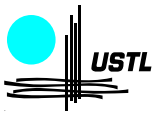


Université des Sciences et Technologies de Lille

THÈSE

présentée et soutenue publiquement le 13 décembre 2006

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Lossan BONDÉ

Transformations de Modèles et Interopérabilité dans la Conception de Systèmes Hétérogènes sur Puce à Base d'IP

Composition du jury

Président :	Pierre BOULET, Professeur	LIFL, Université de Lille I
Rapporteurs :	Henri BASSON, Professeur	LIL, Université du Littoral Côte d'Opale
	Michel HASSENFORDER, Professeur	MIPS ENSISA, Université de Haute Alsace
Examinateurs :	Sylvain LECOMTE , Professeur	LAMIH, Université de Valenciennes
	Cédric DUMOULIN, Maître de conférences	LIFL, Université de Lille I
Directeurs :	Jean-Luc DEKEYSER, Professeur	LIFL, Université de Lille I

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Laboratoire d'Informatique Fondamentale de Lille — UMR 8022 USTL/CNRS
U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 28 77 85 41 – Télécopie : +33 (0)3 28 77 85 37 – email : direction@lifl.fr

Table des matières

Remerciements

Si j'arrive à cette étape de rédaction de ma thèse, c'est parce que j'ai bénéficié de beaucoup de soutien. Aussi, je remerice sincèrement toutes les personnes qui ont contribué à cette œuvre par leur encadrement, leur aide, et leurs conseils.

J'adresse tout d'abord mes remerciements à Jean-Luc Dekeyser, directeur de thèse et chef de l'équipe West, pour mon accueil au sein de son groupe ainsi que pour avoir conduit et guidé mes travaux.

Je remercie Pierre Boulet et Cédric Dumoulin pour avoir co-encadré cette thèse, leurs conseils précieux m'ont permis d'améliorer mes connaissances et d'aboutir à la production de ce travail. Je voudrais aussi les remercier pour le temps qu'ils m'ont accordé tout au long de ces années.

Mes plus sincères remerciements vont également aux membres du jury, qui ont accepté d'évaluer mon travail de thèse. Merci à M. Michel Hassenforder et M. Henri Basson d'avoir accepté d'être les rapporteurs de ce manuscrit, de m'avoir fait l'honneur de juger ce travail avec intérêt. Merci également à M. Sylvain Lecomte d'avoir accepté d'examiner ce travail et de faire partie de mon jury de thèse. À tout le jury, j'exprime ma reconnaissance pour les remarques, les questions et les recommandations qui ont été formulées.

Un grand remerciement est également adressé à tous les membres de l'équipe WEST pour les efforts de chacun pour créer un cadre de travail convivial et très collaboratif. Cette atmosphère m'a permis d'apprécier toutes ces années et de me sentir au sein d'un milieux familial très chaleureux.

Je ne saurais exprimer ma profonde reconnaissance et gratitude à mon épouse pour les multiples relectures des manuscrits, mais également pour les longs moments de solitude endurés pendant la rédaction. Merci également à Baudouin Nzoghu d'avoir accepté de lire le manuscrit.

Je profite également pour dire merci à toute la direction de l'Université Cosendai au Cameroun pour le soutien financier qui m'a permis d'aller jusqu'au bout de ce travail.

Enfin, que tous ceux qui, d'une façon ou d'une autre m'ont soutenu, encouragé, conseillé, trouvent ici l'expression de ma profonde gratitude.

Introduction

Systèmes sur puce (« System on Chip » ou SoC)

Un système embarqué est un système intégré dans un système plus large avec lequel il est interfacé et pour lequel il réalise des fonctions particulières (contrôle, surveillance, communication, etc.). Les systèmes embarqués désignent aussi bien le matériel que le logiciel qui permettent de réaliser ces fonctions. Dans certaines applications, le matériel utilisé est un ordinateur généraliste alors que dans d'autres, il s'agit de matériel dédié. Ces systèmes possèdent souvent la caractéristique principale de fonctionner en temps réel car ils doivent gérer des informations et en déduire des actions avec un délai maîtrisé (soit un délai connu, soit un délai borné). De plus, ils sont souvent dits « critiques », ce qui signifie qu'ils ne doivent jamais faillir (notion de sécurité de fonctionnement oblige). Les domaines dans lesquels on trouve des systèmes embarqués sont de plus en plus nombreux : télécommunication, aéronautique, automobile, électroménager (télévision, four à micro-ondes, etc.).

Pour de nombreux systèmes embarqués, les SoC (« System on Chip ») constituent les composants de base. Un SoC est un circuit intégré qui comporte un ensemble de composants matériels (microprocesseurs, DSP, entrées/sorties ...) connectés entre eux par des bus de communication et une couche logicielle (système d'exploitation temps réel et applicatif)[105]. Les SoC sont aujourd'hui une réalité et leur utilisation va se répandre notamment grâce aux technologies reprogrammables (SoPC : System on Programmable Chip) de type FPGA[1] (« Field-Programmable Gate Array »). La réalisation de ces systèmes repose sur une nouvelle approche de conception qui consiste à associer dans un même composant des cœurs de processeurs exécutant des programmes et des blocs fonctionnels spécifiques appelés IP[2] (« Intellectual Property »). Il s'agit donc d'une conception de systèmes mixtes intégrant logiciel et matériel. Cette conception conjointe du matériel et du logiciel (« Co-Design ») nécessite l'introduction de nouvelles méthodologies traitant des spécifications logicielles et des blocs matériels synthétisables.

[1] Un FPGA est un type de puce logique programmable pouvant contenir des milliers de portes logiques. Il est spécialement utilisé pour le prototypage de circuits intégrés.

[2] Un IP est un programme informatique écrit dans un langage de description de matériel. Par exemple, VHDL, Verilog, etc.

Conception des SoC aujourd'hui

Il existe plusieurs tendances dans la conception des SoC parmi lesquelles la plus en vogue est le recours aux composants virtuels IPs. Le concepteur utilise des IPs très souvent d'origines diverses et ayant des modèles hétérogènes (différents niveaux d'abstractions : comportemental, RTL, etc.). Cette approche permet d'améliorer le délai de mise sur le marché (« time to market »), mais elle nécessite de la part du concepteur de nouvelles méthodes de conception.

Une étude faite lors du congrès ERTS[3] 2006 donne un aperçu des différents problèmes rencontrés dans la conception des systèmes embarqués. Cette étude a clairement montré que la maîtrise de la complexité croissante des systèmes temps réel embarqués est l'un des enjeux majeurs de l'industrie européenne. L'étude de tendances réalisée à l'occasion de cet événement a confirmé la volonté de la communauté de l'embarqué à relever ce défi en mettant un accent particulier sur les points suivants :

- la fiabilité dans les futurs systèmes embarqués,
- la maîtrise de l'ingénierie système et de ses standards émergeants : SysML et AADL,
- l'utilisation des outils de modélisation pour concevoir les systèmes embarqués.

Dans cette dynamique, le monde industriel doit se donner les moyens de modéliser, de concevoir, de développer, d'intégrer et de qualifier comme sûrs, des systèmes embarqués qui doivent aussi rester évolutifs, être facilement maintenables, avec des coûts - de développement, notamment - toujours plus faibles. Les défis sont grands et l'activité autour des méthodes et outils du développement de l'embarqué temps réel bouillonne :

- l'automobile se mobilise autour d'une initiative de standardisation des architectures matérielles et logicielles des systèmes embarqués,
- l'aéronautique explore les ressources de l'Open Source,
- le spatial tend vers l'utilisation des langages standards bien connus pour exprimer ses contraintes embarquées,
- tous s'associent par passion autour de congrès dédiés ou répondent ensemble aux projets de pôles [40].

Par ailleurs, l'information sur ces technologies reste de formes multiples, en perpétuelle régénérescence, souvent diffuse, parfois non publiée.

Contexte de l'étude et approche proposée

Pour répondre au besoin de méthodologie (exprimé plus haut) dans la conception des systèmes embarqués, en général, et des applications de traitement de signal intensif, en particulier, l'équipe DaRT [4] de l'INRIA Futurs a proposé le projet *Gaspard*.

Gaspard est un environnement de co-design (conception conjointe logiciel/matériel) et de simulation pour des applications de traitement de signal intensif. Le principe à la base de Gaspard est d'utiliser les avancées méthodologiques du génie logiciel, notamment les techniques de modélisation et de transformations de modèles proposées par le *Model Driven Architecture*

[3] ERTS (Embedded Real Time Software, http ://www.erts2006.org

[4] DaRT (Data parallelism for Real Time, http ://www.inria.fr/ recherche/equipes/dart.fr.html

(« MDA ») et l'*Ingeniérie Dirigée par les Modèles* ((« IDM ») connue aussi sous l'appellation *Model Driven Engineering* (« MDE »)) dans la conception des systèmes embarqués.

Gaspard vise plusieurs plates-formes de simulation (Java, OpenMP, SystemC, VHDL, etc.) et différents niveaux d'abstractions (TLM, RTL, etc.). Les modèles des différentes plates-formes et niveaux d'abstraction sont générés dans Gaspard par transformations de modèles. L'hétérogénéité des plates-formes visées introduit un problème d'interopérabilité au niveau des plates-formes.

Dans ce travail de thèse, nous proposons une solution pour répondre à ce besoin d'interopérabilité. Cette solution passe par l'introduction de la traçabilité dans les transformations de modèles. Cette traçabilité est ensuite utilisée pour générer un pont (« bridge ») d'interopérabilité.

Les termes clés de ce document sont donc : approches orientées modèles (MDA, IDM), transformations de modèles, traçabilité et interopérabilité.

Plan du document

La suite de ce manuscrit comporte sept chapitres qui s'articulent en deux parties. La première partie, composée des quatres premiers chapitres, introduit les concepts et un état de l'art du domaine d'étude. La seconde présente la contribution de cette thèse et un exemple d'application des travaux réalisés.

Première partie

Dans la première partie, le premier chapitre décrit les approches orientées modèles ; il présente les concepts fondamentaux de ces approches et les principales propositions de leur mise en œuvre. Les chapitres 2 et 3 présentent un état de l'art sur les transformations de modèles, la traçabilité et l'interopérabilité. Le dernier présente le projet Gaspard dans le contexte de ce travail.

Dans cette partie constituant l'état de l'art des travaux relatifs à notre sujet, nous présentons seulement les concepts et principes nécessaires à la compréhension de notre propos et nous donnons des pointeurs vers les sources permettant au lecteur d'approfondir ses connaissances sur le sujet.

Deuxième partie

La deuxième partie, quant à elle, comporte trois chapitres. Le premier, le chapitre 5, traite des transformations de modèles dans Gaspard ; il présente les expérimentations de transformations de modèles que nous avons réalisées au cours de nos travaux. Le chapitre 6 aborde la traçabilité et l'interopérabilité : il présente un modèle de traçabilité et montre comment ce modèle peut être utilisé pour réaliser l'interopérabilité des langages dans un système multi plates-formes. Notre propostion, basée en particulier sur les approches de l'ingénierie dirigée par les modèles nous a permis de modéliser les informations nécessaires

pour la mise en œuvre de l'interopérabilité dans un système réalisé dans plusieurs langages de programmation.

Le chapitre 7 traite de l'interopérabilité des niveaux de simulation. Il présente au moyen d'un exemple, la problématique de mise en mise en œuvre de l'interopérabilité dans une simulation à deux niveaux d'abstractions (TLM CABA et TLM PVT).

Conclusion et perspectives

Enfin, ce document se termine par un bilan du travail réalisé et une présentation de quelques perspectives.

Chapitre 1

Ingénierie dirigée par les modèles (IDM)

La fin des années 60 marque une étape importante dans le monde du logiciel. A cette époque apparaissent les ordinateurs de la troisième génération, de plus en plus puissants et de moins en moins coûteux. Des progrès notables dans le domaine de la construction du matériel informatique sont réalisés. Mais du côté du génie logiciel, la conception et la réalisation des logiciels relèvent encore du domaine de l'artisanat. Alors que de nouvelles machines rendaient possibles des applications jusqu'alors irréalisables, les méthodes de développement logiciel de l'époque ne permettaient pas la prise en compte de grands systèmes. Ce décalage connu sous le nom de *crise du logiciel* fut caractérisé par le fait que :

- la construction de logiciels coûtait très cher (200 millions de dollars pour fabriquer OS-360),
- les délais n'étaient pas respectés (2 ans de retard pour les premiers compilateurs PL/1, Algol 68, ADA),
- les logiciels n'étaient pas évolutifs (parfois écrits en assembleur pour un type de machine) ce qui les rendait très rapidement obsolètes,
- les performances des systèmes étaient parfois poussives (Univac, le système de réservation pour United Air Lines au début des années 75 n'a jamais servi car les temps de réponse étaient trop longs),
- la fiabilité était aléatoire (la sonde américaine *Mariner 1*, lancée le 22 Juillet 1962, qui devait aller sur Vénus, s'est perdue à cause d'une mauvaise instruction),
- la convivialité d'utilisation des systèmes était discutable (des interfaces homme/machine inexistantes).

Pour faire face à cette crise, l'industrie du logiciel a évolué sur plusieurs axes :

- *Paradigme* : plusieurs paradigmes de développement ont vu le jour pour définir un cadre de gestion du développement de logiciel afin de définir un processus, contrôler et maîtriser les coûts et les délais de production des logiciels,
- *Méthodologies de conception* : des méthodologies pour mettre en œuvre les paradigmes de développement,
- *Langages de programmation* : des langages de programmation plus évolués et généralement assortis d'environnement intégré de développement ont vu le jour.

Au nombre des évolutions dans le développement des logiciels, il faut citer l'apparition de plusieurs méthodes de modélisation : Merise [116], SSADM (Structured Systems Analysis and Design Methodology)[45], UML (Unified Modeling Language) [60], etc. Ces méthodes vont permettre d'appréhender le concept de modèle en informatique. Elles proposent des concepts et une notation permettant de décrire le système à concevoir. En général, à chaque étape du cycle de vie du système, un ensemble de documents généralement constitués de diagrammes permettent aux concepteurs, développeurs, utilisateurs et autres entités impliquées de partager leur perception du système. Ces méthodes de modélisation ont été parfois critiquées pour leur lourdeur et leur manque de souplesse face à l'évolution rapide du logiciel. Elles ont conduit à la notion de *modèles contemplatifs* [46] - un modèle qui sert essentiellement à communiquer et comprendre, mais reste passif par rapport à la production. Ainsi, après un demi-siècle de pratique et d'évolution, on constate aujourd'hui que le processus de production de logiciels est toujours centré sur le code.

La figure 1.1 donne un aperçu de l'évolution du génie logiciel dans le temps. Elle montre qu'au fil du temps, des avancées technologiques importantes ont été progressivement effectuées afin de pouvoir maîtriser la complexité croissante du développement des logiciels. Le paradigme objet a beaucoup contribué à améliorer l'aspect structurel de décomposition de l'architecture d'un système et la réutilisation technique de composants ; l'usage des *frameworks applicatifs* a permis d'élever le niveau d'abstraction d'un domaine, les *design patterns* ont permis de décrire les solutions récurrentes, et les techniques telles que la *programmation par aspects*, la *programmation pour sujets* ont permis d'intégrer d'une manière cohérente les multiples facettes de composants logiciels d'un système.

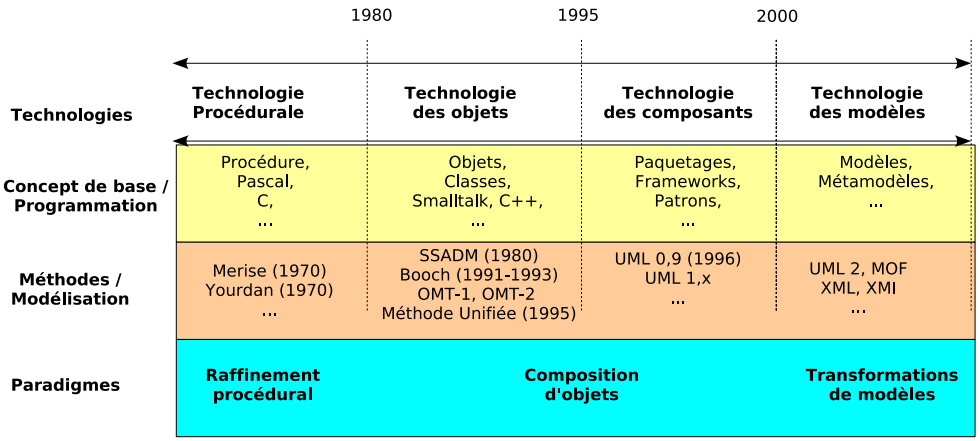

FIG. 1.1 – Evolution du génie logiciel

L'Ingénierie Dirigée par les Modèles (« *IDM* ») vient pallier la déficience des méthodes traditionnelles de modélisation en partant du constat suivant : « *Les modèles contemplatifs sont interprétés par l'homme alors que la préoccupation première de l'informaticien est de produire des artefacts interprétables par la machine* » [46, p. 22]. Elle œuvre à fournir un cadre de développement logiciel dans lequel les modèles passent de l'état passif (contemplatif) à l'état actif (productif) et deviennent les éléments de première classe dans le processus de développement des logiciels.

Dans ce chapitre, nous présentons dans un premier temps le principe et les concepts fondamentaux de l'IDM et dans un deuxième temps nous décrivons le MDA (Model Driven Architecture) comme un exemple de scénario de mise en œuvre de l'IDM.

1.1 Principe et concepts fondamentaux

Notons d'abord que le terme *Ingénierie Dirigée par les Modèles (IDM)* possède plusieurs synonymes (*Model Driven Engineering (MDE)*, *Model Driven Development (MDD)*, *Model Driven Software Development (MDSD)*, etc.). L'IDM est encore un domaine de recherche récent, dynamique et en pleine évolution. Cela se traduit par une pluralité d'idées, de concepts et de terminologies compétitives qui tendent à créer une confusion dans ce domaine. Dans cette section, nous tentons de dégager le principe et les concepts de base généralement acceptés de tous et qui forment le socle de l'IDM.

Principe de l'IDM

Comme nous l'avons déjà indiqué dans l'introduction de ce chapitre, l'idée d'utiliser des modèles pour maîtriser la complexité des logiciels existe depuis des années, mais dans la pratique, son application a été partielle dans les processus de développement des logiciels. En l'occurrence, elle a été mise en œuvre pour la structuration et la composition pendant la phase de conception, et pour la vérification pendant les phases de test.

Le principe de l'IDM consiste à utiliser intensivement et systématiquement les modèles tout au long du processus de développement logiciel. Les modèles devront désormais non seulement être au cœur même du processus, mais également devenir des entités interprétables par les machines. Par analogie au « *tout est objet* » des années 80, on pourrait dire que le principe de base de l'IDM consiste à dire que « *tout est modèle* » [29].

Un système peut être vu sous plusieurs angles ou points de vue. Les modèles offrent la possibilité d'exprimer chacun de ces points de vue indépendamment des autres. Cette séparation des différents aspects du système permet non seulement de se dégager de certains détails, mais permet également de réaliser un système complexe par petits blocs plus simples et facilement maîtrisables. Ainsi, on pourrait utiliser des modèles pour exprimer les concepts spécifiques à un domaine d'application, des modèles pour décrire des aspects technologiques, etc., chacun de ces modèles étant exprimé dans la notation et/ou le formalisme les plus appropriés.

Concepts fondamentaux de l'IDM

Pour décrire les concepts de base de l'IDM nous nous servons de trois notions fondamentales : (1) **l'interprétation des modèles par les machines**, (2) **la productivité des modèles**, (3) **la séparation des différents aspects d'un système**.

Interprétation des modèles par les machines

Pour obtenir des modèles interprétables par les machines, il faut pouvoir formaliser l'expression d'un modèle, c'est-à-dire définir un langage d'expression de modèle. L'IDM propose le concept de *métamodèle* à cette fin.

Définition 1.1. *Un* métamodèle *est un langage qui permet d'exprimer des modèles. Il définit les concepts ainsi que les relations entre concepts nécessaires à l'expression de modèles.*

Un modèle écrit dans un métamodèle donné sera dit conforme à ce dernier. La relation entre modèle et métamodèle (*conforme à*) peut être par analogie aux langages de programmation comparée à la relation entre une variable et son type ou un objet et sa classe (dans le cas des langages objets). On rencontre également cette relation sous la forme *instance de*. Un modèle est une instance d'un métamodèle.

Productivité des modèles

Avant de parler de la productivité des modèles, nous allons d'abord donner une définition de ce qu'est un modèle. Nous retiendrons la définition suivante tirée de [66].

Définition 1.2. *Un* modèle *est une abstraction d'un système étudié, construite dans une intention particulière. Il doit pouvoir être utilisé pour répondre à des questions sur le système.*

Comme le montre cette définition, la notion de modèle va de pair avec la notion de système. En effet, un modèle est conçu pour *représenter* quelque chose que l'on désigne ici par le terme système.

Définition 1.3. *Un* système *est une construction théorique que forme l'esprit sur un sujet (par exemple, une idée qui est mise en œuvre afin d'expliquer un phénomène physique qui peut être représenté par un modèle mathématique) [7].*

Un modèle est productif soit parce qu'il est directement exécutable par une machine, soit parce qu'il permet de produire des artefacts exécutables. Le second cas suppose la possibilité de pouvoir réaliser des opérations sur le modèle pour produire l'artefact exécutable. Cette notion d'opération sur les modèles est connue dans l'IDM sous le concept de *transformations de modèles*. Hubert Kadima [73] nous donne les définitions suivantes :

Définition 1.4. *Une transformation de modèles est la génération d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources conformément à une définition de transformation.*

Définition 1.5. *Une définition de transformation est un ensemble de règles de transformation qui décrivent globalement comment un modèle décrit dans un langage source peut être transformé en un modèle décrit dans un langage cible.*

Définition 1.6. *Une règle de transformation est une description de la manière dont une ou plusieurs constructions dans un modèle source peuvent être transformées en une ou plusieurs constructions dans un modèle cible.*

Si la première définition (1.4) telle que présentée ici est largement acceptée, les définitions 1.5 et 1.6 sont l'objet de controverses. Nous reviendrons sur ces aspects dans le chapitre 2.

Par ailleurs, la notion d'exécution (modèles ou artefacts exécutables) fait intervenir un autre concept, celui de *plate-forme*. Tout système exécutable est défini dans le contexte d'une plate-forme [46, p. 72].

Définition 1.7. *Une* plate-forme *est un système offrant des services nécessaires à la construction, la réalisation ou l'exécution d'autres systèmes.*

Les plates-formes servent de support à la construction et à l'exécution des systèmes. Elles peuvent être logicielles (par exemple, systèmes d'exploitation, intergiciels, machines virtuelles, environnement de programmation, technologies pour le formatage des informations comme XML) ou matérielles (par exemple, PC, PDA, calculateurs ou chipsets).

La notion de plate-forme n'a d'intérêt que dans la mesure où elle peut être modélisée pour être prise en compte dans le processus de développement des systèmes. Il s'avère donc indispensable de définir des modèles de plates-formes. Ces modèles, lorsqu'ils sont intégrés dans la définition d'un système, vont restreindre sa mise en œuvre à l'espace technique de la plate-forme.

Séparation des différents aspects d'un système

Un système étant en général très complexe et donc difficile à appréhender dans son intégralité, il est d'usage de le décomposer en plusieurs sous-systèmes plus simples. Chacun de ces sous-systèmes se focalise sur un aspect particulier du système et fait abstraction des autres aspects. Chaque aspect du système est représenté par un modèle décrit à l'aide d'un métamodèle. Les métamodèles utilisés pour décrire les différents modèles représentant le système peuvent être différents. Cette diversité exige l'utilisation d'un langage unificateur et d'intégration pour les outils qui vont manipuler les différents métamodèles. Pour répondre à ce besoin, la notion de *métamétamodèle* a été définie. Ce concept permettra de disposer d'un langage unique pour décrire les métamodèles.

Synthèse

Dans cette section, nous avons présenté les différents concepts et le principe de l'IDM qui se résument sur la figure 1.2. Cette figure montre que dans l'approche IDM, un système à réaliser est décomposé en plusieurs sous-systèmes. Chaque sous-système aborde un aspect particulier du système général et est décrit à l'aide d'un ou de plusieurs modèles. Chaque modèle est lui-même décrit dans un langage dit métamodèle, lequel est à son tour décrit par un métamétamodèle. La conception dans l'IDM se fait par une modélisation hiérarchique avec quatre niveaux nommés sur la figure par *M0, M1, M2 et M3*.

Par ailleurs, les transformations de modèles, elles-mêmes décrites par des modèles selon un métamodèle, permettent de rendre les modèles productifs dans la chaîne de conception du logiciel. Ainsi, pour réaliser un système selon l'approche de l'IDM, le développeur crée des instances des modèles du niveau *M1*, et applique les transformations pour générer d'autres

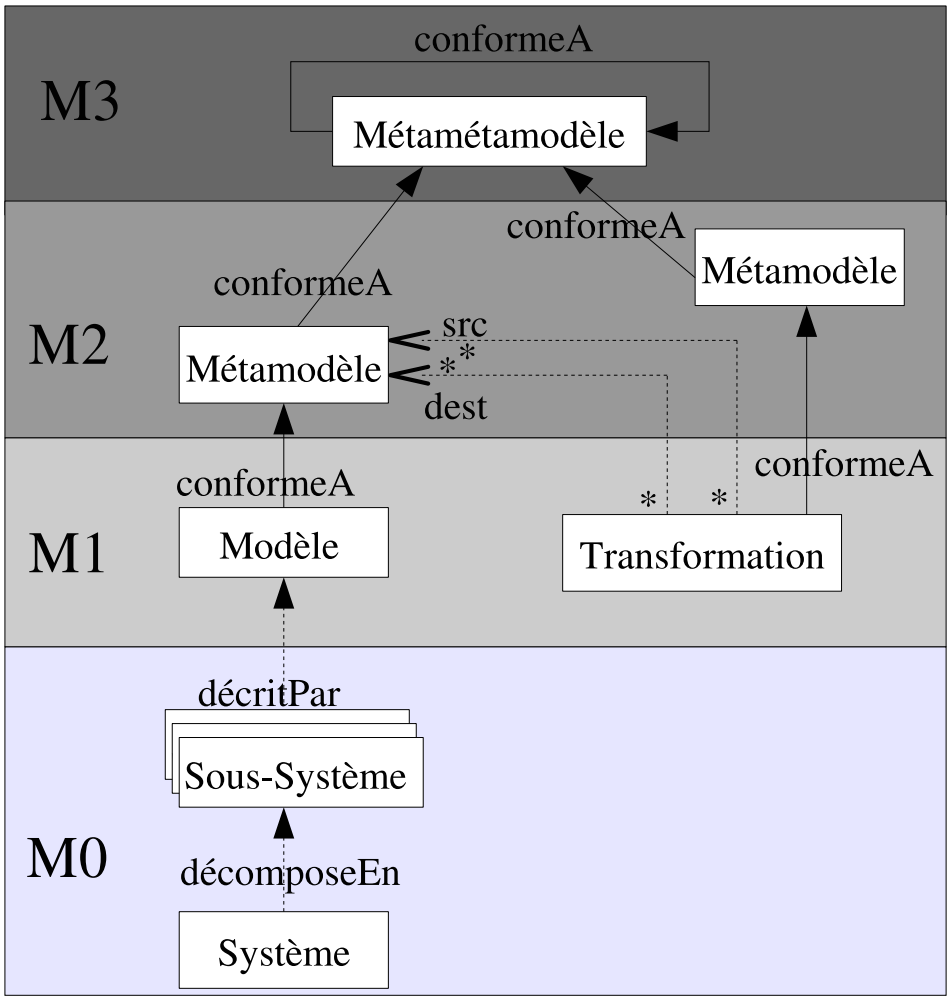

FIG. 1.2 – Concepts et principe de l'IDM

modèles. L'implémentation des différents sous-systèmes, ainsi que le système global se font donc par des séries de transformations sur des instances de modèles.

Cette présentation de l'IDM n'est pas exhaustive, ni normative ; elle comporte seulement les concepts essentiels et stables, ainsi qu'une mise en œuvre de l'IDM d'après notre point de vue.

Après ce bref tour d'horizon sur le principe et les concepts de l'IDM, nous allons dans la section suivante présenter le MDA (Model Driven Architecture) qui est une approche de mise en œuvre de l'IDM dans l'espace des standards OMG.

1.2 Model Driven Architecture (MDA)

L'industrie du logiciel est caractérisée par une constante évolution avec l'émergence de nouvelles technologies. Ces dernières apparaissent et deviennent très vite populaires (par exemple, Java, XML, HTML, SOAP, UML, J2EE, .NET, JSP, ASP, Flash, Web Services). La

plupart des entreprises et compagnies se voient obligées de suivre ces technologies pour les raisons suivantes :

- les clients demandent ces technologies (exemple : les Web Services),
- ces technologies répondent à des besoins (XML pour l'échange d'information, Java pour la portabilité),
- les vendeurs d'outils cessent d'assurer les supports des vieilles technologies et se concentrent sur les nouvelles.

Cette évolution permanente des technologies pose un problème de portabilité des systèmes : comment faire migrer son ancien système vers les nouvelles technologies ?

Pour résoudre certains des problèmes décrits ci-dessus, en novembre 2000, l'*Object Management Group* (OMG[5]) a proposé une approche nommée *Model Driven Architecture* (MDA) [111] pour le développement et la maintenance des systèmes à prépondérance logicielle.

Comme son nom l'indique, l'approche MDA consiste en la définition d'une architecture logicielle basée sur les modèles. Les modèles sont présents dans les différentes phases du processus de développement d'une application. L'idée initiale de l'OMG consistait à décrire séparément les parties des systèmes indépendantes des plates-formes spécifiques (PIM, pour *Platform Independant Models*) et les parties liées aux plates-formes (PSM, pour *Platform Specific Models*). La description des systèmes de façon indépendante de toute plate-forme spécifique permet d'élaborer des modèles pérennes. Comme indiqué dans la section 1.1, le passage du niveau PIM au niveau PSM est réalisé par des transformations automatiques ou semi-automatiques de modèles.

1.2.1 Platform Independant Model (PIM)

La première étape dans la réalisation d'une application informatique consiste à identifier les besoins des utilisateurs. Une fois cette étape réalisée, la phase suivante dite *analyse et conception* vise à structurer l'application en modules et sous-modules. Dans le cycle de développement des logiciels, l'analyse et la conception sont généralement caractérisées par une forte activité de modélisation.

Dans l'approche MDA, les modèles d'analyse et de conception doivent être indépendants de toute plate-forme d'implémentation comme J2EE, .Net, PHP, etc. En effet, la prise en compte très tardive des détails d'implémentation permet de maximiser la séparation des préoccupations entre la logique de l'application et les techniques d'implémentation. C'est pourquoi dans la terminologie MDA, ces modèles sont appelés des PIM (Platform Independant Model). Par ailleurs, ces modèles doivent être productifs et contenir toutes les informations nécessaires pour qu'une génération automatique de code soit envisageable.

1.2.2 Platform Specific Model (PSM)

Le modèle PIM obtenu à l'issue de la phase d'analyse et conception ne contient qu'une spécification fonctionnelle de l'application. Pour une réalisation concrète, une plate-forme

[5]L'OMG (Object Management Group) est un consortium, à but non lucratif, d'industriels et de chercheurs, dont l'objectif est d'établir des standards permettant de résoudre les problèmes d'interopérabilité des systèmes d'information (*http ://www.omg.org*)

d'implémentation doit être choisie. Le choix de la plate-forme va permettre de définir comment le modèle PIM précédemment spécifié va être réalisé au moyen de la plate-forme. MDA propose un modèle dit spécifique à la plate-forme qui complète le modèle PIM en intégrant les informations, détails et contraintes techniques propres, et une solution de réalisation de l'application sur ladite plate-forme (d'où l'appellation PSM : Platform Specific Model). Ces modèles servent à faciliter la génération de code. En effet, MDA considère que le code d'une application peut être facilement obtenu à partir d'un modèle comportant à la fois la spécification fonctionnelle et les informations de la plate-forme d'exécution. Ainsi, les modèles PSM sont essentiellement productifs, mais non pérennes.

Pour élaborer des modèles PSM, MDA propose, entre autres, l'utilisation des profils[6] UML [19]. Par exemple, le profil UML pour le temps réel (RT-UML) est une adaptation de UML au domaine du temps réel. Grâce à ce profil, il est possible de réaliser des modèles pour le développement d'une application temps réel.

Avec MDA, l'activité de développement des applications est donc centrée sur la conception d'un ensemble de modèles et de transformations de modèles. Pour cela, un formalisme de modélisation est indispensable pour pouvoir exprimer ces différents modèles. Pour répondre à ce besoin, MDA propose une architecture de modélisation à quatre niveaux conforme à la figure 1.2. Au plus bas niveau de cette architecture se trouvent les éléments réels de l'application que l'on modélise ; c'est le niveau des objets réels appelé niveau *M0*. Ces objets réels sont modélisés à travers des concepts définis dans un certain langage (Exemple : une classe *Client* dans le langage de modélisation UML). Ce langage se situe au niveau *M1*, le niveau des modèles. Le modèle du niveau M1, sera exprimé à son tour, dans un langage appelé *métamodèle* représenté par le niveau *M2*. Cette chaîne de métamodélisation s'achève au niveau *M3* par la définition d'un *métamétamodèle* dénommé *MOF (Meta Object Facility)* qui se définit lui-même à l'aide de ses propres concepts et qui devrait permettre de décrire n'importe quel langage de modélisation. Cette spécification du MOF est actuellement à sa version 2.0.

Par ailleurs, MDA propose un langage dénommé *MOF2.0 QVT (MOF2.0 Query, View, Transformation)* pour définir les transformations de modèles. L'idée de définir un langage de définition des transformations est motivée par une application du principe de MDA aux transformations. QVT permettra donc d'avoir un langage unique d'expression des transformations, et cela permettra de rendre les définitions de transformations pérennes au même titre que les métamodèles de domaine. Avec QVT, les transformations de modèles dans MDA peuvent alors être schématisées comme le montre la figure 1.3.

Nous reviendrons sur la spécification MOF2.0 QVT dans le chapitre 2.

Synthèse

En résumé, MDA définit un cadre de développement de logiciels dans lequel des modèles abstraits sont utilisés pour décrire les fonctionnalités des applications en faisant asbstraction de tout détail technique d'implémentation. Ces modèles dits PIM sont ensuite enrichis par des informations de plate-forme d'exécution et de choix technologiques d'implémentation pour obtenir des modèles PSM. Ces derniers contiennent toutes les informations nécessaires

[6] Un profil UML est une adaptation du langage UML à un domaine particulier.

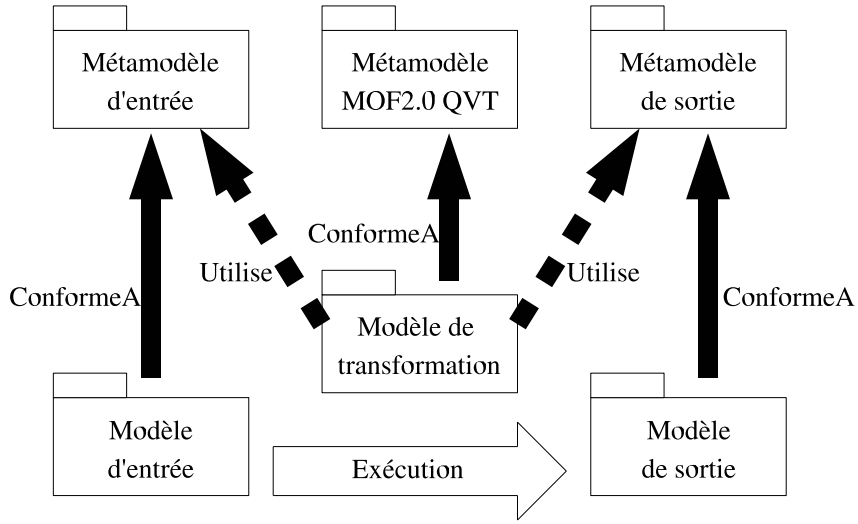

FIG. 1.3 – Transformations de modèles dans MDA

pour pouvoir réaliser une génération automatique du code de l'application. Le passage entre les différentes phases de modélisation est réalisé par des transformations de modèles. L'architecture de base MDA peut être schématisée comme l'indique la figure 1.4.

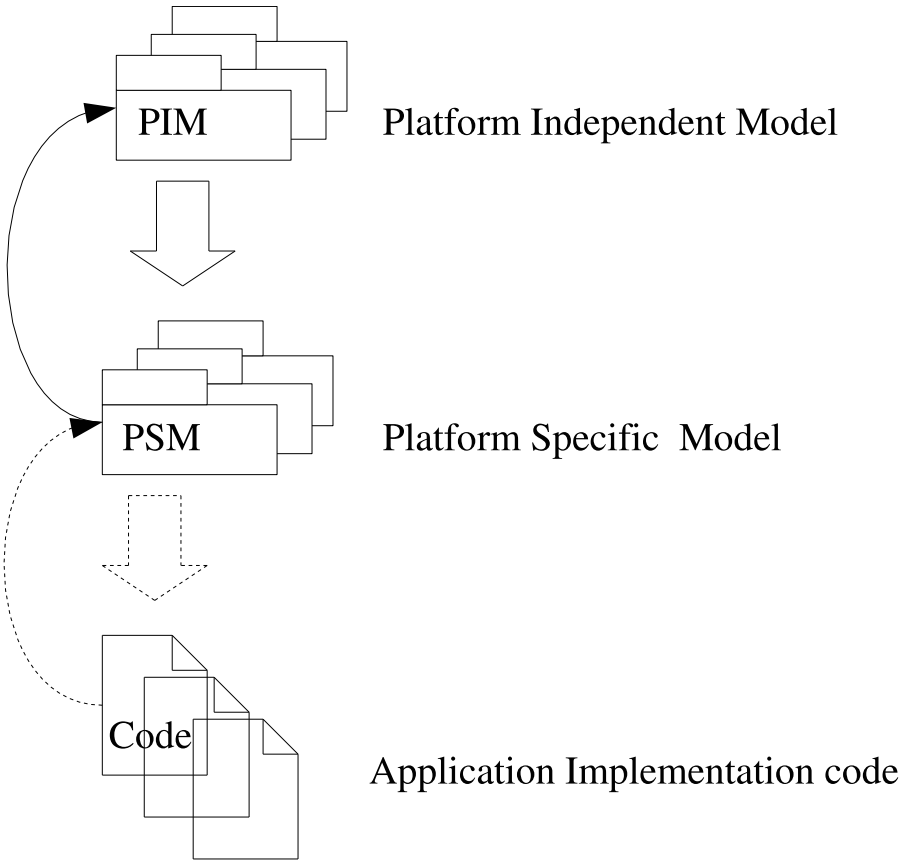

FIG. 1.4 – Aperçu de la méthodologie MDA

MDA est donc la définition d'un scénario de mise en œuvre de l'IDM basé sur les standards de l'OMG, à savoir :

- le langage UML pour décrire les modèles PIM,
- les profils UML pour les modèles PSM,
- le langage MOF pour la métamodélisation,
- le langage MOF 2.0 QVT pour les transformations de modèles.

Au-delà du MDA, il existe d'autres approches de mise en œuvre de l'IDM. Chez Microsoft, l'IDM est mise en œuvre à travers les *software factories* (fabriques de logiciels) [10], [57], [114, p 68-69]. Sa vision est fondée sur l'utilisation de langages de domaines (DSL pour *Domain Specific languages*). Ces langages sont de petite taille, facilement manipulables, transformables, combinables et de ce fait, sont la base de l'automatisation de l'IDM chez Microsoft. Une intégration progressive de la vision de Microsoft est en train d'être mise en œuvre dans Visual Studio. IBM partage également l'idée des DSL et organise l'IDM autour de l'outillage EMF (*Eclipse Modeling Framework*). Il adopte l'architecture multiniveaux de l'OMG fondée sur la pile de métamodélisation dominée par le MOF tout en gardant la possibilité de définir des métamodèles spécialisés pour les DSL.

1.3 Conclusion

L'IDM ouvre de nouvelles voies d'investigation. Elle vise non seulement à favoriser un « génie » logiciel plus proche des métiers en autorisant une appréhension des applications selon différents points de vue (modèles) exprimés séparément, mais elle intègre également, comme fondamentales, la composition et la mise en cohérence de ces perspectives. De plus, elle se veut productive en automatisant la prise en charge des outils relatifs à la validation des modèles, aux transformations et aux générations de code.

L'IDM peut donc être vue comme une famille d'approches qui se développent à la fois dans les laboratoires de recherche et chez les industriels impliqués dans les grands projets de développement de logiciels. Ainsi, avec Eclipse, IBM se base sur Ecore plutôt que sur le MOF, Microsoft se concentre principalement sur des technologies XML et l'approche DSL. Il semble donc acquis que les techniques de l'IDM feront partie intégrante des outils et approches de développement des logiciels de demain et vont profondément influencer les pratiques du génie logiciel.

Dans ce chapitre, nous avons présenté l'approche de l'ingénierie dirigée par les modèles. Son principe et ses principaux concepts ont été introduits à partir de trois notions essentielles : **l'interprétation des modèles par les machines**, **la productivité des modèles** et **la séparation des différents aspects d'un système**. Cette présentation reste incomplète car nous n'avons pas abordé le sujet fondamental des transformations de modèles. Le prochain chapitre sera entièrement consacré à cette question.

Chapitre 2

Transformations de modèles

Les transformations de modèles sont au cœur de l'approche de l'ingénierie dirigée par les modèles. Elles répresentent l'un des grands défis à relever d'un point de vue technique pour envisager une large diffusion de l'ingénierie dirigée par les modèles. Le principe de transformations de modèles est assez largement accepté et intéresse aussi bien les chercheurs que les industriels. Mais il n'existe pas encore de consensus sur la définition et la mise en œuvre d'une transformation. Dans la littérature, de multiples approches sont proposées parmi lesquelles il n'est pas toujours évident de choisir.

Pour guider les développeurs dans le choix des approches et outils de transformations, Tom Mens et autres [89] (*A Taxonomy of Model Transformations*) ont réalisé une étude qui définit des critères objectifs qui permettent d'une part, de réaliser une taxonomie des approches de transformations, et d'autre part, de pouvoir choisir une approche en fonction de ses besoins. Leur étude fait apparaître un certain nombre de termes et définitions importants que nous allons présenter dans la section suivante. Après cette première section de définitions et clarification de termes, une seconde section présente les différentes approches et outils de transformations de modèles. Enfin, la dernière section de ce chaptire est consacrée à la spécification MOF2.0 QVT, le standard OMG pour la définition des transformations de modèles.

2.1 Quelques définitions et clarification de termes

Pour réaliser des transformations de modèles, ces derniers doivent être exprimés dans un certain langage (par exemple, UML pour des modèles de conception, Java pour les modèles de code source). Ce langage de modélisation étant lui-même défini à l'aide d'un métamodèle. En partant des métamodèles sources et cibles de la transformation, on distingue deux types de transformations : les transformations *endogènes* et *exogènes*. Une transformation est dite endogène si les modèles impliqués sont issus du même métamodèle. Mais lorsque les modèles sources et cibles sont de différents métamodèles, la transformation est dite exogène ou encore *translation*. Ces deux catégories de transformations peuvent être subdivisées comme suit :

- transformations endogènes :
  - *Optimisation* - transformation dont le but est d'améliorer les performances tout en maintenant la sémantique,

  - *Refactoring* - transformation qui opère un changement dans la structure pour améliorer certains aspects de la qualité du logiciel tels que la compréhension, la maintenance, la modularité et la réutilisation sans en changer le comportement observable,
  - *Simplification ou normalisation* - transformation dont le but est de réduire la complexité syntaxique.
- transformations exogènes :
  - *Synthèse* - transformation d'un certain niveau d'abstraction vers un niveau d'abstraction moins élevé. Un exemple typique est la génération de code,
  - *Rétro-ingénierie* - inverse de la synthèse,
  - *Migration* - transformation d'un programme écrit dans un langage vers un autre langage du même niveau d'abstraction.

Un autre facteur important à prendre en considération dans les transformations concerne le niveau d'abstraction. Selon le niveau d'abstraction, on distingue les transformations *horizontales* et les transformations *verticales*. Une transformation horizontale est une transformation où les modèles sources et cibles sont du même niveau d'abstraction. A l'opposé, dans une transformation verticale les modèles impliqués sont de différents niveaux d'abstraction. Un exemple typique de transformation verticale est le *raffinement*.

Enfin, dans une transformation, les modèles sources et cibles peuvent ou non appartenir à un même *espace technologique*. Un espace technologique est composé d'un ensemble de concepts, d'un corps de connaissance, d'outils, de compétences, etc., définissant un contexte opérationnel de travail [31]. Par exemple, le XML, MDA, DBMS (Data Base Management System) sont des espaces technologiques. Lorsqu'une transformation implique plusieurs espaces technologiques, des outils d'importation/exportation sont nécessaires pour faire le pont entre ces différents espaces.

2.2 Approches et outils de Transformations

Comme nous l'avons déjà mentionné, les transformations de modèles constituent un aspect clé dans la démarche de développement des logiciels basés sur l'ingénierie dirigée par les modèles. Le succès de cette technologie repose en grande partie sur les transformations de modèles. Leur application (ou utilisation) couvre plusieurs aspects dont :

- la génération de modèles de plus bas niveau et éventuellement du code à partir de modèles de plus haut niveau,
- la synchronisation de modèles de même niveau d'abstraction ou non,
- la rétro-ingénierie de modèles de haut niveau à partir du code ou de modèles de plus bas niveau d'abstraction,
- etc.

Au regard de l'importance que revêtent les transformations de modèles, un intérêt particulier a été initié par l'OMG pour un effort de standardisation. La RFP (Request for Proposal) a été issue en avril 2002, aboutissant à l'élaboration du standard QVT (Query View Transformation). Mais avant cette adoption de QVT, plusieurs approches de transformations de modèles plus ou moins inspirées de la RFP de QVT sont apparues. Le dénominateur commun à toutes ces approches est le principe et les concepts de base des transformations présenté sur la figure 2.1.

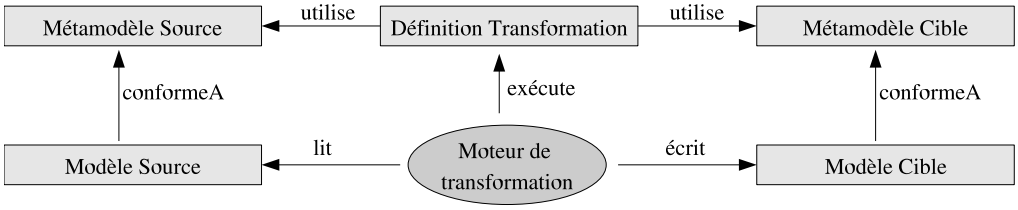

FIG. 2.1 – Concepts de base des transformations de modèles

Au-delà du principe et des concepts de base, les approches varient beaucoup, surtout dans la mise en œuvre qui est souvent faite de façon plus ou moins ad hoc. En se basant sur :

- les publications (littérature) dans le domaine,
- les soumissions en réponse au RFP sur QVT de l'OMG,
- les implémentations d'outils MDA en open-source,
- les implémentations d'outils MDA commerciaux,

Krzystof Czarnecki et Simon Helsen [71], [72] ont publié une classification des différentes approches de transformations de modèles. Cette classification est faite selon un certain nombre de critères que nous présentons ci-dessous.

2.2.1 Critères de classification des approches

Les critères de classification retenus par Krzystof Czarnecki et Simon Helsen [71], [72] se résument dans le tableau 2.1.

Critères	**Significations**
Spécification	Certaines approches fournissent un mécanisme dédié pour spécifier les transformations
Règles de transformation	Unités de base de l'expression de transformation
Contrôle de l'application des règles	Stratégies de localisation des modèles et de détermination de l'ordre d'exécution des règles de transformation
Organisation des règles	Structuration, modularité et réutilisation des règles
Relations entre source et cible	Concerne l'identité des modèles sources et cibles ; sont-ils différents ou non ?
Incrémentation	Possibilité de mise à jour des modèles cibles lorsque les modèles sources correspondants changent
Directivité ou réversibilité	Détermine si la transformation est unidirectionnelle ou multi-directionnelle
Traçabilité	Mécanisme d'enregistrement des liens entre éléments de modèles sources et éléments des modèles cibles

TAB. 2.1 – Critères de classification des approches de transformations de modèles

Dans les lignes qui vont suivre, nous donnons quelques détails sur les critères les plus importants.

Règles de transformation

Une règle de transformation contient deux parties : une partie gauche (left-hand side « LHS ») et une partie droite (right-hand side « RHS »). La LHS exprime des accès aux modèles sources, alors que la RHS indique les expansions (création, modification, suppression) dans les modèles cibles. Chacune des deux parties peut être représentée par une combinaison de :

- *Variables* : une variable contient un élément de modèle (source ou cible) ou une valeur intermédiaire nécessaire à l'expression de la règle,
- *Patterns* : un pattern désigne un fragment de modèle et peut contenir des variables. Il peut être représenté à l'aide d'une syntaxe abstraite ou concrète dans le langage des modèles correspondants. Cette syntaxe peut être textuelle ou graphique,
- *Logique* : une logique permet d'exprimer des calculs et des contraintes sur les éléments de modèles. Cette expression peut être non exécutable (expression de relations entre modèles) ou exécutable. Une logique exécutable peut être sous une forme déclarative ou impérative. Un exemple de logique déclarative : des expressions de requêtes OCL pour extraire des éléments d'un modèle source et la création implicite des éléments du modèle cible via les contraintes. Une logique impérative est souvent exprimée par un langage de programmation qui fait appel à des APIs pour manipuler directement les modèles.

Par rapport aux règles de transformation, les différentes approches de transformation de modèles se distinguent selon les quatre aspects suivants :

- *Séparation syntaxique* : les parties LHS et RHS des règles peuvent être ou non séparées syntaxiquement. C'est-à-dire que dans l'écriture de la règle, on peut voir apparaître clairement la LHS et la RHS (cas classique des règles de réécriture), mais cette séparation peut ne pas être faite (cas d'une règle de transformation implémentée dans un programme Java),
- *Exécution bidirectionnelle* : une règle peut être exécutée dans un sens ou dans les deux sens,
- *Paramétrage de règles* : une règle de transformation peut contenir des paramètres supplémentaires pour permettre une certaine configuration ou adaptation,
- *Structures intermédiaires* : certaines approches ont besoin de construire des structures de modèles intermédiaires.

Relations entre modèles sources et modèles cibles

Selon les relations entre les modèles sources et les modèles cibles, on distingue principalement trois types d'approches :

- le modèle cible est un nouveau modèle différent, créé à partir du modèle source,
- le modèle cible est le même que le modèle source sur lequel des modifications ont été faites,
- une troisième classe d'approche combine les deux premières. Le modèle source peut être un nouveau modèle ou un modèle déjà existant (éventuellement le modèle source).

Contrôle de l'application des règles

L'ordonnancement des règles consiste à définir dans quel ordre les différentes règles de transformations sont utilisées. Les mécanismes d'ordonnancement de règles diffèrent suivant quatres points :

- *La forme* : l'ordonnancement peut être implicite ou explicite. Dans les ordonnancements implicites, l'utilisateur n'a pas de contrôle sur l'algorithme d'ordonnancement qui est entièrement défini par l'outil de transformation. Il ne peut que définir ses patterns et la logique des règles de façon à garantir l'ordre d'exécution qu'il souhaite avoir. Les ordonnancements explicites, quant à eux, ont des constructions qui permettent de contrôler de façon explicite l'ordre d'exécution,
- *La sélection des règles* : elle peut se faire par des conditions explicites. Certaines approches autorisent des choix non déterministes ; elles définissent alors un mécanisme de résolution des conflits,
- *L'itération sur une règle* : elle peut être soit une récursivité, soit une boucle, soit encore une *itération à point fixe*. Cette itération consiste à appliquer la même règle sur le modèle en entrée jusqu'à ce qu'aucun changement ne se produise,
- *La décomposition en phases* : Une transformation peut être découpée en plusieurs phases où chaque phase a un objectif précis. A chaque phase, seul un ensemble restreint de règles sont applicables.

Organisation des règles

Par organisation des règles, il faut entendre composition et structuration de plusieurs règles de transformations. De ce point de vue, on distingue trois points de variation dans les approches :

- *Modularité* : il s'agit de regrouper les règles dans des modules. Un module pourra alors importer d'autres modules et, ainsi, avoir accès à leur contenu,
- *Réutilisation* : le mécanisme de réutilisation permet de définir des règles à partir d'autres règles. Certaines approches offrent un mécanisme d'héritage entre règles,
- *Structure organisationnelle* : les règles de transformation peuvent être organisées suivant la structure du langage des modèles sources ou des modèles cibles ou encore de façon indépendante des langages des modèles.

Traçabilité

La traçabilité dans les transformations consiste à créér et enregistrer des liens entre les éléments des modèles cibles et ceux des modèles sources. Certaines approches de transformations n'offrent pas de mécanisme de traçabilité ; il appartient alors à l'utilisateur de trouver un moyen pour créér et gérer les liens de trace. D'autres, en revanche, fournissent des moyens dédiés pour créer et gérer ces liens. Parmi ces dernières, certaines approches laissent la charge à l'utilisateur d'encoder manuellement dans les règles de transformation la création des liens et d'autres prennent en charge de façon automatique la traçabilité. Un autre élément de distinction entre les outils qui supportent la traçabilité réside dans le stockage

des informations de la trace. Suivant les approches, ce stockage se fait soit dans le modèle source, soit dans le modèle cible ou dans un modèle à part.

Directivité ou réversibilité

Les transformations peuvent être unidirectionnelles ou bidirectionnelles. Les transformations unidirectionnelles s'exécutent dans un seul sens : le modèle cible est produit ou modifié à partir du modèle source. Les transformations bidirectionnelles s'exécutent dans les deux sens. Dans ce cas, les règles de transformations sont alors soit bidirectionnelles ou constituées de deux jeux de règles unidirectionnelles dont chacun permet la transformation dans un sens.

La prise en compte de ces différents critères de classification permet d'identifier plusieurs approches dont nous présentons ci-dessous les principales catégories.

2.2.2 Principales catégories

Au plus haut niveau de la classification, on distingue deux approches de transformations : les transformations de « modèles à modèles » et les transformations de « modèles vers du code source » (génération de code). Les transformations de modèles vers du code source peuvent être vues comme un cas particulier de transformation de modèles à modèles où le métamodèle du modèle cible est la grammaire d'un langage de programmation. Cependant, pour des raisons pratiques (réutilisation de compilateurs existants), il est intéressant de considérer ces deux catégories séparément.

Dans chacune de ces deux catégories, on distingue plusieurs sous-catégories comme le montre la figure 2.2.

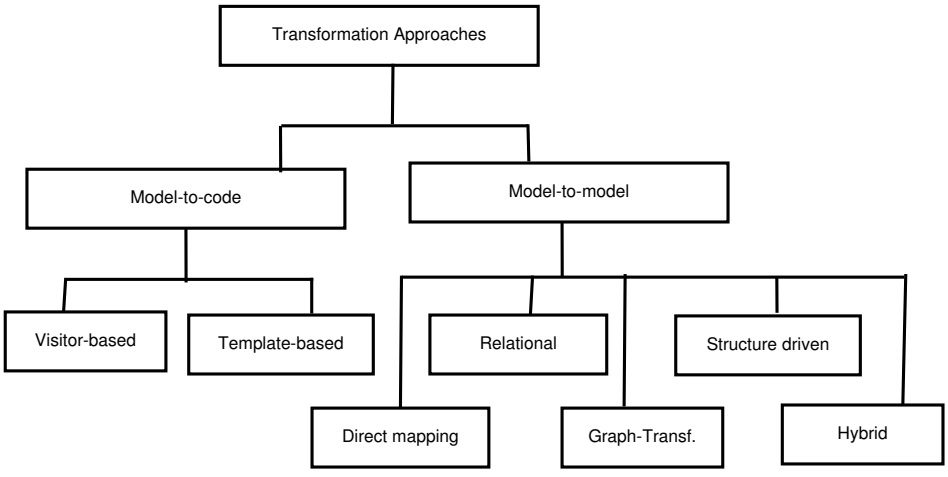

FIG. 2.2 – Approches de transformations de modèles

Approches modèles à modèles

Dans les transformations de modèles à modèles, les modèles sources et cibles peuvent être ou non des instances d'un même métamodèle. Dans cette catégorie, on distingue : les approches qui offrent des mécanismes pour manipuler directement les modèles, les approches relationnelles, les approches basées sur les transformations de graphes, les approches dirigées par la structure et les approches hybrides.

Les approches manipulant directement les modèles

Dans ces approches, une API permet de manipuler la représentation interne des modèles. Elles sont en général implémentées comme un framework orienté objet qui fournit une infrastructure pour organiser les transformations (par exemple, la définition de classes abstraites pour les transformations). Les utilisateurs doivent eux-mêmes réaliser les règles de transformations au moyen d'un langage de programmation comme Java. La combinaison JMI (Java Metadata Interface) et Java est souvent utilisée dans la mise en œuvre de cette approche. JMI est une API basée sur le MOF et permet de créer, sérialiser, accéder aux éléments d'un modèle défini à l'aide du MOF ou ECore.

Les approches relationnelles

Le principe de base de cette approche consiste à établir une relation entre les éléments des modèles sources et cibles. Ces relations seront spécifiées à l'aide de contraintes. Elles sont purement déclaratives et leur spécification n'est pas exécutable. La programmation logique avec les principes d'unification, de recherche et de backtracking constitue un moyen naturel d'implantation de cette approche. Les réponses [104, 33] à la RFP MOF 2.0 QVT sont basées sur ces approches.

Les approches basées sur la transformation de graphes

Les modèles et métamodèles possèdent souvent une représentation graphique apparentée à un graphe. Un modèle, dans ce cas, peut être considéré comme un graphe étiqueté, contraint par des règles de cohérence essentiellement définies comme un métamétamodèle MOF. Les techniques de réécriture de graphes et de transformation de graphes peuvent être appliquées pour des transformations de modèles. D'une manière générale, les systèmes de réécriture de graphes combinent une notation graphique et une notation textuelle afin d'exprimer ces transformations. Un programme de transformation essentiellement composé de règles de réécriture va d'abord sélectionner un fragment d'un graphe source identifié grâce à un langage de navigation. L'application d'un filtre sur ce fragment sélectionné permet de le modifier avant de le recopier dans le graphe cible.

Dans ce cas, la problématique de recherche d'un sous-graphe à transformer est un problème *NP complet* (la complexité est exponentielle selon la taille du sous-graphe à détecter). Il est, dès lors, difficile de disposer d'une solution industrielle satisfaisante à court terme pour les transformations basées sur cette approche. Cette catégorie d'approches est mise en œuvre dans : VIATRA [37], UMLX [43] et BOTL [88] par exemple.

Les approches dirigées par la structure

Dans ces approches, la transformation se réalise en deux phases : la première crée la structure

hiérarchique du modèle cible et la seconde vient compléter le modèle en définissant les valeurs des attributs et des références. Comme exemples d'approches de cette catégorie, on cite : OptimalJ [36], Interactive Objects and Project Technology (IOPT) [95].

Les approches hybrides

Elles combinent plusieurs des approches précédentes. Le langage de transformation de règles est une combinaison d'approches déclarative et impérative. Une combinaison se traduit par des :

- *Mapping rules* qui définissent les relations entre les éléments sources et cibles,
- *Operational rules* qui définissent les règles exécutables (réalisation de la transformation). Les actions (création, modification, suppression) dans ces règles sont précisées.

ATLAS (ATL) [2], Kermeta [6] et ModTransf [42] sont des approches de cette catégorie.

Approches modèles vers code

Dans cette catégorie, on distingue les approches basées sur le parcours de modèles et celles basées sur l'utilisation de templates. Un template, littéralement appelé « patron » ou « gabarit », est une chaîne de caractères servant à formater une entrée ou une sortie. Le texte ci-dessous est un exemple de template.

```
le_tmpl = """
private  %(type) %(name);
public %(type) %(name)( ){
   return this.%(name);
}
public void %(name)(%(type) val) {
return this.%(name)=val;
}
"""
```

Si l'on considère la définition de la fonction *convert* et son appel décrits comme suit :

```
 def convert(tmpl, values) : return tmpl%values

 convert(le_tmpl, {'type' : 'String'; 'name' : 'adresse'}
```

on aura en sortie la chaîne de caractères suivante :

```
"
private String  adresse;
public String adresse( ) {
return this.adresse;
}
public void adresse(String val) {
return this.adresse = val;
}
"
```

Les approches basées sur le parcours de modèles

Ces approches définissent un mécanisme de parcours de la représentation interne d'un modèle et l'écriture du code (texte) dans un flux en sortie. Un exemple utilisant cette approche est Jamda [8], un framework orienté objet qui définit un ensemble de classes pour représenter les modèles UML, une API pour manipuler ces modèles, et un mécanisme pour générer du code.

Les approches basées sur les templates

Cette catégorie d'approches est la plus couramment utilisée dans les outils MDA actuels de génération de code. Un template consiste en un fragment de texte contenant des bouts de métacode permettant :

- d'accéder aux modèles sources,
- d'effectuer une sélection de code,
- de réaliser des expansions itératives.

La structure du template est généralement très proche du code à générer. Comme exemples dans cette catégorie, on peut citer : JET [5], Codagen Architect [3], OptimalJ [36] et ModTransf [42].

Synthèse

Cette classification faite par Czarnecki K. et Helsen S. est certes très intéressante, mais on peut noter le fait que cette dernière est plutôt centrée sur les techniques de mise en œuvre des transformations que sur leur conception. À mon avis, quelle que soit l'approche utilisée, le plus important reste la démarche de conception des transformations. De ce point de vue, on distinguerait deux approches de transformations : celles basées sur la métamodélisation (la définition de la transformation est faite sur la base de métamodèles, ce qui permet de les pérériser), et les autres.

2.2.3 Outils de transformations

On compte aujourd'hui plus d'une centaine d'outils, toutes catégories confondues (open source, commerciaux, moteurs de transformation, générateurs de code, outil de modélisation UML, etc.), qui se réclament être compatibles MDA/MDE.

Naveed Ahsan Tariq et Naeem Akhter [117] ont réalisé une comparaison d'outils MDA. Dans leur travail, ils ont défini un ensemble de critères d'évaluation sur la base des différents aspects du MDA. Un système de scores leur a permis d'attribuer une note à chaque outil et pour chaque aspect, ce qui permet aux développeurs MDA de choisir leurs outils en fonction des besoins de leurs projets.

Les principales conclusions de ces travaux se résument comme suit :

- En termes de *modélisation*, UML est largement supportée par les outils, mais il s'agit en général des anciennes versions d'UML : 1.3, 1.4 et souvent quelques aspects d'UML 2.0. Ce support d'UML concerne surtout les *diagrammes de classes* et quelques outils qui prennent en charge les *cas d'utilisations* (*use cases*),

- En ce qui concerne les transformations, la situation est encore plus dramatique. En effet, très peu d'outils fournissent une infrastructure permettant d'étendre des transformations ou de définir ses propres transformations. La portabilité et l'interopérabilité dans les transformations sont quasiment absentes. Chaque outil définit et utilise son propre langage de script pour définir ses transformations ; d'où le problème de compatibilité entre les outils. Mais on peut espérer qu'avec l'adoption de la version finale de QVT [96], ce problème sera bientôt résolu.

Même s'il existe une multitude d'outils, avec des approches variées, du point de vue de l'architecture fonctionnelle, un outil de transformation de modèles comporte généralement deux principaux blocs fonctionnels :

- un moteur de transformation qui permet d'effectuer toutes les générations (IDL, Corba, DTD, Java, Idlscript, etc.) et transformations nécessaires,
- un référentiel qui permet de stocker les métamodèles.

Les métamodèles et les modèles, y compris les modèles définissant les transformations sérialisées sous des fichiers au format XML, sont exportés ou importés dans le ou les référentiel(s) via des modules d'import/export. Un service de transformation composé d'APIs permet au moteur de transformation de pouvoir lire et écrire dans le ou les référentiel(s).

2.3 QVT, le standard OMG

L'importance des transformations dans les approches dirigées par les modèles se révèle à travers de nombreuses tentatives de mise en œuvre des transformations aussi bien dans le milieu académique, que dans les communautés open sources ou l'industrie.

Dans « MDA Guide » [97], l'OMG parle de transformations entre des modèles de différents niveaux d'abstraction. Son scénario typique de transformations étant les transformations d'un PIM vers un PSM, ce dernier pouvant être utilisé pour la génération de code. Mais ce document (MDA Guide) ne stipule pas comment ces transformations doivent être réalisées. QVT (Queries Views and Transformations), dont la version finale est maintenant adoptée [96], vient en complément du MDA Guide et propose un standard pour les transformations de modèles.

Ce standard a été complètement immergé dans MOF et comporte trois langages (métamodèles) de transformations : *Relations*, *Operational Mappings*, et *Core*.

Deux des trois langages, Relations et Core, sont des langages déclaratifs. Le langage Operational Mappings, quant à lui, est impératif. Cette nature hybride (déclarative et impérative) a été introduite pour satisfaire les besoins et habitudes des différents utilisateurs.

2.3.1 Partie déclarative

La spécification déclarative est structurée dans une architecture à 2 niveaux.

Le langage de Relations est construit autour du concept de *patterns d'objets*. Une transformation est décrite dans ce langage comme un ensemble de *patterns d'objets*. Ces patterns peuvent être mis en correspondance avec des éléments de modèles, instanciés pour modéliser de nouveaux

éléments de modèles ou être utilisés pour appliquer des changements dans des modèles déjà existants. Le langage supporte de façon automatique la création et la suppression des objets, ainsi que la gestion des informations de trace des transformations. L'utilisateur n'a plus qu'à décrire les relations entre les éléments sources et cibles des métamodèles au moyen de patterns d'objets et d'expressions OCL [99]. Le langage fournit également un mécanisme pour identifier les éléments des modèles cibles et définit une syntaxe graphique simple pour exprimer les relations.

Le langage Core est une extension minimale de EMOF et OCL. Dans ce langage, tout comme dans le premier, l'utilisateur ne s'occupe ni de la création ni de la suppression des objets. En revanche, il doit gérer lui-même la trace des transformations. Il définit ses règles de transformations et les informations de trace à l'aide d'un métamodèle MOF. Le langage Core ne supporte pas de mécanisme de patterns ni d'identification des éléments des modèles cibles.

L'absence de mécanisme de trace et d'identification des éléments rend ce langage simple, mais difficilement exploitable, en pratique.

2.3.2 Partie impérative

Le langage *Operational Mappings* constitue la partie impérative de QVT. Il fournit un langage impératif spécifique de domaine pour décrire les transformations. L'OCL est utilisé comme le langage des requêtes avec une extension de celui-ci pour introduire des effets de bord et une syntaxe concrète, familière à la programmation impérative.

QVT propose deux façons d'utiliser le langage *Operational Mappings*. Premièrement, il peut être utilisé pour spécifier une transformation uniquement dans le langage. Deuxièmement, il peut être utilisé de façon hybride. Dans ce cas, l'utilisateur peut spécifier certains aspects de la transformation dans un des langages déclaratifs (Relations ou Core) et implémenter, en boîte noire, des règles dans le langage *Operational Mappings*. C'est ce qui explique la partie *Black-Box* sur la figure 2.3.

Les spécifications déclaratives des langages *Relations* et *Core* peuvent être implantées de deux façons : d'une façon standard au moyen des *Operational Mappings*, ou d'une façon non standard, *Black-Box MOF Operation*. Le langage de *Operational Mappings* est donc le moyen standard d'implémentation des spécifications déclaratives.

Par ailleurs, les *Black-Box MOF Operations* sont des opérations MOF qui peuvent être dérivées des relations et permettent de « plugger » toute implémentation d'une opération MOF ayant la même signature. Cela présente de nombreux avantages :

- la possibilité de coder des algorithmes complexes dans un langage de programmation avec un binding MOF,
- l'utilisation de librairies spécifiques pour calculer les valeurs des propriétés des modèles. En effet, dans les domaines des mathématiques, de l'ingénierie, de la biologie, etc., il existe des librairies spécifiques d'algorithmes qui ne peuvent être exprimés au moyen d'OCL,
- la possibilité d'avoir des implémentations opaques de certaines parties d'une transformation.

2.3.3 Relations entre les trois langages QVT

Les relations entre les trois langages QVT sont représentées à l'aide de la figure 2.3.

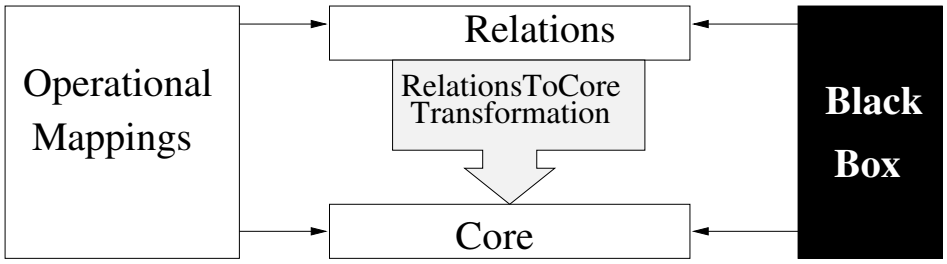

FIG. 2.3 – Architecture des langages QVT

Les règles de transformations décrites à l'aide des langages *Relations language* et *Core Language* peuvent être soit implémentées à l'aide du langage *Operational Mappings Language* ou implémentées en boîte noire, *Black Box*, dans un langage de programmation ou encore fournies par une librairie externe.

La transformation explicite entre le *Relations Language* et le *Core Language* suggère que les transformations décrites dans le Relations Language peuvent être implémentées dans le Core Language à l'aide d'un outil par exécution de la transforamtion *RelationsToCore Transformation*. Cette stratégie est expliquée dans la spécification en faisant une analogie avec la plateforme Java. Le Relations Language serait l'équivalent du langage Java. Le Core Language correspondrait au Java Byte Code et sa sémantique à la spécification du comportement de la Java Virtual Machine (JVM). La transformation *RelationsToCore Transformation* ferait office de compilateur Java. Cette spécification, bien que théoriquement possible, est plus difficile à mettre en œuvre et même inefficace. Il est plus pratique de développer directement un moteur pour le langage de Relations Language que de passer par ce détour.

2.3.4 Implémentation d'outils QVT

La spécification QVT laisse une certaine marge d'implémentation des outils compatibles QVT. Chaque constructeur d'outil a une flexibilité de mise en œuvre de QVT. Cette flexibilité est exprimée sous deux axes comme le montre le tableau 2.2.

La dimension *Language Dimension* définit trois niveaux correspondant aux langages QVT. Un outil est dit compatible avec un langage s'il est capable d'exécuter une transformation écrite dans ce langage.

La dimension *Interoperability Dimension* traite du format d'expression de la définition de la transformation. Elle définit quatre niveaux :

- **Syntax Executable** : l'outil peut lire et exécuter des définitions de transformations écrites dans la syntaxe concrète donnée de la spécification QVT,
- **XMI Executable** : l'outil peut lire et exécuter des définitions de transformations sérialisées au format XMI,

		Interoperability Dimension			
Language Dimension		Syntax Executable	XMI Executable	Syntaxe Exportable	XMI Exportable
	Core				
	Relations				
	Operational Mappings				

TAB. 2.2 – Critères de compatibilité des outils QVT

- **Syntax Exportable** : l'outil peut exporter des définitions de transformations écrites dans la syntaxe concrète du langage,
- **XMI Exportable** : l'outil peut exporter des définitions de transformations en XMI.

2.3.5 Synthèse

La spécification QVT étant maintenant adoptée, on peut espérer dans les mois à venir l'émergence d'outils QVT. En revanche, l'utilisation de trois langages pour décrire les transformations sera, de mon point de vue, source de problèmes. La spécification stipule que chaque outil devra préciser lequel ou lesquels des trois langages il utilise, et pour chaque langage, préciser s'il sait importer ou exporter une spécification soit dans sa syntaxe abstraite (au moyen de XMI par exemple), soit dans sa syntaxe concrète. Par ailleurs, la possibilité d'utiliser des règles implantées en boîte noire mérite plus de précision. Par exemple, quel sera le cadre d'exécution si des implémentations *Black Box* effectuées dans différents langages de programmation sont utilisées pour définir une transformation ? Tout cela montre qu'il y a plusieurs façons d'être un outil compatible QVT (voir tableau 2.2). Le risque sera l'émergence d'outils dits compatibles QVT mais incompatibles entre eux. Chacun sera tenté d'implémenter le sous-ensemble QVT qui l'intéresse et donc de remettre en cause le caractère standard de la spécification. D'ailleurs, les premières annonces d'implémentation QVT le confirment : TCS[7] (Tata Consulting Services) dispose déjà d'un prototype d'implémentation de QVT Relations dénommé ModelMorf [110] ; France Telecom, de son côté, a développé un prototype pour le *QVT Operational Mappings* [9] ; Borland, pour sa part, dispose d'une version commerciale de l'implémentation de QVT/Operational déjà intégrée dans son produit Together Architect 2006[8]. Cette implémentation concerne un sous-ensemble du *QVT Operational Mappings* avec quelques déviations par rapport à la spécification actuelle de QVT.

Peut-on dire que QVT a atteint ses objectifs ? La RFP (Request For Proposal) initiale de QVT comportait trois parties (outre la description des procédures de l'OMG) : les *General Requirements* (attentes d'ordre général), les *Mandatory Requirements* (attentes obligatoires) et les *Optional Requirements* (attentes optionnelles). Par exemple, la mandatory requirement numéro 5 stipule que le langage de définition des transformations doit permettre la création

[7] TCS est l'un des principaux contributeurs du langage *Relations* de QVT
[8] http ://www.borland.com/us/products/together/index.html

de vues sur les métamodèles. Il est difficile de voir comment cette attente a été prise en compte dans la spécification actuellement adoptée.

Par ailleurs, lorsqu'en avril 2002 l'OMG a écrit la RFP pour QVT, les transformations de modèles au sens MDA étaient encore relativement mal maîtrisées ; il n'existait pas vraiment d'expériences solides en la matière, pas de tendance générale de pratique de transformations. C'est ce qui explique, en partie, les divergences entre les huit réponses initiales, desquelles il était visiblement difficile de trouver un consensus. C'est sans doute aussi l'une des raisons qui a retardé l'adoption de la spécification (avril 2002 à novembre 2005).

Enfin, il faut noter que QVT dans son état actuel ne traite que des transformations de modèles à modèles (M2M). Une RFP pour le cas spécifique de la génération de code (M2T) est en cours depuis avril 2004.

2.4 Conclusion

Nous avons montré, dans ce chapitre, la place primordiale qu'occupent les transformations de modèles dans les approches d'ingénierie dirigée par les modèles. De multiples approches et outils de transformations existent, mais sont réalisés selon les points de vue des auteurs. Le site internet *planetmde.org* [9] répertorie les outils de transformations les plus connus et propose même un document sur la comparaison des outils.

La proposition de QVT visant à fournir un standard de définition des transformations est maintenant adoptée. Cependant, vu la complexité de cette spécification, il faudra encore du temps avant de voir apparaître les premiers outils QVT. La pluralité des langages constituant QVT, et par conséquent la diversité de compatibilité QVT, remet en cause le standard et l'on peut se demander si QVT saura atteindre son objectif : « éviter une prolifération de systèmes de transformations ».

En définitive, les transformations de modèles restent encore assez mal maîtrisées. La maturité dans ce domaine pourra être appréciée lorsque de véritables environnements de transformations seront disponibles à l'exemple des environnements de développement intégré (IDE) des langages de programmation modernes. Des tentatives dans ce sens sont en train d'émerger autour de la plate-forme Eclipse et des outils de transformations de modèles réalisés sous forme de « plug-ins » Eclipse tels que Kermeta, ATL, etc.

[9] www.planetmde.org

Chapitre 3

Traçabilité et interopérabilité dans les transformations de modèles

Dans la littérature, on trouve beaucoup de travaux en rapport avec la *traçabilité* et l'*interopérabilité* dans le contexte de l'IDM et, plus précisément, dans celui des transformations de modèles. Le nombre de *workshops* dédiés à ces thèmes témoigne de l'intérêt que leur porte la communauté des chercheurs de l'IDM. Dans ce chapitre, nous proposons un état de l'art non exhaustif sur ces sujets. Les deux premières sections leur sont réservées ; elles définissent et présentent l'importance de chaque sujet pour l'IDM et présentent les différentes approches rencontrées dans la littérature pour leur mise en œuvre. La troisième section présente une critique de l'état de l'art et introduit la solution que nous proposons.

3.1 Traçabilité

La traçabilité dans la conception de logiciels a longtemps été réduite à la traçabilité des besoins des utilisateurs. Elle consistait à mettre en œuvre des sortes d'associations entre les besoins (requirements) des projets et les artefacts de conception et de réalisation des logiciels. Cette forme de traçabilité appelée souvent *ingénierie des besoins* sert principalement à documenter la satisfaction de l'implémentation des systèmes vis-à-vis des besoins des utilisateurs [100, 18]. Cela se fait au moyen de liens définis entre les besoins et les artefacts d'implémentation.

Dans ce document, nous ne nous intéressons pas à ce type de traçabilité. Nous nous focalisons plutôt sur la traçabilité dans le contexte des transformations de modèles.

Dans le cadre de l'ingénierie dirigée par les modèles, de l'automatisation du cycle de développement des logiciels par transformations de modèles, cette conception de la traçabilité doit être revue. Avec l'émergence des langages et outils de transformations de modèles tels que le MOF et QVT, le besoin de mécanismes de traçabilité intrinsèque fournissant des informations de trace entre les éléments de conception et artefacts d'implémentation devient une évidence. La traçabilité est largement reconnue comme essentielle dans l'ingénierie des logiciels et systèmes, et cela se confirme par une littérature abondante et un intérêt de recherche croissant dans ce domaine. L'avantage d'une activité de gestion de la traçabilité est

donc un consensus. Cependant, aujourd'hui, plusieurs raisons font encore que l'adoption d'un tel consensus sur la traçabilité est difficile. Il n'existe pas de définition unique ni de standards. Au nombre des définitions que l'on rencontre, on peut citer les suivantes :

- Selon Netta Aizenbud-Resher et al. [12], la traçabilité est la définition d'une relation entre les différentes entités produites pendant le cycle de développement de logiciels,
- Krzysztof et Simon [71] définissent la traçabilité comme la possibilité dans les transformations de modèles de maintenir des liens entre les éléments des modèles sources et leurs correspondants dans les modèles cibles,
- Bert Vanhooff et Yonlande Berbers [121] considèrent que les liens de traçabilité dans les transformations fournissent un historique complet ou partiel des changements intervenus sur les modèles durant la transformation,
- Du point de vue de Frédéric Jouault [69], un langage ou outil capable de maintenir un ensemble de relations entre les éléments de modèles sources et cibles est un langage ou outil qui supporte la traçabilité.

S'il est vrai qu'il ne peut exister une notion unique de traçabilité qui soit utilisée dans toutes les situations, il n'en demeure pas moins nécessaire qu'une sémantique précise de la traçabilité soit définie. Cette sémantique permettra aux développeurs de mieux capturer le sens des relations spécifiques de traçabilité ainsi que les effets de bord et favoriser la production d'outils plus appropriés pour la gestion et le monitoring de la traçabilité. Par exemple, dans le domaine de la modélisation, si l'on considère les outils UML disponibles depuis quelques années, on constate qu'ils ont aidé à créer des modèles visuels permettant de mieux comprendre les problèmes et les solutions proposées. Ces modèles ont été très utiles, particulièrement dans les phases de développement, mais à cause du manque de traçabilité et de cohérence entre les différents modèles et les outils, ces modèles et les codes associés sont très vite désynchronisés.

Dans la pratique, différents types de traçabilité sont donc utilisés avec différentes caractéristiques et propriétés. Dans la suite de cette section, nous allons présenter les différentes classes représentatives des approches de traçabilité que l'on rencontre dans la littérature.

3.1.1 Approches de conception de la traçabilité

Les aproches de conception de la traçabilité peuvent être classifiées suivant deux critères : l'expression des relations entre éléments sources et éléments cibles, et la gestion des liens de traçabilité.

1. L'expression des relations entre les éléments des modèles sources et cibles
   Les relations entre les éléments sources et cibles peuvent être exprimées soit :
   - directement dans les modèles (sources ou cibles ou les deux),
   - dans un autre modèle qui, dans le cas des transformations de modèles, sera un modèle cible de la transformation.
2. La gestion des liens de traçabilité
   Cette gestion peut être faite soit :
   - au moyen de l'outil de transformation (automatiquement ou manuellement),
   - de façon manuelle.

Dans ces deux cas, les liens de traçabilité peuvent être stockés soit dans l'un des modèles ou de façon externe dans un autre modèle.

A partir de ces critères de classification, nous pouvons distinguer deux grandes catégories d'approches de conception de la traçabilité : les approches où les informations de trace sont stockées en interne dans les modèles et les approches qui proposent un modèle externe de trace. Nous présenterons successivement ces deux approches et nous terminerons cette section par une présentation de l'approche QVT de la traçabilité dans les transformations de modèles. Nous avons préféré présenter l'approche QVT isolément parce qu'elle s'avère hybride par rapport aux deux catégories que nous avons identifiées.

Approches sans modèle de trace

Dans cette catégorie d'approches, la traçabilité est définie comme un ensemble de relations entre des entités sources et des entités cibles. Pour illustrer cette catégorie, nous retiendrons deux approches : l'une proposée par Netta Aizenbud-Resher et al. [12], et l'autre par Bert Vanhooff et Yolande Berbers [121].

L'approche Netta Aizenbud-Resher et al. [12]

Dans l'approche de Netta Aizenbud-Resher et al., un lien de trace est modélisé au moyen du lien *dépendance (dependency)* d'UML. En effet, UML propose un certain nombre de types de relations entre éléments de modélisation. Parmi ces types de relations, les plus utilisés sont les *associations* et les *généralisations*. La relation de *dépendance (dependency)*, quant à elle, est une relation générique qui peut être spécialisée à l'aide d'un stéréotype.

Le but recherché dans ce travail est de définir une approche de traçabilité qui permettra une gestion automatique de la consistence des modèles UML. Pour atteindre cet objectif, les auteurs proposent une *sémantique opérationnelle*[10] pour la traçabilité. Les idées à la base de la définition de cette sémantique peuvent être résumées dans les questions suivantes :

- quelle `information` faut-il maintenir entre les éléments liés par une dépendance ?
- quelles `actions` faut-il déclencher lorsque des changements interviennent dans un ou plusieurs éléments liés par une relation de dépendance ?
- quelles `actions` faut-il entreprendre quand un changement se produit sur une relation, afin de pouvoir assurer la validité de celle-ci ?

Nous proposons le métamodèle de la figure 3.1 pour représenter et décrire les concepts et fondements de cette approche.

Un lien de traçabilité *TraceLink*, défini au moyen d'une relation de dépendance stéreotypée d'UML, établit une relation entre des éléments de modèles représentés par le concept *ModelElement*. Il contient également un certain nombre de propriétés sémantiques (*SemanticProperty*) qui permettent d'opérationnaliser la gestion du lien.

Une propriété sémantique (*SemanticProperty*) est définie à l'aide d'un triplet : *event (événement), condition, action*.

[10] Il faut comprendre par sémantique opérationnelle une sémantique qui n'est pas juste déclarative, mais plutôt exécutable.

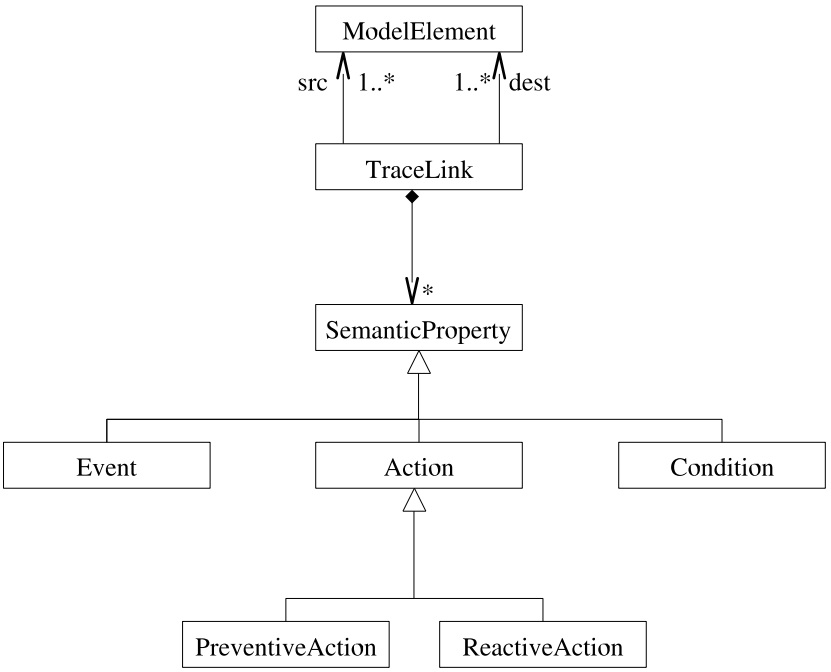

FIG. 3.1 – Métamodèle de la sémantique opérationnelle de traçabilité

Un événement (*event*) correspond à l'occurrence d'un changement dans un modèle. Trois types primitifs d'événements (création, modification et suppression) sont définis.

Une condition (*condition*) est une expression logique décrivant les conditions sous lesquelles un événement qui se produit devra être pris en compte. Les conditions permettent donc d'effectuer un contrôle plus fin sur les événements.

Une action (*action*) spécifie la réponse qui doit être apportée lorsqu'une occurrence d'événement se produit. Une action est soit préventive (*PreventiveAction*), si elle décrit quelque chose qui ne devrait pas se produire dans le modèle, soit réactive (*ReactiveAction*), si elle décrit une réaction à entreprendre quand un changement intervient dans le modèle. Le concept d'action peut être modélisé à l'aide du langage d'action d'UML 2.

Cette approche présente deux avantages majeurs : premièrement, le fait que les éléments utilisés pour produire les informations de la traçabilité puissent être intégrés dès la phase de conception des modèles permet la prise en compte de la traçabilité le plus tôt possible dans le flot de conception des systèmes. Deuxièmement, la définition d'une sémantique opérationnelle rend possible non seulement la génération automatique de la trace, mais aussi une gestion automatique de la consistence et de la synchronisation des modèles.

L'inconvénient principal de l'approche réside dans le fait qu'il n'existe pas de moyen simple de retrouver les informations de trace. En effet, il faut relire l'ensemble des modèles pour reconstruire la trace. L'information de trace est donc difficilement exploitable dans un autre contexte. De plus, la trace se retrouve noyée dans les modèles, ce qui est contraire à la notion de séparation des aspects. Enfin, dans un atelier de transformations de modèles, si un modèle est impliqué dans plusieurs transformations, les éléments de modélisation de la trace peuvent vite devenir un encombrement pour le modèle initial. Il faut cependant noter

que l'utilisation de cette approche dans des modèles de transformations permet de pallier ces inconvénients.

L'approche Bert Vanhooff and Yolande Berbers [121]

Dans l'approche Bert Vanhooff and Yolande Berbers[121], la trace est utilisée pour capturer les effets des transformations sur les modèles. Les informations de trace sont générées par les transformations, et stockées dans les modèles. Les transformations concernées dans cette approche sont des transformations endogènes UML, c'est-à-dire, des transformations dont les modèles sources et cibles sont tous des modèles UML.

Le principe de base de cette approche consiste à dire qu'une transformation est composée d'unités de transformations qui constituent les briques de base pour construire des transformations plus complexes. Chaque unité de transformation réalise un *mapping* entre des éléments des modèles sources et cibles et enregistre les modifications effectuées dans les modèles. Les informations de trace laissées par une unité de transformation peuvent être exploitées ultérieurement par d'autres unités. Dans cette approche, les informations de trace sont composées de :

- la définition de relation de dépendance entre éléments sources et cibles des modèles ; c'est le *mapping* des éléments,
- la définition de relation de dépendance entre les mappings et les unités de transformations qui le créent,
- les marques de suppression. Lorsque des éléments de modèles sont supprimés par des unités de transformation, ces éléments sont marqués sans être physiquement supprimés. Ce marquage permet de conserver l'information de suppression.

Les objectifs attendus de la traçabilité dans l'approche Bert Vanhooff and Yolande Berbers se résument dans les points suivants :

1. les informations de trace pour tout changement important effectué par une unité de transformation sur des modèles doivent être enregistrées par cette unité,
2. les liens de traçabilité doivent être étendus avec des informations spécifques aux unités de transformation pour créer des liens sémantiquement riches,
3. toute information de trace doit être stockée dans les modèles UML, de façon indépendante de la transformation,
4. il doit être possible de pouvoir ajouter manuellement des liens de trace pour les transformations non automatiques ou adaptées manuellement.

Pour satisfaire toutes ces exigences, les auteurs ont défini un métamodèle de transformations de modèles et un profil UML pour spécifier la traçabilité dans le contexte de leurs transformations. La figure 3.2 présente le métamodèle, le profil UML et le mapping entre les deux.

Le concept de *ElementMapping* représente une relation entre zéro ou plusieurs éléments sources (*InputElement*) et un ou plusieurs éléments cibles (*OutputElement*). Un *InputElement* possède un attribut *deleted* de type booléen qui permet d'indiquer si l'élément est supprimé ou pas. Chaque *ElementMapping* est associé à une classe de transformation de type *TFType*. Une unité de transformation *TranformationUnit* possède un certain nombre d'*ElementMapping*.

Les classes *Uses* et *Replaces* sont des spécialisations du concept de *ElementMapping*. La classe *Uses* signifie que les éléments cibles du mapping utilisent les éléments sources sans

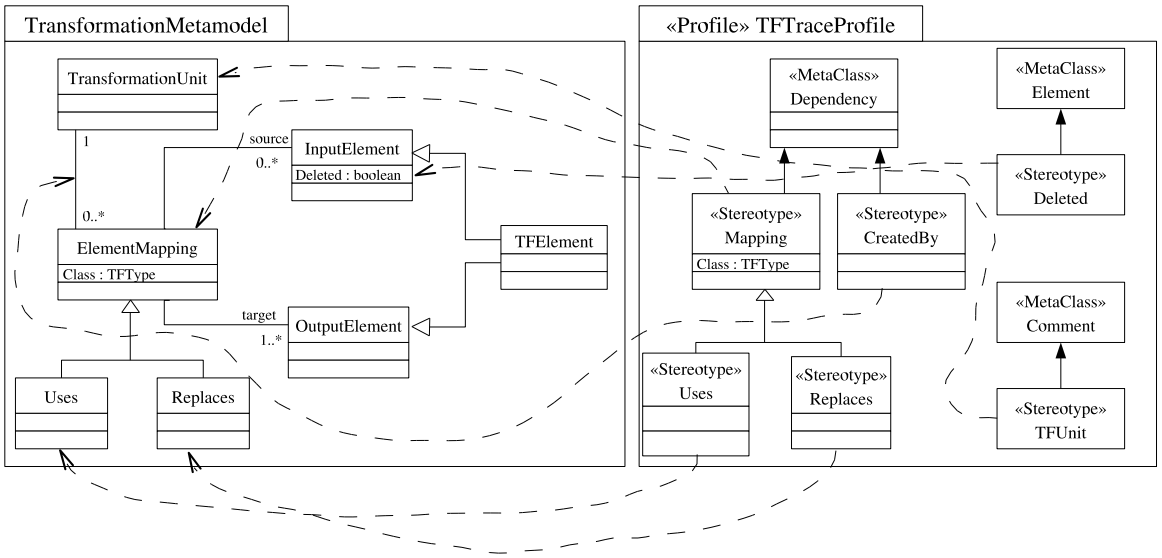

FIG. 3.2 – Métamodèle de la traçabilité et son mapping sur le profil UML

aucune altération (modification) de ces derniers. La classe *Replaces*, quant à elle, signifie que les éléments cibles du mapping remplacent les éléments sources et que ces derniers seront donc marqués comme supprimés.

Le mapping entre les concepts du métamodèle de transformation et les stéréotypes du profil UML est représenté sur la figure 3.2 par des traits en pointillés.

Approches avec modèle de trace

Le principe de `tout est modèle` est sans doute la force motrice dans l'évolution actuelle de l'ingénierie dirigée par les modèles. Il est ainsi naturel de considérer l'information de traçabilité comme un modèle à part entière et, même, comme un modèle cible d'une transformation. Dans cette perspective, tout outil de transformation qui supporte des transformations avec plusieurs modèles en sortie (résultat) sera capable de fournir de la traçabilité sans qu'il soit au préalable nécessaire de définir ou d'étendre les constructions du langage ou moteur de transformation. Ces approches supposent l'existence d'un métamodèle de traçabilité sur la base duquel sera généré le modèle de trace.

Nous proposons de présenter l'approche de traçabilité dans le langage ATL [69] pour illustrer cette catégorie de conception de la traçabilité dans les transformations de modèles.

Un métamodèle simplifié de la traçabilité dans ATL se présente comme le montre la figure 3.3.

D'après ce métamodèle, une information de trace (*TraceLink*) définit une relation entre des éléments sources et cibles des modèles (*AnyModelElement*). Cette information de trace contient également le nom de la règle de transformation qui a été utilisée pour créer ou modifier les éléments cibles à partir des éléments sources.

Dans ATL, la mise en œuvre de la traçabilité peut être réalisée de deux façons. La première consiste à insérer manuellement dans les règles de transformations le code nécessaire pour

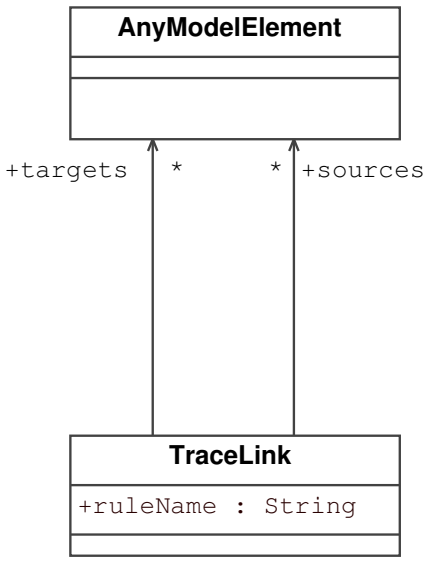

FIG. 3.3 – Métamodèle Simplifié de la traçabilité dans ATL

créer les éléments du modèle de trace. La seconde est d'insérer le code de gestion de la trace de manière automatique dans la définition de la transformation.

Approche QVT de la traçabilité

La spécification MOF définit une classe nommée *Trace Class* dont les attributs font référence à des objets et éléments de modèles impliqués (liés) dans une transformation. Les instances de cette classe *Trace Instances* sont créées lors de l'exécution de la transformation. Ces deux classes sont utilisées dans QVT pour la définition de la trace.

La traçabilité dans QVT peut être réalisée de trois façons : implicite, explicite - avec infrastructure de modélisation de la trace - et explicite - sans infrastructure de modélisation selon le langage utilisé pour définir les transformations.

La réalisation implicite et automatique concerne le Relations Language où par définition la spécification de la transformation établit des relations entre les éléments des modèles qui participent à la transformation. Pour chaque Relation, une Trace Class est dérivée et possède une référence vers chacun des domaines mis en relation.

La réalisation explicite avec infrastructure de modélisation de la trace est définie pour le Core Language. Le concept de *Mapping* du Core Language comporte une zone (*area*) qui consiste en une paire de *patterns* (voir figure 3.4). Ces patterns localisent et créent les instances de *trace classes* qui enregistrent les relations entre les éléments de modèles. Cette figure présente un mapping de deux domaines : un domaine avec un modèle de type L et l'autre avec un modèle de type R. Chaque rectangle dans le schéma représente un pattern. Les colonnes sont appelées des zones (areas). Chaque zone est constituée de deux patterns : le pattern *Guard* (en haut) et le pattern *Bottom* (en bas). Dans cette description du mapping, la zone du milieu correspond aux patterns qui « matchent » le modèle de trace. Le langage

Core language fournit donc à l'utilisateur un mécanisme pour définir son modèle de trace, mais n'assure pas une gestion automatique de celui-ci.

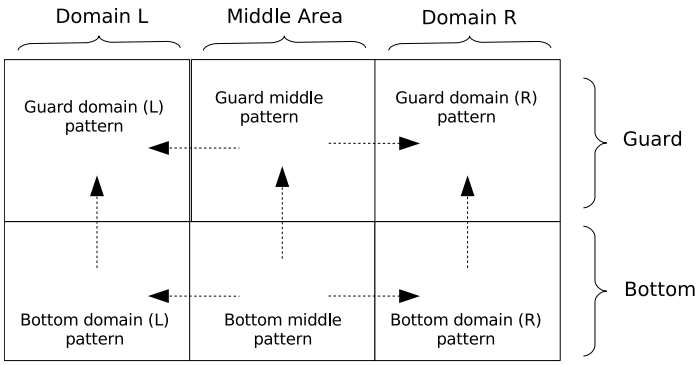

FIG. 3.4 – Domaines et patterns du QVT Core language

Enfin, dans le langage Operationnal Mappings, la gestion de la trace est entièrement à la charge de l'utilisateur.

3.1.2 Synthèse

Nous venons de présenter très brièvement différentes approches de traçabilité dans les transformations de modèles. Cette présentation est loin d'être exhaustive. Par exemple, nous n'avons pas abordé les approches de traçabilité par aspect. En effet, dans le contexte des approches de conception par aspects, la traçabilité peut être considérée comme un aspect transversal dans la conception des systèmes. Ces approches tendent à appliquer les techniques utilisées dans la programmation/conception par aspects pour obtenir de la traçabilité sur les transformations de modèles. Quelques travaux connus dans ce domaine sont : [85], [67], [76] et [115].

Dans ces différentes approches présentées, on constate que le seul fait consensuel sur la traçabilité dans les transformations de modèles est l'établissement de liens entre les éléments sources et les éléments des modèles. Au-delà de ce dénominateur commun, l'implémentation de la traçabilité est réalisée de façon ad hoc et est souvent étroitement liée aux outils de transformations utilisés. Il faut cependant noter que les approches avec modèle de trace représentent un début de solution. Il faudra, dans ce cas, trouver un moyen de découpler le code de génération de la trace, de celui de la transformation elle-même.

Enfin, nous pensons que l'approche QVT proposée pour gérer la traçabilité est loin d'être satisfaisante dans la mesure où seul le langage de *relations* supporte de façon automatique la traçabilité. Or, le caractère déclaratif de ce langage ne laisse pas entrevoir la possibilité d'une génération de modèle de trace indépendant du modèle de la transformation.

Dans la littérature, la traçabilité et l'interopérabilité sont deux problèmes abordés séparément. Nous pensons que dans le cadre des transformations de modèles, la traçabilité peut être un moyen pour résoudre le problème d'interopérabilité dans les systèmes.

Après avoir passé en revue les approches de traçabilité, dans la section suivante, nous proposons un état de l'art sur l'interopérabilité des systèmes.

3.2 Interopérabilité

Aujourd'hui, la complexité et la grande diffusion des systèmes informatiques (ubiquitous computing) autant que l'environnement économique compétitif à l'extrême rendent nécessaire le développement d'architectures logicielles supportant des applications hétérogènes, tant dans les modèles d'informations traitées que dans les modes d'échange et de coopération. Que vous souhaitiez vous connecter aux systèmes de vos partenaires, accéder aux données d'un site central ou connecter des applications écrites dans différents langages de programmation, vous vous heurtez au problème de faire fonctionner ensemble des technologies hétérogènes. Ce problème touche aujourd'hui chaque entreprise.

La *IEEE Standard Glossary of Sotware Engineering Terminology* définit l'interopérabilité comme étant la capacité qu'ont deux ou plusieurs systèmes ou composants de pouvoir échanger de l'information et de pouvoir s'en servir [108].

Haroon Saleem Khan [78] définit l'interopérabilité comme étant le fait que deux entités différentes puissent travailler ensemble.

Pour Brownsword L. et al. [81], l'interopérabilité est la capacité d'une collection d'entités communicantes à partager une information spécifiée et d'opérer sur cette information selon un consensus opérationnel et sémantique.

Le point commun entre toutes ces définitions est l'échange et l'exploitation de l'information. C'est pourquoi Grace A. et Lutz [83] distinguent deux niveaux d'interopérabilité : *l'interopérabilité syntaxique* et *l'interopérabilité sémantique*. L'interopérabilité syntaxique concerne la capacité d'échange de l'information, alors que l'interopérabilité sémantique traite de l'interprétation commune du sens de l'information échangée et de la façon dont celle-ci devra être exploitée. À ces deux niveaux d'interopérabilité identifiés par Grace A. et Lutz, il faut ajouter un troisième niveau : *l'interopérabilité technologique* qui concerne la coopération entre plusieurs entités logicielles issus de différentes technologies d'implémentation.

Pour répondre au besoin d'interopérabilité, différentes solutions et approches ont été proposées. Ces approches peuvent être classées en trois grandes catégories :

- l'interopérabilité dirigée par les cadres méthodologiques,
- l'interopérabilité dirigée par les modèles et les échanges de données,
- l'interopérabilité centrée sur les services.

Dans la suite de ce chapitre, nous présenterons ces différentes catégories d'approches d'interopérabilité tout en faisant ressortir les différents niveaux d'interopérabilité pris en compte. Nous effectuerons une analyse de chaque approche quant à son applicabilité dans le contexte de nos travaux.

3.2.1 Interopérabilité dirigée par les cadres méthodologiques

Dans cette catégorie d'approche, la résolution du problème d'interopérabilité passe par la définition d'une méthodologie de conception et de réalisation des systèmes qui permettra

d'aboutir à des systèmes interopérables. On pourrait ici parler d'interopérabilité par construction. Nous choisirons comme exemple le MDA pour présenter cette famille d'approches.

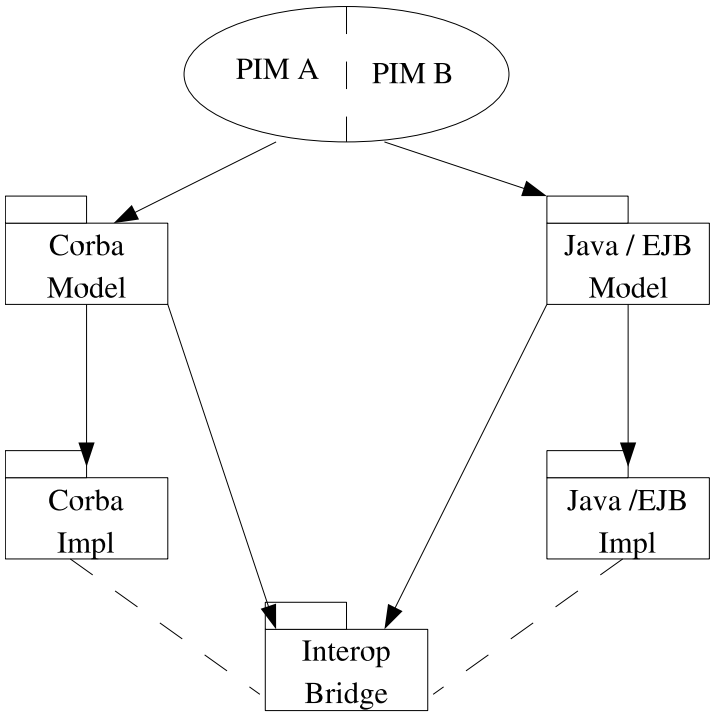

FIG. 3.5 – Interopérabilité technologique dans MDA

La démarche d'interopérabilité dans MDA repose sur le fait qu'un même modèle peut être mis en œuvre sur différentes plates-formes en même temps. Dans ce cas, les différentes parties doivent pouvoir communiquer. De même, il est possible de faire interopérer différentes applications en exprimant leurs relations au niveau PIM. Au moment de la génération des PSMs, des modèles de connexion sont générés, ce qui permet d'obtenir une implémentation des ponts d'interconnexion au moment même de la génération de code comme l'illustre la figure 3.5. Le *Interop Bridge* permet donc de s'affranchir des contraintes d'incompatibilité des plates-formes et constitue, de ce fait, une sorte de pont technologique. Cette interopérabilité relève du niveau technologique.

Cette approche d'interopérabilité, si séduisante soit-elle, reste problématique. Premièrement, il faut noter que MDA ne donne aucune indication quant à la façon concrète de réaliser les *bridges* d'interopérabilité entre les plates-formes (ponts technologiques). Deuxièmement, l'utilisation de cette approche pour prendre en compte des applications déjà existantes nécessite de re-modéliser ces applications pour pouvoir entreprendre la génération des *bridges*. Le coût de cette rétro-ingénierie peut constituer un véritable frein à la mise en œuvre de cette démarche dans ce contexte particulier.

3.2.2 Interopérabilité dirigée par les modèles et les échanges de données

Cette catégorie d'approche d'interopérabilité repose sur l'existence d'un langage commun permettant de définir les modèles ou données à échanger. Elle répond principalement à l'interopérabilité au niveau syntaxique. L'exemple le plus caractéristique de cette famille d'approches est l'interopérabilité par échange de fichier XMI (XML Metadata Interchange) [92]. C'est aussi l'approche retenue par l'OMG pour l'échange de données et modèles entre outils MDA.

XMI permet de décrire une instance du MOF sous forme textuelle grâce au langage XML (eXtensible Markup Language) [4] du W3C. Il définit comment utiliser les balises XML pour représenter un modèle MOF en XML. Les méta-modèles sont décrits par des DTDs (XML Document Type Definition) et les modèles sont écrits à l'aide de documents conformes à la DTD du métamodèle. Ainsi, comme XMI est basé sur XML, les méta-données (tags) et les instances (éléments) sont regroupées dans le même document, ce qui permet à une application de comprendre les instances grâce à leurs méta-données. XMI devient alors le format standard d'échange entre différents outils MDA comme le montre la figure 3.6.

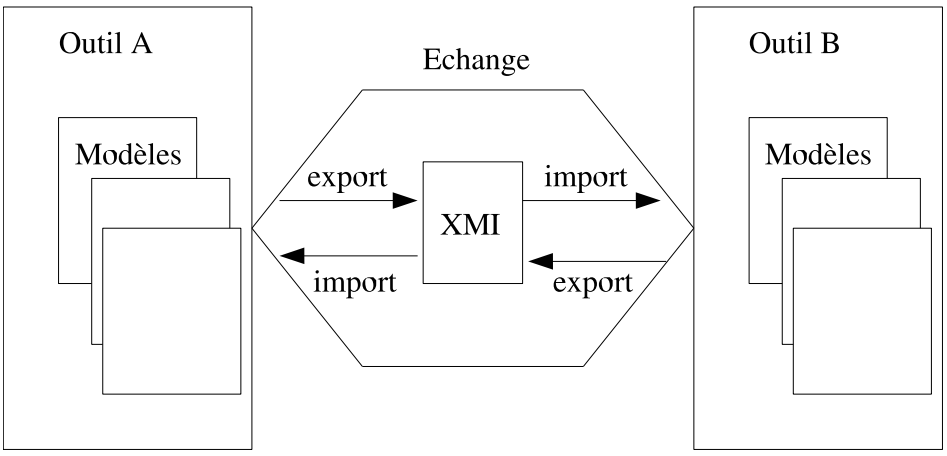

FIG. 3.6 – Interopérabilité syntaxique dans MDA

Le problème de l'interopérabilité basée sur le XMI réside dans l'existence de plusieurs versions de la spécification. De plus, les outils ne supportant généralement que certaines de ces versions, il est donc souvent nécessaire de réaliser des conversions entre différentes versions de fichiers XMI produits par les outils afin de pouvoir réaliser l'échange.

Malgré les limitations et contraintes liées à cette approche, elle reste aujourd'hui la plus utilisée dans les outils de modélisation.

3.2.3 Interopérabilité centrée sur les services

Nous regroupons sous cette catégorie toutes les approches d'interopérabilité centrées sur la notion de service. Un service est une fonction bien définie, autonome, ne dépendant d'aucun contexte. L'interopérabilité dans ces approches passe par la mise en œuvre de services

connus ou découvrables dans l'environnement et qui peuvent être fournis ou consommés par les différentes entités qui doivent interopérer.

Dans cette catégorie d'approches, nous présenterons les approches des *architectures orientées services (SOA)* et l'approche *ModelBus*.

Interopérabilité par les Service-Oriented Architecture (SOA)

Une architecture orientée service (SOA) est une architecture construite autour d'une collection de services avec des interfaces bien définies (e.g DCOM, ORBs basées sur la spécification CORBA) où chaque service expose un contrat comme l'illustre la figure 3.7.

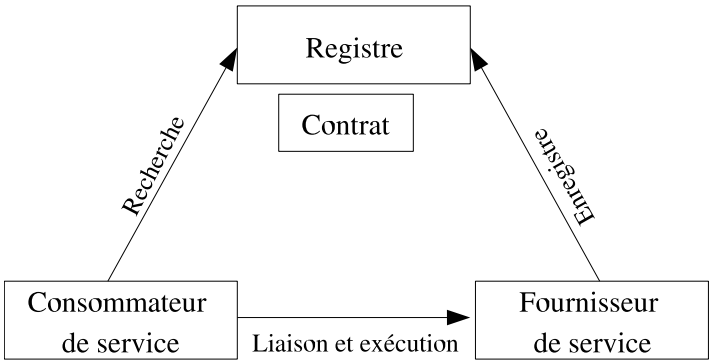

FIG. 3.7 – Aperçu des services orientés architecture

Dans cette approche, un système, ou application, est conçu et implanté comme un ensemble d'interactions entre des services. Dans une architecture orientée service, l'interopérabilité se définit simplement comme la capacité d'invocation d'un service par un potentiel client [87]. Cette interopérabilité est réalisée au moyen :

- d'un adaptateur et d'un protocole communs ; chaque service fournit une interface qui est invoquée à travers le format d'un adaptateur et un protocole qui est compris par tous les clients potentiels du service,
- d'interfaces publiées et d'interfaces découvrables ; chaque service a une interface publiée et une interface découvrable qui permettent aux systèmes de rechercher les services qui répondent à leurs besoins,
- de multiples interfaces de communication ; les services peuvent implanter différentes interfaces de communication,
- de compositionalité ; les systèmes peuvent être construits comme une composition de services.

Les web services sont aujourd'hui l'implémentation la plus connue des SOA. Un service web est une instanciation d'un SOA avec les considérations suivantes :

- les interfaces des services sont décrites par le *Web Services Description Language(WSDL)*,
- l'adapteur est transmis en utilisant le *Simple Object Access Control over HTTP*,
- le *Universal Description Discovery and Integration(UDDI)* est le répertoire de service.

Les SOA permettent d'établir une abstraction de la logique métier ; la technologie d'implantation peut introduire des changements dans la modélisation des processus métiers et

de l'architecture technique avec, tout de même, un couplage faible entre ces modèles. Les approches d'interopérabilité basées sur les SOA se situent donc au niveau de l'interopérabilité syntaxique et sémantique. Interopérabilité syntaxique, à cause du format commun de codage des messages, et interopérabilité sémantique, de par l'existence d'un contrat entre les consommateurs et fournisseurs de services.

Deux problèmes empêchent l'utilisation des SOA dans le contexte de nos travaux : premièrement, leur lenteur fait qu'ils sont inacceptables pour des applications de simulation, et deuxièmement, ils induisent une influence de la technologie sur les processus métiers. A l'opposé, dans le contexte de l'IDM, on souhaite plutôt avoir une modélisation indépendante du métier vis-à-vis de la technologie.

Interopérabilité via le ModelBus

Pour comprendre l'approche *ModelBus*, il convient d'abord de définir la notion de *service de modélisation*.

Définition 3.1. *Un service de modélisation est une opération automatisée sur des modèles. Les paramètres d'entrée et de sortie de cette opération étant des modèles [20].*

Le projet ModelBus est né du constat qu'aujourd'hui, plusieurs outils de modélisation du type *CASE* proposent plusieurs services de modélisation tels que : le stockage, l'édition, les transformations et vérifications de modèles. Cet ensemble de services est en général fourni par des outils hétérogènes. Par exemple,

- chaque outil possède sa propre définition de modèle (Objecteering et Rational Rose ont leur propre métamodèle pour UML1.4). Il est donc difficile d'envisager l'utilisation conjointe de deux services de modélisation,
- chaque outil possède sa propre technologie de représentation des modèles(JMI, EMF ou autre). De plus, quand bien même les outils utilisent XMI pour l'échange, ils n'utilisent pas le même mécanisme pour importer et exporter le XMI. L'existence de plusieurs versions XMI constitue une difficulté supplémentaire,
- chaque outil a son propre mécanisme d'accès aux modèles (Interface grahique utilisateur, Web services, API, ou autre). Par ailleurs, ces mécanismes d'accès peuvent être selon les outils, orientés services, orientés événements, ou les deux,
- enfin, ces outils peuvent être distribués sur un réseau et peuvent tourner sur plusieurs systèmes d'exploitation.

Ce constat révèle le besoin d'interopérabilité des services de modélisation. Xavier Blanc [21] passe en revue toutes les approches d'interopérabilité existantes et montre qu'aucune de ces approches ne permet de résoudre de façon satisfaisante ce problème d'interopérabilité.

La problématique dans ModelBus est donc de fournir un cadre d'interopérabilité entre des outils hétérogènes de modélisation. Nous ne détaillons pas ici l'approche ModelBus ; le lecteur intéressé pourra se référer à [20, 21].

ModelBus est une solution d'interopérabilité proposée dans le contexte de l'ingénierie dirigée par les modèles. Seulement, la problématique d'interopérabilité dans ModelBus (interopérabilité des outils) n'est pas la même que la nôtre (interopérabilité de plates-formes et de niveaux d'abstraction). Néanmoins, nous pensons qu'une utilisation partielle de ModelBus reste possible dans notre approche. Pour ce faire, nous aurons besoin de définir un

certain nombre de services qui mettront en œuvre le besoin d'interopérabilité dans notre approche, et ensuite envisager l'intégration de ces services dans ModelBus.

3.2.4 Synthèse

Dans cette section, nous avons passé en revue les principales catégories d'approches de l'interopérabilité. Nous avons, pour chaque catégorie, identifié les niveaux d'interopérabilité abordés comme le montre le tableau 3.1.

	Interopérabilité Technologique	**Interopérabilité Sémantique**	**Interopérabilité Syntaxique**
MDA	Ponts technologiques	N/A	XMI
XMI	N/A	N/A	format d'échange
SOA	N/A	Contrat	format d'échange
ModelBus	N/A	Contrat	format d'échange

TAB. 3.1 – Niveaux d'interopérabilité des approches

Ce tableau montre que les niveaux technologique et sémantique de l'interopérabilité ne sont pas assez pris en compte dans les approches actuelles d'interopérabilité. Or, dans le contexte de l'ingénierie dirigée par les modèles, cela s'avère indispensable.

Par ailleurs, les approches actuelles se basent toutes sur l'hypothèse de l'existence d'un consensus. Il est donc nécessaire de définir un cadre général d'interopérabilité dans le cas des systèmes utilisant ou exposant différentes technologies et qui ont besoin d'interopérabilité. En l'occurrence, une approche d'interopérabilité entre des systèmes issus de transformations de modèles est largement indispensable dans le flot de conception du MDE. Dans [30], Jean Bézivin et al. proposent une approche d'interopérabilité entre différents outils. Dans leur approche, l'interopérabilité est réalisée par des opérations spécifiques de transformations de modèles. En effet, l'interaction entre outils passe généralement par l'utilisation d'artefacts permettant à un outil d'accéder à des objets créés par d'autres outils et de les utiliser. Il est donc tout à fait envisageable d'implanter un tel mécanisme par des transformations de modèles. Nous pensons que ce type de mise en œuvre de l'interopérabilité couplé avec l'approche MDA peut être une bonne solution dans le cadre de notre travail.

3.3 Conclusion

Dans ce chapitre, nous avons donné un aperçu global des différentes approches de mise en œuvre de la traçabilité et de l'interopérabilité dans les transformations de modèles. Nous avons, dans chaque cas, montré l'inadéquation entre les approches proposées et la conception de systèmes embarqués dans une approche de l'ingénierie dirigée par les modèles. Nous avons, en l'occurrence, montré que l'interopérabilité conceptuelle proposée par la démarche MDA présentait des perspectives intéressantes, mais qu'aucune démarche concrète de mise en œuvre des *ponts technologiques* ou *bridges d'interopérabilité* n'était donnée.

Face à ce constat, nous proposons une nouvelle approche d'interopérabilité basée sur l'approche MDA, mais qui définit une démarche concrète de génération automatique des *ponts technologiques* ou *ponts d'interopérabilité*. Cette démarche utilise la traçabilité dans les transformations des modèles pour produire des modèles contenant toutes les informations nécessaires à la génération automatique des ponts d'interopérabilité. Cette approche sera présentée dans le chapitre 6.

Chapitre 4

Gaspard : Un environnement de conception et de prototypage des SoCs

4.1 Introduction

Avec l'évolution incessante de la technologie du semi-conducteur, il est aujourd'hui possible de mettre sur une même puce plus de 30 millions de transistors[47]. De telles puces intègrent de multiples systèmes complexes (unités de calculs, fonctions d'accélération matérielle, mémoires, unités de micro/nano électro mécanique, etc.). Ces types de puces sont appélées *SoC* pour Systems-on-Chip (systèmes sur puce en français).

La conception des SoCs fait face aujourd'hui à de nombreux défis : un temps de mise sur le marché (time-to-market) très court, le coût de la conception, la complexité du sillicium, la productivité, etc.

4.1.1 Défis actuels de la conception des SoCs

Time-to-market

Le marché du SoC est un marché très sensible aux délais. En général, les concepteurs ont entre six mois et un an pour réaliser un système, une fois le standard publié ; sinon, le concurrent remporte le marché. Cette forte contrainte de temps fait que les ingénieurs n'ont pas la possibilité d'exploiter réellement tous les transistors disponibles aujourd'hui sur la puce. Cette situation conduit à un fossé entre la disponibilité de transistors et le nombre de transistors effectivement exploités. Il est nécessaire d'accroître la productivité des concepteurs par rapport au nombre de transistors utilisés sur les puces.

Coût de la conception

Depuis 2003, l'ITRS tire la sonnette d'alarme sur le coût de conception des SoCs ; le coût de conception constitue la plus grande menace pour l'avenir du semi-conducteur [26]. Selon l'ITRS, le coût de l'investissement nécessaire (recherche, conception et test) pour réaliser

un nouveau SoC est constitué du coût de fabrication qui se chiffre en millions d'euro, et le coût de la conception qui s'élève à des dizaines de millions d'Euro. Quatre-vingts pour-cent (80%) de ces dizaines de millions sont liés à la conception de la partie logicielle du SoC. De nouvelles techniques de conception sont donc plus que jamais nécessaires pour réduire ce coût.

Complexité du sillicium

L'hétérogénéité des technologies utilisées sur une puce (logique, mémoire, analogique mélangé au digital, systèmes micro/nano electro-mécanique, etc.) demande beaucoup de compétences de la part des ingénieurs pour faire interagir toutes ces technologies. De telles puces ne peuvent plus être considérées comme synchrones à cause des délais induits par la propagation de l'horloge. Par ailleurs, ces systèmes sont souvent appelés à être certifiés sans défauts. Tous ces facteurs accroissent considérablement la complexité de la conception des SoCs.

Productivité

Face à la complexité croissante des SoCs (de plus en plus de transistors), il faut une croissance proportionnelle, en termes de productivité, des équipes de conception. A ce propos, l'ITRS identifie plusieurs défis : la réutilisation, la vérification et le test, l'optimisation dirigée par le coût, des plates-formes de conception et d'implémentation sûres, des processus de gestion de la conception.

4.1.2 Quelques réponses aux problèmes de conception des Socs

Pour répondre aux défis actuels de la conception des SoCs, plusieurs approches ont vu le jour. Au nombre de ces approches, on peut retenir deux classes. La première se focalise sur le besoin de formalisme de haut niveau pour concevoir les SoCs ; elle porte sur la définition de profil UML pour les SoCs. La seconde préconise l'utilisation des approches de l'ingénierie dirigée par les modèles.

Profils UML pour les SoCs

Les profils UML sont envisagés dans les SoCs à deux niveaux : la modélisation et l'analyse de systèmes embarqués, d'une part, et la modélisation de systèmes électroniques, d'autre part.

Pour l'analyse et la modélisation de systèmes, deux profils concurrents sont actuellement en cours de standardisation : SysML (System Modeling Language) [118] et MARTE (Modeling and Analysis of Real-time and Embedded Systems) [61].

SysML est un langage générique et graphique pour spécifier, analyser, concevoir et vérifier des systèmes complexes incluant éventuellement du matériel, du logiciel, des informations et procédures. En particulier, il offre des représentations graphiques avec un fondement

sémantique pour modéliser les besoins, le comportement, et la structure. SysML est un sous-ensemble d'UML 2.0 avec les extensions nécessaires pour prendre en compte les exigences de la modélisation de systèmes.

Le profil MARTE cible la conception de systèmes embarqués ; de ce fait, il fournit les moyens aux concepteurs de modéliser l'application et sa plate-forme d'exécution ainsi que le mapping de l'application sur cette plate-forme. Pour permettre la prise en compte du temps, MARTE raffine les concepts de temps d'UML. Le profil SPT (Scheduling, Performance and Time) d'UML 1.x a été également étendu. Ces extensions ont permis de concevoir, à l'aide de MARTE, des modèles de temps logique, discret et continu. Les modèles MARTE seront, à terme, annotés avec des propriétés non fonctionnelles pour exprimer et analyser les diverses contraintes (temps réel, consommation d'énergie, coûts, etc.) qu'un système doit satisfaire. Une extension pour la modélisation des structures répétitives est en cours d'intégration dans MARTE et devra permettre de concevoir des architectures régulières.

Pour ce qui concerne la modélisation des systèmes électroniques, RTL (Register Transfert Level) est jusque-là, le niveau standard dans l'industrie. Dans le souci d'élever le niveau d'abstraction au-dessus du RTL pour permettre une simulation conjointe (logiciel et matériel) plus rapide, un autre niveau d'abstraction ESL (pour Electronic System Level) est en train d'émerger. Il s'agit d'un ensemble de niveaux d'abstractions appelé TLM (pour Transaction Level Modeling). A ce niveau, les communications sont représentées à l'aide d'opérations de lecture (read) et d'écriture (write) dans la mémoire au lieu de signaux propagés à travers des fils comme c'est le cas dans RTL. Cette abstraction permet de gagner d'un facteur de 100 à 1000 en vitesse de simulation. Plusieurs langages sont définis, à ce niveau, parmi lesquels SystemC [65] et System Verilog [11] sont les plus connus. Ces langages possèdent des moteurs de simulation permettant une simulation au cycle près de l'exécution du logiciel sur le matériel. Parallèlement à ces langages de modélisation et simulation de systèmes électroniques, plusieurs profils UML sont proposés [106, 107, 91]. Le profil UML pour les SoCs [91] - pour ne décrire que celui-là - est en cours de standardisation à l'OMG en tant que profil pour la modélisation de systèmes électroniques. Ce profil étend le *Composite Structure Diagram* d'UML en introduisant les concepts nécessaires (modules, channels, ports, horloge) à la modélisation des architectures matérielles. A l'aide de ce profil, on peut automatiquement générer les squelettes de code SystemC ou autres langages équivalents.

Ingénierie dirigée par les modèles pour les SoCs

Dans les paragraphes précédents, nous avons décrit quelques réponses à la problématique de conception des SoCs basées sur la définition de profils UML et de langages de modélisation à un haut niveau d'abstraction pour faciliter la conception des systèmes, en particulier, celle des SoCs.

On pourrait entreprendre une approche IDM pour la conception conjointe des SoCs par une combinaison de plusieurs de ces approches. En effet, on pourrait dans les premières phases de la conception entreprendre l'utilisation de SysML et passer, ensuite, à MARTE pour analyser les propriétés du système et faire le partitionnement entre logiciel et matériel, puis finalement utiliser, par exemple, le profil UML for SoC pour la génération du code de simulation TLM.

La contribution de Gaspard dans la conception des SoCs est centrée sur l'utilisation des techniques de l'IDM. Il propose l'utilisation de plusieurs modèles définis à différents niveaux d'abstractions. Le scénario de conception dans Gaspard commence par une modélisation très abstraite du système et enchaîne des raffinements à des niveaux d'abstractions intermédiaires pour obtenir en fin de processus un système exécutable/simulable.

Les modèles de très haut niveau contiennent uniquement les concepts spécifiques au domaine ; les concepts technologiques sont introduits graduellement dans des modèles intermédiaires. Chaque modèle du processus est décrit à l'aide d'un métamodèle. Le passage des modèles d'un niveau d'abstraction donné au suivant se fait par des opérations de transformations de modèles.

La séparation claire entre les modèles de hauts niveaux d'abstractions et les modèles incluant des aspects technologiques fait qu'il est plus facile de passer d'une technologie à une autre tout en réutilisant les modèles spécifiques au domaine.

La démarche Gaspard se décline en deux volets :

- **une méthodologie** ; Gaspard propose une démarche méthodologique pour concevoir, vérifier et simuler un système embarqué,
- **un environnement de construction de prototype basé sur la simulation** ; Cet environnement se veut un véritable atelier de conception et de simulation de systèmes embarqués. Il offre un cadre de développement mettant en œuvre la méthodologie proposée. Cette plate-forme, actuellement en développement, sera le test bench dans le projet DaRT et disponible en open source.

Dans la suite de ce chapitre, nous présentons plus en détail chacune des deux composantes de l'approche Gaspard.

4.2 Méthodologie de développement Gaspard

Dans cette section consacrée à la méthodologie Gaspard, nous décrivons dans un premier temps le flot de conception en faisant ressortir les différents niveaux d'abstractions proposés et les métamodèles correspondants. Nous abordons ensuite la place de l'UML dans Gaspard.

4.2.1 Flot de conception et métamodèles

La conception des SoCs couvre un large spectre de compétences allant de la modélisation de l'application par agrégation de plusieurs composants fonctionnels, en passant par l'assemblage de composants physiques, la vérification et simulation du système, jusqu'à la synthèse du produit final sur une puce.

Le flot de conception de Gaspard (figure 4.1) est partiellement inspiré du « Y chart » [48] et s'articule autour de plusieurs niveaux d'abstractions. Les concepts et la sémantique de chacun de ces niveaux sont capturés dans des métamodèles. Au plus haut niveau, le concepteur décrit son application logicielle et l'architecture matérielle sur laquelle l'application sera exécutée ou simulée. A partir de ces deux modèles, il est alors possible de définir comment l'ensemble des composants de l'application va être affecté sur l'architecture matérielle. Cette phase dite d'*association* est décrite à l'aide d'un modèle. Comme le montre la figure, le métamodèle

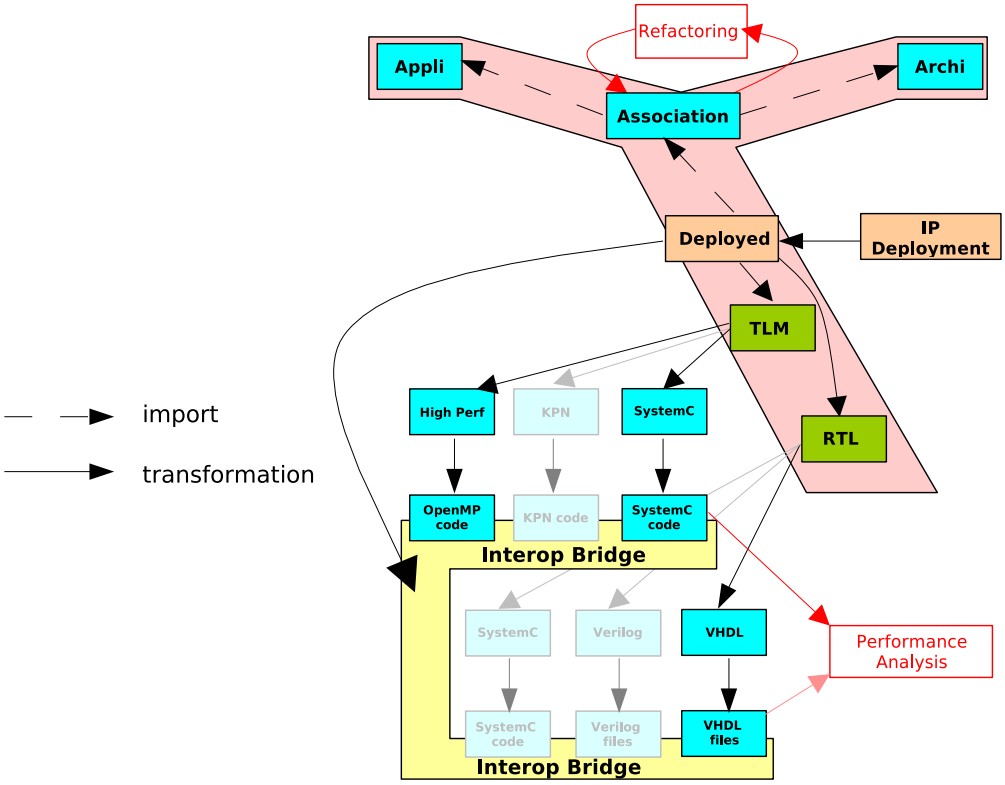

FIG. 4.1 – Flot de conception Gaspard

de l'association dépend de celui de l'application et de l'architecture matérielle. Ces trois modèles (application, architecture et association) sont complètement indépendants de toute technologie d'implémentation. Les informations contenues dans ce modèle sont disponibles dans les niveaux d'abstractions inférieurs.

Une fois l'association réalisée, le modèle obtenu (modèle associé) est enrichi avec des informations de déploiement fournies à l'aide d'un modèle (*IP Deployment*). En effet, certains des différents composants constituant un SoC (processeurs programmables, unités mémoires, éléments d'interconnexion, unités d'entrée/sortie, etc.) peuvent être obtenus auprès de fournisseurs d'IP (Intellectual Property). La réutilisation de composants est incontournable dans la production de SoCs. Le métamodèle *IP Deployment* dans Gaspard permet de fournir des informations sur les IP utilisées pour réaliser certains éléments du modèle associé. Il spécifie également certaines caractérisations nécessaires pour la simulation ainsi que le niveau d'abstraction de simulation (TLM ou RTL) de chaque élément. La fusion des informations dans le modèle associé et le modèle de déploiement permet d'obtenir un modèle déployé (*Deployed*).

Pour permettre la génération de code à partir des modèles abstraits de simulation (TLM et RTL), des métamodèles sont définis pour décrire les langages de simulation/exécution (SystemC, High Perf, VHDL, etc.)

L'utilisation des métamodèles pour décrire les différents niveaux d'abstractions nous permet d'écrire des règles de transformations qui permettent de passer d'un niveau d'abstraction à un autre par transformations de modèles.

Le concept de composant d'UML 2.0 est au cœur de la modélisation dans Gaspard. Les différents éléments de l'application et de l'architecture sont définis à l'aide de la notion de composant moyennant une extension. En l'occurrence, le mécanisme de typage des ports a été redéfini dans Gaspard.

4.2.2 UML 2.0 et Gaspard

UML (Unified Modeling Language) est une notation graphique initialement créée dans le but de promouvoir la communication et, dans une moindre mesure, la productivité parmi les développeurs de l'orienté objet. Il est constitué de treize (13) types de diagrammes dont six (6) servent à décrire la structure et les sept (7) autres à exprimer du comportement. L'ensemble de ces diagrammes permet d'exprimer divers aspects d'un système. Depuis sa standardisation par l'*OMG* en 1997, le succès de l'UML ne cesse de croître. Plusieurs versions d'UML ont été proposées et la dernière, *UML2.0*, standardisée en juin 2003, a été enrichie avec des éléments de modélisation et une sémantique plus étendue et plus systématique que les versions précédentes.

UML propose un mécanisme d'extension (stéréotype et tagged values) qui permet d'adapter le langage à n'importe quel domaine d'activités. Dans l'introduction de ce chapitre, nous avons déjà parlé de plusieurs profils UML pour les SoCs. Un profil correspond à une spécialisation du langage UML pour un domaine donné. Cette spécialisation se fait par la création de nouveaux stéréotypes propres aux concepts du domaine. Ce mécanisme est sans doute l'une des principales raisons du succès d'UML.

Aujourd'hui, le pragmatisme de la notation graphique d'UML plus son extensibilité et l'existence d'outils UML (même en version libre) lui ont permis d'être largement utilisé aussi bien dans les milieux industriels qu'académiques. De plus, avec l'émergence de XMI, l'échange des diagrammes UML entre différents outils a été rendu possible. Ce fut également le début de l'interprétation des schémas, grâce à des parseurs XML. UML cesse, dès lors, d'être une simple notation graphique ; il est aujourd'hui le langage le plus en vogue pour la modélisation.

Dans Gaspard, nous utilisons les diagrammes UML pour l'étude et la co-conception de systèmes sur puce. Notre utilisation d'UML se situe à deux niveaux : (1) pour définir des profils et (2) comme formalisme de modélisation.

Pour permettre aux utilisateurs d'UML de concevoir leurs modèles à l'aide de leurs outils UML favoris, nous avons défini plusieurs profils. Pour chaque métamodèle dont les instances sont des inputs de la plate-forme Gaspard, nous avons défini un profil. Nous avons donc un profil pour définir l'application, un profil pour l'architecture matérielle, un profil pour l'association et un dernier profil est en cours de réalisation pour le déploiement. Un outil Gaspard permet d'importer dans la plate-forme les modèles réalisés en UML et conformes aux profils.

Par ailleurs, le formalisme UML est aujourd'hui le standard utilisé de fait dans les approches de l'IDM pour définir les métamodèles. Le MOF standard OMG pour décrire tout métamodèle est lui-même écrit à l'aide de la notation UML.

4.3 Environnement de développement et de simulation

Dans la section précédente, nous avons présenté le premier volet de Gaspard : l'aspect méthodologie. Maintenant, nous présentons Gaspard en tant qu'environnement de conception, réalisation et de simulation/exécution de systèmes embarqués : c'est l'aspect outillage de Gaspard dont la finalité est de proposer un environnement de mise en œuvre de la méthodologie proposée.

Gaspard est un environnement intégré de développement pour la conception de systèmes sur puce (SoC). Il permet de modéliser, simuler, tester et générer le code pour des SoC. Il est basé sur la plate-forme Eclipse[11]. Ses différentes fonctionnalités sont réalisées sous formes de plugins d'Eclipse.

4.3.1 Gestion des modèles

En tant qu'atelier de prototypage de systèmes embarqués, Gaspard offre plusieurs moyens de gestion de modèles. Basé sur la plate-forme EMF [59], Gaspard permet de générer automatiquement des éditeurs de modèles conformes aux métamodèles Gaspard au format *Ecore*. Ces éditeurs permettent aux utilisateurs de créer et modifier des modèles.

Par ailleurs, les utilisateurs de Gaspard peuvent aussi créer leurs modèles à l'aide de leurs outils de modélisation UML préférés tels que MagicDraw[12], Entreprise Architecte[13], Rational Rose[14], etc. Les modèles créés à l'aide de ces outils et exportés au format XMI peuvent ensuite être chargés dans Gaspard.

Gaspard propose également au concepteur une interface graphique pour réaliser l'association entre l'application et l'architecture matérielle ; dans ce cas, le modèle d'association est alors généré de façon automatique par la plate-forme. Néanmoins, l'utilisateur a toujours la possibilité de concevoir son modèle d'association soit en utilisant les éditeurs de modèles basés sur EMF, ou un outil UML avec le profil de l'association.

Enfin, Gaspard s'interface simplement avec un outil de transformations de modèles pour réaliser les opérations de transformations. Cette intégration peut être faite de plusieurs façons :

- par programmation ; les transformations sont réalisées à l'aide d'un programme Java au moyen d'une API permettant de manipuler des modèles. Dans ce cas, Gaspard fait appel directement aux classes d'implémentation des transformations.
- au moyen d'une API permettant l'intégration d'un outil quelconque de transformations de modèles. L'outil de transformations devient une dépendance externe de Gaspard.
- au moyen de plugins ; il existe de nos jours plusieurs outils de transformations de modèles, développés sous formes de plugins Eclipse (ATL [2], Kermeta [6], etc.). L'intégration de tels outils est directe dans la mesure où Gaspard est développé dans la même plate-forme. La version actuelle de ModTransf (en cours de développement) sera disponible, à terme, sous formes de plugins Eclipse.

[11] http ://www.eclipse.org/
[12] http ://www.magicdraw.com/
[13] http ://www.sparxsystems.com/ea.htm
[14] http ://www-306.ibm.com/software/rational

Gaspard offre donc une grande flexibilité par rapport à l'outillage de mise en œuvre des transformations de modèles. Cette flexibilité est d'autant plus importante que le domaine des transformations de modèles reste encore très mouvementé et que des outils de plus en plus aboutis sont en train d'émerger.

4.3.2 Simulation et exécution dans Gaspard

Pour accomplir sa mission d'atelier de prototypage complet de systèmes embarqués, Gaspard offre au concepteur les outils nécessaires à la génération de code et compilation pour produire les systèmes simulables et exécutables sur plusieurs plates-formes. Une fois la génération terminée, l'exécution ou la simulation peuvent se faire à l'intérieur de Gaspard ou en externe.

4.4 Problèmes d'interopérabilité dans Gaspard

L'interopérabilité dans Gaspard revêt deux aspects principaux : l'interopérabilité des langages et l'adaptation des niveaux d'abstractions pour la simulation. Cette dernière inclut également l'adaptation des protocoles de communication.

4.4.1 Interopérabilité des langages

Comme on peut le constater sur la figure 4.1, Gaspard vise la réalisation des systèmes dans plusieurs plates-formes (au sens langages) d'exécution et de simulation. Sur la figure apparaîssent les plates-formes : *High perf*, *KPN*, et *SystemC*, pour le niveau d'abstraction TLM et *Verilog*, *VHDL*, et *SystemC*, pour le niveau RTL.

Le modèle de KPN (Kahn Process Networks) ou (Réseaux de processus de Kahn) a été proposé par Kahn et MacQueen [74, 75] pour représenter des applications parallèles. Dans ce modèle, les processus qui s'exécutent indépendamment les uns des autres communiquent par des canaux de communication de type FIFO. Dans [14], Abdelkader a proposé une approche basée sur les KPN pour faire de la simulation fonctionnelle et distribuée d'application ARRAY-OL. Le principal intérêt d'une telle approche dans Gaspard est le fait qu'elle permet d'utiliser des IP à distance via une architecture CORBA. En effet dans le domaine des systèmes embarqués, les industriels sont de plus en plus amenés à procéder à des simulations avant de passer à l'étape finale de conception. Lors de cette étape de simulation, les concepteurs des systèmes embarqués et les fournisseurs de composants (IP, notamment) sont confrontés à deux contraintes principales :

- la localité ; les simulateurs de composants et les applications de simulation ne sont pas forcément au même endroit,
- la propriété industrielle, le fournisseur doit protéger le simulateur du composant et le développeur doit protéger l'application simulée.

De là, est née l'idée de la « cyber-entreprise » qui permet d'interconnecter un ensemble de simulateurs, couplés via Internet, pour fournir un ensemble de services.

Le High Perf (High Performance Computing) regroupe un ensemble d'outils pour le calcul scientifique à haute performance permettant de décrire :

- des modèles de programmation parallèle, passage de messages (MPI, par exemple),
- parallélisme de données (exemples : Fortran90, HPF),
- mémoire partagée(OpenMP, par exemple).

Ces outils sont utilisés pour du calcul scientifique distribué sur des grilles de processeurs. Des travaux en cours actuellement ciblent la génération de code parallèle pour la simulation de machines électro-magnétiques.

SystemC, VHDL et Verilog sont des plates-formes de co-simulation logiciel/matériel. SystemC est essentiellement une bibliothèque C++ utilisée pour modéliser des systèmes concurrents en C++. Il fournit une notion de temps et un environnement événementiel de simulation. Sa nature, à la fois séquentielle et concurrentielle, permet de l'utiliser pour décrire et intégrer des composants matériels et logiciels. Il peut donc dans une certaine mesure être considéré comme un langage de description de matériel. Mais contrairement à VHDL et Vérilog, SystemC fournit des mécanismes sophistiqués pour permettre une spécification de haut niveau des interfaces de composants.

Nous venons de passer en revue quelques langages, et nous avons montré leur importance dans le flot de conception Gaspard. Cette multiplicité de langages permettant chacun de traiter des aspects particuliers de Gaspard et définissant ensemble le système complet à concevoir, nécessite d'apporter une solution pour leur interopérabilité.

4.4.2 Interopérabilité des niveaux de simulation

Comme on peut le constater sur la figure 4.1, *Gaspard* est un environnement de simulation multi-niveaux. Sur la figure, nous faisons apparaître seulement les niveaux TLM et RTL par souci de simplicité. Le niveau TLM, lui-même, se subdivise en plusieurs autres niveaux d'abstractions (TLM CABA, TLM PVT, etc.). Plusieurs raisons justifient cette multiplicité de niveaux. Pour comprendre cela, il convient d'abord de se poser la question de savoir ce qu'est un bon environnement de simulation.

Les principales caractéristiques requises pour la simulation efficace des systèmes hétérogènes embarqués sont :

- la flexibilité ; la spécification exécutable (modèle de simulation) doit pouvoir s'adapter « assez facilement » à des changements éventuels dans le flot de conception (méthodologie, outils, technologies, etc.),
- la modularité ; la possibilité d'avoir une vision « composants » ou « modules » du système à simuler, permet d'observer le système sous différents points de vues : hiérarchies, boîtes noires, détails dans un composant, etc.
- l'extensibilité ; la possibilité de la validation du système dans le cas d'intégration de nouvelles fonctionnalités et/ou de nouveaux composants,
- la précision ; le modèle doit être capable de valider les deux critères habituellement observés lors des simulations : le fonctionnement et le temps. Et ce, en fonction des niveaux d'abstractions de la description des différents composants du système.

Nous analysons maintenant nos choix à la lumière de ces caractéristiques. Dans la simulation d'un système, le concepteur fait toujours face à deux objectifs antagonistes : la vitesse et la précision. D'une part, la simulation doit pouvoir se réaliser en un temps raisonnable pour

être applicable, et d'autre part, elle doit fournir suffisamment d'informations pour guider les choix de conception et de réalisation du système. Plus la précision est grande, plus le temps de simulation est long. Il faut donc trouver un compromis entre les deux. Le but des niveaux de simulation est justement de pouvoir définir des degrés de précision de la simulation. Nous illustrons notre propos en prenant l'exemple de *SystemC*[15].

SystemC permet de modéliser des systèmes matériels et logiciels à l'aide de C++. Il s'agit d'une bibliothèque de classes implémentées dans le langage C++ qui fournit des constructions permettant de modéliser les aspects matériels d'un système. SystemC propose plusieurs niveaux de description du comportement des systèmes que nous décrivons ci-dessous.

- BCA (Bus Cycle Accurate) : s'applique à l'interface d'un modèle. Il signifie que la simulation des transactions sur l'interface est correcte au cycle près. Un modèle BCA n'apporte aucune information sur les bits (signaux) de l'interface.
- BA (Bit Accurate) : s'applique à l'interface et à la fonctionnalité d'un modèle. Il signifie que la modélisation des transactions sur l'interface est précise au bit (fil) près. Le terme BA est aussi appelé CABA (cycle accurate / bit accurate).
- UTF (UnTimed Functional) : s'applique à l'interface et à la fonctionnalité d'un modèle. Le modèle ne comporte aucune notion de durée d'exécution, mais seulement un ordre éventuel dans l'exécution des événements. Chaque événement s'exécute en un temps nul. Seul compte l'ordonnancement des événements.
- TF (Time Functional) : s'applique à l'interface et à la fonctionnalité d'un modèle. Le modèle comporte des notions de durée (temps d'exécution des processus, latence, temps de propagation, etc.)
- RTL (Register Transfert Level) : s'applique à l'interface et à la fonctionnalité d'un modèle matériel. Chaque bit, chaque cycle, chaque registre du système est modélisé.

Le fait, donc, de disposer de plusieurs niveaux permet de décrire certaines parties du système avec moins de détails pour accélérer la simulation. C'est comme une sorte de paramétrage de la simulation.

Par ailleurs, l'utilisation d'IP hétérogènes dans *Gaspard* nous contraint à gérer plusieurs niveaux de simulation. En effet, les IP sont conçues pour fonctionner dans des niveaux d'abstractions bien définis ; l'utilisateur doit donc pouvoir s'adapter.

Enfin, dans le contexte d'exploration d'architecture, on peut vouloir tester très rapidement un composant pour savoir s'il peut être fonctionnel dans le système en construction. Il est alors intéressant de pouvoir, dans un premier temps, faire une modélisation à un niveau plus élevé (plus rapide), de tester la fonctionnalité du composant avant d'entreprendre une description plus détaillée. Ici, l'utilisation de plusieurs niveaux d'abstractions nous permet de faire l'économie du temps de modélisation.

En conclusion, trois facteurs justifient notre choix de simulation multi-niveaux : la rapidité, la disponibilité des IP et l'économie (gain) en effort de modélisation.

Cependant, la prise en charge de la simulation à plusieurs niveaux d'abstractions ne va pas de soi ; en effet, les interfaces des composants utilisés varient suivant les niveaux d'abstractions. Par exemple, en TLM PV, l'interface de communication est constituée seulement de deux fonctions (lecture et écriture), alors qu'au niveau RTL l'interface est beaucoup plus dé-

[15] www.systemc.org

taillée, allant jusqu'au niveau de fils par lesquels transitent des bits. Pour prendre en compte des simulations multi-niveaux, il est donc indispensable d'adapter les différentes interfaces. Cette adaptation est réalisée essentiellement par des wrappers (adaptateurs).

Au chapitre 7, nous présentons quelques approches de génération de wrappers et une démarche de leur génération automatique par transformations de modèles dans le cadre de Gaspard.

4.4.3 Vision Gaspard pour la mise en œuvre de l'interopérabilité

La solution idéale pour la résolution de l'interopérabilité dans Gaspard serait de générer automatiquement les interfaces de communication relatives aux deux aspects d'interopérabilité, par application de l'approche IDM. Cette génération se ferait donc par des opérations de transformations de modèles et cela permettrait de décharger complètement les concepteurs des aspects liés à l'interopérabilité.

4.5 Conclusion

Dans ce chapitre, nous avons présenté les différentes facettes de la démarche Gaspard. Cette démarche définit à la fois une méthodologie et l'outillage nécessaire à sa mise en œuvre. Plusieurs niveaux d'abstractions permettent de décrire différents aspects des systèmes au moyen de métamodèles. Les passages entre les niveaux d'abstractions sont réalisés par des transformations de modèles. Il s'agit véritablement d'un processus de développement piloté par les transformations de modèles.

Nous avons également abordé la notion de simulation multi-niveaux dans Gaspard : la simulation d'un système à différents niveaux d'abstractions, dans différents langages (figure 4.1) et éventuellement de façon distribuée. Cette diversité des plates-formes et des langages cibles nécessite de prendre en compte le problème d'interopérabilité identifié sur la figure par le pont d'interopérabilité (*interop Bridge*). La contribution principale de cette thèse se situe au niveau de la conception de ce pont d'interopérabilité.

Gaspard est un couplage de méthodologie et d'outillage et sera, à terme, un outil IDM pour la conception de SoCs. Il permettra de créer très rapidement des prototypages, de les simuler, de se servir des résultats des simulations pour améliorer la conception avant de passer à la réalisation physique du SoC.

Dans cette présentation de Gaspard, nous avons montré que Gaspard utilise plusieurs langages (Fortran, SystemC, KPN, etc.) et plusieurs niveaux de simulation (TLM, RTL). Cela pose deux problèmes d'interopérabilité : l'interopérabilité des langages et l'interopérabilité des niveaux de simulation. Les deux problèmes sont de natures différentes. En effet, au niveau des langages, le problème est de permettre la communication entre des objets par échange de messages ou appels de méthodes. Ces communications peuvent être modélisées aisément par un métamodèle. Pour ce qui est de l'interopérabilité des niveaux de simulation, nous avons affaire à des descriptions d'interfaces matérielles dotées de leur propre protocole de communication. Pour conceptualiser l'interopérabilité dans ce cas, il faut pouvoir modéliser les interfaces et les protocoles de communication. À l'heure actuelle, il n'existe pas de

solution unifiée pour répondre à cette modélisation. Nous avons donc décidé de considérer les deux problèmes séparément. Ainsi, nous proposons au chapitre 6 l'interopérabilité des langages et l'interopérabilité des niveaux de simulation. Ces deux problèmes ne sont pas de la même nature ; au niveau des langages, le problème est de permettre la communication entre des objets par échange de messages ou appels de méthodes. Alors que dans le second cas ; l'interopérabilité des niveaux de simulation, nous avons affaire avec des descriptions d'interfaces matérielles dotés de leur propre protocole de communication. Aussi les solutions à ces deux problèmes diffèrent ; nous proposons au chapitre 6 une solution pour le premier problème et le second sera traité au chapitre 7.

Chapitre 5

Transformations de Modèles dans Gaspard

Comme dans toute approche d'ingénierie dirigée par les modèles, les transformations de modèles constituent une clé dans le succès de l'approche Gaspard. En effet, elles opérationnalisent la productivité des modèles et réalisent le pont entre la modélisation et la production des artefacts nécessaires à l'implémentation des systèmes. Au chapitre 2, nous avons présenté les principales approches de transformations de modèles. Dans ce chapitre, nous nous intéressons aux transformations de modèles dans le contexte particulier de Gaspard.

Dans la première section, nous décrivons de façon générale les transformations de modèles dans Gaspard et dans la seconde section, nous présentons les différentes expérimentations de transformations de modèles que nous avons réalisées dans Gaspard.

5.1 Description générale des transformations de modèles dans Gaspard

Au chapitre 4, nous avons présenté le flot de conception de Gaspard et les différents métamodèles impliqués dans ce flot. Basé sur l'approche MDE, le passage des modèles de haut niveau vers les modèles de plus bas niveau d'abstraction se fait par des transformations de modèles.

Il existe cinq types de transformations de modèles dans Gaspard ; nous les identifions sur la figure 5.1 par les labels *T0,T1,T2,T3 et T4*. Chaque type de transformation correspond à une étape importante dans le processus de développement Gaspard.

L'utilisateur de la plate-forme Gaspard peut utiliser un outil UML avec le profil Gaspard pour créer le modèle de son application, le modèle de l'architecture matérielle et le modèle d'association correspondant. Les modèles ainsi créés sont des instances d'UML. Ils sont ensuite traduits en des modèles instances des métamodèles Gaspard correspondants. Cette traduction est réalisée par les transformations de type *T0*. Ces transformations sont automatiques et sont exécutées lors de l'importation d'un modèle UML dans l'environnement Gaspard.

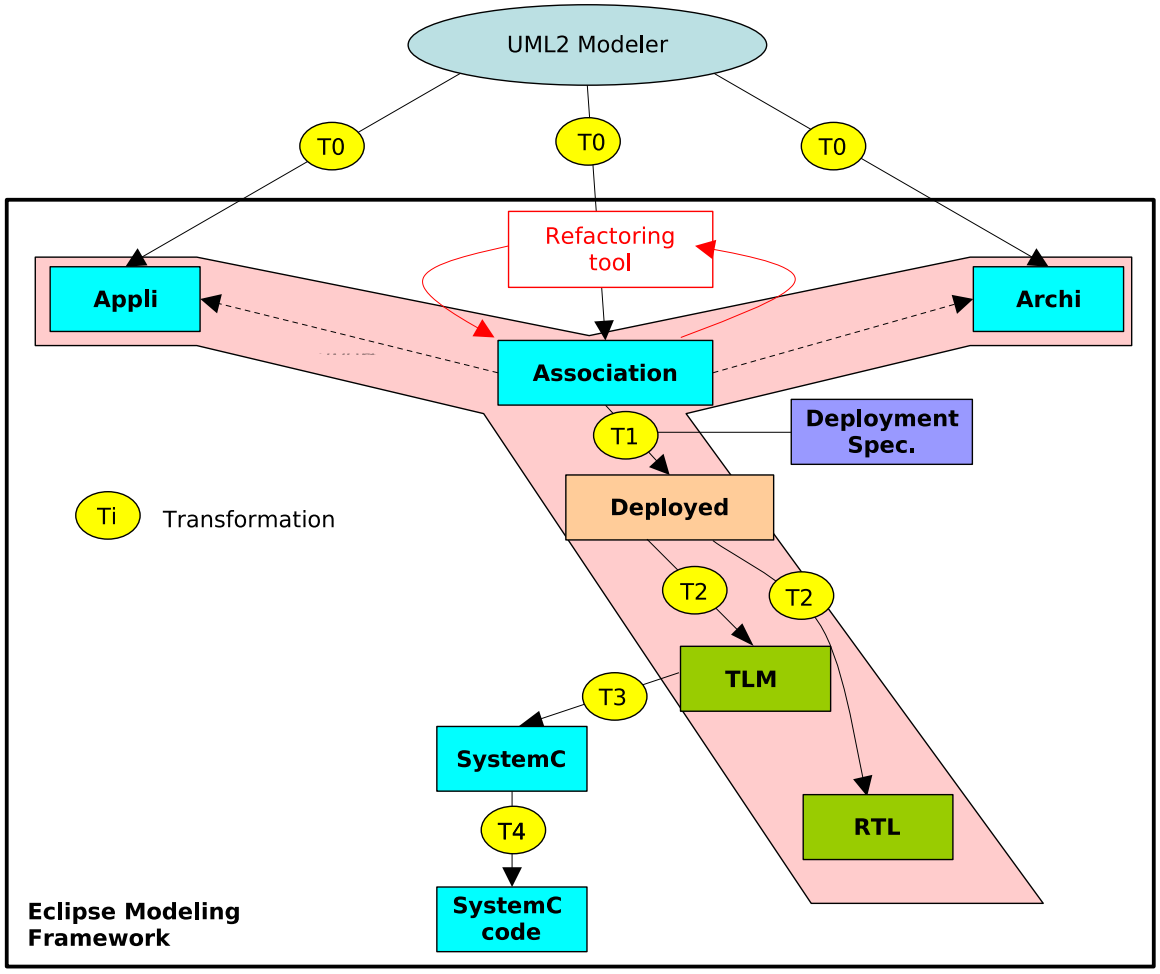

FIG. 5.1 – Transformation de déploiement

La transformation *T1* réalise le déploiement. Le modèle d'association est enrichi par des informations de déploiement. Cet enrichissement se fait actuellement à la main. On peut envisager des outils proposant un déploiement automatique en fonction des informations fournies par les concepteurs. Après cette opération de fusion, le modèle obtenu correspond au système déployé.

Les transformations de type *T2* permettent à partir du modèle déployé de générer de façon automatique les modèles abstraits de simulation (comme TLM et RTL).

Les transformations de type *T3* génèrent les modèles des plates-formes de simulation (SystemC par exemple) ou de langage d'implémentation.

Enfin, la dernière classe de transformation (type 4) correspond à des générations de codes sources pour les différents langages.

Les transformations de modèles dans Gaspard permettent donc d'effectuer des raffinements successifs des modèles définis à un haut niveau d'abstraction vers des modèles de niveau d'abstraction plus bas. A la fin du processus Gaspard, les codes sources du système sont générés et le système est prêt pour être simulé ou exécuté.

5.2 Expérimentations de transformations de modèles dans Gaspard

Au cours de nos travaux, nous avons envisagé les transformations de modèles dans Gaspard de diverses manières. En l'occurrence, nous avons réalisé deux expérimentations sur les transformations de modèles. D'une part, nous avons utilisé le moteur de transformations *ModTransf* [42] et d'autre part, nous avons programmé les transformations en Java avec l'API JMI[16] [70]. Nous allons passer en revue les deux expérimentations et montrer pourquoi il nous a été nécessaire d'envisager une autre approche pour le cas spécifique de Gaspard.

5.2.1 Expérimentation avec ModTransf

ModTransf est un outil de transformations de modèles développé dans le *Laboratoire d'Informatique Fondamentale de Lille (LIFL)* au sein de l'équipe *West*, disponible en open source et téléchargeable à : *http ://modelware.inria.fr/rubrique15.html*. Il a été réalisé sur la base des recommandations issues de la révision des premières propositions à la *RFP* QVT [119] et les propositions subséquentes [98]. Sur la base de ces recommandations et des besoins de l'équipe, les caractéristiques attendues de ModTransf se résument comme suit :

- la possibilité de transformer plusieurs types de modèles MOF, XML, graphe d'objets),
- la réalisation de transformations avec plusieurs modèles d'entrée et de sortie,
- le moteur doit être simple à utiliser,
- la maintenance des règles de transformations doit être facile à réaliser,
- les règles de transformation doivent pouvoir être exprimées de façon déclarative ou impérative,
- l'héritage entre règles doit être possible pour permettre d'écrire des règles à partir d'autres règles déjà définies,
- les règles doivent être réversibles lorsque cela est possible,
- un mécanisme d'extension doit permettre d'utiliser des règles spécifiques implantées dans un langage de programmation (en l'occurrence Java),
- le moteur doit pouvoir faire de la génération de code.

ModTransf a été développé dans le souci de répondre à tous ces besoins. Son principe et son fonctionnnement sont illustrés par la figure 5.2. Il fonctionne à base de règles, c'est-à-dire qu'un ensemble de règles spécifient des *mappings* entre les concepts des métamodèles d'entrée et ceux des métamodèles de sortie et décrivent les actions permettant de réaliser la création des concepts de sortie. Une transformation consiste donc à soumettre au moteur un concept d'entrée ; le moteur recherche la règle appropriée et l'applique pour créer le ou les concept(s) correspondant(s) dans le ou les modèle(s) de sortie. La génération de code suit le même principe à la seule différence que la création des concepts de sortie est substituée à une écriture de code à l'aide d'un mécanisme de *templates*. Dans ce dernier cas, une règle spécifie un ou plusieurs templates à utiliser, chacun des templates contenant des fragments de texte qui sont remplacés par les valeurs des concepts d'entrée.

Les modèles manipulés par ModTransf sont représentés en mémoire sous forme de graphes d'objets. Un objet de ce graphe représente un concept du métamodèle correspondant. Ce concept peut posséder des propriétés qui sont ses objets fils dans le graphe. Ainsi,

[16] http ://java.sun.com/products/jmi/download.html

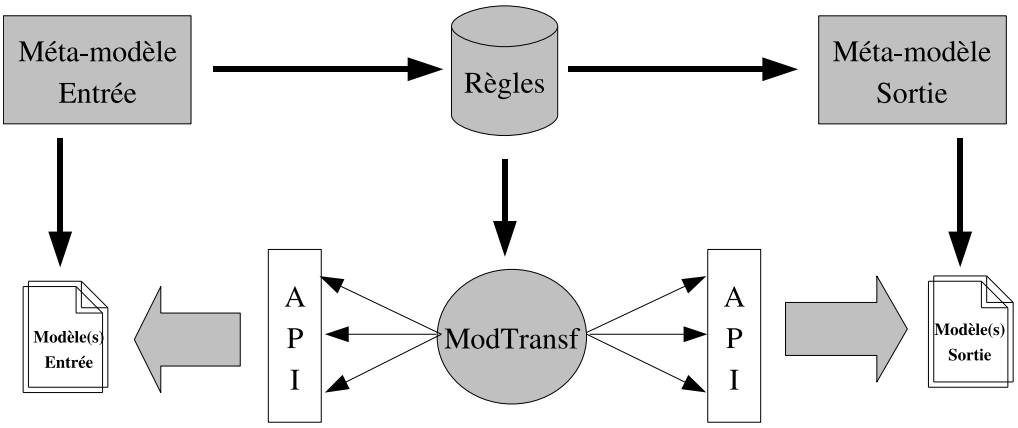

FIG. 5.2 – Vue d'ensemble de ModTransf

un modèle est défini par ses concepts et leurs propriétés qui peuvent récursivement être des concepts. La transformation d'un modèle revient donc à soumettre au moteur le concept racine du graphe d'objets associé ; le moteur recherche la règle la plus adéquate, laquelle rappelle le moteur pour transformer les objets fils et, ainsi, tous les objets du graphe seront visités et transformés. Pour chaque objet, le moteur trouve et applique la règle appropriée.

Comme l'indique la figure 5.2, ModTransf peut être interfacé avec des APIs permettant de manipuler les modèles. Ces APIs permettent d'utiliser différentes technologies telles que JMI dans les implémantations (MDR, NsUML, CIM, ModFact, etc.), EMF, DOM, etc. Chaque implémentation d'API est propre à une technologie de représentation de modèles et est réalisée une fois pour toutes. Ce mécanisme permet à l'utilisateur de pouvoir étendre ModTransf et réaliser des transformations sur ses propres modèles de graphes d'objets.

Définition des transformations dans ModTransf

La notion de règle de transformations se trouve au cœur de la définition de transformations avec ModTransf. Les règles sont exprimées dans une syntaxe XML et sont à la base constituées de deux parties : une partie *garde* et une partie *action*. La partie garde décrit les conditions d'application de la règle et la partie action spécifie l'exécution de celle-ci. Cette structure simple constitue la base et peut être utilisée pour construire des règles plus complexes.

Les règles peuvent être organisées en ensembles, lesquels deviennent alors des unités de recherches pour le moteur. Ainsi, il est possible de préciser un ensemble particulier de règles à utiliser dans une transformation. Ce concept d'ensemble de règles peut donc être utilisé pour réduire le champ de recherche des règles, ou définir des règles qui seront utilisées dans un contexte particulier. Enfin, il est possible de spécifier explicitement au moteur la règle à utiliser. Dans ce dernier cas, la transformation est dite impérative et le moteur ne fait pas de recherche ; il applique directement la règle en question. La recherche des règles se fait dans l'ordre de déclaration des règles, et seule la première règle adéquate trouvée est exécutée. Ce comportement par défaut peut être modifié en indiquant explicitement au moteur d'appliquer toutes les règles satisfaisantes, c'est-à-dire, toutes celles pour lesquelles la garde est vérifiée. Le code ci-après montre un exemple de règle de transformations écrite dans la syntaxe d'écriture des règles dans ModTransf.

Règle de transformation d'une classe UML en classe Java

```
<rule name="class">
  <description> Transform a class to a class</description>
  <domain type="Core.Class" model="uml" >
   <primitiveProperty name="name" varName="n" type="String"/>
   <collProperty name="feature" varName="feature"/>
  </domain>
  <domain type="JavaClass" model="java" >
   <primitiveProperty name="name" varName="n" type="String"/>
   <collProperty name="attributs" varName="contents" type="Attribute"/>
   <collProperty name="methods" varName="contents" type="Method"/>
  </domain>
  <actions>
    <call rule="feature2contents">
      <arg expr="feature"/>
      <arg expr="contents"/>
    </call>
  </actions>
</rule>
```

Une règle est écrite à l'aide de tags XML. Chaque tag possède des attributs ; par exemple, l'attribut *name* du tag *rule* permet de nommer la règle de transformation. Cette règle définit la transformation d'une classe UML en une classe Java.

Les tags *domain* définissent les concepts d'entrée et de sortie de la règle. Les attributs *type* et *model* spécifient respectivement les concepts d'entrée ou de sortie, respectivement les métamodèles d'entrée ou de sortie. Dans cette règle, nous avons un domaine d'entrée et un domaine de sortie. Dans chaque domaine, on peut caractériser les propriétés. La liaison entre les propriétés du domaine d'entrée et celles du domaine de sortie est faite par une variable de liaison (exemple : *varName="feature"*).

Le tag *actions* décrit les opérations réalisées par l'exécution de la règle. Ici, en l'occurrence, l'action *call* permet d'appeler la règle nommée *feature2contents*. Les tags fils *arg* permettent de donner le contexte d'exécution de la règle.

Synthèse

Dans cette première expérimentation des transformations [25, 24], nous avons pu comprendre et tirer un certain nombre de leçons par rapport aux transformations de modèles dans Gaspard. En l'occurrence, nous avons pu constater que ModTransf était particulièrement efficace dans les transformations de type *1 vers 1* et la génération de code. Nous entendons par transformations de type *1 vers 1*, des transformations où un concept du modèle d'entrée est transformé en un concept du modèle de sortie. Cette limitation de l'outil est surtout lié à son formalisme d'expression des règles. Le moteur, par construction, est suffisamment efficace pour réaliser toutes sortes de transformations de modèles (y compris des transformations n vers n), mais la syntaxe d'expression des règles basée sur le XML rend l'ecriture des ces dernière difficile. Il est, par exemple, très difficile dans cette syntaxe de décrire des transformations impliquant des calculs algorithmiques. Or, dans le cas typique des transformations dans Gaspard, de telles situations sont très fréquentes (calcul de matrice de pavage, d'ajustage, des espaces d'itérations, etc.) compte tenu du facteur *calcul scientifique* des applications

Array-OL à la base de Gaspard. Le mécanisme d'extension proposé dans ModTransf devrait permettre de résoudre ce problème, mais l'on constate que l'application de ce mécanisme dans le cadre des transformations Gaspard conduit à réaliser beaucoup d'extensions.

Par ailleurs, le modèle d'exécution de ModTransf ne permet pas de réaliser de façon simple des transformations incrémentales, c'est-à-dire, des transformations effectuées en plusieurs étapes avec éventuellement des modèles intermédiaires incomplets.

Sur la base de ces observations, une nouvelle version du moteur prenant en compte tous ces besoins est en cours de réalisation. En attendant la fin de sa réalisation, nous avons exploré d'autres pistes que nous présentons ci-dessous.

5.2.2 Expérimentation avec Java et l'API JMI

Le standard MOF propose une méthode pour manipuler les modèles à l'aide d'interfaces de programmation. Ces interfaces sont générées à partir d'un méta-modèle et permettent de consulter, créer, supprimer ou modifier les objets instances des modèles [113]. La figure 5.3 présente le mécanisme de manipulation des modèles via des interfaces de programmation.

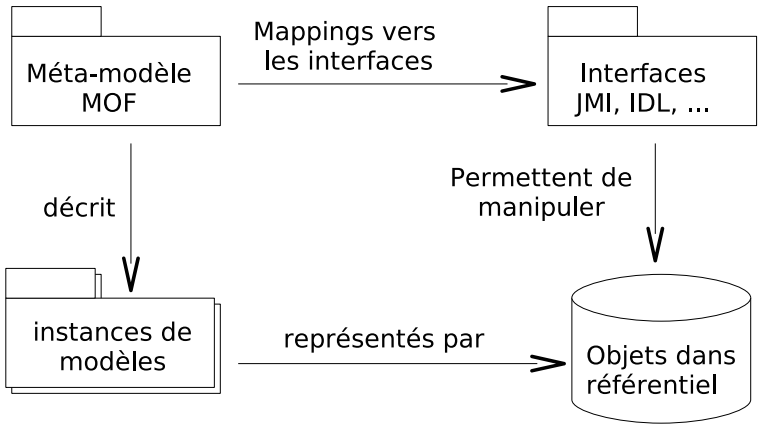

FIG. 5.3 – Manipulation de modèles via des interfaces de programmation

JMI est une spécification de mappings des concepts MOF vers des interfaces Java. Il propose deux types d'interfaces de manipulation de modèles : les interfaces dédiées aux méta-modèles et les interfaces réflexives.

Les premières sont générées à partir d'un méta-modèle spécifique et servent à manipuler uniquement les modèles instances du méta-modèle.

Les deuxièmes sont génériques et permettent de manipuler tous les modèles instances de n'importe quel méta-modèle.

Pour notre expérimentation, nous avons utilisé les interfaces dédiées puisque nous nous intéressions particulièrement à la manipulation de modèles instances des méta-modèles Gaspard.

Il existe plusieurs outils (l'implémentation de référence de JMI par Unisys [120], ModFact [1], Meta Data Repository : MDR [103], etc.) qui permettent de générer des interfaces JMI

et leur implémentation à partir d'un méta-modèle MOF. Nous avons utilisé MDR pour la génération de nos interfaces.

MDR, comme son nom l'indique, est un référentiel de méta-données. Il implémente le MOF et, de ce fait, permet de charger tout méta-modèle MOF (description des méta-données) et de stocker les instances de ce méta-modèle. Les méta-modèles peuvent être importés et exportés via des documents XML conformes au standard XMI. Les méta-données contenues dans le référentiel peuvent alors être gérées par programmation en utilisant l'API JMI.

Pour utiliser MDR, il faut commencer par définir un méta-modèle qui décrit les méta-données. Ce méta-modèle constituant le modèle du langage de modélisation peut être réalisé à l'aide d'outils UML compatibles XMI tels que MagicDraw, Poseidon, etc. Le modèle UML créé doit être traduit en MOF. Cette opération peut être réalisée de plusieurs façons : soit à l'aide de l'outil UML, si ce dernier sait exporter en MOF, soit à l'aide de l'outil *UML2MOF*[17] ou en utilisant tout autre outil générant du MOF à partir d'un modèle UML. Le modèle MOF peut alors être chargé dans MDR qui génère les interfaces JMI pour accéder aux méta-données. MDR peut également fournir les implémentations de ces interfaces de façon automatique. Pour assurer la synchronisation entre les clients et le référentiel, l'implémentation des interfaces JMI fournie par MDR prend en charge la notification de changements aux différents clients concernés.

Une présentation détaillée de MDR sort du cadre de notre travail, mais le lecteur intéressé pourra trouver plus de documents sur le site web du projet (*http ://mdr.netbeans.org*).

Implémentation Java / JMI

L'implémentation de transformations de modèles par programme Java utilisant l'API JMI est schématisée dans la figure 5.4.

Dans cette approche, la réalisation des transformations de modèles revient à l'implémentation d'un programme Java. La particularité de tels programmes réside dans l'utilisation des fonctions spécifiques fournies par l'API JMI. En l'occurrence, le développeur a besoin de fonctions pour créer, lire et écrire des modèles. Les principales fonctions fournies par l'implémentation MDR de JMI sont résumées dans le tableau 5.1 où *mm* et *Element* désignent respectivement un méta-modèle et un concept (élément) de méta-modèle.

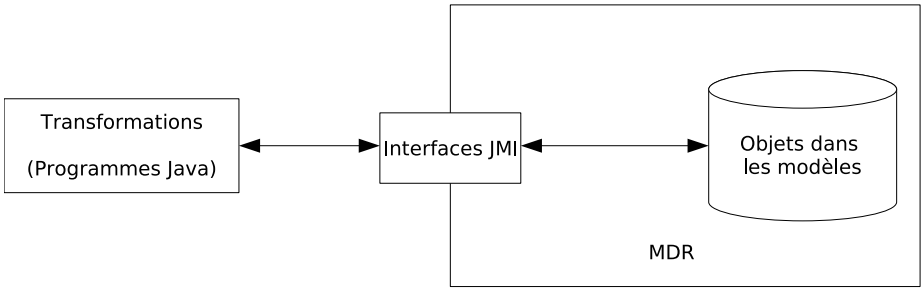

FIG. 5.4 – Programmation de transformations avec Java/JMI

[17]UML2MOF est un outil permettant de transformer des modèles UML 1.4 vers du MOF 1.4. Il est disponible en open source à *http ://mdr.netbeans.org/uml2mof/*

Fonctions	Rôles
mmModelUtil.readModel	permet de lire un fichier XMI contenant un modèle et de le charger dans le référentiel. Le fichier XMI et l'identifiant du modèle dans le référentiel sont passés en paramètres.
mmModelUtil.writeModel	permet d'écrire un modèle contenu dans le référentiel dans un fichier XMI. Le nom du fichier XMI en sortie, ainsi que l'identifiant du modèle dans le référentiel sont passés en paramètres.
mmModelUtil.createModel	Crée un modèle vide, instance du méta-modèle dans le référentiel. Cette instance vide peut ensuite être associée à un modèle existant sous forme de fichier XMI à l'aide de la fonction *mmModelUtil.readModel*.
mmModelUtil.getRoots	permet d'obtenir un itérateur sur l'ensemble des éléments racines contenus dans un modèle. L'identifiant du modèle dans le référentiel est passé en paramètre de la fonction.
mmModelUtil.getElement.createElement	permet de créer une instance de l'élément *Element* d'un modèle dans le référentiel.

Tab. 5.1 – Principales fonctions de l'API JMI

La figure 5.5 donne un extrait du code Java permettant de réaliser une transformation du méta-modèle d'Association de Gaspard vers le méta-modèle TLM. Comme le montre la figure, dans le code de la fonction *main*, le programme Java est appelé en passant en paramètres deux chaînes de caractères. Le premier paramètre correspond au nom du fichier XMI contenant le modèle d'association en entrée de la transformation, et le second correspond au nom du fichier XMI qui sera généré en résultat de la transformation. Une instance de la classe de la transformation *scanModel* est créée à l'aide du constructeur *Assoc2TLM*. Sur cette instance sont exécutées dans l'ordre deux procédures : la première, *run*, est la procédure la plus importante de cette classe ; elle réalise toute l'opération de transformation en cinq étapes comme on peut le voir dans les commentaires du code. La seconde procédure, *generateModel*, écrit simplement dans un fichier XMI en sortie le modèle généré par la transformation et contenu en mémoire dans le référentiel.

Synthèse

Comme on a pu le constater à travers le code de la classe de transformations du modèle d'association vers le modèle TLM présenté précédemment, l'API JMI permet de réaliser des transformations de modèles. Elle rend les transformations de modèles accessibles à tout programmeur Java ; aucune autre compétence technique n'est requise. L'utilisation du langage Java présente un autre avantage. En effet, elle fait bénéficier le concepteur de l'utilisation des outils de développement Java (éditeurs, déboggeurs, etc.) qui existent déjà et de la maturité des approches de développement orienté objet. Enfin, l'utilisation d'un langage de

```
/**
 * This class performs the transformation from
 * an association model to the tlm model
 *
 * @author Lossan BONDE
 */
public class Assoc2TLM {

private UMLPackage extentIn;
public TLMGaspardPackage extentOut;
private Main mainSysc ;
private String sysc;
private String assoc;
/**
 * @param application
 *            the association model file
 */

public Assoc2TLM(String association, String sysc) {
this.assoc = association;
this.sysc = sysc;
extentIn = IspModelUtil.createModel();
IspModelUtil.readModel(association, extentIn);
extentOut =(TLMGaspardPackage)SysCModelUtil.createModel
("fr.lifl.west.gaspard.tlm.impl.core.TLMGaspardPackageImpl");
}

public void run() {
/*
 * Step 1 : creation of the concepts from Application and
 *  Architecture models
 */
mainSysc = (Main) extentOut.getMain().createMain();
Iterator roots = null;
roots = IspModelUtil.getRoots(extentIn).iterator();
while (roots.hasNext()) {
buildNode(roots.next());
}
/*
 * Step 2 : Linking the concepts created in step 1
 */
Iterator roots1 = null;
roots1 = IspModelUtil.getRoots(extentIn).iterator();
while (roots1.hasNext()) {
linkNodes(roots1.next());
}

/*
 * Step 3 : handling TaskAllocation from the Association model
 */
Iterator roots2 = null;
roots2 = IspModelUtil.getRoots(extentIn).iterator();
while (roots2.hasNext()) {
handleTaskAllocation(roots2.next());
}

/*
 * Step 4 : handling DataAllocation from the Association model
 */
Iterator roots3 = null;
roots3 = IspModelUtil.getRoots(extentIn).iterator();
while (roots3.hasNext()) {
handleDataAllocation(roots3.next());
}

/*
 *Step 5 : handling Scheduling from the Association model
 */
Iterator roots4 = null;
roots4 = IspModelUtil.getRoots(extentIn).iterator();
while (roots4.hasNext()) {
handleScheduling(roots4.next());
}
}

public static void main(String[] args) {

String assoc = args[0];
String sysc = args[1];
Assoc2TLM scanModel = new Assoc2TLM(assoc,sysc);
scanModel.run();
scanModel.generateModel();
}
```

FIG. 5.5 – Exemple de Classe Java de transformations

programmation comme Java permet aux développeurs des transformations de disposer de tout le pouvoir d'expression de Java pour implémenter leurs transformations. Il est donc théoriquement possible d'écrire toutes sortes de transformations de modèles.

Néanmoins, dans le contexte de l'ingénierie dirigée par les modèles, cette approche de développement des transformations n'est pas satisfaisante pour deux raisons majeures. Premièrement, dans cette approche, le développement des transformations se trouve centré sur le code Java. Or, l'un des objectifs de l'IDM est justement de permettre aux développeurs de travailler à un niveau d'abstraction plus élevé que le code. Deuxièmement, il faut noter que les transformations programmées en Java sont difficilement maintenables. En effet, une modification dans les méta-modèles entraîne la modification de plusieurs lignes de code Java.

En résumé, le développement de transformations par programmation Java / JMI est simple, directe et peut, de ce point de vue, être un bon moyen pour apprendre à manipuler et transformer des modèles. Cette approche, bien qu'elle présente les inconvénients cités plus haut, reste toujours utilisée dans la pratique. D'une part, à cause de l'immaturité des outils actuels de transformations de modèles, et d'autre part, à cause de l'apprentissage nécessaire pour comprendre et pouvoir mettre en œuvre des transformations dans les autres approches.

5.3 Conclusion

Le succès de la mise en œuvre de l'approche Gaspard basée sur les techniques de l'ingénierie dirigée par les modèles passe par une maîtrise de la conception et de la mise en œuvre des transformations de modèles. Nous avons montré les différents types de transformations de modèles qui jalonnent le processus de développement Gaspard.

Les expérimentations présentées dans ce chapitre nous ont permis de nous rendre compte de la complexité de réalisation des transformations de modèles, ne serait-ce que dans le contexte particulier de Gaspard. Les difficultés rencontrées avec notre outil de transformations ont été exposées et sont prises en compte dans la réalisation de la version en cours de développement. Nous avons également mis en évidence les problèmes des transformations réalisées dans un langage de programmation comme Java.

Tout cela nous a permis de cerner le besoin fondamental de formalisme de description des transformations de façon indépendante de tout outil. Une piste d'investigation dans ce sens consisterait à utiliser le standard MOF 2.0 QVT proposé par l'OMG comme langage de conception. En particulier, la notation graphique du *Relations Language* couplée avec les implémentations *Black Box* présentent des perspectives intéressantes. Maintenant qu'apparaîssent les implémentations QVT : [9] pour la partie impérative et [110] pour la partie déclarative, l'investigation de cette piste peut apporter une réponse intéressante.

Chapitre 6

Conception de l'interopérabilité entre langages dans Gaspard

6.1 Introduction

Au chapitre 4, nous avons identifié deux besoins d'interopérabilité dans Gaspard : l'interopérabilité des langages et l'interopérabilité des niveaux de simulation. Nous abordons ici le premier aspect ; le second sera traité au chapitre 7.

La figure 6.1 rappelle le flot de conception dans *Gaspard*. Dans ce schéma, nous nous intéresserons à la section qui va du modèle déployé (*Deployed*) jusqu'aux systèmes de simulation niveaux *TLM* et *RTL*. Le modèle initial (*Deployed*, que nous appelons ici modèle de référence) est soumis à deux transformations : une transformation vers *TLM* et une autre vers *RTL*. Chacun des deux modèles, indépendants de toute plate-forme de simulation, est projeté vers une ou plusieurs plates-formes technologiques permettant la simulation et/ou l'exécution : *SystemC*, *VHDL* (ces projections sont également des transformations de modèles). Une dernière série de transformations de modèles (générations de codes) permet de générer les codes sources des programmes de simulation. Pour que ces programmes de simulation interagissent, il faut une couche d'interopérabilité qui prendra en charge la communication entre les éléments des différents systèmes. Nous allons montrer que l'on peut générer automatiquement cette couche d'interopérabilité à partir de la trace des transformations.

Notre démarche consiste à identifier les liens de communication entre les éléments des différents modèles des niveaux technologiques. Ces liens de communication sont déterminés à partir des relations entre les éléments dans le modèle de référence ; la reconnaissance de ces liens entre les modèles technologiques est assurée par la traçabilité dans les transformations. À partir du modèle de trace nous déduisons un graphe de dépendances entre composants de différents modèles technologiques. Ce graphe de dépendances des composants est utilisé pour générer automatiquement un modèle de pont d'interopérabilité entre les langages. Le modèle de pont contient toutes les informations nécessaires pour générer l'implémentation de l'interopérabilité.

Pour automatiser le processus de construction du pont d'interopérabilité, nous avons conçu un métamodèle de pont d'interopérabilité et un métamodèle de traçabilité. Les ins-

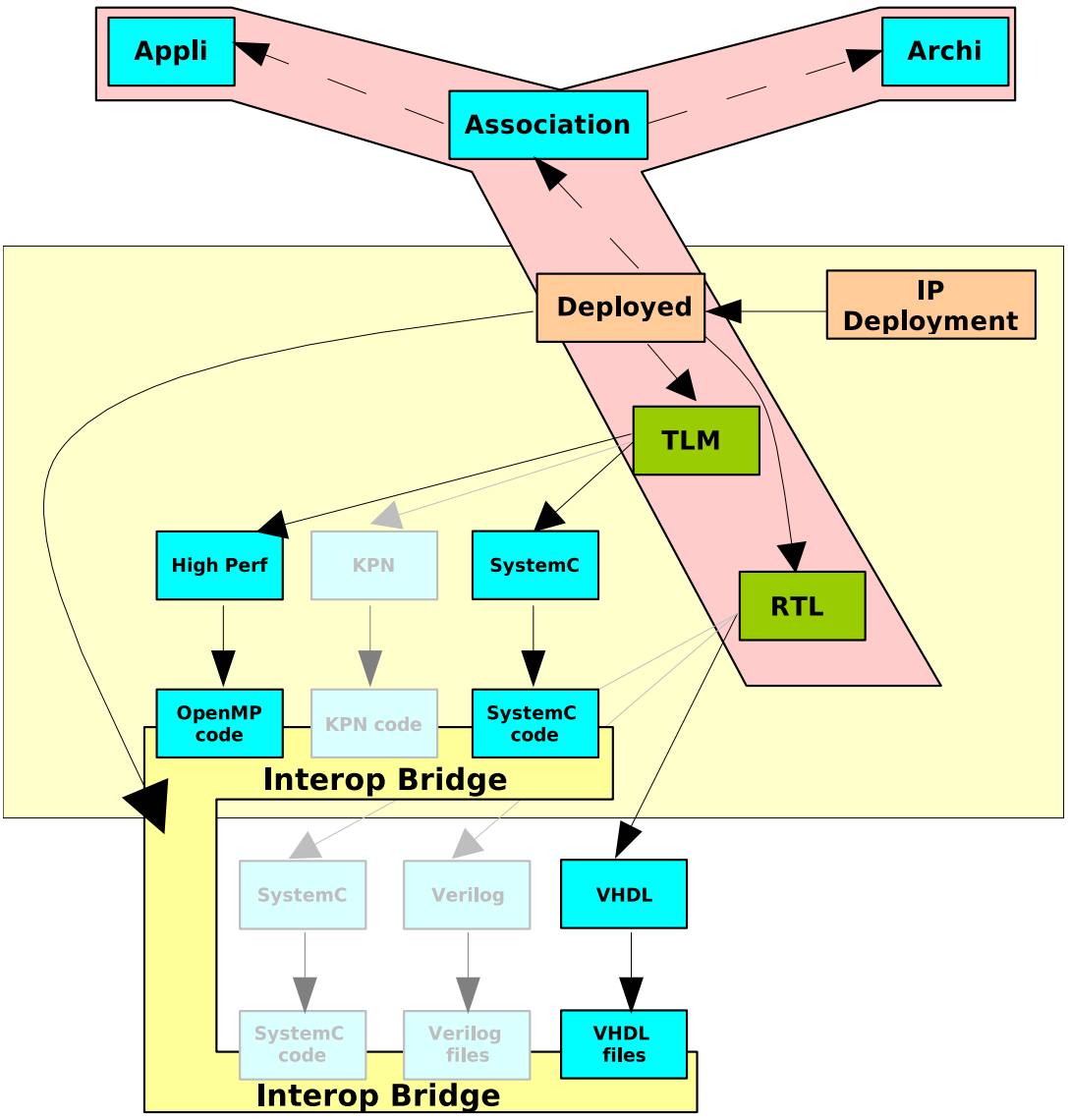

FIG. 6.1 – Flot de conception dans Gaspard

tances du métamodèle de traçabilité (modèle de trace) sont générées pendant les opérations de transformations de modèles. Quant au métamodèle de pont, ses instances (modèle de pont) sont créées par transformations de modèles à partir du modèle de trace. La génération de l'implémentation se fait par génération de code à partir du modèle de pont. La figure 6.2 donne une vision simplifiée de notre proposition de construction du pont d'interopérabilité.

Dans le cadre de cette thèse, nous nous intéressons plus particulièrement à la génération du modèle de pont. Nous ne décrivons pas l'implémentation du pont. Celle-ci dépend d'un choix technologique. Toutefois, nous donnons quelques pistes pour sa réalisation.

La suite du chapitre est constituée de quatre sections. La section 6.2 présente le processus de construction du pont d'interopérabilité ; le principe ainsi que les fondements théoriques

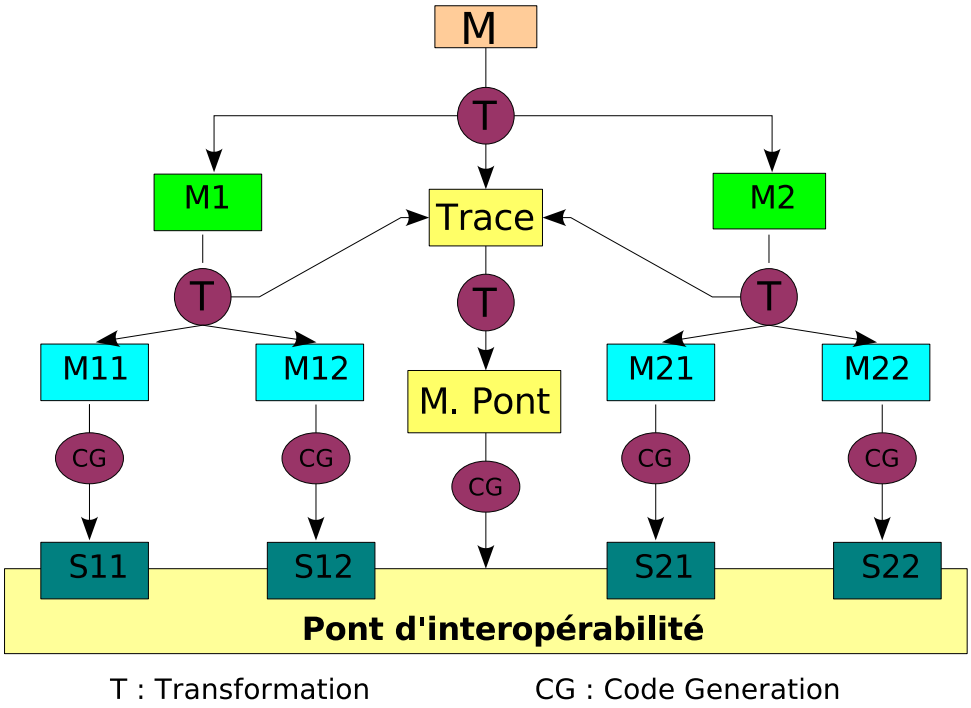

FIG. 6.2 – Aperçu de la construction de l'interopérabilité des langages dans Gaspard

de notre démarche y sont exposés. La section suivante (6.3) traite de l'automatisation du processus ; les différents métamodèles ainsi que les règles de génération des modèles correspondants sont présentés. La section 6.4 donne des indications sur l'implémentation de notre solution. Enfin, la section 6.5 conclut le chapitre par un regard critique sur la solution proposée.

6.2 Processus de construction

Dans le chapitre 3, nous avons présenté diverses solutions au problème d'interopérabilité. Cependant, nous avons vu que celles-ci n'étaient pas appropriées dans le contexte de notre travail. Dans cette section, nous présentons notre approche, plus adaptée au flot de conception Gaspard.

6.2.1 Principe de base

La première étape dans le processus de construction de notre pont d'interopérabilité consiste à déterminer les liens de communication entre les éléments du système à réaliser. Pour ce faire, il est nécessaire de trouver un moyen qui, à partir du modèle de départ, permet d'identifier ces liens. Le principe de base de notre proposition est fondé sur les points suivants :

1. Nous nous intéressons uniquement aux relations dont les éléments sont projetés dans des modèles différents.

2. Les relations possibles entre éléments se réduisent aux cas suivants : relation binaire bi-directionelle et relation binaire mono-directionnelle. Les relations n-aires peuvent toujours se réduire à l'un des cas précédents.
3. Les relations entre les éléments du modèle de départ (modèle de référence) conduisent à des relations à l'aide de mandataires (proxy) dans les modèles d'arrivée. Le mandataire matérialise l'autre extrémité de la relation, il donne accès aux services requis par l'élément. Ces services sont fournis par l'élément dont le mandataire est le représentant. La génération des mandataires est intrinsèque à la transformation. Si un élément utilise une référence, cette référence doit être matérialisée dans le modèle. En effet, les modèles étant indépendants, il n'est pas possible d'exprimer de relation inter-modèles.

Les différents cas de base qui conduisent à une relation de dépendance entre les éléments des modèles cibles sont illustrés par la figure 6.3. Les schémas (a) et (b) de la figure illustrent

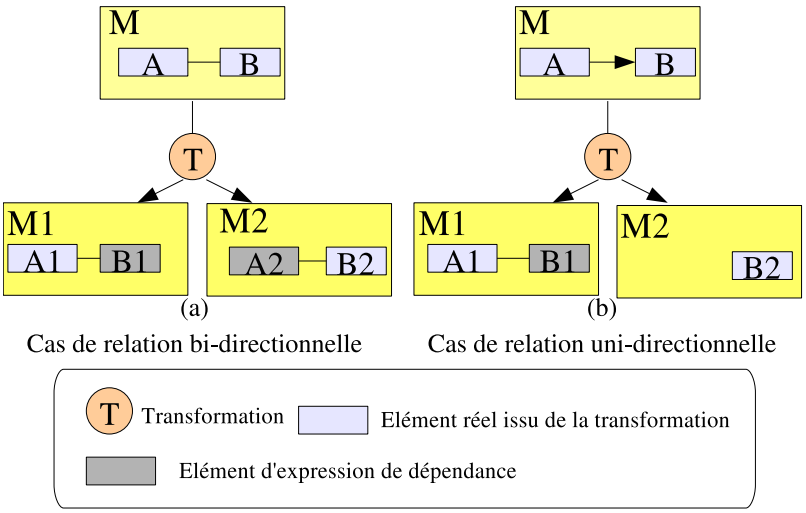

FIG. 6.3 – Cas de base des relations

le point trois et montrent comment, à partir des relations dans le modèle de référence, la transformation induit des mandataires.

À partir des points de ce principe de base, nous allons pouvoir construire un graphe de dépendances des composants qui contient les liens de communication nécessaires. Ce graphe de dépendances des composants contient toutes les informations utiles pour décrire les interfaces de communication et construire le pont d'interopérabilité.

La figure 6.4 présente des exemples de liens de dépendances entre des composants déployés dans Gaspard. Elle illustre également le problème d'interopérabilité des langages. Sur cette figure, le système modélisé est composé de plusieurs composants (*Cxx*) communicants, implantés dans des langages différents (*L1* et *L2*) et répartis sur deux processeurs (*P1* et *P2*). Les liens de dépendances entre les composants de ce système se répartissent en quatre types :

- les liens entre des composants implantés dans le même langage et affectés au même processeur (liens ILIP),
- les liens entre composants d'un même langage mais affectés à des processeurs différents (liens ILEP),

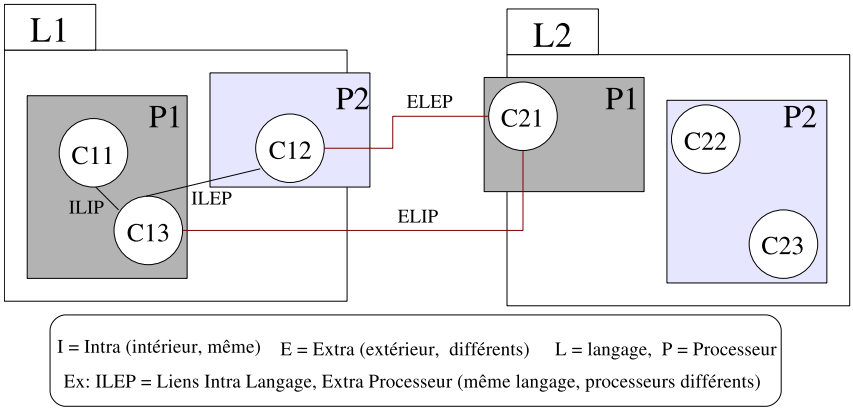

FIG. 6.4 – Types de liens entre composants déployés dans Gaspard

- les liens entre composants de langages différents, mais affectés au même processeur (liens ELIP),
- enfin, nous avons les liens entre composants de langages différents et affectés à différents processeurs (liens ELEP).

Les liens ILIP ne posent aucun problème de communication et ne sont, d'ailleurs, pas pris en compte dans le graphe de dépendances des composants généré à partir de la trace des transformations. Les liens ILEP traduisent des communications entre objets d'un même langage, mais affectés à des processeurs distincts. La gestion de ces communications n'est pas du ressort du pont d'interopérabilité ; elle relève de la responsabilité du mécanisme de communication standard du système. De ce fait, les liens ILEP ne seront pas abordés dans notre étude. Nous nous intéresserons exclusivement aux liens ELIP et ELEP.

Pour résoudre le problème des liens ELIP et ELEP, nous devons donc trouver un moyen permettant de les identifier. En principe, la distinction entre ces deux types de liens n'est pas indispensable pour leur traitement. Cependant, il peut être intéressant, pour des raisons de mise en œuvre, de faire cette distinction.

Dans la suite de cette section, nous décrivons le processus de construction du graphe de dépendances des composants et du pont d'interopérabilité.

6.2.2 Graphe de dépendances des composants

Un graphe de dépendances de composants (GDC) est un graphe où les nœuds du graphe correspondent aux composants et où les arcs représentent les liens de dépendance entre les composants. Ce graphe est inspiré des *graphes de dépendances de services (GDS)* [44, 34, 49, 22].

Graphes de dépendances de services

Les graphes de dépendances de services (GDS) sont des modèles couramment utilisés dans le génie logiciel aussi bien pour la gestion et la programmation de systèmes distribués que pour la configuration de systèmes d'exploitation ou la programmation de sous-systèmes de communication [54]. Dans [22, 49], les graphes de dépendances de services ont été utilisés

pour la construction de systèmes d'exploitation sur mesure pour une application et une architecture données.

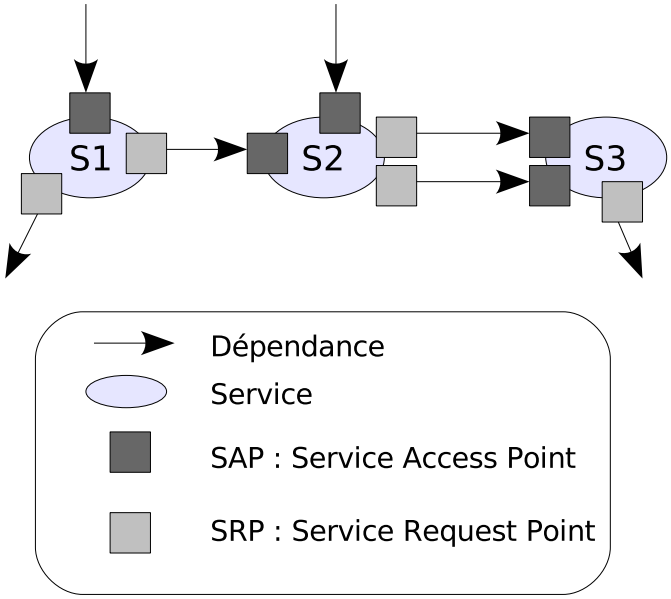

FIG. 6.5 – Exemple de graphe de dépendances de services

La figure 6.5 donne un exemple de graphe de dépendances des services. Les nœuds représentent les services. Chaque service possède des points d'accès (*SAP, pour Service Access Point*) et des points à partir desquels il peut requérir d'autres services (*SRP, pour Service Request Point*). Les arcs expriment les dépendances entre les services. Un arc orienté d'un service S1 vers un service S2 indique que le service S1 dépend du service S2.

Du GDS au graphe de dépendances des composants

L'idée, ici, est d'utiliser les concepts de graphes de dépendances des services pour modéliser les communications entre les composants de notre système. Ces communications sont représentées de façon abstraite à l'aide de relations de dépendances entre les composants. L'ensemble des relations forme un graphe éventuellement cyclique.

Notre modèle de graphe de dépendances des composants est une adaptation du modèle de graphe de dépendances des services. Cette adaptation est décrite comme suit :

- les services (nœuds) sont les composants,
- les SAP deviennent des CAP (pour Component Access Point). Chaque CAP possède les attributs suivants :
  - le *refereeName* contenant le nom du service ; dans notre cas, ce sera essentiellement un nom de fonction/méthode ou attribut,
  - le *refereeType* contenant le type de retour d'une fonction/méthode ou le type d'un attribut,

- les SRP deviennent des CRP (pour Component Request Point). Chaque CRP possède les attributs suivants :
  - le *proxyName* contenant le nom du service ; dans notre cas, ce sera essentiellement un nom de fonction/méthode ou attribut,
  - le *proxyType* contenant le type de retour d'une fonction/méthode ou le type d'un attribut,
- les arcs gardent la même sémantique que dans les graphes de dépendances de services.

La figure 6.6 présente le graphe de dépendances des composants correspondant aux relations de dépendances exprimées par la figure 6.3. Ces relations sont résumées dans le tableau 6.1.

Figure	Dépendances
(a)	A1 ⟶ B2 et B2 ⟶ A1
(b)	A1 ⟶ B2

TAB. 6.1 – Dépendances de la figure 6.3

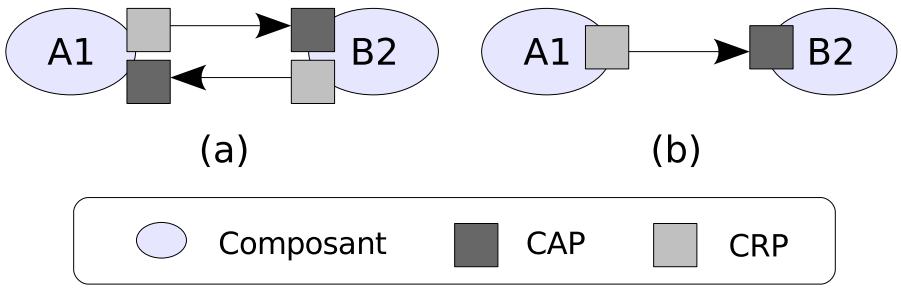

FIG. 6.6 – Graphe de dépendances des composants

Le modèle de graphe de dépendances des composants obtenu permet d'exprimer toutes les relations de dépendances entre les différents composants du système à construire. Nous obtenons ces relations de dépendances à partir d'un modèle de trace généré lors de la transformation de modèles. Notre approche de modélisation et génération de la trace sera présentée dans la section 6.3. Dans la sous-section qui suit, nous allons voir comment générer les interfaces de communication des composants à partir du graphe de dépendances des composants.

6.2.3 Génération des interfaces de communication

Les interfaces de communication permettent aux composants d'exposer (interfaces fournies) et de requérir (interfaces requises) des services à l'environnement. Chaque composant possède donc deux types d'interfaces : les interfaces requises qui lui permettent d'initier des communications, et les interfaces fournies permettant de recevoir des requêtes de la part des autres composants. La connaissance des interfaces est nécessaire pour permettre au pont d'interopérabilité de fournir et relayer les services attendus par les composants.

Dans le graphe de dépendances des composants, nous avons vu que chaque composant impliqué dans une relation de dépendance possède des CAP (Component Access Point) et des CRP (Component Request Point). La génération des interfaces de communication est basée sur les CAP et CRP du composant. Les CAP sont transformés en interfaces fournies et les CRP en interfaces requises suivant le principe ci-dessous :

Algorithm 1 Génération des interfaces à partir des dépendances

Notons :
- C_i les composants,
- FC_i l'ensemble des fonctions du composant C_i ,
- IRC_i l'ensemble des interfaces requises du composant C_i,
- IFC_i l'ensemble des interfaces fournies du composant C_i.

On obtient :
$\forall (C_i \longrightarrow C_j)$ /* C_i dépend de C_j */
  $\forall f \in FC_i$
    ajouter f dans IRC_i
    ajouter f dans IFC_j

Lorsqu'un composant A dépend d'un autre composant B, il y a un mapping direct entre chaque élément IRA (Interface Requise de A) et chaque élément de IFB (Interface Fournie de B), comme on peut le voir dans cet algorithme.

Dans le paragraphe suivant, nous illustrons la démarche par un exemple de génération d'interface IDL.

Illustration : génération d'interfaces IDL

L'IDL (Interface Definition Language) [58] est un langage purement descriptif défini par l'OMG qui permet de décrire les échanges et les interactions entre un client et un serveur en faisant abstraction des langages de programmation. Les interfaces sont utilisées par des objets clients pour appeler des services (appels de méthodes, par exemple). Leurs implémentations sont fournies par d'autres objets. Une définition d'interface écrite en IDL spécifie entièrement les paramètres de chaque opération et fournit toute l'information nécessaire au développement des clients de celle-ci. Des projections (mappings) entre les concepts IDL et les constructions des langages de programmation sont définies pour permettre l'utilisation des interfaces IDL dans le langage de programmation du client.

Un fichier IDL est structuré comme suit :
- 3 éléments principaux,
  - *module* : un espace de définitions,
  - *interface* : un regroupement de services,
  - *méthode* : un service.

<modules>	**module** banque {
<interfaces>	**interface** client {
<methodes>	**string** coordonnees() ;
	boolean anciennete(**in long** annee) ;
	} ;
	} ;

- des éléments secondaires :
  - types,
  - constantes,
  - exceptions,
  - attributs.

L'interface IDL d'un composant décrit l'ensemble des services (principalement l'ensemble des méthodes) du composant qui peuvent être appelés de l'extérieur. Ainsi, pour chaque composant impliqué dans une relation de dépendance, chaque IFC (Interface Fournie du Composant) est transformée en une interface IDL. La réalisation de l'interface est faite par le pont d'interopérabilité (voir 6.2.4). Quant à chaque IRC (Interface Requise du Composant), elle correspond à une IFC dans l'autre composant de la dépendance et est, de ce fait, également décrite en IDL par ce dernier et implémentée par le pont.

Le mapping des concepts du GDC vers les concepts IDL est présenté dans le tableau 6.2.

Concepts du GDC	**Concepts IDL correspondants**
Composant	module
Interface Fournie (IFC)	interface
Fonction dans IFC	méthode
Attribut du composant	Attribut IDL

TAB. 6.2 – Mapping de GDC vers IDL

Les figures 6.7, 6.8 et le code en encadré présentent un exemple de génération d'interfaces IDL pour deux composants (C1 et C2), où le GDC indique une dépendance de C1 vers C2 et vice versa.

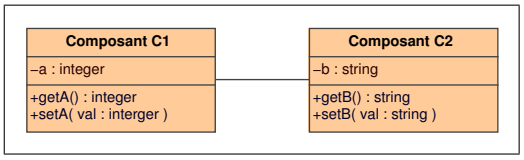

FIG. 6.7 – Modèle UML

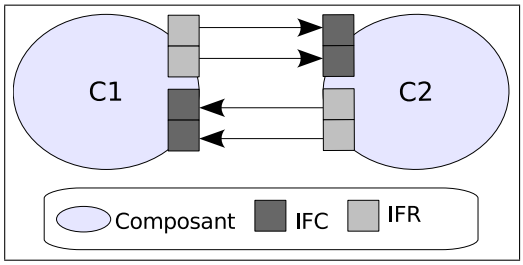

FIG. 6.8 – Graphe de dépendances

Interface IDL de C1	Interface IDL de C2
module C1 {	module C2 {
interface interfaceC1 {	interface interfaceC2 {
readonly attribut short a ;	readonly attribut string b ;
short getA() ;	short getB() ;
void setA(in short val) ;	void setB(in string val) ;
} ;	} ;
} ;	} ;

Synthèse

Nous avons présenté comment les interfaces de communication peuvent être générées automatiquement à partir du graphe de dépendances des composants. Notre démarche a été illustrée en présentant une génération automatique d'interfaces IDL. Cependant, les interfaces de communication ne constituent qu'un aspect de l'interopérabilité. Elles offrent une abstraction du mécanisme de communication des composants. Mais leur utilisation effective n'est possible que dans la mesure où il existe un tiers interlocuteur permettant de relayer les communications initiées sur les interfaces d'un émetteur vers les interfaces du récepteur. Ce tiers interlocuteur, c'est le pont d'interopérabilité.

Dans la sous-section qui suit, nous abordons la conception du pont d'interopérabilité. Il s'agira d'une présentation de la démarche de construction de ce pont ; sa modélisation et sa génération seront abordées à la section 6.3.

6.2.4 Pont d'interopérabilité

Le pont d'interopérabilité est un composant logiciel qui peut être comparé d'un point de vue fonctionnel à un bus ou un réseau d'interconnexion permettant de relier divers éléments. Ce bus logiciel interconnecte tous les composants impliqués dans le graphe de dépendances des composants comme l'illustre la figure 6.9.

La figure 6.10 permet d'illustrer la responsabilité qui incombe au pont d'interopérabilité. Sur cette figure, les applications, *Application 1* et *Application 2*, sont écrites dans deux langages différents : (a) présente les dépendances identifiées par les chiffres 1, 2 et 3 ; (b) correspond à une représentation virtuelle de ce que le pont devrait permettre de faire. Nous utilisons le concept de *proxy*[18] pour exprimer cette représentation virtuelle. Un proxy spécifie l'utilisation d'un élément extérieur à une application, comme si celui-ci existait en local.

Dans la situation de la figure 6.10, faire interopérer les applications, *Application 1* et *Application 2*, revient à mettre en place un mécanisme qui permet à :

- a, dans *Application 1*, d'accéder à x dans *Application2*,
- z, dans *Application 2*, d'accéder à c dans *Application1*,
- b, dans *Application 1*, d'accéder à y dans *Application2* et vice-versa.

Ce mécanisme revient à mettre en place, dans chacune des applications, des *proxies* qui permettent aux éléments de chaque application d'utiliser des fonctionnalités définies ailleurs.

[18]Le terme proxy (mandataire, en français) est utilisé ici pour exprimer une médiation, une abstraction du mécanisme d'accès à un élément. Pour en savoir plus sur les proxies, voir *http ://sebsauvage.net/comprendre/proxy/*

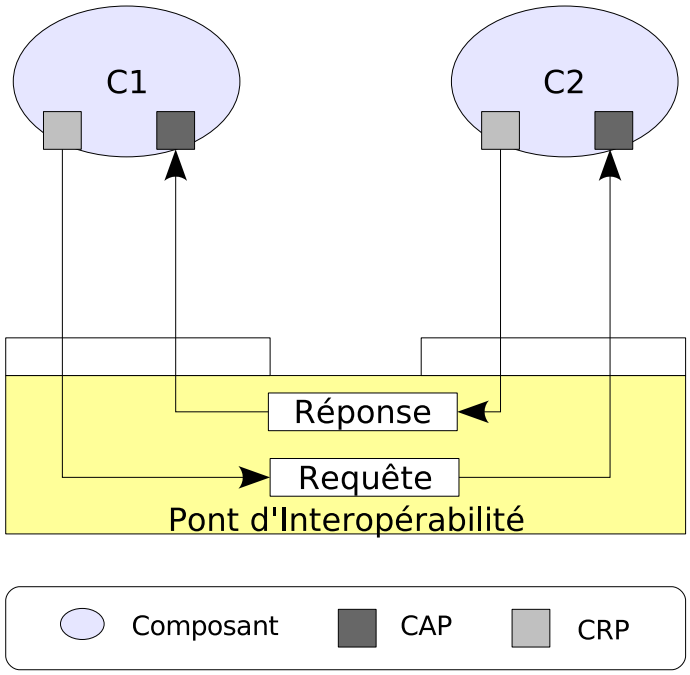

FIG. 6.9 – Principe de fonctionnement du pont

L'application de ce principe sur la figure 6.10 (a) donne la figure 6.10 (b). Les liens de dépendances uni-directionnelles se traduisent par la mise en place d'un proxy du côté de l'élément source de la dépendance. Quant aux liens de dépendances bi-directionnelles, elles conduisent à la création d'un proxy de chaque côté de la dépendance.

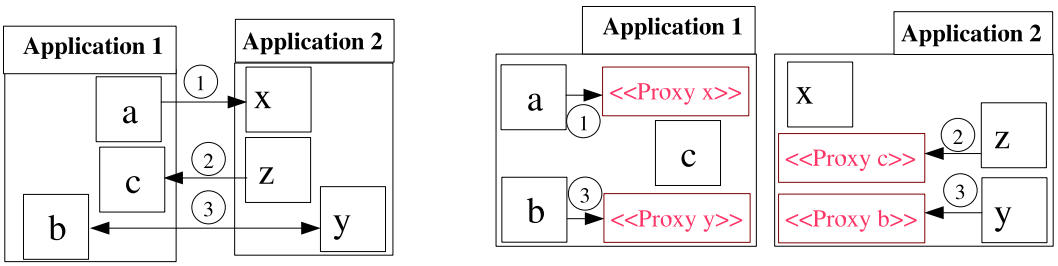

FIG. 6.10 – Traduction des liens de dépendances en proxies

6.3 Automatisation du processus

6.3.1 Métamodèle du pont d'interopérabilité

Identification des besoins

Pour envisager la génération automatique du pont d'interopérabilité par transformations de modèles, il faut définir un métamodèle. Une instance du métamodèle que nous allons définir doit contenir toutes les informations nécessaires à la génération du code de ce bus logiciel qui interconnectera les différents composants écrits dans différents langages. Nous avons déjà décrit le graphe de dépendances des composants et nous avons vu qu'il permettait de générer les interfaces de communication. Ce graphe, constitue une partie du modèle de pont. Afin d'achever la définition de notre métamodèle de pont, il convient de considérer les informations contenues dans ce graphe. En particulier, il faut vérifier que ces informations soient suffisantes pour générer ce pont.

Si l'on considère un ORB CORBA, ses services fondamentaux sont résumés dans le tableau 6.3 [39, 50] ; la seconde colonne contient les noms des services et la troisième décrit les services. Nous analysons dans le tableau 6.4 les pré-requis de la mise en œuvre du service à l'aide de notre graphe de dépendances des composants. De cette façon, nous pourrons savoir si les informations contenues dans le graphe de dépendances sont suffisantes.

N°	Services	Description
1	*Référencement*	définit des références véhiculant les informations nécessaires à la mise en contact des clients avec le serveur détenant des propriétés de l'objet référencé.
2	*Transport*	assure l'échange des données à travers un canal de transport comme une *socket* ou de la mémoire partagée.
3	*Liaison*	projections de la description des interfaces IDL vers les langages de programmation pour l'implantation des objets et la réalisation des applications utilisant ces objets.
4	*Représentation*	emballe et déballe les données de manière cohérente entre le client et le serveur.
5	*Protocole*	définit le format des requêtes.
6	*Aiguillage*	délivre les requêtes à l'implémentation de l'objet
7	*Activation*	mécanisme associant une implémentation à un objet, c'est-à-dire un espace mémoire pour stocker l'état de l'objet et un contexte d'exécution pour les opérations

Tab. 6.3 – Services fondamentaux d'un ORB CORBA

Au regard des données du tableau, il s'avère que notre graphe de dépendances des composants contient toutes les informations nécessaires à l'implémentation du pont. Pour satisfaire les exigences des services 4 et 6, il nous faut cependant assurer une correspondance entre chaque CRP et son CAP correspondant.

N°	Mise en œuvre
1	Les informations contenues dans notre graphe de dépendances sont suffisantes pour mettre en œuvre ce service.
2	Aucune information n'est requise des objets interconnectés.
3	La description des interfaces de nos composants contenue dans notre graphe + la technologie d'implémentation du pont sont suffisantes pour mettre en œuvre ce service.
4	Aucune information n'est requise des objets ; il dépend du choix de format de représentation des données échangées, mais sa réalisation peut nécessiter des conversions de types de données.
5	Relève seulement d'un choix technique.
6	Un mapping entre chaque CRP et le CAP correspondant est suffisant pour réaliser ce service.
7	Aucune information préalable n'est requise des objets.

TAB. 6.4 – Contraintes de réalisation des services

Métamodèle

Notre modèle de pont sera donc constitué du seul graphe de dépendances des composants décrit précédemment. Les principaux concepts du métamodèle se résument dans le tableau 6.5.

Concepts	Rôles
Bridge	concept racine, il contient les relations de dépendances entre les composants
dependency	décrit une relation de dépendance
DependentElement	désigne un élément impliqué dans une relation de dépendance
ComponentAccessPoint (CAP)	expose les fonctionnalités offertes par le composant aux autres composants qui dépendent de celui-ci.
ComponentRequestPoint (CRP)	représente les fonctionnalités requises par le composant et offertes par un ou plusieurs autres composant(s) dont il dépend.

TAB. 6.5 – Concepts du pont d'interopérabilité

La figure 6.11, page 80, correspond au métamodèle décrit à l'aide d'un diagramme de classe d'UML 2. Le pont représenté par le concept *Bridge* contient toutes les relations de dependances conceptualisées à l'aide du concept *Dependency*. Un *Dependency* contient deux références vers *DependentElement* : l'une correspond à la source de la dépendance (*src*) et l'autre, à la cible de la dépendance (*target*). Le *DependentElement* possède un attribut *refElement* qui contient l'identification de l'élément de modèle correspondant ; il contient également des collections de *ComponentAccessPoint* et *ComponentRequestPoint*.

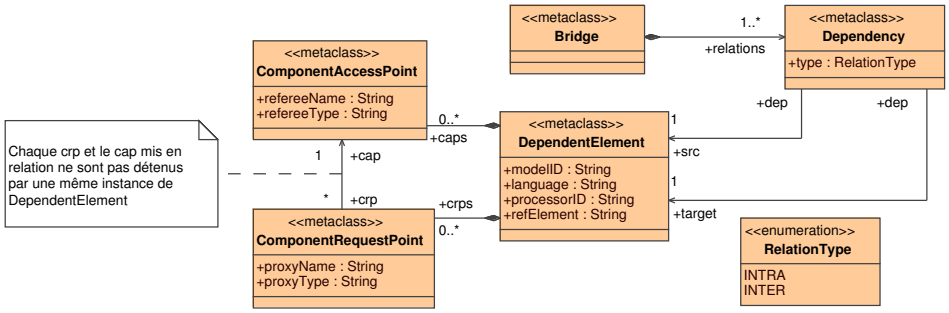

FIG. 6.11 – Métamodèle du pont d'interopérabilité

Une relation entre les *ComponentRequestPoint* et *ComponentAccessPoint* permet de gérer le mapping entre ces deux éléments. Cette relation sera exploitée pour la localisation des implémentations.

Enfin, le type énuméré *RelationType* permet de caractériser le type de dépendance. La valeur INTRA indique que les éléments de la dépendance sont situés sur le même processeur, tandis que la valeur INTER indique que les éléments de la dépendance sont situés sur des processeurs différents.

6.3.2 Métamodèle de la traçabilité

Identification des besoins

Rappelons que le but de notre modèle de trace est de permettre la génération du modèle de pont d'interopérabilité, c'est-à-dire qu'il doit permettre de déterminer les dépendances entre les éléments des modèles cibles des transformations.

Le système de traçabilité que nous construisons va fonctionner dans un flot de conception où sont réalisées une suite de transformations de modèles depuis un modèle de haut niveau d'abstraction jusqu'aux modèles de plates-formes. À chaque phase de la suite de transformations, une trace est générée. Il existe donc plusieurs niveaux de liens de trace. Si l'on considère la figure 6.12 où nous avons une suite de deux transformations (T1 et T2), la transformation T1 produit des liens de trace de niveau 1 et T2 produit des liens de trace de niveau 2. Pour identifier les liens de trace entre les éléments du modèle de départ et les éléments des modèles issus de la dernière transformation (T2), nous ajoutons un niveau supplémentaire ici : 12. Le principe de numérotation est le suivant : pour une suite de n transformations, on distingue les niveaux $1, 2, ..., n$ et le niveau supplémentaire sera numéroté $1n$.

Au minimum, un modèle de trace enregistre des liens entre les éléments de modèles sources et leurs correspondants dans les modèles cibles. Cette trace minimale n'est pas suffisante dans notre cas, car elle ne permet pas de construire notre graphe de dépendances des composants.

Pour comprendre notre démarche de conception de la traçabilité, nous proposons de considérer séparément la gestion des liens entre éléments sources et cibles et le cas des relations de dépendances dans le modèle de référence. Dans chaque cas, nous déduirons les

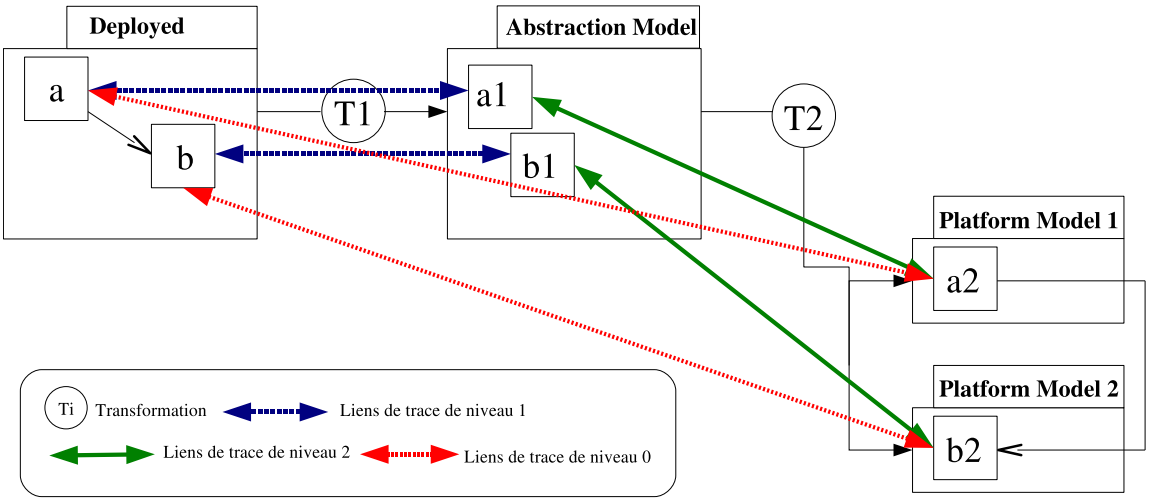

FIG. 6.12 – Relations entre éléments du modèle de référence

concepts nécessaires, et le métamodèle final de la traçabilité sera constitué de l'union des deux ensembles de concepts.

Liens de trace entre éléments de modèles sources et cibles

La figure 6.12, présente un exemple de suite de transformations à partir du modèle de référence (*Deployed*) vers deux modèles de plates-formes (*Platform Model 1* et *Platform Model 1*). A chaque transformation, des liens de trace sont établis entre les éléments sources et cibles de la transformation. Les liens issus de la transformation du modèle de référence au modèle d'abstraction sont des liens de niveau 1. De même, les liens de trace de la transformation du modèle d'abstraction au modèle de plate-forme sont des liens de niveau 2. A partir de ces liens de niveaux 1 et 2, on peut déduire les liens de niveau 12 qui sont les liens entre les éléments du modèle de référence et les éléments des modèles de plates-formes. Ce sont ces derniers qui seront nécessaires à la construction du pont d'interopérabilité.

Nous définissons le concept de *ElementTrace* pour représenter les liens de trace entre éléments source et destination d'une transformation. Un *ElementTrace* possède un attribut *level* de type entier qui indique le niveau (1, 2, ...) du lien, et un couple d'éléments source et cible (*ModelElement*) correspondant aux éléments mis en relation par le lien.

Relations de dépendances dans le modèle de référence

En considérant toujours la figure 6.12, dans le modèle de référence, l'élément *a* dépend de l'élément *b*. Les éléments *a* et *b* sont transformés en *a2* et *b2* respectivement, dans les modèles de plates-formes *Platform Model 1* et *Platform Model 2*. Etant donné que *a2* et *b2* sont dans deux modèles cibles distincts, nous en déduisons une relation de dépendance de *a2* vers *b2*. Si la relation était bi-directionnelle, c'est-à-dire *a* dépendant de *b* et *b* dépendant de *a*, nous aurions eu deux relations de dépendances (*a2* vers *b2* et *b2* vers *a2*). Pour représenter ces liens de dépendances issus de relations de dépendances dans le modèle de référence, nous introduisons le concept de *RelationTrace*. Par ailleurs, la traduction de ces dépendances dans

le modèle de référence en dépendances dans les modèles de plates-formes est représentée par le concept de *ElementDependency*.

Dans ce traitement des relations entre éléments du modèle de référence, nous n'avons considéré que les relations binaires. En effet, nous considérons que les relations n-aires peuvent toujours se décomposer en plusieurs relations binaires.

D'après les deux paragraphes précédents, notre système de trace permet :
- d'enregistrer les liens entre éléments sources et éléments cibles des transformations grâce au concept de *ElementTrace*,
- de déterminer et mémoriser les relations entre les éléments du modèle de référence à l'aide du concept de *RelationTrace*. Ce concept est un concept intermédiaire que nous utilisons dans la génération de la trace. Une fois le processus terminé, ces instances peuvent être détruites,
- d'enregistrer les dépendances entre éléments des modèles cibles au moyen du concept de *Elementdependency*.

En définitive, une information de trace est :
- soit un lien de trace entre un élément dans un modèle source et un autre élément dans un modèle cible,
- soit une relation de dépendance dans le modèle de référence,
- ou une relation de dépendance entre éléments des modèles de plates-formes.

Nous utilisons le concept *TraceRecord* pour généraliser ces trois concepts.

Nous résumons les concepts de notre métamodèle de traçabilité dans le tableau 6.6.

Concepts	**Rôles**
ModelElement	représente un élément de modèle.
ElementTrace	mémorise les liens entre éléments sources et éléments cibles de la transformation.
RelationTrace	mémorise les relations de dépendances entre éléments du modèle de référence.
ElementDependency	contient les relations de dépendances entre éléments des modèles cibles.
TraceRecord	généralise *ElementTrace*, *RelationTrace* et *Elementdependency*.

TAB. 6.6 – Concepts du métamodèle de trace

Métamodèle

Notre métamodèle final correspond à la figure 6.13, page 83, exprimée à l'aide de diagramme de classe UML.

Le concept *TraceModel* est le concept racine de notre métamodèle ; il contient l'ensemble de toutes les informations de trace. Une information de trace est représentée à l'aide du concept *TraceRecord* qui se spécialise en *ElementTrace*, *RelationTrace* et *ElementDependency*. Chacun de ces concepts possède deux références (*srce* et *dest*) vers un *ModelElement* permettant de faire

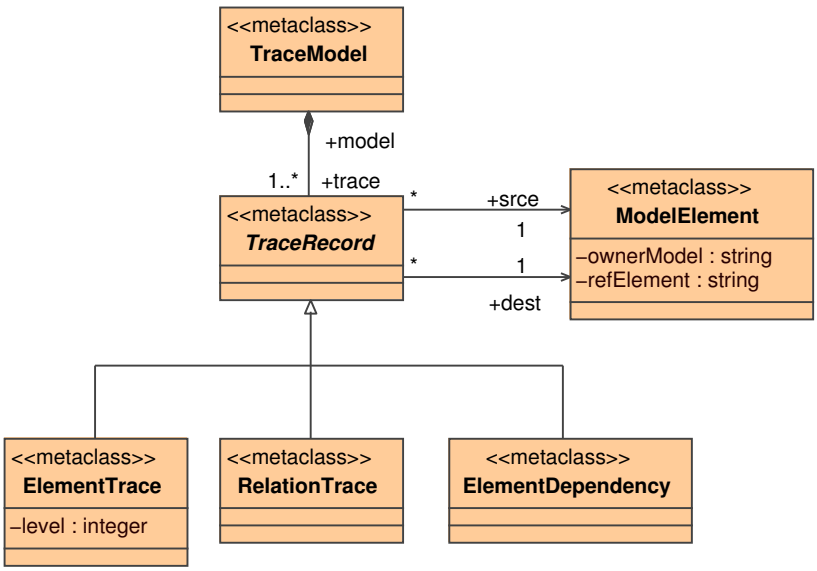

FIG. 6.13 – Métamodèle de trace

le lien avec l'élément de modèle correspondant. Cette liaison se fait par le biais de l'attribut *refElement*.

6.3.2.1 Généralisation du mécanisme de trace

Le mécanisme de trace que nous proposons peut être utilisé dans un flot de conception comportant plusieurs successions de transformations de modèles comme l'illustre la figure 6.14, page 84. Dans ce schéma :

- le modèle *MR* est le modèle de référence, dans Gaspard (figure 6.1, page 68) il correspond au modèle déployé (*Deployed*).
- les T_i sont des transformations de modèles. T_1 et T_2 génèrent des modèles de niveaux d'abstraction (MA_i) tels que *TLM* et *RTL*. Quant à T_3 et T_4, elles génèrent des modèles de plates-formes(MP_i) comme HPF, SystemC, VHDL. Chacune de ces transformations produit également en sortie un modèle de trace. Les différents modèles de trace ($Trace_i$) isssus des transformations T_i sont ensuite fusionnés(*merge*) pour obtenir le modèle de trace final(*Trace*).
- *transf* est une transformation qui prend en entrée le modèle de trace et produit en sortie le modèle du pont d'interopérabilité (*M.Pont*) pour les modèles MP_i.
- *CodeGen* sont des générations de codes sources correspondant aux plates-formes de simulation/exécution.

Maintenant que nous disposons du métamodèle de traçabilité, nous allons dans la sous-section suivante voir comment générer les instances de ce métamodèle, c'est-à-dire les modèles de trace.

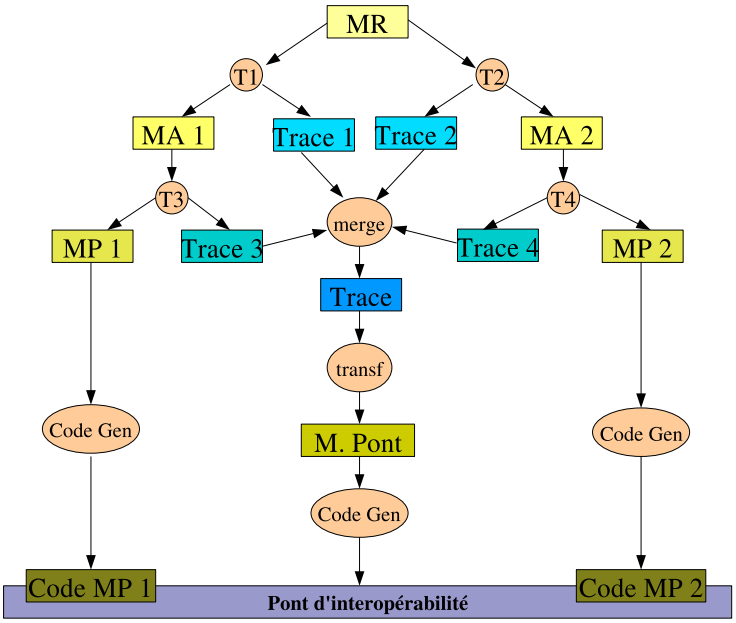

FIG. 6.14 – Approche d'interopérabilité basée sur la trace

6.3.3 Génération du modèle de trace

La génération du modèle de trace se réalise partiellement pendant les transformations de modèles. A l'issue des transformations, un modèle partiel de trace est créé. Une phase post-transformations permet d'analyser ce modèle partiel et de le compléter. Comme nous l'avons précédemment signalé, deux types de transformations dans *Gaspard* font l'objet de notre étude : les transformations du modèle de référence (*Deployed*) vers les modèles de niveaux d'abstractions (*TLM* et *RTL*), et les transformations des niveaux d'abstractions vers les modèles de plates-formes. Pour chacune de ces transformations, nous allons montrer les opérations CRUD(Create, Read, Update, Delete) effectuées sur le modèle de trace.

Transformations vers les modèles de niveaux d'abstractions

A l'initialisation du processus de transformation, une instance vide du modèle de trace est créée. Pendant la transformation, lorsqu'un élément (*elt*) du modèle de référence est soumis au moteur de transformation, les actions suivantes sont réalisées :

- on interroge le modèle pour trouver tous les liens d'association pour lesquels *elt* est source. Chaque couple d'éléments (source et cible de l'association ; soit, par exemple, *elt* et *elt2*) permet de créer une instance *relTrace* de *RelationTrace* du modèle de trace avec :
  - relTrace.srce = elt,
  - relTrace.dest = elt2.
- pour chaque élément *a* du modèle de référence ayant servi à la création d'un élément *b* dans le modèle de niveau d'abstraction, on crée une instance *eltTrace* de *ElementTrace* avec :
  - eltTrace.level = 1,

▹ ▹ eltTrace.srce = a,
▹ ▹ eltTrace.dest = b.

A la fin de cette première phase de transformation, le modèle de trace partiel créé contient les liens de trace de niveau 1 et une mémoire de tous les liens d'associations entre les éléments du modèle de référence. Ce modèle de trace partiel deviendra un modèle d'entrée dans les transformations suivantes.

Transformations vers les modèles de plates-formes

Dans cette phase de transformations des modèles de niveaux d'abstractions vers les modèles de plates-formes, les seules opérations effectuées sur le modèle de trace consistent à créer les liens de trace de niveau 2. Le processus de création de ces liens est identique à celui de la création des liens de niveau 1.

Au terme de cette phase, le modèle de trace est toujours partiel, mais contient toutes les informations nécessaires pour sa finalisation.

Phase post-transformations

Une fois les séries de transformations terminées, la phase post-transformations peut commencer. Cette phase se déroule en deux étapes : la création des liens de trace de niveau 12, la traduction des liens d'associations dans le modèle de référence en relations de dépendances entre les éléments des modèles de plates-formes.

Création des liens de trace de niveau 12

La création des liens de trace de niveau 12 se fait à partir des instances de liens de niveaux supérieurs(1 et 2) selon l'algorithme suivant :

Création des liens de trace de niveau 12

```
Pour tout couple(x,y) formant une instance de ElementTrace de niveau 1
   Pour tout couple(y,z) formant une instance de ElementTrace de niveau 2
       créer une instance eltTrace de ElementTrace avec :
       eltTrace.level = 12
       eltTrace.srce = x
       eltTrace.dest = z
   FinPour
FinPour
```

Traitement des liens d'associations du modèle de référence

Il s'agit, ici, de traduire les liens d'associations dans le modèle de référence par des relations de dépendances dans les modèles de plates-formes.

Soit la fonction *Cible* qui prend en paramètre un élément du modèle de référence et définie comme suit :

```
  Cible(x) = ensemble des éléments des modèles de plates-formes
pour lesquels l'élément x a servi dans leur création.
```

Les valeurs de cette fonction sont données par les liens de trace de niveau 0 dont la création a été traitée dans le paragraphe précédent.

La procédure de traitement est décrite comme suit :

Traitement des liens d'associations du modèle de référence

```
Pour toute relation(a,b) formant une instance de RelationTrace
  déterminer l'ensemble Cible(a)
  déterminer l'ensemble Cible(b)
  Pour tout couple(x,y) ∈ Cible(a) X Cible(b)
   Si x.ownerModel ≠ y.ownerModel alors
     créer une instance eltDep de Elementdependency avec :
     eltDep.srce = x
     eltDep.dest = y
   FinSi
  FinPour
FinPour
```

Après la réalisation des transformations et les phases post-transformations, le modèle de trace complet est généré. Dans ce modèle, les informations contenues dans les instances de *Elementdependency* sont nécessaires et suffisantes pour générer notre modèle de pont d'interopérabilité. Il est évident que dans ce modèle de trace, en particulier dans les *Elementdependency*, il peut y avoir des relations redondantes. Ces redondances peuvent être facilement détectées et supprimées pour épurer le modèle. Cette opération ne présente pas de difficulté particulière. Aussi, nous nous abstenons de la décrire.

6.3.4 Du modèle de trace au modèle de pont d'interopérabilité

La génération du modèle de pont d'interopérabilité est une pure opération de transformation de modèles. Cette transformation prend en entrée le modèle de trace et produit en sortie le modèle de pont d'interopérabilité. La correspondance entre les concepts du métamodèle de trace et ceux du pont d'interopérabilité est donnée dans le tableau 6.7.

Concepts de la trace	Concepts du pont
TraceModel	Bridge
ElementDependency	Dependency
ModelElement	DependentElement

Tab. 6.7 – Correspondance entre concepts des métamodèles de trace et concepts du pont

Traitement du concept TraceModel

Le *TraceModel* est le concept racine du modèle de trace. Lorsque ce concept sera soumis au moteur, une instance vide de *Bridge* sera créée.

Traitement du concept Elementdependency

Chaque instance *edTrace* du concept Elementdependency du modèle de trace est traduit en une instance *edPont* du concept Dependency du modèle de Pont, tel que :

```
  edPont.src = edTrace.srce
edPont.target = edTrace.dest
edPont.type = 'INTRA' si edTrace.srce.processor = edTrace.dest.processor
edPont.type = 'INTER' sinon
```

Traitement du concept ModelElement

Comme l'indique le tableau de correspondance (tableau 6.7), chaque instance du concept ModelElement (*edTrace*) du modèle de trace se transforme en une instance du concept DependentElement (*edPont*) dans le modèle de pont avec :

```
  edPont.modelId = edTrace.ownerModel
edPont.language = edTrace.language
edPont.processor = edTrace.processor
```

Les attributs *ComponentAccessPoint* (CAP) et *ComponentRequestPoint* (CRP) qui sont des collections (relation de compostion) sont calculés comme suit :

Algorithm 2 Calcul des attributs CAP et CRP de edPont

Notons :
- C_f les composants qui dépendent de edPont,
- C_c les composants dont dépend edPont,
- $FC(A)$ l'ensemble des fonctions du composant A ,

On obtient :
$\forall A \in C_f$
  $\forall f \in FC(A)$
    ajouter f dans *edPont.CAP*
et :
$\forall A \in C_c$
  $\forall f \in FC(A)$
    ajouter f dans *edPont.CRP*

6.4 Implémentation du pont d'interopérabilité

Nous avons proposé une démarche de construction d'un modèle de pont d'interopérabilité. L'implémentation de ce pont peut être réalisée à l'aide d'une technologie de pont comme CORBA ou autre ORB, ou faite de façon ad hoc. Dans cette section, nous ne proposons pas une implémentation concrète, mais nous donnons seulement quelques indications sur la façon dont elle pourrait être réalisée.

6.4.1 Implémentation à l'aide d'un ORB CORBA

À ses débuts, CORBA a été souvent critiqué pour ses performances. Mais aujourd'hui, il existe des implémentations légères qui offrent de bonnes performances. Par exemple, *OpenFusion e*ORB SDR C Edition, OpenFusion RTOrb Java(tm) Edition* [19] sont des ORB conçus pour répondre aux besoins d'applications embarquées et temps réel.

Nous pensons que l'implémentation de notre pont d'interopérabilité peut être réalisée à l'aide d'un ORB CORBA. Le modèle de pont généré automatiquement par transformation de modèles pourra alors permettre de :

- générer les interfaces IDL ;
- réaliser les services de référencement, d'aiguillage et de liaison - en exploitant les relations entre les différents éléments, ainsi que le mapping entre les différentes fonctionnalités de ces éléments.

6.4.2 Implémentation ad hoc

Une implémantation ad hoc de notre proposition peut être aussi envisagée. Nous abordons ci-dessous quelques éléments à prendre en compte dans le cadre d'une implémentation ad hoc.

Des relations entre éléments de modèles aux relations entre instances

Notre démarche est basée sur la détermination de relations de dépendances entre les éléments des modèles. Les interfaces de communication générées sont également définies pour chaque élément de modèle. Ces interfaces représentent les proxy. Une application doit alors rechercher l'implémentation du proxy pour pouvoir l'utiliser.

La recherche de l'implémentation d'un proxy peut se faire auprès du pont qui fournit alors un service bien connu, en utilisant un identificateur unique caractérisant le proxy. Cet identificateur peut être construit par la concaténation des noms des différents modèles, allant du modèle de référence au modèle technologique, et la concaténation du nom complet de l'instance (nom de paquetage + nom de l'instance) dans le modèle de référence. Le pont peut lui aussi construire ce nom et fournir un service de mapping entre le nom du proxy et l'objet implémentant le proxy.

La figure 6.15 (version plus complète de la figure 6.2, page 69) illustre cette solution. L'identificateur de B1 sera "M.M1.S1.B1". L'application S1 demandera (lookup) au pont d'interopérabilité l'implémentation du proxy. Le pont lui donnera cette implémentation qui permettra de communiquer avec B2. Le pont détient un registre indiquant que "M.M1.S1.B1" est en relation avec "M.M2.S2.B2".

Sur la figure, on remarque que les transformations *CodeGen* peuvent prendre en entrée le modèle de pont. Dans la génération de code, cela permettrait de prendre en compte les interactions avec le pont, comme par exemple l'enregistrement des instances dans un registre ou la demande auprès du pont de l'initialisation des instances d'un proxy.

[19] http ://www.prismtechnologies.com

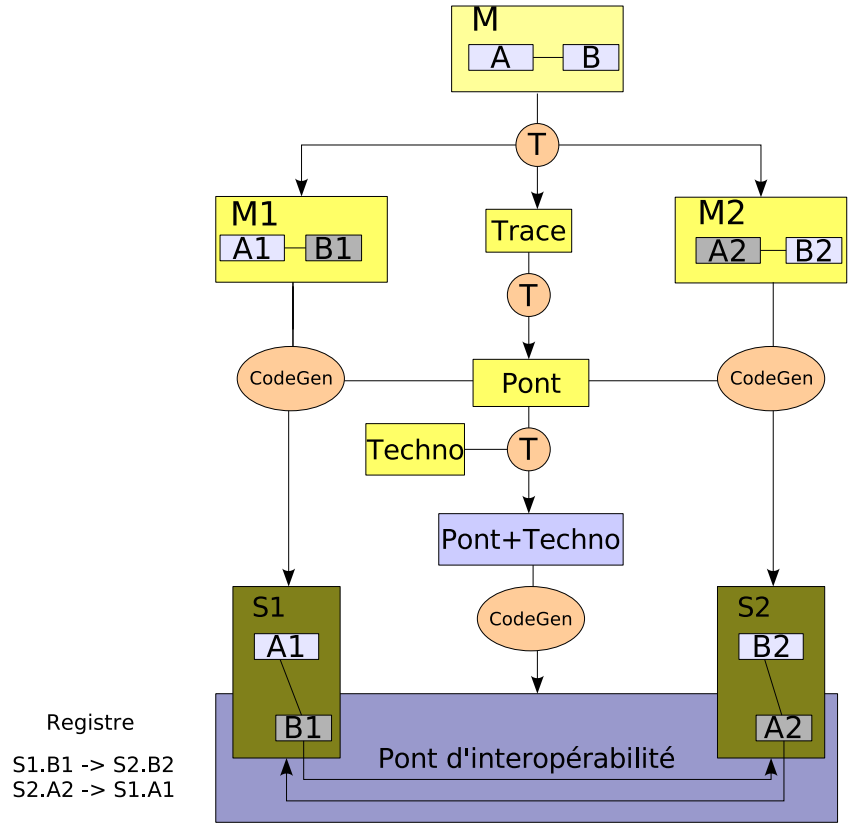

FIG. 6.15 – Processus de construction de l'interopérabilité des langages dans Gaspard

Interfaces de communication et gestion des requêtes

Pour mettre en place le mécanisme de communication inter-systèmes, il est d'abord indispensable dans le cas d'une implémentation ad hoc de définir un langage de description des interfaces de communication. Un langage de description basé sur XML nous semble être plus simple et plus rapide à mettre en œuvre. Dans un deuxième temps, il faudra réaliser le gestionnaire de requêtes qui se chargera de traduire les requêtes. Pour cela, il faut également définir un protocole d'échange des données (au sens large).

Conversions de types de données

Nous avons vu que chaque élément de CAP (Component Access Pont) correspondait à un élément de CRP (Component Request Point). Cette correspondance servira à implémenter les conversions de types nécessaires. Pour chaque correspondance ou le *refereeType* est différent du *proxyType*, une conversion est nécessaire.

6.5 Conclusion

Dans ce chapitre, nous avons proposé une démarche compatible avec la méthodologie Gaspard pour répondre au problème d'interopérabilité des langages. Nous avons proposé un pont d'interopérabilité dont le procesus de construction ainsi que son automatisation, à l'aide de métamodèles et de transformations de modèles, ont été exposés.

Le métamodèle de traçabilité défini, bien qu'il ait été conçu pour répondre à notre besoin, reste un métamodèle de trace général qui peut être utilisé à d'autres fins. Par exemple, il peut être exploité dans le cadre de l'ingénierie des besoins. Dans ce domaine, la trace est souvent utilisée pour établir des relations entre les besoins des utilisateurs et les artefacts d'implémentation. Grâce à ces relations, il est alors possible de faire une analyse des implications d'un changement dans les besoins sur les modèles de conception et d'implémentation.

La génération du modèle de trace ne pose pas de problème particulier. Sa mise en oeuvre dans les outils actuels peut se faire par exemple en intervenant sur le moteur lui-même quand les sources sont disponibles, ou aussi en enrichissant les règles de transformation pour générer un modèle supplémentaire correspondant à la trace.

Nous n'avons pas fourni d'implémentation du pont d'interopérabilité, mais nous avons donné des indications sur la façon dont elle pourraît être réalisée. Ces indications ne constituent pas des contraintes d'implémentation. Au contraire, la mise en œuvre reste ouverte ; toute technologie d'implémentation d'intergiciel peut être utilisée.

Chapitre 7

Interopérabilité pour la co-simulation multi-niveaux dans Gaspard

7.1 Introduction

Dans la description de Gaspard faite au chapitre 4, section 4.4.2, nous avons montré la nécessité de simuler à plusieurs niveaux d'abstraction. Cette simulation multi-niveaux se justifie, d'une part, par la réutilisation d'IP hétérogènes et, d'autre part, par le gain en temps de simulation que procurent les simulations de haut niveau d'abstraction tel que TLM. Cette prise en charge de la simulation à plusieurs niveaux d'abstractions nécessite une adaptation des interfaces de communication des composants. En effet, ces interfaces varient suivant les niveaux d'abstractions.

Pour réaliser l'adaptation de composants ayant des interfaces incompatibles, une possibilité est de modifier les interfaces des composants devant être connectés afin de les rendre compatibles. Mais cette solution n'est pas satisfaisante puisque :

- elle nécessite de modifier les composants et, pour cela, une bonne connaissance de leur fonctionnement et de leur implémentation est indispensable. Le concepteur perd l'intérêt d'un composant livré « clefs en main »,
- il est difficile de garantir que les modifications réalisées n'auront pas modifié la fonctionnalité du composant alors que le composant initial aura peut-être déjà été validé,
- elle augmente les durées de conception.

Ainsi, de nombreuses recherches actuelles se concentrent plutôt sur la mise au point de composants traducteurs de protocole (wrappers). Pour connecter deux composants ayant des interfaces différentes mais compatibles (au sens où le langage d'une interface peut être traduit dans le langage de l'autre interface), le concepteur n'a plus qu'à intercaler un wrapper entre les deux interfaces.

Cependant, la conception des composants wrappers est délicate car il faut prendre en compte plusieurs facteurs dont :

- l'adaptation des protocoles avec leurs éventuels paramètres,
- l'adaptation des interfaces de communication,
- les types des données transférées peuvent nécessiter des opérations de conversion,
- l'adaptation des horloges.

Le facteur le plus critique dans cette tâche est l'adaptation des protocloles de communication ; la plupart du temps, seuls des fragments des protocoles sont effectivement traduits.

Dans ce chapitre, nous proposons dans un premier temps, de présenter quelques travaux relatifs à la conception des wrappers. Nous nous intéresserons notamment aux approches qui proposent une génération automatique. Nous présenterons ensuite notre proposition de solution pour la conception de wrappers dans Gaspard. Cette présentation sera suivie d'une illustration à l'aide d'un exemple.

7.2 Approches de génération de wrapper pour la co-simulation multi-niveaux

Compte tenu de la délicatesse de la conception des wrappers et du fait qu'elle est renouvelée pour chaque application, une automatisation de cette tâche est intéressante et permettrait une meilleure exploration d'architectures et un gain de temps.

De nombreux travaux ont été réalisés dans ce sens parmi lesquels nous pouvons distinguer deux approches pour la génération automatique des interfaces de communication : l'approche par synthèse et l'approche par assemblage.

L'approche par synthèse [101, 102] utilise une description formelle de l'interface à partir de laquelle est extraite une machine d'états finis. Ces méthodes offrent un haut niveau d'abstraction pour la conception des interfaces mais leur conception reste manuelle et restreint les interfaces générées. Cette approche est donc mal adaptée à la conception d'interfaces complexes [56]. De plus, dans le cadre de nos travaux, nous envisageons une génération automatique de nos interfaces.

L'approche par assemblage [56, 55, 64, 122] part d'une spécification du système et en extrait un certain nombre de paramètres tels que les protocoles. Des composantes de bibliothèques sont alors sélectionnées, configurées et assemblées pour générer l'interface de communication. Dans cette approche, le système est spécifié par une architecture virtuelle paramétrable. Les principaux paramètres à spécifier sont :

- les processeurs (type, nombre, configuration),
- les composants utilisés,
- la communication (type, protocole, rôle).

Le SoC est alors vu comme un assemblage de plusieurs instances de ces différents éléments architecturaux. Les interfaces correspondent alors à des ensembles de services fournis/requis à travers le réseau de communication.

Il existe donc des travaux pour la génération d'interfaces de communication (wrappers) pour la co-simulation à plusieurs niveaux d'abstraction. Seulement, ces travaux ne sont pas effectués dans le contexte d'un flot de développement basé sur les approches de l'IDM ; de ce fait, leur intégration dans Gaspard peut s'avérer difficile.

7.3 Conception de wrappers dans Gaspard

Notre objectif dans l'étude de la conception des wrappers dans le cadre de ces travaux est de pouvoir les modéliser à l'aide d'un métamodèle et, par opérations de transformations de modèles, les générer automatiquement.

La plus grande difficulté dans cette démarche est la modélisation des protocoles de communication. Il est très difficile de trouver un formalisme permettant de décrire n'importe quel protocole de communication. En général, on décrit un protocole à l'aide de diagrammes de séquences ou de diagrammes d'états d'UML 2.0. Mais on ne peut pas garantir que tout protocole puisse être décrit à l'aide de ces outils. C'est pourquoi, dans la plupart des solutions existantes [49, 54], l'adapation des protocoles est réalisée à l'aide d'IP matérielles sélectionnées dans une bibliothèque.

Si nous ne pouvons pas proposer un métamodèle pour les protocoles, alors nous ne sommes pas en mesure de proposer une génération complète et automatique de wrappers. Nous proposons donc de résoudre les problèmes au cas par cas ; c'est-à-dire que nous proposons un wrapper pour deux niveaux d'abstraction, deux protocoles et deux interfaces de communication donnés. Mais une fois tous ces paramètres fixés, la génération est alors possible.

Dans ce contexte, nous pensons qu'il est préférable de constituer une bibliothèque de wrappers pour les combinaisons d'interfaces, protocoles et niveaux d'abstractions les plus utilisés dans Gaspard. Les wrappers composant cette bibliothèque seront spécifiés et générés. Dans cette démarche, la traçabibilité dans les transformations servira uniquement à déterminer les composants à adapter et de sélectionner le wrapper approprié dans la bibliothèque. En effet, dans les modèles déployés de Gaspard, la description de chaque composant matériel contient : les niveau et langage de simulation, l'interface de communication. Le mécanisme de trace proposé au chapitre 6, grâce au graphe de dépendances des composants, permettra de détecter les composants en relation qui présentent des interfaces différentes. La figure 7.1 schématise notre démarche. La flèche entre les composants *C1* et *C2* (au niveau des modèles *M1* et *M2*) matérialise un lien de dépendance contenu dans la trace. Une fois ce lien de dépendance détecté, on peut alors sélectionner dans la bibliothèque le wrapper approprié pour interconnecter *C1* et *C2* (au niveau des systèmes *S1* et *S2*).

Nous allons maintenant nous intéresser plus particulièrement au wrapper entre TLM CABA et TLM PVT.

7.4 Illustration : Co-simulation TLM CABA et TLM PVT

Dans cette section, nous proposons une étude de la conception de wrappers pour les niveaux de simulation CABA et PVT à travers un exemple. L'exemple porte sur la co-simulation *SystemC* aux niveaux CABA et PVT d'une architecture matérielle *QuadriPro* [17]. Cette architecture est composée de quatre unités de calculs, de deux mémoires (une mémoire d'instructions et une mémoire de données), et d'un réseau d'interconnexion (un cross Bar) reliant les processeurs aux mémoires, comme le montre la figure 7.2. Chaque unité de calculs est constituée d'un processeur « MIPS » doté d'une mémoire cache.

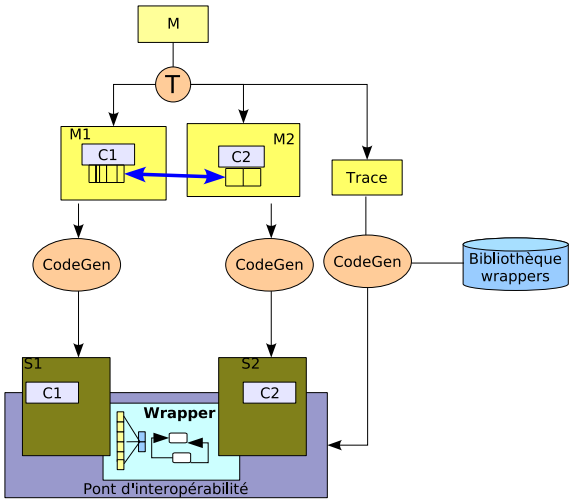

FIG. 7.1 – Détection et selection de wrappers

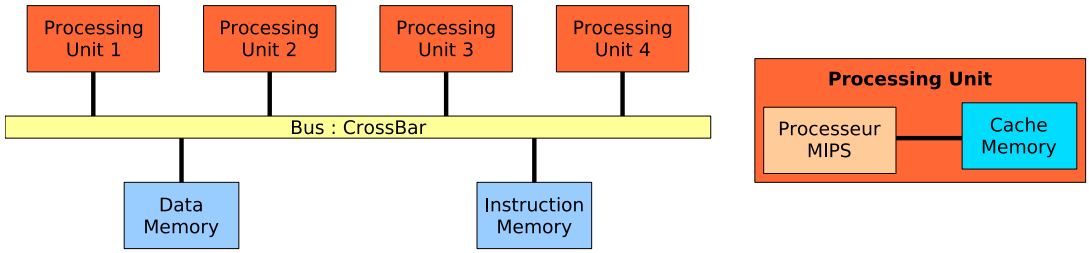

FIG. 7.2 – Modèle de l'architecture QuadriPro

Le but de cet exemple est de mettre en œuvre une simulation à deux niveaux d'abstraction. Nous avons choisi de considérer les quatre processeurs, le réseau d'interconnexion ainsi que la mémoire d'instructions au niveau CABA, et la mémoire de données au niveau PVT. Ce choix est tout à fait arbitraire, mais en considérant le nombre d'accès nécessaires à la mémoire de données partagée, on peut aisément justifier le fait de choisir le niveau PVT (plus rapide pour la simulation) pour la mémoire de données.

La mise en œuvre d'un tel exemple dans l'approche Gaspard commence par une modélisation abstraite de l'architecture. Le modèle abstrait obtenu à cette première étape sera ensuite projeté sur deux modèles ; un au niveau CABA et l'autre au niveau PVT. Cette seconde étape est réalisée par transformations de modèles. À partir de ces deux derniers modèles, nous proposons d'étudier la gestion du wrapper pour l'adaptation de la simulation à ces deux niveaux.

Dans cette illustration, nous ne décrivons pas l'architecture *QuadriPro* ; cette description n'est pas nécessaire pour notre sujet. Dans un premier temps, nous donnons une spécification technique du problème de co-simulation CABA-PVT et ensuite, nous proposons une pseudo-modélisation et l'implémentation du wrapper.

7.4.1 Spécification technique du problème

Aujourd'hui, les progrès technologiques permettent d'intégrer sur un même support de silicium de plus en plus de fonctions. On conçoit, de nos jours, des systèmes intégrés (SoC, pour System on Chip) réunissant des millions de transistors. Ces systèmes embarquent fréquemment plusieurs micro-processeurs, des fonctions de traitement analogique ou numérique du signal, des interfaces multiples avec le monde extérieur.

Les outils et méthodes utilisés pour la conception de tels dispositifs sont encore ceux qui étaient utilisés pour la conception des circuits spécifiques (ASIC, pour Application Specific Integrated Circuits) beaucoup plus simples. Cette inadéquation conduit principalement à la baisse de la productivité relative des concepteurs et à une proportion croissante du temps de conception consacré à la vérification (essentiellement par simulation, actuellement).

Plusieurs pistes sont proposées pour résoudre ces difficultés :
- le recours à des méthodes de vérification complémentaires de la simulation ; les méthodes formelles permettent une vérification exhaustive de tous les comportements d'un système, moyennant éventuellement quelques simplifications (abstractions),
- la réutilisation d'éléments complexes (les composants virtuels ou IP, pour Intellectual Property) dans des projets successifs.

Pour réutiliser plus facilement un composant virtuel, comme un cœur de microprocesseur ou une fonction de traitement de signal, il est indispensable que son interface soit normalisée. Il peut alors être réutilisé dans un autre environnement. Cette stratégie fait l'objet d'une promotion active par un consortium international d'industriels et de laboratoires de recherche ; VSIA[20] (Virtual Socket Interface Alliance) a produit une norme définissant l'interface des composants virtuels, la norme VCI (Virutal Component Interface).

Un composant respectant la norme VCI présentera donc l'avantage inestimable de s'interfacer simplement et sans développements supplémentaires avec une architecture de système intégré utilisant cette même norme. Seulement, tous les concepteurs ne suivent pas toujours la norme ; certains proposent des composants avec des interfaces respectant d'autres normes. Dans le cas qui nous concerne, nous envisageons la conception d'une architecture ayant des composants définis au niveau d'abstraction CABA (Cycle Accurate / Bit Accurate) et respectant la norme VCI et des composants définis au niveau PVT (Transaction Level Modeling Programmer View with Timing). Comme nous l'avons déjà indiqué, les quatre processeurs, le réseau d'interconnexion ainsi que la mémoire d'instructions sont définis au niveau CABA, et la mémoire de données, définie au niveau PVT.

Le fonctionnement des composants des deux niveaux d'abstraction, CABA et PVT, n'est pas le même :
- l'interface VCI comporte 140 ports, alors qu'une interface PVT ne comporte que deux ports. De plus, très peu des 140 ports sont généralement utilisés ; chaque implémentation doit spécifier les ports utilisés,
- la simulation au niveau CABA est précise au cycle près, alors que le modèle PVT ne comporte que des annotations de temps,
- le format des données échangées sur les deux interfaces de communication n'est pas le même. Par ailleurs, le format des données sur une interface VCI est paramétrable par

[20] http ://www.vsi.org

le nombre de bits d'adresses, le nombre de cellules dans un paquet, et le nombre de bits du bus d'erreur (0, 1 ou 2).

Pour réussir à faire une co-simulation à ces deux niveaux d'abstractions, il est indispensable d'adapter les interfaces de communication des deux niveaux. Il s'agira pour nous de générer un adaptateur (Wrapper) qui assure la conversion des messages entre les deux interfaces.

Nous présentons donc dans la sous-section suivante, la norme VCI ainsi que le modèle de communication au niveau PVT, avant d'aborder la mise en œuvre de la génération du wrapper.

Norme VCI

La norme VCI se décline en trois versions plus ou moins élaborées selon la nature et le rôle des composants. Parmi ces trois versions [63, 28] (PVTCI, pour Peripheral VCI ; BVCI, pour Basic VCI et AVCI, pour Advanced VCI), la PVTCI est la plus simple et c'est elle que nous adoptons dans le cadre de cette expérimentation. Elle définit une interface simple et un protocole comportant deux signaux principaux : **VAL** et **ACK** ; ce qui est suffisant pour établir un protocole de communication de type *handshake*. Ce protocole se décrit comme suit : un *initiateur* (émetteur) envoie un signal VAL à une *cible* (récepteur) l'informant que des données valides sont émises dans son interface et la cible, à son tour, informe l'initiateur par un signal ACK que le transfert s'est déroulé avec succès.

La norme VCI définit une communication point à point et asymétrique entre un *initiateur* et une *cible*. L'initiateur émet des requêtes (de lecture ou d'écriture, simples ou par paquets) et la cible y répond, si elle le peut.

Nous représentons une interface de communication à l'aide d'un ensemble de ports. Dans le cas de la norme VCI, les différents ports de l'interface que nous utilisons sont résumés dans le tableau 7.1 où $b \in \{1, 2, 4\}$, $e \in \{1, 2, 3\}$ *et* $n \in \{0..64\}$. Dans le cas de notre expérimentation, nous avons $b = 4$, $e = 3$ *et* $n = 32$.

Nom (taille en bits)	Origine	Description
VAL (1)	Initiateur	indicateur de la validité d'une requête
RD (1)	Initiateur	indicateur d'une commande de lecture ou d'écriture
ADDRESS (n)	Initiateur	Adresse de lecture ou d'écriture
BE (b)	Initiateur	Byte Enable : octets concernés par la requête
WDATA (8b)	Initiateur	données associées à une requête d'écriture
EOP (1)	Initiateur	End of Packet : signale la fin d'un paquet
ACK (1)	Cible	Indicateur de l'acceptation de la requête courante
RDATA (8b)	Cible	Données associées à une requête de lecture
RERROR (e)	Cible	Signalisation des erreurs

Tab. 7.1 – Ports de la norme VCI

L'initiateur et la cible sont tous deux synchrones sur front montant d'une horloge *CLK* et munis d'un signal de réinitialisation synchrone RESET. Ces deux signaux sont gérés par

le système d'accueil. Lorsque l'initiateur désire soumettre une requête à sa cible, il active le signal VAL, positionne les signaux RD, ADDRESS, BE et WDATA et attend l'acceptation de la requête par la cible (signal ACK). Il ne peut soumettre une nouvelle requête tant que la précédente n'est pas acceptée et il ne peut modifier la requête en cours ni même y renoncer. Il est toutefois possible pour la cible d'avoir un comportement asynchrone : celle-ci peut fournir une réponse à l'initiateur en un cycle d'horloge ; les chronogrammes des figures 7.3 et 7.4 illustrent différentes transactions possibles entre l'initiateur et la cible. Le premier chronogramme décrit une séquence de cycles de lecture et d'écriture simples. Le second présente une séquence où la cible traite toutes les requêtes en un cycle d'horloge.

Comme on peut le constater sur ces chronogrammes, quand l'initiateur a une requête à soumettre, il positionne le signal VAL pour informer la cible. Quand la cible est en mesure de répondre, elle active son signal ACK. Toutes les données sont alors transmises pendant que les deux signaux sont actifs. L'initiateur doit garder les données de la requête tant que le signal VAL est actif. De même, la cible doit aussi garder les données (résultats) de la requête tant que le signal ACK est actif. Au top d'horloge suivant, les deux signaux VAL et ACK doivent être désactivés (mise à zéro), sauf si une autre transaction est immédiatement déclenchée sur une interface.

Notons, enfin, que dans le cas d'une requête de lecture, le champ WDATA (write data) est inutilisé, tout comme dans une requête d'écriture, le RDATA (read data) est inutile.

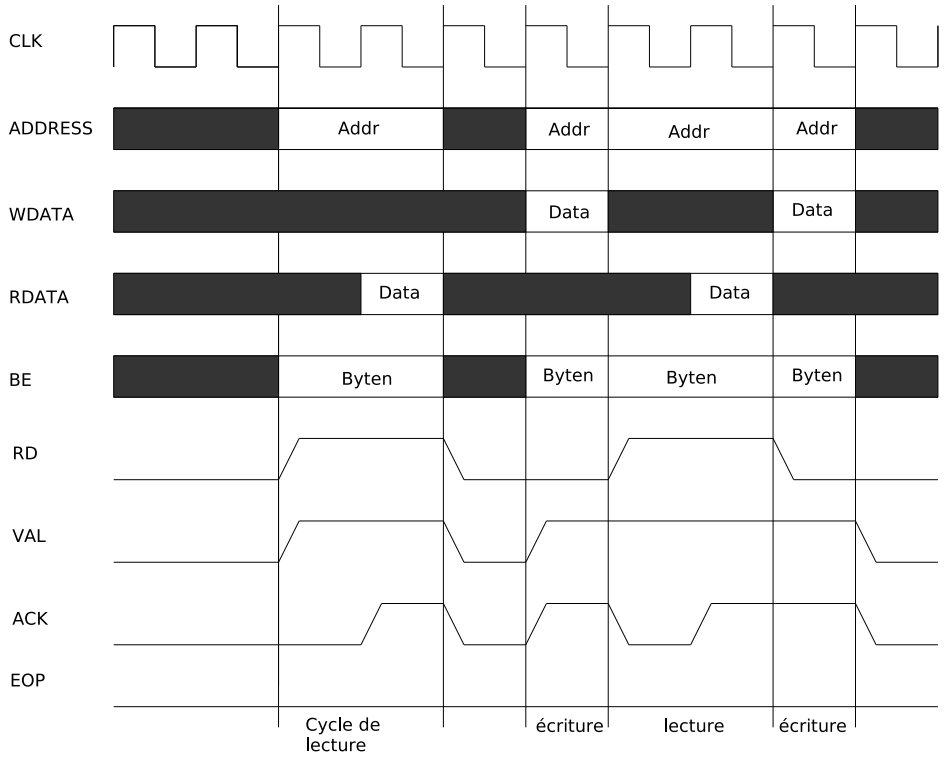

FIG. 7.3 – Exemples de transactions simples

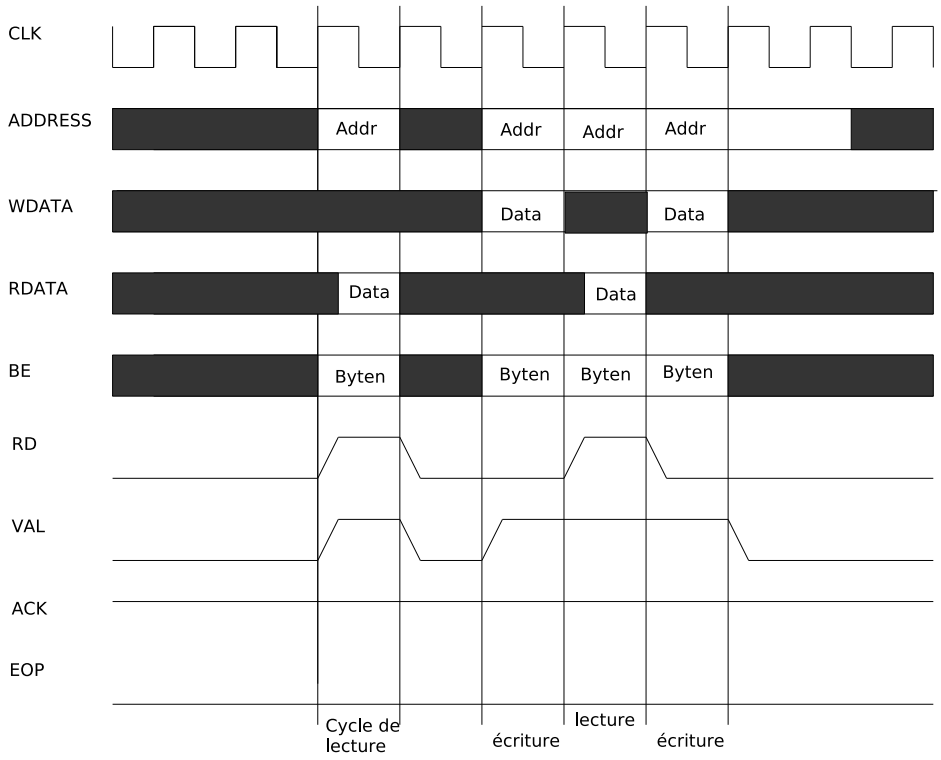

FIG. 7.4 – Exemples de transactions avec une cible asynchrone

Communication au niveau PVT

Face à l'explosion de la complexité des circuits à concevoir et à la croissance exponentielle des données à traiter au niveau RTL - avec les temps de simulation qui en découlent - il devient nécessaire d'élever le niveau d'abstraction dans la conception des circuits intégrés. C'est particulièrement crucial pour les SoC (System on chip) dans lesquels on trouve rassemblés sur un seul circuit plusieurs cœurs de processeurs, de nombreux blocs IP d'origines diverses, une technologie de bus de communication, etc., et pour lesquels le travail de conception n'est plus envisageable au seul niveau RTL. En effet, sur ces circuits, l'importance du logiciel embarqué est primordiale car le matériel n'a aucun sens sans le logiciel qui l'accompagne et qui, bien souvent, devient le facteur différenciateur entre concurrents.

Une récente étude de la société de marché *International Business Strategies* auprès d'ingénieurs de conception de SoC montre, par exemple, que pour des technologies à 250 nm, le développement du logiciel représentait 35 % de l'effort global du design, un chiffre qui grimpe à 55 % pour les circuits en 90 nm. Le même rapport mentionnait que, pour un circuit de 80 millions de portes en 90 nm, le coût de développement du logiciel peut dépasser les 30 millions de dollars. Or, si l'on souhaite tenir des délais de conception de plus en plus courts, il devient impossible d'attendre la mise à disposition d'un prototype matériel du circuit avant d'entamer les développements logiciels. D'où l'idée de remonter d'un niveau d'abstraction la conception et de travailler au niveau transactionnel appelé couramment TLM (Transaction level modeling).

Ce type d'approche consiste à modéliser uniquement les échanges d'informations entre différents modules qui, souvent, représentent les différents blocs IP du SoC. A ce niveau d'abstraction, le circuit est représenté par une plate-forme logicielle au sein de laquelle ces modules communiquent entre eux via des modules d'interconnexion. Un modèle TLM travaille donc uniquement sur des appels de fonctions et des transferts de paquets de données. L'idée est de représenter au plus près l'intention du concepteur quant au comportement global du circuit, sans entrer dans les détails de la description des signaux réalisée au niveau RTL.

Il existe de nombreuses variantes dans le niveau d'abstraction TLM. Néanmoins, on peut distinguer deux grands types. Le premier, appelé *Programmer view* (PVT), est une représentation purement fonctionnelle du circuit sans référence à aucune notion du temps. A ce niveau, le modèle contient toutes les informations nécessaires (et rien de plus) pour que les équipes de développement logiciel puissent travailler, c'est-à-dire faire tourner le logiciel embarqué final - système d'exploitation compris. Le second type, appelé *Programmer View with Timing* (PVT), intègre des informations sur les délais (le timing) qui permettent notamment de travailler sur l'analyse des performances du circuit, sans trop pénaliser les temps de simulation.

Les avantages de cette approche sont nombreux. En effet, on peut démarrer très en amont le développement des logiciels embarqués sur un modèle représentatif du circuit final, tout en bénéficiant de la vitesse de simulation importante, de 100 à 1 000 fois plus rapide qu'au niveau RTL. Il est aussi possible de réaliser une analyse architecturale du circuit afin de réaliser le meilleur partitionnement logiciel/matériel possible sur des bases chiffrées. Enfin, et ce n'est pas le moindre avantage, la vérification fonctionnelle du circuit est améliorée grâce à l'écriture d'un testbench que l'on pourra par la suite réutiliser au niveau RTL.

Au final, avec cette méthodologie, on obtient une meilleure visibilité de son circuit, une meilleure prédiction de ses performances et une plus grande confiance dans le design. Dans le cas de notre étude, nous nous intéressons surtout au niveau d'abstraction PVT.

7.4.2 Vers une génération de wrapper

Avant d'aborder la modélisation du wrapper, nous présentons très succintement le diagramme de machine à états d'UML dont certains concepts sont utilisés dans notre modélisation.

Diagramme de State Machines

UML 2.0 comporte un ensemble de treize diagrammes permettant de décrire la structure et le comportement. Le diagramme de machines à états (*state machines*) peut être utilisé pour modéliser un comportement discret via un système d'*états-transitions*. Il peut également être utilisé pour spécifier le protocole d'usage d'un élément de système. Il existe deux types de *state machines* : les *behavioral state machines* et les *protocol state machines*.

Les *behavioral state machines* sont en général utilisées pour modéliser le comportement individuel des entités (par exemple, des instances de classe). Le formalisme utilisé pour décrire les machines à états de ce type est une variante orientée objet des *Statecharts* de Harel [62].

Les *protocol state machines* expriment les transitions légales qu'un *classifier* peut déclencher. Elles constituent un moyen adéquat pour définir le cycle de vie des objets ou un ordre d'invocation des opérations d'un objet. Les *protocol state machines* n'excluent pas une implémentation particulière et peuvent, de ce fait, être associées à des interfaces et ports. Dans la suite de cette description, nous nous intéressons aux *protocol state machines*. Nous ne présentons que les concepts utilisés dans notre travail ; le lecteur qui désire en savoir davantage sur les machines à états pourra se référer au chapitre 15 de la spécification de la superstructure d'UML 2.0 [60].

Les concepts de base dans un diagramme de machine à états sont : les *états*, les *transitions* et les *actions*. Les états représentent les conditions d'existence de l'objet défini ou les différentes étapes de son modèle de comportement, les transitions spécifient les changements d'états et les progressions dans le comportement de l'objet en réponse à des événements, et les actions définissent un comportement atomique qui peut être exécuté par la machine à états. Ces actions sont relatives à n'importe quel type d'action en UML qui peut être une opération simple, telle qu'une addition, ou une opération plus complexe, telle que l'invocation de méthode.

Dans ce diagramme, les états sont représentés par des rectangles arrondis et les transitions par des arcs orientés, comme illustré par la figure 7.5.

FIG. 7.5 – Notations des états et des transitions dans un diagramme de machine à états en UML

La syntaxe de l'étiquette d'une transition est définie comme suit :

[pre-condition] operation (parameter-list) / [post-condition]

[pre-condition] définit une condition qui doit être vérifiée pour que la transition soit déclenchée, ce qui, dans notre cas, revient à dire qu'elle définit une condition qui doit être satisfaite pour que l'opération indiquée soit invoquée.

operation(parameter-list) spécifie une opération avec la liste de ses paramètres. L'opération ne peut être appelée que lorsque la pré-condition est satisfaite.

[post-condition] définit une condition qui doit être vérifiée après l'exécution de l'opération indiquée, c'est-à-dire, après le déclenchement de la transition.

La figure 7.6 représente un exemple simple d'un diagramme de machine à états en UML avec une pré-condition *C1*, une opération *m1* et une post-condition *C2*.

Modélisation du wrapper CABA-PVT

Dans notre cas d'étude, la mémoire PVT est une mémoire partagée, accessible à toutes les unités de calculs (*procUnit*). Cette mémoire sera donc considérée comme *cible* dans notre

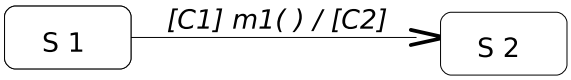

FIG. 7.6 – Exemple Simple d'un diagramme de machine à états en UML

modèle de communication. En effet, la mémoire ne sera jamais initiatrice de transactions. Les unités de calculs accèdent à la mémoire via un réseau d'interconnexion. Ce réseau de communication étant aussi défini au niveau CABA et disposant d'une interface VCI, nous allons concevoir un wrapper entre ce dernier et la mémoire (voir figure 7.7). Les différentes unités de calculs possèdent des interfaces VCI initiateur. Le wrapper contient un module VCI target qui assure la communication avec l'interface VCI initiateur du réseau. Il traduit les requêtes reçues sur son interface VCI target en appel de méthodes (Read/Write) de l'interface PVT. Le réseau d'interconnexion sera donc l'initiateur des communications vers le wrapper.

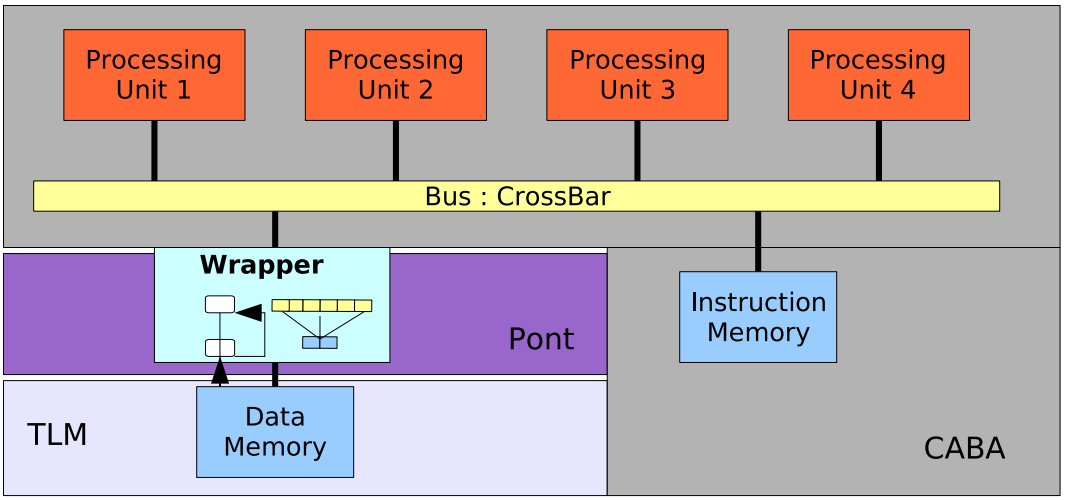

FIG. 7.7 – Architecure l'architecture QuadriPro avec le wrapper

Bien que l'illustration porte sur la conception de wrapper entre le réseau d'interconnexion et la mémoire, l'approche reste valable entre deux composants quelconques pourvu que l'un des deux composants soit défini au niveau CABA et muni d'une interface VCI Initiateur et l'autre au niveau PVT.

Dans notre modélisation qui sera présentée ici, un certain nombre de restrictions ont été faites :

- l'expression du temps n'apparaît pas explicitement. Ètant donné que dans la norme VCI l'initiateur et la cible sont tous les deux synchrones sur front montant d'une horloge CLK et munis d'un signal de réinitialisation synchrone RESET, nous faisons l'hypothèse que ces deux signaux (CLK et RESET) sont gérés par le système global,
- nous limitons également le protocole de communication au protocole *handshake simple*, tel que défini dans la description de l'interface VCI,

- enfin, nous nous limitons à un fonctionnement requête/réponse ; c'est à dire que lorsqu'un initiateur émet une requête, il doit attendre la réponse de la cible avant d'en émettre une autre.

Comme nous l'avons déjà mentionné, le wrapper est un traducteur de protocole ou pour être plus précis, un traducteur d'interface de communication. Il permet de convertir les signaux d'un protocole source vers des signaux d'un protocole cible.

Nous rappelons ici que les composants au niveau CABA utilisent chacun une interface VCI pour communiquer. Dans le tableau 7.1, nous avons déjà donné la liste des ports et signaux d'une interface VCI, nous avons aussi donné la liste des méthodes de communication qui composent une interface de composant défini au niveau PVT.

Sachant que la communication selon la norme VCI s'effectue d'un initiateur vers une cible, le wrapper qui s'intercale entre les deux composants CABA et TLM possédera une interface VCI-Target pour assurer sa communication avec le composant CABA. Nous disons dans ce cas que le wrapper est de type *VCI-Target*. Il doit également disposer d'un module permettant de réaliser des appels aux méthodes du niveau PVT.

Le fonctionnement du wrapper peut être décrit à l'aide d'un automate. Nous utilisons les concepts des *Machines à états* d'UML 2 (présentées plus haut) pour le modéliser.

La figure 7.8 donne une représentation sous forme d'automate du fonctionnement du wrapper dans le cas d'une communication utilisant le protocole *handshake simple*. Comme dans les machines à états, les transitions comportent des pré-conditions, des actions (opérations) et des post-conditions. Les conditions (pré et post) sont écrites entre crochets et portent sur les valeurs des signaux de l'interface VCI. Les fonctions *read* et *write* sont les fonctions de l'interface PVT. Les paramètres de ces fonctions sont : *read*(*ADDRESS*, *RDATA*, *BE*, *t*) *et write*(*ADDRESS*, *WDATA*, *BE*, *t*). Une transition de notre automate est donc constituée d'une ou plusieurs pré-conditions, suivie éventuellement d'une ou de plusieurs action(s) (opérations), elles-mêmes suivies éventuellement d'une ou de plusieurs post-conditions. Les post-conditions déterminent les valeurs des différents signaux après le déclenchement de la transition. Dans le fonctionnement de notre automate, une transition n'est déclenchable que lorsque toutes ses pré-conditions sont satisfaites. Par ailleurs, le déclenchement de la transition se fait sur le front montant de l'horloge du système.

Nous avons montré que le fonctionnement du wrapper pouvait être modélisé à l'aide d'un automate à état de type machines à états d'UML 2. En plus de cet automate qui contrôle le fonctionnement, le wrapper contient un module qui implémente une interface *VCI Target* et une référence sur le composant mémoire partagée qui lui permet de pouvoir appeler les fonctions de celui-ci. Par ailleurs, le wrapper doit également gérer les éventuelles conversions de types de données échangées. Nous considérons, ici, l'existence d'une bibliothèque externe accessible au wrapper et réalisant ces conversions.

Notre wrapper est donc constitué de :
- un automate,
- un module CVI-Target,
- une référence vers le composant mémoire partagée,
- une référence à la bibliothèque de conversion des types de données,
- l'ensemble des ports VCI utilisés.

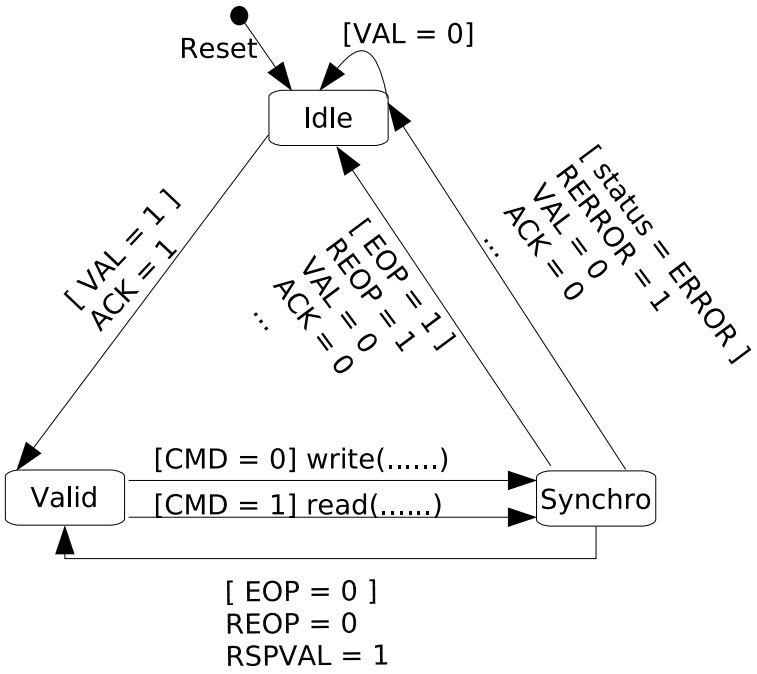

FIG. 7.8 – Automate partiel du wrapper CABA-PVT

Le but de notre démarche est de générer le code en SystemC du wrapper. Pour ce faire, nous définissons un métamodèle du wrapper que nous utiliserons avec un template Jet [5] pour l'opération de génération de code.

Les différents concepts de ce métamodèle sont résumés dans le tableau 7.2.

Concepts	**Significations**
Wrapper	concept racine du métamodèle.
Automaton	automate décrivant le fonctionnement du wrapper.
State	état de l'automate du wrapper.
Transition	transition de l'automate du wrapper.
FunctionCall	appel de méthode TLM.
Parameters	désigne les paramètres dans les appels de méthodes TLM.
Condition	expression booléenne permettant de spécifier les conditions de déclenchement des transitions. L'attribut *isPrecondition* permet de savoir si la condition est une pré-condition ou une post-condition.
Port	désigne un port VCI

TAB. 7.2 – Concepts du métamodèle de wrapper CABA-PVT

La figure 7.9 correspond au métamodèle. Le concept *Wrapper* contient une collection de *Port*. Cette collection correspond à l'ensemble des ports VCI utilisés. En effet, dans chaque utilisation d'une interface VCI, il faut préciser lesquels des 140 ports sont réellement exploités.

Bien que dans le cas qui nous concerne, cette déclaration n'est pas indispensable, nous l'avons toute de même conservée pour être plus général.

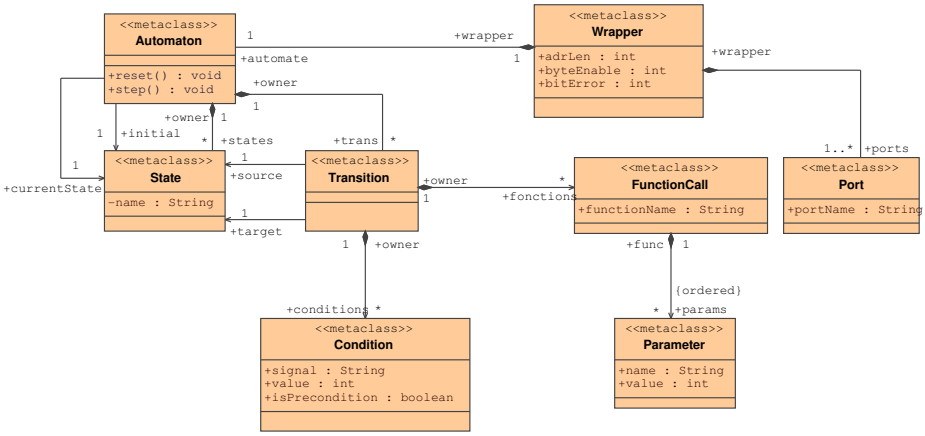

FIG. 7.9 – Métamodèle du wrapper CABA-PVT

Génération du wrapper

Nous arrivons à la dernière étape de notre processus de construction du wrapper de l'exemple. Il s'agit de la génération du code d'implémentation en SystemC. Cette génération de code est réalisée à l'aide de template Jet [5] à partir d'une instance du métamodèle. Le listing suivant correspond au code du template. Le code SytemC généré et le modèle (instance du métamodèle) en entrée de la génération sont fournis en annexe A.1 et A.2 du document. Pour rappel, la figure 7.8 donne une vue graphique partielle du modèle.

Le principe de génération consiste d'abord à charger le modèle instance du wrapper (lignes 28 - 39), et à récupérer dans le modèle les données nécessaires à la déclaration de l'interface VCI target (lignes 53 et 54). Puis, on parcoure l'automate du wrapper. Pour chaque transition, on récupère l'ensemble des post-conditions servant à positionner les valeurs des différents signaux (lignes 93-102 ; 119-128 ; 132-140 ; 144-152 ; 155-163 ; 167-176 et 180-188).

Les états de l'automate deviennent des tests sur les valeurs des signaux VAL, ACK et RD (les trois *if* de la fonction *transition*).

Template de génération de code SystemC du Wrapper

```
1. <%@ jet
2. package="com.ibm.pdc.example.jet.gen"
3. class="TemplateWrapper"
4. imports="java.io.IOException org.eclipse.emf.common.util.URI
5. org.eclipse.emf.ecore.resource.Resource org.eclipse.emf.ecore.resource.ResourceSet
6. org.eclipse.emf.ecore.resource.impl.ResourceSetImpl
```

```
7. org.eclipse.emf.ecore.util.EcoreUtil
8. org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl
9. wrapperMM.Automaton wrapperMM.State wrapperMM.Transition wrapperMM.Wrapper
10. wrapperMM.WrapperMMFactory wrapperMM.SignalSetting wrapperMM.string
11. wrapperMM.WrapperMMPackage wrapperMM.Condition
12. java.util.* java.io.*"
13. %>
14. /*************************************************/
15. // Sujet   : Wrapper SystemC CABA/TLM
16. // Auteurs : Rabie Ben-Atitallah &  Lossan BONDE
17. // Date    : Juillet 2006
18. /*************************************************/
19. #include <systemc.h>
20. #include <stdio.h>
21. #include "shared/soclib_vci_interfaces.h"
22. #include "shared/soclib_segment_table.h"
23. #include "bus_types.h"
24. #include "basic_timed_initiator_port.h"
25.
26. using basic_protocol::basic_timed_initiator_port;
27.
28. <% WrapperMMFactory  fact = WrapperMMFactory.eINSTANCE;
29. URI fileURI=URI.createFileURI(new File("Model.wrappermm").getAbsolutePath());
30. ResourceSet resourceSet1 = new ResourceSetImpl();
31. resourceSet1.getResourceFactoryRegistry().getExtensionToFactoryMap().
32.  put(Resource.Factory.Registry.DEFAULT_EXTENSION,new XMIResourceFactoryImpl());
33. WrapperMMPackage wPackage=WrapperMMPackage.eINSTANCE;
34. Resource resource=resourceSet1.getResource(fileURI,true);
35. Wrapper wModel = (Wrapper) EcoreUtil.getObjectByType(resource.getContents(),
36. wPackage.getWrapper());
37. Automaton automate = (Automaton) EcoreUtil.getObjectByType(resource.
38. getContents(),wPackage.getAutomaton());
39. Transition trans;
40. %>
41.
42. ////////////////////////////////////////
43. // structure definition
44. ////////////////////////////////////////
45.
46. struct GASPARD_VCI_WRAPPER : sc_module {
47.
48. // IO PORTS
49. basic_timed_initiator_port<ADDRESS_TYPE,DATA_TYPE> initiator_port;
50. void run();
51. sc_in<bool>  CLK;
52.
53. ADVANCED_VCI_TARGET<<%=wModel.getAdrLen()%>,<%=wModel.getByteEnable()%>,
```

```
54.  <%=wModel.getBitError()%>>  VCI;
55.
56. const char  *NAME;
57.
58. ADDRESS_TYPE    adr;
59. DATA_TYPE     wdata;
60. int     rdata;
61. int     be;
62. int     cmd;
63. int             t;
64. DATA_TYPE       d;
65. int             eop;
66. int id;
67. ////////////////////////////////////////////////
68. // constructor
69. ////////////////////////////////////////////////
70.
71. SC_HAS_PROCESS(GASPARD_VCI_WRAPPER);
72. GASPARD_VCI_WRAPPER(sc_module_name   insname)    // segment table pointer
73. : sc_module( insname ) ,
74. initiator_port("iport")
75. {
76.  SC_METHOD (transition);
77.  sensitive_neg << CLK;
78.  NAME = (const char*) insname;
79.  printf("Successful Instanciation of GASPARD_VCI_WRAPPER : %s\n",NAME);
80. }; // end constructor
81. void transition()
82. {
83. if (VCI.VAL == true) {
84.   if (VCI.ACK==true){
85. adr = ((int)VCI.ADDRESS.read());
86. wdata   = (int)VCI.WDATA.read();
87. be      = (int)VCI.BE.read();
88. cmd     = (int)VCI.RD.read();
89.             eop     = (int) VCI.EOP.read();
90.             id= (int)VCI.SCRID.read();
91.  if ((cmd & 0x1) == 0x1) {
92.      initiator_port.read( adr , d , t );
93.        <% trans =(Transition) automate.getTrans().get(3);
94.        for(Iterator iter = trans.getSignalvals().iterator(); iter.hasNext();){
95.         Condition cond = (Condition) iter.next();
96.         if (!cond.isIsPrecondition()){
97.         String sigName = cond.getSignal();
98.         int value = cond.getValue(); %>
99.         VCI.<%=sigName%> = <%=value%>
100.         <%}
```

```
101.
102.       } %>
103.       VCI.RDATA = d;
104.       VCI.RSCRID = id;
105.        if (eop==1)
106.        {
107.        <% trans =(Transition) automate.getTrans().get(5);
108.        for(Iterator iter = trans.getSignalvals().iterator(); iter.hasNext();){
109.         Condition cond = (Condition) iter.next();
110.         if (!cond.isIsPrecondition()){
111.         String sigName = cond.getSignal();
112.         int value = cond.getValue(); %>
113.         VCI.<%=sigName%> = <%=value%>
114.         <%}
115.
116.       } %>
117.        }
118.        else {
119.        <% trans =(Transition) automate.getTrans().get(4);
120.        for(Iterator iter = trans.getSignalvals().iterator(); iter.hasNext();){
121.         Condition cond = (Condition) iter.next();
122.         if (!cond.isIsPrecondition()){
123.         String sigName = cond.getSignal();
124.         int value = cond.getValue(); %>
125.         VCI.<%=sigName%> = <%=value%>
126.         <%}
127.
128.       } %>
129.        }
130.      }else{
131.      initiator_port.write( adr , wdata , t );
132.        <% trans =(Transition) automate.getTrans().get(2);
133.        for(Iterator iter = trans.getSignalvals().iterator(); iter.hasNext();){
134.         Condition cond = (Condition) iter.next();
135.         if (!cond.isIsPrecondition()){
136.         String sigName = cond.getSignal();
137.         int value = cond.getValue(); %>
138.         VCI.<%=sigName%> = <%=value%>
139.         <%}
140.       } %>
141.       VCI.RSCRID = id;
142. if (eop==1)
143.        {
144.        <% trans =(Transition) automate.getTrans().get(5);
145.        for(Iterator iter = trans.getSignalvals().iterator(); iter.hasNext();){
146.         Condition cond = (Condition) iter.next();
147.         if (!cond.isIsPrecondition()){
```

```
148.          String sigName = cond.getSignal();
149.          int value = cond.getValue(); %>
150.          VCI.<%=sigName%> = <%=value%>
151.          <%}
152.        } %>
153.         }
154.         else {
155.         <% trans =(Transition) automate.getTrans().get(4);
156.         for(Iterator iter = trans.getSignalvals().iterator(); iter.hasNext();){
157.          Condition cond = (Condition) iter.next();
158.          if (!cond.isIsPrecondition()){
159.          String sigName = cond.getSignal();
160.          int value = cond.getValue(); %>
161.          VCI.<%=sigName%> = <%=value%>
162.          <%}
163.        } %>
164.        }
165. }
166.       else{
167.      <% trans =(Transition) automate.getTrans().get(1);
168.         for(Iterator iter = trans.getSignalvals().iterator(); iter.hasNext();){
169.          Condition cond = (Condition) iter.next();
170.          if (!cond.isIsPrecondition()){
171.          String sigName = cond.getSignal();
172.          int value = cond.getValue(); %>
173.          VCI.<%=sigName%> = <%=value%>
174.          <%}
175.
176.        } %>
177.      }
178.   } else{
179.          if (VCI.RSPACK == true) {
180. <% trans =(Transition) automate.getTrans().get(6);
181.         for(Iterator iter = trans.getSignalvals().iterator(); iter.hasNext();){
182.          Condition cond = (Condition) iter.next();
183.          if (!cond.isIsPrecondition()){
184.          String sigName = cond.getSignal();
185.          int value = cond.getValue(); %>
186.          VCI.<%=sigName%> = <%=value%>
187.          <%}
188.        } %>
189. }
190.         }
191. };
192.
193. };
```

Idée de généralisation

Le métamodèle de wrapper CABA-PVT que nous proposons est en principe aussi valable pour CABA - PV dans la mesure où la différence entre PVT et PV est l'annotation du temps. Cette annotation du temps est prise en compte par un paramètre dans les fonctions *read* et *write* du PVT. On peut même généraliser en disant que ce métamodèle pourrait marcher pour CABA et n'importe quel autre niveau d'abstraction fonctionnel. En poursuivant cette idée de généralisation, on pourraît étendre note métamodèle à celui présenté à la figure 7.10 où le concept de *FunctionCall* a été généralisé en *Action* pour prendre en compte aussi bien les appels de méthodes (fonctions) que des valeurs de signaux. Mais le problème n'est pas aussi simple ; si nous modifions un peu les données de notre problème, en considérant que l'initiateur peut envoyer une requête sans attendre la réponse de la précédente, on s'aperçoit que le wrapper devra disposer d'autres concepts, comme par exemple une FIFO pour gérer les requêtes et leurs réponses.

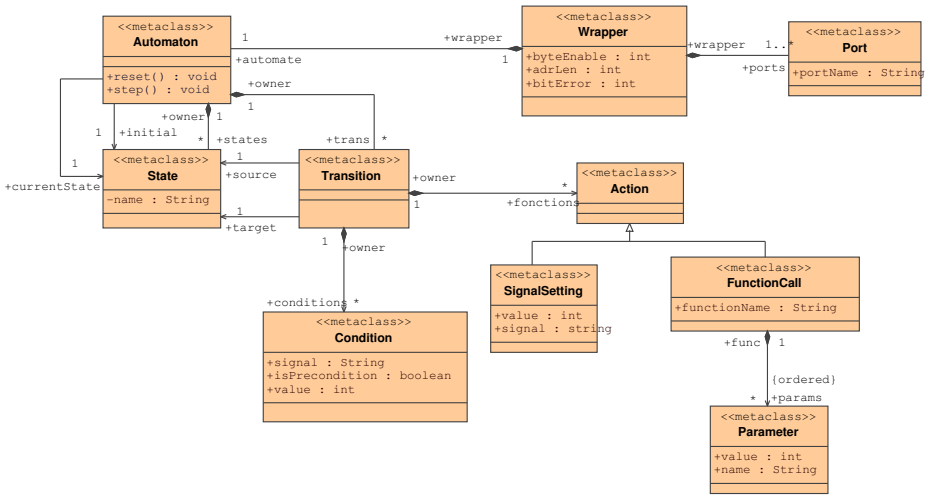

FIG. 7.10 – Métamodèle étendu du wrapper CABA-PVT

Ce travail pourrait se poursuivre par l'étude d'autres cas afin d'aboutir à une classification de wrappers. L'étape suivante consisterait à proposer un métamodèle commun à ces classes de wrappers, puis à étudier la génération automatique à partir des modèles issus de ce métamodèle.

7.5 Conclusion

Dans ce chapitre, nous avons expliqué pourquoi nous n'avons pas pu proposer une solution générale pour la génération automatique de wrappers de simulation multi-niveaux. La

difficulté essentielle de cette limitation étant la modélisation des protocoles de communication. Nous avons néanmoins proposé une solution qui consiste à concevoir une bibliothèque de wrappers qui adaptent des interfaces et protocoles bien spécifiés. La trace des transformations de modèles pourra alors servir à déterminer les composants à adapter, et sélectionner les wrappers appropriés dans une bibliothèque. Enfin, nous avons exposé la démarche de conception de ces wrappers à l'aide d'une adaptation de simulation CABA et PVT.

Dans cette illustration, nous avons aussi montré que les wrappers peuvent être générés automatiquement par transformation de modèle à condition de savoir les modéliser. Par conséquent, il est important de poursuivre la recherche dans ce domaine.

Conclusion et perspectives

Bilan

Dans cette thèse, nous nous sommes intéressés à la conception de l'interopérabilité dans Gaspard. Nous avons proposé une démarche basée sur l'utilisation de la traçabilité dans les transformations de modèles. Ce travail nous a conduits à nous intéresser à deux domaines de recherche :

1. l'Ingénierie Dirigée par les Modèles (IDM) ; nous avons pu expérimenter comment l'IDM ouvre de nouvelles voies d'investigation en matière de conception de logiciels en centrant le développement de logiciels sur la modélisation et, ensuite, par des opérations de transformations de modèles, envisager une automatisation complète du processus. Dans cette approche, nous avons montré que le principal défi qui reste à relever est celui des transformations de modèles. En effet, comme nous l'avons mentionné au chapitre 2, des outils plus évolués et environnement de conception et réalisation des transformations sont attendus pour entrevoir une vulgarisation de l'IDM.

   Nous avons pu mettre en œuvre l'approche IDM pour proposer une solution d'interopérabilité dans Gaspard. Pour ce faire, nous avons développé un métamodèle de pont d'interopérabilité dont les instances (modèles) sont obtenues par des opérations de transformations de modèles de traçabilité conformément à un métamodèle de traçabilité. Les travaux réalisés dans ce cadre ont contribué à trois publications [25, 24, 23] en chapitres de livres.
2. la problématique de conception des systèmes embarqués et, en particulier, la conception de SoCs à base d'IP. En l'occurrence, nous avons pu découvrir et comprendre les enjeux de la co-conception et co-simulation, la place de plus en plus importante qu'occuppe le logiciel dans ces systèmes et la place de celui-ci dans le coût et le temps de conception des systèmes. Le travail réalisé dans ce domaine n'a pa permis de proposer une solution complète pour la co-similation à plusieurs niveaux d'abstraction, faute de trouver un formalisme adéquat pour métamodéliser les protocoles de communication. Nous avons illustré, au chapitre 7, une approche qui consiste à résoudre le problème pour chaque couple de niveaux d'abstractions, de protocoles et d'interfaces de communication donnés.

Bien que la solution proposée pour aborder la question d'interopérabilité des langages ait été conçue dans le cadre des systèmes embarqués, nous pensons qu'elle peut être exploitée dans d'autres contextes. Par exemple, on peut envisager son utilisation dans la modélisation d'une application web à base d'EJB. Le modèle de référence décrira l'application à un haut niveau d'abstraction, et les technologies cibles pourront être le container web et le

container EJB. Le Pont d'interopérabilité (bridge) servira alors à générer les fichiers utiles à l'interconnexion.

Par ailleurs, le métamodèle de traçabilité proposé reste un métamodèle de trace général qui peut être utilisé à d'autres fins. Par exemple, il peut être exploité dans le cadre de l'ingénierie des besoins. Dans ce domaine, la trace est souvent utilisée pour établir des relations entre les besoins des utilisateurs et les artefacts d'implémentation. Grâce à ces relations, il est alors possible de faire une analyse des implications d'un changement dans les besoins sur les modèles de conception et d'implémentation.

Perspectives

Le travail réalisé dans cette thèse donne lieu à quelques perspectives. Au cours de nos travaux, nous avons proposé une démarche et une spécification permettant de résoudre le problème d'interopérabilité dans Gaspard. L'implémentation de la solution n'ayant pas pu être menée à bout, notre première perspective sera de terminer ce travail par une implémentation logicielle du pont d'interopérabilité.

Nous pensons qu'il est important de poursuivre l'étude de l'interopérabilité des niveaux d'abstraction de simulation. Le travail que nous avons effectué pour l'adaptation CABA-PVT est un début. Il serait intéressant de poursuivre les spécifications ; il pourrait se dégager des patterns communs à plusieurs cas d'adaptation, et dans le meilleur des cas, on pourrait aboutir à un métamodèle unique. Il serait alors possible d'automatiser complètement le procesus de génération des wrappers par transformations de modèles.

Par ailleurs, le mécanisme de trace proposé peut être exploité à d'autres fins :

- il pourrait être utilisé pour détecter les éléments de modèles à l'origine d'erreurs d'exécution : c'est à dire, la mise en œuvre de débogage au niveau modèle,
- il pourrait être aussi exploité pour gérer la synchronisation de modèles par déclenchement automatique de transformation. Pour ce faire, un mécanisme de transformations incrémentales sera nécessaire,
- enfin, l'on pourrait par exemple envisager de construire un système de génération automatique de fichier de configuration pour des applications web à base d'EJB à partir de notre approche.

Enfin, le mécanisme d'interopérabilité proposé peut servir à faire des estimations et optimisation du coût de communication dans système ; dans la mesure où les communications qui passent par le pont d'interopérabilité sont connues. Il est alors possible de définir des coefficients de pondération pour chaque type de liens de communication, et ainsi pouvoir réaliser des calculs.

Bibliographie

[1] http://modfact.lip6.fr.

[2] The atlas transformation language (atl). http://www.sciences.univ-nantes.fr/lina/atl/.

[3] Codagen architect 3.0. http://www.codagen.com/products/architect/default.html.

[4] extensible markup language (xml). http://www.w3c.org/xml.

[5] Java emitter templates (jet). part of the eclipse modeling framework. http://eclipse.org/articles/Article-JET2/jet_tutorial2.html.

[6] Kermeta. http://www.kermeta.org/.

[7] Wikipédia l'encyclopédie libre. http://fr.wikipedia.org/wiki/Système.

[8] The java model driven architecture 0.2. http://sourceforge.net/projects/jamda/, 2003.

[9] France Telecom Research & and Development. The modelware/mddi qvt implementation. http://universalis.elibel.tm.fr/qvt/.

[10] Ivan Aaen, Peter Bøtcher, and Lars Mathiassen. The software factory : Contributions and illusions. In *Proceedings of the Twentieth Information Systems Research Seminar*, Oslo, Scandinavia, 1997.

[11] Inc. Accellera Organization. Sytemverilog 3.1a language reference manual. http://www.eda.org/sv/, 2004.

[12] Netta Aizenbud-Resher, Richard F. Paige, Julia Rubin, Yael Shalam-Gafni, and Dimitrios S. Kolovos. Operational semantics for traceability. In *ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings*.

[13] Cockburn Alistair. *Agile Software Development*. Boston : Addison-Wesley, 2002.

[14] Abdelkader Amar. *Environnement de simulation fonctionnel distribué et dynamique pour systèmes embarqués*. PhD thesis, Université des Sciences et Technologies de Lille, 2003.

[15] Paul Arkley and Paul Manson ans Steve Riddle. Enabling traceability. *1st International Workshop on Traceability in Emerging Forms of Software Engineering*, 2002.

[16] Ove Armbrust, Alexis Ocampo, and Martin Soto. Tracing process model evolution : A semi-formal process modeling approach. In *ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings,*.

[17] Rabie Ben Atitallah, Smail Niar, Alain Greiner, Samy Meftali, and Jean Luc Dekeyser. Estimating energy consumption for an mpsoc architectural exploration. In *ARCS'06 : Architecture of Computing Systems*, Frankfurt, Germany, March 2006.

[18] Ramesh B. and Jarke M. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 2001.

[19] Xavier Blanc. *MDA en action* `Ingénierie logicielle guidée par les modèles`. Editions Eyrolles, 2005.

[20] Xavier Blanc. Modelbus : A modelware white paper. `http://www.eclipse.org/proposals/eclipse\_mddi/main\_data/ModelBusWhitePaper\_MDDI.pdf`, 2005.

[21] Xavier Blanc. Modelbus, an open and distributed platform for mdd tool interoperability. `http://neptune.irit.fr/Public/AnglaisV2/Neptune2006/P5\_XBlanc.ppt`, 2006.

[22] Carsten Böke. Combining two customization approaches : Extending the customization tool terecs for software synthesis of real-time execution platforms. In *Proceedings of the Workshop on Architectures of Embedded Systems (AES 2000)*, Karlsruhe, Germany, 2000.

[23] Lossan BONDE, Pierre BOULET, and Jean-Luc DEKEYSER. Traceability and interoperability at different levels of abstractions in model transformations. In *Forum on Specification and Design Languages*. FDL'05, Lausanne, Switzerland, September 2005.

[24] Lossan Bondé, Pierre Boulet, Arnaud Cucurru, Jean-Luc Dekeyser, Cédric Dumoulin, Philippe Marquet, Samy Meftaly, and Mickaël Samyn. *Model Driven Engineering for Distributed Real-Time Embedded Systems.*, chapter Model Driven Architecture for Intensive Embedded Systems. ISTE, Hermes science and Lavoisier, August 2005.

[25] Lossan Bondé, Cédric Dumoulin, and Jean-Luc Dekeyser. *Advances in Design and Specification Languages for SoCs*, chapter Metamodels and MDA Transformations for Embedded Systems. Springer, 2005.

[26] Pierre Boulet, Cédric Dumoulin, and Antoine Honoré. *MODEL DRIVEN ENGINEERING FOR DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS : From MDD Concepts to Experiments and Illustrations*, chapter Model Driven Engineering for System-on-Chip Design. ISTE, 2006.

[27] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2004.

[28] Annette Bunker and Ganesh Gopalakrishnan. Formal specification of the virtual component interface standard in the unified modeling language. `http://www.cs.utah.edu/research/techreports/2001/pdf/UUCS-01-007.pdf`.

[29] Jean Bézivin. In search of a basic principle for model driven engineering. *The European Journal for the Informatics Professional*, V(2), April 2004.

[30] Jean Bézivin, Hugo Brunelière, Frédéric Jouault, and Ivan Kurtev. Model engineering support for tool interoperability.

[31] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the atl model transformation language : Transforming xslt into xquery. In *OOPSLA 2003 Workshop*, Anaheim, California, 2003.

[32] Pons C., Kutsche, and R. D. Traceability across refinement steps in uml modeling. 2003.

[33] CBOP, DSTC, and IBM. Mof query/views/transformations, revised submission. omg document : ad/03-08-03.

[34] E. Chen, Y. Shi, D. Zhang, and G. Xu. A programming framework for service association in ubiquitous computing environments. In *Proceedings of the 2003 Joint Conference of the fourth Pacific Rim Conference on Multi media (ICICS-PCM 2003*, 2003.

[35] Cleland-Huang, J. Chang, C. K., and Christensen M. Event-based traceability for managing evolutionary change. In *IEEE Transactions on Software Engineering, September 2003*.

[36] Compuware. Optimalj 3.0, user's guide,. `http://www.compuware.com/products/optimalj`.

[37] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, and D. Pataricza, A.and Varro. Viatra - visual automated transformations for formal verification and validation of uml models. In *Proceedings. ASE 2002. 17th IEEE International Conference*. Automated Software Engineering, September 2002.

[38] Arnaud Cuccuru. *Modélisation Unifiés des Aspects Répétitifs dans la Conception Conjointe Logicielle/Matérielle des Systèmes sur Puce à Hautes Performances*. PhD thesis, Université des Sciences et Technologies de Lille, 2005.

[39] Isabelle Demeure and Elie Najm. *Les intergiciels : développement récents dans CORBA, Java RMI et les agants mobiles*. Lavoisier, 2002.

[40] Observatoire des technologies du temps réel embarqué. `http://www.EmbeddedTouch.com`.

[41] ITRS Design. 2001 edition. `http://public.itrs.net/`.

[42] Cédric Dumoulin. ModTransf : A model to model transformation engine, November 2004. `http://www.lifl.fr/west/modtransf`.

[43] GMT Consortium Edward D. Willink. Umlx : A graphical transformation language for mda. `www.eclipse.org/gmt`, September 2003.

[44] C. Ensel. A scalable approach to automated service dependency modeling in heterogeneous environments. In *5th International Entreprise Distributed Object Computing Conference (EDOC'01)*, 2001.

[45] Malcon Eva. *SSADM Version 4 : A User's Guide*. McGraw-Hill Publishing Co, April 1994.

[46] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles au-déla du MDA*. Hermès Science, Lavoisier, Janvier 2006.

[47] International Technologie Roadmap for Semiconductors. Design edition. `http://www.itrs.net/common/2005ITRS/Design.pdf`, 2005.

[48] D. D. Gajski and R. Kuhn. Guest editor introduction : New VLSI-tools. 16(12) :11–14, December 1983.

[49] Lovic Gauthier. *Génération de système d'exploitation pour le ciblage de logiciel multitâches sur des architectures multi-processeurs hétérogènes dans le cadre des systèmes embarqués spécifiques*. PhD thesis, Institut National Polytechnique de Grenoble (INPG), laboratoire TIMA, mai 2001.

[50] Jean-Marc Geib, Christophe Gransart, and Philippe Merle. *CORBA Des concepts à la pratique*. DUNOD, 1999.

[51] Férid Gharsalli, Damien Lyonnard, Samy Meftali, Frédéric Rousseau, and Ahmed A. Jerraya. Unifying memory and processor wrapper architecture in multiprocessor soc design. In *ISSS '02 : Proceedings of the 15th international symposium on System Synthesis*, pages 26–31, New York, NY, USA, 2002. ACM Press.

[52] Martins Gills. Survey of traceability models in it project. In *ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings*,.

[53] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First Conference on Requirements Engineering (ICRE'94)*.

[54] Arnaud Grasset. *Synthèse des interfaces de communication dans la conception des systèmes monopuces : de la spécification à la génération automatique*. PhD thesis, Institut National Polytechnique de Grenoble (INPG), laboratoire TIMA, Janvier 2005.

[55] Arnaud Grasset, Frédéric Rousseau, and Ahmed A. Jerraya. Automatique generation of component wrappers by composition of hardware library elements starting from communication service specification.

[56] Arnaud Grasset, Frédéric Rousseau, and Ahmed A. Jerraya. Génération des interfaces de communication pour les systèmes multiprocesseurs monopuces : de la spécification des services de communication vers l'implémentation rtl.

[57] Jack Greenfield and Keith Short. *Software Factories - Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley Publishing Inc., 2004.

[58] Object Management Group. Omg document - formal/02-06-07, corba 3.0 - idl syntax and semantics. `http://www.omg.org/cgi-bin/apps/doc?formal/02-06-07.pdf`, july 2002.

[59] Object Management Group. Using emf, ibm corp. `http://www.eclipse.org`, may 2003.

[60] Object Management Group. Omg document - formal/05-07-04,uml superstructure specification, v2.0. `http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf`, august 2005.

[61] ProMARTE Working Group. Marte initial submission, omg document realtime/05-11-01. `http://www.omg.org/cgi-bin/doc?realtime/2005-11-01`, 2005.

[62] David Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, June 1987.

[63] Amir Hekmatpour and Kenneth Goodnow. Designcon east 2005 : Standards-compliant ip design advantages, problems, and future directions. `http://www.iec.org/events/2005/designcon_east/pdf/1-tp1_hekmatpour.pdf`.

[64] D. Hommais, F. Pétrot, and I. Augé. A pratical toolbox for system level communication synthesis. *RSP*, 2001.

[65] Open SystemC Initiative. The systemc library. `http://www.systemc.org`.

[66] Bézivin J. and Gerbé O. Towards a precise definition of the omg/mda framework. *ASE'01*, nov 2001.

[67] Andrew Jackson, Pablo Sanchez, Lidia Fuentes, and Siobhan Clarke. Towards traceability between ao architecture and design. In *Proceedings of the Early Aspect 2006 Workshop*, Bonn, Germany, March 2006.

[68] Jean-Marc Jezequel. A mda approach to model and implement transformations [online]. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. <http ://drops.dagstuhl.de/opus/volltexte/2005/20> [date of citation : 2005-02-01].

[69] Frédéric Jouault. Loosely coupled traceability for atl. In *ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings*.

[70] Java Community Process JSR 040. Java metadata interface(jmi) specification. `http://www.jcp.org/`, 2002.

[71] Czarnecki K. and Helsen S. Classification of model transformation approaches. 2003.

[72] Czarnecki K. and Helsen S. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 2006.

[73] Hubert Kadima. *Conception orienté objet guidée par les modèles*. Dunod, 2005.

[74] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74 : Proceedings of the IFIP Congres 74*, pages 471–475. IFIP, North-Holland Publishing Co., august 1974.

[75] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77 : Proceedings of the IFIP Congres 74*, pages 993–998. IFIP, North-Holland Publishing Co., 1977.

[76] Mohamad Kassab and Ola Ormandjieva. Towards an aspect-oriented software development model with traceability mechanism. In *Proceedings of the Early Aspect 2006 Workshop*, Bonn, Germany, March 2006.

[77] Beck Kent. *Extreme Programming Explained : embrace Change*. Boston : Addison-Wesley, 2000.

[78] Haroon Saleem Khan. Achieving reusability through interoperability. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems(ECBS'06)*, 2006.

[79] Anneke Kleppe, Jos Warmer, and Win Bast. *MDA EXPLAINED, The Model Driven Architecture : Practice and Promise*. Addison-Wesley, 2003.

[80] Ivan Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, 2005.

[81] Brownsword L. and al. current perspectives on interoperability. Technical report, Carnegie Mellon Software Engineering Institute, 2004.

[82] Tratt L. The converge programming language. Technical report, King's College London, 2005.

[83] Grace A. Lewis and Lutz Wrage. approaches to constructive interoperability. Technical report, Carnegie Mellon Software Engineering Institute, 2004.

[84] Angelina E. Limon and Juan Garbajosa. The need for a unifying traceability scheme. In *ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings*,.

[85] Janet Vander Linden, Robin Laney, and Pete Thomas. Traceability and aosd : From requirements to aspects. In *Proceedings of the Early Aspect 2006 Workshop*, Bonn, Germany, March 2006.

[86] Lindvall M. and Sandahl K. *Software - Practice and Experience*, chapter Practical Implications of Traceability. 1996.

[87] Stevens M. Service-oriented architecture introduction. `http://www.developer.com/services/article.php/1010451`, 2004.

[88] F. Marschall and P. Braun. Model transformations for mda with botl.

[89] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. 04101 discussion – a taxonomy of model transformations [online]. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. <http ://drops.dagstuhl.de/opus/volltexte/2005/11> [date of citation : 2005-02-01].

[90] G. Nicolescu, S. Yoo, and A. Jerraya. Mixed-level cosimulation for fine gradual refinement of communication in soc design. In *DATE'2001*, 2001.

[91] Ed. Object Management Group. Uml extension profile for soc rfc, 2005, omg document realtime/05-03-01. `http://www.omg.org/cgi-bin/doc?realtime/2005-03-01`, 2005.

[92] Object Management Group, Inc. Xml metadata interchange (xmi), version 1.2. `http://www.omg.org/technology/documents/formal/xmi.html`.

[93] Object Management Group, Inc. Meta object facility (mof) specification, version 1.4, 2002.

[94] Object Management Group, Inc. The model driven architecture, September 2003. `http://www.omg.org/mda/`.

[95] Interactives Objects and Project Technology. Mof query/views/transformations, revised submission. omg document : ad/03-08-11, ad/03-08-12, ad/03-08-13. `http://frontline.compuware.com/javacentral/tools/`.

[96] OMG. Mof qvt final adopted specification. OMG Adopted Specification ptc/05-11-01.

[97] OMG. Mda guide. `http://www.omg.org/docs/omg/03-06-01.pdf`, June 2003.

[98] OMG. Mof 2.0 query / views / transformations rfp, 2003. OMG paper.

[99] OMG. Object constraint language 2.0 specification. `http://www.omg.org/docs/ptc/05-06-06.pdf`, June 2005.

[100] Letelier P. A framework for requirements traceability in uml-based projects. In *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering,Sepetember 2002*.

[101] R. Passerone, J. A. Rowson, and A. S.-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. *DAC*, 1998.

[102] R. Passerone and A. S.-Vincentelli. Interface based design. *DAC*, 1997.

[103] Netbeans Project. Metadata repository (mdr). `http://mdr.netbeans.org/`.

[104] QVT-Partners. Mof query/views/transformations, revised submission. omg document : ad/2003-08-08.

[105] Rochit Rajsuman. *System-on-a-Chip Design and Test*. Advantest America RD Center, Inc., 2000.

[106] E. Riccobene, P. Scandurra, A. Rosti2, and S. Bocchio. A uml 2.0 profile for systemc. Technical report, ST Microelectronics, 2004.

[107] E. Riccobene, P. Scandurra, A. Rosti2, and S. Bocchio. A soc design methodology involving a uml 2.0 profile for systemc. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)*, 2005.

[108] Pamela A. Sanders and Jr. John A. Hamilton. Chapter 8 - a process for interoperability. In *Joint Command and Control Interoperability : Cutting the Gordian Knot*, 2003.

[109] Douglas C. Schmidt. Model driven engineering. *IEEE Computer Society*, feb 2006.

[110] TCS (Tata Consulting Services). Modelmorf qvt/relation implementation. `http://www.tcs-trddc.com/modelmorf/index.html/`.

[111] the OMG staff MDA Soley R. Model driven architecture, November 2000. disponible à l'addresse : `http://www.omg.org/mda/presentations.htm`.

[112] Yves Sorel. massively parallel computing systems with real time constraints - the "algorithm architecture adequation" methodoly. In *Proceedings of the 1st International Conference on Massively Parallel Computing Systems*, pages 44–54, Los Alamitos, CA, USA, May 1994. IEEE Computer Society Press.

[113] Prawee Sriplakich. Techniques des transformations de modèles basées sur la méta-modélisation. Master's thesis, Université Pierre et Marie Curie, 2003.

[114] Thomas Stahl and Markus Völter. *Model-Driven software Development*. Heidelberg : Wiley, 2006.

[115] Marta S. Tabares and Ana Moreina. Towards a meta aspect for traceability. In *Proceedings of the Early Aspect 2006 Workshop*, Bonn, Germany, March 2006.

[116] Hubert Tardieu, Arnold Rochfeld, and René Colletti. *La Méthode Merise : Principes et outils*. Editions d'Organisation, 1991.

[117] Naveed Ahsan Tariq and Naeem Akhter. *Comparison of Model Driven Architecture (MDA) based tools*. PhD thesis, Karolinska University Hospital, 2005.

[118] SysML Merge Team. Sysml specification v. 1.0 (draft), omg document ad/06-03-01. `http://www.omg.org/cgi-bin/doc?ad/06-06-01`, 2006.

[119] T.Gardner, C.Grifin, A. Koehler, and R.Hauser. A review of omg mof 2.0 query / views / transformations submissions and recommendations towards the final standard, 2003. Review of QVT proposals.

[120] Unisys. Java metadata interface(jmi) reference implementation (ri). `http://ecommunity.unisys.com`.

[121] Bert Vanhooff and Yolande Berbers. Supporting modular transformation units with precise transformation traceability metadata. In *ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings*.

[122] S. Vercauteren, B. Lin, and H. De Man. Constructing application-specific heterogeneous embedded architectures from custom hw/sw applications. *DAC*, 1996.

Annexe A

Code SystemC et modèle du wrapper CABA-PVT

A.1 Code SystemC du wrapper

```
/*************************************************/
// Sujet   : Wrapper SystemC CABA/TLM
// Auteurs : Rabie Ben-Atitallah &  Lossan BONDE
// Date    : Juillet 2006
/*************************************************/
#include <systemc.h>
#include <stdio.h>
#include "shared/soclib_vci_interfaces.h"
#include "shared/soclib_segment_table.h"
#include "bus_types.h"
#include "basic_timed_initiator_port.h"

using basic_protocol::basic_timed_initiator_port;

////////////////////////////////////////
// structure definition
////////////////////////////////////////

struct GASPARD_VCI_WRAPPER : sc_module {

// IO PORTS
basic_timed_initiator_port<ADDRESS_TYPE,DATA_TYPE> initiator_port;
void run();
sc_in<bool>  CLK;

ADVANCED_VCI_TARGET<32,4,0>  VCI;
```

```
const char  *NAME;

ADDRESS_TYPE    adr;
DATA_TYPE     wdata;
int     rdata;
int     be;
int     cmd;
int             t;
DATA_TYPE       d;
int             eop;
int id;
////////////////////////////////////////////////
// constructor
////////////////////////////////////////////////

SC_HAS_PROCESS(GASPARD_VCI_WRAPPER);
GASPARD_VCI_WRAPPER(sc_module_name   insname)    // segment table pointer
: sc_module( insname ) ,
initiator_port("iport")
{
 SC_METHOD (transition);
  sensitive_neg << CLK;
  NAME = (const char*) insname;
 printf("Successful Instanciation of GASPARD_VCI_WRAPPER : %s\n",NAME);
}; // end constructor
void transition()
{
if (VCI.VAL == true) {
 if (VCI.ACK==true){
adr = ((int)VCI.ADDRESS.read());
wdata   = (int)VCI.WDATA.read();
be      = (int)VCI.BE.read();
cmd     = (int)VCI.RD.read();
                eop     = (int) VCI.EOP.read();
                id= (int)VCI.SCRID.read();
  if ((cmd & 0x1) == 0x1) {
      initiator_port.read( adr , d , t );
         VCI.ACK = 1
         VCI.RSPVAL = 1
         VCI.RERROR = 0
         VCI.RPKTID = 0
         VCI.RTRDID = 0
       VCI.RDATA = d;
       VCI.RSCRID = id;
        if (eop==1)
        {
         VCI.REOP = 1
```

```
        }
        else {
         VCI.REOP = 0
        }
      }else{
      initiator_port.write( adr , wdata , t );
         VCI.ACK = 1
         VCI.RDATA = 0
         VCI.RSPVAL = 1
         VCI.RERROR = 0
         VCI.RPKTID = 0
         VCI.RTRDID = 0
       VCI.RSCRID = id;
if (eop==1)
        {
         VCI.REOP = 1
        }
        else {
         VCI.REOP = 0
       }
}
     else{
         VCI.ACK = 1
     }
  } else{
         if (VCI.RSPACK == true) {
         VCI.ACK = 1
         VCI.RSPVAL = 0
         VCI.RDATA = 0
         VCI.RERROR = 0
         VCI.RSCRID = 0
         VCI.RPKTID = 0
         VCI.RTRDID = 0
         VCI.REOP = 0
}
        }
};
};
```

A.2 Modèle du wrapper

```
<?xml version="1.0" encoding="UTF-8"?>
<wrapperMM:Wrapper xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:wrapperMM="wrapperMM" adrLen="32" byteEnable="4">
  <automate currentState="//@automate/@states.0" initial="//@automate/@states.0">
```

```
    <states name="Idle"/>
    <states name="Valid"/>
    <states name="Synchro"/>
    <trans source="//@automate/@states.0" target="//@automate/@states.0">
      <conditions signal="VAL" isPrecondition="true"/>
    </trans>
    <trans source="//@automate/@states.0" target="//@automate/@states.1">
      <conditions signal="VAL" value="1" isPrecondition="true"/>
      <conditions signal="ACK" value="1"/>
    </trans>
    <trans source="//@automate/@states.1" target="//@automate/@states.2">
      <conditions signal="ACK" value="1"/>
      <conditions signal="RDATA"/>
      <conditions signal="RSPVAL" value="1"/>
      <conditions signal="RERROR"/>
      <conditions signal="RPKTID"/>
      <conditions signal="RTRDID"/>
    </trans>
    <trans source="//@automate/@states.1" target="//@automate/@states.2">
      <conditions signal="VAL" value="1" isPrecondition="true"/>
      <conditions signal="ACK" value="1" isPrecondition="true"/>
      <conditions signal="RD" value="1" isPrecondition="true"/>
      <conditions signal="ACK" value="1"/>
      <conditions signal="RSPVAL" value="1"/>
      <conditions signal="RERROR"/>
      <conditions signal="RPKTID"/>
      <conditions signal="RTRDID"/>
    </trans>
    <trans source="//@automate/@states.2" target="//@automate/@states.1">
      <conditions signal="REOP"/>
    </trans>
    <trans source="//@automate/@states.2" target="//@automate/@states.0">
      <conditions signal="REOP" value="1"/>
    </trans>
    <trans source="//@automate/@states.2" target="//@automate/@states.0">
      <conditions signal="ACK" value="1"/>
      <conditions signal="RSPVAL"/>
      <conditions signal="RDATA"/>
      <conditions signal="RERROR"/>
      <conditions signal="RSCRID"/>
      <conditions signal="RPKTID"/>
      <conditions signal="RTRDID"/>
      <conditions signal="REOP"/>
    </trans>
  </automate>
  <ports portName="VAL"/>
  <ports portName="REOP"/>
```

```
  <ports portName="RSPVAL"/>
  <ports portName="RD"/>
  <ports portName="ADDRESS"/>
  <ports portName="BE"/>
  <ports portName="WDATA"/>
  <ports portName="EOP"/>
  <ports portName="ACK"/>
  <ports portName="RDATA"/>
  <ports portName="RERROR"/>
  <ports portName="SCRID"/>
  <ports portName="RSCRID"/>
  <ports portName="RPKTID"/>
  <ports portName="RTRDID"/>
</wrapperMM:Wrapper>
```