et du type d'interpolation dépend complètement du système modélisé. Nous estimons donc qu'il est important que le concepteur puisse d'une part choisir le patron d'adaptation entre deux parties hétérogènes d'un modèle et que d'autre part qu'il puisse éventuellement en ajuster les paramètres.

De façon à illustrer ces derniers propos, nous considérons le modèle d'un système de guidage de véhicule qui intègre les conditions de trafic courantes lorsqu'il calcule un itinéraire. Comme cela est illustré par la figure 4.1, ce système est composé d'un algorithme de routage qui calcule l'itinéraire à partir de la destination et de la position courante de la voiture, et d'un sous-système qui récupère les informations concernant le trafic via un réseau. L'information sur l'état du trafic, lorsqu'elle est disponible, est utilisée par l'algorithme de routage pour minimiser la durée du trajet. L'algorithme de routage reçoit régulièrement la position de la voiture et met à jour l'itinéraire. Un formalisme à flots de données synchrones est particulièrement adapté pour modéliser ce type d'algorithme, qui est apparenté à un système de traitement du signal. Le système de récupération d'information sur le trafic est développé par un fournisseur, qui utilise un formalisme a événements discrets afin de pouvoir modéliser le temps de réponse du réseau. Si l'on utilise Ptolemy II (voir la section 3.3.4.4) pour modéliser ce système de guidage, les modèles de calcul correspondants sont représentés respectivement par les domaines SDF (Synchronous Data Flow) et DE (Discrete Events). Avec Ptolemy, il n'est pas possible d'embarquer le modèle DE (représentant le sous-système de récupération d'informations sur le trafic) directement dans le modèle SDF. En effet, dans le domaine SDF les entrées sont présentées en flot continu aux composants, qui doivent fournir immédiatement leurs réponses de la même manière, tandis que dans DE les événements peuvent survenir à n'importe quel moment et surtout de manière non synchrone (dans notre exemple, le système de récupération d'informations sur le trafic ne peut produire de réponse que lorsqu'il a reçu une réponse à sa requête sur le réseau). En conséquence, le concepteur doit modifier le modèle DE en ajoutant un composant d'échantillonnage qui délivre les données de manière synchrone en répétant le dernier échantillon de donnée produit lorsqu'aucune donnée nouvelle n'est disponible. Les problèmes qui surviennent dans ce type de configuration sont les suivants :

- Le modèle du système de récupération d'information est altéré et ne représente plus le comportement du composant qui sera fourni par le fournisseur tiers. Cela peut provoquer des incohérences à la fin du cycle de développement.
- Puisque l'adaptation sémantique entre les deux modèles est faite à l'intérieur du modèle du sous-système, celle-ci n'est pas protégée de changements réalisés par le fournisseur du sous-système.
- Si le formalisme utilisé dans l'un des modèles change (comme cela peut être nécessaire pour raffiner le modèle par exemple), l'adaptation sémantique doit être exprimée de nouveau dans le nouveau formalisme. Cela est incompatible avec les principes de modularité et de

réutilisation.
Il est donc important de permettre au concepteur de spécifier la politique d'adaptation sémantique en dehors des modèles qui sont assemblés.

4.3 Préambule : approche proposée et concepts sous-jacents

Nous proposons une approche de la composition de modèles hétérogène appelée *ModHel'X* ayant pour objectifs de :

1. permettre la spécification de la sémantique d'un langage de modélisation de manière exécutable et sans référer à une quelconque instance de modèle (c'est à dire au niveau méta-modèle) ;
2. permettre la spécification explicite du mécanisme de composition à utiliser entre des modèles hétérogènes ;
3. permettre l'exécution de modèles hétérogènes (pour la simulation notamment).

Nous donnons dans les sections suivantes une introduction progressive aux concepts sous-jacents de notre approche et une présentation intuitive des principes de celle-ci.

4.3.1 Encapsulation du comportement : boîtes noires

Nous adoptons une approche orientée composants, c'est-à-dire que le système est décomposé en éléments qui réalisent une ou plusieurs fonctions et qui, par leur coopération, réalisent la fonctionnalité globale du système. Dans notre approche, nous considérons les composants comme des *boîtes noires*. Le principe des boîtes noires est largement utilisé dans de nombreux domaines, notamment dans le domaine de la cybernetique [Wie48]. Nous appelons nos composants boîtes noires des *blocks (blocs)*.

A la différence d'approches telles que le CORBA Component Model (CCM) [OMGc], nous ne faisons aucune hypothèse sur la façon dont nos composants interagissent les uns avec les autres (c'est à dire sur leur protocole de communication qui, dans le CCM est l'appel de méthode asynchrone par défaut). La notion de bloc que nous introduisons ici est beaucoup plus abstraite que la notion de composant. De cette façon, nous évitons de limiter notre approche à certaines classes de paradigmes de modélisation.

Du point de vue d'un modèle, tout ce qu'il est possible de savoir sur le comportement des blocs qui le composent est ce qu'il est possible d'observer au niveau de leurs entrées et de leurs sorties. Ce qui se passe à l'intérieur d'un bloc, ni le fait même qu'il se passe quelque chose ou non, n'est donc pas visible du modèle dans lequel le bloc est utilisé.

Le comportement d'un bloc (lorsque l'on « ouvre » la boîte noire) peut être décrit par un modèle. Comme les blocs sont des boîtes noires, le modèle qui décrit leur comportement est

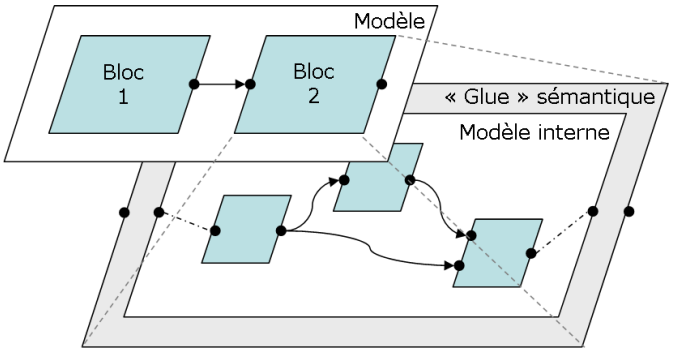


Fig. 4.2 – Hierarchie et encapsulation

complètement découplé du modèle dans lequel ils sont utilisés, qui décrit le comportement global du système. Nous tirons donc ici parti des principes de *l'encapsulation* et de la *hiérarchie*. Ces principes constituent un moyen simple de combiner des modèles hétérogènes pour former un modèle global, ainsi qu'un mécanisme simple d'abstraction [dAH01b] (le modèle interne d'un bloc constituant une vue plus détaillée du comportement du bloc que le bloc lui-même). Par ailleurs, le calcul du comportement d'un bloc peut être *délégué* à son modèle interne, ce qui renforce le découplage entre les modèles. Enfin, par cette structuration des modèles, nous restreignons l'hétérogénéité localement à la frontière entre deux modèles : le modèle dans lequel est utilisé un bloc et le modèle interne du bloc. C'est à cette frontière que le mécanisme de composition des sémantiques, une sorte de « glue » sémantique, est mis en place. Ces principes sont illustrés sur la figure 4.2.

4.3.2 Observation de l'exécution : mises à jour de blocs, modèles de calcul et snapshots

Les composants de nos modèles étant des boîtes noires, leur fonctionnement interne est rendu opaque et il ne nous est pas possible de contrôler le cours de leur exécution. En conséquence, nous adoptons une autre approche de l'exécution : plutôt que de *déclencher* leur comportement, nous nous contentons *d'observer* leur comportement. L'observation d'un bloc peut donc être déclenchée à n'importe quel moment sans tenir compte de son modèle interne. Cela signifie qu'un bloc peut avoir un comportement actif même lorsqu'il n'est pas observé. Lorsqu'il est nécessaire d'obtenir une observation du comportement d'un bloc, celui-ci se voit demander une *mise à jour* de son comportement observable, c'est-à-dire de ses sorties (en fonction des entrées qui lui auront éventuellement été fournies). Il est alors de sa responsabilité de fournir une observation cohérente de son comportement au moment où cela lui est demandé. Le comportement d'un bloc est donc vu à travers une suite d'observations.

Pour définir le comportement global d'un modèle, nous nous appuyons sur le concept de *modèle de calcul (MoC)* (voir les sections 3.3.2.3 et 3.3.4.4). Dans Ptolemy II, le modèle de calcul définit la façon dont sont orchestrés les déclenchements des acteurs et la façon dont les données sont transmises entre eux. Dans notre approche, le modèle de calcul est l'ensemble des règles qui permettent de calculer une observation d'un modèle à partir de l'observation de chacun des blocs qui le composent, de manière fidèle à la sémantique du paradigme de modélisation impliqué dans le modèle. Le modèle de calcul définit (a) la façon dont doivent être orchestrées les observations des blocs et (b) la façon dont doivent être transmises les observations d'un bloc à l'autre (car pour donner une mise à jour de son comportement, un bloc peut avoir besoin d'informations sur le comportement d'autres blocs dont il dépend). Le comportement d'un modèle est donc vu à travers une suite d'observations construites à partir d'observations des blocs qui le composent. Ces observations définissent, tout comme pour les blocs, l'état de l'interface du modèle (c'est-à-dire ses sorties). Nous détaillons la façon dont nous décrivons les modèles de calcul dans ModHel'X dans la section 4.5.

L'obtention d'une observation d'un modèle en combinant les observations des blocs qui le composent selon les règles définies par un MoC se fait quel que soit le niveau du modèle considéré dans la hiérarchie du modèle d'un système. Lorsqu'un bloc ayant un modèle interne se voit demander une mise à jour de son comportement, il délègue en fait l'observation à son modèle interne et cette observation est calculée selon les règles définies dans le modèle de calcul associé à son modèle interne. Prenons l'exemple du modèle représenté sur la figure 4.3. Le modèle racine est composé de deux blocs A et B et implique le modèle de calcul $MoC_{ext/A\&B}$. Le bloc A a un modèle interne qui obéit au modèle de calcul $MoC_{int/A}$ qui est différent de $MoC_{ext/A\&B}$. Une observation du modèle racine est calculée en réalisant des observations des blocs A et B suivant les règles du modèle de calcul $MoC_{ext/A\&B}$. Lorsque le bloc A se voit demander une mise à jour de son comportement, il délègue le calcul de l'observation à son modèle interne. Cette

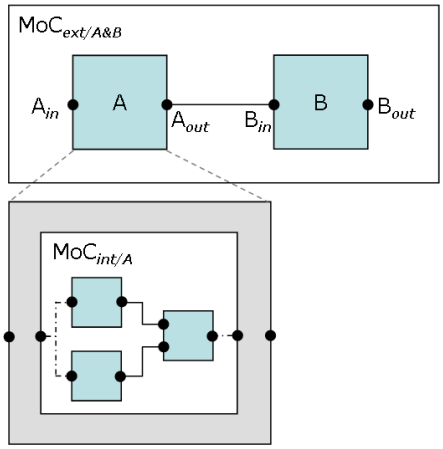


FIG. 4.3 – Hétérogénéité et observation hiérarchique

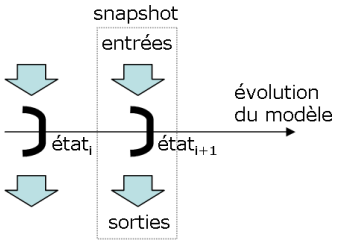


FIG. 4.4 – Exécution d'un modèle : série de snapshots

observation est calculée en réalisant des observations des blocs qui composent ce modèle interne suivant les règles du modèle de calcul $MoC_{int/A}$. Nous appelons *modèle hétérogène* un modèle dans lequel il existe différents niveaux hiérarchiques impliquant des modèles de calcul différents. Nous détaillons la façon dont le calcul d'une observation est propagé hiérarchiquement à travers le différents niveaux d'un modèle hétérogène dans la section 4.5.9.

La « glue » sémantique entre deux modèles impliquant des modèles de calcul différents est chargée d'adapter les mécanismes d'observation entre les deux modèles. Elle peut être vue en quelque sorte comme un adaptateur (voir la section 3.3.5.3) autour du modèle interne d'un bloc. La délégation du calcul de l'observation d'un bloc à son modèle interne passe par l'intermédiaire de cet adaptateur, ainsi que le résultat de l'observation une fois le calcul terminé. La modélisation explicite de cette notion de « glue » sémantique est l'une des principales différences entre notre approche et celle développée dans Ptolemy II. Nous détaillons la façon dont nous permettons la spécification du mécanisme d'adaptation dans la section 4.5.9.

L'observation du modèle racine de la hiérarchie, qui représente donc le système dans son ensemble, est appelée *snapshot* [CL85] car l'état observable (c'est-à-dire l'état de l'interface) de tous les blocs qui composent le modèle est alors exactement défini, et donc par extension celui du modèle également. Une exécution d'un modèle est donc une série de snapshots, comme représenté sur la figure 4.4.

Le principe de l'observation est un point clé de notre approche : ce principe permet de rendre notre approche indépendante de toute sémantique d'exécution. Nous pouvons ainsi embarquer des processus asynchrones dans un modèle sans les synchroniser, car nous nous contentons de les observer à des moments qui sont appropriés pour le modèle global (ces moments sont donc définis par le modèle de calcul qui lui est associé). Nous réalisons en fait un changement de paradigme d'exécution par rapport à Ptolemy, même si les concepts fondamentaux sont relativement similaires.

4.3.3 Modèles d'Exécution (MoEs)

Nous avons vu dans la section 3.3.2.3 que le terme modèle de calcul (MoC) désigne les éléments sémantiques de base sous-jacents à une classe de paradigmes de modélisation. Le modèle de calcul détermine des propriétés telles que :

Le mode de communication : le MoC définit sous quelle forme les informations transitent entre les éléments du modèle (événements, signaux, appels de méthodes, etc.) mais également par quel type de média ces informations circulent entre les éléments du modèle

(FIFO, canaux de taille limitée, etc.).

La concurrence : le MoC définit si les différents éléments du modèle fonctionnent en concurrence. La concurrence peut être explicite (l'utilisation d'un opérateur spécifique est nécessaire pour créer un nouveau flux de contrôle) ou implicite (le fait qu'un modèle contienne plus d'un élément implique que ceux-ci fonctionnent en concurrence). S'il y a concurrence entre les éléments du modèle, le modèle de calcul peut définir comment celle-ci est gérée (modèle synchrone, entrelacement, etc.). Le modèle de calcul peut également définir une relation d'ordre (causalité) totale ou partielle entre les différents éléments du modèle.

Le type de synchronisation : le MoC définit comment l'exécution des différents éléments du modèle est synchronisée (synchronisation par les données de manière synchrone, asynchrone, synchronisation par le contrôle).

Le type de temps : le MoC définit la façon dont est représentée et utilisée la notion de temps dans le modèle (absence de notion de temps, utilisation d'un ordre partiel/total sur des événements, existence d'une notion de distance entre événements, temps continu, etc.).

Nous remarquons qu'il peut exister plusieurs manières d'implémenter un même modèle de calcul. Prenons par exemple le modèle de calcul des équations différentielles. De telles équations peuvent être utilisées pour représenter le comportement en temps continu d'un système. Il existe différentes méthodes, comme celles d'Euler et de Runge-Kuta par exemple, pour résoudre numériquement un système d'équations différentielles, c'est-à-dire trouver une fonction qui les vérifie. Si l'on souhaite implémenter le modèle de calcul des équations différentielles dans Ptolemy II par exemple, il faudra décrire un directeur dont le rôle est de résoudre le système d'équation décrit dans le modèle auquel il est associé afin de trouver la fonction qui représente le comportement du système. Un tel directeur peut alors être implémenté soit selon la méthode de résolution d'Euler soit selon celle de Runge-Kuta. Nous concluons que, de manière générale, les règles définies par un modèle de calcul ne sont en fait pas nécessairement directement exécutables.

Proposition 4. *Nous appelons* **Modèle d'Exécution (Model of Execution – MoE)** *une implémentation exécutable d'un modèle de calcul.*

Un modèle d'exécution est donc en quelque sorte un algorithme chargé d'appliquer les règles définies par un modèle de calcul. En ce sens, il existe une relation d'abstraction/raffinement entre un modèle de calcul et un modèle d'exécution : le comportement obtenu en exécutant un modèle selon son modèle d'exécution doit être conforme au comportement défini par les règles du modèle de calcul.

Comme dans notre approche nous nous intéressons en particulier à l'exécution des modèles (pour la simulation notamment), nous manipulons en fait des modèles d'exécution plutôt que des modèles de calcul. Cependant, pour la clarté et la concision de ce mémoire, nous utiliserons par abus de langage le terme modèle de calcul ou MoC.

4.3.4 Aspects temporels et synchronisation

Ainsi que cela est illustré sur la figure 4.5, adaptée de [Lee97], les modèles de temps utilisés dans différents modèles de calcul peuvent être très variés : temps réel, horloges logiques, ordre partiel sur des échantillons de signal, etc. Notre approche doit non seulement les supporter tous mais également supporter leur combinaison dans un même modèle. Pour cela, il est nécessaire que notre approche comporte des outils permettant (1) de spécifier un modèle de temps dans la description d'un modèle de calcul et (2) de décrire comment deux modèles de temps dans deux modèles de calcul peuvent être mis en relation.

Nous avons choisi la succession des snapshots représentant l'exécution d'un modèle comme abstraction sous-jacente pour l'expression de modèles de temps. En effet, c'est le seul élément de

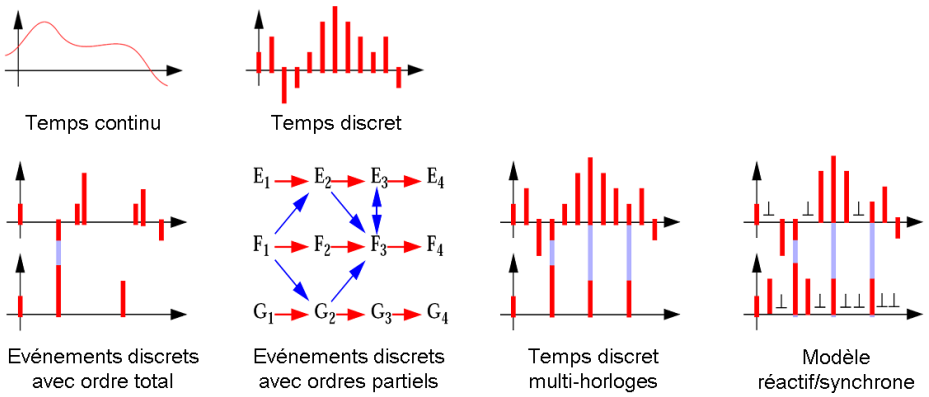


FIG. 4.5 – Différents modèles de temps (source : [Lee97])

« cadencement » qui soit partagé par les différents modèles de calcul d'un modèle dans notre approche. Dans ModHel'X, le modèle de temps est donc représenté, pour chaque modèle de calcul, sous la forme d'une étiquette temporelle sur la séquence des snapshots. Il s'agit une approche volontairement discrète car nous nous intéressons à l'exécution (numérique) des modèles.

Un snapshot doit pouvoir être réalisé non seulement dès que l'environnement, c'est-à-dire les données d'entrée du modèle changent, mais également dès que le temps change dans l'un des modèles de la hiérarchie. Afin de provoquer un snapshot, les blocs ou les MoCs peuvent produire des contraintes sur leur date au prochain snapshot. L'ensemble des contraintes émises pour l'ensemble des niveaux hiérarchiques du modèle détermine, avec les données d'entrée, le moment auquel le prochain snapshot doit être réalisé.

Illustrons ce principe sur l'exemple des automates temporisés. Dans un automate temporisé il peut exister des transitions, dites temporisées, qui sont automatiquement déclenchées lorsqu'un délai expire. Dans le modèle de calcul des automates temporisés, le temps peut être représenté par des dates sous la forme d'entiers par exemple, permettant d'étiqueter les snapshots localement. Une règle du modèle de calcul peut alors spécifier que, lors de l'entrée dans un état d'où part une transition temporisée, une contrainte est émise de manière à ce que le prochain snapshot ait lieu au plus tard au moment où la temporisation de la transition expire pour que ce comportement soit effectivement observé si aucune autre transition n'est provoquée par les entrées du modèle.

Les modèles de temps de deux modèles de calcul utilisés dans deux niveaux hiérarchiques voisins peuvent être synchronisés grâce au mécanisme d'adaptation présent à la frontière entre les deux modèles. Les contraintes peuvent également être propagées à travers les niveau hiérarchiques jusqu'au modèle racine. La modélisation du temps sur la base de la séquence des snapshots et l'utilisation de contraintes démarquent notre approche de celle de Ptolemy, dans laquelle le modèle racine pilote l'exécution des autres niveaux de la hiérarchie. Nous verrons en détail comment cela est réalisé en pratique dans notre approche dans la section 4.5.10.

4.3.5 Architecture globale de l'approche ModHel'X

La figure 4.6 illustre le principe de fonctionnement de notre approche. Pour qu'un modèle hétérogène puisse être interprété et exécuté dans notre approche, il est nécessaire :

- de décrire la structure du modèle sous la forme d'un ensemble de niveaux imbriqués hiérarchiquement ;
- de spécifier de manière exécutable les différents modèles de calcul en jeu dans les différents niveaux hiérarchiques du modèle ;
- de spécifier la façon dont les différents niveaux hiérarchiques hétérogènes interagissent à leurs frontières par l'intermédiaire de « glues » sémantiques.

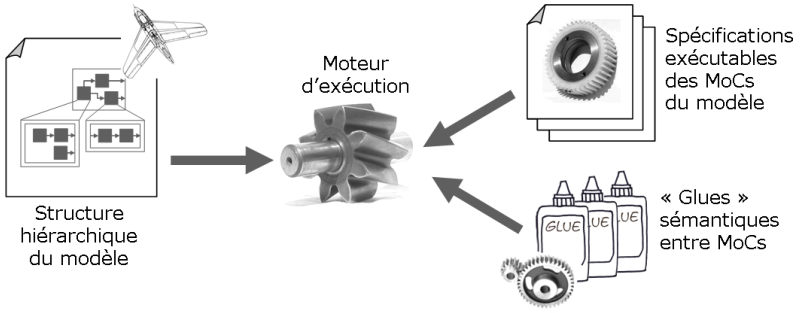


FIG. 4.6 – Principe de fonctionnement de ModHel'X

Pour réaliser ces différentes étapes, nous proposons d'une part une syntaxe abstraite pour représenter la structure des modèles, et d'autre part une sémantique abstraite et un langage pour décrire les modèles de calcul et les mécanismes de composition de modèles qui permettent de calculer le comportement de telles structures. L'architecture logique de notre approche est représentée sur la figure 4.7.

Pour représenter la structure des modèles comme exposé dans la section 4.3.1, nous avons tout d'abord mis au point une syntaxe abstraite. Nous avons formalisé cette syntaxe sous la forme d'un méta-modèle décrit dans le MOF [OMGf], de manière à pouvoir bénéficier notamment des techniques de transformation de modèles (voir la section 3.3.3.1). Cette syntaxe est inspirée de celle de Ptolemy mais nous proposons des modifications de manière à permettre la description explicite des mécanismes de composition de modèles.

Puis nous avons développé une « sémantique abstraite » [LZ07] servant de base pour la spécification des modèles de calcul. Cette « sémantique abstraite » se présente sous la forme d'un algorithme dont le rôle est d'appliquer à un modèle dont la structure est conforme à notre syntaxe abstraite les règles correspondant à un modèle de calcul. Les étapes de cet algorithme sont « abstraites » c'est-à-dire que les détails de leur exécution restent à définir. Cet algorithme peut donc être vu comme un modèle d'exécution générique, le terme modèle d'exécution étant entendu dans le sens défini dans la section 4.3.3. Il faut donc spécialiser ce modèle d'exécution et en décrire les étapes pour obtenir un modèle d'exécution concret qui corresponde à un modèle de calcul donné.

Enfin, nous avons défini les spécifications d'un langage permettant de décrire les étapes de notre modèle d'exécution générique. Ce langage est également utilisé pour décrire les mécanismes de composition entre modèles hétérogènes. C'est un langage impératif à effets de bord, car il doit permettre de décrire des exécutions et donc de décrire comment l'état d'un modèle est modifié au cours de l'exécution.

Nous détaillons ces éléments de notre approche dans les sections suivantes et illustrons l'application de notre méthode par l'implémentation d'un framework et la réalisation d'un cas d'étude dans le chapitre suivant.

4.4 Représentation des modèles hétérogènes : syntaxe abstraite générique

Suivant les paradigmes de modélisation, les éléments utilisés pour représenter un modèle sont très différents les uns des autres. Par exemple, dans le paradigme des machines à états finis, les concepts utilisés pour décrire un système sont les « états » et les « transitions » tandis que dans le paradigme SDF (Synchronous DataFlow), les concepts utilisés pour décrire un système sont les « processus », les « canaux de communication », les « signaux », etc. Dans ModHel'X nous devons permettre de représenter un modèle dans n'importe quel paradigme. Pour cela nous

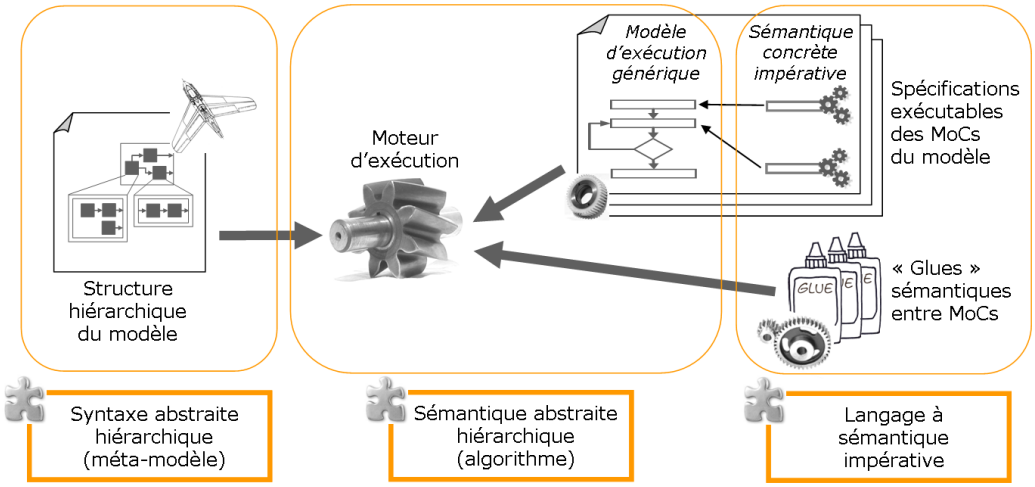


FIG. 4.7 – Architecture logique de ModHel'X

avons mis au point un ensemble de concepts de modélisation abstraits très élémentaires, dénotés par les atomes d'une syntaxe abstraite générique. Nous représentons cet ensemble de concept sous la forme d'un méta-modèle. Ces concepts ont une sémantique générique, qui devra être spécialisée dans la spécification du modèle de calcul correspondant au paradigme choisi pour permettre la représentation de concepts spécifiques à ce paradigme.

Nous décrivons ces différents concepts dans les sections suivantes et donnons une vue globale de notre méta-modèle dans la section 4.4.6. Nous décrivons comment ces concepts peuvent être spécialisés dans la section 4.6.1.

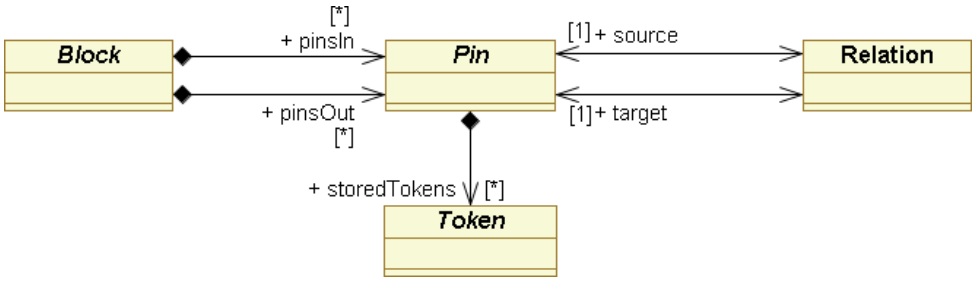
4.4.1 Blocs, points d'interface, relations et jetons

Comme nous l'avons introduit en préambule, nous adoptons une approche orientée composants, c'est-à-dire que le système est décomposé en éléments qui réalisent une ou plusieurs fonctions et qui, par leur coopération, réalisent la fonctionnalité globale du système. Rappelons que, dans notre approche, nous considérons les composants comme des boîtes noires. Nous appelons un composant boîte noire un *bloc (Block)*.

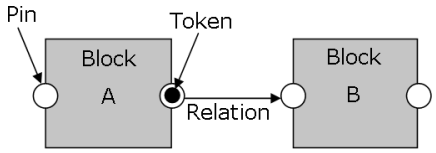
Comme les blocs sont des boîtes noires, tout ce qu'il est possible d'observer de leur comportement est ce qui est visible à leur interface. Nous modélisons l'interface d'un bloc par un ensemble de « points d'interface ». Un *point d'interface (Pin)* définit ce qui est observable du comportement du bloc – il joue alors le rôle de sortie – mais il peut également définir ce dont le bloc a besoin pour pouvoir donner une observation de son comportement – il joue alors le rôle d'entrée. Un point d'interface peut représenter aussi bien la notion de port au sens classique du terme que la notion de nœud de contrôle par exemple.

Nous représentons les informations qu'il est possible d'observer à l'interface d'un bloc par des « jetons ». Un *jeton (Token)* représente donc de manière abstraite un échantillon d'information. La quantité ainsi que le type d'information représenté par un jeton dépend du modèle de calcul considéré. Un jeton peut représenter aussi bien des échantillons de données que des événements ou des jetons de contrôle (comme dans les réseaux de Petri par exemple). Lorsqu'ils sont déposés sur les points d'interface des blocs, les jetons peuvent être pris en compte par les blocs lors de la mise à jour de leur interface pour l'observation. Le type de persistance des jetons sur les points d'interface (le fait qu'un jeton reste présent sur un point d'interface après avoir été lu par un bloc pour sa mise à jour, par exemple) dépend à la fois du comportement des blocs et du modèle de calcul considéré.

Dans une approche à base de composants, les composants sont mis en relation les uns avec



(a) Extrait du méta-modèle : blocs, points d'interface, relations et jetons



(b) Deux blocs interconnectés

FIG. 4.8 – Blocs, points d'interface, relations et jetons

les autres pour former le système. Cette mise en relation peut avoir deux aspects : un aspect structurel et un aspect comportemental. En effet, les relations entre les composants peuvent dénoter des éléments de structure physique du système, comme des fils de cuivre par exemple. Dans d'autres paradigmes, les relations entre les composants peuvent dénoter un mode de communication entre les composants, comme la synchronisation par rendez-vous par exemple. Enfin, elles peuvent également dénoter le flot de contrôle entre les composants, comme c'est le cas dans les réseaux de Petri pour les relations entre les places. La sémantique qui est donnée aux relations existant entre les composants dépend en fait complètement du paradigme de modélisation. Comme nous cherchons à représenter des modèles quel que soit le paradigme utilisé, nous ne conservons dans notre syntaxe abstraite que la notion de « mise en relation », indépendamment de toute notion de structure physique ou de mode d'interaction.

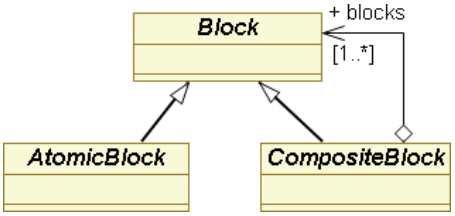
Une *relation (Relation)* permet donc simplement de structurer une description d'un système en mettant en relation deux blocs via leurs points d'interface. Les relations sont unidirectionnelles et ont donc un point d'interface source et un point d'interface cible. Elles peuvent représenter tout type d'interaction (contrôle et données) entre deux blocs : canal de communication, rendez-vous, transition, etc.

La figure 4.8a montre la façon dont sont représentés ces concepts dans notre méta-modèle. La figure 4.8b illustre l'utilisation de ces concepts sur un exemple simple composé de deux blocs A et B. La syntaxe graphique utilisée ici est une syntaxe simple relativement intuitive que nous utiliserons pour la plupart de nos exemples mais qui ne constitue pas une syntaxe concrète définitive pour notre approche. Nous représentons les blocs par des carrés, les points d'interface par des cercles, les jetons par des disques et les relations par des flèches.

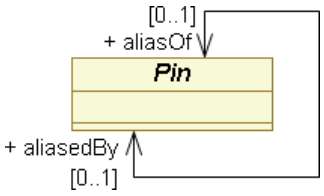
4.4.2 Blocs atomiques et blocs composites

Comme nous l'avons vu dans la section 3.3.5, l'un des avantages d'une approche orientée composants est qu'il est possible, avec des composants interconnectés, de former de nouveaux composants dits « composites ». Pour que cela soit également possible avec notre syntaxe abstraite, nous appliquons à notre méta-modèle le patron composite [GHJV95]. Ce patron permet de manipuler un ensemble de blocs interconnectés de la même manière qu'un bloc. Il permet donc d'encapsuler un ensemble de blocs interconnectés dans un bloc, c'est-à-dire d'introduire

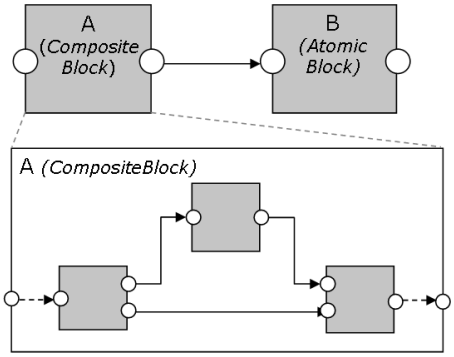
de la hiérarchie dans notre syntaxe abstraite. Nous introduisons donc dans notre méta-modèle deux types de blocs : les blocs atomiques, qui constituent les feuilles du patron composite, et les blocs composites. La figure 4.9a illustre l'application de ce patron sur notre méta-modèle.



(a) Blocs atomiques et blocs composites



(b) Points d'interface et alias



(c) Exemples de blocs atomique et composite

FIG. 4.9 – Blocs atomiques et blocs composites

Un *bloc atomique (AtomicBlock)* est une « vraie » boîte noire. Le comportement d'un bloc atomique est décrit dans un langage externe à notre approche, comme du C ou de l'Esterel par exemple. Conceptuellement, un bloc atomique peut être considéré comme un observateur construit autour d'un programme qui n'est pas nativement supporté par notre approche. Son rôle est de donner une observation du comportement de ce programme lorsque cela lui est demandé. Les blocs atomiques sont les « briques de base » dans la description d'un comportement. Notre approche vient avec une bibliothèque de blocs atomiques (l'annexe C.3 présente l'implémentation de quelques blocs atomiques).

Un *bloc composite (CompositeBlock)* est un bloc construit à partir d'autres blocs interconnectés les uns avec les autres. La notion de bloc composite telle que nous l'utilisons ici constitue une facilité syntaxique qui peut être comparée à la notion de macro : un bloc composite est une structure qui peut être réutilisée dans différents modèles et qui est équivalente à la recopie pure et simple de son contenu à l'endroit où elle est utilisée.

Comme un bloc composite est un bloc, il dispose également d'une interface composée de points d'interface par lesquels il peut être connecté à d'autres blocs. La particularité des points d'interface d'un bloc composite est qu'ils correspondent en fait à des points d'interface de certains des blocs qui le composent. Les points d'interface du composite permettent en quelque sorte de rendre visible à l'extérieur les points d'interface des blocs qu'il encapsule. Ainsi une entrée du composite correspond à une entrée de l'un des blocs qui le composent, et respectivement une sortie d'un des blocs qui le composent correspond à une sortie du composite. Ces points d'interface sont en fait des « alias » les uns des autres. Cette notion est représentée par une relation d'alias entre deux points d'interface dans notre méta-modèle, comme cela est illustré

sur la figure 4.9b.

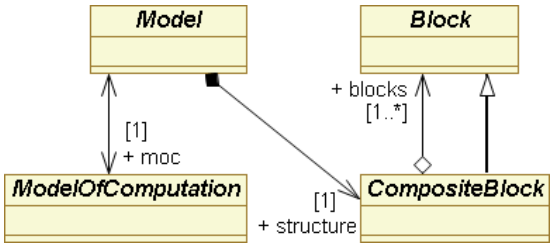
Le bloc *A* de la figure 4.9c est un bloc composite dont la structure est représentée dans le rectangle placé en dessous. Le bloc *A* pourrait être purement et simplement remplacé par cette structure dans le modèle dans lequel il est utilisé. Les relations d'alias entre les points d'interface du composite et les points d'interface de ses blocs internes sont représentées par des flèches en pointillés.

4.4.3 Modèles et modèles de calcul

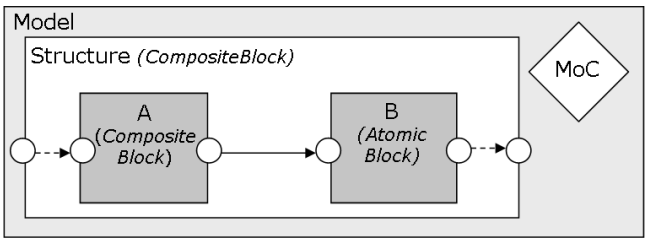
Telle quelle, une structure à base de blocs (comme un bloc composite par exemple) n'est pas interprétable puisque les éléments dont elle est constituée n'ont pas de sémantique définie. Pour rendre une structure à base de blocs interprétable, il faut lui associer un modèle de calcul. Cette structure représente alors une description du comportement d'un système, c'est-à-dire un modèle. Dans notre syntaxe abstraite, un *modèle (Model)* est donc composé d'une structure et d'un modèle de calcul, chargé de donner la sémantique de la structure. La structure d'un modèle est représentée par un bloc composite.

Comme nous l'avons introduit dans les sections 4.3.3 et 4.3.2, le *modèle de calcul (ModelOfComputation)* définit la façon dont est interprété le modèle lors de l'exécution. Le modèle de calcul orchestre les observations des différents blocs composant la structure du modèle et les combine pour obtenir une observation du modèle.

La figure 4.10a montre un extrait de notre méta-modèle illustrant ces concepts. Notre méta-modèle spécifie ainsi qu'à chaque instance du concept *Model* doit être associée une instance du concept *CompositeBlock* et une instance du concept *ModelOfComputation*. La figure 4.10a illustre l'utilisation de ces concepts sur un modèle très simple constitué des blocs *A* et *B* de la figure 4.9c.



(a) Modèle, bloc composite et modèle de calcul



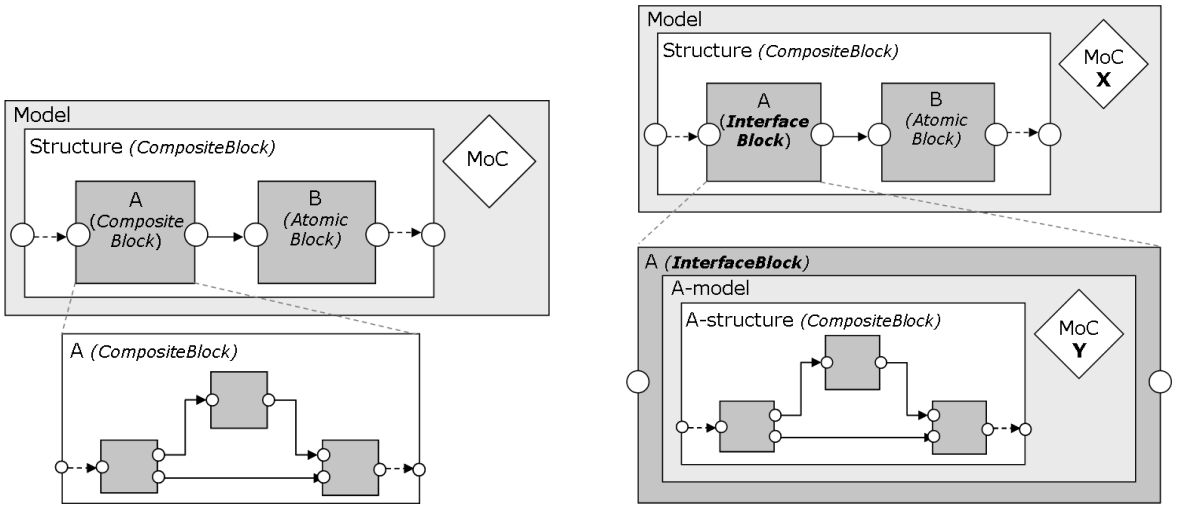
(b) Exemple de modèle

FIG. 4.10 – Modèle : structure (bloc composite) et modèle de calcul

4.4.4 Hiérarchie et hétérogénéité : blocs d'interface

Nous avons vu que la notion de bloc composite permettait d'introduire la hiérarchie dans notre syntaxe abstraite. Lorsqu'un bloc composite est placé dans la structure d'un modèle,

comme c'est le cas sur la figure 4.11a, il est interprété par le modèle de calcul associé à la structure du modèle. La hiérarchie introduite par les blocs composites ne permet donc pas l'hétérogénéité des modèles de calcul dans les différents niveaux hiérarchiques d'un modèle.



(a) Hiérarchie homogène : bloc composite

(b) Hiérarchie hétérogène : bloc d'interface

FIG. 4.11 – Hiérarchie et hétérogénéité : blocs d'interface

Pour cela, nous introduisons la notion de *bloc d'interface (InterfaceBlock)* dans notre syntaxe abstraite. De la même façon qu'un bloc composite permet de manipuler un ensemble de blocs comme un bloc, un bloc d'interface permet de manipuler un modèle comme un bloc. Nous enrichissons donc le patron du composite en permettant, ainsi que cela est illustré sur la figure 4.11b, de combiner hiérarchiquement deux modèles. Comme un bloc d'interface est un bloc, il est donc considéré comme une boîte noire du point de vue du modèle dans lequel il est utilisé : son modèle interne reste caché contrairement à ce qui se passe dans le cas d'un bloc composite (voir la section précédente). Ainsi, comme cela est illustré sur la figure 4.11b, les modèles de calcul associés aux deux modèles peuvent être différents. Un *bloc d'interface* est donc un bloc dont le comportement est décrit par un modèle ModHel'X interne, qu'il encapsule et dont il adapte la sémantique vis-à-vis du modèle externe.

La notion de bloc d'interface a un rôle très important dans notre approche car non seulement un bloc d'interface permet l'hétérogénéité hiérarchique mais de plus il joue le rôle d'adaptateur entre le modèle dans lequel il est utilisé et son modèle interne. Pour pouvoir combiner les sémantiques de ces deux modèles, le bloc d'interface a accès à leurs deux modèles de calcul : il a accès directement au modèle de calcul associé au modèle dans lequel il est utilisé et il a accès au modèle de calcul associé à son modèle interne par l'intermédiaire de son modèle interne. Nous détaillons la façon dont l'adaptation est réalisée en pratique par les blocs d'interface dans la section 4.5.9.

4.4.5 Paramètres

Les points d'interface permettent de représenter ce qui est observable du comportement des blocs et, par extension, des modèles (via leur structure) pendant l'exécution. Or, dans les approches orientées composants, le comportement d'un composant est souvent rendu paramétrable de manière à pouvoir être facilement adapté à différents contextes de réutilisation. Nous soulignons ici que la notion de paramètre est différente de la notion de point d'interface. En effet, la valeur d'un paramètre peut généralement être considérée comme constante lors de l'exécution

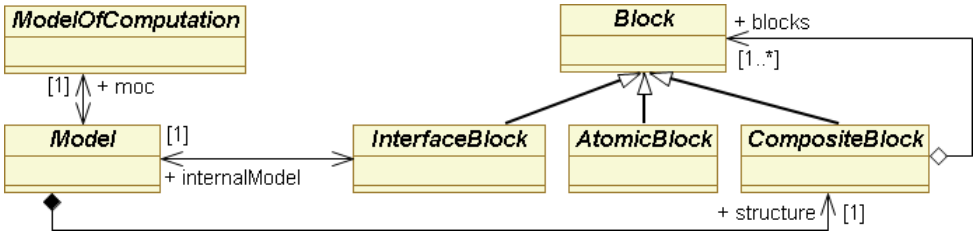


FIG. 4.12 – Représentation de la notion de bloc d'interface dans notre méta-modèle

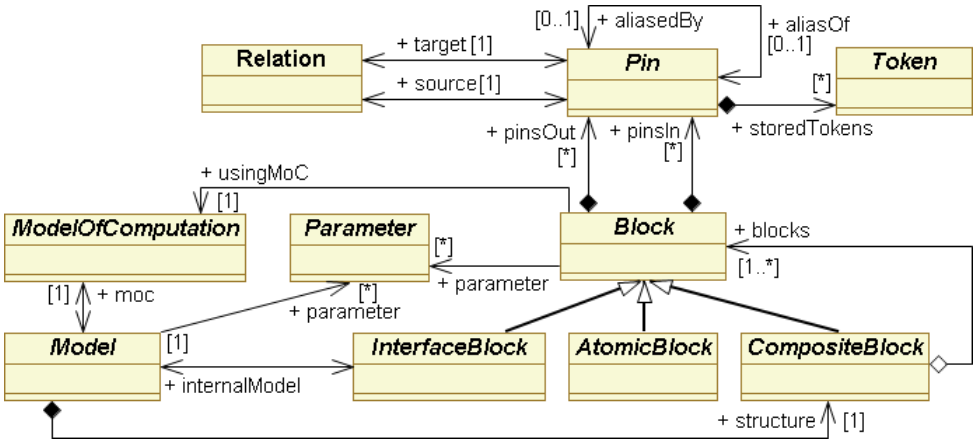


FIG. 4.13 – Syntaxe abstraite de ModHel'X (méta-modèle générique complet)

alors que les informations observées sur les points d'interface varient dynamiquement au cours de l'exécution puisqu'elles sont représentatives du comportement du système modélisé. Nous prenons en compte cette notion de *paramètre (Parameter)* dans notre syntaxe abstraite.

4.4.6 Récapitulatif : meta-modèle complet

Le méta-modèle complet correspondant à la syntaxe abstraite générique de ModHel'X est représenté sur la figure 4.13. Cette syntaxe abstraite constitue un élément central de notre approche car elle nous permet de représenter la structure des modèles de manière homogène, ce qui rend la gestion de l'hétérogénéité plus simple, comme nous l'avons vu dans la section 2.5.4. Les éléments de notre syntaxe abstraite constituent donc un support sur lequel l'hétérogénéité est exprimée par l'intermédiaire des modèles de calcul. En effet, le modèle de calcul est chargé d'interpréter toute description structurelle de modèle conforme à notre méta-modèle de manière fidèle à la sémantique donnée par le paradigme de modélisation choisi. De manière à faciliter la description de modèles de calcul dans notre approche, nous proposons un cadre générique pour la spécification de la sémantique de tout modèle d'exécution. C'est ce cadre générique que nous présentons dans la section suivante.

4.5 Spécification exécutable de MoCs : sémantique abstraite générique (modèle d'exécution générique)

Dans notre approche nous nous intéressons tout particulièrement à l'exécution des modèles hétérogènes. Exécuter un modèle hétérogène dans notre contexte nécessite de disposer de descriptions exécutables des modèles de calcul impliqués dans le modèle (donc de modèles d'exécution). Nous proposons de décrire un modèle de calcul sous la forme d'un algorithme.

Les modèles de calcul étant très différents les uns des autres, leurs modèles d'exécution, et

donc les algorithmes les décrivant, peuvent être très différents. Tirant parti de la syntaxe abstraite générique que nous avons définie, nous proposons un algorithme d'exécution générique capable d'appliquer les règles d'un modèle de calcul à toute structure conforme à notre méta-modèle. Cet algorithme d'exécution générique définit un cadre pour la spécification des modèles de calcul (en fait, de modèles d'exécution) et constitue donc un modèle d'exécution générique. Les étapes de cet algorithme sont abstraites et doivent être être complétées avec les instructions d'exécution spécifiques à un modèle de calcul pour obtenir un modèle d'exécution concret. Ce modèle d'exécution générique peut être considéré comme définissant une « sémantique abstraite » (au sens de [LZ07]) car la sémantique qu'il définit sous la forme d'un algorithme doit être spécialisée et rendue concrète pour décrire un modèle de calcul.

Nous décrivons l'ensemble de notre algorithme générique dans cette section. Nous décrivons comment cet algorithme peut être utilisé pour décrire un modèle de calcul particulier dans la section 4.6.

4.5.1 Boucle de déclenchement des snapshots

Comme nous l'avons introduit dans la section 4.3.2, notre approche est basée sur le principe de l'observation. L'exécution d'un modèle est constituée d'une série d'observations du modèle appelées « snapshots ». Notre algorithme contient donc tout d'abord une boucle permettant d'effectuer la série de snapshots nécessaire à l'exécution d'un modèle.

Cette boucle de réalisation des snapshots est représentée sur la figure 4.14. L'exécution d'un modèle commence par une étape appelée `setup` et termine par une étape appelée `wrapup`. L'étape de `setup` est une étape de pré-initialisation qui comporte notamment la vérification de la conformité de la structure du modèle par rapport au méta-modèle et aux contraintes éventuelles liées au modèle de calcul. L'étape de `wrapup` est l'étape finale de l'exécution, durant laquelle les résultats de la simulation peuvent être affichés et les ressources allouées au calcul libérées.

L'étape permettant de déterminer si le calcul d'un nouveau snapshot est nécessaire est appelée `newSnap`. Les conditions déterminant si un nouveau snapshot est nécessaire dépendent de paramètres spécifiés par le concepteur lorsqu'il souhaite exécuter le modèle. Si ces conditions sont remplies alors un nouveau snapshot est calculé. Sinon, l'exécution termine.

Les étapes de calcul d'un snapshot sont détaillées dans les sections suivantes.

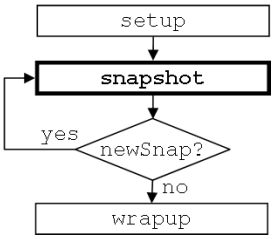


Fig. 4.14 – Boucle de calcul d'une série de snapshots

4.5.2 Déterminisme du calcul d'un snapshot

Comme nous l'avons introduit dans la section 4.3.2, calculer un snapshot d'un modèle requiert de réaliser des observations de chacun des blocs qui le composent. Le modèle de calcul est chargé de réaliser ces observations de manière à ce que l'observation résultante soit fidèle à la sémantique du paradigme de modélisation choisi. Afin de ne pas introduire d'incertitudes sur la sémantique des modèles de calcul que nous allons décrire dans notre approche, nous souhaitons que notre algorithme soit déterministe. Nous choisissons de réaliser les observations des blocs qui composent le modèle *séquentiellement*.

Rappelons tout d'abord que, quelle que soit sa nature (atomique, composite ou d'interface), un bloc fournit une observation de son comportement, c'est-à-dire met à jour son interface, lorsque cela lui est demandé (voir la section 4.3.2). Pour réaliser un snapshot, le modèle de calcul doit donc demander successivement aux blocs qui composent le modèle une mise à jour de leur interface. Ces observations successives doivent cependant former un snapshot c'est-à-dire une observation « instantanée » du modèle complet. Le problème de la consistance d'une telle observation globale est similaire au problème de la définition de l'état d'un système distribué [CL85], à la différence que :

- le calcul du snapshot est effectué de manière centralisée par le modèle de calcul ;
- nous nous plaçons dans le contexte de l'exécution de modèles, et plus précisément de la simulation. En conséquence, le temps qui s'écoule dans le référentiel du modèle est sans rapport avec le temps qui s'écoule dans le monde réel (c'est-à-dire le temps par rapport auquel peut être mesuré la durée du calcul d'un snapshot).

Si nous transposons les résultats décrits dans [CL85] au problème du calcul de l'observation globale d'un modèle par un modèle de calcul, nous obtenons les contraintes suivantes :

1. Pour que le snapshot ne soit pas « flou », il est impératif que les observations locales des blocs aient lieu à des moments suffisamment proches dans le temps du modèle mais sans perturber l'exécution. Pour tenir compte de cette contrainte dans notre algorithme, nous utilisons le fait que, dans notre contexte, le temps du modèle est déconnecté du temps du calcul. Il nous est donc possible de « figer » le contexte dans lequel le calcul du snapshot est effectué. Nous décrivons ce mécanisme dans la section 4.5.3.
2. Pour que le snapshot ait un sens, les observations locales des blocs doivent être réalisées dans un ordre tel que les notions de causalité, de concurrence, etc. du paradigme considéré soient respectées. Pour tenir compte de cette contrainte, il est nécessaire que notre algorithme permette d'ordonnancer les observations des blocs selon la sémantique du paradigme considéré. Nous décrivons comment cela est réalisé dans la section 4.5.4.

4.5.3 Contexte de calcul d'un snapshot

4.5.3.1 Gel du contexte de calcul

Afin que le snapshot calculé ne soit pas « flou », nous figeons le contexte dans lequel celui-ci a lieu. Le contexte de calcul d'un snapshot est constitué d'une part des données représentant l'état de l'environnement du système au moment où le snapshot est calculé (c'est-à-dire les données d'entrée du système) et d'autre part de l'état dans lequel se trouvent les blocs composant le modèle à ce même moment.

Pour ce faire, le calcul d'un snapshot commence dans notre algorithme par une étape appelée `startOfSnapshot`, comme cela est illustré sur la figure 4.15. C'est au cours de cette étape que les données représentant l'environnement au moment du snapshot sont acquises. Ces données sont figées pour toute la durée du snapshot (durée qui existe dans le temps « réel » mais qui est nulle dans le temps du modèle).

C'est également au cours de cette étape que les blocs composant le modèle doivent se figer dans leur état courant. Comme nos blocs sont des boîtes noires, nous choisissons de les « prévenir » qu'ils vont être observés afin qu'ils figent eux-même leur état. Lors de l'étape `startOfSnapshot`, tous les blocs sont donc prévenus de l'imminence du snapshot de manière à pouvoir fournir une observation de leur comportement au même moment du point de vue de l'observateur (c'est-à-dire le modèle de calcul), où qu'ils en soient dans leurs exécutions respectives.

De manière symétrique, le calcul d'un snapshot termine par une étape appelée `endOfSnapshot` pendant laquelle les données représentant l'observation du modèle qui vient d'être calculée sont

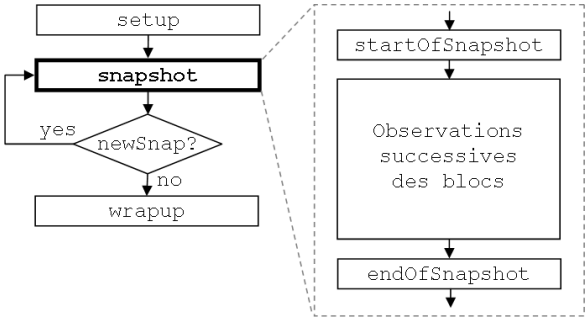


FIG. 4.15 – Gel du contexte de calcul d'un snapshot : `startOfSnapshot` et `endOfSnapshot`

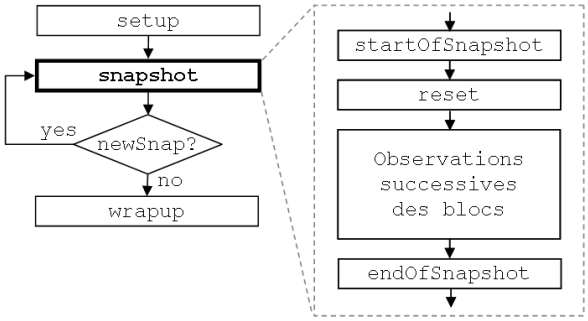


FIG. 4.16 – Initialisation des variables de calcul d'un snapshot : `reset`

rendues visibles vis-à-vis de l'environnement et les blocs sont prévenus de la fin du snapshot et peuvent de nouveau faire évoluer leur état.

4.5.3.2 Initialisation des variables de calcul

Pendant le calcul du snapshot, c'est-à-dire pendant le calcul des observations successives des blocs constituant le modèle, le modèle de calcul peut avoir besoin de stocker des informations intermédiaires. Une étape d'initialisation de ces variables de calcul est donc nécessaire au début de chaque snapshot. Cette initialisation est effectuée par une opération appelée `reset` et peut notamment dépendre des données acquises sur l'environnement lors du `startOfSnapshot`.

4.5.4 Observations successives des blocs : ordonnancement et propagation

Comme nous l'avons introduit ci-dessus, pour calculer un snapshot d'un modèle qui ait un sens, le modèle de calcul doit observer successivement les blocs qui le composent dans un ordre tel que les notions de causalité, de concurrence, etc. du paradigme de modélisation considéré soient respectées. Prenons l'exemple d'un langage de modélisation basé sur le modèle de calcul Synchronous DataFlow (SDF) pour illustrer le choix de l'ordre dans lequel observer les blocs d'un modèle.

Dans le modèle de calcul SDF, des processus échangent, de manière synchrone les uns par rapport aux autres, des paquets de données sur des canaux de communication. L'existence d'un canal de communication entre une sortie d'un processus A et une entrée d'un processus B induit une dépendance causale entre ces deux processus car le processus B ne peut effectuer son calcul que lorsque A a effectué le sien (les données transitant de manière instantanée sur le canal de communication). Dans notre syntaxe abstraite, les processus peuvent être représentés par des blocs et les canaux de communication par des relations. L'observation d'un processus permet d'obtenir une observation des données produites sur ses sorties à l'instant de l'observation.

Pour commencer à calculer un snapshot, le modèle de calcul SDF doit tout d'abord choisir un premier bloc à observer. Le premier bloc observable est dans ce cas un processus qui ne dépend d'aucun autre processus, mais qui peut dépendre des entrées du modèle, qui ont été acquises lors de l'étape `startOfSnapshot`. Il s'agit dans ce cas du premier bloc selon l'ordre topologique établi par les canaux de communication. Le modèle de calcul demande alors à ce bloc une mise à jour de son interface. Cette mise à jour permet d'obtenir une observation des données produites sur ses sorties. Ces données doivent alors être transmises aux blocs qui dépendent du bloc courant afin qu'ils soient en mesure de mettre à jour leur interface à leur tour. Le modèle de calcul choisit alors le bloc suivant à observer selon l'ordre topologique, c'est-à-dire un bloc qui ne dépend que du bloc précédent et éventuellement des entrées du modèle. Il lui demande alors de mettre à

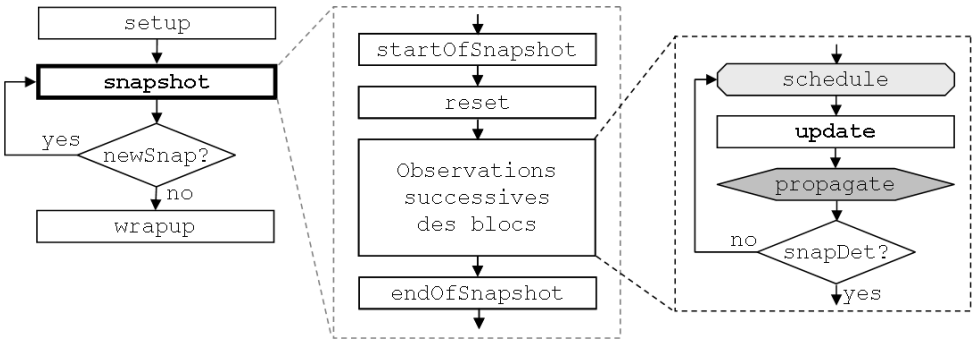


FIG. 4.17 – Boucle pour les observations successives des blocs dans un snapshot

jour son interface, etc.

Sur cet exemple, nous voyons que l'étape d'observation d'un bloc peut être décomposée en trois tâches effectuées en boucle, représentées sur la figure 4.17 :

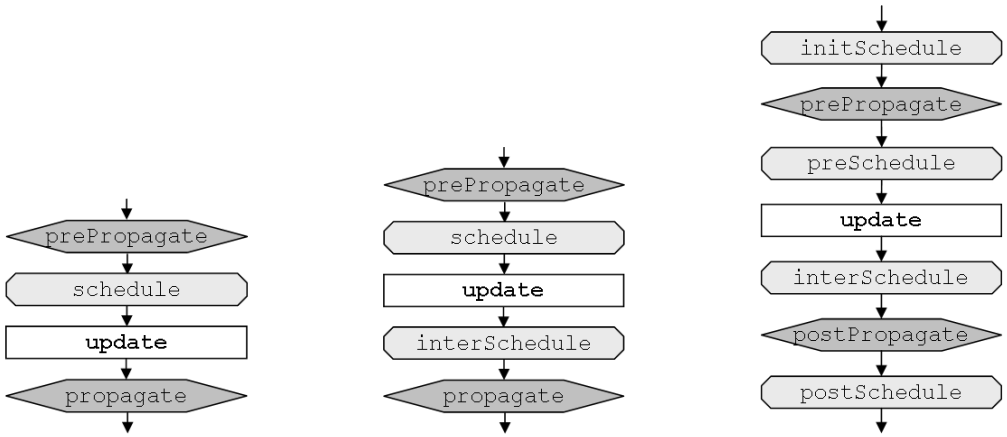
1. Choix du bloc à observer : cette tâche est appelée `schedule`. Dans notre exemple avec le modèle de calcul SDF, cette opération doit rendre le premier bloc suivant un ordre topologique.
2. Observation du bloc (demande de sa mise à jour) : cette tâche est appelée `update`. La mise à jour à proprement parler est réalisée par le bloc lui-même.
3. Propagation du résultat de l'observation pour pouvoir observer le bloc suivant : cette tâche est appelée `propagate`. Dans notre exemple avec le modèle de calcul SDF, cette opération recopie les données observées sur les points d'interface du bloc courant sur les points d'interface des blocs cibles des canaux de communication (représentés par des relations).
4. Répétition des étapes précédentes jusqu'à ce que le snapshot soit déterminé : cette tâche est appelée `snapDet`. Nous discutons de cette notion de « détermination » d'un snapshot et donc de l'arrêt de la boucle d'observation des blocs dans la section 4.5.6.

Ces étapes constituent la boucle de base d'observation des blocs pour le calcul d'un snapshot.

Remarquons que l'effet des opérations `schedule` et `propagate` dépend en fait du modèle de calcul considéré. Dans le cas d'un modèle de calcul dans lequel les composants s'exécutent concurremment, l'ordonnancement modélise la nature de la concurrence et des mécanismes de synchronisation du modèle de calcul. Ainsi, dans un modèle de calcul tel que CSP (Communicating Sequential Processes) dans lequel les processus communiquent par rendez-vous, l'ordre d'observation des blocs ne sera pas un simple ordre topologique comme dans le modèle de calcul SDF. Respectivement, la propagation des observations modélise la nature des mécanismes de communication du modèle de calcul. Ainsi, dans un modèle de calcul tel que DE (Discrete Events) dans lequel les processus communiquent par événements datés, la propagation des données observées sur l'interface d'un bloc après sa mise à jour ne sera pas systématiquement instantanée comme dans le modèle de calcul SDF et dépendra de la date courante du modèle. Ces opérations `schedule` et `propagate` doivent donc être décrites de manière spécifique selon la sémantique de chaque modèle de calcul. Nous décrivons comment ces opérations peuvent être décrites dans la section 4.6.

4.5.5 Entrelacement des opérations d'ordonnancement et de propagation

Dans l'enchaînement des étapes présenté dans les sections précédentes, à chaque tour de boucle, le bloc à observer est choisi, puis le bloc choisi est mis à jour, puis les données obtenues lors de la mise à jour sont propagées pour être disponibles pour le bloc suivant.



(a) Ajout d'une opération de propagation (`prePropagate`)

(b) Ajout d'une opération d'ordonnancement (`interSchedule`)

(c) Généralisation de l'entrelacement

FIG. 4.18 – Entrelacement des opérations d'ordonnancement et de propagation

Or, il peut arriver que le choix du bloc à observer dépende de la façon dont sont propagées les données. Ainsi par exemple, si les données à propager contiennent un destinataire, il faudra d'abord qu'elles soient propagées afin d'être remises au destinataire qui devra être ensuite observé. Dans ce cas, `schedule` ne peut avoir lieu qu'après `propagate`. Or ce n'est pas le cas sur notre algorithme lors du premier tour de boucle. Il faudrait donc rajouter une étape de propagation avant le choix du bloc à mettre à jour. Nous appelons cette étape `prePropagate` (voir la figure 4.18a).

Par ailleurs, il peut arriver que le résultat de l'observation du bloc conditionne le bloc auquel doivent être propagées les données. Ainsi par exemple, si le bloc observé modifie le flot de contrôle de l'exécution, le bloc auquel propager les données de manière à ce qu'il puisse fournir un résultat correct lors de sa mise à jour au tour de boucle suivant dépend du résultat de la mise à jour du bloc courant. Or ce n'est pas possible dans notre algorithme car aucune opération d'ordonnancement ne précède directement d'opération de propagation, que ce soit avant ou après l'opération de mise à jour. Il faudrait donc rajouter une étape d'ordonnancement avant la propagation des résultats de la mise à jour. Nous appelons cette étape `interSchedule` (voir la figure 4.18b).

De manière plus générale, nous nous apercevons que l'enchaînement de base `schedule-update-propagate` ne suffit pas pour permettre l'expression de n'importe quelle politique d'exécution. Notre algorithme devant être suffisamment générique pour supporter l'expression de modèles d'exécution très différents, nous généralisons donc ces principes de manière à permettre n'importe quel enchaînement d'opérations d'ordonnancement et de propagation avant et après la mise à jour du bloc courant et pour n'importe quel tour de boucle (initial et final compris). Nous alternons ainsi successivement les opérations d'ordonnancement et de propagation avant et après la mise à jour du bloc courant, comme cela est représenté sur la figure 4.18c. Suivant les modèles de calcul, certaines de ces étapes peuvent être laissées vides si elles n'ont pas d'utilité dans la description.

La séquence d'opérations de base à chaque tour de boucle pendant le calcul d'un snapshot est donc (1) de choisir un bloc en fonction de l'état du modèle et des données d'entrée disponibles (`initSchedule`), (2) propager les données d'entrée pour ce bloc (`prePropagate`), puis (3) de choisir un bloc à observer (`preSchedule`), (4) demander à ce bloc de mettre à jour son interface

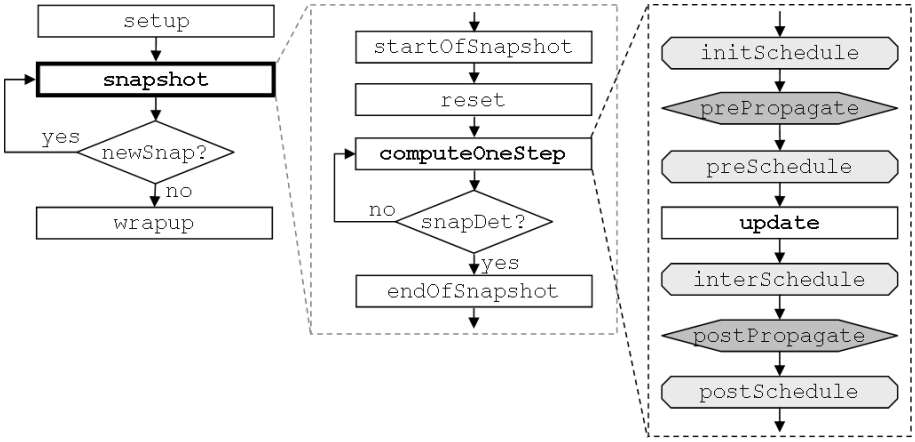


FIG. 4.19 – Boucle d'observation des blocs avec entrelacement

(`update`), (5) choisir un bloc en fonction de l'état du modèle et des données obtenues lors de la mise à jour (`interSchedule`), (6) propager les données suivant le bloc choisi (`postPropagate`) et enfin (7) choisir un bloc suivant les données qui viennent d'être propagées (`postSchedule`). Cette séquence d'opérations est construite de manière à ce qu'un bloc puisse être ordonnancé dès qu'une information nouvelle sur l'état du modèle est obtenue (nouvelles entrées, nouvelles sorties, propagation de données), et que la propagation des données puisse dépendre du bloc qui a été ordonnancé.

Comme cet enchaînement d'opérations constitue l'ensemble des étapes nécessaires à l'observation d'un bloc dans le calcul d'un snapshot, c'est-à-dire à l'exécution d'un *pas* de calcul, nous les regroupons sous une opération unique que nous appelons `computeOneStep`, comme cela est représenté sur la figure 4.19.

4.5.6 Arrêt de la boucle d'observation des blocs

La boucle doit s'arrêter lorsque le snapshot est « déterminé », c'est-à-dire qu'une observation du modèle a été obtenue. L'étape vérifiant cette condition est appelée `snapDet`. Si le snapshot n'est pas « déterminé », une nouvelle observation est effectuée. Sinon le modèle de calcul passe à l'étape `endOfSnapshot` qui termine le snapshot.

Dans une première approche, la condition d'arrêt de la boucle (c'est-à-dire la condition de « détermination » d'un snapshot) pourrait être que tous les blocs du modèle ont été observés une fois. Cependant, cette condition n'est pas suffisante pour traiter le cas de modèles de calcul dans lesquels des boucles de rétroaction instantanées sont autorisées, ce qui est le cas pour le modèle de calcul Réactif/Synchrone (SR) par exemple. En effet, lorsque de telles boucles sont autorisées dans un modèle, les entrées d'un bloc peuvent dépendre instantanément de ses sorties. Un bloc pouvant être utilisé dans une boucle de rétroaction instantanée doit donc être capable de produire une ou plusieurs de ses sorties en ne disposant que d'une partie de ses entrées. Un tel bloc est appelé bloc « non strict » [Edw97]. Par exemple, une porte logique OU est non stricte car sa sortie est connue dès que l'une de ses entrées est vraie. Observer une seule fois un bloc non strict lors du calcul d'un snapshot n'est alors pas suffisant pour connaître son comportement, qui peut être modifié en fonction de la sortie produite lors de l'observation. La condition d'arrêt de la boucle doit donc être une condition de point fixe telle que la boucle est exécutée tant que de nouvelles données sont produites par l'observation des blocs.

Cependant, cette condition de point fixe n'est valable que sous une hypothèse de monotonie croissante et bornée de l'observation des blocs : des observations successives d'un même bloc doivent permettre d'obtenir de plus en plus d'information sur ce bloc jusqu'à ce que tous les points d'interface du bloc aient une valeur observable stable. Si cela n'est pas le cas, notre

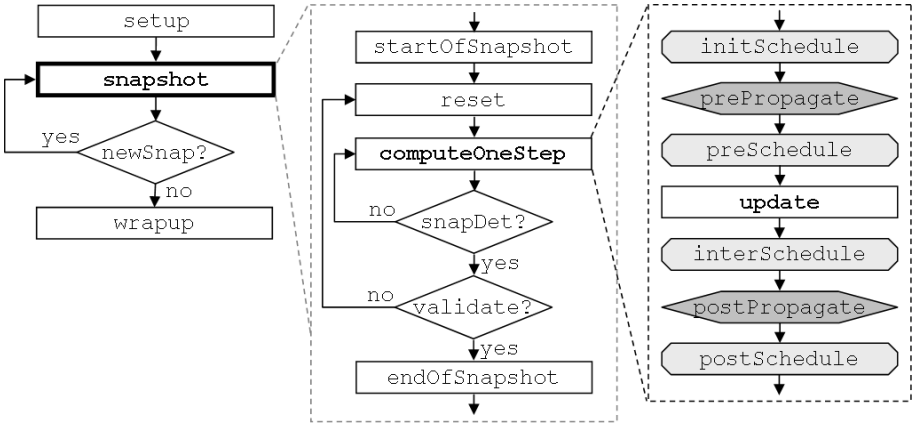


FIG. 4.20 – Algorithme d'exécution générique complet avec étape de validation

algorithme peut ne pas converger. Pour éviter ce cas, une condition minimale d'arrêt de la boucle est l'obtention d'une observation pour tous les points d'interface du modèle, et notamment pour ses sorties. En tout état de cause, cette condition d'arrêt peut dépendre du modèle de calcul considéré et peut donc être modifiée pour tenir compte des spécificités d'un modèle de calcul.

4.5.7 Validation d'un snapshot

Comme nous venons de le voir, le calcul d'un snapshot correspond au calcul d'un point fixe via l'observation successive des blocs composant le modèle jusqu'à ce que l'observation globale du modèle se stabilise. Or il peut arriver que ce point fixe ne soit pas unique. Prenons l'exemple d'une fonction $f : E \rightarrow E$ continue qui admet plusieurs points fixes, c'est-à-dire pour laquelle l'équation $f(x) = x$ admet plusieurs solutions. Pour trouver un des points fixes de cette fonction, il est possible d'utiliser une méthode itérative qui consiste à déterminer la limite vers laquelle converge (si elle converge) la suite récurrente définie par $u_0 \in E$ et $u_{n+1} = f(u_n)$. Le point fixe trouvé avec cette méthode (en admettant que la convergence soit obtenue) dépend du point de départ u_0 de la suite.

De la même manière, s'il existe plusieurs points fixes possibles pour un modèle, le point fixe obtenu par notre algorithme, qui dépend des paramètres de calcul utilisés par le modèle de calcul, peut ne pas être celui qui correspond au comportement du modèle. Afin de détecter les points fixes invalides et d'atteindre celui qui correspond au comportement du modèle, notre algorithme doit être capable d'effectuer plusieurs fois le calcul d'un même snapshot tout en permettant de modifier les paramètres de calcul. Pour cela, notre algorithme comporte une étape de validation appelée `validate`.

L'opération `validate` d'un modèle de calcul permet de définir dans quelles conditions un snapshot est jugé satisfaisant ou non par ce modèle de calcul. Si le snapshot calculé n'est pas satisfaisant, le modèle de calcul réinitialise les variables de calcul utilisées via l'opération `reset` et recommence le calcul du snapshot avec de nouveaux paramètres. Il est important de souligner que seuls les paramètres de calcul changent et non les données acquises représentant l'environnement. En effet, les données de l'environnement sont acquises uniquement lors de l'étape `startOfSnapshot`. De même, le résultat d'un snapshot non satisfaisant n'est pas visible du point de vue de l'environnement du modèle car ils ne sont rendus visibles que lors de l'étape `endOfSnapshot`.

Les conditions de validation d'un snapshot peuvent inclure la validation des observations locales des blocs par les blocs eux-mêmes (voir la section 4.5.8 suivante). Ainsi par exemple, considérons un composant dont le rôle est de détecter la date à laquelle le signal qu'il reçoit en entrée dépasse un seuil donné. Le modèle dans lequel ce composant est utilisé fait appel à un mo-

dèle de calcul qui discrétise les signaux sur le temps continu avec une fréquence f paramétrable. Le composant reçoit donc des échantillons de son signal d'entrée avec une période T. Lorsqu'il détecte que le seuil a été franchi entre l'échantillon de signal précédent et l'échantillon courant, il connaît la date à laquelle le signal a dépassé le seuil avec une précision de T. Si cette précision est insuffisante par rapport à ses critères de qualité, le composant peut invalider le snapshot, qui pourra alors être recalculé par le modèle de calcul avec une fréquence d'échantillonnage supérieure.

Cette étape de validation complète notre algorithme, qui est représenté avec l'ensemble de ses étapes sur la figure 4.20. Dans les sections suivantes nous nous intéressons à la façon dont cet algorithme est exécuté et comment il s'applique dans le cas d'un modèle à plusieurs niveaux hiérarchiques hétérogènes.

4.5.8 Délégation de l'exécution aux éléments du méta-modèle

4.5.8.1 Opérations d'exécution

A la manière de Kermeta (voir la section 3.3.2.1), nous définissons les étapes de notre algorithme comme des opérations sur les éléments de notre méta-modèle. Ainsi l'exécution en elle-même sera en fait déléguée aux instances de ces éléments, c'est-à-dire aux éléments du modèle exécuté. Seule la boucle de réalisation des séries de snapshots est réalisée par un programme que nous appelons le moteur d'exécution . Ce moteur pilote l'exécution d'un modèle mais délègue l'exécution de chaque étape de calcul aux éléments qui composent le modèle.

La réalisation de la boucle d'observation des blocs sur un modèle est déléguée au modèle de calcul qui lui est associé (instance de l'élément `ModelOfComputation` de notre méta-modèle).

Les blocs sont, eux, pourvus (a) des opérations `startOfSnapshot` et `endOfSnapshot` qui permettent de gérer le contexte de calcul, (b) d'une opération `update` qui est appelée par le modèle de calcul lorsqu'il requiert une mise à jour de l'interface d'un bloc et (c) d'une opération `validate` qui leur permet de valider le snapshot en cours. L'opération `update` est essentielle car elle permet d'observer l'interface d'un bloc, qui est vu par ailleurs comme une boîte noire. Les blocs disposent également d'autres opérations telles que `setup`, `wrapup` et `reset`.

Le détail des opérations de chacun des éléments du méta-modèle est présenté sur la figure 4.21. Certaines des opérations illustrées sur cette figure seront discutées dans les sections suivantes.

4.5.8.2 Variables de calcul

Nous avons vu que le modèle de calcul pouvait stocker des données intermédiaires dans des variables de calcul au cours du calcul du snapshot et notamment de la boucle d'observation des blocs. De plus, il est nécessaire de stocker, au début de chaque snapshot, les données acquises de l'environnement du modèle et qui constituent le contexte de calcul. De la même manière que les opérations d'exécution de notre algorithme sont réalisées par des méthodes sur les éléments de notre méta-modèle, ces variables peuvent être réalisées par des attributs définis sur les éléments du méta-modèle.

Afin de séparer les données de calcul intermédiaires pertinentes au cours d'un seul snapshot des données permettant de mémoriser l'état du modèle tout au long de l'exécution, nous adoptons les principes suivant :

- Les variables permettant de mémoriser l'état du modèle tout au long de l'exécution sont représentées par des attributs de l'élément *Model*. Ces variables ne changent de valeur que lors des opérations `startOfSnapshot` et `endOfSnapshot`.
- Les variables de calcul intermédiaires utilisées tout au long du calcul d'un snapshot et notamment au cours de la boucle d'observation des blocs sont représentées par des attributs

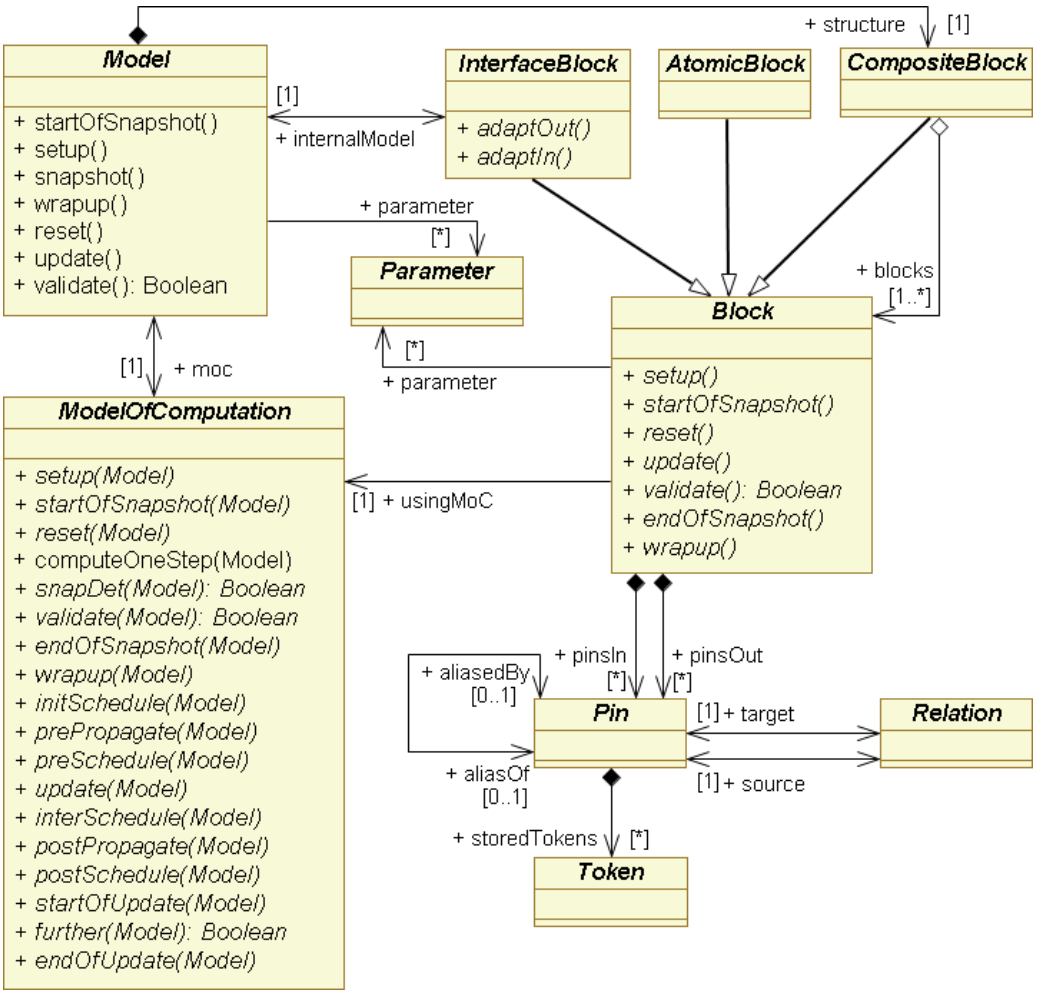


FIG. 4.21 – Opérations d'exécution des éléments du méta-modèle

de l'élément *ModelOfComputation*. Ces variables peuvent changer de valeur à n'importe quel moment du calcul.

De cette manière, les attributs d'un modèle représentent son état, tandis que les variables utilisées pour calculer son prochain état sont représentées par des attributs du modèle de calcul.

4.5.9 Hiérarchie et hétérogénéité : mise à jour d'un bloc d'interface

La notion de bloc d'interface est un concept essentiel de notre syntaxe abstraite car elle permet l'hétérogénéité via la hiérarchie. En effet, le comportement d'un bloc d'interface est décrit dans un modèle interne dont le modèle de calcul peut être différent de celui du modèle dans lequel le bloc est utilisé. De plus, un bloc d'interface joue le rôle d'adaptateur sémantique entre son modèle interne et le modèle dans lequel il est utilisé. En conséquence, l'opération de mise à jour `update` d'un bloc d'interface est particulière.

4.5.9.1 Mise à jour du modèle interne

Le rôle de l'opération `update` pour les blocs est de mettre à jour les points d'interface du bloc avec les données représentant ce qu'il est possible d'observer de son comportement. Dans le cas d'un bloc d'interface, le comportement du bloc est défini par son modèle interne. Pour obtenir une mise à jour d'un bloc d'interface, il faut donc obtenir une mise à jour de son modèle

interne. Nous employons ici volontairement le terme « mise à jour » car, comme nous allons le détailler par la suite, une mise à jour d'un modèle est différente d'un snapshot d'un modèle.

Les modèles sont donc pourvus d'une opération de mise à jour, également appelée `update`, et l'opération `update` d'un bloc d'interface contient donc un appel à l'opération `update` de son modèle interne.

4.5.9.2 Opérations d'adaptation sémantique

Le modèle interne d'un bloc d'interface peut impliquer un modèle de calcul différent de celui impliqué dans le modèle dans lequel le bloc d'interface est utilisé (que nous appellerons modèle externe pour plus de concision). Le bloc d'interface joue alors le rôle d'adaptateur sémantique entre les deux modèles de calcul de ces deux modèles. Ce rôle d'adaptateur sémantique implique les responsabilités potentielles suivantes (non exclusives entre elles) :

- Adaptation du flot de contrôle : transmission ou non des données au modèle interne, mise à jour du modèle interne immédiatement ou lors d'un snapshot suivant, etc.
- Adaptation des données : interprétation, traduction, transformation des données fournies en entrée du modèle interne avant sa mise à jour et de même pour les données produites en sortie du modèle interne après sa mise à jour.
- Adaptation des notions de temps : interprétation, traduction, transformation des étiquettes temporelles présentes sur les données, adaptation des contraintes de temps, etc. (voir la section 4.5.10 pour plus de détails sur la gestion du temps).

Pour cela, un bloc d'interface est pourvu de deux opérations d'adaptation : `adaptIn` et `adaptOut`. Le rôle de `adaptIn` est de réaliser l'adaptation nécessaire *avant* la mise à jour du modèle interne. Symétriquement, le rôle de `adaptOut` est de réaliser l'adaptation nécessaire *après* la mise à jour du modèle interne. Ainsi par exemple, si le modèle interne d'un bloc d'interface requiert que deux de ses entrées soient disponibles simultanément pour pouvoir fournir tout ou partie de ses sorties mais qu'elles sont fournies dans deux snapshots distincts dans le modèle externe, alors le bloc d'interface stockera la première de ces entrées et attendra que la seconde soit rendue disponible avant de les fournir au modèle interne et de provoquer sa mise à jour. Inversement, si les deux entrées doivent être exclusives pour le modèle interne mais que le modèle externe les fournit simultanément, le bloc d'interface les délivrera au modèle interne dans deux snapshots distincts. Une adaptation similaire peut avoir lieu en sortie du modèle interne, avant la transmission des données produites par la mise à jour du modèle interne au modèle externe.

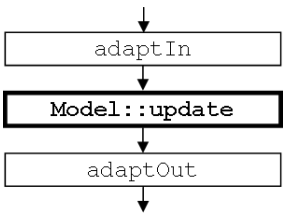


FIG. 4.22 – Mise à jour d'un bloc d'interface

Les opérations `adaptIn` et `adaptOut` permettent en fait de spécifier explicitement comment les sémantiques des modèles de calcul impliqués dans les modèles interne et externe d'un bloc d'interface sont adaptées avant et après la mise à jour du modèle interne. En conséquence, ces opérations représentent le sens qui est donné à la combinaison de ces deux modèles de calcul.

4.5.9.3 Boucle de mise à jour du modèle interne

Après l'adaptation en entrée spécifiée dans l'opération `adaptIn`, le modèle interne se voit demander une mise à jour de son interface (c'est-à-dire de ses sorties).

Tout d'abord, de la même manière que pour un snashot, les données présentes sur les entrées du modèle doivent être acquises. Cette tâche est effectuée par l'opération `startOfUpdate`. Cette opération peut être considérée comme le pendant de l'opération `startOfSnapshot` : lorsque l'opération `startOfSnapshot` permet d'acquérir les données de l'environnement avant la réalisation d'un snapshot sur un modèle, l'opération `startOfUpdate` permet d'acquérir les données produites par les blocs environnants avant la réalisation d'un update sur un modèle embarqué dans un bloc d'interface. L'opération symétrique de `startOfUpdate` est appelée `endOfUpdate` et permet de rendre les données résultant de la mise à jour du modèle disponibles pour les blocs environnants (voir la figure 4.23).

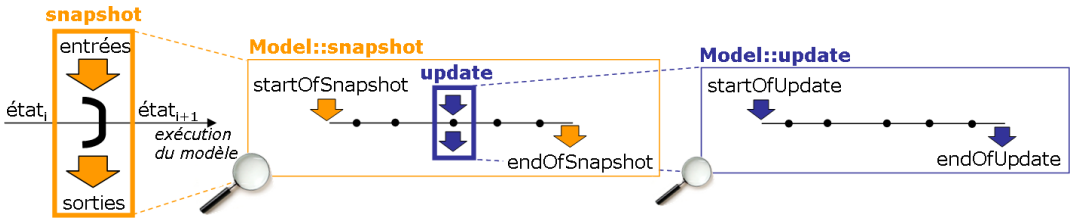


FIG. 4.23 – `startOfSnapshot`/`endOfSnapshot` et `startOfUpdate`/`endOfUpdate`

Puis, toujours comme pour un snapshot, il est nécessaire de mettre à jour les blocs du modèle pour obtenir une mise à jour du modèle. Il faut donc ordonnancer la mise à jour des blocs et propager les données obtenues. Le calcul d'un pas de mise à jour (c'est-à-dire la mise à jour d'un bloc) est donc identique au calcul d'un pas de snapshot. Pour que la mise à jour corresponde à une observation d'un comportement conforme vis-à-vis de la sémantique du modèle de calcul associé à ce modèle interne, le calcul d'un pas de mise à jour est donc effectué par l'opération `computeOneStep` du modèle de calcul interne (voir la section 4.5.5).

Enfin, de la même manière que pour calculer un snapshot, cette opération doit être effectuée en boucle. Cependant, la condition d'arrêt de la boucle de mise à jour d'un modèle interne ne peut pas être identique à la condition d'arrêt de la boucle d'un snapshot. En effet, nous avons vu dans la section 4.5.6, que la condition d'arrêt de la boucle d'un snapshot était a minima l'obtention d'une valeur d'observation pour tous les points d'interface du modèle observé. Cependant, si le modèle représente le comportement d'un bloc non strict, il peut ne pas être en mesure de fournir une valeur observable pour tous ses points d'interface lorsque cela lui est demandé (en fonction des entrées qui lui ont été fournies). Pour tenir compte de ce cas de figure, l'opération déterminant la condition d'arrêt de la boucle de mise à jour d'un modèle est distincte de l'opération `snapDet`. Cette opération est appelée `further` car elle a pour rôle de déterminer si le calcul de la mise à jour du modèle peut être mené plus avant ou si toutes les valeurs observables compte tenu du contexte de calcul ont été fournies. L'ensemble des opérations de mise à jour d'un modèle interne est représenté sur la figure 4.24. Toutes ces opérations sont réalisées par le modèle de calcul associé au modèle interne.

4.5.9.4 Récapitulatif : exécution hiérarchique

Notre algorithme s'exécute à travers tous les niveaux de la hiérarchie d'un modèle par l'intermédiaire des blocs d'interface. Afin d'illustrer cette exécution hiérarchique, considérons le modèle de la figure 4.25, repris de la section 4.4.4 décrivant le principe des blocs d'interface.

Le diagramme de séquence de la figure 4.26 montre la chaîne d'appel des opérations de notre algorithme lors du calcul d'un snapshot sur ce modèle. Le calcul commence par l'opération `startOfSnapshot`, qui s'exécute à travers tous les niveaux de la hiérarchie afin de figer le contexte de calcul du snapshot. Puis, après l'initialisation des variables de calcul (opération `reset`), qui s'exécute à travers tous les niveaux de la hiérarchie, le modèle de calcul « X » réalise l'observation successive des blocs du modèle racine (appelé dans notre exemple « Model »). Lorsque le bloc

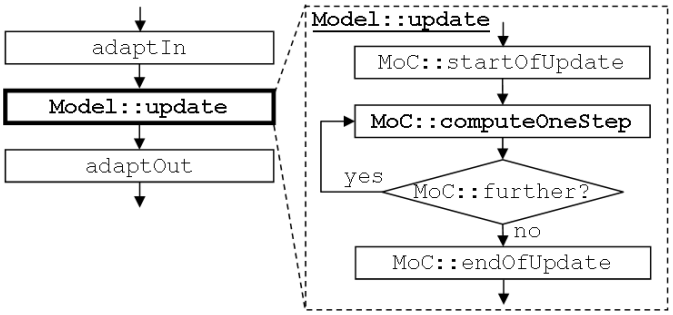


FIG. 4.24 – Mise à jour du modèle interne d'un bloc d'interface

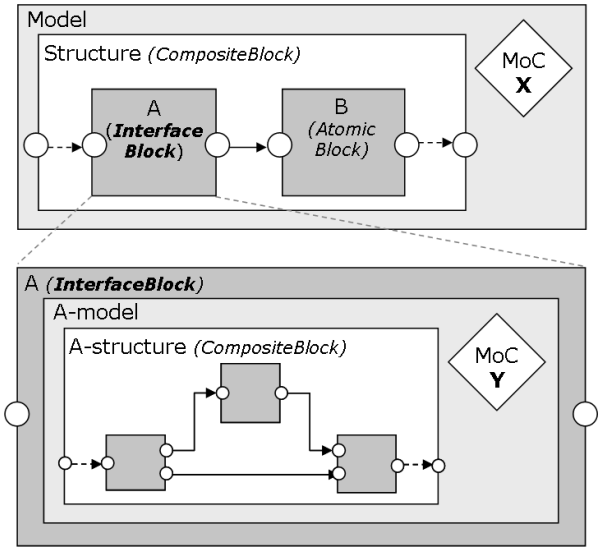


FIG. 4.25 – Exemple de modèle hiérarchique

d'interface « A » est observé, il réalise l'adaptation sémantique nécessaire et délègue la mise à jour à son modèle interne « A-Model ». Le calcul de cette mise à jour est effectué par le modèle de calcul « Y » qui lui est associé. Celui-ci demande des mises à jour successives aux différents blocs du modèle « A-Model » (un seul de ces blocs est représenté sur la figure). Une fois le snapshot déterminé, l'étape de validation est exécutée à travers tous les niveaux de la hiérarchie, afin que tous les blocs puissent valider l'observation obtenue. Si le snapshot n'est pas validé, le calcul est effectué de nouveau (boucle de validation). Sinon le calcul du snapshot est alors terminé, les données calculées sont fournies à l'environnement (opération `endOfSnapshot`) et le snapshot suivant peut alors être effectué.

Soulignons sur cet exemple que les snapshots ne sont donc réalisés que sur le niveau racine d'un modèle hiérarchique, qui représente le système global. Un modèle qui est embarqué dans un bloc d'interface se voit seulement demander des mises à jour. A la fin du calcul d'un snapshot, l'état observable de tous les blocs du modèle est normalement défini (du moins de tous les blocs pertinents selon les modèles de calcul impliqués dans chacun des niveaux).

4.5.10 Modélisation du temps

Comme nous l'avons introduit dans la section 4.3.4, notre approche doit permettre : (1) de spécifier un modèle de temps dans la description d'un modèle de calcul et (2) de décrire comment deux modèles de temps dans deux modèles de calcul peuvent être mis en relation. Afin de supporter ces deux types de tâches, nous adoptons pour ModHel'X une approche inspirée du

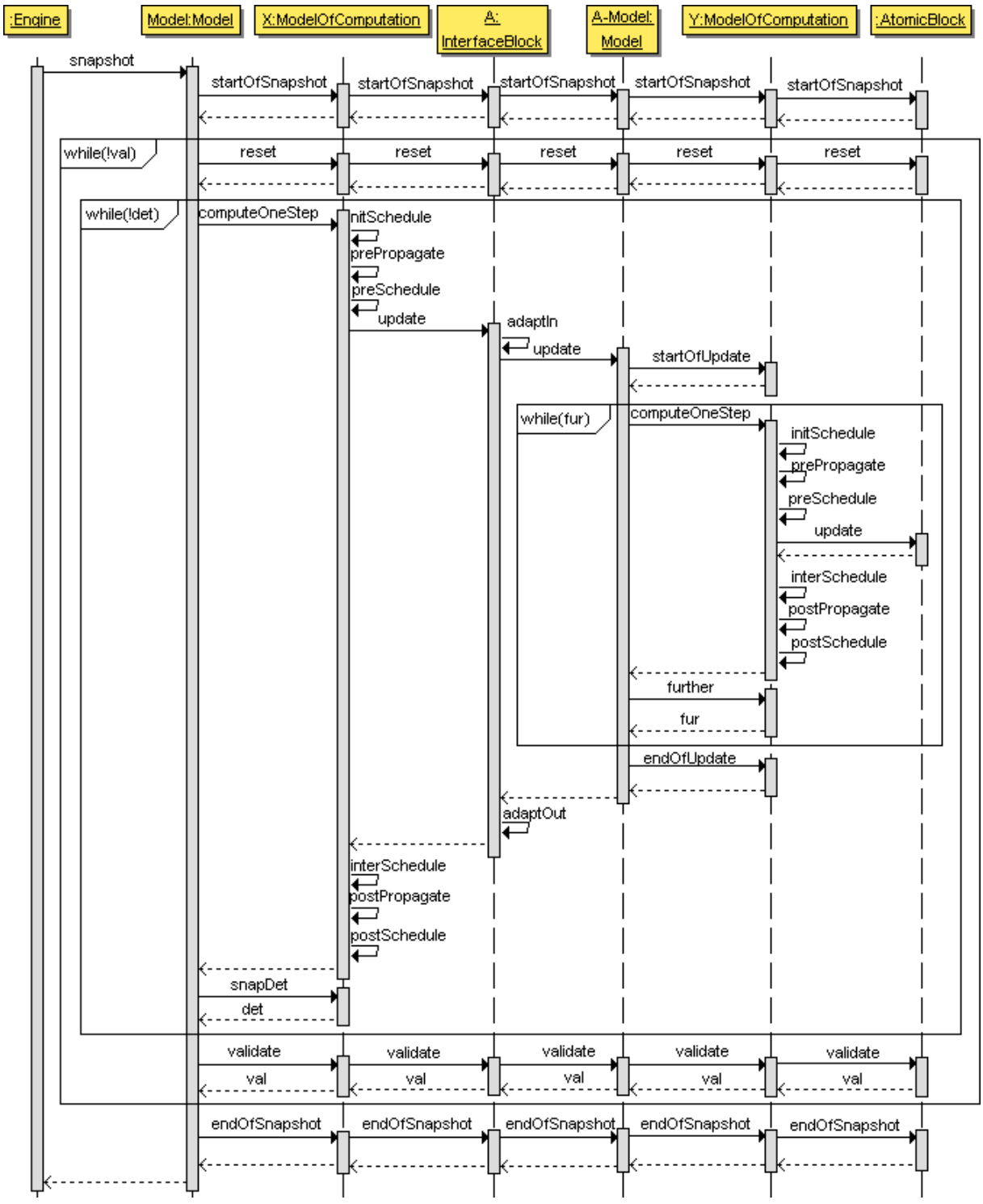


FIG. 4.26 – Diagramme de séquence de l'exécution hiérarchique

modèle de temps proposé dans le profil MARTE (UML Profile for Modeling and Analysis of Real-time and Embedded systems) [OMGj].

4.5.10.1 Le temps dans le profil MARTE

La modélisation du comportement en UML est essentiellement non temporelle. Un modèle de temps et un modèle de causalité existent dans sa sémantique, mais ne permettent pas la prise en compte de notions de temps complexes (temps réel, multi-horloge, etc.). L'un des objectifs du profil MARTE est d'étendre UML avec un support pour l'expression de modèles de temps et de propriétés temporelles.

Le concept de « temps » dans MARTE est compris au sens large du terme : il inclut le temps physique, qu'il soit considéré de façon continue, discrète ou dense, le temps logique, différentes notions de durée, et même le temps « multi-formes » (possibilité de lier des occurrences d'événements à des grandeurs non temporelles, par exemple des distances, des angles, etc.). Le modèle de temps proposé dans MARTE [OMGj, p. 51] est très général et permet l'expression de ces diverses notions.

Dans MARTE, toute notion de temps est construite à partir d'une *base de temps*. Une base de temps contient un ensemble totalement ordonné d'instants. Différentes bases de temps peuvent être assemblées et structurées par des relations pour former une *structure de temps*. Les relations permettent de définir des dépendances entre les bases de temps ainsi rassemblées en exprimant des notions telles que la coïncidence ou la précédence entre instants.

MARTE permet également de modéliser des *horloges*. Une horloge définit en fait, dans MARTE, un moyen d'accéder au temps. Elle réfère donc à une base de temps et dispose d'une unité. Une valuation du temps sur les instants et sur les intervalles entre instants (pour définir la notion de durée) est possible. Une horloge donne notamment accès à son temps « courant ». Des *contraintes d'horloges* peuvent être utilisées pour définir les relations entre différentes horloges. Ces contraintes sont exprimées à l'aide d'un langage déclaratif d'expression de contraintes appelé CCSL (Clock Constraint Specification Language).

En appliquant le profil MARTE, les événements et les comportements définis en UML peuvent être directement et explicitement liés à un modèle de temps via une horloge.

4.5.10.2 Représentation du temps dans ModHel'X

Dans ModHel'X, nous adoptons un principe similaire à celui de MARTE : tout modèle de temps est exprimé par rapport à une base de temps constituée d'un ensemble ordonné d'instants. Notre modèle d'exécution générique fournit une base de temps générique constituée de la succession des snapshots. Sur cette base de temps générique, un modèle de temps peut être défini de manière propre à chaque modèle de calcul : la succession des snapshots constituant un ensemble d'instants, chaque modèle de calcul peut définir une relation d'ordre, des étiquettes temporelles (que nous appelons « dates » pour plus de concision), une unité de temps et une notion de distance (permettant d'exprimer le concept de durée) sur les snapshots.

Nous ne proposons pas de système particulier d'expression pour les modèles de temps, contrairement à MARTE qui définit un ensemble de stéréotypes utilisables dans des modèles UML. La gestion du temps se fait dans les différentes étapes de notre modèle d'exécution générique. La façon de gérer le temps fait donc partie de la description de ces étapes (voir la section 4.6.2).

Remarquons au passage que si les snapshots définissent des instants sur un modèle de temps, les observations successives des blocs dans un snapshot définissent en fait des « micro-pas » [Bou91] successifs dans un même instant. Le temps n'avance pas au cours du calcul d'un snapshot, au même titre que les données représentant l'environnement n'évoluent pas au cours

du calcul d'un snapshot. Si une date doit être calculée pour un snapshot, elle ne peut être effective que lors du `startOfSnapshot` ou du `endOfSnapshot`, c'est-à-dire soit au moment où le contexte de calcul est figé soit au moment ou le snapshot calculé est rendu visible.

4.5.10.3 Mise en relation de modèles de temps

Le fait de proposer l'expression des modèles de temps sur la base unique de la succession des snapshots permet d'obtenir la généricité nécessaire à l'expression de modèles de calcul très variés. Cela facilite également la combinaison de différents modèles de temps dans un même modèle puisque la succession des snapshots sert de base commune pour l'expression du temps dans tous les modèles de calcul impliqués dans un modèle.

La mise en relation des modèles de temps définis dans deux modèles de calcul est effectuée par les blocs d'interface. Tout comme ils adaptent le contrôle et les données, les blocs d'interface sont chargés d'adapter les notions de temps dans les modèles de calcul associés à leurs modèles externes et internes dans leurs opérations `adaptIn` et `adaptOut`.

Il est important de souligner ici que, contrairement à MARTE, nous n'utilisons pas un système de contraintes globalement définies pour mettre en relation deux modèles de temps. La mise en relation est locale à la frontière entre deux modèles de calcul ayant deux modèles de temps différents. Nous verrons dans la section 4.7 que cela peut avoir des effets limitants.

4.5.10.4 Déclenchement des snapshots : contraintes de temps

Dans un modèle hétérogène, chaque modèle de calcul définit son propre modèle de temps et définit une étiquette de temps (une date) pour chaque snapshot indépendamment dans son niveau hiérarchique. Le temps n'avance donc pour chaque modèle de calcul que lorsqu'un snapshot a lieu. En conséquence, il est nécessaire de faire un snapshot pour que le temps avance dans un modèle. La question qui se pose dans ce contexte est donc : quelles sont les conditions de déclenchement d'un snapshot par le moteur d'exécution ?

Nous prenons en compte trois possibilités dans notre approche. Tout d'abord, il est possible pour le concepteur de donner des paramètres au moteur d'exécution pour l'exécution d'une série de snapshots (il peut choisir par exemple de réaliser 10 snapshots successifs). Il peut également déclencher les snapshots manuellement. D'autre part, lors de l'exécution, des données d'entrée représentant l'environnement du système peuvent être fournies au modèle. Les variations de ces données sont susceptibles de provoquer une réaction du système, qu'il est nécessaire d'observer. Chaque échantillon de ces données fourni au modèle provoque donc un snapshot. Enfin, le système peut également réagir au passage du temps (déclenchement d'une action après l'expiration d'un délai par exemple). Dans ce cas, le snapshot est provoqué parce qu'une certaine « date » dans le modèle de temps impliqué dans le modèle du système est atteinte. Pour tenir compte de ce cas de figure, nous introduisons la notion de *contrainte de temps*.

Une contrainte de temps définit une date dans le contexte d'un modèle de calcul (et donc d'un modèle de temps) à laquelle le prochain snapshot doit être effectué. Les contraintes de temps peuvent être produites au cours du calcul d'un snapshot par les blocs d'un modèle ou par le modèle de calcul qui lui est associé. Concrètement, un modèle de calcul maintient une liste de contraintes dans laquelle les blocs peuvent ajouter les contraintes qu'ils produisent lors de leur mise à jour. A la fin du calcul du snapshot courant, le modèle de calcul résout l'ensemble des contraintes produites afin de déterminer la date du prochain snapshot dans son propre contexte de temps. La production de cette contrainte résultante déclenche un nouveau snapshot après la fin du calcul du snapshot courant.

Dans un modèle hétérogène, les blocs et les modèles de calcul de chaque niveau hiérarchique du modèle peuvent produire des contraintes au cours du snapshot. Les contraintes émises dans chaque niveau sont résolues par le modèle de calcul concerné. Lors de la mise à jour d'un bloc

d'interface, la mise à jour de son modèle interne peut provoquer la production de contraintes de temps. Ces contraintes de temps sont adaptées par le bloc d'interface au même titre que les données d'entrée et de sortie du modèle. Après traduction, le bloc d'interface peut alors ajouter ces contraintes dans la liste de contraintes du modèle de calcul de son modèle externe. Ainsi, les contraintes produites par des modèles des niveaux les plus bas de la hiérarchie sont « traduites » puis transmises vers les niveaux supérieurs via les blocs d'interface et peuvent donc être prises en compte pour le déclenchement du snapshot suivant sur le modèle racine. Ce principe est particulièrement important puisqu'il permet que l'exécution du modèle ne soit pas pilotée uniquement par le modèle racine, comme cela est le cas dans Ptolemy par exemple.

Pour le moment, l'expression des contraintes est limitée à la définition de dates minimum pour le prochain snapshot, ces dates étant exprimées dans les modèles de temps des modèles de calcul concernés. Comme une relation d'ordre est obligatoirement définie entre deux dates dans notre modèle, la résolution des contraintes est donc simple. L'expression de contraintes plus complexes est cependant tout à fait envisageable. Il faudrait pour cela introduire un langage d'expression de contraintes assorti d'un solveur de contraintes adéquat.

4.6 Méthode de description d'un modèle de calcul dans notre approche

Nous avons présenté dans les deux sections précédentes la syntaxe abstraite générique que nous avons mise au point pour permettre de réaliser des modèles hétérogènes ainsi que la sémantique abstraite servant de base pour l'expression de la sémantique des modèles de calcul. Dans cette section nous détaillons comment un modèle de calcul particulier peut être décrit dans notre approche de façon à pouvoir être utilisé dans un modèle hétérogène.

Dans un premier temps, nous montrons comment définir une syntaxe abstraite spécifique à un modèle de calcul à partir de notre syntaxe abstraite générique. Puis nous montrons comment donner une sémantique concrète à un modèle de calcul en s'appuyant sur la sémantique abstraite que nous avons définie dans la section précédente.

4.6.1 Syntaxe spécifique : spécialisation de la syntaxe abstraite générique

Les concepts constituant notre syntaxe abstraite sont génériques, c'est-à-dire que nous partons du postulat que ces concepts peuvent être utilisés pour décrire la structure des modèles quel que soit le modèle de calcul considéré. Chaque modèle de calcul interprète alors ces concepts d'une façon particulière. Ainsi par exemple, les blocs sont interprétés comme des processus dans le modèle de calcul Synchronous DataFlow (SDF).

4.6.1.1 Problématique : représentation de concepts spécifiques

Il peut arriver que, pour un paradigme donné, différents éléments soient représentés par le même élément dans notre syntaxe mais ne doivent pas être interprétés de la même manière par le modèle de calcul. Prenons l'exemple d'une version simple du paradigme des machines à états finis telle que celle présentée dans [MFJ05]. Dans ce paradigme, les concepts élémentaires utilisés pour la modélisation sont les états et les transitions. Il faut donc représenter ces concepts dans notre syntaxe. Nous avons vu que les blocs de notre syntaxe représentent des unités élémentaires de comportement (voir la section 4.4.1) qui, interconnectées, réalisent par interaction la fonctionnalité du système. Si l'on respecte ce principe, les états ne peuvent pas être représentés par des blocs car un état représente une situation statique pour le système. Par contre une transition comporte une garde et une action qui correspondent chacune à une unité de comportement : la garde a pour comportement l'évaluation d'une expression booléenne (présence ou non d'un

événement), le résultat de son exécution étant le résultat de l'évaluation, tandis que l'action a pour comportement l'émission d'un événement, le résultat de son exécution étant l'événement produit. Les concepts de garde et d'action peuvent donc être représentés dans notre syntaxe abstraite par des blocs. Cependant, le modèle de calcul doit pouvoir différencier les gardes des actions dans ses opérations d'ordonnancement et de propagation car, notamment, une garde doit être observée avant l'action à laquelle elle est associée sur la transition. Afin de rendre possible ce type de différenciation, nous permettons la spécialisation des concepts de notre syntaxe abstraite. Le terme « spécialisation » est entendu ici au sens de l'héritage des approches objets, qui permet le polymorphisme. De cette manière, chaque concept de notre syntaxe abstraite générique peut être spécialisé de manière à représenter des notions spécifiques à un modèle de calcul lorsque cela est nécessaire.

4.6.1.2 Exemple de la représentation des machines à états finis (FSM)

Pour illustrer le principe de la spécialisation des concepts de notre méta-modèle générique pour représenter des concepts spécifiques à un modèle de calcul, nous allons montrer comment des automates à états finis peuvent être représentés dans ModHel'X. Nous considérons ici une version simple du modèle de calcul des automates à états finis, FSM.

Afin de pouvoir représenter des automates à états finis dans ModHel'X, il est nécessaire de décider comment utiliser les éléments du méta-modèle de ModHel'X pour représenter les événements, les gardes, les actions, les transitions et les états.

Evénements Les événements considérés dans notre version simple de FSM sont des événements qui sont distingués par leur nom. Pour les représenter dans ModHel'X, nous spécialisons l'élément `Token` du méta-modèle générique par un élément `FSMEvent`, comme cela est représenté sur la figure 4.27.

Gardes et actions Nous choisissons de représenter les gardes et les actions d'un automate FSM par des blocs dans ModHel'X. En effet, comme nous l'avons rappelé dans la section précédente, les blocs représentent des unités élémentaires de comportement. Une garde a pour comportement l'évaluation d'une expression booléenne (présence ou non d'un événement), le résultat de son exécution étant le résultat de l'évaluation. Une action a pour comportement l'émission d'un événement, le résultat de son exécution étant l'événement produit. Un bloc représentant une garde ou une action pourra être indifféremment un bloc atomique, un bloc composite ou même un bloc d'interface encapsulant un modèle interne (qui peut impliquer un MoC différent du MoC FSM).

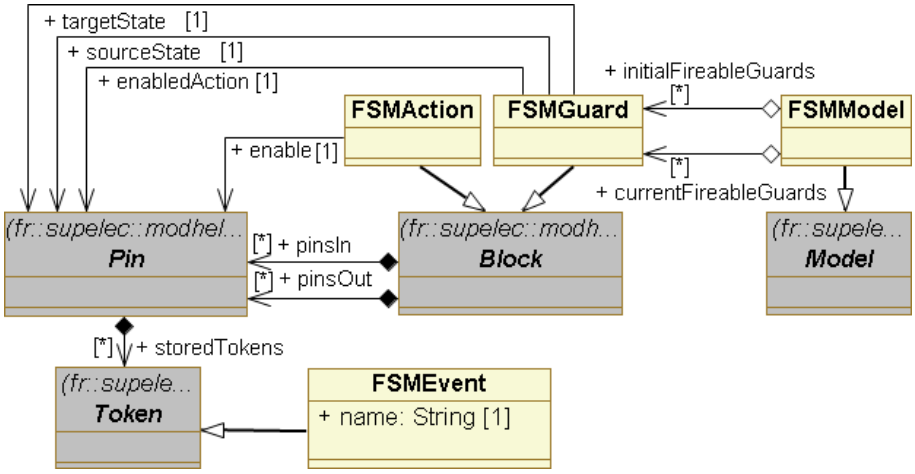


Fig. 4.27 – Méta-modèle spécialisé pour le modèle de calcul Finite State Machine (FSM)

Afin de différencier les gardes des actions (le modèle de calcul ne les ordonnancera pas de la même manière puisque, sur une transition, la garde doit être évaluée avant que l'action ne soit réalisée), nous spécialisons donc la notion de `Block` en deux éléments `FSMGuard` et `FSMAction` dans le méta-modèle spécifique pour FSM, représenté sur la figure 4.27.

Transitions Dans FSM, une transition est constituée d'une garde et d'une action qui lui est associée (l'action étant optionnelle) et qui sera réalisée si la transition est tirée. Dans notre représentation, pour pouvoir représenter le fait qu'une `FSMAction` particulière est liée à une `FSMGuard` particulière pour former une transition, nous ajoutons à `FSMGuard` un point d'interface particulier appelé `enabledAction` qui va permettre de l'associer, grâce à une `Relation`, à une `FSMAction` qui possède, de son côté, un point d'interface particulier appelé `enable`.

Soulignons que, en conséquence, la notion de transition n'est pas représentée explicitement dans la représentation que nous avons choisie : seules les gardes et les actions associées apparaissent dans un modèle.

Etats Dans FSM, les états représentent des situations statiques pour le système et non pas des unités de comportement dynamiques. En conséquence, nous choisissons de ne pas les représenter par des blocs dans ModHel'X. Par ailleurs, dans FSM, un état a un ensemble de transitions qui le quittent, chacune d'elle menant à un autre état qui peut devenir l'état suivant de l'automate si la transition est tirée c'est-à-dire si la garde de la transition est satisfaite. Or dans la représentation que nous avons choisie dans ModHel'X, une transition est constituée obligatoirement d'une `FSMGuard`. Nous pouvons donc choisir de représenter la notion d'état par un ensemble de `FSMGuard` « tirables », une `FSMGuard` « tirable » déclenchant un changement d'état si elle est satisfaite (c'est-à-dire si elle est évaluée à vrai). Soulignons que la notion d'état n'est alors pas représentée de manière explicite. Il s'agit d'un choix de conception. Les états auraient pu être représentés explicitement et cela peut notamment être utile si les états peuvent correspondre à des activités du système, comme c'est le cas dans le langage UML.

Pour pouvoir, à partir d'une `FSMGuard`, connaître l'ensemble des `FSMGuard` qui deviennent tirables lorsque celle-ci est satisfaite, une `FSMGuard` comprend un point d'interface particulier appelé `targetState` qui va permettre de la relier, en utilisant des `Relations`, à chacune des `FSMGuard` qui deviennent tirables si elle est satisfaite. Une `FSMGuard` comprend également un point d'interface appelé `sourceState` qui permet à d'autres `FSMGuard` de la désigner en tant que `FSMGuard` tirable.

Pour compléter cette représentation, il nous faut définir ce qu'est un modèle dans le modèle de calcul FSM. Pour cela, nous spécialisons l'élément `Model` de notre méta-modèle générique pour créer un élément `FSMModel`. Un `FSMModel` hérite donc des caractéristiques d'un `Model` mais a, en plus, un état initial et un état courant qui seront chacun représentés par des ensembles de gardes tirables, `initialFireableGuards` et `currentFireableGuards`. Précisons que la figure 4.27 ne fait pas apparaître toutes les caractérisques héritées du méta-modèle générique de ModHel'X (dont une réprésentation complète est donnée par la figure 4.21 p. 91). Ainsi par exemple, `FSMModel` héritant de `Model`, il est possible d'accéder à l'ensemble des blocs qui constituent la structure d'un `FSMModel`, que ceux-ci soient des `FSMGuard` ou des `FSMAction`, grâce à l'association qui existe entre `Model` et `CompositeBlock` et à la relation de composition qui existe entre `CompositeBlock` et `Block` (voir la figure 4.21 p. 91).

Pour illustrer comment la syntaxe spécifique ainsi obtenue peut être utilisée, considérons par exemple l'automate représenté sur la figure 4.28. Il s'agit d'un automate à deux états E1 et E2. La transition de E1 vers E2 dispose d'une garde G1 qui dépend de l'événement $evtx$. Le détail de cette garde n'est pas montré, il peut s'agir d'une expression booléenne complexe. La transition de E2 vers E1 dispose d'une garde G2 qui dépend de l'événement $evty$ et d'une action A2 qui produit un événement $evtz$. La figure 4.29 montre comment l'automate de la figure 4.28 est représenté

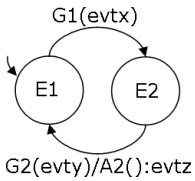


FIG. 4.28 – Exemple d'automate simple à deux états

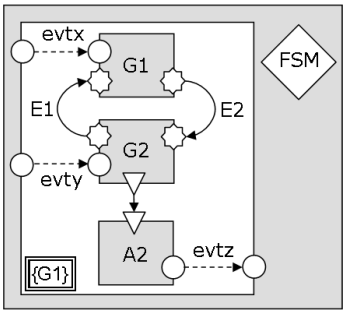


FIG. 4.29 – Exemple de représentation d'un automate simple à deux états dans ModHel'X

dans la syntaxe de ModHel'X en utilisant les concepts spécialisés que nous venons de décrire. Sur cette figure, nous avons utilisé une version étendue de notre syntaxe concrète graphique simple dans laquelle des étoiles et des triangles sont utilisés pour représenter les différents points d'interface particuliers des gardes et des actions. Dans la représentation obtenue, la garde G1 présente sur la transition qui va de E1 à E2 est représentée par la `FSMGuard` G1. Elle n'a pas d'action associée. L'ensemble des gardes qui deviennent tirables lorsque G1 est satisfaite est {G2}. En effet, dans le modèle originel, lorsque l'automate se trouve dans l'état E2, qui est l'état cible de la transition qui porte G1, l'ensemble des transitions tirables contient uniquement la transition portant la garde G2. La garde G2 présente sur la transition qui va de E2 à E1 est représentée par la `FSMGuard` G2. L'action qui lui est associée sur la transition est représentée par la `FSMAction` A2. L'ensemble des gardes qui deviennent tirables lorsque G2 est satisfaite est {G1}. En effet, dans le modèle originel, lorsque l'automate se trouve dans l'état E1, qui est l'état cible de la transition qui porte G2, l'ensemble des transitions tirables contient uniquement la transition qui porte la garde G1. L'état initial de l'automate apparait dans un rectangle en bas à gauche du modèle : il est représenté par l'ensemble des gardes tirables initiales {G1}. En effet, dans le modèle originel, l'état initial de l'automate est E1 et, dans cet état, l'ensemble des transitions tirables contient uniquement la transition qui porte la garde G1.

Soulignons qu'il est possible de définir une syntaxe concrète graphique plus intuitive que celle utilisée sur la figure 4.29 afin que l'automate soit représenté dans ModHel'X de la même manière que l'automate originel présenté sur la figure 4.28. Notons au passage que des contraintes OCL peuvent être utilisées pour compléter le méta-modèle spécifique mis au point pour un MoC particulier. Ces contraintes peuvent notamment être utilisées lors de la vérification de la conformité d'un modèle à ce méta-modèle. Dans la section 4.6.2, nous montrons comment les opérations d'exécution du modèle de calcul FSM peuvent être décrites afin que tout modèle conforme au méta-modèle spécifique représenté sur la figure 4.27 soit interprété comme un automate à états finis.

La syntaxe abstraite obtenue par spécialisation de la syntaxe abstraite générique de ModHel'X est propre au modèle de calcul considéré et fait partie de sa description dans ModHel'X. Cette spécialisation est réalisée par un expert du modèle de calcul. Il s'agit d'une tâche difficile car elle implique d'imaginer comment représenter les concepts du modèle de calcul en utilisant les éléments de notre syntaxe abstraite, en les spécialisant au besoin. Soulignons que différents experts peuvent aboutir à différentes spécialisations de notre syntaxe abstraite pour représenter un même langage : il s'agit réellement d'une tâche de conception et différents choix sont donc possibles.

4.6.2 Sémantique spécifique : concrétisation de la sémantique abstraite

Nous avons vu dans la section 4.5 que notre algorithme définit un modèle d'exécution générique qui constitue une sémantique abstraite pour les modèles de calcul. Ce modèle d'exécution générique est un algorithme dont les étapes sont réalisées par des méthodes des éléments du méta-modèle décrivant notre syntaxe abstraite.

4.6.2.1 Spécialisation de l'élément `ModelOfComputation`

Notre méta-modèle comprend notamment un élément `ModelOfComputation` qui représente le concept de modèle de calcul. Une partie des étapes de notre algorithme d'exécution générique est réalisée par des méthodes de l'élément `ModelOfComputation`, dont notamment la boucle de calcul d'un snapshot. Ces étapes sont génériques, c'est-à-dire que les méthodes correspondantes sont en fait abstraites, et leur sémantique doit être définie pour décrire un modèle de calcul concret.

Pour définir un modèle de calcul particulier, il est nécessaire de commencer par créer une spécialisation de l'élément `ModelOfComputation` dans laquelle les méthodes correspondant seront concrètes. Prenons l'exemple du modèle de calcul des machines à états finis (FSM). Pour implémenter ce modèle de calcul dans notre approche il faut créer un élément `FSMMoC` qui hérite de `ModelOfComputation`. Cet élément fait partie du méta-modèle spécifique du modèle de calcul tel que nous l'avons introduit dans la section précédente. Pour obtenir une sémantique concrète pour ce modèle de calcul à partir de notre modèle générique d'exécution, il est nécessaire de décrire le corps des méthodes de cet élément. Nous proposons pour cela d'utiliser un langage informatique, que nous introduisons dans la section suivante.

Remarquons ici que les méthodes d'exécution d'un modèle de calcul sont décrites au niveau méta. Cela permet en particulier que toutes les instances de ce modèle de calcul calculent de la même manière une observation d'un modèle. Notons également qu'un modèle de calcul n'est pas un singleton et que plusieurs instances d'un même modèle de calcul peuvent donc être impliquées dans un même modèle sans pour autant qu'elles n'interfèrent.

4.6.2.2 Problématique : langage de spécification des modèles de calcul

Nous décrivons l'exécution des modèles de manière constructive. Il faut donc que les expressions du langage utilisé pour décrire les étapes de notre algorithme soient des expressions à effet de bord, c'est-à-dire que leur évaluation modifie l'état du modèle. De plus, ce langage doit également permettre de naviguer sur les différents éléments d'un modèle. Par ailleurs, comme notre approche est une approche de modélisation, notre langage doit avoir un niveau d'abstraction suffisant pour cacher un certain nombre de détails d'implémentation de l'exécution. Enfin, nous souhaitons pouvoir, à terme, utiliser notre approche pour la vérification et la validation de modèles. Cela implique de disposer d'une description formelle de la sémantique de chaque modèle de calcul dans notre approche. Pour cela, nous travaillons sur la formalisation de la sémantique de notre algorithme d'exécution. Comme cet algorithme est complété par des descriptions des opérations pour chaque modèle de calcul, il est également nécessaire que notre langage dispose d'une sémantique formellement définie.

Créer un nouveau langage spécifique pour notre approche ne nous a pas semblé judicieux. Il se serait ajouté à la multitude des langages créés pour de multiples besoins spécifiques. Nous avons donc étudié différentes possibilités parmi des langages existants les plus matures et les plus répandus du domaine de la modélisation dirigée par les modèles, et en particulier de l'approche MDA pour bénéficier de l'effort de standardisation de l'OMG.

Les langages de transformation de modèle répondent en partie à nos besoins tels qu'énoncés ci-dessus. En effet, ceux-ci permettent de naviguer sur les différents éléments d'un modèle et permettent de spécifier comment ces éléments sont modifiés par la transformation. Comme ils

permettent de travailler au niveau des modèles, ils ont en général un niveau d'abstraction relativement élevé. De plus, les transformations de modèles font l'objet d'un standard de l'OMG appelé QVT (MOF Query/Views/Transformations) [OMGe]. De nombreux langages de transformation de modèles existent, parmi lesquels ATL [JAB+06], SmartQVT [DB], GReAt [fSISIVU], MOLA [KBC04] ou encore Story Diagrams [FNTZ00].

Bien qu'il ne s'agisse pas à proprement parler d'un langage de transformation de modèles, Kermeta [MFJ05] (voir la section 3.3.2.1) correspond également à certains de nos critères. En effet, c'est un langage impératif qui permet de décrire de manière opérationnelle la sémantique d'un langage dont les concepts sont représentés par un méta-modèle .

Nous avons finalement choisi d'utiliser un sous-ensemble du langage ImperativeOCL. Ce langage, spécifié dans la partie Operational QVT du standard QVT, est une extension impérative d'OCL. SmartQVT en est une implémentation. Notre choix s'est porté sur ce langage car :

- il fait partie de la spécification du standard QVT de l'OMG ;
- il s'appuie sur le langage standard OCL de l'OMG qui :
  - a un niveau d'abstraction élevé ;
  - est spécialement conçu pour permettre la navigation sur les éléments d'un modèle ;
  - dispose d'une sémantique formelle partielle (non normative) qui peut, à terme, servir de base à la formalisation de la version impérative.

Cependant, ImperativeOCL définit un grand nombre d'expressions à la sémantique complexe. Pour faciliter une définition formelle future, nous avons choisi de n'utiliser qu'un sous-ensemble d'ImperativeOCL. Nous donnons une description succincte du sous-ensemble que nous avons retenu dans l'annexe B.

4.6.2.3 Exemple du modèle de calcul des machines à états finis (FSM)

Reprenons l'exemple du modèle de calcul des machines à états finis (FSM) que nous avons introduit dans la section 4.6.1.2. Nous considérons dans cet exemple une version simple de FSM dans laquelle :

- les automates sont déterministes, c'est-à-dire qu'à partir d'un état donné, une seule transition sortante est possible lors d'une réaction ;
- les événements produits au cours d'une réaction ne sont pas pris en compte dans la réaction courante mais seulement dans la réaction suivante ;
- il n'y a pas de notion de « run-to-completion » : à partir du moment où l'automate change d'état, la réaction est terminée et les autres événements éventuellement présents sont perdus.

Nous considérons cette version simplifiée pour l'exemple, mais une version plus complexe de ce modèle de calcul peut très bien être implémentée dans notre approche. Dans la section 4.6.1.2, nous avons présenté comment un automate à état fini simple pouvait être représenté en utilisant notre syntaxe abstraite. Rappelons simplement ici que les gardes et les actions présentes sur les transitions d'un automate sont représentées par deux types de blocs : `FSMGuard` et `FSMAction`. La mise à jour (c'est-à-dire l'appel de l'opération `udpate`) sur une `FSMGuard` permet d'observer le résultat de l'évaluation de la garde. La mise à jour (c'est-à-dire l'appel de l'opération `udpate`) sur une `FSMAction` permet d'observer le résultat de l'action que le bloc représente. Nous détaillons dans les paragraphes suivants comment décrire la sémantique d'exécution du modèle de calcul FSM dans notre approche.

Tout d'abord, comme nous l'avons introduit ci-dessus, nous complétons le méta-modèle spécifique de FSM avec l'élément `FSMMoC` qui spécialise `ModelOfComputation`, comme cela est représenté sur la figure 4.30. Puis nous devons déterminer le contenu des opérations d'exécution de ce modèle de calcul et, éventuellement, les variables de calcul dont il a besoin. Pour cela, il nous faut détailler comment un snapshot est réalisé sur un modèle représentant un automate.

Le comportement spécifié par un automate est composé de changements d'états. Dans la version simple du modèle de calcul FSM que nous considérons, un changement d'état d'un automate intervient lorsque l'arrivée d'un événement déclenche une garde sur une transition. Il faut donc réaliser un snapshot chaque fois qu'un événement arrive en entrée du modèle car celui-ci peut déclencher une garde. Calculer un snapshot c'est dans ce cas mettre à jour (c'est-à-dire appeler l'opération `update` sur) toutes les gardes tirables de l'état courant afin de voir si l'une d'elles s'évalue à vrai. Deux cas sont alors possibles : soit une des gardes s'évalue à vrai et l'automate change alors d'état, soit aucune garde ne s'évalue à vrai et l'automate reste dans l'état courant. Dans le premier cas, il faut alors mettre à jour la/les action(s) associée(s) à la garde tirée et modifier l'état courant de l'automate. Le snapshot est terminé lorsque toutes les actions ont été mises à jour. Dans le deuxième cas, le snapshot est terminé lorsque toutes les gardes ont été mises à jour.

Les actions à réaliser au cours du calcul d'un snapshot sont donc les suivantes :

- Au début du snapshot, il faut déterminer l'ensemble des gardes tirables à partir de l'état courant du modèle. L'ensemble des blocs à mettre à jour pendant le calcul est donc alors l'ensemble des gardes tirables à partir de l'état courant du modèle.
- A chaque tour de boucle du calcul du snapshot (voir la section 4.5.4), il faut choisir un bloc à mettre à jour (c'est-à-dire sur lequel appeler l'opération `update`).
- Après la mise à jour de ce bloc, s'il s'agit d'une garde, il faut vérifier si cette garde s'est évaluée à vrai. Dans la représentation que nous avons choisie, lorsqu'une garde s'évalue à vrai elle produit un jeton sur son point d'interface `enabledAction`. Si c'est le cas, l'automate change d'état c'est-à-dire que l'ensemble des gardes tirables change, et il faut mettre à jour les actions associées à l'état. Dans les tours de boucle suivant il ne faudra donc plus mettre à jour les gardes tirables mais les actions associées à la garde qui a été tirée. Dans le cas où la garde mise à jour ne s'est pas évaluée pas à vrai, alors il faut simplement, dans les tours de boucle suivants, continuer de mettre à jour les gardes de l'ensemble des gardes tirables.
- Le snapshot termine soit lorsque toutes les gardes ont été mises à jour (c'est le cas si aucune ne s'est évaluée à vrai) ou lorsque toutes les actions associées à la garde tirée ont été mises à jour.

Dans ce contexte, le modèle de calcul `FSMMoC` a besoin, pour réaliser le calcul d'un snapshot, des variables suivantes (représentées sur la figure 4.30) :

- `fireableGuards` : variable qui, au début du snapshot, contient l'ensemble des gardes tirables courantes et, à la fin du snapshot, contient l'ensemble des gardes tirables suivantes. Cette variable est donc initialisée avec l'état du modèle (attribut `currentFireableGuards` sur l'élément `FSMModel`) au début du snapshot, et permet de modifier l'état du modèle à la fin du snapshot (rôle respectif des opérations `startOfSnapshot` et `endOfSnapshot`).
- `blocksToUpdate` : variable qui contient l'ensemble des blocs à mettre à jour dans le snapshot courant. Au début du snapshot, elle contient l'ensemble des gardes tirables courantes. Après chaque mise à jour, la garde mise à jour est retirée de l'ensemble. Si une garde s'évalue à vrai (c'est-à-dire produit un jeton sur son point d'interface `enabledAction`), cet ensemble contient alors l'ensemble des actions associées à cette garde. Après chaque mise à jour, l'action mise à jour est retirée de l'ensemble. Le snapshot est terminé lorsque cet ensemble est vide.
- `currentBlock` : variable qui contient le bloc à mettre à jour dans le tour de boucle courant.

Nous présentons ci-dessous les algorithmes en ImperativeOCL des opérations `preSchedule` et `interSchedule` pour le modèle de calcul FSM. Le code complet du modèle de calcul FSM est présenté en Java dans l'annexe C.1.

L'opération `preSchedule` (algorithme 4.1) réalise simplement le choix du bloc à mettre à jour parmi les blocs de la variable `blocksToUpdate`. L'opération `interSchedule` (algorithme 4.2)

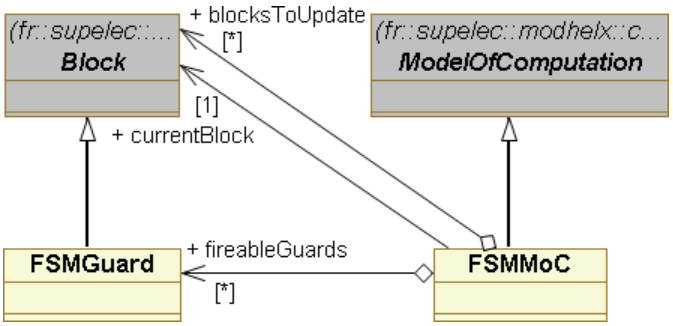


FIG. 4.30 – Spécialisation de l'élément `ModelOfComputation` pour le modèle de calcul FSM

vérifie le résultat de la mise à jour du bloc choisi. C'est elle qui modifie l'ensemble des blocs à mettre à jour, ainsi qu'éventuellement l'ensemble des gardes tirables, à chaque tour de boucle du calcul du snapshot. Les autres opérations d'ordonnancement sont laissées vides pour ce modèle de calcul car elles n'ont pas de rôle particulier à jouer dans le calcul. Les opérations de propagation sont vides également. En effet, le mode de communication est extrêmement simple dans notre version simplifiée de FSM : nous avons choisi de ne pas tenir compte des événements produits dans la réaction courante. En conséquence, les seuls événements à prendre en compte dans une réaction sont ceux présents en entrée du modèle. Les points d'interface des gardes peuvent donc être des alias des entrées du modèle de façon à ce que celles-ci reçoivent directement les événements présents en entrée du modèle. De la même manière, les points d'interface des actions sont des alias des sorties du modèle de façon à ce que les événements produits au cours de la réaction soient directement disponibles sur les sorties du modèle. L'inconvénient de cette méthode est que les événements produits en sortie qui pourraient faire réagir l'automate doivent être présentés en entrée du modèle au snapshot suivant. Dans une version plus évoluée de ce modèle de calcul, une file d'événements pourrait être gérée par le `FSMMoC` afin de supporter la prise en compte des événements produits au cours de la réaction. Cela permettrait également de supporter le mécanisme de « run-to-completion ».

Listing 4.1: FSMMoC::preSchedule(m:Model)

```
// The block to update is one of the blocks of the "blocksToUpdate" set
self.currentBlock := self.blocksToUpdate→first();
```

Listing 4.2: FSMMoC::interSchedule(m:Model)

```
if (self.currentBlock.oclIsKindOf(FSMGuard){ // If the block that we have just updated is a guard
        if (self.currentBlock.enabledAction.storedTokens→size()>0){ // If the guard evaluated to true
                self.currentBlock.enabledAction.storedTokens→clear(); // The enabling token is consumed

                self.fireableGuards→clear(); // The state of the automaton changes
                self.currentBlock.targetState.outcomingRelations→forEach(r){ // and is composed of the target guards
                        self.fireableGuards→add(r.target.isInputForBlock); // of the current guard
                }

                self.blocksToUpdate→clear(); // And the blocks to be updated next
                self.currentBlock.enabledAction.outcomingRelations→forEach(r){ // are the enabled actions
                        self.blocksToUpdate→add(r.target.isInputForBlock); // of the current guard
                        r.target.storedTokens→add(new FSMEvent()); // An enabling token is provided to each enabled
                          action
                }
        }else{ // Else, if the guard evaluated to false
                self.blocksToUpdate→remove(self.currentBlock); // It is not to update anymore
        }
} else { // Else, if the block that we have just updated is an action
        self.blocksToUpdate→remove(self.currentBlock); // It is not to update anymore
}
```

4.6.3 Description de l'adaptation sémantique entre deux modèles de calcul

Une fois plusieurs modèles de calcul décrits dans notre approche, il est possible de les utiliser pour créer des modèles (homogènes ou hétérogènes). Nous avons vu que deux modèles impliquant des modèles de calcul différents peuvent être assemblés grâce à un bloc d'interface. C'est ce bloc d'interface qui réalise l'adaptation sémantique nécessaire entre ces deux modèles. Cette adaptation sémantique est spécifiée par le concepteur lorsqu'il assemble ses deux modèles. Pour cela, il utilise le langage que nous avons introduit dans la section précédente et décrit les opérations `adaptIn` et `adaptOut` du bloc d'interface utilisé. Nous décrivons un exemple d'adaptation sémantique paramétrable réalisée par un bloc d'interface dans la section 5.3.4 du chapitre 5.

Il est important de souligner ici que l'adaptation sémantique utilisée entre deux modèles impliquant deux modèles de calcul différents peut donc être complètement spécifiée par le concepteur au moment de l'assemblage, et ce en dehors des modèles assemblés. Cependant, il existe souvent des façons usuelles d'assembler des modèles hétérogènes. Afin de ne pas obliger le concepteur à décrire à nouveau de telles politiques d'adaptation à chaque utilisation, nous donnons la possibilité de définir des *patrons d'adaptation sémantique*. Chaque patron est réalisé sous la forme d'un bloc d'interface dont les opérations `adaptIn` et `adaptOut` décrivent une méthode usuelle d'adaptation pour une paire de modèles de calcul donnée. Le corps de ces opérations est spécifié à l'aide du langage impératif introduit dans la section précédente. Ces opérations peuvent faire appel à des paramètres, qui permettront d'ajuster l'adaptation réalisée dans le contexte d'un modèle particulier. Un tel bloc d'interface peut être simplement instancié et placé directement dans un modèle (après lui avoir attribué un modèle interne). Modifier la valeur de ses paramètres permet un premier niveau de personnalisation. Il est également possible de personnaliser l'adaptation réalisée par ce bloc d'interface en créant un nouveau bloc d'interface qui le spécialise (toujours au sens de l'héritage) de manière à pouvoir ajouter des attributs, des paramètres et/ou redéfinir ses opérations d'adaptation.

Afin d'illustrer cette notion de patron d'adaptation sémantique, prenons l'exemple d'un modèle hiérarchique à deux niveaux. Le modèle le plus haut dans la hiérarchie met en œuvre un modèle de calcul à temps continu, tandis que le modèle le plus bas dans la hiérarchie met en œuvre un modèle de calcul à temps discret. Il faut donc décrire l'adaptation sémantique réalisée par le bloc d'interface entre les MoCs utilisés dans chacun des modèles. Plusieurs patrons d'adaptation paramétrables sont envisageables. Par exemple, un premier patron classique est d'échantillonner avec une période paramétrable les signaux fournis par le modèle à temps continu afin de les adapter en entrée du modèle à temps discret et de maintenir les dernières valeurs discrètes produites en sortie du modèle à temps discret afin de les adapter pour le modèle à temps continu. Une variante de ce patron serait d'interpoler à l'ordre 1 (ou à un ordre supérieur) les valeurs discrètes produites en sortie du modèle à temps discret afin de les adapter pour le modèle à temps continu. Ces deux patrons sont utilisables directement dans un modèle et peuvent être paramétrés selon les besoins spécifiques du modèle considéré.

Une bibliothèque de patrons d'adaptation peut donc être constituée, et ainsi faciliter la réalisation de modèles hétérogènes par les concepteurs.

4.7 Positionnement et discussion

Après avoir présenté en détails notre approche pour la composition de modèles hétérogènes, nous proposons de la comparer avec les approches les plus similaires présentées dans le chapitre 3. Nous nous positionnons notamment par rapport aux approches de composition de sémantique et d'agrégation de modèles vues dans la section 3.3.4. Nous discutons également les apports et les limites de notre approche.

4.7.1 ModHel'X et Ptolemy

Ptolemy II (voir la section 3.3.4.4) a été une source d'inspiration dans la conception de ModHel'X. En conséquence, ModHel'X et Ptolemy II sont similaires dans leurs concepts sous-jacents. Cependant, nous proposons des contributions sur différents aspects par rapport à l'approche développée dans Ptolemy II.

Tout d'abord, la description des mécanismes d'adaptation utilisés entre deux modèles impliquant des modèles de calcul différents est explicite dans ModHel'X. Ce mécanisme permet au concepteur d'avoir la main sur l'adaptation qui est réalisée et de pouvoir la modifier le cas échéant. Cela permet également une meilleure réutilisabilité des modèles.

Nous avons également réalisé un changement de paradigme d'exécution par rapport à Ptolemy. En effet, dans Ptolemy, l'exécution est réalisée en « déclenchant » (fire) des acteurs tandis que dans ModHel'X, l'exécution est réalisée en observant des blocs. Grâce à ce changement de paradigme et à l'ajout d'un mécanisme de contraintes temporelles, même les niveaux les plus bas de la hiérarchie d'un modèle peuvent influer sur le cours de l'exécution. Dans Ptolemy, l'exécution du modèle est pilotée par le niveau racine du modèle.

Par ailleurs, si Ptolemy comme ModHel'X fournissent tous deux un cadre pour la description de la sémantique des modèles de calcul (une « sémantique abstraite »), leurs algorithmes d'exécution respectifs sont très différents. Les différences majeures proviennent du changement de paradigme d'exécution : l'un calcule des déclenchements d'acteurs et l'autre des observations de blocs. Une autre différence provient de la modélisation explicite de l'adaptation sémantique entre modèles. Enfin, nous avons inclus une étape de validation dans l'algorithme de ModHel'X, qui permet de re-calculer un snapshot si celui-ci n'est pas validé (voir la section 4.5.7). Cette étape existe dans le domaine CT (Continuous Time) de Ptolemy II uniquement, nous l'avons généralisée dans ModHel'X afin qu'elle existe dans tous nos modèles de calcul.

Remarquons enfin que, tout comme Ptolemy, ModHel'X est conçu pour supporter la gamme la plus large possible de langages de modélisation. Cela implique que son algorithme d'exécution peut être moins performant qu'un algorithme optimisé pour un langage donné. Par ailleurs, pour supporter la spécification explicite de l'adaptation sémantique, nous avons découpé notre algorithme en étapes de grain plus fin que celles de Ptolemy. Nous prévoyons d'étudier l'impact de ce choix sur les performances.

4.7.2 ModHel'X et la composition de « Semantic Units »

Comme nous l'avons vu dans les sections 3.3.2.2 et 3.3.4.3, les « Semantic Units (SUs) » permettent d'attacher une sémantique formelle à un méta-modèle représentant la syntaxe abstraite d'un langage. Une SU « primaire » capture les éléments sémantiques de base d'une catégorie particulière de langages, c'est-à-dire un modèle de calcul. Les auteurs proposent des mécanismes permettant de composer différentes SUs primaires afin de former des SUs plus complexes appelées SU dérivées, qui peuvent elles-mêmes être composées avec d'autres SUs.

Ce mécanisme est particulièrement intéressant du point de vue de la réutilisabilité car les SUs primaires les plus courantes sont décrites une fois pour toutes. De plus, une SU dérivée définit en fait un autre modèle de calcul, c'est-à-dire un nouveau langage de modélisation, directement utilisable dans différents modèles. Dans ModHel'X, le mécanisme de composition de modèles de calcul est l'abstraction hiérarchique via le concept de bloc d'interface. S'il est possible de réutiliser un bloc d'interface et de changer ses paramètres pour l'adapter à un contexte différent, ce bloc d'interface ne définit pas un nouveau langage de modélisation. Pour définir un nouveau modèle de calcul sur la base de modèles de calcul primaires dans ModHel'X, il est nécessaire de le décrire entièrement (syntaxe et sémantique).

A contrario, lorsque l'on veut composer un modèle impliquant une SU A avec un modèle impliquant une SU B, il est nécessaire de décrire d'abord la composition des deux SU (donnant

la SU dérivée AB) puis de modifier et recoller les modèles de manière à ce qu'ils soient conformes à cette nouvelle SU. Dans ModHel'X, un bloc d'interface permet de recoller facilement et sans les modifier deux modèles faisant appel à des modèles de calcul A et B.

Ces deux mécanismes sont donc complémentaires : la composition des SU permet une bonne réutilisabilité des descriptions des modèles de calcul et la composition hiérarchique de modèles hétérogènes telle que nous l'implémentons dans ModHel'X permet une bonne réutilisabilité des modèles.

4.7.3 Coût d'utilisation de ModHel'X

Comme nous l'avons vu dans la section 4.6 précédente, l'ajout de langages de modélisation dans ModHel'X se fait en deux étapes.

Tout d'abord, un expert d'un langage de modélisation doit décrire les éléments de structure et de sémantique de ce langage en spécialisant notre syntaxe et notre sémantique abstraites. De plus, puisque notre objectif n'est pas de remplacer les outils existants mais bien de permettre leur utilisation conjointe de manière contrôlée, cet expert définit également des transformations entre le méta-modèle original du langage et notre méta-modèle. Ces tâches sont les plus difficiles à réaliser car elles nécessitent à la fois une très bonne maîtrise du langage de modélisation concerné et une bonne connaissance de notre approche.

Puis, pour chaque paire de modèles de calcul pouvant être mis en relation dans un modèle hétérogène, des experts doivent définir des patrons d'interaction, qui modélisent des manières « standard » de combiner des modèles qui obéissent à ces modèles de calcul. La politique d'interaction utilisée dans un modèle particulier sera une spécialisation de l'un de ces patrons, adaptée en utilisant des paramètres.

Ces étapes représentent l'effort principal nécessaire pour bénéficier de l'approche ModHel'X. La première étape est réalisée une fois pour toutes pour chacun des langages de modélisation que l'on souhaite utiliser. La conception de patrons d'interaction dépend à la fois des domaines techniques en jeu et des habitudes des concepteurs puisqu'il s'agit de formaliser le savoir-faire et leur manière habituelle de résoudre les problèmes d'hétérogénéité. Il est nécessaire de définir au moins un patron d'adaptation pour chaque paire de modèles de calcul que l'on souhaite faire interagir dans un modèle. Cependant, il est inutile de définir un tel patron pour toutes les paires de modèles de calcul possible. En effet, faire interagir certains modèles de calcul peut ne pas avoir de sens. D'autre part, même lorsqu'ils sont utilisés ensemble dans un modèle, certains modèles de calcul n'interagissent jamais directement. En conséquence, le nombre de combinaisons de modèles de calcul à spécifier est bien moins important que le nombre de combinaisons qui sont théoriquement possibles lorsque l'on considère N modèles de calcul.

4.7.4 Modèles de calcul supportés

Considérer qu'une même structure de modèle puisse être interprétée comme un automate ou comme un modèle à événements discrets suivant le modèle de calcul qui lui est associé peut paraître un peu extrême. Cependant, ce choix semble intéressant puisque Ptolemy supporte sur cette base des paradigmes de modélisation aussi différents que les machines à états finis, les équations différentielles ou les réseaux de processus. C'est un choix que nous avons également adopté dans ModHel'X. Ainsi, ModHel'X supporte, comme Ptolemy, une large gamme de modèles de calcul.

4.7.4.1 Modèles de calcul à temps continu

Cette gamme de modèles de calcul inclut les modèles de calcul à temps continu. Comme pour Ptolemy, les comportements continus sont approximés grâce à une discrétisation qui est

calculée par un solveur. Dans ModHel'X, les différents instants discrets correspondent à des snapshots successifs. L'étape de validation de notre algorithme permet de recalculer un snapshot si l'approximation obtenue n'est pas satisfaisante vis-à-vis d'un intervalle d'erreur par exemple (voir la section 4.5.7).

4.7.4.2 Modèles de calcul avec dépendances cycliques

Les modèles de calcul qui permettent les dépendances cycliques dans les modèles sont également supportés par ModHel'X. En effet, la boucle de calcul d'un snapshot permet d'itérer vers un point fixe et ainsi de supporter des modèles de calcul tel que SR (Synchronous/Reactive). Remarquons que l'on n'a de garantie d'atteindre le point fixe systématiquement que si tous les blocs sont monotones par rapport à un ordre partiel défini par le modèle de calcul.

4.7.4.3 Modèles de calcul non déterministes

Puisque les opérations de notre algorithme sont invoquées séquentiellement, le calcul d'un snapshot est normalement déterministe. Cependant, il existe des modèles de calcul dans lesquels le non-déterminisme est souhaité. Par exemple, certains formalismes à base de machines à états finis autorisent d'avoir, dans un automate, plusieurs transitions avec des gardes identiques sortant d'un même état. Lorsque la garde est validée, le choix de la transition à prendre est aléatoire. Notre approche doit donc permettre l'expression de tels modèles de calcul.

Nous permettons l'expression du non-déterminisme dans la description de modèles de calcul en autorisant l'utilisation de fonctions du type `random` qui rendent un élément choisi pseudo-aléatoirement dans un ensemble. Remarquons que des modèles de calcul non déterministes peuvent être utiles pour simuler un système mais ne sont pas appropriés pour des applications formelles telles que le test, le model-checking ou la validation.

4.7.5 Langage de description de modèles de calcul

Comme présenté dans la section 4.6.2, nous avons choisi un sous-ensemble du langage ImperativeOCL pour décrire les étapes de notre modèle d'exécution générique. Ce choix présente cependant différents inconvénients.

Tout d'abord, nous avons constaté à l'usage que les descriptions réalisées avec ce langage étaient souvent très verbeuses. Nous aimerions, à terme, utiliser un langage beaucoup plus concis et d'un niveau d'abstraction plus élevé.

Par ailleurs, nous souhaitions un langage dont la sémantique soit définie formellement (fondée mathématiquement) afin de pouvoir utiliser notre approche pour la vérification et la validation de modèles dans le futur. Même si une sémantique formelle existe pour OCL (bien qu'elle ne soit pas normative), les extensions impératives définies dans ImperativeOCL ne sont pas fondées mathématiquement. Définir une sémantique formelle pour un langage est une tâche difficile, et ImperativeOCL définit un grand nombre d'expressions à la sémantique complexe. Pour faciliter une définition formelle future, nous avons choisi de n'utiliser qu'un sous-ensemble d'ImperativeOCL.

Cependant, le fait d'avoir choisi un sous-ensemble de ce langage nous contraint dans le choix des outils supports (éditeurs, interpréteurs, compilateurs...). Dans les faits, il existe encore peu d'outils supportant ImperativeOCL. De plus, avoir choisi de n'utiliser qu'un sous-ensemble du langage nécessite d'adapter l'outil choisi (voir le chapitre suivant sur l'implémentation de notre approche).

En conséquence, nous ne considérons pas ce choix comme définitif pour le moment et étudions d'autres solutions potentielles.

4.7.6 Expression des mécanismes de combinaison de modèles de calcul

Les mécanismes de combinaison de modèles de calcul, représentés par les opérations d'adaptation sémantique des blocs d'interface, sont également spécifiés en utilisant notre sous-ensemble d'ImperativeOCL. Or, dans la pratique, ces mécanismes de combinaison font partie des modèles, c'est-à-dire qu'ils sont visibles et modifiables par les concepteurs, et pas seulement par les experts des langages de modélisation. En termes d'ergonomie, il serait souhaitable que ces mécanismes de combinaison soient en fait décrits comme des modèles et non pas comme des opérations d'exécution. Notamment, nous envisageons à terme de pouvoir décrire ces mécanismes de combinaison en utilisant la même syntaxe que celle utilisée pour décrire les modèles, c'est-à-dire notre syntaxe abstraite. Ainsi, ces mécanismes d'adaptation pourraient être décrits en utilisant des blocs, des relations, etc., et la syntaxe concrète pourrait également être graphique.

4.7.7 Transmission d'informations à travers plusieurs niveaux hiérarchiques

Les blocs d'interface réalisent une adaptation sémantique locale entre deux modèles impliquant deux modèles de calcul différents. Le fait que l'adaptation soit réalisée de manière locale pour des paires de modèles de calcul simplifie l'expression des mécanismes de combinaison puisque seules deux sémantiques (et non N) sont à combiner. Nous avons vu que cela favorisait également la modularité et la réutilisabilité.

Cela comporte cependant un inconvénient majeur : lors de l'adaptation réalisée par un bloc d'interface, certaines informations (étiquettes de temps, données) peuvent être supprimées par nécessité car elles ne sont pas compatibles avec la sémantique du modèle qui est en contact via le bloc d'interface. Or ces informations peuvent être sémantiquement pertinentes ou même nécessaires dans un modèle situé à un autre niveau de la hiérarchie. Prenons l'exemple d'un modèle constitué de trois niveaux hiérarchiques : dans le niveau 1 (modèle racine), le modèle de calcul utilise des dates du temps réel, dans le niveau 2 (modèle intermédiaire), seule la notion d'événement à un sens pour le modèle de calcul utilisé, et dans le niveau 3 (modèle bas), le modèle de calcul gère des événements datés dans le temps réel. Lors de l'adaptation de données provenant du modèle bas vers le modèle intermédiaire, les dates sont retirées car elles n'ont pas de sens dans la sémantique du modèle de calcul intermédiaire. Cependant, lors de l'adaptation des données provenant du modèle intermédiaire vers le modèle racine, une date du temps réel doit permettre d'étiqueter les informations transmises. Or cette information a été perdue dans le modèle intermédiaire. Cet exemple illustre le fait qu'il est parfois nécessaire de permettre la transmission d'informations à travers des niveaux hiérarchiques dans lesquels ces informations n'ont pas de sens. Cela est particulièrement vrai concernant les informations temporelles.

Parmi les solutions que nous envisageons pour résoudre ce problème, deux types de mécanismes nous semblent intéressants : permettre de masquer (et non supprimer) les informations lors de l'adaptation sémantique ou permettre de définir des relations globales (notamment temporelles) entre les sémantiques des modèles de calcul utilisés dans un modèle hétérogène. Dans ce deuxième type de solution, il pourrait être envisageable d'utiliser un mécanisme similaire aux contraintes entre horloges définies dans MARTE (voir la section 4.5.10). Cependant, cela pose la question de la résolution globale des contraintes.

4.7.8 Composabilité

Dans ModHel'X, le problème de la composabilité se pose à deux niveaux : lors de la composition de différents blocs selon un modèle de calcul, et lors de la composition de deux modèles impliquant deux modèles de calcul différents via un bloc d'interface.

Dans le premier cas, il est nécessaire que les deux blocs soient composables dans la sémantique du modèle de calcul choisi. Il faut également pouvoir s'assurer que le modèle de calcul tel qu'il a été spécifié dans ModHel'X réalise correctement la composition.

Dans le deuxième cas il est nécessaire que les sémantiques des deux modèles de calcul mis en contact via le bloc d'interface soient composables, c'est-à-dire qu'un sens puisse être donné à la composition de ces deux modèles de calcul. Cela signifie, dans notre contexte, qu'il faut qu'une intersection entre ces deux sémantiques puisse être trouvée pour exprimer l'adaptation sémantique réalisée par le bloc d'interface. Dans un deuxième temps, il faut également pouvoir s'assurer que le bloc d'interface tel qu'il a été spécifié dans ModHel'X réalise correctement la composition.

Ces problèmes sont particulièrement difficiles. Nous avons restreint l'objectif de nos travaux dans un premier temps à la réalisation de ces compositions. Nous ne proposons donc pas d'outils permettant de vérifier la composabilité ni de s'assurer de la correction de la composition réalisée. Cependant un certain nombre de techniques pourront être intégrées à notre approche dans le futur pour rendre ces services. En effet, nous avons vu dans la section 3.3.5 du chapitre précédent que différentes approches existaient concernant le problème de la composabilité des composants. Le problème de la composabilité des modèles de calcul a également été exploré, notamment à travers l'utilisation d'automates d'interface dans le contexte de Ptolemy. Par contre, le problème de la correction de la spécification d'un modèle de calcul n'a, à notre connaissance, pas été abordé.

4.7.9 Conformité entre un modèle d'exécution et un modèle de calcul

Nous avons proposé dans la section 4.3.3 le concept de modèle d'exécution. Dans notre terminologie, un modèle d'exécution est une spécialisation opérationnelle d'un modèle de calcul. En ce sens, un modèle d'exécution peut être considéré comme un raffinement exécutable d'un modèle de calcul. Dans ce contexte se pose donc la question de la conformité d'un modèle d'exécution par rapport au modèle de calcul qu'il implémente.

Vérifier cette conformité est un problème d'autant plus difficile que modèles d'exécution comme modèles de calcul sont des modèles de sémantiques et non des modèles de systèmes. Deux solutions sont a priori envisageables pour vérifier cette conformité : utiliser un jeu de modèles « témoin » sur lesquels appliquer le modèle d'exécution d'une part et le modèle de calcul d'autre part pour comparer la façon dont ceux-ci est interprété, ou comparer directement les sémantiques définies respectivement par le modèle d'exécution et le modèle de calcul.

Le premier type de solution est du domaine du test. Les problématiques soulevées par ce type de solution sont des problématiques classiques du test et concernent notamment le choix du jeu de modèles « témoin » et des scénarios de test. Il est par ailleurs nécessaire de disposer d'une description « exploitable » du modèle de calcul (c'est-à-dire qui permette d'obtenir une interprétation du modèle témoin qui soit comparable à celle calculée par le modèle d'exécution), ce qui n'est pas évident.

Le deuxième type de solution nécessite de disposer de définitions mathématiques pour le modèle d'exécution et le modèle de calcul, ce qui est difficile à obtenir et également difficile à comparer.

Le fait qu'il existe une relation d'abstraction/raffinement entre un modèle d'exécution et un modèle de calcul permet également d'envisager une perspective intéressante pour notre approche : nous souhaiterions, à terme, être capables de construire automatiquement l'implémentation en ModHel'X d'un modèle d'exécution qui raffine un modèle de calcul donné. Nous avons vu dans le chapitre 2 que ce type d'opération était réalisable par transformation mais posait cependant de nombreuses difficultés.

4.8 Conclusion

Dans ce chapitre, nous avons présenté une approche de la composition de modèles hétérogènes orientée vers l'exécution de modèles. Cette approche, appelée ModHel'X, repose sur le concept de

Modèle de Calcul (MoC). Nous avons développé une syntaxe abstraite permettant de représenter des modèles hiérarchiques impliquant différents modèles de calcul. Les mécanismes d'adaptation sémantique entre les différents modèles de calcul du modèle sont spécifiés explicitement via une structure de modélisation particulière appelée « bloc d'interface ». Nous avons également développé une sémantique abstraite pour les modèles de calcul sous la forme d'un algorithme générique d'exécution. L'instanciation de cet algorithme et la description de ses étapes génériques permet de décrire de manière opérationnelle la sémantique d'un modèle de calcul. Nous proposons une méthode d'utilisation de notre approche, également décrite dans ce chapitre. Enfin, après avoir positionné notre approche vis-à-vis de travaux existants, nous avons mis en lumière ses avantages et ses limites.

Par l'application des paradigmes des boîtes noires et des snapshots, l'approche que nous proposons dans cette thèse se démarque d'autres travaux existants. Les principales contributions présentées dans ce chapitre – faisant partie de notre approche – sont, d'une part, le concept de bloc d'interface et, d'autre part, l'algorithme générique d'exécution hiérarchique avec étape de validation. L'utilisation de contraintes de temps sur un modèle générique de temps inspiré de MARTE est également un élément original de notre approche. Dans le chapitre suivant, nous décrivons une implémentation de cette approche sous la forme d'un framework, lui aussi appelé ModHel'X (framework pour la Modélisation Hétérogène & eXécutable). Nous illustrons l'utilisation de ce framework et l'application de notre approche sur un cas d'étude.

5

Application et validation : le framework ModHel'X

5.1 Introduction

Dans le chapitre précédent, nous avons présenté en détails l'approche de composition de modèles que nous proposons. Le présent chapitre décrit l'implémentation que nous avons réalisée de cette approche. Celle ci se présente sous la forme d'un framework : le framework ModHel'X (framework pour la Modélisation Heterogène & eXécutable). Nous présentons tout d'abord comment nous avons conçu ce framework, en détaillant les différents cas dans lequel il doit pouvoir être utilisé et en expliquant les choix techniques qui en ont découlé. Nous exposons ensuite les difficultés rencontrées lors de la réalisation effective du framework et enfin nous présentons l'architecture finale de l'implémentation réalisée.

Grâce à ce framework, nous avons pu réaliser différentes expérimentations sur notre approche. Nous avons notamment décrit différents modèles de calcul et réalisé puis simulé différents modèles, homogènes comme hétérogènes. Nous présentons dans ce chapitre un modèle représentant un cas classique d'hétérogénéité. Nous détaillons la façon dont nous avons réalisé ce modèle avec ModHel'X et montrons comment l'approche que nous proposons permet de traiter ce cas classique d'hétérogénéité différemment des approches existantes.

5.2 Implémentation de notre approche : le framework ModHel'X

5.2.1 Conception du framework

Nous avons implémenté notre approche sous la forme d'un ensemble d'outils, un « framework », permettant de réaliser les différentes activités nécessaires à l'exécution de modèles hétérogènes. Dans cette section, nous présentons les différents éléments de la conception de notre framework.

5.2.1.1 Utilisateurs du framework et cas d'utilisation

La figure 5.1 présente les différents cas d'utilisation de notre framework.

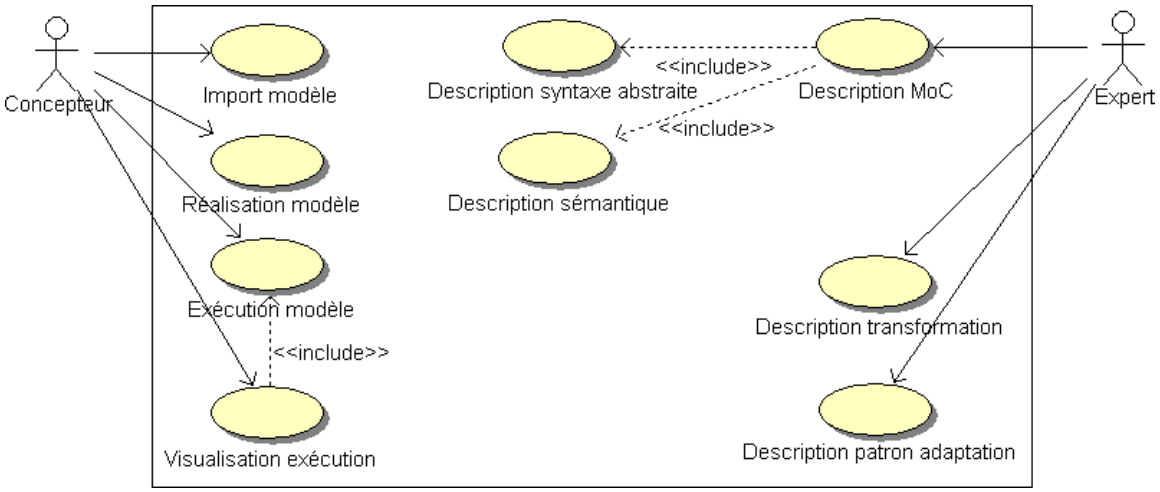


Fig. 5.1 – Cas d'utilisation du framework ModHel'X

Nous distinguons deux types d'utilisateurs de notre framework : (a) les concepteurs de systèmes, qui utilisent le framework pour construire des modèles hétérogènes pour les systèmes qu'ils conçoivent et (b) les spécialistes de langages de modélisation, qui utilisent le framework pour décrire de manière exécutable les modèles de calcul sous-jacents à des langages de modélisation et

pour définir des patrons d'adaptation sémantique entre modèles de calcul. Les concepteurs sont les utilisateurs principaux du framework, tandis que les experts sont des utilisateurs secondaires qui permettent au framework de fonctionner.

Vis-à-vis du framework, les concepteurs doivent pouvoir :

- importer des modèles homogènes ou hétérogènes existants ;
- construire des modèles homogènes ou hétérogènes ;
- exécuter des modèles homogènes ou hétérogènes ;
- visualiser les modèles avant, pendant et après l'exécution.

Par ailleurs, les experts doivent pouvoir :

- décrire la syntaxe abstraite et la sémantique d'un langage de modélisation sous la forme d'un modèle de calcul ;
- décrire les transformations permettant d'importer un modèle, c'est-à-dire de le transformer depuis son langage de modélisation originel vers la syntaxe abstraite et la sémantique du modèle de calcul correspondant ;
- décrire des patrons d'interaction entre modèles de calcul sous la forme de blocs d'interface prédéfinis.

5.2.1.2 Architecture du framework

Comme nous l'avons vu dans les chapitres précédents, notre approche s'inscrit dans le contexte MDA. Nous avons donc souhaité que notre framework s'appuie sur des standards de ce domaine. Compte tenu du fait que notre approche est centrée sur le méta-modèle générique décrit dans la section 4.4, nous avons cherché un outil permettant de décrire des méta-modèles et capable de générer automatiquement du code à partir de ceux-ci. Le standard MDA de facto permettant de réaliser cela est EMF (Eclipse Modeling Framework).

5.2.1.2.a EMF (Eclipse Modeling Framework)

EMF (Eclipse Modeling Framework) [Ecle] est un outil développé par la fondation Eclipse dans le cadre du projet Modeling, dont il est le pilier central. EMF s'appuie sur un méta-méta-modèle (niveau 3 dans la hiérarchie définie par l'OMG, voir la section 2.4.1.5.c) appelé Ecore [Eclb]. Ecore est basé sur MOF 2.0 [OMGf] et est similaire à Essential-MOF (EMOF), un sous-ensemble dérivé de MOF.

L'objectif d'EMF est de permettre la génération automatique d'un ensemble d'outils support en Java pour manipuler des modèles. A partir d'un méta-modèle, EMF génère notamment :

- un ensemble de classes d'implémentation en Java pour chacun des éléments du méta-modèle ;
- un ensemble de classes support permettant la création, la modification et la sauvegarde d'instances des éléments du méta-modèle, permettant ainsi de manipuler des modèles de manière programmatique ;
- un éditeur de modèles simple basé sur Eclipse.

Avec EMF, la génération des outils supports à partir d'un méta-modèle se fait en deux phases : (1) importation du méta-modèle dans EMF puis (2) génération de l'ensemble des classes Java nécessaires. Le méta-modèle est fourni à EMF sous la forme d'un modèle, qui peut être décrit en UML, en XML Schema ou même sous la forme d'interfaces Java annotées. Lors de l'import, EMF réalise une opération de promotion du modèle au niveau méta-modèle. Le résultat de cette opération est l'obtention d'un méta-modèle conforme à Ecore et d'un modèle de génération de code synchronisé avec le méta-modèle Ecore et comportant des informations spécifiques pour la génération Java. Le code Java est généré à partir de ce modèle de génération.

Pour chaque élément du méta-modèle d'entrée, EMF crée :

- Une interface qui contient les définitions des attributs de l'élément ainsi que les définitions des méthodes d'accès à ces attributs.

– Une classe d'implémentation squelette qui implémente l'interface. Les méthodes d'accès aux attributs sont notamment générées.

Le code Java généré est conçu pour supporter la réflexivité, la persistance (sérialisation XMI) et la notification (patron de conception Observer [GHJV95]). Ainsi, les méthodes d'accès aux attributs des classes d'implémentation contiennent du code qui gère automatiquement les notifications de modifications effectuées sur les attributs.

Un point intéressant est que, si des méthodes sont définies sur les éléments du méta-modèle importé, celles-ci sont également générées par EMF (du moins leur squelette) dans les différentes classes concernées. Il est alors possible de les compléter manuellement. De plus, EMF supporte la re-génération du code sans que le corps des méthodes qui ont été complétées manuellement soient modifiées (utilisation d'annotations). Cette fonctionnalité est très intéressante dans notre contexte puisque des méthodes d'exécution sont définies sur les éléments de notre méta-modèle et que leur contenu doit être décrit lors de la spécification d'un modèle de calcul.

5.2.1.2.b Papyrus

Nous avons choisi de décrire notre méta-modèle sous la forme d'un modèle UML. Pour cela, nous avons utilisé Papyrus [CEA], un outil de modélisation UML graphique disponible sous la forme d'un plugin sur la plate-forme Eclipse. Papyrus est compatible avec les standards UML 2 [OMGl] et DI2 (Diagram Interchange) [OMGd] de l'OMG. Il supporte également la définition et l'application de profils UML. Ainsi il nous est possible d'appliquer le profil Ecore à notre modèle. Par ailleurs, contrairement à une majorité d'outils de modélisation UML, Papyrus est open source.

5.2.1.2.c GMF (Graphical Modeling Framework)

Le projet Modeling Eclipse comporte de nombreux sous-projets correspondant à des outils venant compléter EMF. Parmi ces sous-projets, on trouve notamment des initiatives pour l'amélioration de l'interface de manipulation de modèles d'EMF. En effet, l'éditeur de modèle généré par EMF est un éditeur très basique qui présente les modèles sous forme arborescente uniquement, et l'interface de manipulation programmatique de modèles est d'une utilisation fastidieuse et verbeuse.

GMF (Graphical Modeling Framework) [Ecld] est un outil qui fournit un composant de génération de code ainsi qu'une infrastructure d'exécution pour le développement d'éditeurs graphiques basés sur EMF. Il s'appuie pour cela sur GEF (Graphical Editing Framework) [Eclc], un framework d'édition graphique basé sur le patron Model-View-Controller (MVC). GEF propose d'une part un support de rendu graphique 2D pour représenter les modèles et d'autre part un ensemble de fonctionnalités pour la création d'éditeurs de modèles. A partir d'un méta-modèle, GMF permet d'utiliser conjointement le code généré par EMF ainsi que le support graphique fournit par GEF pour créer un éditeur de modèles graphique et complet basé sur Eclipse.

Dans le contexte de notre projet, GMF est le complément idéal d'EMF pour nous permettre d'obtenir rapidement et facilement un éditeur pour les modèles ModHel'X.

5.2.1.2.d Vue globale de l'architecture du framework

L'architecture technique du framework ModHel'X est représentée sur la figure 5.2. Elle est centrée sur EMF et implémente les éléments suivants de notre approche :

– le méta-modèle générique (voir la section 4.4) avec ses opérations d'exécution (voir la section 4.5.8) ;
– les méta-modèles spécifiques pour les différents modèles de calcul (voir la section 4.6.1) ;
– les sémantiques d'exécution des modèles de calcul (voir la section 4.6.2).

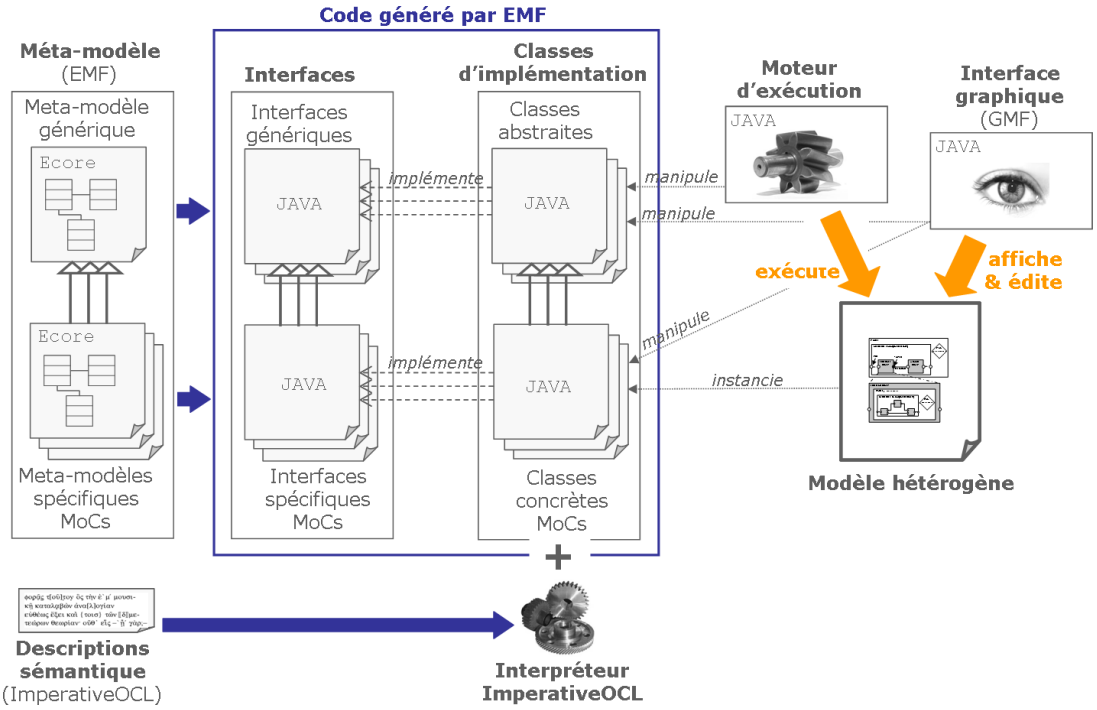


FIG. 5.2 – Architecture technique du framework ModHel'X

Les différents éléments de cette architecture sont les suivants :

Méta-modèle : Notre méta-modèle est importé dans EMF selon le processus décrit plus haut. Il est divisé en deux paquetages : le paquetage contenant les éléments génériques et le paquetage destiné à recevoir les méta-modèles spécifiques des différents modèles de calcul, eux-mêmes organisés en différents paquetages.

Code généré par EMF : Le code généré par EMF est constitué d'interfaces et de classes d'implémentation. Comme le méta-modèle, il est divisé en deux paquetages : un pour le code correspondant au méta-modèle générique et l'autre pour les différents paquetages correspondant aux différents méta-modèles spécifiques.

Interpréteur ImperativeOCL : Le corps des opérations d'exécution des modèles de calcul est décrit en ImperativeOCL et est fourni en entrée d'un interpréteur qui est appelé à la volée lors de l'exécution. Cette architecture permet d'éviter la traduction intermédiaire des descriptions en ImperativeOCL.

Moteur d'exécution : Il s'agit d'un singleton chargé de dérouler la boucle de déclenchement des snapshots.

Interface graphique : L'interface graphique est réalisée avec le framework GMF et s'appuie sur le code généré par EMF.

Remarquons que la fonctionnalité d'import de modèle n'apparaît pas dans cette architecture. Elle peut en fait être réalisée avec tout outil de transformation de modèles capable de prendre en entrée notre méta-modèle soit dans son format UML soit dans son format Ecore (éventuellement sérialisé en XMI). Décrire une transformation de modèles implique bien sûr de disposer également du modèle à transformer et de son méta-modèle source [1] dans des formats acceptables par l'outil en question. Le modèle résultant d'une transformation est généralement obtenu en XMI pour la plupart des outils. Par exemple, ATL (Atlas Transformation Language) [JAB+06], qui fait

1. Voir la section 3.3.3 du chapitre 3 pour plus de détails sur le principe des transformations de modèles.

également partie du projet Modeling d'Eclipse, supporte XMI et Ecore en format d'entrée des modèles et des méta-modèles et génère le modèle résultant de la transformation en XMI. Comme EMF génère automatiquement le code de chargement et de sérialisation en XMI des modèles, notre framework peut prendre en entrée un modèle décrit en XMI. En conséquence, nous avons volontairement laissé de côté la fonctionnalité d'import car elle peut être réalisée par un autre outil de manière complètement indépendante de notre framework.

5.2.2 Réalisation du framework et difficultés rencontrées

Lors de l'implémentation effective de l'architecture décrite dans la section précédente, nous avons rencontré différentes difficultés. Celles-ci nous ont amené à modifier notre architecture de manière assez profonde. Nous détaillons ici les causes et conséquences de ces difficultés.

5.2.2.1 Description de la sémantique des modèles de calcul

Comme nous l'avons vu dans le chapitre précédent, le choix d'un sous-ensemble du langage ImperativeOCL pour la description de la sémantique des modèles de calcul résulte de différents besoins (voir la section 4.6.2.2). Ce choix a cependant plusieurs inconvénients, comme nous l'avons évoqué dans la section 4.7.5. Notamment, en ce qui concerne l'implémentation du framework, le fait d'avoir choisi un sous-ensemble de ce langage nécessite de pouvoir adapter l'interpréteur choisi pour le framework.

Dans une étude préliminaire des interpréteurs disponibles, nous avons constaté qu'il existait peu d'implémentations d'ImperativeOCL, et aucune qui soit directement adaptable à nos besoins. Devant l'effort nécessaire pour modifier et intégrer un tel outil à notre framework et pour pouvoir commencer au plus tôt à expérimenter l'implémentation de modèles de calcul avec notre approche, nous avons adopté la solution temporaire suivante : le code ImperativeOCL de chacune des méthodes d'exécution des modèles de calcul est traduit manuellement en Java et copié dans les méthodes d'exécution des classes Java générées par EMF pour les modèles de calcul. Afin d'obtenir avec Java des expressions relativement similaires au code ImperativeOCL, nous avons défini différentes fonctions auxiliaires telles que `select`, `collect`, etc.

Une étude complète des outils répondant le mieux à nos besoins ainsi qu'une première expérimentation sur l'adaptation d'un interpréteur OCL au framework ModHel'X a été réalisée par la suite et est présentée dans [Cip08]. Dans cette étude, l'interpréteur OCL OSLO [HRW] a été choisi comme base d'implémentation et modifié afin de supporter des expressions à effets de bord. Différentes difficultés ont été rencontrées en ce qui concerne l'intégration au framework ModHel'X. Ces difficultés concernent notamment (1) la transmission du modèle lors de l'appel de l'interpréteur à la volée lors de l'exécution, (2) la transmission du code ImperativeOCL des modèles de calcul et (3) la détermination du contexte (au sens OCL du terme) dans lequel l'interpréteur doit évaluer le code ImperativeOCL qui lui est fourni. En conséquence, l'intégration de cet interpréteur à ModHel'X demandera encore différents travaux d'expérimentation.

5.2.2.2 Spécialisation du méta-modèle générique de ModHel'X

Les autres difficultés que nous avons rencontrées lors de l'implémentation de notre framework sont liées à l'étape de spécialisation du méta-modèle générique de ModHel'X pour créer un méta-modèle spécifique pour un modèle de calcul. Du fait de cette particularité de notre approche, l'utilisation d'EMF s'est avérée très complexe. Nous détaillons dans les paragraphes suivants les différents problèmes rencontrés ainsi que les solutions que nous avons envisagées.

5.2.2.2.a Mécanisme d'import du méta-modèle d'EMF

Le mécanisme d'import de méta-modèle d'EMF a tout d'abord compliqué la gestion du développement de notre framework, que nous souhaitions réaliser avec un système de gestion de

versions pour Eclipse (Subversion [Tri]).

Comme nous l'avons vu dans le paragraphe de présentation d'EMF, EMF permet d'obtenir automatiquement une version Ecore d'un méta-modèle à partir d'une version UML de celui-ci, plus facile à manipuler car graphique. Pour cela, il est nécessaire de créer un nouveau projet EMF dans Eclipse et d'importer un fichier UML à partir duquel EMF génère un fichier Ecore. Puis le code structurel correspondant peut être obtenu automatiquement à partir du méta-modèle Ecore (en fait à partir du modèle de génération mais celui-ci est synchronisé automatiquement avec le méta-modèle Ecore). Enfin, le code structurel peut être complété avec le code d'exécution nécessaire (corps des méthodes). Un marquage peut être utilisé afin de protéger le code modifié d'une éventuelle re-génération du code après modification du méta-modèle Ecore. Cependant, ce processus présente deux manques :

1. des modifications effectuées au niveau du code ne sont pas répercutées sur le méta-modèle Ecore (pas de « round-trip engineering ») ;
2. aucun processus de mise à jour du méta-modèle Ecore (ni du code) n'est prévu lorsque des modifications sont apportées au méta-modèle dans sa version UML : il est nécessaire dans ce cas de créer un nouveau projet EMF en important le méta-modèle modifié.

Le deuxième point présenté ici implique que le projet EMF en question peut difficilement être géré par un serveur de versions, qui attache des propriétés au niveau du projet Eclipse.

Dans notre contexte, l'ajout d'un modèle de calcul à ModHel'X implique de créer un méta-modèle spécifique en spécialisant les classes de notre méta-modèle générique. Cette opération est réalisée au niveau du méta-modèle dans sa version UML. Afin de répercuter les modifications sur le code généré, il est alors malheureusement nécessaire de supprimer le projet EMF existant et d'en recréer un nouveau afin d'obtenir la version Ecore du méta-modèle ainsi modifié et pouvoir générer le code correspondant. Ce processus est incompatible avec une gestion sous Subversion du code de notre framework.

Une solution envisageable pour résoudre ce problème consiste à créer des classes d'implémentation indépendantes de celles générées par EMF, gérées dans un projet Eclipse à part sauvegardé sous Subversion, et implémentant les interfaces générées par EMF. Ces classes d'implémentation sont destinées à recevoir le code d'exécution des modèles de calcul. En décomposant le code ainsi, il est possible de re-créer autant de fois que nécessaire le projet EMF sans impacter le projet contenant le code modifié. Nous avons mis en place cette architecture dans une des premières versions de notre framework. Malheureusement, nous avons rencontré d'autres difficultés par la suite qui nous ont amené à modifier plus avant notre architecture.

5.2.2.2.b Héritage multiple

Lors de la création d'un méta-modèle spécifique pour un modèle de calcul, la spécialisation de certains éléments de notre méta-modèle générique crée une situation d'héritage multiple.

Pour illustrer ce cas, reprenons le méta-modèle spécifique créé pour le modèle de calcul FSM (voir la section 4.6.1). Nous avons vu qu'il était nécessaire de spécialiser la notion de bloc dans ce méta-modèle afin que le modèle de calcul puisse faire la distinction entre gardes et actions. Or la notion de bloc est déjà spécialisée dans notre méta-modèle générique par les notions de bloc atomique, bloc composite et bloc d'interface (voir la section 4.4 du chapitre précédent). Dans ce contexte, créer un bloc atomique qui représente une garde pour le modèle de calcul FSM, par exemple, implique que ce bloc hérite à la fois de `FSMGuard` et de `AtomicBlock`. C'est cette situation que nous avons représentée sur la figure 5.3. Sur cet exemple, `FSMSimpleGuard` représente une garde qui devient vraie lorsque son événement déclencheur est présent sur l'une de ses entrées. `FSMSimpleAction` représente une action qui produit un événement lorsqu'elle est déclenchée (c'est-à-dire lorsque sa garde associée devient vraie).

Cette situation pose problème dans notre contexte car l'héritage multiple n'est pas supporté par Java. EMF permet de contourner ce problème en tirant parti du support de l'héritage

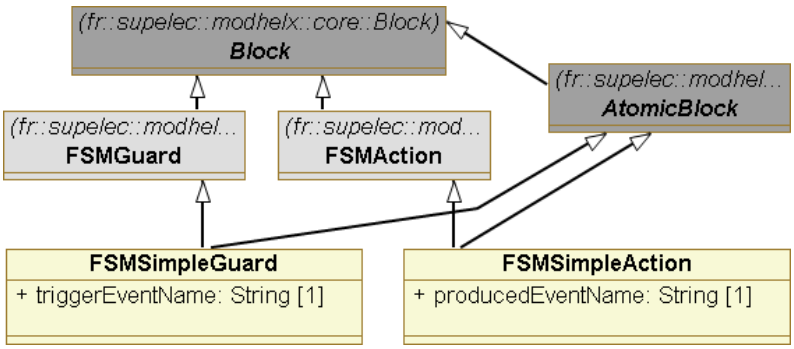


FIG. 5.3 – Méta-modèle spécifique pour FSM et héritage multiple

multiple sur les interfaces (une interface étant générée pour chacun des éléments du méta-modèle). En ce qui concerne les classes d'implémentation, cependant, il est nécessaire d'indiquer à EMF quelle classe de base la classe d'implémentation doit étendre (en effet, chaque classe d'implémentation ne peut étendre qu'une classe d'implémentation de base). Il faut pour cela ajouter le stéréotype ≪ `extend` ≫ sur la relation d'héritage qui doit être considérée comme principale.

5.2.2.2.c Collections paramétrées et covariance

Lors de la création d'un méta-modèle spécifique pour un modèle de calcul, la spécialisation de certains éléments de notre méta-modèle générique peut également créer des problèmes de covariance sur le code généré par EMF.

Afin d'illustrer ce problème, reprenons de nouveau en exemple le méta-modèle spécifique créé pour le modèle de calcul FSM. Dans la section 4.6.1 du chapitre précédent, nous n'avons spécialisé qu'une partie des éléments de notre méta-modèle générique. Cependant, pour certains modèles de calcul, il peut être nécessaire de spécialiser tous les éléments du méta-modèle. Si, pour FSM, l'on décide de spécialiser tous les éléments du méta-modèle générique, on en vient à spécialiser, par exemple, l'élément `Pin` que nous n'avions pas spécialisé dans la section 4.6.1. Nous avions déjà par ailleurs spécialisé l'élément `Token` en `FSMEvent` afin de modéliser la notion d'événement. Considérons une partie du méta-modèle spécifique ainsi obtenu, représentée sur la figure 5.4. Dans ce cas de figure, EMF génère pour l'élément `Pin` une interface et une classe d'implémentation qui comportent une méthode `getStoredTokens` retournant une collection paramétrée `EList<Token>` contenant les jetons stockés sur le point d'interface. La classe `FSMPin` hérite de cette méthode. Cependant, ce sont des `FSMEvent` qui sont stockés sur ce type de point d'interface, et non simplement des `Token`. On souhaiterait alors surcharger la méthode `getStoredTokens` et modifier son type de retour afin qu'elle retourne une collection `EList<FSMEvent>`. Malheureusement, la covariance des types de retour ne peut s'appliquer dans ce cas précis car `EList<FSMEvent>` n'est pas considéré comme un sous type de `EList<Token>`

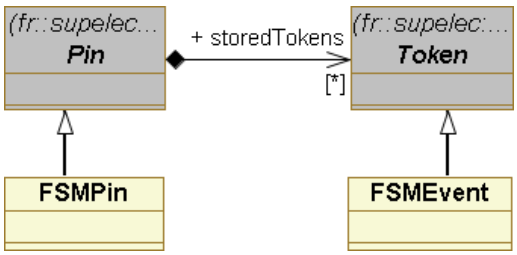


FIG. 5.4 – Spécialisation des éléments `Pin` et `Token` du méta-modèle pour FSM

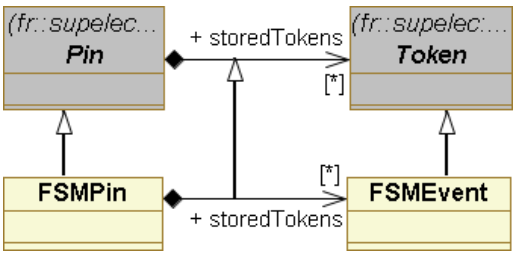


FIG. 5.5 – Spécialisation de l'association entre `Pin` et `Token` pour FSM

en Java (problème des types paramétrés). Il faut en fait ici que le type de retour de la méthode `getStoredTokens` soit `EList<? extends Token>` pour obtenir la covariance et surcharger la méthode. Malheureusement, il n'existe à notre connaissance aucun moyen de forcer EMF à générer un tel code pour le moment [2].

Les différentes solutions que nous avons envisagées afin de résoudre ce problème sont les suivantes :

Modification manuelle du code généré. Cette solution s'avère complexe car (1) plusieurs éléments du méta-modèle sont en fait concernés par ce problème (les contraintes de temps par exemple), et (2) une telle modification se répercute sur de multiples parties du code généré par EMF (méthodes `eGet`, `eSet`, etc.). De plus, les modifications seraient à effectuer sur le code généré à chaque fois qu'un nouveau modèle de calcul est décrit dans le framework.

Ajout de relations de généralisation entre associations. La figure 5.5 représente l'application de cette solution dans le cadre de l'exemple que nous avons présenté ci-dessus. Malheuresement, si ce type de relations est supporté par UML 2.0 [RJB04, p. 425] et par PapyrusUML, celles-ci n'existent pas dans Ecore et ne sont donc pas prises en compte par EMF lors de la génération du code.

Utilisation de contraintes. Deux types de contraintes peuvent être utilisées dans le cas présenté ici. D'une part, UML définit le mot clé `subsets` permettant de contraindre une relation de manière similaire à la relation de généralisation de la figure 5.5. D'autre part, l'utilisation d'une contrainte OCL telle que celle présentée sur le listing 5.1 permet de contraindre le type des éléments de l'ensemble `storedTokens`. Malheureusement, aucune de ces deux techniques n'est prise en compte par EMF, du moins dans la version que nous avons utilisée.

Listing 5.1: Contrainte OCL complétant le méta-modèle spécifique de FSM

```
{context FSMPin inv: self.storedTokens->forAll(t|t.oclIsTypeOf(FSMEvent))}
```

Ce problème nous a fait perdre beaucoup de temps dans la réalisation de notre framework et a eu des conséquences lourdes sur notre implémentation. En effet, n'ayant pas réussi à trouver de solution satisfaisante, nous avons décidé d'abandonner provisoirement EMF afin de pouvoir quand même tester l'application de notre approche sur différents modèles de calcul en utilisant une version simplifiée du framework.

La spécialisation des éléments de notre méta-modèle générique pour un modèle de calcul soulève par ailleurs une question de conception globale dans le cadre de notre approche : faut-il spécialiser tous les éléments du méta-modèle ou uniquement ceux estimés nécessaires pour le modèle de calcul considéré ? Nous n'avons pas pu vraiment répondre à cette question dans le cadre des travaux présentés dans ce mémoire. Au vu des difficultés que la spécialisation de tous les éléments du méta-modèle engendre au niveau de l'implémentation dans notre framework, nous avons choisi pour le moment de ne spécialiser que les éléments strictement nécessaires afin de réduire l'impact du problème. L'expérimentation des deux solutions sur un grand nombre de modèles de calcul sera nécessaire afin de mettre au point des règles de « bonne » description des modèles de calcul dans notre approche.

5.2.3 Solution finale retenue

Notre décision de renoncer provisoirement à l'utilisation d'EMF a impliqué différentes modifications sur l'architecture de notre framework. L'architecture finale implémentée est représentée sur la figure 5.6.

2. Nous avons utilisé les versions 3.3.2 d'Eclipse (Europa) et 2.3.2 d'EMF.

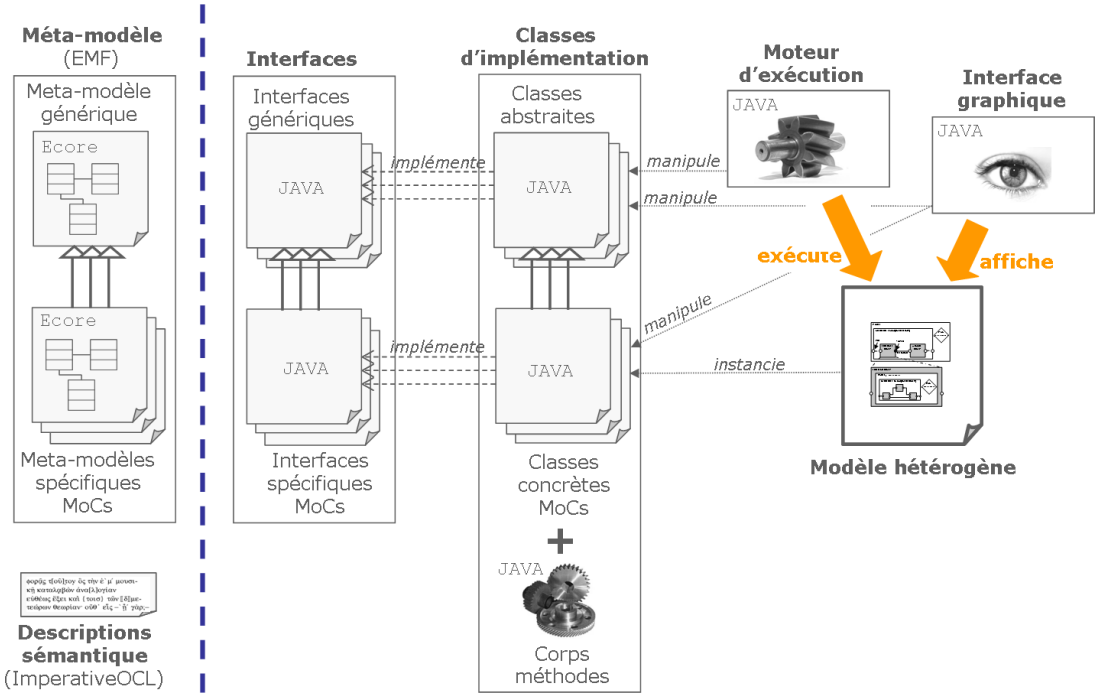


FIG. 5.6 – Architecture technique finale du framework ModHel'X

Afin de faciliter le portage du code pour une future version du framework qui s'appuierait sur EMF, nous avons conçu le code de notre framework de manière similaire au code obtenu par génération avec EMF (séparation des interfaces et des classes d'implémentation notamment). Les principales différences concernent :

- Les collections : nous utilisons des types standards de collections, tels que `ArrayList` par exemple, et non pas les types définis par EMF (`EList`, etc.). Le passage à EMF demandera une adaptation de ces types.
- Les notifications : nous n'avons pas implémenté de système de notification (patron de conception Observer) afin de ne pas rendre le code trop complexe et sachant que le système de notification pourra, à terme, être complètement généré par EMF.

Les principaux éléments de l'architecture résultant de nos modifications sont les suivants :

Méta-modèle : Notre méta-modèle est toujours importé dans EMF et organisé ainsi que nous l'avons décrit dans la section 5.2.1.2.

Code Java : Le code Java est créé manuellement et synchronisé manuellement avec le méta-modèle. Il est cependant toujours organisé ainsi que nous l'avons décrit dans la section 5.2.1.2.

Descriptions ImperativeOCL : Les descriptions en ImperativeOCL sont traduites manuellement en Java et ajoutées directement dans les classes d'implémentation. Nous avons utilisé des fonctions auxiliaires afin de rendre le code plus concis et plus ressemblant au code ImperativeOCL.

Moteur d'exécution : Il s'agit d'un singleton chargé de dérouler la boucle de déclenchement des snapshots.

Interface graphique : L'interface graphique est une interface élémentaire, réalisée avec JGraph [3]. Elle ne permet que la visualisation des modèles, ceux-ci devant être créés de manière pro-

3. JGraph [JGr] est une bibliothèque Java open source, basée sur Swing et spécialement conçue pour permettre la représentation graphique de graphes.

grammatique. Elle a été réalisée de manière complètement indépendante des classes correspondant aux éléments du méta-modèle de manière à pouvoir être remplacée dès que possible par une interface complète créée avec GMF.

Le code de notre framework, correspondant à l'architecture décrite dans cette section, est disponible en ligne [Har].

Soulignons ici que, dans cette version modifiée, le framework ModHel'X ne constitue qu'un outil expérimental, loin de l'outil complet que nous souhaitions réaliser. Il n'est donc pas du tout adapté aux utilisateurs cibles présentés dans la section 5.2.1.1.

5.3 Cas d'étude : le modèle de la machine à café

Afin d'illustrer l'utilisation de notre framework, et plus généralement de notre approche, nous présentons dans cette section un exemple de modèle hétérogène réalisé avec ModHel'X. L'exemple que nous avons choisi est un cas classique d'hétérogénéité impliquant conjointement deux modèles de calcul avec des notions de temps différents : l'un met en œuvre le temps discret et l'autre met en œuvre uniquement la notion d'événement (non daté). Nous montrons dans cette section comment il est possible de traiter ce cas avec ModHel'X et comparons avec la façon dont celui-ci peut être traité avec Ptolemy II.

5.3.1 Description générale de l'exemple et motivation

Nous considérons un système composé d'une machine à café et d'un utilisateur. Nous nous plaçons dans le contexte où un modèle préliminaire du comportement de la machine à café ainsi qu'un modèle représentant le comportement d'un utilisateur type ont été réalisés. Notre objectif est de vérifier par simulation que le comportement de la machine ainsi modélisé est non seulement cohérent mais également qu'il est compatible avec celui de l'utilisateur. Pour cela, nous souhaitons utiliser le modèle de la machine à café et le modèle de l'utilisateur dans un modèle global les mettant en interaction. Le comportement attendu du système formé de la machine et de l'utilisateur est le suivant : tout d'abord l'utilisateur introduit une pièce dans la machine, puis il presse le bouton « café » et enfin il obtient son café après un délai correspondant au temps de préparation du café par la machine.

La machine à café est modélisée par un automate à états finis (car, à cette étape du processus de conception, nous nous focalisons sur la logique de son comportement). Ce modèle est basé sur le modèle de calcul Finite State Machine (FSM) que nous avons introduit dans la section 4.6.1.2 du précédent chapitre. Le comportement de l'utilisateur est modélisé par un programme Java. Pour le modèle global, nous utiliserons le modèle de calcul Discrete Events (DE) afin de pouvoir modéliser la date à laquelle intervient chaque interaction entre l'utilisateur et la machine : (1) introduction de la pièce, (2) appui sur le bouton « café » et (3) distribution du café.

Pour servir notre objectif, nous souhaitons donc utiliser le modèle basé sur FSM de la machine à café et le programme Java représentant l'utilisateur dans le modèle global du système basé sur DE. Le résultat de la combinaison hiérarchique du modèle du système et du modèle de la machine à café est donc hétérogène. Une telle combinaison est un exemple classique d'hétérogénéité qui est très bien traité par des outils tels que Ptolemy II. Cependant, nous allons voir comment il est possible de gérer les relations entre les modèles de calcul DE et FSM différemment dans ModHelX. Dans les sections suivantes, nous détaillons tout d'abord l'implémentation que nous avons réalisée de ces deux modèles de calcul dans ModHel'X puis, dans la section 5.3.4, nous détaillons comment l'adaptation sémantique peut être implémentée et personnalisée afin de mieux correspondre à nos besoins dans cet exemple.

5.3.2 Modèle du système global : MoC Discrete Events (DE)

Le modèle de calcul Discrete Events (DE) (voir l'annexe A.2 pour une description succincte de ce modèle de calcul) est un modèle de calcul basé sur le temps discret. Ainsi, dans DE, le temps est représenté par une ligne temporelle construite sur une séquence d'événements composés d'une valeur et d'une date.

Un modèle DE est construit à partir de composants qui représentent des processus consommant et produisant des événements. Ces composants sont reliés par des connexions. L'existence d'une connexion entre deux composants signifie qu'ils peuvent recevoir des événements l'un de l'autre (chaque composant « s'abonne » aux événements produits par l'autre). Chaque connexion représente une suite d'événements placés sur la ligne de temps, c'est-à-dire un signal. Lorsqu'un événement est produit, une date d'occurrence lui est attribuée par le composant producteur et l'événement ne sera présenté au composant destinataire que lorsque cette date sera atteinte. La propagation des événements sur une connexion est considérée comme instantanée. A l'inverse, un délai peut exister entre le moment où un composant reçoit un événement et le moment où il réagit à cet événement en produisant lui même un événement.

Les modèles DE sont utilisés généralement pour modéliser des réseaux de communication ou du matériel électronique. Des langages tels que VHDL ou Verilog, ainsi que l'outil SimEvents de The MathWorks, implémentent un modèle de calcul de type DE.

5.3.2.1 Méta-modèle spécifique pour DE

Comme DE est un modèle de calcul impliquant un modèle de temps discret, il est nécessaire de définir la façon dont est représenté le temps (voir la section 4.5.10 pour plus de détail sur la façon dont le temps est modélisé dans notre approche). Nous choisissons de représenter les dates par des entiers. Une date entière est donc attribuée à chaque snapshot. Les contraintes de temps sont composées d'une date et d'un composant auteur, les composants étant représentés par des blocs. Enfin, les événements sont représentés par des jetons assortis d'une valeur (entière également pour simplifier), d'une date et d'un point d'interface de destination.

La figure 5.7 montre le méta-modèle spécifique ainsi obtenu pour DE. Les différentes variables nécessaires pour les éléments `DEModel` et `DEMoC` sont discutées dans la section suivante.

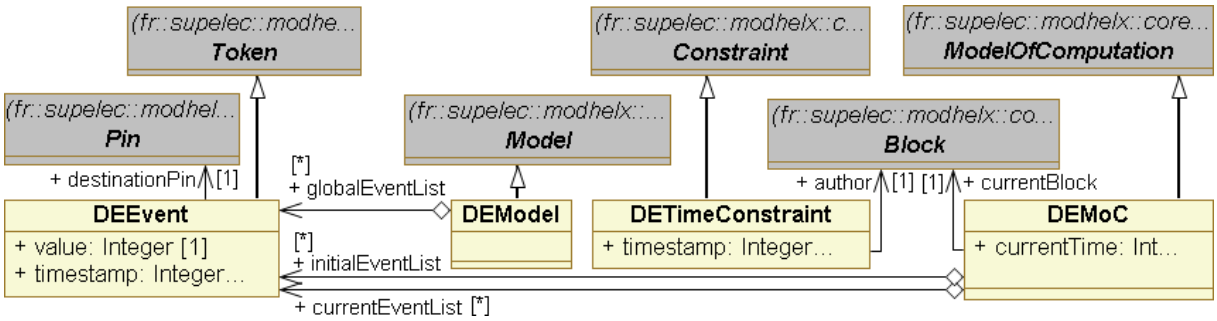


Fig. 5.7 – Méta-modèle spécifique pour DE

5.3.2.2 Sémantique d'exécution spécifique pour DE

La sémantique que nous avons choisie pour l'exécution des modèles DE est similaire à la sémantique implémentée par Ptolemy II.

5.3.2.2.a Principes d'exécution

En ce qui concerne la gestion du temps, nous utilisons une file d'événements globale ainsi qu'une notion de temps courant qui permet de s'assurer que les événements sont traités dans

l'ordre chronologique. Les événements qui sont produits par un bloc sur l'un de ses points d'interface de sortie sont reçus sur tous les points d'interface d'entrée connectés à celui-ci via des relations. La date associée à l'événement est déterminée par le bloc qui l'a produit. La réaction des blocs est considérée comme instantanée, le délai de fonctionnement propre à chaque bloc étant pris en compte dans la date associée aux événements produits. Dans la file des événements globale, les événements sont triés suivant leur date et le point d'interface (donc le bloc) auquel ils sont destinés. Un événement est retiré de la file des événements et déposé sur le point d'interface auquel il est destiné lorsque sa date est égale à la date courante. Alors, le bloc destinataire est mis à jour et peut traiter l'événement reçu. Les événements qu'il peut alors produire sont placés dans la file.

Lorsqu'un snapshot est pris, la date courante est déterminée en fonction des dates des événements présents dans la file et des contraintes de temps produites par les blocs (soit lors du `startOfSnapshot`, soit lors du `endOfSnapshot` du snapshot précédent). A chaque pas de calcul (c'est-à-dire à chaque tour de boucle dans le calcul du snapshot), nous considérons les blocs auteurs de contraintes de temps pour la date courante ainsi que les blocs destinataires d'événements à la date courante. Nous mettons à jour l'élément minimal parmi tous ces blocs selon l'ordre topologique de manière à respecter les dépendances causales entre blocs. Les événements qui sont délivrés à leur bloc destinataire sont retirés de la file. Un snapshot est déterminé seulement lorsque tous les événements à la date courante ont été traités. Les événements dont la date est dans le futur par rapport à la date courante restent dans la file pour être traités lors des snapshots suivants.

Dans DE, l'état du modèle est représenté par la file globale des événements qui doivent être traités au cours de l'exécution : `globalEventList` (voir la figure 5.7).

5.3.2.2.b Difficultés spécifiques

Les difficultés classiques dans l'exécution de modèles DE sont (1) la gestion des événements simultanés, (2) la gestion des composants qui sont des sources d'événements c'est-à-dire qui produisent des événements de façon proactive, et (3) le traitement des boucles instantanées.

Pour résoudre la difficulté (1) et assurer un comportement déterministe, nous traitons les événements simultanés séquentiellement dans l'ordre topologique de leurs destinataires.

En ce qui concerne la difficulté (2), elle est résolue par l'utilisation des contraintes temporelles (voir la section 4.5.10). Les sources d'événements doivent produire des contraintes de façon à déclencher leur mise à jour à une date donnée. Ces contraintes peuvent être produites soit lors du `setup` afin de déclencher le premier snapshot, soit lors des `startOfSnapshot` et `endOfSnapshot` si les contraintes dépendent d'événements qui se produisent au cours de l'exécution.

Les boucles instantanées sources de la difficulté (3) sont soit des relations directes entre les sorties et les entrées d'un même bloc, soit des boucles qui comprennent un ou plusieurs blocs produisant des événements dont la date est identique à celle des événements qu'ils ont reçus. Le premier cas pose problème pour le tri topologique mais ce type de boucle est facilement détectable lors d'une vérification statique avant l'exécution. Le deuxième cas ne peut pas être détecté statiquement mais ne pose pas de problème à l'exécution, même si un comportement non désiré du modèle peut en résulter. En conséquence, nous laissons l'utilisateur responsable de l'usage de telles boucles.

5.3.2.2.c Principe du calcul d'un snapshot

Les actions à réaliser au cours du calcul d'un snapshot avec DE sont les suivantes :

– Au début du snapshot, il faut prendre en compte les événements présents sur les entrées du modèle et les déposer dans la file des événements du modèle.

- Il faut ensuite déterminer la date courante à partir des événements présents dans la file des événements du modèle et de l'ensemble des contraintes de temps qui ont été émises. Cette date courante permet de sélectionner les événements à traiter initialement pendant le snapshot. D'autres événements pourront éventuellement s'ajouter à cette file si des blocs en produisent pendant le snapshot.
- A chaque tour de boucle du calcul du snapshot, il faut choisir un bloc à mettre à jour (c'est-à-dire un bloc sur lequel appeler l'opération `update`). Ce choix dépend des contraintes émises pour la date courante ainsi que des événements à traiter à la date courante. Ce choix doit respecter l'ordre topologique (et donc les dépendances causales entre blocs).
- Une fois le bloc choisi, il faut déposer sur ses entrées les événements qui lui sont destinés.
- Après la mise à jour du bloc, il faut déplacer les événements qu'il a produit dans la file.
- Le snapshot est déterminé lorsque tous les événements et toutes les contraintes dont la date est égale à la date courante ont été traités. Ne doivent rester dans la file des événements à traiter que les événements dont la date est dans le futur par rapport à la date du snapshot courant.
- A la fin du snapshot, la file des événements du modèle doit être mise à jour : il faut supprimer les événements traités et ajouter les événements produits avec une date future.

5.3.2.2.d Variables de calcul

Dans ce contexte, le modèle de calcul `DEMoC` a besoin, pour réaliser le calcul d'un snapshot, des variables suivantes (représentées sur la figure 5.7) :

- `initialEventList` : variable qui contient les événements à traiter initialement pendant le snapshot. Cette variable n'est pas modifiée pendant le snapshot, elle permet de préserver l'état du modèle en cas de non validation du snapshot et de mettre à jour la file des événements du modèle à la fin du snapshot (elle contient les événements à supprimer de la file).
- `currentEventList` : variable qui, au début du snapshot, contient les événements à traiter pendant le snapshot et qui, à la fin du snapshot, contient les événements produits au cours du snapshot et dont la date est dans le futur. Elle permet de mettre à jour la file des événements du modèle à la fin du snapshot (elle contient les événements à ajouter à la file).
- `currentBlock` : variable qui contient le bloc à mettre à jour dans le tour de boucle courant.

5.3.2.2.e Opérations d'exécution

Nous présentons ci-dessous le rôle respectif des différentes opérations du modèle de calcul DE ainsi que quelques-uns de leurs algorithmes respectifs en ImperativeOCL. Le code complet du modèle de calcul DE est présenté en Java dans l'annexe C.2.

`startOfSnapshot` : C'est l'opération `startOfSnapshot` qui a pour rôle de prendre en compte les événements présents sur les entrées du modèle afin de les inclure à la file des événements.

`reset` : La détermination de la date courante et de l'ensemble initial des événements à traiter à la date courante est réalisée par l'opération `reset` afin que l'état du modèle soit préservé en cas de non validation du snapshot.

`initSchedule` : L'opération `initSchedule` a ensuite pour rôle de déterminer le bloc à mettre à jour dans le tour de boucle courant. Pour cela, elle prend en compte les événements à traiter à la date courante ainsi que les contraintes émise par les blocs pour cette date. Le code de cette opération est présenté en ImperativeOCL sur le listing 5.2.

`prePropagate` et `postPropagate` : Le dépôt des événements à traiter de la file sur le bloc courant puis le déplacement des événements produits par ce bloc dans la file des événements

sont réalisés respectivement par les opérations `prePropagate` et `postPropagate`. L'opération `postPropagate` présente une particularité : lorsque les points d'interface du bloc courant sont des alias des sorties du modèle, les événements produits ne sont pas déplacés dans la file des événements mais restent sur les points d'interface concernés. En effet, ces événements ne sont pas destinés à des blocs mais à l'environnement du modèle. Le code de l'opération `postPropagate` est présenté en ImperativeOCL sur le listing 5.3. Afin d'alléger le code, nous faisons appel à une fonction auxiliaire `moveEventsToList(Pin p, Set<DEEvent> list)`. Son rôle est le suivant : pour chaque événement produit sur `p` et pour chaque relation partant de `p`, cette fonction réalise une copie de l'événement avec pour destinataire le bloc cible de la relation puis dépose l'événement dans `list`. Le code complet de cette fonction peut être trouvé dans l'annexe C.2.

`snapDet` : L'opération `snapDet` est chargée de vérifier si toutes les contraintes et tous les événements à la date courante ont été traités.

`validate` : La validation du snapshot, réalisée par l'opération `validate` est déléguée à l'ensemble des blocs constituant le snapshot.

`endOfSnapshot` : Enfin, la mise à jour de la file des événements du modèle est réalisée dans le `endOfSnapshot`.

Les autres opérations, c'est-à-dire `preSchedule` et `interSchedule` ne jouent aucun rôle pour ce modèle de calcul.

Listing 5.2: DEMoC::initSchedule(m:Model)

```
// Search for the blocks that have to receive events at the current time
OrderedSet<Block> blocklist := self.currentEventList→collect(e:Event|e.destinationPin.isInputForBlock);

// Add the blocks having produced a constraint at the current time
blocklist := blocklist→union(
  self.constraints→select(c:Constraint|c.timestamp=self.currentTime)
  →collect(c:Constraint|c.author));

// If blocks have to receive events or have produced constraints at the current time...
if(blocklist→notEmpty()){
        self.topologicalSort(blocklist, m.structure); // Sort the list by topological order
        self.currentBlock := blocklist→first(); // The block to update is the first block of the list

        // If it is a block that has produced a constraint,
        if(self.constraints→exists(c:Constraint|c.author=self.currentBlock and c.timestamp=self.currentTime))
                // Then remove (one of) the corresponding constraint(s)
                self.constraints := self.constraints→excluding(
                  self.constraints→select(c:Constraint|c.author=self.currentBlock and c.timestamp=self.currentTime)
                  →first());
}
```

Listing 5.3: DEMoC::postPropagate(m:Model)

```
self.currentBlock.pinsOut→forEach(p){ // Check the outputs of the updated block
        if(p.storedTokens→notEmpty()){ // If it has produced tokens
                // And if the considered output is not aliased by an output of the model
                if(p.aliasedBy→empty() || (p.aliasedBy→notEmpty() && p.aliasedBy.isOutputForBlock!=m.structure)){
                        // Then move the produced events to the current event list
                        self.moveEventsToList(p,self.currentEventList);
                }
        }
}
```

5.3.2.2.f Opérations de mise à jour spécifiques aux modèles internes

Dans le paragraphe précédent nous n'avons pas discuté des opérations `startOfUpdate`, `endOfUpdate` et `further`, qui sont respectivement les pendants de `startOfSnapshot`, `endOfSnapshot` et `snapDet` pour les modèles internes des blocs d'interface (voir la section 4.5.9.3 du chapitre précédent).

Or, nous avons vu que, dans DE, les événements présents sur les points d'interface d'entrée d'un modèle au début d'un snapshot sont « acquis » et placés dans la file globale des événements à traiter (cela est réalisé par l'opération `StartOfSnapshot`). Lorsque le modèle de calcul DE est utilisé dans un modèle interne, c'est-à-dire dans un modèle placé à l'intérieur d'un bloc d'interface, les événements présents sur ses points d'interface d'entrée doivent également être ajoutés à la file des événements à traiter à chaque fois que le modèle est *mis à jour*, et pas seulement au début d'un snapshot. C'est le rôle de l'opération `startOfUpdate`. Celle-ci doit donc être décrite pour le modèle de calcul DE. Le listing 5.4 présente le code de cette opération en ImperativeOCL.

Listing 5.4: DEMoC::startOfUpdate(m:Model)

```
m.structure.pinsIn→forEach(p){ // Check the inputs of the model
        if(p.storedTokens→notEmpty()){ // If it has received tokens
                // Then move the produced events to the current event list
                self.moveEventsToList(p,self.self.currentEventList)
        }
}
```

L'opération `endOfUpdate` n'a pas de rôle particulier pour DE puisque les événements produits par les blocs dont les points d'interface sont aliasés par des sorties du modèle ne sont pas déplacés dans la file des événements par l'opération `postPropagate` et sont donc vus sur les sorties du modèle à la fin de la mise à jour de celui-ci.

Enfin, le modèle de calcul DE étant strict (voir la section 4.5.9.3), la mise à jour du modèle s'arrête lorsque tous les événements à la date courante ont été traités et lorsque toutes les contraintes à la date courante ont été traités. En conséquence, la méthode `further` est équivalente à l'opposé de la méthode `snapDet`.

5.3.2.3 Modèle du système dans ModHel'X

La figure 5.8 montre le modèle du système impliquant le modèle de calcul DE tel qu'il est représenté dans le framework ModHel'X.

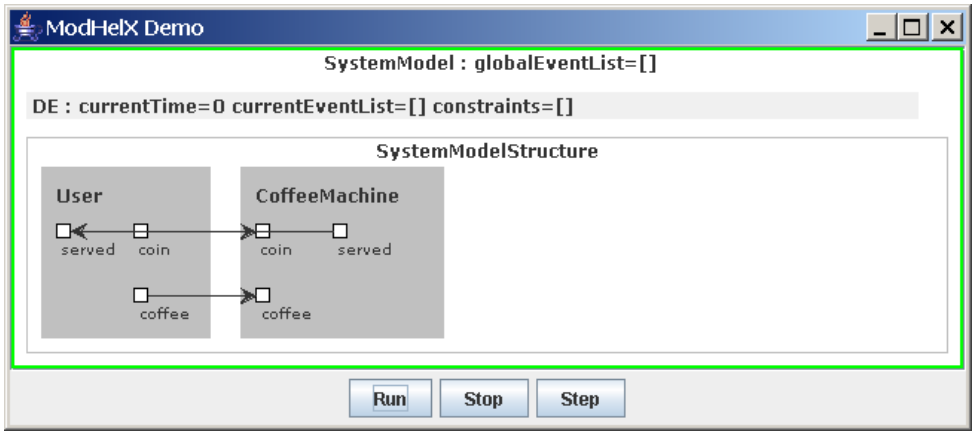


FIG. 5.8 – Modèle du système composé de l'utilisateur et de la machine à café (DE)

Le bloc de gauche représente un utilisateur qui consomme des cafés successifs (en boucle infinie) à des intervalles de temps δ_{user}. Nous avons implémenté cet utilisateur sous la forme d'un

bloc atomique dont nous avons décrit le comportement en Java. Ce bloc dispose de deux points d'interface de sortie lui permettant d'émettre respectivement les événements `coin` – représentant l'insertion de la pièce – et `coffee` – représentant l'appui sur le bouton – ainsi que d'un point d'interface d'entrée, lui permettant de recevoir l'événement `served` – représentant la réception du café. Ce bloc est une source d'événements. En conséquence, lors de son `setup`, ce bloc produit une contrainte de temps afin de déclencher un premier snapshot au temps t_0 , date à laquelle il émet l'événement `coin`. Il produit alors une autre contrainte de temps afin de déclencher un autre snapshot après un délai δ_{coin}, c'est-à-dire au temps $t_0 + \delta_{coin}$, date à laquelle il émet l'événement `coffee`. Il attend alors de recevoir l'événement `served` et émet alors un nouvel événement `coin` avec un délai δ_{user}, c'est-à-dire à la date $t_{served} + \delta_{user}$. Le code Java du bloc atomique représentant l'utilisateur est présenté dans l'annexe C.3.1.

Le bloc de droite représente la machine à café. Il s'agit ici d'un bloc d'interface destiné à recevoir le modèle de la machine à café en modèle interne. Nous présentons le modèle de la machine à café dans la section suivante et détaillons le bloc d'interface et l'adaptation sémantique qu'il réalise dans la section 5.3.4.

L'état du modèle (c'est-à-dire la liste globale des événements) ainsi que les différentes variables d'exécution du modèle de calcul DE sont visibles sur la représentation graphique du modèle afin de pouvoir suivre leur évolution tout au long de l'exécution. Sur la figure 5.8, le modèle est représenté au début de l'étape de `setup`.

5.3.3 Modèle interne de la machine à café : MoC Finite State Machine (FSM)

Le comportement de la machine à café est représenté par un automate, c'est à dire par un modèle impliquant le modèle de calcul Finite State Machine (FSM, voir l'annexe A.1 pour une description succincte de ce modèle de calcul). Nous utilisons ici une version simple de ce modèle de calcul, telle que nous l'avons introduite dans la section 4.6.1.2 du chapitre précédent.

5.3.3.1 Méta-modèle et sémantique d'exécution spécifiques pour FSM

Le méta-modèle spécifique et la sémantique d'exécution de FSM ont été présentés respectivement dans les sections 4.6.1.2 et 4.6.2.3 du chapitre précédent.

En ce qui concerne la représentation des modèles FSM avec notre méta-modèle, rappelons simplement ici que :

- les blocs sont utilisés pour représenter les gardes et les actions présentes sur les transitions ;
- les gardes sont reliées entre elles par des relations (ordre causal entre transitions) ;
- une action est reliée à la garde qui la déclenche (relation de cause à effet) ;
- la notion d'état n'est pas modélisée explicitement : un état est représenté par un ensemble de gardes tirables ;

Rappelons également que le calcul d'un snapshot se déroule de la façon suivante pour FSM :

- Au début du snapshot, il faut déterminer l'ensemble des gardes tirables à partir de l'état courant du modèle. L'ensemble des blocs à mettre à jour pendant le calcul est donc alors l'ensemble des gardes tirables à partir de l'état courant du modèle.
- A chaque tour de boucle du calcul du snapshot (voir la section 4.5.4), il faut choisir un bloc à mettre à jour.
- Après la mise à jour de ce bloc, s'il s'agit d'une garde, il faut vérifier si cette garde s'est évaluée à vrai. Dans la représentation que nous avons choisie, lorsqu'une garde s'évalue à vrai elle produit un jeton sur son point d'interface `enabledAction`. Si c'est le cas, l'automate change d'état c'est-à-dire que l'ensemble des gardes tirables change, et il faut mettre à jour les actions associées à l'état. Dans les tours de boucle suivant il ne faudra donc plus mettre à jour les gardes tirables mais les actions associées à la garde qui a été tirée. Dans le cas où la garde mise à jour ne s'est pas évaluée pas à vrai, alors il faut

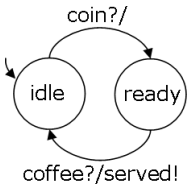


FIG. 5.9 – Automate de la machine à café (représentation originelle)

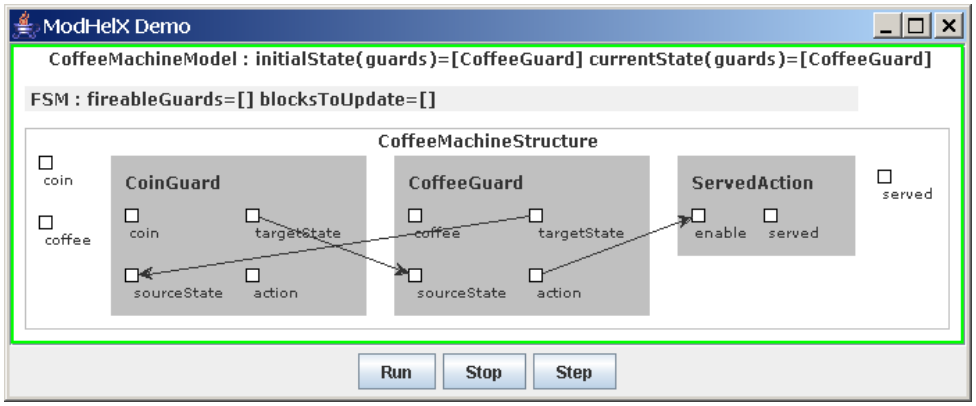


FIG. 5.10 – Modèle de la machine à café (FSM) dans ModHelX

simplement, dans les tours de boucle suivants, continuer de mettre à jour les gardes de l'ensemble des gardes tirables.

– Le snapshot termine soit lorsque toutes les gardes ont été mises à jour (c'est le cas si aucune ne s'est évaluée à vrai) ou lorsque toutes les actions associées à la garde tirée ont été mises à jour.

Le code complet du modèle de calcul FSM est présenté en Java dans l'annexe C.2.

5.3.3.2 Modèle interne de la machine à café dans ModHel'X

L'automate de la machine à café est présenté dans son formalisme originel sur la figure 5.9. Il s'agit d'un automate simple, à deux états : « idle », qui est un état de repos (état initial de l'automate) et « ready », qui est un état d'attente. L'automate passe de l'état « idle » à l'état « ready » lorsqu'un utilisateur introduit une pièce. Si l'utilisateur appuie alors sur le bouton « coffee », l'automate revient alors à l'état « idle » et produit l'événement « served » signifiant que le café est servi.

Comme nous l'avons introduit au début de cette section, ce modèle de comportement est un modèle préliminaire focalisé sur la logique du comportement de la machine. Ce modèle pourra par exemple servir à réaliser le contrôleur de la machine. Nous avons volontairement choisi un modèle simple afin de ne pas compliquer l'exemple. En effet, la représentation graphique basique que nous avons choisie pour notre méta-modèle générique ne donne pas une représentation très intuitive des automates. Dans un modèle plus avancé, nous pourrions tenir compte de la possibilité d'annuler afin de récupérer la pièce introduite dans la machine par exemple.

Soulignons par ailleurs que nous utilisons ici une version non temporisée des automates. Aucune notion de date ni de délai n'existe dans un tel modèle de calcul. En conséquence, ce modèle ne tient pas compte d'un éventuel délai entre l'appui sur le bouton et l'obtention du café.

La figure 5.10 montre le modèle de la machine à café, impliquant le modèle de calcul FSM, tel qu'il est représenté dans le framework ModHel'X. La garde de la transition entre les états « idle » et « ready » est représentée par le bloc `CoinGuard`. La garde de la transition entre les

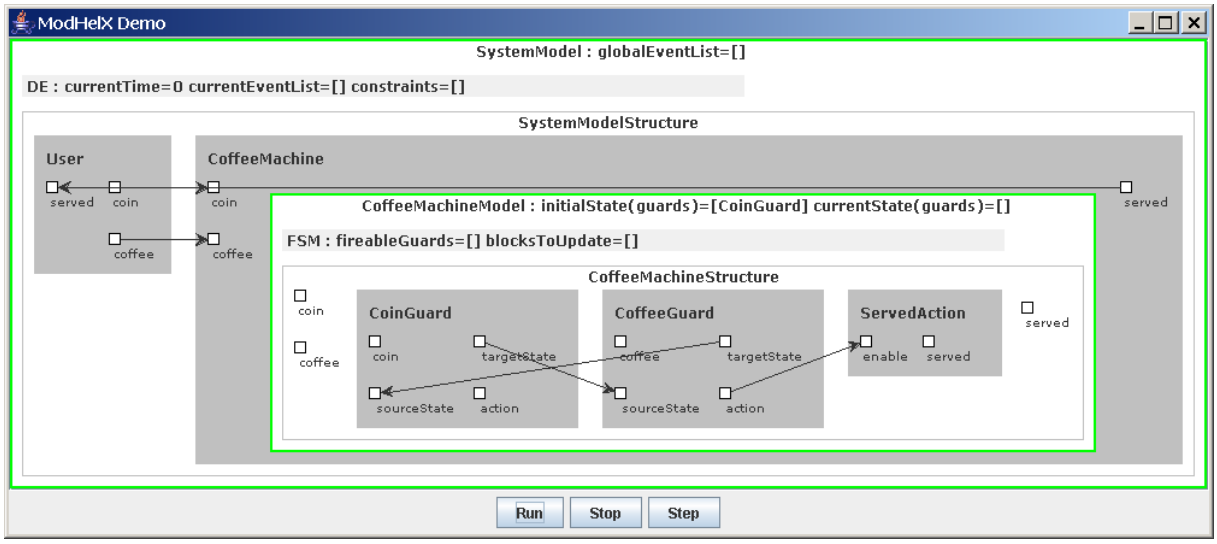


FIG. 5.11 – Modèle global hétérogène de l'exemple de la machine à café

états « ready » et « idle » est représentée par le bloc `CoffeeGuard` et son action associée par le bloc `ServedAction`.

Les deux blocs `CoinGuard` et `CoffeeGuard` sont des instances de `FSMSimpleGuard`, une spécialisation de `FSMGuard`. Une `FSMSimpleGuard` est une garde qui devient vraie (et donc produit un jeton sur son point d'interface « action ») lorsque son événement déclencheur est présent. Ainsi les gardes `CoinGuard` et `CoffeeGuard` deviennent vraies respectivement lorsque les événements `coin` et `coffee` sont présents. Le bloc `ServedAction` est une instance de `FSMSimpleAction`, une spécialisation de `FSMAction`. Une `FSMAction` est une action qui produit un événement lorsque sa garde associée devient vraie c'est-à-dire lorsqu'elle reçoit un jeton déclencheur sur son point d'interface « enable ». Le code Java de ces deux types de blocs est présenté dans les annexes C.3.2 et C.3.3.

Les entrées du modèle sont aliasées par les entrées respectives `coin` et `coffee` des gardes `CoinGuard` et `CoffeeGuard`. Cela signifie que ces deux gardes voient sur leurs entrées respectives tout événement présent sur les entrées du modèle. La sortie du modèle est un alias de la sortie `served` de l'action `ServedAction`. Cela signifie que tout événement produit par cette action est visible sur la sortie du modèle.

L'état initial du modèle est constitué de l'ensemble des gardes tirables {`CoffeeGuard`}. L'état courant du modèle est initialisé au `setup`, moment de l'exécution auquel est représenté le modèle sur la figure 5.10. Les variables de calcul `fireableGuards` et `blocksToUpdate` du modèle de calcul FSM sont visibles sur l'interface afin de pouvoir suivre leur évolution tout au long de l'exécution.

5.3.4 Combinaison des modèles DE et FSM

Disposant d'un modèle du système basé sur DE et d'un modèle du comportement de la machine à café basé sur FSM, notre objectif est d'utiliser le modèle de la machine à café dans le modèle du système afin de pouvoir simuler son comportement. Pour cela, les deux modèles sont combinés hiérarchiquement grâce à un bloc d'interface de manière à ce que le modèle de la machine à café devienne le modèle interne du bloc représentant la machine à café. La figure 5.11 présente le modèle hétérogène résultant de cette combinaison hiérarchique dans ModHel'X.

Le bloc d'interface représentant la machine à café dispose de points d'interface identiques à ceux du modèle FSM. Remarquons au passage que cela n'est pas une règle générale : les points d'interface d'un bloc d'interface et de son modèle peuvent être différents si l'adaptation

sémantique réalisée par le bloc compense la différence (génère les jetons de données manquant pour les points d'interface supplémentaires, etc.). Dans notre exemple, le bloc représentant la machine à café dispose de deux points d'interface d'entrée : `coin` pour « recevoir la pièce de l'utilisateur » et `coffee` pour « détecter l'appui sur le bouton café ». Il dispose également d'un point d'interface de sortie `served` lui permettant de « prévenir l'utilisateur que son café est servi ». Les points d'interface correspondants de la machine à café et de l'utilisateur sont connectés par des relations.

5.3.4.1 Modèle usuel d'adaptation sémantique entre DE et FSM

Les modèles du système et de la machine à café impliquant des modèles de calcul différents, DE et FSM, il est nécessaire de mettre en place un mécanisme d'adaptation sémantique entre les modèles du système et de la machine à café afin d'obtenir un modèle hétérogène global cohérent. Cette adaptation sémantique est réalisée par le bloc d'interface de la machine à café. Comme nous l'avons vu dans la section 4.5.9 du chapitre précédent, celui-ci doit réaliser une adaptation des sémantiques des deux modèles de calcul lors de l'exécution de ses opérations `adaptIn` et `adaptOut`.

DE et FSM partagent la notion d'événement. Cependant, FSM n'attache aucune notion de date aux événements, contrairement à DE. En conséquence, lorsque des événements DE passent dans FSM (via les points d'interface `coin` ou `coffee` sur notre exemple), le bloc d'interface doit transformer ces événements en des événements FSM. Une solution simple est de supprimer leur date. Inversement, lorsque des événements FSM passent dans DE (via le point d'interface `served` sur notre exemple), le bloc d'interface doit transformer ces événements à des événements DE. Il doit donc leur ajouter une date. Tout le problème de cette adaptation est de *déterminer quelle date ajouter aux événements FSM passant dans DE afin que le modèle reste cohérent.*

Il existe une manière acceptable de résoudre ce problème : il s'agit de donner aux événements passant de FSM à DE la date du dernier événement qui est passé de DE à FSM. C'est notamment ce type d'adaptation qui est implémenté par Ptolemy II. Il est possible d'implémenter cette méthode dans un bloc d'interface « patron », c'est-à-dire un bloc d'interface dont les opérations `adaptIn` et `adaptOut` décrivent une méthode usuelle d'adaptation pour une paire de modèles de calcul donnée (voir la section 4.6.3 du chapitre précédent). Ce bloc d'interface stocke dans une variable la date maximale de tous les événements DE qu'il reçoit en entrée avant de créer les événements FSM correspondants (sans date donc). Il utilise la valeur stockée dans cette variable pour affecter une date à tous les événements DE qu'il doit créer en sortie.

Cependant, dans le cadre de notre exemple, cette méthode ne tient pas compte du délai de préparation du café que nous souhaitions faire apparaitre dans notre modèle DE et qui constitue une caractéristique importante du modèle du système. C'est pourquoi nous proposons de rendre ce patron d'adaptation paramétrable de manière à permettre l'obtention d'une date pour les événements passant de FSM à DE qui soit plus pertinente vis-à-vis du modèle dans lequel ce patron est utilisé.

5.3.4.2 Personnalisation du modèle d'adaptation

Ce patron d'adaptation DE/FSM peut être amélioré simplement en ajoutant au bloc d'interface un paramètre qui représente le délai ΔT présent entre les entrées et sorties du bloc – nous le nommons `preparationTime` dans notre exemple car il représente le délai de préparation du café par la machine. La valeur de ce paramètre est ajoutée à la date des événements reçus en entrée du bloc lors de l'affectation d'une date aux événements produits en sortie du bloc. Le code de l'opération `adaptOut` du bloc d'interface `CoffeeMachine` qui réalise ce calcul est présenté sur le listing 5.5 ci-dessous. Le code du bloc d'interface `CoffeeMachine` est présenté en Java dans l'annexe C.3.4.

Remarquons au passage qu'une personnalisation plus poussée du patron d'adaptation proposé ici est possible. Le calcul de la date à affecter aux événements DE peut notamment être beaucoup plus complexe et dépendre de plusieurs paramètres.

Listing 5.5: CoffeeMachineDEFSM::adaptOut()

```
self.internalModel.structure.pinsOut→select(pInt:Pin|pInt.storedTokens→notEmpty())
  →forEach(pInt:Pin){ // Check all the output pins of the internal model
        self.pinsOut→forEach(pExt:Pin){ // If FSM events have been produced by the internal model...
              pExt.storedTokens→append( // ... they become DE events on the outputs of the block
                new DEEvent( // ... with time stamps = last stored time stamp + serving delay
                  self.tLastDEevt + self.parameters→select(name="preparationTime")));
        }
        pInt.storedTokens→clear(); // FSM events are cleared
}
```

Il est important de souligner ici que, si ce modèle avait été réalisé dans Ptolemy II, la personnalisation que nous proposons dans cette section aurait impliqué de modifier le modèle du système. En effet, les mécanismes d'adaptation sémantique entre modèles de calcul sont codés en dur dans le noyau de Ptolemy et ne sont donc pas accessibles au niveau du modèle. Pour prendre en compte le délai de préparation du café dans le modèle hétérogène, il aurait donc fallu ajouter un composant « retard » dans le modèle DE entre la sortie de l'acteur représentant la machine à café et l'entrée de l'acteur représentant l'utilisateur.

5.3.5 Exécution du modèle

Lorsque l'on exécute cet exemple, le moteur d'exécution calcule (au moins) trois snapshots successifs. Le tableau 5.1 résume la trace typique de l'exécution de ce modèle dans ModHel'X (des exécutions différentes peuvent être obtenues en faisant varier les paramètres du modèle, etc). Nous ne montrons dans ce tableau que les pas d'exécution de l'algorithme qui modifient les différentes variables d'exécution. Chaque ligne montre les valeurs de chacune des variables *après* l'exécution du pas de calcul correspondant.

Durant le `setup`, le bloc représentant l'utilisateur produit une contrainte de manière à pouvoir être observé à la date 0 (de façon à ce qu'un snapshot soit réalisé lorsqu'il insère la pièce dans la machine à café). Cette contrainte déclenche le premier snapshot. Au début du snapshot la file des événements du modèle DE est vide et l'ensemble des gardes tirables du modèle FSM contient le bloc `CoinGuard`. A cause de la contrainte produite par le bloc représentant l'utilisateur, la date courante du modèle DE est 0 au premier snapshot. Aucun autre bloc n'ayant produit de contrainte ou émis d'événement, le bloc représentant l'utilisateur est choisi pour être mis à jour lors du `initSchedule` du premier pas d'exécution. Lorsqu'il est mis à jour, le bloc représentant l'utilisateur émet l'événement `coin` avec la date 0. Un deuxième pas d'exécution est réalisé car la date de l'événement `coin` est égale à la date courante du snapshot. Le bloc représentant la machine à café est choisi pour être mis à jour puisqu'il est le destinataire de l'événement `coin`. Avant de déléguer la mise à jour à son modèle interne, le bloc d'interface exécute son opération `adaptIn` pour supprimer la date de l'événement `coin`. Celui-ci est alors déposé sur le point d'interface `coin` du modèle interne. La garde tirable courante `CoinGuard` est mise à jour et consomme l'événement `coin`. `CoffeeGuard` devient la garde tirable courante. Puisque aucun autre événement n'a été produit par l'automate, l'exécution de l'opération `adaptOut` du bloc d'interface n'a pas d'effet. La file d'événements de DE reste vide et le snapshot est donc déterminé. Le bloc représentant l'utilisateur produit une contrainte lors du `endOfSnapshot` de manière à être observé au temps δ_{coin} (c'est-à-dire au moment ou il presse le bouton « café » de la machine à café).

A cause de la contrainte produite par le bloc représentant l'utilisateur, un second snapshot est déclenché et la date courante pour DE est alors δ_{coin}. Au début du snapshot, la file des

Pas d'exécution	**MoC DE**		**InterfaceBlock**	**MoC FSM**
	File des événements	**Contraintes**	**Dernière date stockée**	**Gardes tirables courantes**
Setup	[]	[user : 0]	$t_{lastDEevt} = 0$	`CoinGuard`
Premier snapshot :				
début du snapshot	[]	[user : 0]	$t_{lastDEevt} = 0$	`CoinGuard`
mise à jour de `User`	[coin : 0]	[]	$t_{lastDEevt} = 0$	`CoinGuard`
adaptIn sur `CoffeeMachine`	[]	[]	$t_{lastDEevt} = 0$	`CoinGuard`
mise à jour de `CoinGuard`	[]	[]	$t_{lastDEevt} = 0$	`CoffeeGuard`
adaptOut sur `CoffeeMachine`	[]	[]	$t_{lastDEevt} = 0$	`CoffeeGuard`
fin du snapshot	[]	[user : δ_{coin}]	$t_{lastDEevt} = 0$	`CoffeeGuard`
Second snapshot :				
début du snapshot	[]	[user : δ_{coin}]	$t_{lastDEevt} = 0$	`CoffeeGuard`
mise à jour de `User`	[coffee : δ_{coin}]	[]	$t_{lastDEevt} = 0$	`CoffeeGuard`
adaptIn sur `CoffeeMachine`	[]	[]	$t_{lastDEevt} = \delta_{coin}$	`CoffeeGuard`
mise à jour de `CoffeeGuard`	[]	[]	$t_{lastDEevt} = \delta_{coin}$	`CoinGuard`
mise à jour de `ServedAction`	[]	[]	$t_{lastDEevt} = \delta_{coin}$	`CoinGuard`
adaptOut sur `CoffeeMachine`	[served : $\delta_{coin} + \Delta T$]	[]	$t_{lastDEevt} = \delta_{coin}$	`CoinGuard`
fin du snapshot	[served : $\delta_{coin} + \Delta T$]	[]	$t_{lastDEevt} = \delta_{coin}$	`CoinGuard`
Troisième snapshot :				
début du snapshot	[served : $\delta_{coin} + \Delta T$]	[]	$t_{lastDEevt} = \delta_{coin}$	`CoinGuard`
mise à jour de `User`	[...]	[]	$t_{lastDEevt} = \delta_{coin}$	`CoinGuard`
...				

Tab. 5.1 – Trace de l'exécution du modèle hétérogène de l'utilisateur et de la machine à café

événements de DE est vide et l'ensemble des gardes tirables du modèle FSM contient le bloc `CoffeeGuard`. Les mêmes pas d'exécution qu'au premier snapshot se déroulent, jusqu'à ce que l'événement `coffee`, produit par le bloc représentant l'utilisateur avec la date δ_{coin} soit délivré au bloc représentant la machine à café. Pendant l'exécution de l'opération `adaptIn`, la date de l'événement `coffee` est retirée et sa valeur stockée. L'événement `coffee` est alors déposé sur le point d'interface `coffee` du modèle interne. La garde tirable courante `CoffeeGuard` est mise à jour. `CoffeeGuard` consomme l'événement `coffee`, `CoinGuard` devient la garde tirable courante et l'action `ServedAction` est ordonnancée pour mise à jour. Lorsqu'elle est mise à jour, l'action `ServedAction` produit l'événement `served` et la mise à jour du modèle FSM est terminée. L'opération `adaptOut` calcule la date de l'événement `served` en additionnant la valeur du paramètre `preparationTime` à la date du dernier événement stocké, c'est-à-dire la date de l'événement `coffee`. L'événement `served` est donc émis par la machine à café avec une date égale à $\delta_{coin} + \Delta T$ et stockée dans la file des événements de DE. Puisque cet événement a une date dans le futur par rapport à la date courante du snapshot, et qu'aucun autre événement à la date courante ne reste dans la file, le snapshot est déterminé.

Le dernier snapshot est déclenché car l'événement `served`, qui est dans la file des événements de DE, reste à traiter. La date courante dans le modèle DE devient alors $\delta_{coin} + \Delta T$ et l'événement `served` est remis au bloc représentant l'utilisateur. Ce qui arrive ensuite dépend de l'implémentation qui a été faite du bloc représentant l'utilisateur. Dans notre exemple, ce bloc représente un utilisateur qui boit du café toute la journée à des intervalles de temps δ_{user}. Il produit donc un nouvel événement `coin` immédiatement avec une date égale à $\delta_{coin} + \Delta T + \delta_{user}$ et le cycle des snapshots se répète.

5.4 Conclusion

Nous avons présenté dans ce chapitre l'implémentation que nous avons réalisée de notre approche : le framework ModHel'X. Notre objectif en réalisant ce framework était de pouvoir

mettre en œuvre notre approche de manière concrète et expérimenter son utilisation sur différents modèles de calcul et différents modèles. Pour cela, nous nous sommes placés dans un contexte technique standard du domaine MDA. Lors de la réalisation, nous nous sommes heurtés à de nombreuses difficultés, liées à la fois aux spécificités de notre approche et aux fonctionnalités encore peu poussées des outils supports du domaine MDA par rapport à ces spécificités. Nous avons présenté les problèmes rencontrés et montré comment ceux-ci nous ont amenés à modifier de manière provisoire l'architecture choisie pour le framework afin de pouvoir mener des expérimentations sur notre approche dans un délai raisonnable.

Les différents tests réalisés sur notre implémentation nous ont tout d'abord permis de valider notre approche. Nous avons ainsi pu vérifier la bonne exécution de notre algorithme hiérarchique sur un modèle à plusieurs niveaux. Par ailleurs, même si les difficultés rencontrées lors de l'implémentation nous fait prendre du retard dans la validation globale de notre approche, nous avons pu décrire plusieurs modèles de calcul dans notre framework et exécuter des modèles homogènes et hétérogènes les utilisant. Nous avons présenté dans ce chapitre deux modèles de calcul différents et un exemple de modèle hétérogène impliquant conjointement ces deux modèles de calcul et mettant en oeuvre un mécanisme d'adaptation sémantique paramétrable entre ceux-ci. Nous avons réalisé un autre modèle de calcul, le modèle de calcul « *charts » pour les systèmes modaux [HBMVN07], dans une version de notre framework antérieure à celle décrite dans ce chapitre. Nous travaillons actuellement à la description du modèle de calcul Synchronous DataFlow (SDF) (voir l'annexe A.4.3).

Nous continuons également de travailler sur différentes solutions afin de résoudre les différents problèmes rencontrés dans la mise en œuvre des outils support MDA dans notre framework. Nous estimons que l'architecture que nous avons conçue pour ce framework sera réalisable en utilisant des versions améliorées des outils support. Il s'agit cependant d'un projet à long terme.

6 Conclusion

Les travaux présentés dans ce mémoire traitent de l'hétérogénéité des modèles dans le contexte de l'application de l'Ingénierie Dirigée par les Modèles (IDM) au développement de systèmes complexes. Nous abordons ce sujet selon deux axes : la proposition d'un cadre d'étude pour le domaine de la Modélisation Multi-Paradigme, et la conception d'une approche destinée à faciliter l'utilisation conjointe de modèles hétérogènes tout au long du cycle de développement.

6.1 Problématique

La problématique qui nous intéresse dans ce mémoire résulte du constat suivant : lors de l'application de l'IDM au développement de systèmes complexes, il est à la fois inévitable et essentiel d'utiliser un ensemble de paradigmes de modélisation différents qui dépendent à la fois :

1. des différents *domaines* métiers en jeu dans le système conçu : matériel, logiciel, contrôle, traitement du signal, mécanique, etc.
2. des différentes *activités* du cycle de développement : analyse des besoins, conception, test, validation, etc.
3. des différents niveaux d'*abstraction* auxquels le système est étudié : niveau système, niveau algorithme, niveau registre, etc.
4. des différents aspects spécifiques à analyser lors de l'ensemble du cycle de développement, qui nécessitent de disposer de différentes *vues* du système : fonction, performance, consommation d'énergie, etc.

Nous montrons en effet que, tout au long du cycle de développement, le concepteur passe par différents *objectifs de modélisation* qu'il est possible de caractériser par un quadruplet $<$*domaine, abstraction, activité, vue*$>$. En conséquence, à un instant donné du cycle, le système en cours de conception est représenté par un ensemble de modèles décrits dans des paradigmes différents, aussi appelés *modèles hétérogènes*. Mener un raisonnement global sur cet ensemble de modèles est alors très difficile et source de nombreuses erreurs.

Les recherches sur cette problématique ont récemment donné naissance au domaine de recherche appelé « Modélisation Multi-Paradigme ». Cependant, les problèmes liés à l'hétérogénéité des modèles sont bien antérieurs à l'IDM. Des solutions de natures diverses ont d'ailleurs été mises au point par différentes communautés de chercheurs pour résoudre des problèmes d'hétérogénéité des modèles spécifiques à leurs disciplines. L'un des enjeux majeurs de la modélisation multi-paradigme est d'apporter une vision globale sur les problématiques, l'état de l'art et la terminologie liés à l'hétérogénéité des modèles.

6.2 Apports

Partant de ce constat, nous définissons un cadre pour l'étude de la modélisation multi-paradigme. Ce cadre s'appuie sur les quatre sources d'hétérogénéité que nous avons identifiées (domaines, niveaux d'abstraction, activités, vues). Dans ce cadre, nous déclinons les axes de recherche envisagés en fonction des caractéristiques <*domaine, abstraction, activité, vue*> des objectifs de modélisation des concepteurs de systèmes. Ces caractéristiques sont en effet liées à des problématiques spécifiques : adaptation sémantique entre modèles de domaines différents, conformité d'un raffinement à un modèle plus abstrait, consistance des différents modèles d'un composant construits pour différentes activités, et enfin cohérence et synchronisation des modèles représentant différentes vues d'un même composant.

Toujours dans ce cadre, nous étudions la pertinence d'un ensemble de techniques issues de diverses disciplines vis-à-vis du traitement de l'hétérogénéité des modèles. Nous présentons un état de l'art de ces techniques en détaillant comment celles-ci abordent le problème de l'hétérogénéité. Nous complétons cet état de l'art en proposant des critères qui définissent les qualités que nous estimons essentielles pour ces techniques de modélisation multi-paradigme : leur ouverture à de multiples paradigmes, leur utilisabilité à plusieurs étapes du cycle de conception et enfin leur capacité à effectuer des vérifications sur les modèles en s'appuyant sur des formalismes mathématiques.

Nous nous intéressons ensuite plus particulièrement à la caractéristique *domaine* des objectifs de modélisations. Nous proposons une approche, appelée *ModHel'X*, dont l'objectif est de permettre de composer de manière flexible des modèles faisant appel à des langages de modélisation issus de domaines métiers différents. Cette approche peut être appliquée à n'importe quel langage de modélisation. Elle est dédiée à l'exécution de modèles hétérogènes, leur validation formelle n'étant envisagée qu'en perspective pour le moment.

ModHel'X se situe dans la lignée de Ptolemy II et, comme celui-ci, s'appuie sur une représentation uniforme des modèles quel que soit le langage de modélisation dans lequel ils sont décrits. Ceci est rendu possible par la mise en œuvre du concept de « modèle de calcul (MoC) ». Utiliser ce concept consiste à considérer qu'un langage de modélisation peut être vu comme un ensemble de règles qui indiquent comment interpréter la structure d'un modèle, cet ensemble de règles constituant le *modèle de calcul*. Les règles qui constituent un modèle de calcul définissent deux aspects sémantiques : les aspects contrôle et les aspects communication. A la différence de Ptolemy, dans lequel certaines des règles qui constituent un modèle de calcul sont attachées à des éléments de structure d'un modèle (les « ports »), dans ModHel'X les règles constituant le modèle de calcul sont toutes attachées à une entité distincte, appelée « modèle de calcul », qui est chargée d'interpréter le modèle. Ainsi, un même modèle peut être interprété de deux façons différentes en remplaçant l'entité « modèle de calcul » qui lui est associée.

Dans ModHel'X, nous proposons une méthode permettant d'exécuter un modèle selon les règles constituant le modèle de calcul qui lui est associé. Dans cette méthode, le comportement d'un modèle est défini par la succession des observations de ses composants. L'activité des composants n'est pas contrainte de se produire à des instants précis, mais l'hypothèse est faite qu'un composant peut donner une vue consistante de son interface lorsque cela lui est demandé. La construction d'une observation d'un modèle consiste donc à observer les composants du modèle et à propager les informations ainsi obtenues aux autres composants jusqu'à ce que plus aucune nouvelle information ne soit obtenue. Les règles qui constituent le modèle de calcul déterminent, avec la structure du modèle, dans quel ordre les composants du modèle seront observés et de quelle manière les informations obtenues seront propagées aux autres composants. L'algorithme de construction des observations est donc une succession d'étapes comprenant le choix d'un composant à observer, l'observation par mise à jour de l'interface de ce composant, et la propagation dans le modèle des informations observées. Un modèle de calcul est décrit dans ModHel'X par la définition de ces opérations élémentaires.

La convergence du processus de construction d'une observation est garantie en l'absence de dépendances cycliques entre les composants. Lorsque de telles dépendances sont autorisées, ce sont les règles du modèle de calcul qui déterminent le point fixe qui constitue l'observation. L'absence de point fixe constitue une erreur. L'existence de plusieurs points fixes peut entraîner la construction d'une observation qui n'est pas le point fixe souhaité. Il est alors possible dans notre approche de reconstruire l'observation avec des paramètres différents afin de converger vers un point fixe compatible avec la sémantique du modèle de calcul. Ce mécanisme est une généralisation à tous les modèles de calcul d'un mécanisme introduit spécifiquement dans le domaine CT (Continuous Time) de Ptolemy II .

ModHel'X supporte l'hétérogénéité en permettant de composer hiérarchiquement des modèles impliquant des modèles de calcul différents. Pour cela, il est possible, dans ModHel'X, de décrire le comportement d'un composant à l'aide d'un modèle qui peut être interprété selon un modèle de calcul différent de celui dans lequel le comportement du composant est observé. Cette structuration de l'hétérogénéité par la hiérarchie utilisée dans ModHel'X est similaire à celle mise en œuvre dans Ptolemy II. Cependant, à la différence de Ptolemy, dans ModHel'X, l'adaptation sémantique entre les modèles de calcul interne et externe d'un composant est spécifiée explicitement à l'interface entre les deux modèles et fait donc partie de la conception du système. Il est possible, dans ModHel'X, de créer des patrons d'adaptation sémantique, qui définissent des modèles d'adaptation sémantique pour des paires de modèles de calcul. Un concepteur de système peut choisir d'utiliser l'un de ces patrons, le paramétrer pour l'adapter au système considéré, ou même concevoir sa propre adaptation sémantique entre deux modèles. L'adaptation sémantique explicite entre deux modèles impliquant des modèles de calcul différents est l'un des principaux apports de cette thèse.

Notre approche est implémentée dans un framework appelé, lui aussi, ModHel'X. Ce framework a été conçu pour être intégré à la plate-forme Eclipse et pour s'appuyer sur le Eclipse Modeling Framework (EMF) afin de bénéficier de l'outillage d'ingénierie des modèles existant. Il définit un méta-modèle permettant la représentation uniforme des modèles et permet la simulation du comportement de modèles hétérogènes en appliquant les opérations élémentaires d'ordonnancement, d'observation et de propagation telles qu'elles sont définies pour les modèles de calcul associés à ces modèles. Ce framework permet à des spécialistes de décrire la sémantique de langages de modélisation sous la forme de modèles de calcul et de combiner des modèles impliquant des modèles de calcul différents à des fins expérimentales. Il est distribué sous licence Eclipse Public Licence.

6.3 Perspectives

Les travaux que nous présentons dans ce mémoire ouvrent des perspectives pour le domaine de la modélisation multi-paradigme, pour l'approche de composition de modèles hétérogènes que nous proposons et pour l'implémentation que nous en avons réalisé.

La Modélisation Multi-Paradigme

Le domaine de la modélisation multi-paradigme est un domaine jeune qui fédère des communautés très différentes. Les travaux que nous présentons dans ce mémoire nous ont permis d'affiner la définition de ce domaine. De nombreuses perspectives sont envisageables, tant sur le plan de la caractérisation du domaine que sur le plan de la taxonomie des approches. Il nous paraît notamment intéressant d'affiner nos critères de comparaison pour les approches de modélisation multi-paradigme afin de former une classification plus complète.

L'approche ModHel'X

Les travaux que nous avons réalisés sur la composition de modèles hétérogènes pour l'exécution soulèvent différentes questions, que nous avons abordées dans la section 4.7 du chapitre 4. Ces questions ouvrent autant d'axes de réflexion pour nos travaux sur ModHel'X, que ce soit sous la forme d'extensions à court terme ou de thèmes de recherche à plus long terme. Parmi ceux-ci, nous nous intéressons plus particulièrement aux sujets suivants :

Sémantique d'exécution des modèles de calcul Nous décrivons la sémantique d'exécution des modèles de calcul de manière opérationnelle avec un langage impératif. Dans un objectif d'amélioration de l'utilisabilité de notre approche, il nous paraît intéressant d'envisager d'autres solutions, telles que l'utilisation d'un langage déclaratif, afin d'obtenir un niveau d'abstraction et un niveau d'expressivité plus satisfaisants.

Mécanismes de composition de modèles de calcul Dans la même optique, nous aimerions permettre la description des mécanismes de composition de modèles de calcul sous la forme de modèles, par exemple sous la forme d'assemblage de composants d'adaptation. Il s'agit d'un problème à part entière car cela implique de pouvoir utiliser des composants qui obéissent simultanément à deux modèles de calcul différents. Nous parlons donc ici d'hétérogénéité non hiérarchique et les résultats présentés dans [Mbo04] pourraient être appliqués.

Sémantique formelle L'un des axes de recherche que nous envisageons à plus long terme concerne la formalisation mathématique de notre approche. Notre objectif est d'obtenir des descriptions formelles des modèles de calcul décrits dans ModHel'X. De telles descriptions pourraient alors être utilisées par d'autres outils pour faire du model-checking, de la validation, etc. L'utilisation des co-algèbres nous paraît être une piste intéressante car elles supportent nativement la notion d'observation. Il faudrait certainement les combiner avec des algèbres afin d'obtenir le caractère constructif des sémantiques d'exécution.

Test de conformité pour les modèles hétérogènes Notre approche, orientée vers l'exécution des modèles, s'applique particulièrement au domaine du test. Différents points nous paraissent intéressants à traiter dans le cadre d'une adaptation de notre approche au test de conformité :

- L'application du concept de modèle de calcul à la combinaison de propriétés vérifiées unitairement sur les blocs formant le modèle ;
- Les mécanismes de traduction de propriétés entre modèles de calcul ;
- La génération de cas de test pour les modèles hétérogènes.

Extension à d'autres types d'hétérogénéité Notre approche traitant de l'hétérogénéité des domaines, il serait intéressant d'envisager de l'étendre aux autres types d'hétérogénéité que nous avons définis dans nos travaux :

- En ce qui concerne l'hétérogénéité des niveaux d'abstraction, le sujet de la relation de conformité entre des modèles hétérogènes à des niveaux d'abstraction différents nous semble important à traiter. En effet, un des intérêts de l'IDM est de permettre de valider la conception d'un système plus en amont. Cette validation n'a d'intérêt que s'il est possible de vérifier que des modèles raffinés du système sont conformes aux spécifications qu'ils implémentent. Si certains formalismes permettent de prouver que les raffinements de composants sont conformes à leurs spécifications, le problème de la conformité d'un modèle d'exécution à un modèle de calcul reste ouvert.
- Vis-à-vis de l'hétérogénéité des vues, l'utilisation des principes définis dans des approches telles que Rosetta [KA03] dans notre contexte permettrait d'aborder la composition de différentes vues d'un même composant sous l'angle de la simulation.
- Enfin, en capitalisant sur la sémantique formelle que nous nous proposons de définir et sur l'extension au test que nous envisageons, notre approche pourrait être étendue à d'autres activités du cycle de développement (model-checking, test, etc.).

Le framework ModHel'X

L'implémentation que nous avons réalisée de notre approche peut être développée, tant à court terme qu'à long terme, pour mieux cibler les concepteurs de systèmes. En effet, dans sa version actuelle, le framework ModHel'X est un outil uniquement destiné à des fins expérimentales. Son utilisation nécessite une très bonne connaissance de notre approche et de la notion de modèle de calcul.

La mise en œuvre d'EMF et de GMF, telle qu'elle était prévue initialement, devrait dans un premier temps faciliter l'utilisation du framework par des experts des modèles de calcul. Nous travaillons actuellement sur ce sujet afin de trouver des moyens de contourner les limitations que nous avons évoquées dans le chapitre 5.

A plus long terme, ModHel'X pourrait être intégré en tant que moteur sous-jacent dans une chaîne de modélisation qui fournirait une interface plus adaptée pour les concepteurs de systèmes. En effet, ModHel'X n'est pas en soi un nouvel outil de modélisation mais bien un framework permettant de fédérer des outils de modélisation existants, tels que Simulink, Scade ou Rational Software Architect. Ce sont les experts de ces outils qui seront chargés de décrire dans ModHel'X la sémantique des langages de modélisation correspondant et de définir la manière dont des modèles décrits dans ces langages peuvent être composés.

Troisième partie

Annexes

A

Modèles de calcul courants

A.1 Modèle de calcul « Finite State Machine (FSM) »

Le modèle de calcul FSM est un modèle de calcul fondamental très utilisé. Un automate, ou machine à états (state machine), permet de modéliser le comportement d'un système lorsque celui-ci peut être décomposé explicitement en états discrets successifs. Lorsque l'espace d'états du système modélisé est fini, on parle d'automates finis.

Un modèle FSM est constitué de trois éléments : les états, les transitions et les actions. Un état représente un état du système. Le comportement du système dans un état peut être non défini, ou peut être défini par un autre modèle, que l'on nomme le « raffinement » de l'état. Si ce raffinement est lui-même un automate, on a un automate hiérarchique. Chaque transition est associée à un événement qui déclenche le passage du système de l'état de départ de la transition à son état de destination. Il est aussi possible d'associer une garde à une transition, qui indique si la transition doit être effectuée. Les actions correspondent à la description d'activités qui doivent être exécutées à un moment donné, comme par exemple lorsqu'une transition est suivie, lorsque le système entre dans un état, etc.

Les automates sont particulièrement bien adaptés à la modélisation de la logique de commande, notamment pour les systèmes critiques. En effet, les automates se prêtent à des méthodes formelles d'analyse qui permettent de valider certaines propriétés. Enfin, il est facile d'implémenter un automate aussi bien en logiciel qu'en matériel. Un inconvénient des automates est leur faible expressivité : il s'agit d'une modélisation d'assez bas niveau, et de petites modifications de la spécification d'un système peuvent entraîner de grands changements dans la structure d'un automate. Un autre inconvénient est que le nombre d'états d'un automate peut devenir très grand, même pour un système de complexité modérée.

A.2 Modèle de calcul « Discrete Events (DE) »

Le modèle DE repose sur le concept de temps discret, qui prend la forme d'une ligne temporelle globale constituée d'une suite d'événements discrets. Un événement est composé d'une valeur et d'une date. Les composants sont des processus qui traitent les événements et en produisent. Une connexion entre deux composants représente une suite d'événements placés sur la ligne de temps, aussi appelée signal. Lorsqu'un événement est produit, sa date d'occurrence est indiquée, et l'événement ne sera présenté au composant destinataire que lorsque cette date sera atteinte. Il peut exister un délai entre la réception par un composant d'un événement et la production d'un autre événement en réaction à celui-ci. Les événements ayant chacun une date distincte, le temps séparant deux instants a une signification.

Les modèles DE sont utilisés pour la modélisation de réseaux de communication et pour la

modélisation de matériel. Des langages comme VHDL ou Verilog implémentent un modèle de calcul de type DE.

A.3 Modèle de calcul « Synchronous/Reactive (SR) »

Dans le modèle réactif synchrone (SR), les connexions entre composants représentent des valeurs alignées sur les tics d'une horloge globale. Le modèle ne suppose pas que les tics soient régulièrement espacés dans le temps ni que les signaux échangés aient une valeur pour tous les tics de l'horloge. La présence de cette horloge globale implique que le modèle SR est également un modèle basé sur le Temps (discret). Chaque tic de l'horloge correspond à un instant où le système doit réagir à son environnement. Les composants représentent des relations, pouvant être partielles, entre leurs entrées et leurs sorties à chaque tic. Dans ce modèle, la transmission des données, tout comme la réaction d'un composant à ses entrées, est considérée comme instantanée. Lorsqu'un modèle comporte une boucle dans les connexions entre ses constituants, un composant peut avoir une entrée qui dépend instantanément de ses sorties. Le comportement du modèle est alors défini comme le point fixe du comportement en boucle ouverte. Ce point fixe peut être calculé directement lors d'une phase de compilation, ou être calculé itérativement à l'exécution. Si le point fixe n'existe pas, où s'il en existe plusieurs, le comportement du modèle n'est pas défini.

Ce modèle de calcul est souvent utilisé pour modéliser des systèmes ayant une logique de contrôle complexe, par exemple les applications temps réel critiques. Des langages comme Lustre, Esterel et Signal utilisent ce modèle.

A.4 Modèles de calcul à base de processus

A.4.1 Modèle de calcul « Process Networks (PN) »

Dans le modèle PN, les composants sont des processus séquentiels indépendants, communiquant par passage asynchrone de messages via des files. Une connexion entre deux composants représente donc une file de messages, généralement de type FIFO. Soulignons que le concept de temps n'intervient pas dans ce modèle : la seule contrainte concerne l'ordre des messages dans les files de communication, il n'existe aucun ordre total sur aucun ensemble d'objets. L'exécution d'un tel modèle n'est a priori pas déterministe.

A.4.2 Modèle de calcul « Kahn Process Networks (KPN) »

Les réseaux de processus de Kahn (KPN) sont un cas particulier du modèle PN dans lequel un processus ne peut pas savoir si une file de message est vide ou non. La taille des files de message n'est pas limitée ce qui implique que l'écriture dans la file n'est pas bloquante (la file ne peut jamais être pleine). La lecture est consommatrice et, à l'inverse de l'écriture, celle-ci est bloquante lorsque la file est vide. Ces restrictions garantissent que le réseau est déterministe. Sa terminaison et son occupation mémoire sont même décidables sous certaines conditions.

Les réseaux de processus sont particulièrement bien adaptés au traitement du signal. Par contre ils sont très mal adaptés à la spécification de logiques de contrôle complexes, la logique de contrôle y étant limitée au routage des données.

A.4.3 Modèles de calcul « DataFlow Process Networks (DF) » et « Synchronous DataFlow (SDF) »

Les modèles de calcul à flot de données (DF, SDF, etc) sont souvent considérés comme un cas particulier des réseaux de processus dans lequel les processus sont construits comme des

séquences d'activations atomiques des composants. Dans le modèle DF, les composants sont des opérateurs effectuant un calcul atomique déclenché par la disponibilité des données. A chaque déclenchement d'un composant, celui-ci consomme un nombre donné de jetons sur ses entrées et produit un nombre donné de jetons sur ses sorties. Les nombres de jetons consommés et produits sont déterminés par les règles de déclenchement de chacun des composants.

Le modèle SDF est une version restrictive du modèle DF dans laquelle le taux de consommation et de production des jetons est statiquement fixé pour tous les composants. Cette propriété est particulièrement utile car elle permet de déterminer un ordonnancement statique des composants.

A.4.4 Modèle de calcul « Communicating Sequential Processes (CSP) »

Dans certains systèmes, les processus communiquent par rendez-vous plutôt que par file de messages. Le modèle CSP est conçu pour représenter spécifiquement ce type de système. Le mécanisme de rendez-vous correspond à un mode de communication synchrone dans lequel l'action de communication est atomique et instantanée. Ce modèle de calcul est celui utilisé par les langages tels que Lotos et Occam. Les modèles à rendez-vous sont adaptés à la modélisation de systèmes dans lesquels le problème principal est le partage de ressources. Leur principal inconvénient est qu'il est difficile d'obtenir des modèles déterministes en utilisant ces modèles de calcul.

B

Extension impérative d'OCL pour la description de modèles d'exécution

Nous présentons dans ce chapitre un langage impératif dont le rôle est de permettre la description des étapes de notre algorithme. Ce langage est un sous-ensemble du langage ImperativeOCL [OMGe] décrit dans la spécification QVT [OMGe] et basé sur le langage OCL [OMGh].

Il est important de souligner ici que le présent chapitre ne constitue pas une spécification du langage mais a plutôt pour objectif de délimiter les parties reprises des spécifications OCL et QVT. Les détails de spécification ne sont pas stabilisés pour le moment. Des informations complémentaires peuvent être trouvées dans les spécifications OCL et QVT.

B.1 Objectifs du langage

Avec ce langage, nous voulons être capable de :
- déclarer et assigner des variables ;
- accéder aux attributs et extrémités d'associations des éléments d'un modèle et les modifier ;
- appeler les opérations des éléments d'un modèle ;
- exprimer des instructions conditionnelles ;
- répéter des instructions (sur les éléments d'une collection ou jusqu'à ce qu'une condition soit vraie).

B.2 Syntaxe abstraite

B.2.1 Types

OCL, sur lequel s'appuie ImperativeOCL, est un langage typé. Cela signifie qu'un type est associé à toute expression et que des règles de typage déterminent de quelle façon des expressions bien formées peuvent être construites.

Les types primitifs du langage sont les booléens, les entiers, les réels et les chaines de caractères.

Les types complexes du langage sont les collections paramétrées et les énumérations. Les différents types de collections paramétrées supportés sont les suivants :
- `Set<T>` : ensemble compris au sens mathématique du terme, c'est-à-dire collection non ordonnée et sans doublons d'éléments du type T ;
- `Bag<T>` : sac, c'est-à-dire collection non ordonnée avec possibilité de doublons. ;
- `OrderedSet<T>` : ensemble muni d'une relation d'ordre sur les éléments ;
- `Sequence<T>` : séquence, c'est-à-dire collection ordonnée avec possibilité de doublons.

Les énumérations sont des listes ordonnées de valeurs littérales qui sont définies par les utilisateurs.

OCL permet de manipuler les éléments d'un modèle. Pour ce faire, chaque élément du méta-modèle définissant les éléments du modèle définit également un type. Dans notre contexte, chaque élément du méta-modèle générique qui décrit notre syntaxe abstraite et chaque élément d'un méta-modèle spécialisé pour un modèle de calcul définit donc un type.

Nous devons par ailleurs permettre à l'utilisateur de définir des types de données complexes qui peuvent être utiles dans le cadre de certains modèles de calcul. Ces types peuvent être ajoutés en dérivant `OclAny`, qui est par ailleurs un super-type pour tous les types excepté les collections.

B.2.2 Opérations

Les opérations existant sur les types primitifs sont les opérations classiques définies pour les booléens, les entiers, les réels et les chaines de caractères (voir l'Annexe A de la spécification d'OCL [OMGh, p. 191]).

Les opérations définies sur les types correspondant à des éléments d'un modèle sont :
- les opérations d'accès aux attributs des éléments ;
- les opérations de navigation qui permettent d'accèder aux extrêmités d'association des éléments ;
- les opérations des éléments (c'est-à-dire les méthodes), traitées comme des *fonctions* lors de leur appel ;
- des opérations prédéfinies permettant par exemple de collecter toutes les instances d'un même type, etc.

Les collections sont assorties d'opérations de parcours et de sélection des éléments, dont notamment :
- l'itération simple (`forEach`) ;
- la sélection d'éléments respectant une condition (`select`) ;
- la collection d'éléments dérivés des éléments de la collection (`collect`).

B.2.3 Expressions du langage

Les expressions du langage sont celles d'OCL auxquelles ont été ajoutées des expressions impératives. Les types d'expressions supportés sont :
- les littéraux, qui représentent des valeurs ;
- les déclarations de variable, qui associent un nom de variable à un type de variable ;
- les assignations de variables, qui associent une expression à une variable, le type de l'expression devant être compatible avec le type auquel est associée la variable ;
- les appels de propriétés, qui permettent d'accéder aux propriétés d'un élément d'un modèle (ses attributs, ses opérations et ses extrémités d'associations), de les modifier ou de les exécuter ;
- les structures de contrôle :
  - les expressions conditionnelles (`if-else`) ;
  - les expressions de boucle (`while` et `forEach`).

B.2.4 Contexte (`self`)

Les expressions OCL sont décrites dans un contexte spécifié explicitement, qui correspond généralement à un type d'élément (une classe particulière dans un modèle UML par exemple). Elles sont utilisées pour définir des invariants, des pré ou des post conditions dans ce contexte.

Notre utilisation du langage est un peu différente : nous décrivons le corps d'opérations d'exécution. Dans ce cadre, le contexte est implicite et il s'agit d'une instance du type d'élément

sur lequel est définie l'opérations décrite. Le mot-clé `self` désigne le contexte courant, c'est-à-dire l'instance du type d'élément sur lequel est définie l'opérations décrite. Par exemple, lorsque l'on décrit le corps de l'opération `initSchedule` du modèle de calcul `DEMoC` qui est un sous-type de `ModelOfComputation`, `self` réfère à une instance de ce modèle de calcul.

B.3 Syntaxe concrète

Nous adoptons la syntaxe concrète définie pour ImperativeOCL (pas encore très stable malheureusement). Il est à mentionner que l'affectation est notée `:=` tandis que la comparaison est notée `=` (syntaxe provenant d'OCL).

B.4 Sémantique

Une sémantique formelle non normative est proposée pour OCL dans l'annexe A de la spécification [OMGh, p. 191]. Cependant aucune sémantique formelle n'est proposée pour l'extension impérative pour le moment.

C

Détails d'implémentation du framework ModHel'X

C.1 Implémentation du modèle de calcul FSM

C.1.1 Méta-modèle spécifique pour FSM

La figure C.1 montre le méta-modèle spécifique du modèle de calcul FSM. Les différents éléments de ce méta-modèle sont présentés dans la section 4.6.1.2.

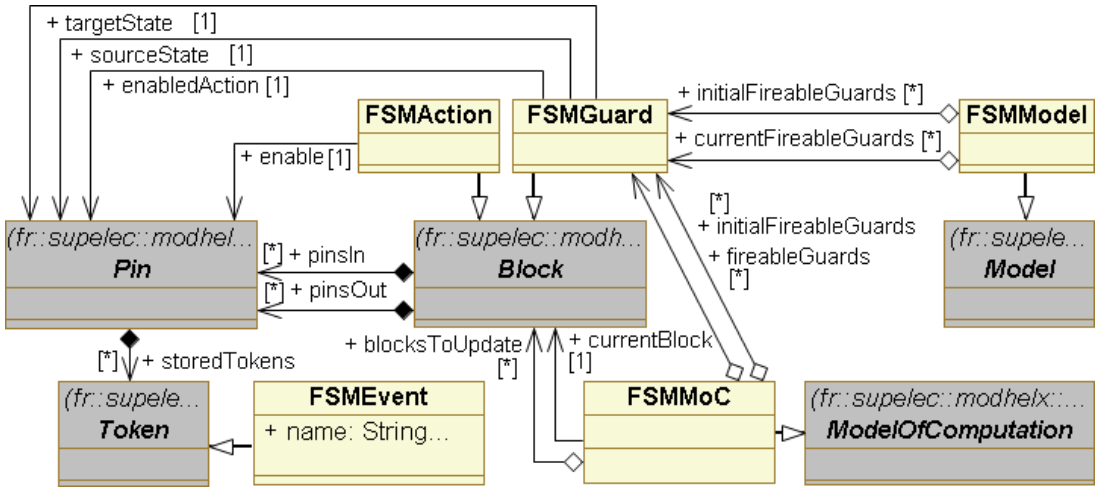


FIG. C.1 – Méta-modèle spécifique pour FSM

C.1.2 Sémantique d'exécution spécifique pour FSM : code Java

Les principes de l'exécution de modèles FSM sont exposés dans la section 4.6.2.3. Le listing C.1 suivant présente le code complet du modèle de calcul FSM en Java.

Listing C.1 – Classe FSMMoCImpl

```
/**
 *
 * FSMMoCImpl
 *
 * @author Cecile Hardebolle {cecile.hardebolle@supelec.fr}
 */
public class FSMMoCImpl extends ModelOfComputationImpl implements FSMMoC {

    //Liste initiale des gardes tirables dans l'état courant
    protected ArrayList<FSMGuardImpl> initialFireableGuards;
```

```
//Liste des gardes tirables dans l'etat courant
protected ArrayList<FSMGuardImpl> fireableGuards;
//Liste des blocks à mettre à jour
protected ArrayList<? extends Block> blocksToUpdate;
//Block courant à mettre à jour
protected Block currentBlock;

public FSMMoCImpl() {
    super("FSM");
    this. initialFireableGuards  = new ArrayList<FSMGuardImpl>();
    this. fireableGuards  = new ArrayList<FSMGuardImpl>();
    this.blocksToUpdate = new ArrayList<Block>();
}

public String toString(){
    return "␣"+this.getName()+"␣:␣fireableGuards="+this.fireableGuards+"␣blocksToUpdate="+this.blocksToUpdate;
}

/*=========================================================*/
/* Méthodes d'exécution pour les MoCs au niveau racine
/*=========================================================*/

public void setup(Model m) {
    System.out. println (this. toString() + "␣:␣setup");

    // Initialisation de l'état courant du modèle avec l'etat initial du modèle
    ((FSMModel)m).setCurrentFireableGuards(((FSMModel)m).getInitialFireableGuards());

    //Setup pour tous les blocs du modèle
    for (Block b : m.getStructure().getBlocks()) {
        b.setup();
    }

    ModhelxViewer.refresh();
}

@SuppressWarnings("unchecked")
public void startOfSnapshot(Model m) {
    System.out. println (this. toString() + "␣:␣sos");

    //StartOfSnapshot pour tous les blocs
    for (Block b : m.getStructure().getBlocks()) {
        b.startOfSnapshot();
    }

    // Initialisation de la liste initiale des gardes tirables avec l'état courant du modèle
    this. initialFireableGuards  = (ArrayList<FSMGuardImpl>) (((FSMModel)m).getCurrentFireableGuards()).clone();

    ModhelxViewer.refresh();
}

@SuppressWarnings("unchecked")
public void reset(Model m) throws ModHelXExecutionException {
    System.out. println (this. toString() + "␣:␣reset");

    //Reset pour tous les points d'interface de sortie du modèle
    for (PinImpl p : m.getStructure().getPinsOut()) {
        p. reset();
    }

    //Reset pour tous les blocs
    for (Block b : m.getStructure().getBlocks()) {
        b. reset();
    }

    // Initialisation de la liste des gardes tirables courante et de la liste des blocs à mettre à jour
```

```
        this . fireableGuards  = (ArrayList<FSMGuardImpl>) this.initialFireableGuards.clone();
        this .blocksToUpdate = (ArrayList<FSMGuardImpl>) this.initialFireableGuards.clone();

        ModhelxViewer.refresh();
    }

    public void initSchedule(Model m) {
        System.out. println ( this . toString() + "␣:␣initsched␣(empty)");
        //Méthode sans objet pour FSM
        ModhelxViewer.refresh();
    }

    public void prePropagate(Model m) {
        System.out. println ( this . toString() + "␣:␣preprop␣(empty)");
        //Méthode sans objet pour FSM
        ModhelxViewer.refresh();
    }

    public void preSchedule(Model m) throws ModHelXExecutionException {
        System.out. println ( this . toString() + "␣:␣presched");

        //Choix du bloc à mettre à jour (premier de la liste des blocs à mettre à jour)
        if (! this .blocksToUpdate.isEmpty())
            this .currentBlock = this .blocksToUpdate.get(0);
        else
            throw new ModHelXExecutionException("FSMMoCImpl␣:␣preSchedule␣:␣pas␣de␣bloc␣à␣mettre␣à␣jour.");

        ModhelxViewer.refresh();
    }

    public void update(Model m) {
        System.out. println ( this . toString() + "␣:␣update");

        //Mise à jour du bloc courant
        this .currentBlock.update();

        ModhelxViewer.refresh();
    }

    public void interSchedule(Model m) {
        System.out. println ( this . toString() + "␣:␣intersched");

        if ( this .currentBlock instanceof FSMGuardImpl){
            //Si le bloc que l'on vient de mettre à jour est une garde
            if (((FSMGuard)this.currentBlock).getEnabledAction().getStoredTokens(). size ()>0){
                //Si la garde s'est évaluée à vrai (un jeton a été produit)
                //Le jeton produit par la garde est consommé
                ((FSMGuard)this.currentBlock).getEnabledAction().getStoredTokens(). clear ();

                //L'état de l'automate (= l'ensemble des gardes tirables courantes) change
                this . fireableGuards . clear ();
                for( Relation r :(( FSMGuard)this.currentBlock).getTargetState() .getOutcomingRelations()){
                    //Et est composé de l'ensemble des gardes cibles de la garde courante
                    this . fireableGuards .add(((FSMGuardImpl)r.getTarget().getIsInputForBlock()));
                }

                //L'ensemble des blocs à mettre à jour ensuite change
                this .blocksToUpdate.clear();
                ArrayList<BlockImpl> list = new ArrayList<BlockImpl>();
                for( Relation r :(( FSMGuard)this.currentBlock).getEnabledAction().getOutcomingRelations()){
                    //Et est composé de l'ensemble des actions déclenchées par cette garde
                    list .add(r.getTarget() . getIsInputForBlock ());

                    //Un jeton est déposé sur chacune de ces actions afin qu'elles sachent qu'elles ont été déclenchées
                    ArrayList <FSMEventImpl> enabletokens = new ArrayList<FSMEventImpl>();
                    enabletokens.add(new FSMEventImpl());
```

```
                r.getTarget().setStoredTokens(enabletokens);
            }
            this.blocksToUpdate = list;
        }else{ //Si la garde s'est évaluée à faux (aucun jeton produit)
            //Alors elle est retirée de la liste des blocs à mettre à jour
            this.blocksToUpdate.remove(this.currentBlock);
        }
    }else{ //Si le bloc que l'on vient de mettre à jour est une action
        //Alors elle est retirée de la liste des blocs à mettre à jour
        this.blocksToUpdate.remove(this.currentBlock);
    }

    ModhelxViewer.refresh();
}

public void postPropagate(Model m) {
    System.out.println(this.toString() + "␣:␣postprop␣(empty)");
    //Méthode sans objet pour FSM
    ModhelxViewer.refresh();
}

public void postSchedule(Model m) {
    System.out.println(this.toString() + "␣:␣postsched␣(empty)");
    //Méthode sans objet pour FSM
    ModhelxViewer.refresh();
}

public boolean snapDet(Model m) {
    System.out.println(this.toString() + "␣:␣snapdet");

    //Le snapshot est déterminé s'il n'y a plus de blocs à jour
    // et qu'un nouvel état a été déterminé pour l'automate
    boolean snapdet = this.blocksToUpdate.isEmpty() && !this.fireableGuards.isEmpty();

    ModhelxViewer.refresh();
    return snapdet;
}

public boolean validate(Model m) {
    System.out.println(this.toString() + "␣:␣validate");

    //Validate pour tous les blocs
    boolean b = true;
    for (Block block : m.getStructure().getBlocks()) {
        if (!block.validate())
            b = false;
    }

    ModhelxViewer.refresh();
    return b;
}

@SuppressWarnings("unchecked")
public void endOfSnapshot(Model m) {
    System.out.println(this.toString() + "␣:␣eos");

    //EndOfSnapshot pour tous les blocs
    for (Block b : m.getStructure().getBlocks()) {
        b.endOfSnapshot();
    }

    //Mise à jour de l'état du modèle avec le nouvel état calculé dans le snapshot
    ((FSMModel)m).setCurrentFireableGuards((ArrayList<FSMGuardImpl>) this.fireableGuards.clone());

    ModhelxViewer.refresh();
}
```

```
    public void wrapup(Model m) {
        System.out.println(this.toString() + "␣:␣wrapup");

        //Wrapup pour tous les blocs
        for (Block b : m.getStructure().getBlocks()) {
            b.wrapup();
        }

        ModhelxViewer.refresh();
    }

    /*=========================================================*/
    /* Methodes pour les mocs internes
    /*=========================================================*/

    public void startOfUpdate(Model m) {
        System.out.println(this.toString() + "␣:␣startOfUpdate");
        //Méthode sans objet pour FSM
        ModhelxViewer.refresh();
    }

    public void endOfUpdate(Model m) {
        System.out.println(this.toString() + "␣:␣endOfUpdate");
        //Méthode sans objet pour FSM
        ModhelxViewer.refresh();
    }

    public boolean further(Model m) {
        System.out.println(this.toString() + "␣:␣further");
        ModhelxViewer.refresh();

        //Méthode équivalente à !snapDet pour FSM
        return !this.snapDet(m);
    }
}
```

C.2 Implémentation du modèle de calcul DE

C.2.1 Méta-modèle spécifique pour DE

La figure C.2 montre le méta-modèle spécifique du modèle de calcul DE. Les différents éléments de ce méta-modèle sont présentés dans la section 5.3.2.

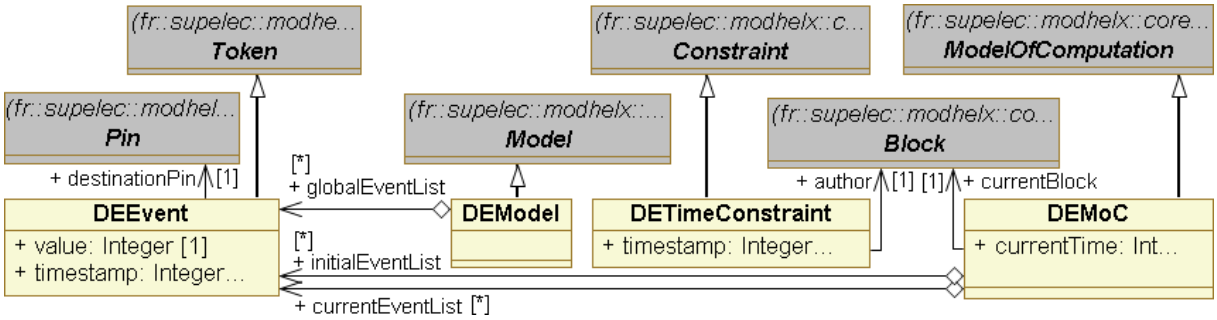


FIG. C.2 – Méta-modèle spécifique pour DE

C.2.2 Sémantique d'exécution spécifique pour DE : code Java

Les principes de l'exécution de modèles DE sont exposés dans la section 5.3.2. Le listing C.2 suivant présente le code complet du modèle de calcul DE en Java.

Listing C.2 – Classe DEMoCImpl

```
/**
 *
 * DEMoCImpl
 *
 * @author Cecile Hardebolle {cecile.hardebolle@supelec.fr}
 */
public class DEMoCImpl extends ModelOfComputationImpl implements DEMoC {

    //Liste initiale des événements à traiter
    protected ArrayList<DEEventImpl> initialEventList;
    //Liste courante des événements à traiter
    protected ArrayList<DEEventImpl> currentEventList;
    //Bloc courant à mettre à jour
    protected Block currentBlock;
    //Date courante
    protected int currentTime;

    public DEMoCImpl() {
        super("DE");
        this.initialEventList = new ArrayList<DEEventImpl>();
        this.currentEventList = new ArrayList<DEEventImpl>();
        this.constraints = new ArrayList<DETimeConstraintImpl>();
        this.currentTime = 0;
    }

    public String toString(){
        return "␣"+this.getName()+"␣:␣currentTime="+this.currentTime+"␣currentEventList="+this.currentEventList
        +"␣constraints="+this.constraints;
    }

    public int getCurrentTime() {
        return this.currentTime;
    }

    /*==========================================================*/
    /* Méthodes d'exécution pour les MoCs au niveau racine
```

```
/*=========================================================*/

public void setup(Model m) {
    ModhelxViewer.refresh();

    System.out.println(this.toString() + "␣:␣setup");

    //Setup pour tous les blocs du modèle
    for (Block b : m.getStructure().getBlocks()) {
        b.setup();
    }

    ModhelxViewer.refresh();
}

@SuppressWarnings("unchecked")
public void startOfSnapshot(Model m) {
    System.out.println(this.toString() + "␣:␣sos");

    //StartOfSnapshot pour tous les blocs
    for (Block b : m.getStructure().getBlocks()) {
        b.startOfSnapshot();
    }

    //Prise en compte des événements présents sur les entrées du modèle
    // (ceux-ci sont ajouté à la liste des événements courants)
    for (Pin pInModel : m.getStructure().getPinsIn()) {
        //Chaque entrée du modèle est un alias d'une entrée d'un bloc
        //Ce sont sur les entrées des blocs que sont stockés les événements à récupérer
        Pin aliasin = pInModel.getAliasOf();
        if (aliasin != null){
            if (pInModel.getStoredTokens()!= null && !pInModel.getStoredTokens().isEmpty()) {
                Collection<DEEventImpl> evtsInModel = (Collection<DEEventImpl>) pInModel.getStoredTokens();
                for (DEEvent evt : evtsInModel) {
                    //Pour chaque événement, on réalise un clone
                    // (afin que l'état du modèle soit préservé en cas de non validation du snapshot)
                    DEEventImpl clone = DEFactoryImpl.getInstance().createDEEvent();
                    clone.setValue(evt.getValue());
                    clone.setTimestamp(evt.getTimestamp());
                    clone.setDestinationPin(aliasin);

                    //Et on ajoute l'événement ainsi cloné à la liste des événements courants du modèle
                    ((DEModel)m).getGlobalEventList().add(clone);
                }
            }
        }
    }

    ModhelxViewer.refresh();
}

@SuppressWarnings("unchecked")
public void reset(Model m) throws ModHelXExecutionException {
    System.out.println(this.toString() + "␣:␣reset");

    //Reset des sorties du modèle
    for (Pin b : m.getStructure().getPinsOut()) {
        b.reset();
    }

    //Reset pour tous les blocs du modèle
    for (Block b : m.getStructure().getBlocks()) {
        b.reset();
    }

    //On calcule la date courante à partir de la contrainte de date minimum et de l'événement de date minimum
```

```
        if (! this . constraints .isEmpty() && !this. currentEventList .isEmpty()){
            int minconsttime = DETimeConstraintImpl.getMinTime((Collection<DETimeConstraintImpl>) this.constraints);
            int minevttime = DEEventImpl.getMinTime(this.currentEventList);
            this .currentTime = (minconsttime<minevttime?minconsttime:minevttime);
        }else if ( this . constraints .isEmpty()){
            this .currentTime = DEEventImpl.getMinTime(this.currentEventList);
        }else if ( this . currentEventList .isEmpty()){
            this .currentTime = DETimeConstraintImpl.getMinTime((Collection<DETimeConstraintImpl>) this.constraints);
        }else{
            throw new ModHelXExecutionException("DEMoCImpl␣:␣reset␣:␣liste␣des␣contraintes␣et␣liste␣des␣evenements␣
              vides.");
        }

        // Initialisation de la liste initiale des événements à traiter au temps courant à partir de celle du modèle
        this . initialEventList = DEEventImpl.select(((DEModelImpl)m).globalEventList,this.currentTime);

        // Initialisation de la liste courante avec la liste initiale
        this . currentEventList = (ArrayList<DEEventImpl>) this. initialEventList . clone ();

        ModhelxViewer.refresh();
    }

    @SuppressWarnings("unchecked")
    public void initSchedule (Model m) throws ModHelXExecutionException {
        System.out. println ( this . toString () + "␣:␣initsched ");

        //On va lister les blocs qu'il faut mettre à jour dans le snapshot courant
        ArrayList<BlockImpl> blocklist = new ArrayList<BlockImpl>();

        //Ces blocs sont soit des destinataires d'événements à traiter au temps courant,
        for(DEEvent evt:this . currentEventList ){
            blocklist .add(evt. getDestinationPin () . getIsInputForBlock ());
        }

        //Soit des blocs ayant emis une contrainte au temps courant
        ArrayList<DETimeConstraintImpl> curconst =
          DETimeConstraintImpl.select((Collection<DETimeConstraintImpl>) this.constraints,this.currentTime);
        for(DETimeConstraint constr:curconst){
            blocklist .add(constr.getAuthor());
        }

        //Si la liste ainsi obtenue n'est pas vide
        if (! blocklist .isEmpty()) {
            //Alors le premier bloc à mettre à jour est le premier de la liste selon l'ordre topologique
            this .currentBlock = (Block) Collections .min( blocklist , new BlockComparator());

            //S'il s'agit d'un bloc ayant émis une contrainte, il faut retirer celle-ci de la liste des contraintes
            DETimeConstraint constraint = DETimeConstraintImpl.selectOne((Collection<DETimeConstraintImpl>)
              this.constraints,this.currentTime, this . currentBlock);
            if ( constraint != null ) this . constraints .remove(constraint);
        }else{
            throw new ModHelXExecutionException("DEMoCImpl␣:␣initschedule␣:␣liste␣de␣bloc␣vide.");
        }

        ModhelxViewer.refresh();
    }

    public void prePropagate(Model m) {
        System.out. println ( this . toString () + "␣:␣preprop");

        //On va chercher dans la file des événements les événements qui sont destinés au block courant
        // et qui portent la date courante
        for (Pin currCompPinIN : this.currentBlock. getPinsIn ()) {
            ArrayList<DEEventImpl> evttoprocess =
              DEEventImpl.select(this.currentEventList, this .currentTime,currCompPinIN);
```

```
            //On dépose ces événements sur les entrées du bloc
            currCompPinIN.setStoredTokens(evttoprocess);

            //On les enlève de la liste des événements à traiter
            this.currentEventList.removeAll(evttoprocess);
        }

        ModhelxViewer.refresh();
    }

    public void preSchedule(Model m) {
        System.out.println(this.toString() + " : presched (empty)");
        //Méthode sans objet pour DE
        ModhelxViewer.refresh();
    }

    public void update(Model m) {
        System.out.println(this.toString() + " : update");

        //Mise à jour du bloc courant
        this.currentBlock.update();

        ModhelxViewer.refresh();
    }

    public void interSchedule(Model m) {
        System.out.println(this.toString() + " : intersched (empty)");
        //Méthode sans objet pour DE
        ModhelxViewer.refresh();
    }

    public void postPropagate(Model m) {
        System.out.println(this.toString() + " : postprop");

        //Les événements produits sur chacuns des points d'interface du bloc mis à jour
        // sont déplacés dans la liste des événements à traiter,
        for (Pin currentCompPin : currentBlock.getPinsOut()) {
            if(currentCompPin.getStoredTokens()!= null){
                //Seulement si les points d'interface du bloc ne sont pas aliasés par des sorties du modèle
                Pin aliasedBy = currentCompPin.getAliasedBy();
                if(aliasedBy==null || (aliasedBy!=null && !aliasedBy.getIsOutputForBlock().equals(m.getStructure()))){
                        this.moveEventsToList(currentCompPin,this.currentEventList);
                }
            }
        }

        ModhelxViewer.refresh();
    }

    /**
     * Méthode privée pour déplacer les événements présents sur les sorties d'un bloc
     * dans une liste d'événements
     */
    @SuppressWarnings("unchecked")
    private void moveEventsToList(Pin psource, Collection<DEEventImpl> listtarget){
        if(psource.getStoredTokens()!=null){
            ArrayList<DEEventImpl> setOfProducedEvts = (ArrayList<DEEventImpl>) psource.getStoredTokens();
            //Les événements produits doivent être transmis à tous les blocs connectés
            // au point d'interface sur lequel ils ont été émis
            for (DEEventImpl producedEvent : setOfProducedEvts) {
                //Donc, pour chaque événement produit
                for (Relation r : psource.getOutcomingRelations()) {
                    //Et pour chaque relation partant de ce point d'interface, on crée une copie
                    //Le destinataire de la copie est le point d'interface cible de la relaction
                    DEEventImpl evt = new DEEventImpl(producedEvent.getValue(), producedEvent.getTimestamp(),
                      r.getTarget());
```

```
                //Chaque copie est ajoutée à la liste fournie
                listtarget.add(evt);
            }
            //Remarque : si aucune relation ne part du point d'interface, l'événement n'est pas déplacé dans la
               liste
            // car il n'est alors pas à traiter (il n'est destiné à aucun bloc)
        }
    }
    //Les événements initiaux sont supprimés du point d'interface
    psource.getStoredTokens().clear();
}

public void postSchedule(Model m) {
    System.out.println(this.toString() + "␣:␣postsched␣(empty)");
    //Méthode sans objet pour DE
    ModhelxViewer.refresh();
}

@SuppressWarnings("unchecked")
public boolean snapDet(Model m) {
    System.out.println(this.toString() + "␣:␣snapdet");

    //Le snapshot est déterminé si :
    // - tous les événements au temps courant ont été traités
    boolean allevtstreated = false;
    ArrayList<DEEventImpl> remainingevents = DEEventImpl.select(this.currentEventList,this.currentTime);
     allevtstreated = remainingevents.size()==0;

    // - toutes les contraintes au temps courant ont été traités
    boolean allconsttreated = false;
    ArrayList<DETimeConstraintImpl> remainingcontraints =
      DETimeConstraintImpl.select((Collection<DETimeConstraintImpl>) this.constraints, this.currentTime);
     allconsttreated = remainingcontraints.size()==0;

    ModhelxViewer.refresh();
    return allevtstreated && allconsttreated;
}

public boolean validate(Model m) {
    System.out.println(this.toString() + "␣:␣validate");

    //Validate pour tous les blocs
    boolean b = true;
    for (Block block : m.getStructure().getBlocks()) {
        if (!block.validate())
            b = false;
    }

    ModhelxViewer.refresh();
    return b;
}

@SuppressWarnings("unchecked")
public void endOfSnapshot(Model m) throws ModHelXExecutionException {
    System.out.println(this.toString() + "␣:␣eos");

    //EndOfSnapshot pour tous les blocks
    for (Block b : m.getStructure().getBlocks()) {
        b.endOfSnapshot();
    }

    //On met à jour la liste des événements du modèle :
    // - en enlevant les événements traités pendant le snapshot
    ((DEModelImpl)m).globalEventList.removeAll(this. initialEventList );
```

```
        // - et en ajout les événements produits pendant le snapshot
        ((DEModelImpl)m).globalEventList.addAll(this.currentEventList);

        //On verifie qu'il ne reste pas de contrainte pour la date courante
        if (!this.constraints.isEmpty()){
            int minconsttime = DETimeConstraintImpl.getMinTime((Collection<DETimeConstraintImpl>) this.constraints);
            if (minconsttime == this.currentTime) throw new ModHelXExecutionException("DEMoCImpl : eos : il reste
              une contrainte a la date courante.");
        }

        ModhelxViewer.refresh();
    }

    public void wrapup(Model m) {
        System.out.println(this.toString() + " : wrapup");

        //Wrapup pour tous les blocks
        for (Block b : m.getStructure().getBlocks()) {
            b.wrapup();
        }
        ModhelxViewer.refresh();
    }

    /*=========================================================*/
    /* Methodes pour les mocs internes                         */
    /*=========================================================*/

    public void startOfUpdate(Model m) {
        System.out.println(this.toString() + " : startOfUpdate");

        //Les événements présents sur les entrées du modèle doivent être pris en compte
        //(donc ajoutés à la liste des événements courants à traiter )
        for (Pin pin : m.getStructure().getPinsIn()){
            if (pin.getStoredTokens()!=null && !pin.getStoredTokens().isEmpty()){
                this.moveEventsToList(pin,this.currentEventList);
            }
        }

        ModhelxViewer.refresh();
    }

    public void endOfUpdate(Model m) {
        System.out.println(this.toString() + " : endOfUpdate");
        //Méthode sans objet pour DE
        ModhelxViewer.refresh();
    }

    public boolean further(Model m) {
        System.out.println(this.toString() + " : further");
        ModhelxViewer.refresh();

        //Méthode équivalente à !snapDet pour DE
        return this.snapDet(m);
    }
}
```

C.3 Bibliothèque de blocs

C.3.1 Bloc atomique `DEUser`

Le listing C.3 suivant présente le code Java du bloc atomique représentant l'utilisateur dans le cas d'étude présenté dans la section 5.3. Le comportement de cet utilisateur est spécifié dans la section 5.3.2.3.

Listing C.3 – Classe DEUserImpl

```
/**
 *
 * DEUserImpl
 *
 * @author Cecile Hardebolle {cecile.hardebolle@supelec.fr}
 */
public class DEUserImpl extends AtomicBlockImpl{

    //Délai entre l'émission d'une pièce et l'appui sur le bouton café
    protected int deltaCoin;
    //Délai entre la réception d'un café et l'émission d'une nouvelle pièce
    protected int deltaUser;

    //Variables internes
    private boolean coinDone; //Pièce émise
    private boolean coffeeServed; //Café reçu

    protected DEUserImpl() {
        super("User");
        this.coinDone = false; //Au début, pas de pièce émise
        this.coffeeServed = false; //Au début, pas de café reçu

        this.deltaCoin = 1; //Sauf changement, délai de 1 entre émission d'une pièce et appui sur le bouton café
        this.deltaUser = 1; //Sauf changement, délai de 1 entre réception d'un café et émission d'une nouvelle pièce

        //Création d'un point d'interface d'entrée pour la réception des cafés
        ArrayList<PinImpl> pinsin = new ArrayList<PinImpl>();
        PinImpl pin = CoreFactoryImpl.getInstance().createPin();
        pin.setName("served");
        pin.setIsInputForBlock(this);
        pinsin.add(pin);
        this.setPinsIn(pinsin);

        //Création des points d'interface de sortie
        ArrayList<PinImpl> pinsout = new ArrayList<PinImpl>();
        //Pour l'émission des pièces
        PinImpl pin1 = CoreFactoryImpl.getInstance().createPin();
        pin1.setName("coin");
        pin1.setIsOutputForBlock(this);
        pinsout.add(pin1);
        //Pour l'appui sur le bouton café
        PinImpl pin2 = CoreFactoryImpl.getInstance().createPin();
        pin2.setName("coffee");
        pin2.setIsOutputForBlock(this);
        pinsout.add(pin2);
        this.setPinsOut(pinsout);
    }

    @SuppressWarnings("unchecked")
    public void setup() {
        System.out.println(this.getName()+" : setup");

        //Contrainte initiale pour la première mise à jour
        DETimeConstraintImpl constinit = DEFactoryImpl.getInstance().createDETimeConstraint();
        constinit.setConstraintTime(((DEMoC)this.getUsingMoC()).getCurrentTime()); //A la date courante
        constinit.setAuthor(this);
```

```
        ArrayList<DETimeConstraintImpl> consts = (ArrayList<DETimeConstraintImpl>)
          this.getUsingMoC().getConstraints();
        consts.add( constinit );
        this.getUsingMoC().setConstraints(consts);
    }

    public void startOfSnapshot() {
        System.out.println(this.getName()+" : sos (empty)");
    }

    public void reset() {
        System.out.println(this.getName()+" : reset");
        for(Pin p : this.getPinsIn()){ // Réinitialisation des entrées
            p.reset();
        }
        for(Pin p : this.getPinsOut()){ // Réinitialisation des sorties
            p.reset();
        }
    }

    public void update() {
        System.out.println(this.getName()+" : update");

        Pin servedPin = PinImpl.selectOnePin(this.getPinsIn(),"served");
        if(servedPin.getStoredTokens().isEmpty()){ //Si le café n'a pas été reçu
            ArrayList<DEEventImpl> se = new ArrayList<DEEventImpl>();
            if(!this.coinDone){ //Et si la pièce n'a pas encore été émise
                Pin coinPin = PinImpl.selectOnePin(this.getPinsOut(),"coin");

                //Alors on émet la pièce
                DEEventImpl coin = DEFactoryImpl.getInstance().createDEEvent();
                coin.setValue(0);
                coin.setTimestamp(((DEMoC)this.getUsingMoC()).getCurrentTime()); //A la date courante
                se.add(coin);
                coinPin.setStoredTokens(se);

                this.coinDone = true; //La pièce a été émise

            }else if(this.coinDone){ //Et si la pièce a déjà été émise
                Pin coffeePin = PinImpl.selectOnePin(this.getPinsOut(),"coffee");

                //Alors on appuie sur le bouton café
                DEEventImpl coffee = DEFactoryImpl.getInstance().createDEEvent();
                coffee.setValue(0);
                coffee.setTimestamp(((DEMoC)this.getUsingMoC()).getCurrentTime()); //A la date courante
                se.add(coffee);
                coffeePin.setStoredTokens(se);

                this.coinDone = false; //La pièce émise va être consommée
            }
        }else{ //Sinon si le café vient d'être reçu
            this.coffeeServed = true;
            servedPin.getStoredTokens().clear(); //On consomme le café
        }

    }

    public boolean validate() {
        System.out.println(this.getName()+" : validate");
        return true; //On valide toujours le snapshot
    }

    @SuppressWarnings("unchecked")
    public void endOfSnapshot() {
        System.out.println(this.getName()+" : eos");
```

```
        if (this.coinDone){ //Si la pièce vient d'être émise
            //Alors on produit une contrainte pour pouvoir appuyer sur le bouton café
            DETimeConstraintImpl constsuiv = DEFactoryImpl.getInstance().createDETimeConstraint();
            constsuiv.setConstraintTime(((DEMoC)this.getUsingMoC()).getCurrentTime()+this.deltaCoin); //A la date
              courante + deltaCoin
            constsuiv.setAuthor(this);
            ArrayList<DETimeConstraintImpl> consts = (ArrayList<DETimeConstraintImpl>)
              this.getUsingMoC().getConstraints();
            consts.add(constsuiv);
            this.getUsingMoC().setConstraints(consts);

        }else if (this.coffeeServed){ //Si le café vient d'être servi
            //Alors on produit une contrainte pour demander un nouveau café (émettre une nouvelle pièce)
            DETimeConstraintImpl constsuiv = DEFactoryImpl.getInstance().createDETimeConstraint();
            constsuiv.setConstraintTime(((DEMoC)this.getUsingMoC()).getCurrentTime()+this.deltaUser); //A la date
              courante + deltaUser
            constsuiv.setAuthor(this);
            ArrayList<DETimeConstraintImpl> consts = (ArrayList<DETimeConstraintImpl>)
              this.getUsingMoC().getConstraints();
            consts.add(constsuiv);
            this.getUsingMoC().setConstraints(consts);
            this.coffeeServed = false;
        }
    }

    public void wrapup() {
        System.out.println(this.getName()+"␣:␣wrapup␣(empty)");
    }

    public int getDeltaCoin() {
        return this.deltaCoin;
    }

    public void setDeltaCoin(int deltaCoin) {
        this.deltaCoin = deltaCoin;
    }

    public int getDeltaUser() {
        return this.deltaUser;
    }

    public void setDeltaUser(int deltaUser) {
        this.deltaUser = deltaUser;
    }

}
```

C.3.2 Bloc atomique `FSMSimpleGuard`

Le listing C.4 suivant présente le code Java du bloc atomique représentant une garde simple pour le modèle de calcul FSM. Ce type de garde est mis en œuvre dans le cas d'étude présenté dans la section 5.3 et son comportement est décrit dans la section 5.3.3.2.

Listing C.4 – Classe FSMSimpleGuardImpl

```
/**
 * FSMSimpleGuardImpl
 * Une FSMSimpleGuard est une garde qui réagit à la présence d'un événement
 * dont le nom est specifié par triggerEvtName
 *
 * @author Cecile Hardebolle {cecile.hardebolle@supelec.fr}
 */
public class FSMSimpleGuardImpl extends FSMGuardImpl implements AtomicBlock {

    //Nom de l'événement déclencheur de la garde
    protected String triggerEvtName;

    protected FSMSimpleGuardImpl(){
        super("FSMSimpleGuard");

        this.triggerEvtName = "trigger"; //Par défaut, déclenche sur l'événement "trigger"

        //Création d'un point d'interface d'entrée pour la réception de l'événement déclencheur
        ArrayList<PinImpl> pinsin = new ArrayList<PinImpl>();
        pinsin.addAll(this.getPinsIn());
        PinImpl datapin = CoreFactoryImpl.getInstance().createPin();
        datapin.setName(this.triggerEvtName);
        datapin.setIsInputForBlock(this);
        pinsin.add(datapin);

        this.setPinsIn(pinsin);
    }

    public void setup() {
        System.out.println(this.getName()+" : setup (empty)");
    }

    public void startOfSnapshot() {
        System.out.println(this.getName()+" : sos (empty)");
    }

    public void reset() {
        System.out.println(this.getName()+" : reset");
        for(PinImpl p : this.getPinsOut()){ // Réinitialisation des sorties
            p.reset();
        }
    }

    public void update() {
        System.out.println(this.getName()+" : update");

        PinImpl pevtIn = CoreFactoryImpl.getInstance().select(this.getPinsIn(), this.triggerEvtName);
        //Si un événement a été reçu
        if(pevtIn.getStoredTokens().size()==1){
            //Et qu'il s'agit de l'événement déclencheur
            if(((FSMEventImpl)pevtIn.getStoredTokens().get(0)).getName().equals(this.triggerEvtName)){
                //Alors on consomme l'événement
                pevtIn.clearStoredTokensAndAlias();

                //Et on produit un événement déclencheur pour l'action associée
                FSMEventImpl evt = FSMFactoryImpl.getInstance().createFSMEvent();
                evt.setName("enable");
                ArrayList<FSMEventImpl> evts = new ArrayList<FSMEventImpl>();
```

```
                evts.add(evt);
                this.getEnabledAction().setStoredTokens(evts);
            }
        }

    }

    public boolean validate() {
        System.out.println(this.getName()+" : validate");
        return true; //On valide toujours le snapshot
    }

    public void endOfSnapshot() {
        System.out.println(this.getName()+" : eos (empty)");
    }

    public void wrapup() {
        System.out.println(this.getName()+" : wrapup (empty)");
    }

    public String getTriggerEvtName() {
        return this.triggerEvtName;
    }

    public void setTriggerEvtName(String inEvtName) {
        PinImpl ptrigger = CoreFactoryImpl.getInstance().select(this.getPinsIn(), this.triggerEvtName);
        if(ptrigger!=null){
            ptrigger.setName(inEvtName);
            this.triggerEvtName = inEvtName;
        }else{
            System.out.println("Probleme de modification du nom de l'événement déclencheur pour
              FSMSimpleGuardImpl");
        }
    }
}
```

C.3.3 Bloc atomique `FSMSimpleAction`

Le listing C.5 suivant présente le code Java du bloc atomique représentant une action simple pour le modèle de calcul FSM. Ce type d'action est mis en œuvre dans le cas d'étude présenté dans la section 5.3 et son comportement est décrit dans la section 5.3.3.2.

Listing C.5 – Classe FSMSimpleActionImpl

```
/**
 * FSMSimpleActionImpl
 * Une FSMSimpleAction est une action qui produit un événement dont le nom
 * est specifié par producedEvtName lorsqu'elle est activée
 *
 * @author Cecile Hardebolle {cecile.hardebolle@supelec.fr}
 */
public class FSMSimpleActionImpl extends FSMActionImpl implements AtomicBlock {

    //Nom de l'événement produit par l'action
    protected String producedEvtName;

    protected FSMSimpleActionImpl(){
        super("FSMSimpleAction");

        this.producedEvtName = "event"; //Par défaut, produit un événement "event"

        //Création d'un point d'interface de sortie pour la production de l'événement
        ArrayList<PinImpl> pinsout = new ArrayList<PinImpl>();
        pinsout.addAll(this.getPinsOut());
        PinImpl datapin = CoreFactoryImpl.getInstance().createPin();
        datapin.setName(this.producedEvtName);
        datapin.setIsOutputForBlock(this);
        pinsout.add(datapin);

        this.setPinsOut(pinsout);
    }

    public void setup() {
        System.out.println(this.getName()+" : setup (empty)");
    }

    public void startOfSnapshot() {
        System.out.println(this.getName()+" : sos (empty)");
    }

    public void reset() {
        System.out.println(this.getName()+" : reset");
        for(PinImpl p : this.getPinsOut()){ // Réinitialisation des sorties
            p.reset();
        }
    }

    public void update() {
        System.out.println(this.getName()+" : update");

        //Si l'action a été déclenchée
        if(this.getEnable().getStoredTokens().size()>0){
            //Alors on consomme le jeton de déclenchement
            this.getEnable().getStoredTokens().clear();

            //Et on produit l'événement en sortie
            PinImpl datapin = CoreFactoryImpl.getInstance().select(this.getPinsOut(), this.producedEvtName);
            if(datapin!=null){
                FSMEventImpl evtout = FSMFactoryImpl.getInstance().createFSMEvent();
                evtout.setName(this.producedEvtName);
                ArrayList<TokenImpl> evtsout = new ArrayList<TokenImpl>();
                evtsout.add(evtout);
```

```
                datapin.setStoredTokens(evtsout);
            }
        }
    }

    public boolean validate() {
        System.out.println(this.getName()+" : validate");
        return true; //On valide toujours le snapshot
    }

    public void endOfSnapshot() {
        System.out.println(this.getName()+" : eos (empty)");
    }

    public void wrapup() {
        System.out.println(this.getName()+" : wrapup (empty)");
    }

    public String getOutEvtName() {
        return this.producedEvtName;
    }

    public void setOutEvtName(String outEvtName) {
        PinImpl datapin = CoreFactoryImpl.getInstance().select(this.getPinsOut(), this.producedEvtName);
        if(datapin!=null){
            datapin.setName(outEvtName);
            this.producedEvtName = outEvtName;
        }else{
            System.out.println("Probleme de modification du nom de l'événement produit pour FSMSimpleActionImpl");
        }
    }
}
```

C.3.4 Bloc d'interface `CoffeeMachineDEFSM`

Le listing C.6 suivant présente le code Java du bloc d'interface représentant la machine à café dans le cas d'étude présenté dans la section 5.3. Ce bloc d'interface permet d'encapsuler un modèle FSM afin de l'inclure dans un modèle DE. Son comportement est décrit dans la section 5.3.4.

Listing C.6 – Classe CoffeeMachineDEFSMImpl

```
/**
 *
 * CoffeeMachineDEFSMImpl
 *
 * @author Cecile Hardebolle {cecile.hardebolle@supelec.fr}
 */
public class CoffeeMachineDEFSMImpl extends InterfaceBlockImpl {

    //Variable permettant de stocker la date du dernier événement "coffee" reçu
    private int tLastDEevt;

    protected CoffeeMachineDEFSMImpl(){
        super();

        this.tLastDEevt = 0;

        //Création du paramètre "preparationTime" représentant le temps de préparation du café
        ParameterImpl preparationTime = CoreFactoryImpl.getInstance().createParameter();
        preparationTime.setName("preparationTime");
        preparationTime.setValue("0");
        this.getParameters().put("preparationTime", preparationTime);


        //Création des points d'interface d'entrée
        ArrayList<PinImpl> pins = new ArrayList<PinImpl>();
        //Pour recevoir la pièce
        PinImpl pin1 = CoreFactoryImpl.getInstance().createPin();
        pin1.setName("coin");
        pin1.setIsInputForBlock(this);
        pins.add(pin1);
        //Pour recevoir l'appui sur le bouton café
        PinImpl pin2 = CoreFactoryImpl.getInstance().createPin();
        pin2.setName("coffee");
        pin2.setIsInputForBlock(this);
        pins.add(pin2);
        this.setPinsIn(pins);

        //Création d'un point d'interface de sortie pour émettre le café
        pins = new ArrayList<PinImpl>();
        PinImpl pin = CoreFactoryImpl.getInstance().createPin();
        pin.setName("served");
        pin.setIsOutputForBlock(this);
        pins.add(pin);
        this.setPinsOut(pins);
    }

    public void adaptIn(){
        System.out.println(this.getName()+" : adaptIn");

        Pin extcoinpin = PinImpl.selectOnePin(this.getPinsIn(), "coin");
        Pin extcoffeepin = PinImpl.selectOnePin(this.getPinsIn(), "coffee");

        if(extcoinpin.getStoredTokens().size()==1){ //Si la pièce a été reçue
            Pin intcoinpin = PinImpl.selectOnePin(this.getInternalModel().getStructure().getPinsIn(),"coin");
            ArrayList<FSMEventImpl> intevts = new ArrayList<FSMEventImpl>();
            FSMEventImpl intevt = FSMFactoryImpl.getInstance().createFSMEvent(); //Un événement FSM équivalent
              (mais sans date) est créé
```

```
            intevt.setName("coin");
            intevts.add(intevt);
            intcoinpin.setStoredTokens(intevts); //Et déposé sur le point d'interface d'entrée du modèle FSM

            extcoinpin.getStoredTokens().clear(); //Et la pièce est consommée

        }else if(extcoffeepin.getStoredTokens().size()==1){ //Sinon si le bouton café a été appuyé
            DEEvent coffeeevt = (DEEvent) extcoffeepin.getStoredTokens().get(0);
            this.tLastDEevt = coffeeevt.getTimestamp(); //La date de cet événement est stockée

            Pin intcoffeepin = PinImpl.selectOnePin(this.getInternalModel().getStructure().getPinsIn(),"coffee");
            ArrayList<FSMEventImpl> intevts = new ArrayList<FSMEventImpl>();
            FSMEventImpl intevt = FSMFactoryImpl.getInstance().createFSMEvent(); //Un événement FSM équivalent
              (mais sans date) est créé
            intevt.setName("coffee");
            intevts.add(intevt);
            intcoffeepin.setStoredTokens(intevts); //Et déposé sur le point d'interface d'entrée du modèle FSM

            extcoffeepin.getStoredTokens().clear(); //Et l'événement est consommé
        }

        ModhelxViewer.refresh();
    }

    public void adaptOut(){
        System.out.println(this.getName()+" : adaptOut");

        Pin intservedpin = PinImpl.selectOnePin(this.getInternalModel().getStructure().getPinsOut(),"served");
        if(intservedpin.getStoredTokens().size()==1){ //Si le café a été servi par le modèle FSM
            Pin extservedpin = PinImpl.selectOnePin(this.getPinsOut(),"served");
            ArrayList<DEEventImpl> extevts = new ArrayList<DEEventImpl>();
            DEEventImpl extevt = DEFactoryImpl.getInstance().createDEEvent(); //Un événement DE équivalent est créé
            //Avec une date égale à la somme de la date du dernier événement "coffee" reçu et du délai de préparation
              en paramètre
            extevt.setTimestamp(this.tLastDEevt+Integer.parseInt(this.getParameters().get("preparationTime").getValue()));
            extevt.setValue(0);
            extevts.add(extevt);
            extservedpin.setStoredTokens(extevts); //Et déposé sur le point d'interface de sortie du bloc d'interface

            intservedpin.getStoredTokens().clear(); //Et l'événement est consommé
        }

        ModhelxViewer.refresh();
    }

    public void setPreparationTime(int time){
        this.getParameters().get("preparationTime").setValue(""+time);
    }
}
```

Table des figures

Liste des tableaux

Table des listings

Nos publications

[BH08] Boulanger, F. and C. Hardebolle: *Simulation of Multi-Formalism Models with ModHel'X.* In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation (ICST 2008)*, pages 318–327, Lillehamer, Norway, April 2008. IEEE Computer Society.

[Har] Hardebolle, C.: *ModHel'X – Framework for Heterogeneous & eXecutable Modeling.* `http://wwwdi.supelec.fr/logiciels/modhelx`.

[HB07] Hardebolle, C. and F. Boulanger: *ModHel'X: A Component-Oriented Approach to Multi-Formalism Modeling.* In Lara, T. Levendovszky J. de and P. J. Mosterman (editors): *Proceedings of the 2007 Workshop on Multi-Paradigm Modeling (MPM'07) at the 10th IEEE/ACM International Conference on Model-Driven Engineering Languages and Systems (MODELS 2007)*, volume 1, pages 49–60, Nashville TN, United States, October 2007. BME-DAAI Technical Report Series. Best paper, published in Models in Software Engineering – Workshops and Symposia at MoDELS 2007 (LNCS).

[HB08] Hardebolle, C. and F. Boulanger: *ModHel'X: A Component-Oriented Approach to Multi-Formalism Modeling.* Models in Software Engineering – Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers, 5002/2008:247–258, June 2008.

[HB09] Hardebolle, C. and F. Boulanger: *Multi-formalism modelling and model execution.* International Journal of Computers and Applications (IJCA), special issue on the International Summer School on Software Engineering, To be published in 2009. 12 pages.

[HBMVN07] Hardebolle, C., F. Boulanger, D. Marcadet, and G. Vidal-Naquet: *A Generic Execution Framework for Models of Computation.* In *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007), at the European Joint Conferences on Theory and Practice of Software (ETAPS 2007)*, pages 45–54. IEEE Computer Society, March 2007.

Bibliographie

[Abr96] Abrial, J. R.: *The B-Book – Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AFI] AFIS: *L'Ingénierie Système.* `http://www.afis.fr/praout/ingsys/ingsys.htm`.

[AFR06] Amyot, D., H. Farah, and J. F. Roy: *System Analysis and Modeling: Language Profiles*, volume 4320/2006 of *Lecture Notes in Computer Science*, chapter Evaluation of Development Tools for Domain-Specific Modeling Languages, pages 183–197. Springer Berlin / Heidelberg, 2006.

[AG94] Allen, R. and D. Garlan: *Formalizing architectural connection.* In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press, ISBN 0-8186-5855-X.

[And96] André, C.: *Representation and analysis of reactive behaviors: A synchronous approach.* In *CESA'96 (Computational Engineering in Systems Applications)*, pages 19–29. IEEE-SMC, 1996.

[Att08] Attiogbé, C.: *Mastering specification heterogeneity with multifacet analysis.* In *Proceedings of the MoVaH'08 Workshop on Modeling, Validation and Heterogeneity, held in conjunction with the first IEEE International Conference on Software Testing, verification and validation ICST 2008*, April 2008.

[BB02] Bézivin, J. et X. Blanc: *Promesses et interrogations de l'approche MDA.* Développeur Référence – Septembre 2002, 2002.

[BBC05] Bracciali, A., A. Brogi, and C. Canal: *A formal approach to component adaptation.* Journal of Systems and Software, 74:45–54, January 2005.

[BBJ07] Bezivin, J., M. Barbero, and F. Jouault: *On the applicability scope of model driven engineering.* In *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007), at the European Joint Conferences on Theory and Practice of Software (ETAPS 2007)*, pages 3–7. IEEE Computer Society, March 2007.

[BBS06] Basu, A., M. Bozga, and J. Sifakis: *Modeling Heterogeneous Real-time Systems in BIP.* In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM06)*, pages 3–12, September 2006.

[BC85] Berry, G. and L. Cosserat: *The ESTEREL Synchronous Programming Language and its Mathematical Semantics.* In *Seminar on Concurrency, Carnegie-Mellon*

University, pages 389–448, London, UK, 1985. Springer-Verlag, ISBN 3-540-15670-4.

[BCCSV05] Benveniste, A., B. Caillaud, L. P. Carloni, and A. L. Sangiovanni-Vincentelli: *Tag machines*. In *Proceedings of the 5th ACM International Conference On Embedded Software (EMSOFT 2005)*, pages 255–263. ACM, September 2005.

[BCM+94] Brown, A. W., D. J. Carney, E. J. Morris, D. B. Smith, and P. F. Zarrella: *Principles of CASE tool integration*. Oxford University Press, Inc., New York, NY, USA, 1994, ISBN 0-19-509478-6.

[BD99] Bröhl, A. P. und W. Dröschel: *Das V-Modell*. Der Standard in der Softwareentwicklung mit Praxisleitfaden, 2, 1999.

[Béz02] Bézivin, J.: *Les nouveaux défis des systèmes complexes et la réponse MDA de l'OMG*. Journées Francophones pour l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents 2002, 2002. `http://www2.lifl.fr/jfiadsma2002/talks/jfiadsma2002-Bezivin.pdf`.

[Béz05] Bézivin, J.: *On the unification power of models*. Software and System Modeling, 4(2):171–188, 2005.

[BGJ91] Benveniste, A., P. Le Guernic, and C. Jacquemot: *Synchronous programming with events and relations: the SIGNAL language and its semantics*. Sci. Comput. Program., 16(2):103–149, 1991, ISSN 0167-6423.

[BGM91] Bernot, G., M. C. Gaudel, and B. Marre: *Software testing based on formal specifications: a theory and a tool*. Software Engineering Journal, 6(6):387–405, November 1991, ISSN 0268-6961.

[BH98] Börger, E. and J. K. Huggins: *Abstract State Machines 1988-1998: Commented ASM Bibliography*. CoRR, cs.SE/9811014, 1998.

[BH08] Boulanger, F. and C. Hardebolle: *Simulation of Multi-Formalism Models with ModHel'X*. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation (ICST 2008)*, Lillehamer, Norway, April 2008. IEEE Computer Society.

[BHLM91] Buck, J. T., S. Ha, E. A. Lee, and D. G. Messerschmitt: *Ptolemy: A mixed-paradigm simulation/prototyping platform in c++*. In *Proceedings of the C++ At Work Conference*, Santa Clara, CA, November 1991.

[BJV04] Bézivin, J., F. Jouault, and P. Valduriez: *On the Need for Megamodels*. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004. `http://www.sciences.univ-nantes.fr/lina/atl/www/papers/OOPSLA04/bezivin-megamodel.pdf`.

[BLP+02] Balarin, F., L. Lavagno, C. Passerone, A. L. S. Vincentelli, M. Sgroi, and Y. Watanabe: *Modeling and designing heterogeneous systems*. In *Concurrency and Hardware Design, Advances in Petri Nets*, volume 2549 of *Lecture Notes in Computer Science*, pages 228–273, London, UK, 2002. Springer, ISBN 3-540-00199-9.

[Bou91] Boussinot, F.: *Reactive C: An Extension of C to Program Reactiv Systems*. Software Practice and Experience, pages 401–428, apr 1991.

[BPSV01] Burch, J. R., R. Passerone, and A. L. Sangiovanni-Vincentelli: *Overcoming Heterophobia: Modeling Concurrency in Heterogeneous Systems*. In *Proceedings of the second International Conference on Application of Concurrency to System Design*, page 13, June 2001.

[BSC+97] Balarin, F., E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, , and A. Sangiovanni-Vincentelli: *Hardware-Software Co-design of Embedded Systems – The POLIS approach.* Kluwer Academic Publishers, 1997.

[Car] Carnegie Mellon Software Engineering Institute: *Computer-Aided Software Engineering (CASE) Environments.* `http://www.sei.cmu.edu/legacy/case/case_whatis.html`.

[Cas85] Case, A. F.: *Computer-aided software engineering (CASE): technology for improving software development productivity.* SIGMIS Database, 17(1):35–43, 1985, ISSN 0095-0033.

[CBP+04] Carloni, L. P., M. Di Benedetto, R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli: *Modeling techniques, programming languages and design toolsets for hybrid systems.* Technical report, IST - Columbus Project, 2004. `http://www.columbus.gr/documents/public/WPHS/Columbus_DHS3_0.2_Cover.pdf`.

[CEA] CEA: *Papyrus – Open source tool for graphical UML2 modeling.* `http://www.papyrusuml.org`.

[CH06] Czarnecki, K. and S. Helsen: *Feature-based survey of model transformation approaches.* IBM Syst. J., 45(3):621–645, 2006, ISSN 0018-8670.

[Cha84] Chapiro, D.: *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Dept. of Computer Science, Standford University, 1984.

[CHM+99] Coste, P., F. Hessel, Ph. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, and A. A. Jarraya: *Multilanguage design of heterogeneous systems.* In *CODES '99: Proceedings of the seventh international workshop on Hardware/software codesign*, pages 54–58, New York, NY, USA, 1999. ACM, ISBN 1-58113-132-1.

[Cip08] Cipu, A.: *Interpréteur pour une extension impérative d'OCL.* Rapport technique 2008-06-30-DI-AC, Département Informatique de Supélec, 2008.

[CL85] Chandy, K. M. and L. Lamport: *Distributed Snapshots: Determining Global States of Distributed Systems.* ACM Transactions on Computer Systems, 3(1):63–75, February 1985.

[CM08] Codescu, M. and T. Mossakowski: *Heterogeneous colimits.* In *Proceedings of the MoVaH'08 Workshop on Modeling, Validation and Heterogeneity, held in conjunction with the first IEEE International Conference on Software Testing, verification and validation ICST 2008*, April 2008.

[CSAJ05] Chen, Kai, Janos Sztipanovits, Sherif Abdelwahed, and Ethan K. Jackson: *Semantic anchoring with model transformations.* In *First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005*, Lecture Notes in Computer Science, pages 115–129. Springer, 2005.

[CSN+05] Chen, K., J. Sztipanovits, S. Neema, M. Emerson, and S. Abdelwahed: *Toward a semantic anchoring infrastructure for domain-specific modeling languages.* In *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)*, pages 35–44, New Jersey, September 2005.

[CSN07] Chen, K., J. Sztipanovits, and S. Neema: *A case study on semantic unit composition.* In *MISE '07: Proceedings of the International Workshop on Modeling in Software Engineering*, page 3, Washington, DC, USA, 2007. IEEE Computer Society, ISBN 0-7695-2953-4.

[CSW08] Clark, T., P. Sammut, and J. Willans: *Applied Metamodelling: A Foundation for Language-Driven Development*. Ceteva, second edition edition, 2008. http://www.ceteva.com/book.html.

[dAH01a] Alfaro, L. de and T. A. Henzinger: *Interface automata*. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM, ISBN 1-58113-390-1.

[dAH01b] Alfaro, L. de and T. A. Henzinger: *Interface Theories for Component-Based Design*. In Henzinger, T. A. and C. M. Kirsch (editors): *Embedded Software, First International Workshop, EMSOFT 2001*, volume 2211, pages 148–165, Tahoe City, CA, USA, October 2001. Springer.

[Dav03] Davis, J.: *Gme: the generic modeling environment*. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–83, New York, NY, USA, 2003. ACM, ISBN 1-58113-751-6.

[DB] Dupe, G. and M. Belaunde: *SmartQVT*. http://smartqvt.elibel.tm.fr/index.html.

[DD06] Duchien, L. et C. Dumoulin (rédacteurs): *Actes des 2èmes journées sur l'Ingénierie Dirigée par les Modèles*. PlanetMDE, 2006, ISBN 2-7261-1290-8. http://megaplanet.org/idm06/actes.pdf.

[dLV02] Lara, J. de and H. Vangheluwe: *ATOM^3: A Tool for Multi-formalism Modelling and Meta-modelling*. In *5th Fundamental Approaches to Software Engineering International Conference (FASE 2002)*, pages 595–603, April 2002.

[dS13] Saussure, F. de: *Cours de linguistique générale*. Payot, 1995"a édition, 1913.

[dSA04] Système (AFIS), Association Français d'Ingénierie: *Glossaire de Base de l'Ingénierie de Systèmes*, octobre 2004. http://www.afis.fr/nav/gt/is/doc/Glossaire_Base-V1-2.pdf.

[Dur71] Durand, D.: *La Systémique*. Presses Universitaires de France – PUF, 2006"a édition, 1971.

[Ecla] Eclipse: *Eclipse Modeling Project*. http://www.eclipse.org/modeling/.

[Eclb] Eclipse: *Ecore*. http://download.eclipse.org/modeling/emf/emf/javadoc/2.4.1/org/eclipse/emf/ecore/package-summary.html.

[Eclc] Eclipse: *GEF (Graphical Editing Framework*. http://www.eclipse.org/gef/.

[Ecld] Eclipse: *GMF (Graphical Modeling Framework*. http://www.eclipse.org/modeling/gmf/.

[Ecle] Eclipse Foundation: *Eclipse Modeling Framework (EMF)*. http://www.eclipse.org/modeling/emf/.

[Edw97] Edwards, S. A.: *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. PhD thesis, University of California, Berkeley, mar 1997.

[EGdL+05] Ehrig, K., E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and Sz. Varró-Gyapay: *Model transformation by graph transformation: A comparative study*. In *Proceedings of MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005. http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2005/mtip05.pdf.

[EJL+03] Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong: *Taming heterogeneity – The Ptolemy approach.* Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software, 91(1):127–144, January 2003.

[ES06] Emerson, M. and J. Sztipanovits: *Techniques for metamodel composition.* In *OOPSLA – 6th Workshop on Domain Specific Modeling*, pages 123–139, October 2006. `http://chess.eecs.berkeley.edu/pubs/289.html`.

[Fav06] Favre, J. M.: *Megamodelling and Etymology.* In Cordy, James R., Ralf Lämmel, and Andreas Winter (editors): *Transformation Techniques in Software Engineering*, number 05161 in *Dagstuhl Seminar Proceedings*, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[FE98] Fritzson, P. and V. Engelson: *Modelica – A Unified Object-Oriented Language for System Modeling and Simulation.* In *European Conference on Object-Oriented Programming (ECOOP98)*, pages 67–90, July 1998.

[FEBF06] Favre, J. M., J. Estublier et M. Blay-Fornarino (rédacteurs): *L'Ingénierie Dirigée par les Modèles : au-delà du MDA*. Hermès – Lavoisier, 2006. Traité IC2 – Information – Commande – Communication.

[Fer05] Feredj, M.: *Études des méthodes de conception de composants domaine-polymorphes.* Thèse de doctorat, Université Paris-Sud XI et Supélec, 2005.

[FMC00] Forsberg, K., H. Mooz, and H. Cotterman: *Visualizing Project Management, A Model for Business and Technical Success.* John Wiley & Sons, Inc., 2000.

[FNTZ00] Fischer, T., J. Niere, L. Torunski, and A. Zündorf: *Story diagrams: A new graph rewrite language based on the unified modeling language and java.* In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 296–309, London, UK, 2000. Springer-Verlag, ISBN 3-540-67203-6.

[Fon07] Fondement, F.: *Concrete syntax definition for modeling languages.* PhD thesis, EPFL, 2007. `http://library.epfl.ch/theses/?nr=3927`.

[fSISIVU] Software Integrated Systems (ISIS) Vanderbilt University, Institute for: *Graph Rewriting And Transformation (GReAt).* `http://repo.isis.vanderbilt.edu/tools/get_tool?GReAT`.

[GGMC+07] Gössler, G., S. Graf, M. E. Majster-Cederbaum, M. Martens, and J. Sifakis: *Ensuring properties of interaction systems.* In Reps, Thomas W., Mooly Sagiv, and Jörg Bauer (editors): *Program Analysis and Compilation, Theory and Practice, Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2007.

[GHJV95] Gamma, E., R. Helm, R. Johnson, and J. M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Reading, Mass: Addison-Wesley, 1995.

[GS05] Gössler, G. and J. Sifakis: *Composition for component-based modeling.* Science of Computer Programming, 55(1-3):161–183, March 2005, ISSN 0167-6423.

[GSCK04] Greenfield, J., K. Short, S. Cook, and S. Kent: *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley & Sons, 2004.

[GYNJ01] Gerin, P., S. Yoo, G. Nicolescu, and A. A. Jerraya: *Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures.* In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific Design Automation*, pages 63–68, New York, NY, USA, 2001. ACM, ISBN 0-7803-6634-4.

[Har] Hardebolle, C.: *ModHel'X – Framework for Heterogeneous & eXecutable Modeling.* `http://wwwdi.supelec.fr/logiciels/modhelx`.

[HB08] Hardebolle, C. and F. Boulanger: *ModHel'X: A Component-Oriented Approach to Multi-Formalism Modeling.* Models in Software Engineering – Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers, 5002/2008:247–258, June 2008.

[HBMVN07] Hardebolle, C., F. Boulanger, D. Marcadet, and G. Vidal-Naquet: *A Generic Execution Framework for Models of Computation.* In *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007), at the European Joint Conferences on Theory and Practice of Software (ETAPS 2007)*, pages 45–54. IEEE Computer Society, March 2007.

[HCRP91] Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud: *The synchronous data flow programming language LUSTRE.* Proceedings of the IEEE, 79(9):1305–1320, September 1991, ISSN 0018-9219.

[Hol98] Holland, J. H.: *Emergence: From Chaos to Order.* Helix Books Addison-Wesley, 1998.

[HP85] Harel, D. and A. Pnueli: *Logics and models of concurrent systems*, volume 13 of *Nato Asi Series F: Computer And Systems Sciences*, chapter On the development of reactive systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985, ISBN 0-387-15181-8.

[HPW01] Heck, B., J. V. R. Prasad, and L. Wills: *Software enabled control for innovative control technologies for autonomous highly agile and extreme-performance aerial vehicles*, 2001. `http://controls.ae.gatech.edu/sec/status/may01.pdf`.

[HRW] Hein, C., T. Ritter, and M. Wagner: *OSLO – Open Source Library for OCL.* `http://oslo-project.berlios.de/`.

[HS06] Henzinger, T. A. and J. Sifakis: *The embedded systems design challenge.* In *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*, pages 1–15. Springer, August 2006.

[HS07] Henzinger, T. A. and J. Sifakis: *The discipline of embedded systems design.* Computer, pages 32–40, October 2007.

[IBM] IBM: *PL/I.* `http://www-306.ibm.com/software/awdtools/pli/`.

[IEE] IEEE: *IEEE 1666™Open SystemC Language Reference Manual.* `http://standards.ieee.org/getieee/1666/index.html`.

[IMTA05] Inverardi, P., L. Mostarda, M. Tivoli, and M. Autili: *Synthesis of correct and distributed adaptors for component-based systems: an automatic approach.* In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 405–409, New York, NY, USA, 2005. ACM, ISBN 1-59593-993-4.

[INC] INCOSE: *A Consensus of the INCOSE Fellows.* `http://www.incose.org/practice/fellowsconsensus.aspx`.

[JAB+06] Jouault, F., F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez: *ATL: a QVT-like transformation language.* In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 719–720, 2006.

[Jac] Jacobs, B.: *Introduction to Coalgebra. Towards Mathematics of States and Observations.* `http://www.cs.ru.nl/B.Jacobs/CLG/JacobsCoalgebraIntro.pdf`, In preparation, draft electronically available.

[Jac00] Jackson, D.: *Enforcing Design Constraints with Object Logic.* In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 1–21, London, UK, 2000. Springer-Verlag, ISBN 3-540-67668-6.

[JGr] JGraph Ltd.: *JGraph – Java Graph Visualization and Layout.* `http://www.jgraph.com/`.

[Joh] Johnson, S. C.: *Lex & yacc.* `http://dinosaur.compilertools.net/#yacc`.

[KA03] Kong, C. and P. Alexander: *The Rosetta Meta-Model Framework.* In *Proceedings of the IEEE Engineering of Computer-Based Systems Symposium and Workshop (ECBS'03)*, April 2003.

[KASS03] Karsai, G., A. Agrawal, F. Shi, and J. Sprinkle: *On the use of graph transformation in the formal specification of model interpreters.* Journal of Universal Computer Science, 9(11):1296–1321, November 2003. `http://www.jucs.org/jucs_9_11/on_the_use_of`.

[KBC04] Kalnins, A., J. Barzdins, and E. Celms: *Model transformation language MOLA.* In *Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers*, Lecture notes in computer science, pages 62–76. Springer, 2004.

[Ken02] Kent, S.: *Model driven engineering.* In Butler, Michael J., Luigia Petre, and Kaisa Sere (editors): *Third International Conference on Integrated Formal Methods (IFM)*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298, Turku, Finland, May 2002. Springer.

[KGGV07] Krstic, M., E. Grass, F.K. Gurkaynak, and P. Vivet: *Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook.* Design & Test of Computers, IEEE, 24(5):430–441, 2007, ISSN 0740-7475.

[KMB+96] Kieburtz, R. B., L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton: *A software engineering experiment in software component generation.* In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 542–552, Washington, DC, USA, 1996. IEEE Computer Society, ISBN 0-8186-7246-3.

[Lee97] Lee, E. A.: *The ptolemy project – modeling and design of reactive systems*, 1997. `http://ptolemy.eecs.berkeley.edu/presentations/97/tektronix.pdf`.

[Liu00] Liu, J. W. S.: *Real-Time Systems.* Prentice Hall, 2000.

[LLEL03] Liu, X., J. Liu, J. Eker, and E. A. Lee: *Software-Enabled Control: Information Technology for Dynamical Systems*, chapter Heterogeneous Modeling and Design of Control Systems. Wiley-IEEE Press, April 2003.

[LMS03] Lavagno, L., G. Martin, and B. Selic (editors): *UML for real: design of embedded real-time systems.* Kluwer Academic Publishers, Norwell, MA, USA, 2003, ISBN 1-4020-7501-4.

[LNK+01] Ledeczi, A., G. Nordstrom, G. Karsai, P. Volgyesi, and M. Maroti: *On metamodel composition.* In *Proceedings of the 2001 IEEE Conference on Control Applications (CCA)*, Mexico City, Mexico, September 2001.

[LSV98] Lee, E. A. and A. L. Sangiovanni-Vincentelli: *A framework for comparing models of computation.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17(12):1217–1229, 1998.

[LX04] Lee, E. A. and Y. Xiong: *A Behavioral Type System and Its Application in Ptolemy II.* Formal Aspects of Computing, 16(3):210–237, August 2004. `http://ptolemy.eecs.berkeley.edu/publications/papers/03/behavioraltypesystem/`.

[LZ07] Lee, E. A. and H. Zheng: *Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems.* In *Proceedings of the 7th ACM and IEEE international conference on Embedded software (EMSOFT'07)*, pages 114–123, New York, NY, USA, 2007. ACM.

[MA03] Morel, B. and P. Alexander: *Automating Component Adaptation for Reuse.* In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 142–151, Montreal, Canada, October 2003. IEEE Computer Society.

[MB07] Maraninchi, F. and T. Bouhadiba: *42: Programmable Models of Computation for a Component-Based Approach to Heterogeneous Embedded Systems.* In *6th ACM International Conference on Generative Programming and Component Engineering (GPCE'07)*, pages 53–62, October 2007.

[Mbo04] Mbobi, M.: *Modélisation hétérogène non hiérarchique.* Thèse de doctorat, Université Paris-Sud XI et Supélec, 2004.

[Men] Mentor Graphics, Inc.: *Seamless.* `http://www.mentor.com/seamless`.

[MFJ05] Muller, P. A., F. Fleurey, and J. M. Jézéquel: *Weaving Executability into Object-Oriented Meta-Languages.* In Kent, S. and L. Briand (editors): *Proceedings of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS/UML 2005)*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer. `http://www.kermeta.org/documents/articles/Muller05a`.

[MG06] Mens, T. and P. Van Gorp: *A taxonomy of model transformation.* Electr. Notes Theor. Comput. Sci., 152:125–142, 2006.

[Mon91] Montgomery, S. L.: *AD/Cycle: IBM's Framework for Application Development and Case.* Van Nostrand Reinhold, 1991.

[MPS07] Mateescu, R., P. Poizat, and G. Salaün: *Behavioral adaptation of component compositions based on process algebra encodings.* In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 385–388, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-882-4.

[MV04] Mosterman, P. J. and H. Vangheluwe: *Computer automated multi-paradigm modeling: An introduction.* Simulation: Transactions of the Society for Modeling and Simulation International, 80(9):433–450, 2004. Special Issue: Grand Challenges for Modeling and Simulation.

[NMK+02] Nicolescu, G., S. Martinez, L. Kriaa, W. Youssef, S. Yoo, B. Charlot, and A. A. Jerraya: *Application of Multi-domain and Multi-language Cosimulation to an Optical MEM Switch Design.* In *Proceedings of ASP-DAC 2002: 7th Asia South Pacific Design Automation Conference and the 15th International Conference on VLSI Design*, pages 426–431, Washington, DC, USA, 2002. IEEE Computer Society.

[Nor99] Nordstrom, G. G.: *Metamodeling – Rapid Design and Evoluion of Domain-Specific Modeling Environments.* PhD thesis, Vanderbilt University, Electrical Engineering, May 1999.

[OMGa] OMG: *Catalog of UML Profile Specifications.* `http://www.omg.org/technology/documents/profile_catalog.htm`.

[OMGb] OMG: *Common Warehouse Metamodel (CWM), version 1.1.* `http://www.omg.org/technology/documents/formal/cwm.htm`.

[OMGc] OMG: *Corba component model*. http://www.omg.org/technology/documents/formal/components.htm.

[OMGd] OMG: *Diagram Interchange (DI)*. http://www.omg.org/technology/documents/formal/diagram.htm.

[OMGe] OMG: *Meta Object Facility (MOF) 2.0 Query/View/Transformation*. http://www.omg.org/docs/formal/08-04-03.pdf.

[OMGf] OMG: *MetaObject Facility (MOF), version 2.0*. http://www.omg.org/mof/.

[OMGg] OMG: *Model Driven Architecture (MDA)*. http://www.omg.org/mda/.

[OMGh] OMG: *Object Constraint Language (OCL) – version 2.0*. http://www.omg.org/technology/documents/formal/ocl.htm.

[OMGi] OMG: *System Modeling Language (SysML)*. http://www.omgsysml.org/.

[OMGj] OMG: *UML Profile for Modeling and Analysis of Real-time and Embedded systems (MARTE)*. http://www.omgmarte.org/.

[OMGk] OMG: *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (QFTP)*. http://www.omg.org/spec/QFTP/index.htm.

[OMGl] OMG: *Unified Modeling Language (UML), version 2.1.2*. http://www.omg.org/technology/documents/formal/uml.htm.

[OMGm] OMG: *XML Metadata Interchange (XMI), version 2.1.1*. http://www.omg.org/spec/XMI/2.1.1/.

[OSC] OSCI: *Open SystemC Initiative*. http://www.systemc.org/home.

[Poh03] Pohjonen, R.: *Boosting embedded systems development with domain-specific modeling*. RTC Magazine, pages 57–61, 2003.

[PY96] Pezzè, M. and M. Young: *Generation of multi-formalism state-space analysis tools*. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 172–179, New York, NY, USA, 1996. ACM, ISBN 0-89791-787-1.

[RC03] Ray, A. and R. Cleaveland: *Architectural Interaction Diagrams: AIDs for system modeling*. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 396–406, Washington, DC, USA, 2003. IEEE Computer Society, ISBN 0-7695-1877-X.

[RJB04] Rumbaugh, J., I. Jacobson et G. Booch: *UML 2.0 – Guide de référence*. Campus Press, 2004, ISBN 2-7440-1820-1.

[RJB05] Rumbaugh, J., I. Jacobson, and G. Booch: *The Unified Modeling Language Reference Manual, Second Edition*. Pearson Eduction, Inc., 2005, ISBN 0-321-24562-8.

[RV00] Roques, P. et F. Vallée: *UML en action : De l'analyse des besoins à la conception en Java*. Eyrolles, 2000.

[SA06] Streb, J. and P. Alexander: *Using a Lattice of Coalgebras For Heterogeneous Model Composition*. In *Proceedings of the MoDELS Workshop on Multi-Paradigm Modeling*, pages 27–38, Genova, Italy, October 2006.

[Sch95] Schürr, A.: *Specification of graph translators with triple graph grammars*. In *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, London, UK, 1995. Springer-Verlag, ISBN 3-540-59071-4.

[SK97] Sztipanovits, J. and G. Karsai: *Model-integrated computing*. IEEE Computer, 30(4):110–111, 1997, ISSN 0018-9162.

[Sta88] Stankovic, J. A.: *Misconceptions about real-time computing: A serious problem for next-generation systems.* Computer, 21(10):10–19, 1988, ISSN 0018-9162.

[Sto96] Storey, N.: *Safety Critical Computer Systems.* Addison Wesley, 1st edition, 1996.

[Sys] System@tic Paris Région: *Projet Usine Logicielle.* `www.usine-logicielle.org`.

[Tei07] Teich, J. (editor): *Reconfigurable Computing Systems, Special Topic Issue of Journal it - Information Technology*, volume 49. Oldenbourg Verlag, Munich, 2007.

[TGL07] Teehan, Paul, Mark Greenstreet, and Guy Lemieux: *A survey and taxonomy of gals design styles.* IEEE Des. Test, 24(5):418–428, 2007, ISSN 0740-7475.

[Tim07] Timmerman, M.: *Embedded systems: Definitions, taxonomies, field.* Dedicated Systems Experts, 2007. `ftp://ftp.cordis.europa.eu/pub/ist/docs/embedded/`.

[Tri] Trigris.org: *Subversion – Open source version control system.* `http://subversion.tigris.org/`.

[vB82] Bertalanffy, L. von: *Théorie générale des systèmes.* DUNOD, 2002"a édition, 1982, ISBN 2-10-006349-9.

[vDKV00] Deursen, A. van, P. Klint, and J. Visser: *Domain-specific languages: an annotated bibliography.* SIGPLAN Not., 35(6):26–36, 2000, ISSN 0362-1340.

[Vit11] Vitruve: *Les Dix Livres d'architecture.* M. Ch.-L. Maufras et al., 1847"a édition, 1511. `http://remacle.org/bloodwolf/erudits/Vitruve/index.htm`.

[Wie48] Wiener, N.: *Cybernetics – Control and Communication in the Animal and the Machine.* The MIT Press, 1965th edition, 1948.

[YS97] Yellin, D. M. and R. E. Strom: *Protocol specifications and component adaptors.* ACM Transactions on Programming Languages and Systems, 19(2):292–333, 1997, ISSN 0164-0925.

[Zur05] Zurawski, R. (editor): *Embedded Systems Handbook.* Industrial Information Technology. CRC Press, 2005.

[Zur07] Zurawski, R.: *Embedded systems in industrial applications – challenges and trends.* In *IEEE Second International Symposium on Industrial Embedded Systems – SIES'2007*, Lisbon, Portugal, 2007. IEEE.

Résumé

Dans le contexte de l'Ingénierie Dirigée par les Modèles, l'utilisation de multiples paradigmes de modélisation pour développer un système complexe est à la fois inévitable et essentielle. Les modèles qui représentent un tel système sont donc hétérogènes, ce qui rend tout raisonnement global sur le système difficile. L'objectif de la modélisation multi-paradigme est de faciliter l'utilisation conjointe de modèles hétérogènes pendant le cycle de développement. Les travaux exposés dans cette thèse concernent l'étude de l'hétérogénéité des modèles et la conception d'une approche pour la modélisation multi-paradigme des systèmes.

Nous caractérisons les causes de l'hétérogénéité des modèles par rapport au cycle de développement puis identifions différents types d'hétérogénéité. En nous basant sur ces causes d'hétérogénéité, nous proposons un cadre d'étude pour le domaine de la modélisation multi-paradigme avec différents axes de recherche.

La multidisciplinarité de la modélisation multi-paradigme rend applicables des techniques issues de différents domaines. Nous proposons un état de l'art et une classification des techniques dont nous avons étudié la pertinence par rapport à l'hétérogénéité. La gamme des techniques présentées inclut les transformations de modèles, la composition de méta-modèles, la composition de modèles, l'adaptation de composants, la co-simulation ou encore les méga-modèles.

Nous présentons ensuite ModHel'X, l'approche de composition de modèles pour la modélisation multi-paradigme que nous avons développée. Elle s'appuie sur le concept de modèle de calcul et permet :

1. de spécifier la sémantique d'un langage de modélisation de manière exécutable à travers la spécialisation opérationnelle d'une sémantique abstraite pour les modèles de calcul ;
2. de spécifier explicitement les mécanismes de composition à utiliser entre des modèles hétérogènes via une structure de modélisation appelée bloc d'interface ;
3. de simuler le comportement global de modèles hétérogènes par un algorithme générique d'exécution que nous avons défini.

Une implémentation de ModHel'X a été réalisée sous la forme d'un framework s'appuyant sur EMF (Eclipse Modeling Framework).

Mots clés : modélisation multi-paradigme, hétérogénéité des modèles, modélisation hétérogène, composition de modèles, langage de modélisation, modèle de calcul, exécution de modèles, Ingénierie Dirigée par les Modèles (IDM)

Abstract

In the context of Model Driven Engineering, the use of multiple modeling paradigms for developing complex systems is both unavoidable and essential. It results in the heterogeneity of the models representing the considered system and makes global reasoning about the system difficult. The objective of multi-paradigm modeling is to ease the joint use of heterogeneous models during the development cycle. In the work presented in this dissertation, we focus on the study of the heterogeneity of models and propose an approach to multi-paradigm modeling.

We first qualify the causes of the heterogeneity of models with respect to the development cycle and we identify several types of heterogeneity. On this basis, we propose a framework for the study of the domain of multi-paradigm modeling with several research axis.

The multidisciplinarity of multi-paradigm modeling allows the use of techniques from various fields. We propose a survey and a classification of the techniques which are relevant with respect to heterogeneity. The range of the techniques that we present includes model transformation, meta-model composition, model composition, component adaptation, co-simulation and mega-models.

Then we present ModHel'X, the approach to the composition of models for multi-paradigm modeling that we developed. It relies on the concept of model of computation and allows:

1. the specification of the semantics of a modeling language in an executable way by specializing an abstract semantics for models of computation that we developed;
2. the explicit specification of the composition mechanism between heterogeneous models through a special modeling structure called interface block;
3. the simulation of the global behavior of heterogeneous models thanks to a generic execution algorithm that we defined.

ModHel'X has been implemented in a framework based on EMF (Eclipse Modeling Framework).

Keywords: multi-paradigm modeling, heterogeneity of models, heterogeneous modeling, composition of models, modeling language, model of computation, execution of models, Model Driven Engineering (MDE)