Numéro d'ordre : 3[illegible]4

Thèse présentée pour obtenir le titre de Docteur de l'Université des Sciences et Technologies de Lille, spécialité Informatique

UNE SIMULATION FONCTIONNELLE D'UN SYSTÈME MONOPUCE DÉDIÉ AU TRAITEMENT DU SIGNAL INTENSIF

Une approche dirigée par les modèles

par

Mickaël SAMYN

Université des sciences et Technologies de Lille
LIFL - UPRESA 8022 - Bât. M3 - UFR IEEA
59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 28 77 85 41 - Fax. : +33 (0)3 28 77 85 37
Email : mickael.samyn@lifl.fr

Soutenue le 14 décembre 2005

Devant la commision d'examen formée de
Pierre BOULET (Président du jury)
Gilles GONCALVES (Rapporteur)
Smaïl NIAR (Rapporteur)
Jean-Luc DEKEYSER (Directeur de thèse)
Samy MEFTALI (Co-Directeur de thèse)

Pour Karine et Frédéric

Remerciements

Je tiens à remercier toutes les personnes qui ont rendu cette thèse possible par leur soutien, leur aide et leur contributions.

J'aimerais remercier, en particulier :

Pierre BOULET pour son accueil lors de mon arrivé dans l'équipe et pour avoir accepter de présider ce jury.
Gilles GONCALVES et Smaîl NIAR pour avoir pris le temps de juger mon travail de thèse.
Jean-Luc DEKEYSER pour m'avoir accueilli et encadré durant ces trois ans de thèse.
Samy MEFTALI pour ses conseils lors de mes différents développements ainsi que pour avoir pris le temps de corriger mes différents documents.

J'aimerais également remercier Philippe MARQUET, Cédric DUMOULIN et Jean-Luc LEVAIRE pour leur accueil au sein de l'équipe West.

N'oublions pas les anciens thésards de WEST, Chadi ALJUNDI et AbdelKader AMAR, toujours prêt à répondes à nos questions et à nous aider dans toutes nos démarches.

Une pensée également à Arnaud CUCCURU et Philippe DUMONT, qui ont ou vont soutenir cette année.

Je souhaite bon courage à tous les autres thèsard :

Rabie BEN ATITALLAH
Lossan BONDÉ
Ouassila LABBANI
Sébastien LE BEUX
Ashish MEENA
Eric PIEL
Joël VENNIN

bonne fin de thèse

Pour terminer avec ces remerciements, je voudrais remercier Karine, ma femme, et Frédéric, mon fils, pour m'avoir soutenu et supporté au long de ces trois années de thèse et particulièrement cette dernière année.

Merci également à Jean-Jacques et Astrid, mes beau-parents, et Manuel et Maria, mes grand-parents, qui m'ont soutenu durant mes études.

Table des matières

Chapitre 1

Introduction

1.1 Contexte

Les nouveaux systèmes monopuces deviennent de plus en plus complexe, et ceci est dû à la demande grandissante de puissance de calcul et de fonctionnalités. Par exemple, dans le monde de la téléphonie mobile, en quelques années les appareils téléphoniques portables sont passés du simple combiné permettant d'emettre et de recevoir des appels téléphoniques à des mini-ordinateurs intégrant des caméras/appareils photos, des agendas, des navigateurs internet, des programmes de lecture de courrier électronique, des lecteurs multimédias, et même pour les plus récents des recepteurs de télévision. Ces nouveaux applicatifs ont entraîné une augmentation de la compléxité des systèmes développés. D'architecture simple, monoprocesseur, les systèmes actuels sont basés sur des architectures multi processeur, multi mémoire connectés grâce à des moyens d'interconnexions très complexe de type NoC (Network on Chip). Parallelement à cette montée en puissance des applicatifs et du matériel, la taille des systèmes a diminué et on a également vu une augmentation de l'autonomie de ceux-ci. La diminution de la taille des gravures des nouvelles puces libére de plus en plus d'espace sur les puces qu'il est nécessaire de combler. Les méthodes de conception actuelles ne permettent plus de suivre l'évolution et ainsi de profiter de tout l'espace disponible.

Le tableau 1.1 montre quelques exemples de systèmes monopuce très répandus dans le commerce de nos jours. Ces systèmes (jeux, processeur, réseau, ...) intègrent plusieurs processeurs et sont d'une grande complexité (supérieure à 1 millions de portes). Ils disposent aussi de plusieurs Mbits de mémoire embarquée [Mef02].

	Processeurs	Mémoire embarquée	Logique spécifique	Exemple typique
Terminal XDSL	1 MCU 1 DSP	> Mbits	> M portes VDSL (ST)	STEP 1
Multimédia	1 MCU plusieurs DSP	> > Mbits	< M portes (Philips)	TRIMEDIA
Processeurs réseau	plusieurs MCU plusieurs DSP	> > Mbits	> M portes INTEL	IXPIZDE
Processeurs de jeux	plusieurs MCU plusieurs DSP	> > Mbits	> > M portes (Sony)	PlayStation

TAB. 1.1 – Exemples de systèmes monopuce modernes

1.2 Objectifs

L'objectif du projet de l'équipe est de montrer qu'il est possible d'utiliser des méthodes issues du domaine du génie logiciel pour concevoir une méthodologie de conception de système monopuce. En utilisant la méthodologie « MDA[1] », il est possible de concevoir des systèmes en garantissant à chaque transformation la validité du nouveau modèle. Mon travail permet, dans un premier temps, de pouvoir générer du code permettant de valider le modèle par simulation. Dans un deuxièmre temps, il permet d'extraire des données permettant d'évaluer certains critères de performances permettant de savoir si le système répond au cahier des charges, et ce le plus tôt possible dans la conception. En effet, toute remise en question des choix provoque une augmentation du temps de mise sur le marché ainsi que le coût de conception du système. Si à un très haut niveau, il est possible d'écarter des architectures matériels ou des placements, on diminue d'autant le risque de retour en arrière, et donc les coûts liés à ces développement inutiles.

Le flot de conception Gaspard, au début de cette thèse se résumait à une description de l'application, de l'architecture et de l'association dans les meta modèles correspondants. Malheureusement ceci ne permet pas de profiter pleinement de l'apport de l'approche MDA dans le contexte de la conception de systèmes monopuces. En effet, l'aboutissement de l'association n'est pas un modèle simulable (exécutable). Donc le concepteur ne pourra ni le valider du point de vue fonctionnalité, ni estimer ses performances. Ainsi, la contribution principale de ce travail et de répondre à ces besoins en complètant la chaine de conception Gaspard.

1.3 Plan de la thèse

Dans le premier chapitre, je vais présenter les prérequis nécessaires à la compréhension de ce document. Tout d'abord, je vais présenter en quoi consiste la méthodologie MDA, puis

[1]Model Driven Architecture

quelques éléments du langage Array-OL et enfin je vais présenter les différents principes de la bibliothéque SystemC™. Pour continuer, je vais parler de trois projets en relation avec notre vision de la conception de systèmes monopuces. Le premier projet est le projet MILAN. MILAN est un approche académique développé par le département EE-Systems de l'université de Californie du Sud[2] permettant la définition de systèmes sous forme de schémas hiérarchiques et ensuite la simulation et l'analyse de performance. Le deuxième projet est une approche développé par Fujitsu. Le troisième projet est une RFC d'un groupement industriel japonais. La RFC, qui a pour nom « UML for SoC », présente un profil UML permettant la modélisation de systèmes embarqués grâce à UML. Dans le chapitre suivant, je présente le projet de notre équipe : GASPARD. GASPARD est basé sur une méthodologie en « Y » permettant la conception de systèmes monopuces. D'une application et une architecture modélisées en UML, on effectue le placement et l'ordonnancement de l'application sur l'architecture. De ce placement, on génére un modèle permettant la génération, la simulation et l'analyse de performance, le sujet de ma thèse. Le chapitre suivant traite du métamodèle permettant la génération du code nécessaire à la simulation et à l'analyse de performances. Ce chapitre présente les différents concepts du métamodéle aussi bien pour l'exécution (*Processor*, ...) que pour l'analyse des performances (*RunningPerformance*, ...). Ensuite, le chapitre suivant présente une implémentation possible des différents modules permettant la simulation et l'analyse de performances. Dans une première partie, je présente l'interface de communication des différents modules générés de manière automatique, ensuite je montre comment analyser et extraire des données de la simulation. Le dernier chapitre présente un exemple permettant de mettre en évidence les différentes informations que l'on peut extraire d'une simulation. La conclusion et les perspectives de ce travail sont données à la fin du document.

[2]USC University of Southern California

Chapitre 2

Prérequis

Je présente, dans ce chapitre, différents travaux relatifs à la simulation et à l'analyse de performance de systèmes embarqués. Ensuite, je présente les différents prérequis nécessaires à la compréhension de ce document. Je commencerai par un rappel de ce qu'est la méthodologie de l'architecture dirigée par les modèles. Puis, j'enchaînerai avec une présentation de ModTransf, un moteur de transformation développé dans l'équipe. Ensuite, je m'attarderai un peu plus sur SystemC™, une librairie de cosimulation. Enfin, je présenterai brièvement le langage Array-OL, un langage de spécification d'applications de traitement du signal.

2.1 Les aspects génie logiciel

2.1.1 Architecture dirigée par les modèles (MDA)

Principes

MDA fournit un ensemble de lignes pour structurer les spécifications exprimées grâce à des modèles ainsi que les corrélations entre ces modèles. L'initiative MDA et les standards qui la supportent permettent de spécifier, dans un même modèle, les fonctionnalités de l'application ainsi que le comportement que l'on doit réaliser sur de multiples plateformes. MDA autorise différentes applications à être intégrées en désignant explicitement leurs modèles. Cela simplifie l'intégration et l'interopérabilité et supporte l'évolution des systèmes

(choix d'implémentation) lorsque les technologies changent. Les trois buts principaux de MDA sont la portabilité, l'interopérabilité et la réutilisation.

La portabilité d'un système dépend des sous-systèmes qui le composent. L'ensemble des sous-systèmes d'un système particulier est souvent appelé la *plateforme*. La portabilité, ainsi que la réutilisation, d'un tel système sont assurés si tous les sous-systèmes utilisent des interfaces standardisées et des modèles d'utilisation.

MDA fournit un modèle comprenant un sous-système portable capable d'utiliser une des multiples implémentations spécifiques de la plateforme. Ce modèle est utilisable, à plusieurs reprises, dans la spécification des systèmes. Les cinq concepts importants liés à ce modèle sont :

1. *Modèle* - Un modèle est la représentation d'une partie d'une fonction, de la structure et/ou du comportement d'une application ou du système. Une représentation est dite formelle quand elle est basée sur un langage qui possède une syntaxe, une sémantique et éventuellement des règles d'analyse, d'inférence ou des preuves de construction. La syntaxe peut être graphique ou textuelle. La sémantique peut être définie plus ou moins formellement, en terme de description d'observations (échange de message, états des objets et changement d'états, ...), ou par la traduction des constructions d'un langage de plus haut niveau en d'autres constructions possédant une signification bien définie. Les règles optionnelles d'inférence définie quelles propriétés non spécifiées peuvent être déduites du modèle. Dans la méthodologie MDA, une représentation non formelle, dans le sens défini précédemment, n'est pas un modèle.
2. *Plateforme* - Un ensemble de sous-systèmes/technologies fournissant un ensemble cohérent de fonctionnalité à travers des interfaces et des modèles d'utilisation spécifiés. Tout sous-systèmes dépendant de cette plateforme peut utiliser ces fonctionnalités sans se soucier de l'implémentation.
3. *Modèle indépendant de la plateforme (PIM)* - Un sous-système modélisé ne contenant aucune information spécifique sur la plateforme, ou de la technologie, utilisé pour le réaliser.
4. *Modèle spécifique à la plateforme (PSM)* - Un sous-système modélisé incluant des informations sur les technologies spécifiques utilisées pour sa réalisation sur une plateforme spécifique et, par conséquent, contenir éventuellement des éléments qui sont spécifiques à la plateforme.
5. *Placement* - Spécification d'un mécanisme de transformation des éléments d'un modèle respectant un métamodèle particulier en éléments d'un autre modèle respectant un autre (éventuellement le même) métamodèle. Un placement peut être exprimé comme des associations, des contraintes, des règles, des patrons paramétrés pouvant être affectés durant le placement, ou encore d'autres formes non déterminées pour le moment.

Par exemple, dans le cas de CORBA, la plateforme est spécifiée grâce à un ensemble d'interfaces et un modèle d'utilisation qui constitue le « CORBA Core Specification ». La plateforme CORBA est indépendante du système d'exploitation et des langages de

programmation. Les spécifications de l'OMG pour « Trading Object Service », constituées par des spécification d'interfaces dans le langage de définition d'interfaces de l'OMG (OMG IDL), peut être considéré comme un PIM, du point de vue de CORBA, car indépendant du système d'exploitation et des langages. Lorsque la description des interfaces (IDL) est transformée, par exemple, en C++, le résultat obtenu peut être considéré comme un PSM où la plateforme est le langage C++ ainsi que l'implémentation de l'ORB en C++.

Moteurs de transformations

Pour réaliser le placement d'un PIM sur un PSM, il existe des outils appelés moteurs de transformation. Un moteur de transformation est un outil prenant en entrée un ou plusieurs modèles (dans le sens MDA du terme) et produit un ou plusieurs modèles en sortie. Pour cela, il existe deux approches, une première consistant au développement d'outils spécifiques, dépendant des modèles d'entrée et de sortie. Une seconde approche serait de développer un outil générique pour effectuer le travail. Il existe déjà un certain nombre de moteurs de transformation, certains commerciaux et d'autres gratuits (voir même open source).

Parmi ces nombreux moteurs de transformation, On peut citer, à titre d'exemple, ModFact. ModFact est un moteur de transformation libre développé par le LIP6[1]. Le but de ModFact est de fournir un environnement pour construire des applications en suivant les concepts de l'Architecture Dirigée par le Modèles. ModFact est, en fait, un ensemble d'outils permettant de générer du MOF[2] grâce au projet repository. Le projet QVT Engine permet d'effectuer des transformations sur les modèles et enfin le Model bus permettant l'interopérabilité des outils précédemment cités. Le projet Repository permet de stocker les différents modèles sous forme MOF. Il permet de générer le MOF en tant qu'objets CORBA ou d'objets JMI, en fournissant une API permettant de manipuler les modèles générés. Le projet QVT Engine permet d'appliquer des transformations, définit par des règles de transformations, sur les modèles en entrée afin de fournir un nouveau modèle en sortie (voir figure 2.1). Ces règles de transformations sont définies en utilisant le langage TRL (Transformation rules language). Les entrées et sorties du moteur de transformation sont effectuées grâce à des fichiers au format XMI. Pour cela, il faut fournir au moteur les métamodèles d'entrée, de sortie et celui des régles. Ensuite, on lui fournit le modèle d'entrée ainsi que les régles de transformation et le moteur produit le modèle de sortie, conforme au métamodèle fournit.

Un autre moteur de transformation, modTransf, développé dans notre laboratoire est également disponible. ModTransf est un moteur de transformation et un générateur de code suivant les directives de la méthodologies MDA. ModTransf prend en entrée un métamodèle et un modèle, ainsi que le métamodèle du modèle de sortie et il fournit un modèle en sortie, respectant le métamodèle fournit. Ces transformations sont effectuées en appliquant des règles fournies dans un fichier de règles. Il est possible de faire de nombreuses actions

[1]Laboratoire d'Informatique de Paris 6

[2]Metamodel Object Facilities

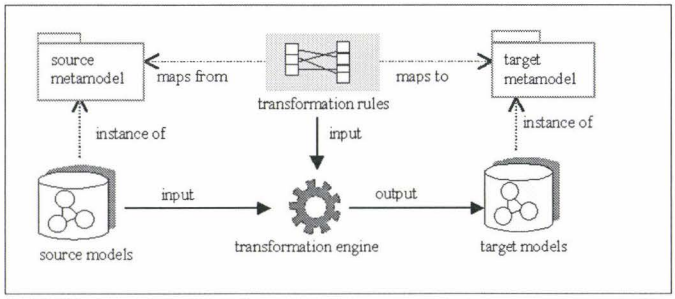


FIG. 2.1 – *Principe de fonctionnement de QVT Engine*

comme la génération d'un fichier respectant un patron, créer des objets dans le modèle de sortie en fonction des concepts du modèle d'entrée (voir section 5.1.1).

2.1.2 SystemC™

SystemC™ [ABB+03, SVD+02] est une bibliothèque C++ permettant la cosimulation logicielle/matérielle à différents niveaux d'abstraction.

Les niveaux d'abstraction

Traditionnellement, les niveaux d'abstraction en SystemC™ sont regroupés en quatre catégories

- UnTimed Functional (UTF)
- Timed Functional (TF)
- Cycle Accurate Byte Accurate (CABA)
- Register Transfer Level (RTL)

Le niveau fonctionnel (UTF) permet de définir un système à un très haut niveau d'abstraction, permettant la communication des objets par des appels de méthode. Le niveau fonctionnel temporise (TF), permet de raffiner le système précédemment défini, en ajoutant la notion de temps d'exécution. Le niveau de précision au cycle et bit (CABA) permet de raffiner la définition du système pour s'approcher au plus près du fonctionnement du système final. Et enfin, le niveau transfert de registre (RTL) permet de définir le fonctionnement réel du système et permet également la synthèse de celui-ci (sous certaines conditions).

Actuellement, ces niveaux d'abstraction sont remis en cause [], et on ne distingue plus que trois niveaux d'abstraction : le niveau UTF, le niveau RTL et tout ce qui se trouve entre les deux est appelé niveau transactionnel (Transaction Level ou TL). Le niveau transactionnel est lui même subdivisé en quatre niveaux d'abstraction.

Les concepts de base

SystemC™ possède trois concepts principaux, héritant de la classe *sc_object*. Ces concepts sont les modules (*sc_module*), les canaux de communication (*sc_channel*) et les ports (*sc_port*). En schématisant, un système décrit en SystemC™ est un ensemble de modules, communicant par différents canaux de communication via des ports. Les modules possèdent un ou plusieurs processus, permettant la définition de son comportement.

Les concepts dérivés

De manière générale, il est plus facile d'utiliser les classes dérivées de ports et de canaux. En effet, SystemC™ fournit un certain nombre de ports et de canaux prédéfinis, possédant des méthodes spécifiques pour faciliter leur utilisation. Pour les ports, je peux citer

- **sc_in** (port d'entrée),
- **sc_out** (port de sortie),
- **sc_inout** (port bidirectionnel),
- **sc_in_clk** (port d'entrée d'horloge),
- ...

et pour les canaux

- **sc_signal**,
- **sc_buffer**,
- **sc_fifo**,
- **sc_mutex**,
- **sc_semaphore**,
- **sc_clock**,
- ...

Il faut noter que tout les classes de ports et de canaux sont définis à l'aide de patron (« template C++ ») permettant de définir, entre autres choses, le type de donnée que l'on manipule.

Les processus

Il existe trois types de processus en SystemC™ : les méthodes (**SC_METHOD**), les "threads" (**SC_THREAD**) et les "clocked thread" (**SC_CTHREAD**). Les méthodes sont les processus les plus rapides existant en SystemC™, utilisées pour décrire les systèmes au niveau fonctionnel. Les threads sont utilisés pour décrire les systèmes à tous les autres niveaux d'abstraction. Un « thread » est une fonction C/C++, contenant une boucle, permettant de définir le comportement associé à ce module. Ces fonctions peuvent contenir des wait permettant de suspendre l'exécution jusqu'à ce que le module reçoive un signal. Les signaux permettant la réactivation du thread sont stockés dans la liste de sensitivité (sensitive). Les « clocked threads » sont des « threads » qui possède une sensitivité sur un signal d'horloge (sc_clock).

Les types de donnée

SystemC™ fournit un certain nombre de type de données [ABB+03], principalement des types de bas niveau, pour la conception de système. Par type de bas niveaux, j'entends les types de manipulation de bit, comme le montre la table 2.1.

Nom	Désignation
sc_bit	bit simple deux valeurs
sc_logic	bit simple quatre valeurs
sc_int	entier signé de 1 à 64 bits
sc_uint	entier non signé de 1 à 64 bits
sc_bigint	entier signé de taille arbitraire
sc_biguint	entier non signé de taille arbitraire
sc_bv	vecteur de bit (sc_bit)
sc_lv	vecteur de bit (sc_logic)
sc_fixed	nombre signé à virgule fixe paramétré
sc_ufixed	nombre non signé à virgule fixe paramétré
sc_fix	nombre signé à virgule fixe
sc_ufix	nombre non signé à virgule fixe

TAB. 2.1 – Type de donnée en SystemC™

En plus de ces types, il est possible d'utiliser tous les types de bases définit par le langage C++ (int, bool, double, ...). Ces types sont gérés sans aucun problème par le moteur SystemC™. Et enfin il est possible de définir des types de donnée personnalisée. Dans ce cas, il est nécessaire de définir deux méthodes [ABB+03]. Ces méthodes sont :

- comparaison (inline bool operator == (const monType& maVariable) const)
- trace (extern void sc_trace(sc_trace_file *tf, const monType& maVariable, const sc_string& name))

Une fois ces deux méthodes définies, le simulateur est capable de manipuler n'importe quel type de données définie par l'utilisateur.

Le simulateur SystemC™

Nous allons maintenant voir comment fonctionne le simulateur SystemC™ pour permettre la simulation d'un système.

1. Tous les signaux d'horloge qui doivent changer de valeur mettent à jour cette valeur
2. Tous les SC_METHOD/SC_THREAD dont les entrées ont été modifiées sont exécutés. Les méthodes sont exécutées entièrement tandis que les threads sont exécutés jusqu'au prochain wait. Ces différentes fonctions ne sont pas exécutées dans un ordre défini
3. Tous les SC_CTHREAD ont leurs sorties mises à jour et sont placés dans une file d'attente. Les sorties des SC_METHOD/SC_THREAD sont également mis à jour

4. Les étapes 2 et 3 sont répétées jusqu'à ce qu'il n'y ait plus de modification des valeurs des signaux
5. Tous les SC_CTHREAD placés dans la file d'attente à l'étape 3 sont exécutés. Leurs sorties seront mises à jour au prochain front d'horloge et sont également sauvegardées en interne
6. L'horloge passe au prochain front et le simulateur reprend à l'étape 1

Les observations

Il y a plusieurs manières d'observer en SystemC™. La manière la plus simple est d'utiliser le système de trace fournit avec le langage. En effet, SystemC™ est capable de générer la trace des valeurs des signaux. Il est possible de générer ces traces dans plusieurs formats : le format « vcd », le format « wif » et le format « isdb ».

Une seconde manière est de développer des modules spécialisés pour extraire différentes informations. Il est possible, par exemple, d'ajouter des afficheurs dans l'application pour suivre l'évolution de l'exécution.

Enfin, pour ne pas surcharger l'application par l'ajout de modules qui ralentiront l'exécution de l'application, il est possible d'ajouter, directement dans l'application, des affichages écran ou encore écrire dans des fichiers. Cette méthode permet également de garder une trace d'exécution et ainsi de pouvoir corriger les différents problèmes liés au développement.

2.1.3 Array-OL

Le langage Array-OL constitue une des bases de notre approche pour la spécification d'applications de traitement de signal.

Array-OL [DG,BDL+] a été développé par Thomson Marconi Sonar (maintenant Thales Underwater Systems), dans le but de répondre aux besoins de spécification, de standardisation, et d'efficacité du traitement de signal multidimensionnel. Array-OL est basé sur l'expression de dépendances. Il propose un formalisme visuel dans lequel le traitement de signal apparaît sous la forme d'un graphe de tâches. Chaque tâche est exécutée sur des tableaux multidimensionnels. Les cibles de compilation de ce langage sont des réseaux de station de travail, essentiellement pour le paramétrage de ces applications, et des systèmes monopuce pour les systèmes embarqués.

Array-OL propose une approche à deux niveaux. Le premier niveau, qualifié de « global », définit un ordonnancement de tâches sous la forme de dépendances entre tâches et tableaux. Le second niveau, qualifié de « local », définit les actions élémentaires réalisées par les tâches sur les éléments de tableau.

Le modèle global

Le modèle global tire son formalisme du modèle « process network ». L'application est représentée sous la forme d'un graphe, ou un nœud représente une tâche, et un arc

une dépendance entre deux tâches. Chaque arc symbolise un tableau. Cependant, dans le modèle process network, les arcs du graphe symbolisent un flot continu d'éléments, et toutes les tâches s'exécutent en parallèle. Dans le modèle Array-OL, un arc symbolise un tableau unique (qui peut être de taille infinie), et chaque tâche est exécutée une seule fois. La tâche produit ses tableaux de sortie à partir de ses tableaux d'entrée. La spécification de la tâche ainsi que les détails sur l'exploitation des éléments du tableau ne sont pas visibles à ce niveau de spécification.

Les applications de traitement de signal sont organisées autour d'un flux de données régulier et potentiellement infini. Array-OL manipule ce flux par le biais de tableaux, ces tableaux pouvant par conséquent avoir une dimension infinie.

Certaines dimensions spatiales des tableaux utilisés dans le traitement de signal correspondent à des capteurs. Ces capteurs peuvent être disposés sous la forme de cercle. Par conséquent, les dimensions des tableaux dans Array-OL peuvent être toriques, pour matérialiser la disposition circulaire des capteurs.

Le modèle local

Le modèle local permet de modéliser dans le détail la spécification d'une tâche. Il définit la manière d'accéder aux données du tableau, et les traitements qui doivent être effectués sur ces données. L'exécution complète de la tâche est divisée en plusieurs unités de traitement plus petites, appelées « tâches élémentaires ». Une tâche élémentaire s'applique sur des parties du tableau, appelées « motifs ». Les motifs de sortie (motifs dans le tableau de sortie) sont produits en appliquant la tâche élémentaire sur les motifs extraits des tableaux d'entrée. Une tâche peut alors être vue comme un itérateur, dont chaque pas d'itération est indépendant.

Pavage et Ajustage

Les motifs sont des tableaux multidimensionnels. Des éléments équidistants dans un motif sont également équidistants dans un tableau. Un motif peut alors être défini par une origine dans le tableau et un ensemble de vecteurs (les vecteurs d'ajustage, avec un vecteur pour chaque dimension du motif).

Deux motifs de sortie équidistants sont produits par deux motifs d'entrée équidistants. Le pavage du tableau par les motifs est donné par un premier motif dans chaque tableau, et par un ensemble de vecteurs de pavage.

Tâches élémentaires et définition hiérarchique

Pour chaque itération de pavage, les motifs d'entrée sont extraits des tableaux d'entrée, et une tâche élémentaire est appliquée sur ces motifs pour produire les motifs de sortie. Ces motifs produits sont ensuite stockés dans les tableaux de sortie. Une librairie de tâches élémentaires est disponible sur différentes architectures, pour des tâches data parallèles génériques.

Une extension hiérarchique d'Array-OL permet au programmeur de définir ses propres tâches par assemblage d'autres tâches. Les motifs d'entrée et de sortie du premier niveau sont considérés comme des tableaux sur le sous niveau hiérarchique. Cette construction hiérarchique peut être appliquée autant de fois que nécessaire.

2.2 Travaux relatif

De nombreuses équipes, aussi bien académiques qu'industrielles, s'intéressent à la conception de système monopuce. Différents outils sont mis à la disposition des concepteurs de systèmes.

2.2.1 MILAN

Parmi ces travaux, je peux citer MILAN[3] [VK01, AAJ+01, MP02, LDN+01], développé par le département EE-Systems de l'université de Californie du Sud[4]. Il nous propose une approche basée sur les modèles, permettant de simuler et d'évaluer les performances des systèmes ainsi modélisés. MILAN repose sur trois métamodèles, le premier permettant de modéliser l'application, le deuxième servant à modéliser les ressources et le dernier permettant d'exprimer les contraintes du système. Tout cela étant intégré dans un environnement de modélisation générique appelé GME[5] fournissant l'interface graphique permettant de manipuler les différents schéma. Le métamodèle de ressources permet de définir les différents composants matériels et leurs interconnexions en utilisant des diagrammes de blocs hiérarchiques. Le métamodèle d'application est basé sur une représentation hiérarchique de flux de signaux, avec des extensions permettant, par exemple, de spécifier des alternatives d'implémentation ou la spécification explicite du composant. Cette extension permet de modéliser l'ensemble de l'espace d'exploration plutôt qu'un point particulier de cet espace. Pour définir cet espace, les besoins de l'application, les contraintes et toutes les autres spécifications sont définis en utilisant le langage OCL[6]. OCL est un langage permettant de définir les contraintes entre les différents objets du système. La méthodologie d'évaluation est basée sur des modèles de performances spécifiques à chaque composant proposé dans la bibliothèque. Ce modèle est basé sur un modèle haut niveau permettant l'évaluation de différents placements d'application sur l'architecture proposé. D'après les auteurs, MILAN devrait supporter plusieurs classes de simulateurs : un simulateur fonctionnel, principalement SystemC™ ou MATLAB, et devrait supporter des simulateurs bas niveau permettant l'évaluation de la consommation et des performances.

[3]Model based Integrated simuLAtioN
[4]USC University of Southern California
[5]Generic Modeling Environment
[6]Object Constraint Language

2.2.2 SLOOP

Une autre approche s'appelle SLOOP[7] [ZMS], développé par Fujitsu. Malheureusement, s'agissant d'une approche industrielle, on trouve très peu de document sur cette approche. SLOOP utilise quatre modèles pour la conception de système monopuce, et ce de manière incrémentale, avant l'implémentation logicielle/matérielle. Chacun de ces modèles détaille trois différents aspects du système ciblé : la fonctionnalité, la structure et le temps. Le premier modèle, appelé conceptuel, permet de décrire les résultats de l'analyse du cahier des charges du client[8] en utilisant des techniques de conceptions orientées objets. Cette analyse est similaire à celle effectuée pour la conception d'un logiciel. Cette analyse permet au concepteur d'extraire les différents aspects fonctionnels et non fonctionnels requis. Les aspects non fonctionnels sont, par exemple, la taille, la forme, la consommation, ou les performances.

Le deuxième modèle, le modèle fonctionnel, se focalise sur la structure fonctionnelle du système sans considération des éléments de l'architecture. Ce modèle permet d'exprimer le parallélisme au niveau des tâches ainsi que les différentes communications entres celles ci. En résumé, le modèle fonctionnel se compose de processus et de communication entre ces processus. Dans ce modèle, deux indicateurs permettent de mesurer la charge du modèle de manière statistique : la charge de calcul et la charge de communication.

Le troisième modèle permet de définir les ressources physiques de l'architecture. Ces ressources peuvent être de calcul ou de communication. Les ressources de calcul (DSP[9], ASIC[10], ...) sont utilisés pour implémenter les processus du modèle fonctionnel. Les ressources de communication (bus, mémoires, ...) sont utilisées pour implémenter les canaux de communication entre les processus du modèle fonctionnel. Chaque ressource du modèle d'architecture est paramétré grâce à différents attributs. SLOOP fournit une bibliothèque de classes paramétrable permettant aux concepteurs de manipuler simplement les classes.

Le dernier modèle, appelé performance, permet de placer les processus du modèle fonctionnel sur les ressources définis dans le modèle d'architecture, et ce de manière explicite. Il permet également de placer les canaux de communication sur les ressources de communication. Avec ce modèle, il est possible d'éffectuer une simulation statisque de l'application placé sur l'architecure. Cette simulation permet de découvrir goulot d'étranglement de cette architecture. Cela permet au concepteur d'améliorer le système pour pouvoir satisfaire aux conditions de performance.

La méthodologie SLOOP utilise C++ et SystemC™ comme un langage de description pour implémenter chacun des modèles. Les concepteurs peuvent valider la fonctionnalité et les performances grâce à une vérification basée sur la simulation. UML est utilisé comme un langage de spécification antérieur à l'implémentation des modèles. SLOOP introduit deux phases pour réaliser les modèles. Une phase de modélisation qui spécifie les résultats

[7]System Level design with Object-Oriented Process

[8]entité demandant la conception du système

[9]Digital Signal Processor

[10]Application-Specific Integrated Circuit

de l'analyse et la conception en utilisant UML. Dans une seconde phase, les modèles UML sont implémentés en C++/SystemC™ pour obtenir un modèle exécutable.

2.2.3 UML for SoC

Enfin, une RFC[11] soumise à l'OMG présentant un profil étendant UML pour les systèmes monopuces. Ce profil a été soumis par un groupement industriel regroupant Fujitsu, IBM et NEC. Ce groupe est également soutenu par CANON, CATS Co., Metabolics, RICOH COMPANY, Toshiba, UML for SoC Forum et UMTP Japan. Cette approche propose un ensemble d'extension permettant de modéliser des systèmes monopuces, en ajoutant de la sémantique par ajout de stéréotype aux métaclasses UML. Il propose également l'ajout d'un nouveau diagramme, le diagramme de structure pour systèmes monopuce[12]. Pour créer un modèle exécutable au niveau système, les concepteurs doivent créer aussi bien des diagrammes de classe que le diagramme de structure.

Mes travaux de recherche se situent dans le projet DaRT (INRIA). Celui-ci a bien entendu évolué pendant mon travail de thèse. Je présente ici le projet tel qu'il était défini au début de ma thèse. C'est sur cette définition que je m'appuierai pour présenter ma contribution. Néanmoins les résultats récents du projet DaRT sont accessibles sur [Dar].

[11] Request For Comments
[12] SoC structure diagram

Chapitre 3

Gaspard : Graphical Array Specification for PARallel and Distributed computing

3.1 Présentation générale

3.1.1 Objectif du projet

Nos activités de recherche concernent depuis de nombreuses années les modèles et techniques de compilation data parallèles. Depuis 5 ans, ces acquis ont permis d'aborder le problème particulier du traitement de signal intensif en respectant des contraintes temps réel. Aujourd'hui, notre projet repose sur cette double expérience : il vise à ouvrir cette nouvelle voie d'application du paradigme data parallèle. Dans un premier temps il convient d'identifier les caractéristiques propres aux systèmes de traitement de signal intensif afin de pouvoir aboutir à la définition d'un cadre formel de spécification d'algorithmes hautes performances, à la mise en œuvre des techniques de compilation adéquates, à la validation par prototype de supports d'exécution pour différentes plates-formes (SMP, SoC – *System On Chip* ou systèmes sur silicium, systèmes distribués), voire à la caractérisation de modules architecturaux particulièrement adaptés à ce type de fonctionnement. Toutes ces actions sont rattachées à des projets multi-partenaires européens, pour la plupart des projets ITEA.

Système sur silicium, System On Chip *(SoC)*

Un système sur silicium, *System On Chip*, SoC, est un système hétérogène, composé d'éléments de natures diverses (composant logiciel, matériel, ASIC...). Ils sont généralement conçus spécifiquement pour une application donnée (ou une classe réduite d'applications).
Vu l'importante croissance de complexité que connaissent les SoC, particulièrement dans des domaines tels que le traitement de signal intensif, il devient de plus en plus nécessaire de réutiliser dans leur conception des composants déjà élaborés. De tels composants peuvent être des processeurs, DSP, mémoires, bus de communication... Ils sont généralement catalogués et échangés sur le réseau Internet en respectant les standards VSIA *Virtual Socket Interface Alliance* (`http://www.vsi.org/`), sous forme de boîtes noires (l'utilisateur ne peut avoir accès qu'à l'interface). Ce sont les composants virtuels réutilisables (*Intellectual Property, IP*).

3.1.2 Contexte économique et technique

Dans les dix prochaines années, le développement de systèmes logiciels et matériels hautes performances jouera un rôle crucial dans le domaine des télécommunications et des applications multimédia. Ces systèmes devront traiter tout un ensemble de problèmes divers, suivant des points de vue différents en fonction du niveau d'étude : de la spécification de l'application à la réalisation de systèmes matériels embarqués, en passant par la mise en œuvre sur des COTS (*Components Off The Shelf*) hautes performances. Ils concerneront des traitements particuliers comme le traitement intensif (filtrage numérique, JPEG2000), le traitement et/ou la transmission d'images. Ils exigeront des environnements de programmation pour la spécification, la simulation/vérification, la compilation et l'exécution afin de réduire les temps de développement et donc de mise sur le marché. L'architecture de ces systèmes sera fondamentalement hétérogène. Elle sera basée sur l'intégration de diverses unités de calcul (logicielle et matérielle) consacrées aux fonctions spécifiques comme le traitement intensif, la prise de décision et la surveillance. Malheureusement, l'effort de programmation et de mise en exécution de tels systèmes numériques devient de plus en plus complexe. L'évolution des environnements comme souvent ne rivalise pas avec l'évolution des technologies employées.

En outre, les applications complexes de traitement de signal jouent aujourd'hui un rôle important sur beaucoup de marchés différents :

- Les consommateurs de systèmes de télécommunications exigent l'accès en direct à l'information depuis un vaste choix de dispositifs, tels que des ordinateurs, des émissions satellites, des PDA, des agendas électroniques, des téléphones sans fil. L'évolution permanente des besoins en communication impose aux compagnies de télécommunications de déployer encore plus de débit et de supporter un service temps réel sur leurs réseaux câblés et sur les réseaux sans fil. Ces réseaux, et les systèmes associés, devront fournir plus d'un téraoctet d'information en mode synchrone pour satisfaire des applications telles que la vidéo sur demande, la visioconférence. Avec l'avènement des réseaux de 3^{e} génération et l'utilisation de GPRS, d'UMTS, beaucoup d'utilisa-

teurs grand public emploieront des systèmes mobiles personnels comme PDA ou PC portables pour communiquer, en fonctionnement mobile, par l'Internet et l'intranet de leur entreprise. Tous ces dispositifs seront construits à partir d'IP (*Intellectual Properties*) et pourront fournir à l'utilisateur des fonctionnalités efficaces pour les applications multimédia. Des antennes innovatrices et intelligentes doivent être conçues et testées pour supporter les services demandés en termes de couverture, de service et de débit. La simulation précise de ces nouveaux matériels est une nécessité pour l'industrie, et, dus à leur complexité intrinsèque, les développements d'applications et de composants dédiés restent particulièrement difficiles. Les systèmes de spécification des algorithmes, des solutions à base de réutilisation d'IP et les simulations de ces systèmes doivent permettre de réduire le temps de conception sans diminuer l'exactitude des résultats. C'est un enjeu industriel qui peut pérenniser le leadership dans ce domaine.

- Le trafic aérien continue à augmenter, avec pour résultat un plus fort encombrement et de nombreux conflits dans l'espace aérien. Des systèmes de commande de trafic aérien sont en cours de modernisation afin de supporter des charges de circulation plus élevées et de satisfaire des prises de décision plus efficaces d'aiguillage. L'amélioration du fonctionnement temps réel des systèmes augmentera la sûreté et l'efficacité du transport aérien. Les données à prendre en compte sont nombreuses et elles aussi, en voie d'explosion. Le calcul intensif en temps réel jouera un rôle critique dans la nouvelle réglementation du trafic aérien. D'ailleurs, les canaux de télécommunications supplémentaires pour les liaisons de transmission de données au sol et vers les satellites, exigent le placement optimal à bord d'un grand nombre d'antennes. Les solutions à base d'IP ou de COTS devraient permettre une meilleure conception d'antennes tout en réduisant les coûts d'installation.
- L'automobile sollicitant des traitements de l'information à tout moment depuis n'importe quel emplacement, l'industrie de l'automobile se doit de fournir un certain nombre de services nomades d'une richesse équivalente aux services des réseaux sans fil. Les automobilistes recevront de l'information en temps réel, leur position GPS, et des données dynamiques par des stations au sol et des satellites. Connaissant la destination, les systèmes embarqués produisent les mises à jour en temps réel de l'itinéraire à suivre, valident ces changements sur les interfaces visuelles du véhicule, et permettent ainsi d'éviter les accidents, les travaux routiers, ou l'encombrement du trafic en temps réel. Des antennes supplémentaires pour les GSM et les liens satellites sont nécessaires pour la transmission et le positionnement. La quantité des données à traiter, ainsi que la diversité des traitements à effectuer orientent les concepteurs vers des systèmes embarqués hétérogènes construits à base d'IP hautes performances (SIMD, DSP). Les environnements de spécification et de simulation sont indispensables afin de réduire les coûts et les temps de conception, et donc de rester concurrentiel sur ce marché international.
- Les systèmes temps réel sont déjà fortement utilisés dans les communautés scientifiques pour la surveillance et la commande d'instruments. Le nombre de plus en plus important de capteurs, ainsi que leur distribution géographique imposent l'utilisation

de nouvelles techniques temps réel. La mise en œuvre de nouvelles expérimentations demande une gestion temps réel d'un grand nombre d'événements où le calcul intensif reste indispensable. Des systèmes construits à base de COTS SMP sont dans ce contexte une solution attirante, car peu coûteuse, assez performants même s'il ne sont pas toujours adaptés au traitement temps réel. Des systèmes d'exploitation standard ou reconnus comme tel par la communauté scientifique seront amenés à supporter des applications temps réel tout en conservant les fonctionnalités offertes par le système d'origine.

Les applications demandent de plus en plus de puissance, en particulier lorsque le traitement doit être effectué sur des flux de données importants. L'Europe, et la France en particulier, occupe traditionnellement une place forte dans le domaine des technologies de systèmes embarqués ainsi que dans les domaines des télécommunications et de l'automobile. Dans ces deux secteurs industriels, la majorité des nouvelles innovations sont basées sur des systèmes embarqués. La capacité de développer rapidement les systèmes embarqués de qualité à forte rentabilité est devenue un facteur concurrentiel crucial. Ces systèmes sont soit construits par assemblage d'IP, soit à partir de cartes multiprocesseurs de type COTS. Dans ces deux cas les environnements de spécification, de validation, de simulation, de compilation et de support d'exécution sont à la base du développement de tels systèmes. Ils doivent répondre aux exigences des industriels : réduire le *time to market*, satisfaire les caractéristiques de l'applicatif, garantir la réutilisabilité, proposer un cadre formel de description de haut niveau, mettre en œuvre les techniques de compilation, de placement et d'ordonnancement de façon automatique, enfin de s'adapter aux standards de l'industrie et du monde du logiciel libre (philosophie *Open Source*).

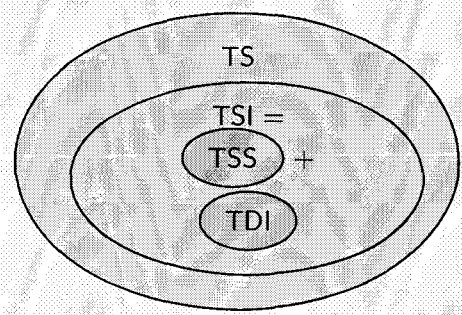
Ces architectures embarquées ont de multiples unités de traitement et très souvent certaines d'entre elles sont parallèles. En effet, exploiter le parallélisme dans l'architecture permet, à travail égal, de réduire la fréquence d'horloge du circuit et par effet de bord sa tension d'alimentation. Cette diminution entraîne une baisse de consommation importante pour la durée de vie des batteries qui alimentent la plupart du temps de tels circuits.

Deux domaines de l'informatique sont amenés à se rejoindre : celui du traitement de signal systématique associé au traitement de données intensif où le traitement de quantité d'informations doit être assuré en respect de contraintes sur le temps ; et celui du calcul hautes performances afin d'exploiter au mieux le parallélisme et de prendre en compte des flux de données intensifs. C'est dans cet objectif que le projet DaRT se situe.

3.1.3 Positionnement général du projet

Le projet DaRT est construit autour de trois axes. Plus qu'un regroupement d'actions, il s'agit dans ce projet de définir des actions de recherche à différents niveaux pour un même domaine de recherche : le parallélisme de données dans le cadre particulier du traitement de signal intensif (voir l'encart de définition page ci-contre). Pour ce seul domaine d'application, nous identifions trois couches complémentaires :

Traitement de signal intensif (TSI)



La partie du traitement de signal (TS) qui nous intéresse est sa partie la plus intensive, composée du traitement de signal systématique (TSS) et du traitement de données intensif (TDI) plus irrégulier. Le TSS correspond à la première phase de traitement des signaux et consiste principalement à l'application de filtres et à des traitements très réguliers (indépendants de la valeur des signaux) appliqués systématiquement aux signaux d'entrée pour en extraire les caractéristiques intéressantes. Celles-ci sont ensuite traitées par des calculs plus irréguliers (dépendants de la valeur de ces grandeurs) dans la phase de TDI.
Ce schéma en deux phases se retrouve dans beaucoup d'applications de traitement de signal ou de l'image. En voici quelques exemples représentatifs venant de nos partenaires industriels.

Récepteur de radio numérique.

Cette application en émergence fait appel à une partie frontale de TSS consistant à la numérisation de la bande de réception, la sélection du canal et l'application de filtres permettant d'éviter les parasites. Les données fournies par ces traitements systématiques sont ensuite envoyées dans le décodeur dont le traitement est plus irrégulier (synchronisation, démodulation, etc.).

Traitement sonar.

Une chaîne de traitement sonar classique se compose d'une première étape systématique : la veille bande large, suivie d'un traitement de données : la poursuite. La première phase prend en entrée les signaux produits par les hydrophones (microphones répartis autour du sous-marin) et, par une suite de traitements systématiques produisant des voies, couples (direction, intensité), représentant les échos captés. Ces échos sont ensuite analysés par la poursuite pour identifier et suivre au cours du temps les objets les produisant.

Encodeur/décodeur JPEG-2000.

JPEG-2000 est un nouveau format standard de compression d'images. Le fonctionnement de l'encodeur [Ada01] suit le même schéma en deux phases. La première partie (du prétraitement à la décomposition en ondelettes) est systématique. C'est dans la deuxième partie de l'encodage qu'apparaissent des traitements irréguliers (quantification, deux étages d'encodage). Le décodeur fonctionne exactement à l'inverse de l'encodeur et fait donc se suivre une phase de TDI et une phase de TSS.

- La spécification formelle d'applications, puis la spécification formelle du placement et de l'ordonnancement de celles-ci sur une architecture particulière.
- La génération de code depuis les spécifications et donc la mise en œuvre de techniques de compilation, en respect du placement et de l'ordonnancement, pour des cibles diverses : des simulateurs d'IP distribués, des SoC, des machines à mémoire partagée ou grappe (*cluster*) à base de COTS.
- Enfin pour le cas particulier des systèmes SMP, ceux-ci étant à notre avis une solution en émergence et prometteuse pour les machines embarquées à forte capacité de calcul, nous proposons de définir un modèle d'ordonnancement de processus à garantie temps réel. Bien que plus large que le seul traitement de signal intensif, ce système devient aussi une des cibles particulières de nos travaux autour des techniques de compilation. Dans le contexte des machines SMP à très grand nombre de processeurs où les temps de latence d'accès à la mémoire sont fortement variables, nous avons réalisé en collaboration avec l'UC Dublin, une etude le comportement de réseaux d'alignement pouvant présenter une garantie de temps de latence constant.

Pour ces trois axes de recherche au sein du projet DaRT, il est possible d'ores et déjà d'identifier les problèmes technologiques à résoudre. Nous les regroupons ici suivant quatre points de vue.

Méthodologie pour le développement d'applications TSI

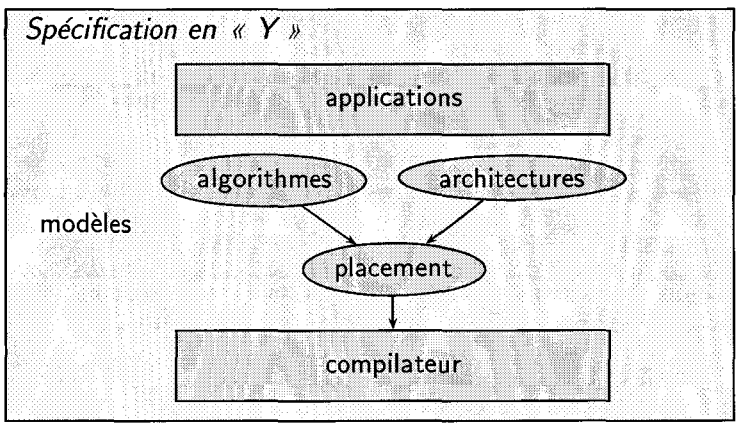
Le niveau actuel de développement des applications pour les systèmes de traitement de signal intensif peut être comparé à l'utilisation de l'assembleur sur les premières machines. En effet l'hétérogénéité des composants et la rareté d'un environnement de développement pour chacun d'entre eux font que l'on a souvent recours à la programmation assembleur des divers IP qui constituent le système embarqué. Il en va de même pour la gestion de l'ordonnancement et des communications inter-IP. Pour le développement sur SMP les choses ne vont guère mieux : même si ici on dispose d'un véritablement environnement, rien ne permet de gérer finement les placements et les migrations des données sur une grappe ou entre les grappes. Dans les deux cas, l'utilisateur navigue entre spécification de l'application et optimisation du code produit sur une machine donnée, souvent à l'aide d'outils de bas niveau où la différenciation entre ces deux tâches n'est pas clairement établie.

Les enjeux ici sont les mêmes que ceux de la programmation classique : modéliser pour unifier et réutiliser ! La différence vient du domaine d'application lui-même. Il permet de proposer un cadre spécifique pour le développement d'applications data-parallèles de traitement de signal et donc de limiter la portée d'une proposition d'un cadre formel. La construction de notre approche fait suite à l'observation des techniques déjà utilisées par les concepteurs d'applications, mais souvent sans jamais les avoir formellement spécifiées. Nous proposons un modèle basé sur l'assignation unique et l'expression explicite de dépendances qu'elles soient temporelles ou spatiales. Dans un souci de respect des standards utilisés dans le monde industriel, nos propositions sont intégrées dans un formalisme UML (*Unified Modeling Language*).

Model Driven Architecture

L'approche MDA est proposée par l'OMG afin d'apporter une certaine pérennité aux logiciels et à leurs modèles. L'approche MDA consiste à séparer les modèles en modèles indépendants de toutes plates-formes (PIM, *Platform Independant Model*) et en modèles dépendants de la plate-forme cible (PSM, *Plateform Specific Model*). Par plate-forme, l'OMG entend essentiellement une plate-forme logicielle (une technologie comme Java, CORBA, EJB...). L'OMG prévoit d'automatiser le passage d'un PIM vers un PSM à l'aide d'outils de projection. Ainsi une application exprimée en PIM pourra-t-elle être portée vers différentes technologies, et vers les technologies futures et à venir. L'OMG prévoit aussi l'automatisation de la génération de code à partir d'un PSM (ex : d'un modèle représentant une application Java vers le code Java correspondant). L'approche MDA s'appuie sur les technologies UML 2.0, MOF et XMI.
Notre démarche suit une même philosophie. En effet, les modèles d'application, d'architecture et de placement sont indépendants de la plate-forme d'exécution. Ce n'est que lors d'une dernière phase de transformation de modèles que nous projetons le modèle d'application placée et ordonnancée, sur un modèle dépendant des choix technologiques. Un même modèle pourra ainsi être décliné sur plusieurs niveaux de simulation ou encore plusieurs technologies de composants.

Multi-cible : SoC, COTS, simulation



De la diversité de nos cibles, naît le besoin d'indépendance entre application et architecture. Actuellement les développeurs d'applications de ce type, de par le manque d'outil de spécification de haut niveau, mélangent l'application elle-même avec son exécution sur une machine particulière. Le constat de ce type de développement est le manque de réutilisabilité, de dynamicité et donc un travail lourd et souvent répétitif. Là encore, l'observation des techniques mises en œuvre dans le monde industriel nous amène à proposer une séparation de la spécification de l'application, de la spécification de l'architecture et de la spécification du placement d'une application sur une architecture particulière. Cette méthodologie de spécification en « Y » (voir l'encart) autorise par construction la réutilisation aussi bien de l'application que de l'architecture. Là encore notre souci de respect des standards nous a orienté vers une construction « Y » dans un formalisme UML et suivant une philosophie de type MDA (*Model Driven Architecture*) [Boa01] (voir l'encart page précédente).

Placement/ordonnancement et génération de code d'applications TSI

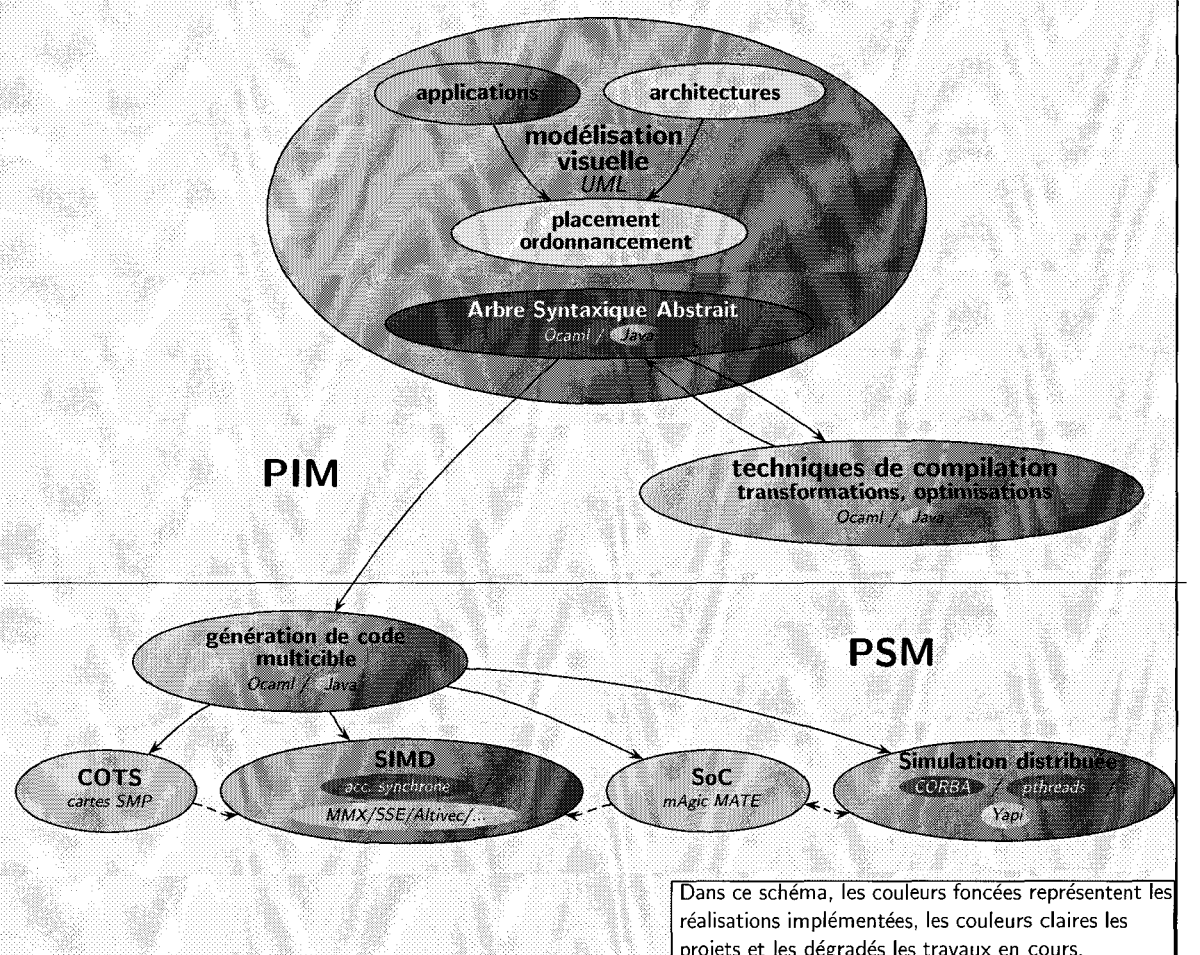
La finalité d'un tel système reste bien sûr la génération d'un exécutable pour une architecture donnée. Notre architecture en « Y » autorise la mise en œuvre de techniques de compilation (placement, ordonnancement, transformation de nids de boucles, barrières de synchronisation...) dès la spécification du placement. Ainsi des techniques soit déduites des techniques standard de parallélisation, soit construites par rapport au domaine particulier visé, sont applicables au plus haut niveau et avant toute génération de code. Néanmoins la diversité des cibles (IP, COTS, simulateurs via VSIA...) demande la mise à disposition de nombreux générateurs de code. Dans notre environnement Gaspard (voir l'encart page suivante) un effort particulier est réalisé pour faciliter l'intégration de nouveaux générateurs, en particulier pour des architectures distribuées.

Système d'exploitation temps réel sur machine SMP

Une alternative actuellement en émergence dans la communauté industrielle consiste à tenter de remplacer les SoC par des machines multiprocesseurs du commerce. La motivation est double, il s'agit de réaliser à moindre coût des machines satisfaisant les contraintes de temps d'exécution, mais aussi d'introduire dans les applications temps réel, des nouvelles fonctionnalités telles que bases de données, multi-tâches, visualisation... Des processus temps réel devront alors cohabiter avec des processus standard, cette diversité ne devant nuire en aucune façon au traitement temps réel.

Gaspard

Gaspard est notre plate-forme de prototypage. Il est structuré autour d'un arbre syntaxique abstrait représentant une application placée sur une architecture. Cet arbre est obtenu par modélisation visuelle (suivant le schéma en « Y »). La compilation se fait par des transformations de cet arbre puis génération de code multicible.



Dans ce schéma, les couleurs foncées représentent les réalisations implémentées, les couleurs claires les projets et les dégradés les travaux en cours.

Nous constatons que les nouvelles générations de machines SMP sont à même de fournir la puissance de calcul nécessaire à certaines applications de traitement de signal intensif. Ces machines sont construites initialement pour un traitement multithreadé dédié au calcul intensif. Lorsqu'il s'agit de garantir en même temps les temps de réponse de certains threads dits temps réel, avec prise en compte des interruptions produites par des cartes d'acquisition, les systèmes d'exploitations standard utilisés (par exemple Linux) ne peuvent satisfaire les contraintes d'exécution. De ce constat est née la nécessité d'un système d'exploitation temps réel pour ces machines. Allier le calcul intensif, le temps réel, les architectures SMP 64 bits et un système d'exploitation Open Source comme Linux est un enjeu économique crucial à moyen terme. Le problème technique sous-jacent est difficile ; il s'agit de proposer une modification du noyau sans être trop intrusif afin de suivre les évolutions de celui-ci. Les travaux sur l'ordonnancement Linux temps réel que nous proposons ne sont que le cœur du système qui doit être « entouré » des outils nécessaires à son exploitation. L'évolution vers des machines NUMA ne facilite pas la tâche : la localité des données entraînant nécessairement des perturbations sur les temps d'accès.

3.1.4 Modélisation et spécification

Le projet DaRT repose sur l'idée simple de proposer au plus haut niveau, dans un environnement unique et si possible standard, un formalisme et une méthodologie qui permettent de spécifier explicitement l'ensemble des informations que possède le concepteur de l'application et qui seront exploitées par les différentes couches de notre environnement : analyse de code, placement, ordonnancement, génération de code...

Ce souci est bien entendu universel, il devient réaliste lorsque l'on exploite un domaine applicatif particulier : le traitement de signal systématique et le traitement de données intensif qui en découle seront nos cibles privilégiées. Par le biais de collaborations industrielles, il convient de s'inspirer, voire de calquer, les méthodes actuellement utilisées. Elles sont souvent multiples et correspondent à plusieurs niveaux de spécification où différence entre spécification et exécution est rarement exposée.

Dans notre spécialisation d'UML en ISP UML (Intensive Signal Processing UML), nous voulons satisfaire les objectifs suivants :

S'affranchir des langages de programmation, en proposant une spécification visuelle au moins au niveau des expressions des dépendances si ce n'est au niveau du code des tâches élémentaires.

Réduire les temps de développement :

- en permettant la réutilisation totale ou partielle d'applications existantes, ou d'architectures existantes ;
- en proposant des bibliothèques de composants prêts à l'emploi.

Respecter les standards et donc l'adéquation avec divers outils qui les respectent.

Exprimer complètement le parallélisme potentiel d'une application : parallélisme de tâche ou de données.

Spécifier à différents niveaux une architecture, autant au niveau réseau d'échange d'une grappe qu'au niveau transfert de données sur une carte SMP ou dans un SoC.

Le modèle « Y »

Le modèle « Y » repose sur un environnent permettant une spécification visuelle des applications TSI, des architectures cibles et du déploiement des applications sur les architectures. Ce modèle de construction particulier permet de différencier la spécification, le support d'exécution et l'exécution proprement dite de l'application sur ce support. Séparer ces modèles correspond à une habitude de programmation dans ce domaine d'application. Il est présent dans des environnements dédiés pour le *co-design* (conception conjointe logicielle/matérielle), pour la validation, pour la simulation et pour la génération de code.

La séparation des modèles ouvre les voies de la réutilisabilité. Une même architecture ou application pourra apparaître dans plusieurs projets, en particulier on doit pouvoir réutiliser une application sur une nouvelle architecture, développer une nouvelle application ou transformer une application sur une architecture existante. Le placement reste bien sûr lié à l'application et à l'architecture, même s'il est indépendant de la plate-forme d'exécution (SystemC, etc). La séparation permet également un partage des tâches de développement par métier; définir une application ne demande pas les mêmes compétences que celles nécessaires pour définir une architecture : assignation à la personne la plus compétente dans le domaine, pas d'interférence avec les autres domaines. La réutilisation ainsi que la répartition vers les personnes les plus qualifiées contribuent à une diminution des temps de développement et permettent donc de réduire le *time to market.*

Nous proposons un environnement visuel de spécification pour cette architecture « Y » construit autour d'une même métaphore. Il doit faciliter l'échange entre les différents métiers sans nécessiter la maîtrise de plusieurs langages. Il en va de même pour les environnements de spécification utilisés : pas d'apprentissage de plusieurs outils ou interfaces.

À chaque modèle on associe une méthodologie sachant que les interférences entre les trois modèles imposent par construction la définition d'une méthodologie collaborative et cohérente. Cette méthodologie transversale guide la réalisation complète d'une application TSI. Elle assure la cohérence entre les modèles, et en permet l'exploitation automatique par les outils de compilation, de génération de code, de transformation et de simulation.

De cette analyse nous pouvons extraire un certain nombre de critères communs aux trois modèles identifiés :

Utiliser le même formalisme : un utilisateur doit pouvoir passer aisément d'une spécification à une autre sans avoir à apprendre de nouveaux concepts. Il s'agit de l'expression des dépendances : de données spatiales et temporelles pour les applications, de flux de données dans le temps pour les composants matériels.

Utiliser la même notation : ici nous retenons le « langage visuel » UML, auquel nous ajoutons des extensions et des règles de constructions. Ces extensions et ces règles sont, dans un premier temps, réunies dans un profil UML dédié au TSI . Nous envisageons par la suite de fournir un langage dérivé d'UML et dédié au TSI. Ce langage sera exprimé à l'aide du MOF.

Le modèle « Y » pour les applications de télécommunications embarquées
Le projet itea *Prompt2Implementation (P2I)*

Dans le but de concevoir un système de co-design pour des applications de télécommunications embarquées, le projet P2I propose de mettre en place le modèle « Y » en adaptant des outils de spécification d'applications, d'architectures, et de placements existants. Ayant pour objectif la définition d'une méthodologie non-ambiguë pour la spécification, la simulation, le test et l'implémentation de tels systèmes prouvés, le projet P2I reprend l'approche « Y » avec comme outils de spécification Esterel Studio pour la spécification de l'application et les tests fonctionnels couplé à SynDEx pour la spécification de l'architecture et du placement. Ces deux approches devraient être unifiées dans un environnement RTE UML (*Real Time Embedded UML*). Partant de briques existantes, on peut espérer rapidement valider le modèle « Y » et son intégration dans le langage UML. Il doit alors être possible à partir de la spécification, de vérifier et de tester une application puis de générer du code efficace après placement par la méthode AAA, en particulier pour des applications de télécommunications. Notre action dans ce projet concerne effectivement la validation du modèle « Y » par prototypage via un profile/framework UML. Un effort particulier sera fait vers l'OMG (`http://www.omg.org/`) par le biais de propositions de standardisation. À la différence de l'objectif du projet DaRT, ces outils existants ne supportent pas explicitement le paradigme data-parallèle que nous tentons d'intégrer par expression des dépendances de données.

Utiliser la même représentation interne : les outils visuels et les outils d'exploitation des modèles utilisent des représentations internes (représentations en mémoire ou persistantes). Nous proposons une représentation commune à tous les outils que nous développons, ainsi que des interfaces/bibliothèques de base qui en permettent la manipulation.

Utiliser les mêmes outils externes : les outils utilisés doivent être les mêmes pour les différentes spécifications, afin que les utilisateurs puissent passer facilement d'une spécification à une autre.

Assurer une exploitation automatique : les différentes méthodologies permettrons d'obtenir des modèles d'applications placées exploitables automatiquement par différents outils tels que des générateurs de code, des simulateurs...

UML comme langage de modélisation

Le choix d'UML [Obj01c] comme langage de modélisation commun s'est imposé du fait de ses avantages :

- UML est un standard reconnu et de plus en plus utilisé, dans le milieu industriel en particulier.
- Il offre des mécanismes d'extension (*stereotypes*, *tagged values*, *profils*) nous permettant d'apporter nos propres éléments sans modifier le langage UML lui-même.
- Il n'impose pas de méthodologie d'utilisation. Nous pouvons valider notre propre méthodologie, en particulier notre modèle « Y ».
- Il est visuel aussi bien que textuel.

- Différents outils visuels existent déjà autour de ce standard (Rational Rose [Rat01], Objecteering [Obj02c], Tau G2 [Tel02], etc.).
- UML est modélisé par un métamodèle lui même spécifié par le MOF [Obj00] (sous-ensemble d'UML). Ceci nous permet de fournir notre propre métamodèle qui sera dans un premier temps une extension de celui d'UML. Plus tard, par réduction de ces spécifications notre métamodèle ne fournira que ce qui est réellement nécessaire à la spécification d'applications TSI.
- L'échange de modèles entre les outils est (plus ou moins) assuré par le standard XMI/XML [Obj02b]. Les outils que nous développerons bénéficieront d'une représentation interne basée sur le métamodèle de notre langage ISP UML, (exprimés à l'aide du MOF). La persistance, ainsi que l'échange avec les autres outils, se fait à l'aide de XML/XMI.

UML a été originellement conçu afin de modéliser les artefacts d'un système à forte composante logicielle. Ceci comprend la spécification d'applications, d'architectures et du déploiement d'applications sur les architectures. Il est donc théoriquement d'ores et déjà possible de spécifier une application TSI avec UML. Cependant, aucune méthodologie permettant une exploitation automatique du modèle n'existe pour les applications TSI. C'est cette lacune que nous nous proposons de combler. De plus, les concepts d'UML permettant la spécification visuelle d'une architecture et le déploiement de l'application basée sur cette architecture sont reconnus comme étant les « parents pauvres » d'UML. Là aussi nous proposerons une méthodologie utilisant UML et permettant la spécification et le déploiement d'architectures complexes utilisées dans le TSI.

D'autres projets proposant la modélisation d'applications temps réel ou embarquées ont aussi fait le choix d'UML comme langage de modélisation. Ainsi, le domaine des télécommunications est en train de proposer une nouvelle version de SDL (Syntax Description Language), SDL-2000 [Tel99] orientée UML. Le modèle théorique de SDL repose sur un ensemble d'automates à états finis fonctionnant en parallèle dans des agents (sorte de classes) qui communiquent entre eux par des échanges de signaux. SDL ne permet pas la spécification d'architectures ou le déploiement. La communauté UML propose quant à elle des extensions d'UML (UML-RT, RT-UML [SR98, Obj02a]) dédiées à la modélisation d'applications temps réel. Ces propositions sont elles aussi tournées vers des applications communiquant par échanges de signaux. Elle permettent la génération de code, généralement C/C++, pour l'architecture cible, mais n'intègrent pas la spécification d'architecture. Le projet INRIA Triskell a pour objectifs la « validation de logiciels répartis décrits avec UML ». La construction d'applications se fait par assemblage de composants. Le projet est validé par une plate-forme d'expérimentation, UMLaut [HLPJ99], réalisé en Eiffel et utilisant une représentation interne du métamodèle UML. UMLaut permet la transformation du modèle, et la génération de code (uniquement C pour l'instant). La proposition « Embedded UML » [?] propose de faire la synthèse des modèles existant en ne retenant que les points attractifs. Elle propose de séparer la spécification de l'application, de l'architecture et du déploiement, le lien vers les plates-formes visées se faisant dans la spécification du déploiement. Là aussi la communication entre les « blocs » se fait par échange de signaux. Il faut aussi mentionner des modèles comme SystemC [Ope02] et Yapi [dKES+00], qui ne

possèdent pas de modélisation visuelle (ni même UML) mais permettent une spécification des applications en C++. Notre approche se différencie des projets précédents par le fait que nous voulons modéliser des applications par l'expression des dépendances, plutôt que par les échanges de signaux ou l'envoi de messages. De plus, nous voulons clairement séparer la spécification de l'application de l'architecture et du déploiement, et ainsi permettre la réutilisation d'applications et d'architectures.

UML est bien reconnu comme standard pour la modélisation visuelle : il est, jusqu'à présent, surtout utilisé comme outil d'aide à la spécification. Son utilisation en tant qu'outil permettant de générer l'application elle-même n'est pas encore répandue, mais plusieurs travaux vont dans ce sens, dont la nouvelle proposition UML 2.0. L'exploitation d'un modèle pour la génération d'une application n'est possible que si la description du modèle s'accompagne de règles très strictes de modélisation. Dans notre cas, l'exploitation automatique des modèles est possible car nous nous restreignons à la spécification d'applications TSI en suivant des règles précises que nous fixerons. Ces règles font partie de la méthodologie que nous proposons pour la spécification d'applications TSI.

Spécification d'applications TSI : ISP UML

Notre objectif concerne la spécification d'algorithmes à un haut niveau d'abstraction. Nous avons déduit de l'observation des différents modèles de spécification utilisés par nos collaborateurs industriels qu'un système de traitement de signal intensif devait respecter les contraintes suivantes.

Assignation unique : les données sont principalement des tableaux produits par une tâche élémentaire de traitement et consommés sans modification par d'autres tâches élémentaires. Cette assignation unique de tableaux facilite la spécification visuelle d'une application.

Unification des dimensions temporelles et spatiales : les dimensions des tableaux sont en général associées à des concepts propres à l'application (hydrophones, capteurs, énergie...). L'une d'entre-elles peut être associée au temps, elle permet alors l'identification des différentes valeurs de ces mêmes concepts durant la vie de l'application. Cette dimension devient alors de taille infinie pour une application embarquée.

Expression des dépendances temporelles et spatiales : elle représente le seul lien qui unisse les différents objets manipulés par le programme. Elle fournit les dépendances entre les éléments des objets (tableaux) en entrée et en sortie de chaque tâche élémentaire. Seuls les objets nécessaires à la production des autres objets sont identifiés (au niveau des tableaux ou des éléments de tableaux). Elle recouvre autant les dimensions spatiales que temporelles. Elle permet la mise en œuvre directe des techniques de compilation. Elle garantie la concurrence par l'expression d'ordres partiels basés sur les dépendances entre objets. Nous identifions trois types de dépendances :

1. Dépendances « tableaux » : modèle global d'Array-OL, réseaux de processus, Yapi ;
2. Dépendances « itératives » : modèle local d'Array-OL, `forall` des langages data-parallèles ;

3. Dépendances « alternatives » : application irrégulière, traitement non systématique, conditionnelles.

Universalité d'un langage de programmation : le domaine d'application doit couvrir le traitement de signal intensif. Au traitement systématique du signal est souvent associé un traitement de données intensif. Celui-ci est souvent irrégulier, il manipule des structures de données dynamiques et les traitements sont à comportement variable.

Nous avons montré dans nos précédents travaux que ces contraintes permettent l'expression d'applications TS complexes, ainsi que l'exploitation automatique des spécifications résultantes par les compilateurs ou simulateurs.

L'intégration de ces contraintes dans un environnement comme UML passe par la définition d'une métaphore. Celle-ci est liée directement à notre approche originale pour la modélisation des applications TSI : la modélisation par spécification de dépendances. La plupart des outils de spécification d'applications TS existants ou en développement (SystemC, Embedded UML, Triskell, UML-RT) proposent une modélisation basée sur la description de composants communicant entre eux par échange de signaux ou de messages. Dans notre approche, une application se modélise en spécifiant les composants à exécuter, et les objets dont dépend chaque composant pour pouvoir s'exécuter. Les composants sont reliés entre eux par l'expression des dépendances de données. Une construction hiérarchique des composants permet de limiter notre métaphore à cette notion de composant. Cette approche se prête bien à une modélisation visuelle, en particulier en UML. Les figures 3.1 et 3.2 illustrent une représentation possible, conforme à la future norme UML 2.0, des dépendances entre composants.

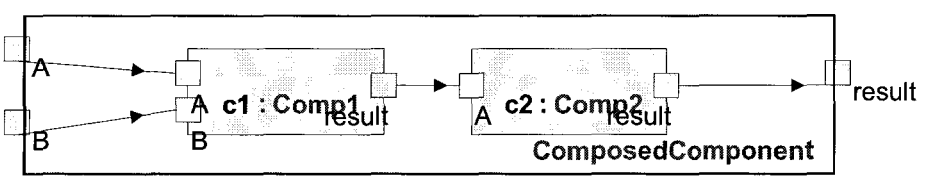


FIG. 3.1 – Diagramme de structure d'un composant composé.

Spécification d'architectures

La deuxième étape de construction du modèle « Y » consiste à spécifier une architecture matérielle. Afin de conserver un environnement unique de spécification c'est aussi en UML qu'il faudra supporter cette spécification. Or, même s'il existe des éléments de modélisation pour cette spécification en UML (nœuds, et connexions), il est évident qu'il ne s'agit pas d'une priorité forte retenue par l'OMG. Les environnements de co-design qui utilisent également un système de spécification d'architecture et de placement demandent beaucoup plus de précision sur les caractéristiques de l'architecture. C'est encore plus vrai lorsque les applications visées concernent le temps réel.

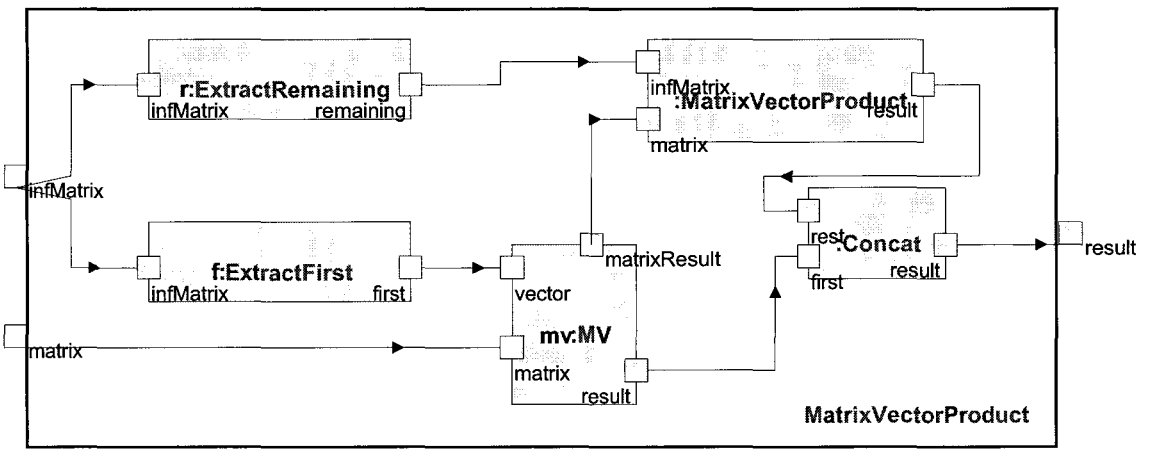


FIG. 3.2 – Exemple de composant récursif : un produit de matrice-vecteur récursif. Cet exemple est une modélisation sans utilisation de retards de l'exemple de la page 86 de [Dem01]. infMatrix et result sont des tableaux infinis (flux) de vecteurs. Chaque vecteur subit un produit (dans le composant MV) avec une matrice (matrix) qui évolue en fonction du produit précédent.

Notre approche consiste donc à enrichir les capacités de modélisation d'architecture d'UML. Plusieurs remarques devront nous guider dans cette définition :

- Il faut proposer un modèle de spécification relativement semblable à celui utilisé pour l'application, dans le but principal de limiter l'effort du programmeur si celui-ci est en charge des deux spécifications, de permettre une lecture rapide de l'algorithme ou de l'architecture quelque soit le métier de l'utilisateur. Les notions de composant, de graphe de dépendances et d'itérateur seront vraisemblablement conservées ou adaptées.
- Il doit permettre la modélisation des cibles des applications : SoC, COTS, simulateurs distribués... Dans cette diversité d'architectures on retrouve évidemment des points de convergence : multiprocesseurs, communications, hétérogénéité globale, homogénéité locale.
- Il faut pouvoir décrire différents niveaux de spécification pour différents niveaux de simulation. La précision de l'architecture permettra un placement fin, une description plus grossière est parfois suffisante en particulier dans le cas d'une simulation distribuée.
- La construction des architectures doit être hiérarchique et itérative.
- Deux types de composant semblent nécessaires : les composants actifs qui participent au traitement de l'information et les composants passifs qui stockent les données.
- Comme toujours dans notre démarche, il faudra tirer partie des environnements utilisés par nos partenaires industriels tels que : Array-OL architecture, mAgiSim, Vcc, SynDEx...

Cette proposition de modélisation d'architectures devrait être commune à ISP UML et RTE UML.

Déploiement d'une application sur une architecture

Étape finale avant la génération de code pour simulation ou exécution, le placement de l'application sur l'architecture est, dans le modèle « Y », également spécifié par le programmeur (même si des outils de placement/ordonnancement automatique peuvent intervenir sur cette spécification). Dans les outils précités, cette action correspond souvent à des liens entre le graphe de l'algorithme et le schéma de l'architecture. En UML le déploiement n'est guère plus riche que la spécification d'architecture. Nous devons intégrer dans notre modèle de nouveaux éléments de modélisation pour le placement (ou le déploiement) adapté à nos deux modèles déjà présentés. Là encore une approche similaire à celles proposées pour ces deux modèles permettrait de faciliter le développement d'applications placées sur des architectures.

Ces nouveaux éléments de modélisation représenterons les liens entre algorithme et architecture. Ceux-ci, de par la construction du modèle « Y », restent indépendant l'un de l'autre : une même application peut être placée sur plusieurs architectures et réciproquement. À ce niveau de spécification, le modèle reste indépendant de la plate-forme d'exécution ou de simulation. C'est alors, en respect de la philosophie MDA, que la projection sur des modèles d'exécution tels que SystemC, VHDL, CORBA ou un simulateur d'IP via VSIA produira une représentation hétérogène spécifique aux plates-formes concernées. L'ensemble de ces codes devra être interopérable afin d'assurer les communications entre les IP. La figure 3.3 résume cette démarche.

La solution envisagée repose sur l'expression des associations entre les composants de l'application et les éléments actifs et passifs de l'architecture. Les mêmes itérateurs sont là encore utilisés pour la spécification de placements répétitifs sur le temps ou sur l'espace.

La méthode proposée devrait intégrer le placement explicite qui sera validé dans le contexte de RTE UML.

3.1.5 Compilation et simulation distribuée

À partir d'une spécification de haut niveau d'une application, d'une architecture et du déploiement de la dite application sur cette architecture, il reste un gros travail de compilation et d'optimisation pour obtenir un code (embarqué ou de simulation) avec les meilleures performances possibles pour garantir des temps de réponse rapides. De nombreuses difficultés scientifiques et techniques sont à étudier dans ce domaine.

Compilation efficace d'un modèle de haut niveau : il faut tirer partie de la représentation des dépendances de données pour générer le code le plus efficace possible. De nombreuses *techniques d'optimisations* sont connues. La difficulté réside dans l'*intégration* de ces optimisations au sein du même code. Pour la majeure partie, ces optimisations peuvent se faire indépendamment de la plate-forme d'exécution.

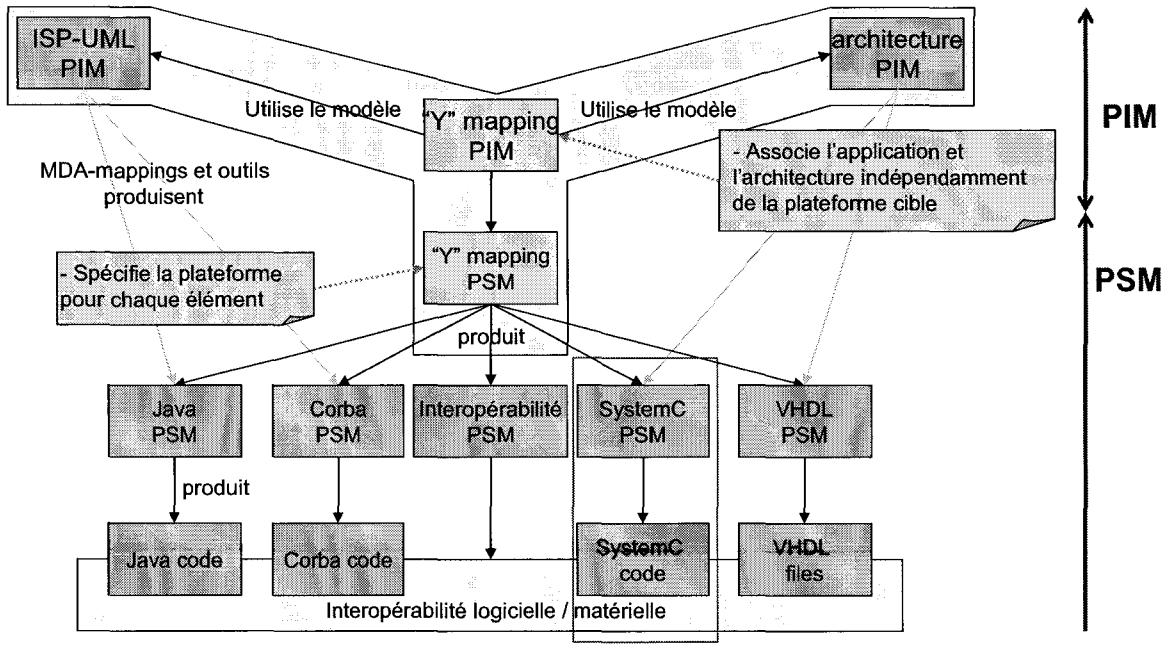


FIG. 3.3 – Démarche de spécification dans une philosophie de type MDA.

En effet, nous ne nous intéressons pas à l'optimisation des tâches élémentaires de l'application, mais à leur enchaînement efficace.

Placement et ordonnancement : c'est le point clé de l'utilisation efficace des multiples unités d'exécution de la cible matérielle. Cette cible étant *hétérogène*, il peut y avoir de fortes contraintes sur le placement (manuel, semi-automatique, voire complètement automatique). Des *transformations de code* (factorisation du graphe de composants, changement de granularité, etc.) peuvent être nécessaires pour exprimer de tels placements. Le placement et l'expression des dépendances permettent alors de choisir un ordonnancement de l'application. Les points difficiles sont ici les choix interdépendants d'un placement et d'un ordonnancement assurant le temps réel. Cette phase est, elle aussi, indépendante de la plate-forme d'exécution. Elle utilise les modèles de l'application et de l'architecture pour produire un modèle de l'application placée.

Génération de code : nous voulons, à partir d'un tel modèle d'application placée, être capables de générer un code d'exécution pour diverses architectures telles que des SoC, des architectures à base de COTS ou des simulateurs distribués de telles architectures. Mis à part les problèmes de placement et d'ordonnancement, l'*hétérogénéité* des cibles matérielles et leur nature *répartie* font de la génération de code un problème techniquement difficile. C'est lors de cette étape que nous spécialisons le modèle d'application placée pour la plate-forme d'exécution grâce à des modèles décrivant ces plates-formes.

Simulation distribuée : dans ce cas, les contraintes de placement sont moins fortes mais le support peut être *dynamique* (évolution du matériel, du logiciel, tolérance aux pannes, etc.). Outre cette difficulté technique supplémentaire, se pose le problème du *niveau de simulation.* En effet, on peut vouloir simplement une simulation fonctionnelle de l'application pour valider l'algorithme mais on peut aussi bien vouloir simuler le fonctionnement de l'application sur un simulateur de la cible matérielle choisie. Nous sommes alors confrontés à des problèmes très complexes de *couplage* des simulateurs de chaque composant matériel.

Compilation et placement spatio-temporel

Une des idées directrices de notre modélisation est de donner suffisamment de liberté au programmeur pour qu'il puisse exprimer le parallélisme de son application. En particulier, il ne doit pas pouvoir exprimer de fausses dépendances de données, dues à un ordre d'écriture séquentiel ou à la réutilisation de la mémoire. Ceci libère l'utilisateur des contraintes liées à l'exécution de son application. Il y a deux conséquences sur le compilateur : d'une part il a toutes les informations nécessaires pour optimiser le code, mais de l'autre, il doit prendre les bonnes décisions et la qualité de ses optimisations est primordiale dans les performances de l'application. Nous pensons que, dans le cadre restreint des applications TSI, les techniques d'optimisation sont suffisamment mûres pour permettre une compilation efficace.

Le modèle d'application d'ISP UML a les propriétés suivantes :

- assignation unique ;
- expression des seules dépendances de flot de données ;
- unification des dimensions temporelles et spatiales ;
- récursif ;
- composants d'ordre supérieur.

L'assignation unique, la récursivité et les composants d'ordre supérieur apparentent ce modèle aux langages fonctionnels purs, c'est-à-dire sans effet de bord. La compilation et la parallélisation de tels langages [SW95, PPRS01, GMS00, Mot00] fait appel à des techniques telles que le typage statique, la manipulation des fonctions comme des citoyens de première classe du langage, la dérécursivation, l'évaluation partielle, etc. D'autre part, le fait d'exprimer uniquement les dépendances de flot permet l'utilisation de nombreuses techniques de parallélisation automatique [BDSV98] [Lim01, DRV00, AK01] (transformation de boucles, tuilage, etc.). Des langages spécifiques au traitement de signal temps réel proposent d'autres types d'optimisations [Ree95]. L'étude des interactions entre ces différents types d'optimisations est un des objectifs du projet DaRT.

Le choix des structures de données utilisées (tableaux à assignation unique) pose le problème majeur de l'utilisation efficace de l'espace mémoire. En identifiant quelles dimensions représentent l'espace et lesquelles représentent le temps, la réutilisation de l'espace de stockage devient possible. Des études allant dans ce sens existent [LF97, MAL93, TP01].

Le placement et l'ordonnancement d'une application sont deux étapes intimement liées du développement de l'application. Il s'agit en effet de la même opération dans les espaces respectivement physique et temporel. Les meilleurs ordonnancements et placements choisis

indépendamment peuvent conduire à des exécutions séquentielles. Il faut donc tenir compte de l'un pour calculer l'autre ou, idéalement, trouver un critère d'optimisation permettant de les réaliser conjointement. La plupart du temps, on optimise l'un puis on utilise ce résultat pour calculer l'autre. Les deux choix d'optimiser d'abord le placement ou d'abord l'ordonnancement existent dans le littérature [DDGV01].

La stratégie choisie par la méthodologie AAA [Sor94, SL97, GLS99] (Adéquation Algorithme Architecture) consiste en l'optimisation simultanée du placement et de l'ordonnancement par des heuristiques gloutonnes. Dans le cadre du projet ITEA Prompt*2Implementation* nous avons collaboré avec l'équipe de l'action INRIA Ostre (qui a développé la méthodologie AAA) pour produire du code pour leur logiciel SynDEx [SL00] à partir de notre modèle. Le placement et l'ordonnancement sont alors automatiques dans le cadre de contraintes posées par l'utilisateur.

Cependant, comme le domaine d'application que nous étudions est restreint et que notre modèle de spécification unifie les dimensions temporelles et physiques des données, une optimisation conjointe plus précise du placement et de l'ordonnancement nous semble possible. Dans la suite de nos travaux, nous rechercherons donc une méthode unifiant le placement et l'ordonnancement au sein du même formalisme pour, à terme, automatiser plus efficacement le placement spatio-temporel d'une application sur une architecture donnée. La proposition de THIES, VIVIEN, SHELDON et AMARASINGHE [TVSA01] dans le cas des ordonnancements affines et des vecteurs d'occupation nous encourage dans cette voie.

Array-OL

Array-OL [BDL+01] est un langage dédié au TSS. Il est basé sur la constatation que la complexité de telles applications vient des accès aux données (toujours des tableaux) et non des fonctions de calcul. On exprime donc visuellement le contrôle de l'application et textuellement les fonctions élémentaires de calcul. Array-OL est un langage à assignation unique où seules les dépendances de flot de données sont exprimées et où les dimensions spatiales et temporelles des tableaux sont banalisées. Il est à la base de nos travaux sur les langages pour le TSI.

Lors d'études précédentes [BDL+01, Sou01], nous avons mis au point des transformations de code Array-OL vers Array-OL. De telles transformations sont nécessaires pour exprimer le placement d'une application Array-OL. Elles permettent en effet de choisir la granularité de l'application, de factoriser le graphe de composants ou de faire un compromis entre occupation mémoire et temps de calcul.

Ces travaux sont le point de départ de futures recherches. Nous allons étendre ces transformations de code dans trois directions :

1. Proposer des stratégies d'enchaînement des transformations élémentaires optimisant certains critères (minimisation de la taille mémoire, des recalculs induits, adaptation à l'architecture mémoire de la cible).
2. Étendre leur champ d'application du traitement de signal systématique au traitement de données intensif.
3. Étudier plus précisément le lien entre les opérateurs de description de tableaux et le modèle polyédrique pour enrichir mutuellement les techniques proposées dans ces deux domaines.

Conjointement à ces extensions, une méthode adaptée à la partie irrégulière du modèle ISP UML comprenant l'enchaînement d'une phase d'optimisation « fonctionnelle » pour se

Génération de code pour système temps réel sur SMP

Les systèmes d'exploitation temps réel sur machine SMP sont une des cibles privilégiées de notre compilateur. Comme nous l'expliquons dans la section ??, nous disposons, dans le système ARTiS que nous proposons, de processeurs « temps réel » et de processeurs non dédiés à l'exécution de processus ayant des contraintes de temps. La compilation pour de tels systèmes implique la définition des bornes de temps (données au niveau de la spécification), et le placement des processus ayant de telles bornes sur les processeurs temps réel. Ce placement étant réalisé par le système d'exploitation, l'expression des bornes de temps sera la principale difficulté de la génération de code pour ce genre de système.

ramener à une expression plus adaptée aux techniques de parallélisation et d'optimisation de langages du type Fortran sera évaluée.

Génération de code

Toutes les étapes de compilation précédentes sont indépendantes de la plate-forme d'exécution. C'est lors de cette étape finale que le code est spécialisé pour la plate-forme d'exécution choisie. On peut vouloir en effet générer différents codes pour une même application ordonnancée sur des composants virtuels pour la simuler à différents niveaux ou encore générer le code exécutable sur les composants matériels.

Les architectures visées sont de plusieurs types partageant à des degrés divers des caractéristiques d'hétérogénéité et de distribution :

- les systèmes sur silicium sont très hétérogènes, mais vu l'absence de système d'exploitation, les transferts de données et les temps d'exécutions sont très finement contrôlables ;
- les architectures à base de cartes multiprocesseurs sont plus homogènes et disposent souvent de compilateurs pour des langages de plus haut niveau (C, C++) ;
- enfin, les systèmes de métacalcul proposent de nombreux services pour assurer l'interopérabilité et la communication entre les composants logiciels de l'application. Le problème majeur réside ici dans la synchronisation entre ces composants. Nous en parlons plus en détail dans la section suivante.

Les applications de TSI sont encore majoritairement écrites en assembleur pour obtenir un maximum de performance. Au vu de la complexité croissante des architectures d'exécution, le coût en temps de développement devient prohibitif. De plus l'hétérogénéité de l'architecture oblige à générer du code dans des langages différents pour chaque composant pour lesquels des assembleurs optimiseurs ou des compilateurs de langages de bas niveau existent généralement. L'objectif de cette phase de compilation est de générer du code pour les différents assembleurs ou compilateurs natifs en fonction du placement et de l'ordonnancement choisi. Rappelons ici que notre modèle n'exprime que l'enchaînement des tâches et le placement des données, pas les tâches élémentaires de calcul. Celles-ci continuent à être développées (par exemple sous forme de bibliothèque) dans les langages natifs des

Cyber-entreprise

Lorsqu'une entreprise veut développer une application de traitement intensif sur un SoC, elle définit généralement l'architecture d'exécution en même temps que son application. Cette architecture est composée de différents composants matériels vendus par différents fournisseurs. Vu la complexité grandissante de ces applications et le coût des composants matériels, des simulations sont nécessaires avant de finaliser les choix.
Ces simulations se font à divers niveaux, fonctionnel d'abord, puis de plus en plus précis jusqu'à être valides au bit et au cycle d'horloge près. Ce n'est que lorsque toutes les simulations donnent un résultat satisfaisant que la décision de fondre le circuit peut être prise.
Se posent alors des problèmes complexes de propriété industrielle : le fournisseur de composant ne doit pas avoir accès au code de l'application simulée et réciproquement, le développeur d'application ne doit pas avoir accès au code du simulateur. Le développement d'une *cyber-entreprise* permettant le couplage via l'Internet des différents simulateurs (restant chez le fournisseur de composants) peut permettre un tel modèle de développement. Se posent alors la définition de l'interface des *composants virtuels* (VC) et de leur interaction.
Les constructeurs se sont regroupés au sein de la *Virtual Socket Interface Alliance* (`http://www.vsi.org/`) afin de définir des standards facilitant la réutilisation de VC pour la conception de SoC.

architectures cibles. Par contre, l'expression du contrôle de l'application reste à la charge du générateur de code. Une bonne partie des difficultés est reportée d'une part dans la phase d'ordonnancement et de placement et d'autre part dans l'écriture des noyaux de calcul que sont les composants élémentaires. L'expression du contrôle peut devenir complexe après transformation du code. Des méthodes comme celles de [BF98] peuvent alors être utilisées. Cependant, elles sont inutiles dans la plupart des cas et une génération simple des itérateurs est souvent possible.

Dans une première version de Gaspard, nous avons pu générer du code pour plusieurs cibles : en C++ avec une bibliothèque pthreads pour une station de travail multithreadée, en C++ et CORBA pour un système de métacalcul et en macro-assembleur pour un processeur de traitement de signal SIMD, l'accélérateur synchrone. Ces premiers développements nous ont amenés à une réflexion sur la modularisation et la paramétrisation du générateur de code.

La démarche de construction du générateur de code sera analogue à celle qui est proposée pour les architectures logicielles dans MDA. Nous allons étudier le passage d'un modèle de l'application indépendant de la cible matérielle à un modèle adapté aux spécificités de cette cible. Cette transformation de modèles sera le plus automatique possible. Selon le choix de plates-formes d'exécution, la génération d'un dispositif (logiciel ou matériel) permettant l'interopérabilité des codes et circuits finaux pourra être nécessaire.

Simulation distribuée

La méthode la plus classique pour la validation des SoC est le prototypage. Elle consiste en une réalisation totale ou partielle du système à concevoir, ce qui s'avère souvent très

coûteux et lent à réaliser [PaRH98]. Dans le cycle de développement d'une application (voir l'encart sur la cyber-entreprise page ci-contre) les méthodes de validation par simulation présentent une alternative incontournable. Ces dernières peuvent être classées globalement en deux classes. La première classe est dite validation compositionnelle, elle consiste à traduire tous les modules du SoC dans un langage choisi, puis de simuler le système ainsi obtenu sur une machine et avec un seul simulateur. L'avantage de cette méthode est la grande vitesse d'exécution à cause de l'unicité du simulateur, cependant elle est très fastidieuse et coûteuse à mettre en œuvre. Elle pose également le problème de la propriété intellectuelle, car généralement l'utilisateur n'as pas accès au comportement de l'IP. Des méthodologies basées sur cette approche son décrites dans [Wil92] et [ZJ93]. Nous nous intéresserons dans ce projet à la seconde classe des méthodes de simulation, la simulation distribuée. Cette distribution pose de nombreux problèmes : hétérogénéité, dynamicité du support, couplage de simulateurs, performance des communications ou encore expression des contraintes de temps.

Nous avons participé au projet européen ITEA Sophocles au sein duquel ont été étudiés les problèmes liés à la cyber-entreprise. Nous nous intéressons à l'utilisation des interfaces des composants virtuels [VSI01] telles qu'elles sont définies par les fournisseurs de tels composants pour construire des simulateurs. Les problèmes plus particuliers que nous étudions se situent au niveau support d'exécution : proposer des bibliothèques permettant de coupler les composants distribués en assurant de bonnes performances.

Un modèle d'exécution de choix est celui des réseaux de processus proposé par Gilles KAHN [Kah74, KM77]. Ce modèle est particulièrement bien adapté à la modélisation des applications à flot de données et des architectures matérielles. Cependant il ne permet pas une expression générale du parallélisme de données. Des travaux menés dans le projet Ptolemy [Lee01, ML02] décrivent un modèle à flot de données synchrone multidimensionnel. Une adaptation de ce modèle d'exécution à notre modèle de spécification pourrait être très intéressante.

Un premier travail [ABD01] utilisant CORBA [Obj01a] pour gérer l'interopérabilité nous a permis de définir un support d'exécution basé sur une interconnexion des composants multithreadés par des files d'attente. Un des points originaux de ce travail par rapport à d'autres approches [PR98, Kea99] d'utilisation de CORBA pour le calcul intensif est la dynamicité. En effet, l'assemblage des composants peut-être modifié pendant l'exécution, permettant un équilibrage de charge dynamique et la tolérance aux pannes.

Plusieurs directions s'offrent à nous à la suite de ces travaux :

- intégrer notre approche avec la spécification de CORBA pour le parallélisme de données [Obj01b] ;
- proposer un modèle de placement des applications de TSI sur un tel support ;
- étudier en collaboration avec Philips un modèle de placement sur leur support d'exécution Yapi [dKES+00] basé sur les réseaux de processus couplé avec notre support pour lui apporter la distribution des calculs.

Dans le projet DaRT, la simulation distribuée consiste à simuler une application TSI sur un SoC dans un environnement hétérogène composé de plusieurs simulateurs différents (par exemple VHDL, SystemC, Matlab). La diversité des modèles (avec ou sans référence

de temps) utilisés à différents niveaux de simulation pose des problèmes de couplage entre les simulateurs des composants virtuels intervenant dans le système. À plus long terme, nous nous intéresserons au couplage de simulateurs de plus bas niveau pour lesquels les problèmes de synchronisation deviennent particulièrement critiques. En effet, une simulation au bit et au cycle près d'une application sur un SoC suppose l'échange très fréquent de petites quantités de données. Des techniques d'anticipation et de regroupement des communications peuvent alors être envisagées pour réduire les délais de communication.

Pour la réalisation de ce projet, nous envisageons de charger chaque simulateur de simuler un module du système dans le langage dans lequel ce dernier est spécifié. Chaque module sera encapsulé dans une enveloppe de simulation décrite en SystemC. L'utilisation de SystemC est d'un grand intérêt car, en plus du fait qu'il dispose de primitives et de concepts permettant de valider un SoC du niveau fonctionnel jusqu'à un bas niveau d'implémentation, son aspect orienté objet procurera à l'environnement de simulation flexibilité et extensibilité.

Les simulateurs communiquent entre eux par l'intermédiaire des enveloppes de simulation via un bus de simulation. Les résultats obtenus lors des travaux sur les réseaux de processus distribués en CORBA sont réutilisés pour la réalisation de ce bus de simulation. Dans le cas d'utilisation d'IP résidant chez leur fournisseur, le concepteur ne dispose que de l'interface du module, telle que définie par [VSI01] (l'interface : entrées/sorties, les protocoles de communication sur les ports, etc.), les enveloppes de simulation peuvent résider dans la machine centrale et communiquer avec le simulateur de l'IP (chez le fournisseur) à distance, sans avoir réellement accès à son comportement. Les enveloppes assurent également :

- la conversion des types des données échangées entre les simulateurs : ceci permet de réaliser une simulation distribuée du SoC à n'importe quel niveau d'abstraction ;
- la conversion de la nature des signaux échangés entre les simulateurs : échanger les données et les signaux de synchronisation entre les simulateurs.

Cet environnement permettra de valider le fonctionnement d'une application TSI sur un SoC à tous les niveaux d'abstraction correspondant aux différentes étapes de conception. Le fait de distribuer géographiquement la simulation présente des intérêts multiples, en particulier :

- permettre la conception et la vérification d'un SoC par plusieurs équipes spécialisées dans des domaines différents, par exemple une application nécessite un module logiciel en C et un module matériel en VHDL, la communication étant garantie par le standard VSIA ;
- éviter le problème des licences d'utilisation des simulateurs. Il n'est plus nécessaire de disposer des licences correspondantes à tous les simulateurs chez toutes les équipes participant à la conception du SoC.

On trouve dans la littérature beaucoup de travaux sur la simulation des SoC. On peut citer ceux de [Mar02] où est présenté l'outil MCI permettant la cosimulation distribuée de systèmes multilangages. Il permet de simuler un système ayant des modules décrit en VHDL et C, cependant ils doivent être décrits au même niveau d'abstraction. Dans [Nic02] un outil de génération automatique de simulation distribuée, basé sur SystemC, est décrit.

Il permet de simuler des systèmes décrits avec différents langages et à n'importe quel niveau d'abstraction. Par contre l'outil ne permet pas la simulation distribuée géographiquement, et ne cible pas une classe précise de SoC, ce qui rend difficile son utilisation et engendre une mise à jour fréquente de ses bibliothèques vu le nombre de simulateurs qui doivent être supportés. Ainsi notre travail présente plusieurs aspects d'originalité notamment pour l'utilisation d'un bus de simulation distribuée géographiquement en SystemC. Nous proposons la génération automatique d'un environnement de simulation géographiquement distribué, basé sur SystemC et l'assemblage d'IP suivant les standards VSIA (notons que notre objectif est de vérifier la communication entre les modules IP du SoC et non pas leur comportement interne). Cet environnement de simulation distribuée est bien sûr la cible privilégiée de notre environnement Gaspard. La spécification formelle d'architecture d'un SoC à partir d'IP au niveau du modèle « Y » et le placement de l'application en UML permettent la génération d'entrées pour chacun des simulateurs distribués.

3.2 Positionnement et contribution de la thèse

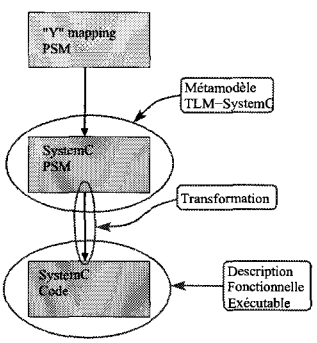


FIG. 3.4 – *Contribution de la thèse*

La contribution apportée par cette thèse à la plateforme Gaspard se situe principalement dans la partie basse du modèle " Y ". En effet, avant ce travail de thèse Garpard permettait uniquement de modéliser des applications des architectures pour réaliser une association. Les techniques de simulation étaient donc exclues. Ainsi, j'ai développé donc dans le cadre de cette thèse un méta modèle SystemC au niveau TLM, et réalisé la transformation automatique à partir de modèles d'association. La génération des modèles de simulation est automatique.

Chapitre 4

Le métamodèle TLM-SystemC™

Dans ce chapitre, je montre les différents concepts nécessaires pour la simulation de système embarqués dans le cadre d'applications de traitement du signal intensif développées en Array-OL.

4.1 Les concepts

Dans cette section, nous allons définir les éléments nécessaires pour la simulation d'une application Array-OL.

4.1.1 Partie logicielle

Array-OL est constitué de deux éléments : les tilers et les tâches. Les tilers permettent d'extraire ou d'écrire des motifs dans un tableau, alors que les tâches permettent d'effectuer un traitement particulier sur ces motifs.

Les tilers

La fonction d'un *Tiler* est la lecture/écriture de motifs depuis/dans un tableau, commme le montre la figure 4.1.

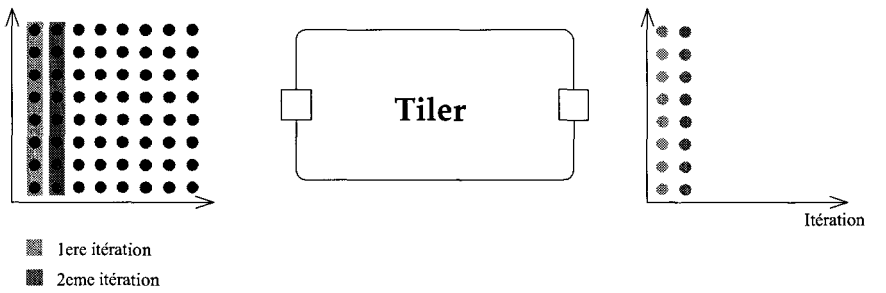


FIG. 4.1 – *Fonctionnement d'un tiler*

Formules d'itération pour un motif

- $\forall \overrightarrow{X_q}, 0 \leq \overrightarrow{X_q} < \overrightarrow{Q}, (\overrightarrow{O} + P \times \overrightarrow{X_q}) \mod \overrightarrow{M}$ donne les coordonnées de l'origine du motif ;
- $\forall \overrightarrow{X_d}, 0 \leq \overrightarrow{X_d} < \overrightarrow{D}, (\overrightarrow{O} + P \times \overrightarrow{X_q} + F \times \overrightarrow{X_d}) \mod \overrightarrow{M}$ donne l'ensemble des coordonnées des points du motif pour l'itération $\overrightarrow{X_q}$.

Pour une itération donnée, le tiler extrait un motif et le place à sa sortie. Pour cela, il faut fournir au tiler un certain nombre d'informations :

- origine (notée $\overrightarrow{O}$),
- taille du tableau (notée $\overrightarrow{M}$),
- matrice de pavage (notée P),
- matrice d'ajustage (notée F),
- limite d'itération de pavage (notée $\overrightarrow{Q}$),
- limite d'itération d'ajustage (notée $\overrightarrow{D}$).

La matrice P permet de calculer l'origine du motif pour une itération donnée. La matrice F définit les différents points de ce motif. Le vecteur $\overrightarrow{D}$ définit le nombre de fois que l'on itère dans chaque dimension pour sélectionner tous les points du motif courant et le vecteur $\overrightarrow{Q}$ définit le nombre de fois que l'on itère dans chaque dimension pour extraire tous les motifs.

La formule 4.1.1 présente les formules pour le calcul des coordonnées des points d'un motif. Pour toute itération $\overrightarrow{X_q}$ dans l'espace d'itération définit par $\overrightarrow{Q}$, on obtient l'origine du motif en multipliant la matrice P par $\overrightarrow{X_q}$ et en ajoutant le vecteur $\overrightarrow{O}$. On utilise le modulo pour ne pas sortir du tableaux d'entrée.

La seconde formule nous montre comment obtenir tous les points du motifs en parcourant l'espace d'itération définit par $\overrightarrow{D}$.

Pour l'exemple de la figure 4.1, les différents paramètres sont les suivants :

- origine : (0,0),
- taille du tableau : (8,8),
- matrice de pavage : (1,0),

- matrice d'ajustage : (0,1),
- matrice Q : (8),
- matrice D : (8).

Ce qui donne, pour une itération $\overrightarrow{X_Q} = (4)$, on obtient l'origine
$\overrightarrow{O}_{\overrightarrow{X_Q}} = \overrightarrow{O} + P * \overrightarrow{X_Q} = (0,0) + (1,0) * (4) = (4,0)$
donc l'origine du motif associé à l'itération $\overrightarrow{X_Q}$ est (4,0).

Pour une itération $\overrightarrow{X_D} = (3)$ en gardant toujours la valeur $\overrightarrow{X_Q} = (4)$, on obtient le point
$\overrightarrow{P}_{\overrightarrow{X_D}} = \overrightarrow{O} + P * \overrightarrow{X_Q} + F * \overrightarrow{X_D} = (0,0) + (1,0) * 4 + (0,1) * 3 = (4,0) + (0,3) = (4,3)$
Le point associé est donc le point (4,3).

La définition même des tilers implique que ceux-ci fonctionnent en paire. S'il existe un ou plusieurs tilers en entrée, il y a necessairement un ou plusieurs tilers en sortie. Un tiler représente en fait un parallélisme potentiel dans l'application comme le montre les calculs précédent. Il n'y pas de dépendance entre les itérations. Pour schématiser, un tiler représente une boucle, on peut choisir de faire n fois l'exécution d'une tâche, ou alors n/m fois l'exécution de m tâches identiques en paralléle.

Les tâches

En Array-OL, les tâches ne prennent, en entrée, que des motifs et génèrent de nouveaux motifs. Un motif est un sous-tableau, voir le tableau entier, sur lequel s'applique un traitement. On peut distinguer deux types de tâches : les tâches élémentaires, qui ne font qu'appliquer une fonction sur les motifs d'entrées et les tâches hiérarchiques.

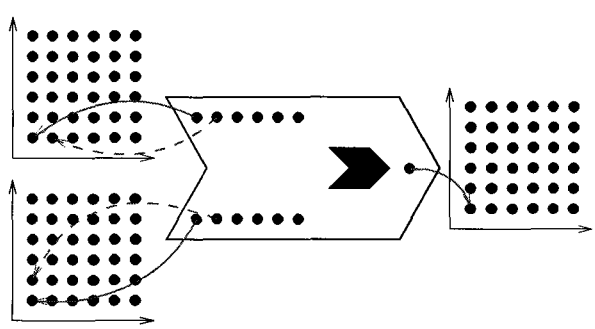


FIG. 4.2 – *Fonctionnement d'une tâche élémentaire Array-OL. Bien que non représentés, il y a trois Tiler permettant l'extraction des données en entrée et l'écriture des données en sortie.*

La tâche élémentaire, représentée à la figure 4.2, consomme une ligne dans chacun de ses tableaux d'entrée (l'extraction des données étant réalisée par des tilers) et produit un motif de dimension 1 (appelé singleton) lui même écrit dans un tableau grâce à un tiler.

Comme les tâches travaillent sur des tableaux complets, il est nécessaire que tous les points du tableaux d'entrée soient calculés pour que la tâche suivante puisse effectuer son traitement.

La figure 4.3 illustre une tâche hiérarchique Array-OL. Les entrées de la tâche sont transmises aux éléments internes de celle-ci. Différents traitements sont effectués par ces composants et, une fois ces traitements terminés, les données de sortie sont retransmises à la sortie de la tâche hiérarchique .

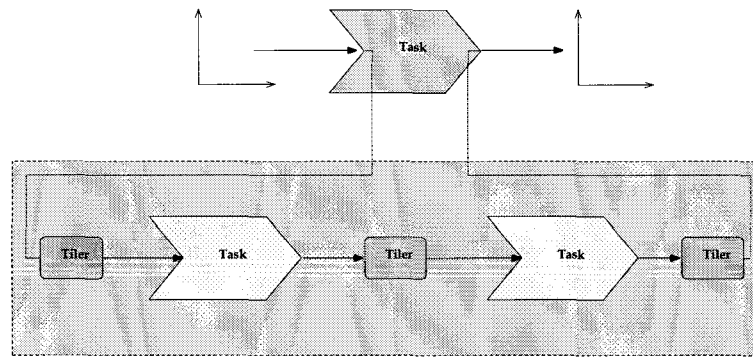


FIG. 4.3 – *Fonctionnement d'une tâche hiérarchique Array-OLcomposée de deux sous-tâches et de trois tilers.*

4.1.2 Partie matérielle

Le but de notre approche est de permettre la conception de systèmes monopuces. Ces systèmes sont, par définition, hétérogénes c'est à dire composés de logiciel et de matèriel. Cette composition implique que notre métamodéle doit comporter ces deux composantes. Or, Array-OL ne comporte pas de concepts matériels. Pour palier à ce manque, nous avons introduit quelques concepts matériels permettant la définition d'architectures matérielles basiques.

Les nouveaux concepts sont :

- processeur,
- mémoire,
- moyen d'interconnexion.

Ces concepts nous permettent de définir un grand nombre d'architectures (grille de processeur, ...). La limite est que l'on ne se permet pas d'avoir un composant dédié (ASIC) ou programmable (FPGA).

Les processeurs

La notion de processeur permet de définir un support physique pour l'exécution de nos différentes tâches. La notion de hiérarchie introduite dans Array-OL étant purement

virtuelle, on considére que l'application ne comporte que des tâches élémentaires et des tilers.

Les processeurs contiennent donc un certain nombre de tâches. Ils doivent permettre et fournir tous les services necessaires à leurs exécutions. Ces services sont l'accessibilité aux mémoires atteignables par ce processeur et la synchronisation lors de l'exécution de tâches interdépendantes sur des processeurs différents.

Les mémoires

Les mémoires permettent le stockage de l'ensemble des tableaux et motifs nécessaire à l'exécution de l'application sur une architecture particuliére.

Les mémoires doivent donc fournir deux services, la lecture et l'écriture d'une donnée.

Les moyens d'interconnexions

Afin de permettre la communication entre les différents éléments de l'architecture, il est nécessaire de définir des moyens d'interconnexions. Il existe trois manières de connecter des éléments matériels :

- point-à-point,
- bus,
- réseau d'interconnexion pour système monopuces (NoC[1]).

la connexion point-à-point permet de réaliser des connexions de manière simple et peu coûteuse. L'inconvénient est qu'il n'y a pas de fléxibilité. Si un des éléments de l'architectures est modifié, cela peut se répercuter sur l'ensemble du système.

Un autre moyen, plus flexible, est le bus. Le bus permet de concevoir le système de manière modulaire. Le changement d'un élément de l'architecture peut imposer l'ajout d'un adaptateur entre le bus et ce nouvel élément mais ne change rien pour les autres éléments. Bien que plus flexible, le bus est un moyen d'interconnexion très couteux. Le coût d'un bus augmente exponentiellement en fonction du nombre de ports.

Enfin, le NoC est un moyen complexe pour interconnecter les éléments de l'architecture. Il peut être taillé sur mesure pour être utilisé dans des systèmes très complexes (+ de 50 IPs). Mais un tel moyen d'interconnexion peut présenter des goulots d'étranglements.

4.2 Le métamodèle

4.2.1 Les concepts

Dans cette section, je vais présenter comment ces différents concepts ont été modélisés dans le métamodèle ainsi que les concepts qui ont été rajoutés afin de permettre la modélisation un système monopuce.

[1]pour Network-on-Chip

La partie « logicielle »

La figure 4.4 montre les concepts de *ComputationTask* et de *Tiler*, héritant tout deux du même concept de *ComputationModule*.

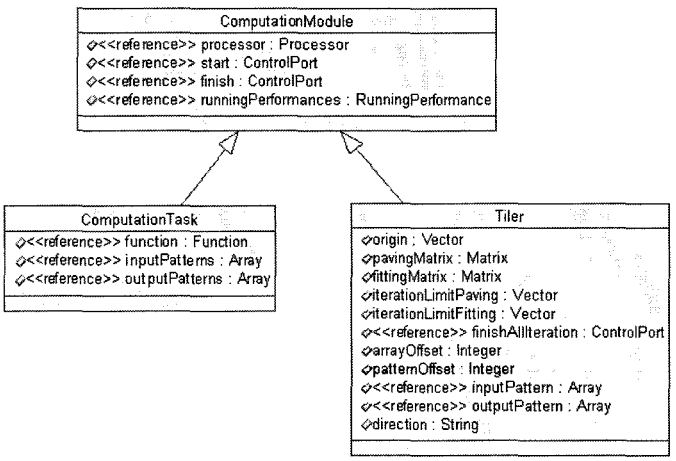


FIG. 4.4 – *Concepts de Tiler et de ComputationTask héritant du même ancêtre ComputationModule*

ComputationTask

Nous pouvons voir qu'il existe, dans la définition du concept de *ComputationTask*, une référence sur un concept nommé *Function* (figure 4.5). Il existe également deux attributs nommés inputPatterns et outputPatterns permettant de savoir, respectivement, sur quelles données le traitement va s'effectuer et de savoir où les résultats vont être placés à la fin du traitement.

Tiler

Il est necessaire de fournir au *Tiler* les informations permettant l'extraction (ou l'écriture) des motifs dans le tableau d'entrée (ou de sortie). Comme le montre la figure 4.4, le tiler posséde tous les attributs définit dans la section 4.1.1.

- origine,
- pavage (Paving),
- ajustage (Fitting),
- limite d'itération de pavage (IterationLimitPaving),
- limite d'itération d'ajustage (IterationLimitFitting),
- décalage tableau (ArrayOffset),
- décalage motif (PatternOffset),
- direction.

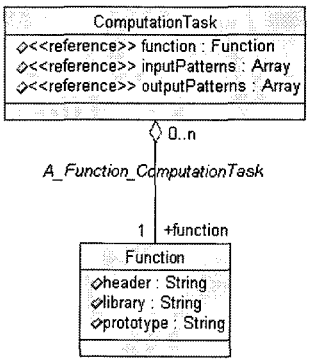


FIG. 4.5 – *Association entre ComputationTask et Function définissant la fonction devant être exécutée*

Les attributs ArrayOffset et PatternOffset permettent de connaître le décallage entre l'adresse 0 de la mémoire où est stocké le tableau et l'origine de celui-ci. L'attribut direction permet de savoir si le tiler est placé en entrée de la tâche ou en sortie.

La partie « matérielle »

Dans cette partie, je présente les différents concepts matériels permettant de définir l'architecture matérielle du système à modéliser.

Main

Ce concept (figure 4.6) permet de définir le système, que l'on souhaite modéliser, dans son ensemble. Il possède un certain nombre d'attributs permettant de connaître tous les éléments matériels de l'architecture. Ces attributs sont :

- ownedProcessors
- owendIntercennects
- ownedMemories
- bridges
- synchronizationSignals

Les attributs ownedProcessors, ownedInterconnects, ownedMemories et bridges contiennent, respectivement, la liste des *Processor*, *Interconnect*, *Memory* et *Bridge* présent dans notre architecture. L'attribut synchronizationSignals contient la liste des signaux permettant la synchronisation inter-processeurs (voir section4.2.2).

Processor

Le concept de *Processor* (figure 4.7) permet de définir le support matériel d'exécution des tâches. Tout module de calcul doit être inclus dans un *Processor* afin d'avoir une base

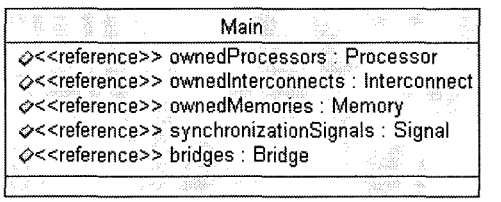


FIG. 4.6 – *Le concept Main.*

d'exécution. En effet, les différents modules logiciels ne peuvent pas directement accéder aux mémoires, il est nécessaire de passer par le processeur pour y accéder.

Nous pouvons voir que le *Processor* possède différents attributs.

- ownedComputationModule
- main
- signals
- size
- runningPerformances

Le premier attribut permet de connaître la liste des *ComputationModule* qui vont être exécutés sur ce *Processor*. Le deuxième permet d'avoir une référence sur le concept racine de notre modèle. Le troisième sert à définir l'ordonnancement des tâches par la création de signaux entre les différents ports de ces tâches. Les attributs « size » et « runningPerformances » seront détaillés à la section 4.3, lors de la présentation de l'analyse de performance.

Memory

Le concept de *Memory* permet de définir les éléments du modèle qui vont être utilisés pour stocker les différents tableaux et motifs nécessaires à l'exécution de l'application. Comme nous pouvons le voir à la figure 4.8, une mémoire possède six attributs :

- main
- capacity
- dataType
- time
- power
- space

L'attribut main permet d'avoir une référence sur la racine de notre modèle. Le second attribut permet de définir le nombre de données de type dataType que l'on peut stocker dans une instance de *Memory*. L'attribut dataType permet de préciser le type de donnée manipulé. Enfin, les trois derniers attributs sont utiles pour l'analyse de performances et seront explicités dans la section 4.3.

La définition du type de donnée dépend du langage cible. Pour SystemC™, il est nécessaire de définir le type de donnée car tous les composants de base SystemC™ sont définis grâce à des patrons qui necessitent la définition du type de donnée.

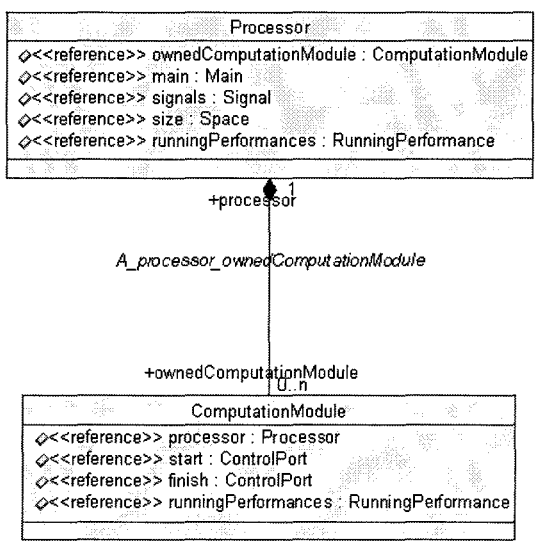


FIG. 4.7 – *Concept de Processor. On peut voir l'association avec le concept ComputationModule permettant de connaître la liste des modules internes.*

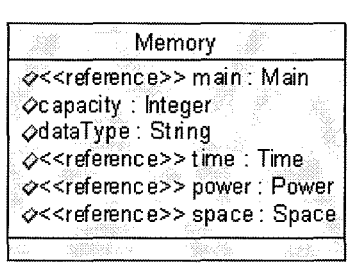


FIG. 4.8 – *Illustration du concept de Memory.*

InterConnect

Maintenant que les concepts de *Processor* et de *Memory* sont introduits dans notre métamodèle, il faut expliciter comment ces deux concepts peuvent communiquer entre eux. Pour cela, j'ai introduit le concept d'*Interconnect* (figure 4.9). Comme précisé dans la section 4.1.2, il existe trois grandes familles de moyens d'interconnexions :

- point-à-point,
- bus,
- réseau d'interconnexion (Noc).

Dans les systèmes actuels, le moyen d'interconnexion le plus répandu est le bus. Mais au vu de la complexité des systèmes à venir, les industriels s'intéressent de plus en plus aux réseaux d'interconnexion (NoC pour Network-on-Chip).

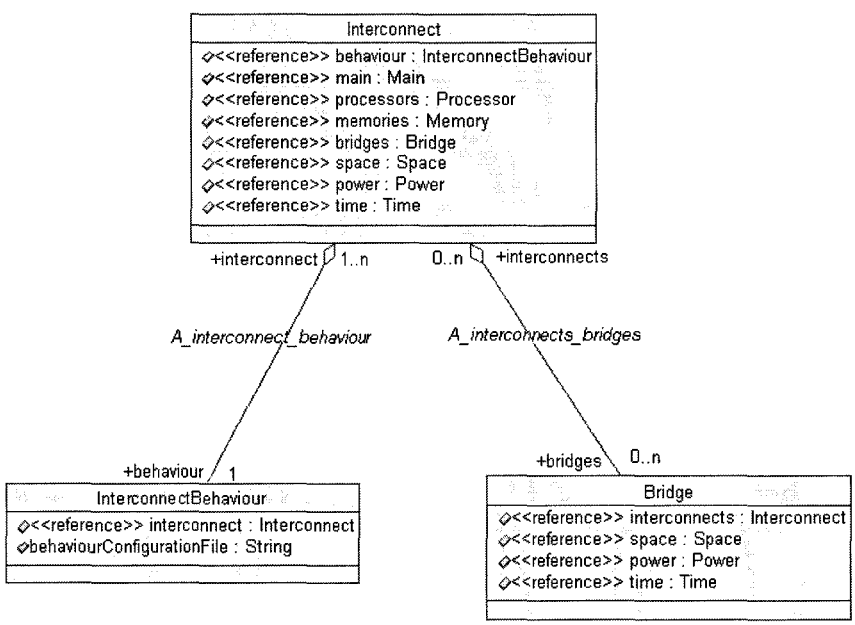


FIG. 4.9 – *Le concept d'Interconnect.*

Le concept d'*Interconnect* possède huit attributs :

- interconnectBehaviour
- main
- processors
- memories
- bridges
- space
- power
- time

L'attribut interconnectBehaviour permet de configurer, lors de la génération du code, le comportement de l'interconnect. Le deuxième permet d'avoir une référence sur la racine du modèle. Les troisième et quatrième attributs permettent de connaître les éléments matériels connectés à l'*Interconnect*. L'attribut bridges permet de savoir les *Bridge* connectés à cet *Interconnect*. Pour terminer, les trois derniers paramètres seront expliqués dans la section 4.3.

Bridge

Un bridge est un composant permettant d'interconnecter deux autres moyens d'interconnexion comme, par exemple, deux bus (voir figure 4.9. Un bridge connaît donc deux *Interconnect* (attribut interconnects) et prend en charge la liaison entre ces deux composants. Il posséde également de attribut permettant l'analyse de performance (attributs space, cost, time et power).

4.2.2 Les concepts d'ordonnancement

Bien que certain systèmes monopuces dédiés au traitement du signal intensif sont d'une grande compléxité en terme de nombre et de nature des composants, une grande partie des systèmes actuels ne contiennent qu'un nombre limité de processeurs.

Ceci implique souvent le déploiement des des différentes tâches sur les processeurs. D'où la nécessité de l'ordonnancement des différentes tâches. Pour cela, j'ai introduit deux nouveaux concepts, permettant de spécifier cet ordonnancement, les concepts de *Port* et de *Signal*.

Port

Pour permettre l'ordonnancement des tâches les unes par rapports aux autres, j'ai ajouté le concept de *ControlPort*. Comme nous pouvons le voir à la figure 4.4, le concept *ComputationModule* possède deux *ControlPort*, un « start » et un « finish ». Grâce à ces deux ports, il est possible d'ordonnancer les tâches entre elles. En plus de ces deux ports, le tiler en possède un troisième permettant de sortir de la boucle lorsque l'espace d'itération $\overrightarrow{Q}$ est terminé.

Un *Port* possède trois attributs, le premier, systemCModule permet de savoir à quel module appartient ce port. Le second, « direction », permet de spécifier s'il s'agit d'un port d'entrée, de sortie ou bidirectionnel. Enfin, l'attribut type permet de définir le type de donnée qui sera véhiculé sur ce port.

Signal

Le concept de *Signal* (figure 4.11) permet de connecter un couple de port permettant à ces deux ports de s'envoyer des informations. Contrairement aux ports, les signaux ne sont pas orientés mais sont typés.

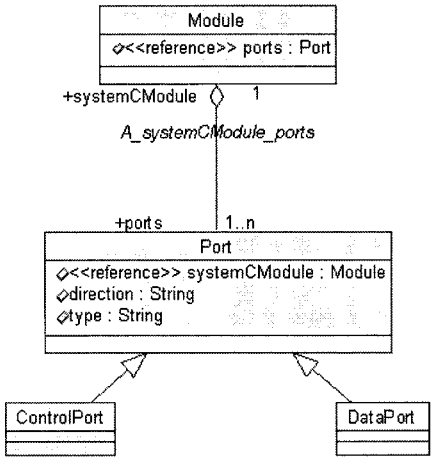


FIG. 4.10 – *Illustration du concept de Port. Nous pouvons voir qu'il existe deux types de Port : les ControlPort et les DataPort*

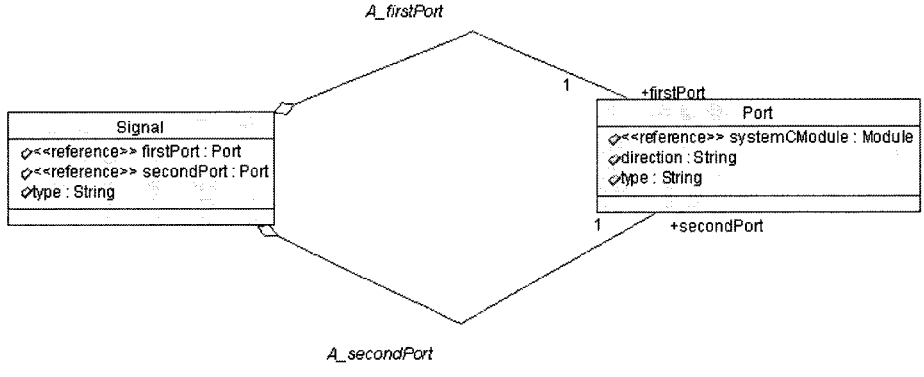


FIG. 4.11 – *Le concept de Signal.*

4.3 L'estimation de performances

Dans le flot de conception des systèmes actuels, la validation de l'aspect applicatif par simulation n'est plus suffisante. Il est également nécessaire d'évaluer les performances de ces systèmes, et cela pour deux raisons : réduire le temps de mise sur le marché et réduire les coûts de développement en évitant des retours en arrière dû à de mauvaises performances du système final.

4.3.1 Les critères

On peut imaginer un très grand nombre de critères de performance pour un système. Tout dépend de son utilisation, de l'environnement où le système évolue, ... Comme nous travaillons dans le cadre des systèmes embarqués, dédiés au traitement de signal intensif, j'ai réduit ce nombre à quatre critères :

- puissance (*Power*)
- taille (*Space*)
- temps d'exécution (*Time*)
- coût (*Cost*)

En parlant de puissance, je veux bien sûr parler de consommation électrique du système. Ce critère est critique dès lors que l'on parle de système embarqué. La plupart de ces systèmes fonctionnent de manière autonome, sur batterie par exemple, ce qui rend primordial la réduction de la consommation d'énergie.

L'évaluation du temps d'exécution du système final est également un point très important dans la conception de systèmes monopuce. En effet, il s'agit d'un paramétre crucial pour la réalisation d'application tel que des codecs video. Pour cela, il me paraissait nécessaire d'ajouter un critère permettant cette évaluation.

Un autre critère important, dans le monde des systèmes embarqués, est la taille. Un des objectifs principaux des concepteurs de systèmes embarqués est de fournir le système le plus performant et tout cela dans un espace le plus réduit possible. Plus le système est petit, plus il peut inclure de fonctionnalités. Voilà pourquoi, j'ai trouvé qu'il pouvait être intéressant de prendre en compte ce critère dans notre modèle d'évaluation de performance.

Pour terminer avec la présentation de ces critères, le critère de coût permet d'évaluer le coût, en terme financier, du système (achat d'IP, ...).

Il est vrai que l'estimation des deux derniers critères peut être fait durant la phase de modélisation, mais il me semble judicieux de les faire tout de même apparaître car ils peuvent être d'une grande utilité si l'on veut faire, par exemple, de l'exploration d'architecture. À performances égales, il est nécessaire d'avoir d'autres critères pour choisir une architecture plutôt qu'une autre.

4.3.2 Intégration dans le métamodèle : Les nouveaux concepts

Dans la section précédente, j'ai présenté les critères que j'avais choisi de faire apparaître afin de permettre de l'évalutation de performance. Dans cette section, je montre comment

j'ai intégré ces différents critères dans le métamodéle. La figure 4.12 montre comment les différents concepts précédemment cités ont été ajouté dans le métamodéle.

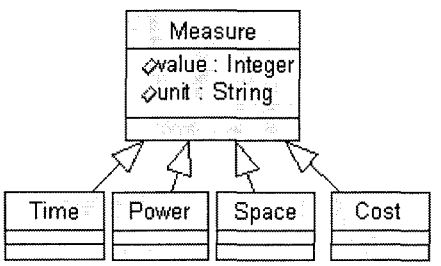


FIG. 4.12 – *Les différents critères d'estimation de performance, tous dérivant du concept Measure*

Le critère **Power**

Le critère de consommation énérgétique a été ajouté aux concepts logiciels et matériels de notre métamodéle. Pour la partie matériels (voir figures 4.7, 4.8 et 4.9), ce critère permet de définir la consommation du système, au repos. Il est caratérisé par l'attribut power. Pour pouvoir estimer la consommation du système, lors de l'exécution de l'application, j'ai ajouté un concept *RunningPerformance* qui sera détaillé plus loin dans ce chapitre.

Le critère **Time**

Contrairement au critère *Power*, le critère de temps s'applique aux concepts de *Memory* et *Interconnect* pour la partie matérielle. Pour la partie logicielle, il s'agit d'un couple *Processor-ComputationModule* qui sera explicité à la section traitant du concept *Running-Performance.*

Le critère **Space**

Le critére d'espace permet d'évaluer la taille du système final. Ce critère ne concerne donc que les éléments matériels de notre métamodéle. Pour cela, un attribut space a été ajouté à chacun des concepts matériels, c'est à dire au *Processor* (figure 4.7), au *Memory* (figure 4.8) et à l'*Interconnect* (figure 4.9).

Le critère **Cost**

Ce critère s'applique à tous les concepts de notre métamodéle. Que ce soit pour la partie matèrielle (achat ISS, ...) que pour la partie logicielle (aaa coût de développement, ...). Pour cela, j'ai ajouté, à chacun des concepts du métamodéle, un attribut *cost.*

Le concept **RunningPerformance**

Le concept *RunningPerformance* permet de définir les différents attributs, pour l'estimation de performance, pour le couple *Processor-ComputationModule*. Il permet de définir le temps d'exécution, la consommation et le coût d'une tâche sur un processeur particulier.

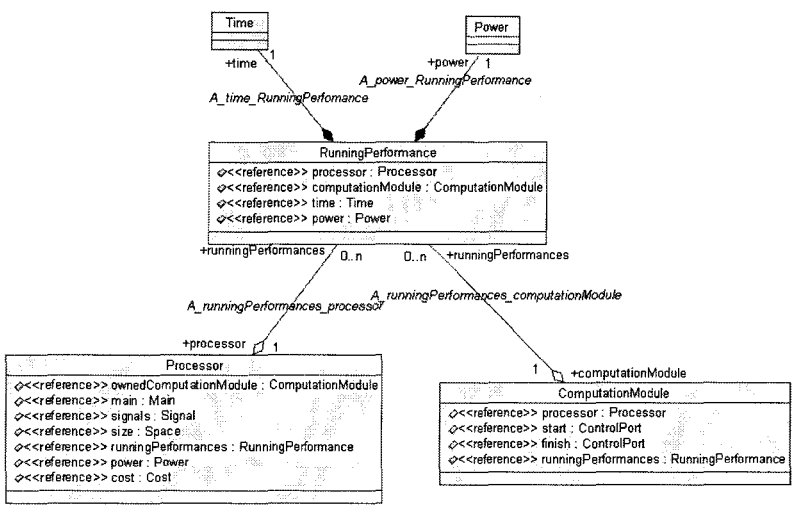


FIG. 4.13 – *Le concept RunningPerformance. Il permet de définr un couple Processor et ComputationModule en ajoutant les critères associés.*

Chapitre 5

Utilisation de la transformation de modèle pour la simulation de systèmes monopuces

Ce chapitre montre comment utiliser la transformation de modèles pour permettre la simulation de systèmes monopuce. Dans une première partie, nous allons voir comment utiliser les transformations de modèles pour générer un code exécutable et ce, quelque soit le langage. Ensuite, je vais présenter comment ce déroule une simulation en respectant Array-OL et le métamodèle. Et pour finir, la méthodologie permettant l'analyse de performances du système modélisé sera présentée.

5.1 Génération de code pour la simulation

Dans cette section, je présente différentes manières de générer le code nécessaire à la simulation d'un système respectant le métamodèle précédemment défini. Dans un premier temps, je présente l'utilisation de ModTransf, un moteur de transformation générique, pour générer ce simulateur. Puis, je vais montrer comment générer un générateur de code

spécifique au langage cible. Enfin je montre les avantages et les inconvénients de ces deux façon de générer le simulateur du système.

5.1.1 Utilisation d'un outil de transformation générique : ModTransf

Dans un premier temps, je montre comment on peut utiliser le moteur de transformation ModTransf pour parcourir le modèle et générer les modules nécessaires à la simulation du système.

Les « patrons » Velocity

Le moteur de transformation est basé sur le générateur de code Velocity. Velocity est un moteur de substitution, basé sur Java. Il permet aux concepteurs de faire référence à des méthodes définies dans du code Java. Ils peuvent ainsi travailler en équipe avec des programmeurs Java pour développer des sites web dans l'architecture MVC (Modèle-Vue-Contrôleur).

La syntaxe Velocity

Dans la table 5.1, je résume les principales commandes du langage Velocity. Étant à la base du moteur de transformation, il est nécessaire de connaître ces commandes afin d'utiliser le moteur dans de bonnes conditions.

Syntaxe Velocity	**Définition**
${Variable} ou $Variable	Variable Velocity
${Variable.propriété}	Accès à une propriété d'une variable
${Variable.méthode()}	Appel de la méthode méthode
#set {$name = "SAMYN"}	Affecte à la variable name la valeur « SAMYN »
\	Le caractère d'échappement
#if (test) ... #elseif (test) ... #else ... #end	La structure conditionnelle
#foreach ($value in $listValue) ... #end	Boucle sur toutes les valeur d'une liste
#include (file, [file, ...])	Inclus un ou plusieurs fichier tel quel
#parse (file)	Inclus un fichier en relançant le mécanisme de génération
#stop	Arrête le processus de génération

TAB. 5.1 – Les principales commandes du langage Velocity

Les « patrons » TLM-SystemC™

À partir du langage de patrons de Velocity, j'ai écrit des patrons spécifiques pour la simulation de système en utilisant ce métamodèle. Pour cela, je suis parti de modules que

j'ai écrit en SystemC™ et j'ai modifié toutes les parties pouvant être générées de manière automatique grâce au générateur Velocity.

Les règles

Tout d'abord, il est nécessaire de définir les règles permettant de parcourir les modèles. Pour chaque concept de notre modèle, il faut définir les différentes actions à effectuer.

```
<rule ruleName="mainFile">
  <description>
    Generate code for the main.cpp file
  </description>
  <leftConditions>
    <concept type ="Main"
     model="sysc"
     use="required"
    />
  </leftConditions>
  <actions>
    <filewriter name="writer"
     filename="main"
     fileExtension=".cpp"
     path="/tmp/generatedSystemC/"
    />
    <generate writer="writer"
     template="templates/Main.vm"
    />
    <closeFilewriter
     name="writer"
    />
  </actions>
</rule>
```

FIG. 5.1 – Exemple de régle pour le moteur ModTransf

Dans le cas de la génération de code, la principale action à effectuer est la génération à partir d'un fichier de « patron » Velocity. La figure 5.1 donne un exemple de règle pour la génération du concept *Main*.

On peut voir que pour générer ce fichier, il faut rencontrer le concept *Main* de notre métamodèle. Une fois que l'on rencontre ce concept, le moteur réalise les actions se trouvant dans la section délimitée par les balises <actions> et </actions>.

Dans la section des actions à effectuer, on demande au moteur de créer un fichier appelé « main », avec l'extension « .cpp », dans de répertoire « /tmp/generatedSystemC ». Puis,

on demande de générer le contenu du fichier en utilisant le fichier de patrons « Main.vm ». Enfin, on demande au moteur de fermer le fichier.

5.1.2 Utilisation d'un moteur de transformation dédié

une autre façon de générer le modéle est l'utilisation des classes générées à partir du métamodèle, en manipulant directement les concepts en Java. Ce générateur de code se compose de deux parties distinctes. Une partie permettant de parcourir les modèles et de lancer les actions spécifiques à chacun des concepts. Une seconde partie permettant de générer les fichiers, un peu dans l'esprit des « patrons » Velocity.

Le parcours de modèle

Le parcours des modèles TLM-SystemC™ se fait par un parcours d'arbre, le métamodèle étant créé pour être parcouru ainsi. On part du concept racine *Main*, puis on parcours chacune des listes contenant les concepts de *Processor*, *Memory*, *Interconnect* et *Bridge*. Puis, pour le concept de *Processor*, on parcourt la liste des *ComputationModule*.

La génération des modules

Pour chacun des concepts, j'ai créé une méthode pour générer les différents fichiers nécessaires à la simulation du système. En plus des méthodes, il a été nécessaire de créer un certain nombre de fichiers supplémentaires, seulement pour des questions de simulation.

5.1.3 Les avantages et inconvénients

Le principal avantage de la génération dédié est la rapidité de la génération, puisqu'il ne s'agit que d'appel de méthode. Le problème de ce type de génération est que, si l'on désire changer le langages, il faut récrire toutes les méthodes. Tandis que, lorsque l'on utilise un moteur générique, il est possible de changer le langage sans avoir à recompiler le générateur.

5.2 La simulation

Dans cette section, nous allons voir comment se déroule la simulation d'un système modélisé grâce à notre métamodèle ainsi que les différentes informations que l'on peut en retirer. Dans cette partie, je présente des modules effectivement développés pour permettre la simulation des systèmes modélisés.

5.2.1 Déroulement de la simulation

Dans cette partie, je présente comment fonctionne la simulation d'un système modélisé à l'aide de notre métamodèle.

Fonctionnement d'un Processor

Un *Processor* est composé de trois parties, la partie contenant les *ComputationModule*, un module spécifique pour réaliser les accès mémoires(MemoryAccess) et un dernier permettant de de réaliser les différentes synchronisations inter processeurs (SynchronizationModule).

Le module MemoryAccess

Ce module permet aux *ComputationModule* de réaliser des accès mémoires sans avoir à se soucier du protocole utilisé. Dans la table 5.2, nous pouvons voir la liste des ports d'un MemoryAccess ainsi que leurs fonctions.

Port	Type de données	Fonction
dataItem	UD[1]	Permet de récupérer les données en mémoire
addressListSource	vector<address>	Liste des adresses des données sources
addressListDestination	vector<address>	Liste des adresses de destination
valideMA	bool	Validation du MemoryAccess
ackMem	bool	Accusé réception des actions mémoires
functionSelect	bool	Sélection du mode (Lecture/Écrit ure)
memoryAddress	address	Adresse de la donnée en mémoire
selectMode	bool	Sélection du mode de transfert
valideMemory	bool	Validation de la mémoire
finish	bool	Fin de l'exécution du MemoryAccess
data4task	vector<UD>	Données pour les tâches
bus_release	bool	Relâchement du bus
bus_request	bool	Requête du bus
grant_access	bool	Autorisation d'accès au bus

TAB. 5.2 – Tableau récapitulatif des ports d'un MemoryAccess

Comme on peut le voir dans cette table, le MemoryAccess a deux modes de fonctionnement : mémoire vers mémoire ou mémoire vers tâche. Pour le premier mode de fonction-

[1] Définit par l'utilisateur

nement, il faut fournir au MemoryAccess la liste des adresses source et destination. Cela permet au module de réaliser les différents accès mémoire nécessaires au déplacement des données. Ce mode est utilisé pour les *Tiler*. Le second mode de fonctionnement permet de transférer des données d'une tâche vers les mémoires ou inversement. Pour cela, il suffit de fournir au module la liste des adresses des données sur le port addressListSource et de sélectionner la lecture/écriture via le port functionSelect. La sélection du mode que l'on désire utiliser se fait à travers le port selectMode.

Algorithme 1 – Fonctionnement d'un *MemoryAccess*

tant que simulation continue **faire**
  Attendre le signal sur le port valideMA
  si selectMode == VRAI **alors**
    Cas 1
  sinon
    Cas 2
  fin si
fin tant que

Le module SynchronizationModule

Le module SynchronizationModule permet de réaliser les synchronisations entre les différents processeurs du système. La table 5.3 montre les différents ports de notre module.

Port	Type de donnée	Fonction
processorSynchronization	SynchronizationSignal	Permet d'émettre/recevoir des signaux de synchronisation
can_i_start	bool	Demande d'autorisation de démarrage pour un *ComputationModule*
you_can_start	bool	Autorisation de démarrage
send_synchro	bool	Demande d'envoi de synchronisation

TAB. 5.3 – Tableau récapitulatif des ports d'un SynchronizationModule

Lorsqu'une tâche doit démarrer, celle-ci envoie un signal sur le port can_i_start. Le module de synchronisation vérifie alors qu'il a reçu tous les signaux de synchronisation, s'il existent, pour démarrer cette tâche. Si oui, il envoie un signal sur le port you_can_start sinon la tâche est mise en attente jusqu'à ce que le signal soit envoyé. Cette vérification est faite

à chaque fois que le module de synchronisation reçoit un signal sur le port processorSynchronization. A la fin de son exécution, si la tâche doit envoyer un signal de synchronisation vers un autre processeur, elle envoie un signal sur le port send_synchro. À ce moment là, le module se charge d'émettre le signal vers les autres processeurs.

Le module Processor

Dans la table 5.4, on peut voir les ports « visibles » de l'environnement extérieur à notre processeur.

Port	Type de donnée	Fonction
dataItem	UD	Idem que table 5.2
ackMem	bool	Idem que table 5.2
memoryAddress	address	Idem que table 5.2
valideMemory	bool	Idem que table 5.2
bus_request	bool	Idem que table 5.2
bus_release	bool	Idem que table 5.2
grant_access	bool	Idem que table 5.2
processorPort	SynchronizationSignal	Idem que table 5.3
start	bool	Permet de démarrer le module
finish	bool	Signale la fin d'exécution

TAB. 5.4 – Tableau récapitulatif des ports d'un Processor

Le signal start permet de faire démarrer le module Processor. Une fois ce signal reçu, il est immédiatement transmis à la première tâche de la liste d'exécution. Une fois le signal transmis, les différents modules internes interagissent entre eux et effectuent les accès mémoires via les ports « visibles » du Processor ainsi que les synchronisations entre les différents processeurs.

Fonctionnement d'un Memory

Le module Memory permet de stocker des données nécessaires à notre système. Pour cela, le module permet de lire/écrire des données dans un tableau interne. Les accès se font grâce aux différents ports du module, décris dans la table 5.5.

Pour initier un accès mémoire, il faut :

- Écrire sur le port memAddress l'adresse désirée
- Écrire sur le port read :
  - la valeur true pour lire une donnée
  - la valeur false pour écrire une donnée
- placé la donnée sur le port data (en cas d'écriture)
- Écrire la valeur true sur le port validate

Port	Type de donnée	Fonction
memAddress	address	Permet de spécifier à quel emplacement on désire écrire
data	UD	La donnée à écrire ou lu
read	bool	Permet de spécifier le mode lecture/écriture
validate	bool	Valide les accès à la mémoire
ack	bool	Accusé réception

TAB. 5.5 – Tableau récapitulatif des ports d'un Memory

- Attender le signal ack
- Récupérer la donnée (en cas de lecture)

Ce module permet de simuler une mémoire monoport, avec une seule lecture/écriture par activation. Même si en pratique on peut rencontrer des mémoires multi ports, les mono port restent quand même tres largement utilisées et permettent la conception de la quasi-totalité des architectures matérielles. Ceci est encore plus vrai dans le contexte embarqué car si un besoin de mémoires multi port se présente, il suffira au concepteur de rajouter un adaptateur matériel à la mono port, offrant ainsi plusieurs ports logiques d'accès à la mémoire [Mef02].

Fonctionnement d'un Interconnect

Vu les manières diverses d'interconnecter différents modules matériels, j'ai écrit un module permettant de simuler un bus à file d'attente. La table 5.6 décrit les ports nécessaires pour l'utilisation de ce bus pour effectuer une simulation.

Lorsqu'un processeur désire effectuer une communication, il envoie d'abord un signal sur le port bus_request. Une fois le bus disponible, il envoie au processeur un signal pour lui signifier la disponibilité du bus via le port bus_grant_access. Dès la reception de ce signal, le processeur positionne les différents signaux sur les ports data_master, address_master, read_not_write_master et valide_slave conformément à ce qui à été dit précédement. Enfin le processeur attend l'accusé réception de la mémoire sur le port slave_acknowledgement_master.

Du coté du bus, lorsqu'il reçoit un signal bus_request, il vérifie s'il est occupé ou non. Si oui, la demande est ajoutée dans la file d'attente sinon, le bus valide la demande par l'envoi d'un signal sur le port bus_grant_access. Une fois que la communication peut être effectuée, le bus attend le signal valide_slave. Il retransmet les signaux de contrôle sur les ports coté mémoire. En plus de cela, il effectue un décodage d'adresse pour savoir sur quel esclave la requête doit être envoyée. Une fois que l'esclave à terminer son activité, il renvoie

Port	Type de donnée	Fonction
bus_request	bool	Permet de spécifier à quel emplacement on désire écrire
bus_grant_access	bool	La donnée à écrire ou lu
bus_release	bool	Permet de spécifier le mode lecture/écriture
bus_select_slave	bool	Valide les accès à la mémoire
data_master	UD	Accusé-réception
address_master	address	Accusé-réception
read_not_write_master	bool	Accusé-réception
slave_acknowledgement_master	bool	Accusé-réception
valide_slave	bool	Accusé-réception
data_slave	UD	Accusé-réception
address_slave	address	Accusé-réception
read_not_write_slave	bool	Accusé-réception
acknowledgement_slave	bool	Accusé-réception

TAB. 5.6 – Tableau récapitulatif des ports d'un Bus

un signal sur le port acknowledgement_slave qui est aussitôt retransmis à l'initiateur de la communication qui peut, soit libérer le bus, soit initier une autre communication.

Fonctionnement de ComputationModule

Maintenant que l'on sait comment fonctionne la partie matérielle de notre système, il ne reste plus qu'à s'intéresser à la partie logicielle. Dans un premier temps, je vais présenter en détails comment fonctionne un ComputationTask. Puis je vais présenter le fonctionnement du *Tiler*.

ComputationTask

Un module permettant la simulation d'un *ComputationTask* permet d'exécuter une tâche « élémentaire » présise. Par élémentaire, je veux dire qu'il n'y a pas de tâches Array-OL à l'intérieur de notre tâche. Il s'agit d'une fonction C/C++,n pouvant faire appel à des classes, des structures, accéder aux disques, ...

Dans la table 5.7, les différents ports d'un *ComputationTask* sont représentés.

L'algorithme 2 nous présente le fonctionnement d'un module *ComputationTask*. Lors de la réception su signal sur le port start, le module demande l'autorosation de démarrer au SynchronizationModule, en envoyant un signal sur le port can_i_start. Une fois que la réponse lui est donnée, via le port you_can_start, le module cherche à récupérer les données qui lui sont nécessaires pour son exécution. Pour cela, il demande au MemoryAccess, en envoyant

Port	Type de donnée	Fonction
start	bool	Permet de démarrer la tâche
finish	bool	Signale la fin d'exécution
can_i_start	bool	Demande d'autorisation de démarrage
you_can_start	bool	Autorisation de démérrage
synchro	bool	Demande d'envoi d'une synchro
inputPort	vector<UD>	Données en entrée
outputPort	vector<UD>	Données en sortie
patternALS	vector<address>	Adresses des données d'entrée
patternALD	vector<address>	Adresses des données de sortie
valideMemoryAccess	bool	Validation du MemoryAccess
functionSelect	bool	Séléction du mode lecture/écriture
finishMA	bool	Signale la fin d'exécution du MemoryAccess

TAB. 5.7 – Tableau récapitulatif des ports d'un ComputationTask

Algorithme 2 – Fonctionnement d'un *ComputationTask*

tant que simulation continue **faire**
Attendre le signal sur le port start
Envoyer un signal sur le port can_i_start
Attendre un signal sur le port you_can_start
Ecrire les adresses du motif d'entrée sur le port PatternALS
Selectionner le mode lecture du *MemoryAccess* grâce au port functionSelect
Activer le module *MemoryAccess* en envoyant un signal sur le port valideMemoryAccess
Attendre le signal de terminaison du *MemoryAccess* via le port finishMA
Récupérer le motif d'entrée sur le port inputPort
Effectuer le traitement sur le motif
Ecrire le motif de sortie sur le port outputPort
Selectionner le mode écriture du *MemoryAccess* via le port functionSelect
Valider le *MemoryAccess*
Attendre la terminaison de l'exécution du *MemoryAccess* via le port FinishMA
Envoyer le signal de synchro via le port synchro
Envoyer un signal sur le port finish
fin tant que

différents signaux sur les port patternALS, functionSelect et valideMemoryAccess. Une fois toutes les données récupérées, le traitement peut être effectué. Une fois cette opération terminée, le module écrit les motifs résultats, via le module MemoryAccess, en jouant sur les signaux patternALD, functionSelect et valideMemoryAccess. Une fois toutes les données écrites en mémoire, le module envoie, si nécessaire, un signal de synchro et se termine en envoyant un signal sur le port finish.

Tiler

Le tiler a un fonctionnement très similaire à celui des tâches. La table 5.8 donne la liste des ports d'un tiler. Comme l'on peut le voir dans cette table, il y a un certain nombre de ports en commun avec la tâche mais il n'y a aucun port de données. Le tiler déplace des données d'un emplacement mémoire à un autre sans les manipuler directement. Ces manipulations se font grâce au MemoryAccess.

Port	**Type de donnée**	**Fonction**
start	bool	Permet de démarrer la tâche
finish	bool	Signale la fin d'exécution
can_i_start	bool	Demande d'autorisation de démarrage
you_can_start	bool	Autorisation de démérrage
synchro	bool	Demande d'envoi d'une synchro
ALS	vector<address>	Adresses des données d'entrée
ALD	vector<address>	Adresses des données de sortie
valideMemoryAccess	bool	Validation du MemoryAccess
finishMA	bool	Signale la fin d'exécution du MemoryAccess
functionSelect	bool	Séléction du mode lecture/écriture
finishAllIteration	bool	Signale la fin de parcours de l'espace d'itération

TAB. 5.8 – Tableau récapitulatif des ports d'un Tiler

Voyons maintenant comment se déroule l'exécution d'un tiler. Tout comme la tâche, le tiler attend un signal sur le port start et vérifie la possibilité de commencer grâce au signal can_i_start. Une fois l'exécution commencé, en fonction de sa position (tiler d'entrée ou de sortie), il calcule les coordonnées des points qu'il doit lire/écrire, et crée un vecteur contenant toutes les adresses de ces points. S'il s'agit d'un tiler d'entrée, il écrit ce vecteur

sur le port ALS. S'il s'agit d'un tiler de sortie, il l'écrit sur le port ALD. Sachant que les motifs ont une place fixe et connue en mémoire.

5.3 L'analyse de performance

Dans cette section, je présente comment l'analyse de performance a été intégré lors de la génération du simulateur, et comment on peut récupérer les différentes informations nécessaires pour pouvoir les traiter.

5.3.1 Évaluation des paramétres Cost et Size

Ces deux paramétres ne sont pas fonction de l'exécution du système. Ce qui permet lors de la phase de génération ou d'initialisation d'évaluer directement ces valeurs. Pour le critère de taille, une fois que l'on connait tous les éléments matériels de notre architecture, il est facile de connaître la valeur de ce paramètre. Il suffit, lors de la génération, de parcourir l'ensemble des éléments matériels et d'additionner les valeurs des paramètres *Size* (voir algorithme 3).

Algorithme 3 – Évaluation du paramétre *Size*

```
taille_global ← 0
pour tout element dans element_materiels faire
  si element.taille() <> NULL alors
    taille_global ← taille_global + element.taille()
  fin si
fin pour
```

Pour le paramétre *Cost*, l'évaluation est un peu plus complexe. Par exemple, le coût de développment d'une application sur un processeur plutôt que sur un autre est différent. Pour cela, il faut regarder pour chaque tâche placée sur un processeur si on connaît les informations de coût de développement. C'est à dire s'il existe un *RunningPerformance* nous donnant cette information entre la tâche et le processeur (voir algorithme 4).

5.3.2 Évaluation du paramétre Power

L'évalutation du critère de consommation énergétique est un peu plus complexe. Il existe une valeur de base de consommation, à laquelle on ajoute une surconsommation liée à l'utilisation de cette ressource. Pour évaluer la consommation énergétique, on réalise un graphe de consommation pour chacun des éléments de l'architecture. Ce qui permet de faire un retour au concepteur. Les différents graphes sont visualisables en utilisant un logiciel comme gnuplot[1].

[1]Téléchargeable à l'adresse suivante `http://www.gnuplot.info`

Algorithme 4 – Évaluation du paramétre *Cost*

```
global_cost ← 0
pour tout element dans hardware_elements faire
  si element.cost() <> NULL alors
    global_cost ← global_cost + element.cost()
  fin si
fin pour
pour tout processor dans processor_list faire
  pour tout cm dans processor.ownedComputationModule faire
    si exist_running_performance(processor, cm) alors
      global_cost ← global_cost + running_performance(processor, cm).cost()
    fin si
  fin pour
fin pour
```

Memory

Pour les mémoires, on suppose que la consommation engendrée par un accès mémoire est constant, sur une méme mémoire. De ce fait, il est possible d'évaluer la consommation énérgétique de la mémoire grâce à l'algorithme 5.

Algorithme 5 – Évaluation du paramétre *Power* pour le concept *Memory*

Requis: Consommation de base notée W_{base}
Requis: Consommation d'un accès mémoire notée $W_{\text{accès}}$
Requis: Fichier externe $fichier\_consommation$

```
pour tout top d'horloge t faire
  si accès_mémoire alors
    fichier_consommation ← t : W_base + W_accès
  sinon
    fichier_consommation ← t : W_base
  fin si
fin pour
```

Grâce à cet algorithme, on génére un fichier qui, pour chaque activation, donne la consommation du module.

Interconnect

Comme souligné à plusieurs reprises, il existe différents moyens d'interconnexion. Ces différents moyens ne mettent pas en oeuvre les mêmes mécanismes ce qui entraîne une certaine difficultée dans l'estimation de la consommation énergétique. Le seul point commun

entre ces différents moyens d'interconnexion est le fait de transiter des informations. C'est pour cela que j'ai défini un paramétre **power** définissant la consommation nécessaire au transfert d'une information via ce moyen.

Pour estimer la consommation énergétique des moyens d'interconnexions, cela se déroule un peu comme pour les mémoires, comme le montre l'algorithme 6.

Algorithme 6 – Évaluation du paramétre *Power* pour le concept *Interconnect*

Requis: Consommation de base notée W_{base}
Requis: Consommation d'une communication notée $W_{\text{communication}}$
Requis: Fichier externe $fichier\_consommation$
  pour tout top d'horloge t **faire**
    si communication **alors**
      $fichier\_consommation \leftarrow t : W_{base} + W_{\text{communication}}$
    sinon
      $fichier\_consommation \leftarrow t : W_{base}$
    fin si
  fin pour

Pour chaque instant t, on regarde s'il y a une communication, si oui alors on écrit dans le fichier la valeur $t : W_{base} + W_{\text{communication}}$ permettant d'évaluer la consommation de la communication. Sinon, on insére la valeur $t : W_{base}$ pour exprimer le fait que le moyen d'interconnexion est au repos.

Processor

Pour les processeurs, la consommation énergétique dépend de la tâche qui est exécutée. Pour chaque tâche exécutée, il faut trouver la consommation associé à ce processeur pour cette tâche afin d'évaluer la consommation à cet instant. L'algorithme 7 présente comment est faite l'évaluation de la consommation pour un processeur.

Pour obtenir une évaluation correcte de la consommation, il est nécessaire de fournir une évaluation des performances pour chaque tâche et ce, pour chaque processeur.

5.3.3 Évaluation du paramétre Time

L'estimation du critère *Time* se fait par une analyse des fichiers de traces générés par le simulateur (voir section 2.1.2). De ces fichiers, on peut extraire des informations capitales telles que le temps d'exécution de chacune des tâches, dépendant des accès mémoire et de la charge du bus, la charge de chacun des éléments matériels de l'architecture.

Algorithme 7 – Évaluation du paramétre *Power* pour le concept *Processor*

Requis: Consommation au repos notée W_{repos}
Requis: Fichier externe $fichier\_consommation$
pour tout top d'horloge t **faire**
si tâche à exécutée (notée T) **alors**
pour tout runningPerformance **dans** proc.runningPerformance **faire**
si runningPerformance.computationModule $= T$ **alors**
$fichier\_consommation \leftarrow t : W_{\text{repos}} + runningPerformance.power$
fin si
fin pour
sinon
$fichier\_consommation \leftarrow t : W_{\text{repos}}$
fin si
fin pour

Memory

Tout comme l'analyse de la consommation d'énergie, on suppose que le temps d'accès à une donnée en mémoire est constant. De ce fait, il est facile d'évaluer le temps d'exécution pour les différentes mémoires.

Interconnect

.L'estimation du temps d'exécution des moyens d'interconnexion est aussi difficile que l'évaluation de la consommation du fait de leurs diversités. Pour évaluer ce critères, j'ai donc ajouté un paramétre permettant une évaluation de tout le mécanisme de communication et ce, quelque soit ce mécanisme.

Processor

Pour les processeurs, le temps d'exécution dépend des tâches qui doivent être exécutées. Tout comme les autres critéres d'évaluation, les paramétres sont définis dans le concept *RunningPerformance*.

Chapitre 6

Expérimentation

ans ce chapitre, je présente un exemple d'application de la chaîne de conception Gaspard et montrer les différentes informations que l'on peut retirer d'une simulation. Dans une première partie, je présente l'application, l'architecture et le placement choisi. Puis nous allons voir les données permettant d'analyser les performances du système.

6.1 L'application

6.1.1 Partie logicielle

Dans ce chapitre, l'expérimentation est faite sur une application de traitement d'image. Il s'agit d'un « scaler », cette application permet de réduire une image 512x512 pixels en une image 256x256 pixels. Pour calculer la valeur du pixel, courant de l'image de sortie, on calcul la moyenne de quatre pixels contigüs de l'image d'entrée (voir l'algorithme 8).

De cette application, on peut retirer une représentation Array-OL. En effet, on peut voir que toutes les itérations de cette application sont indépendantes. On peut donc calculer tous les pixels de l'image en sortie, en utilisant 256x256 processus. La représentation Array-OLde cette application se trouve à la figure 6.1.

Pour définir complétement l'application Array-OL, il faut calculer les différents paramétres présentés à la section 4.1.1.

Algorithme 8 – Algorithme du « scaler »

pour i **de** 0 **à** 511 **par pas de** 2 **faire**
  pour j **de** 0 **à** 511 **par pas de** 2 **faire**
    Extraire les pixels ({i,j}, {i+1,j}, {i,j+1}, {i+1,j+1})
    Calculer la valeur $\frac{\sum p_i}{4}$
    Écrire la valeur dans l'image de sortie à la coordonnée $(\frac{i}{2}, \frac{j}{2})$
  fin pour
fin pour

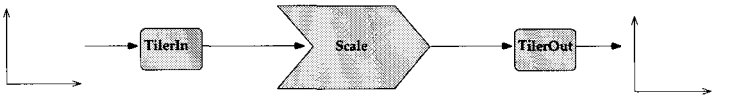


FIG. 6.1 – *Application « Scaler » en Array-OL*

Les valeurs de ces différents paramétres sont :
- Tiler d'entrée
  - Origine : $\overrightarrow{O} = \{0, 0\}$
  - Matrice de Pavage : $P = \{\{2, 0\}, \{0, 2\}\}$
  - Matrice d'Ajustage : $F = \{\{1, 0\}, \{0, 1\}\}$
  - Limite d'itération de Pavage : $\{256, 256\}$
  - Limite d'itération d'Ajustage :$\{2, 2\}$
  - Dimension du tableau d'entrée : $\{512, 512\}$
- Tiler de sortie
  - Origine : $\overrightarrow{O} = \{0, 0\}$
  - Matrice de Pavage : $P = \{\{1, 0\}, \{0, 1\}\}$
  - Matrice d'Ajustage : $F = \{\}$
  - Limite d'itération de Pavage : $\{256, 256\}$
  - Limite d'itération d'Ajustage : $\{\}$
  - Dimension du tableau de sortie : $\{256, 256\}$

6.1.2 Partie matérielle

Le support d'exécution pour cette application est composé de 6 composants. Deux processeurs, trois mémoires et un bus permettant de connecter les processeurs aux mémoires. La sélection de la mémoire se fait au niveau du bus. Le bus décode l'adresse fournit par le processeur afin de choisir le bon banc mémoire.

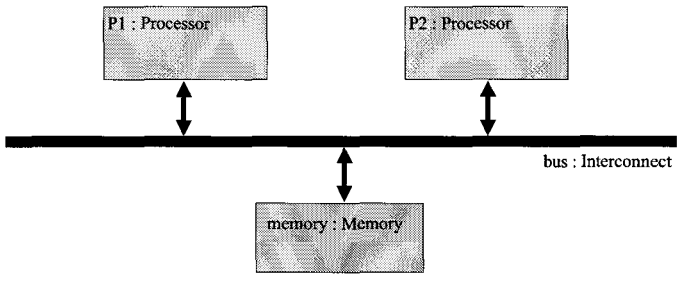


FIG. 6.2 – *Architecture cible*

6.1.3 Le placement

Il est possible d'envisager plusieurs maniéres de placer cette application sur une architecture biprocesseur. On peut, d'une part placer l'application telle quelle sur l'architecture. D'autre part, on peut placer deux, ou plusieurs, instances en découpant l'espace d'itération en morceaux. Ce découpage est rendu possible car toutes les itérations sont indépendantes.

Le placement direct de l'application sur l'architecture nous impose de placer les deux tilers et la tâche élémentaires sur les deux processeurs disponibles.

Pour cela, il faut faire un choix. Dans ce cas précis, j'ai choisi de placer les deux tilers sur un même processeur et la tâche sur le second (voir figure 6.3).

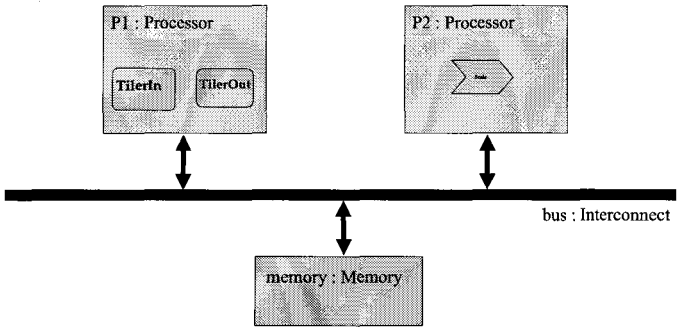


FIG. 6.3 – *Placement Simple*

Il s'agit d'un placement basique, ne demandant aucun prétraitement de l'application. Bien entendu, toutes les données sont stockées dans la mémoire de notre architecture, aussi bien les images d'entrée que celles de sortie, mais également les motifs nécessaires à l'exécution de application Array-OL.

6.1.4 L'ordonnancement

Une fois les composants logiciels placés, il faut ordonnancer ces différents composants pour assurer exécution correcte de l'application. Pour cela, lorsque l'ordre d'exécution

n'est pas spécifié, il faut rajouter différentes dépendances pour assurer l'ordre d'exécution totale sur les différents processeurs. Une fois cet ordre total obtenu, il faut synchroniser les différents processeurs lorsque cela est necessaire afin de garantir l'ordre d'exécution global de l'application.

Voyons cela sur notre exemple.

6.2 Simulation

Maintenant que l'application est placée et ordonnancée sur l'architecture cible, il est possible de générer le code nécessaire à la simulation.

6.2.1 Le code généré

A partir de ce modèle, la génération de code se fait de manière automatique. Dans la table suivante, nous pouvons voir la liste des fichiers générés ainsi que leur utilité.

Nom du fichier	**Utilité**
tiler.h	La classe générique Tiler
memoryaccess.h	Le module gérant les accès mémoires des processeurs
memory.h	Le module mémoire
synchro.h	Le module de synchronisation
bus.h	Le module d'interconnexion
data_proc_one.h	Les données d'initialisation du processeur 1
data_proc_two.h	Les données d'initialisation du processeur 2
datatype.h	Définition des types de données
file.h	Le module de gestion de file
identification.h	Le module permettant l'identification des composants
iteration.h	Le module gerant les itérations
matrix.h	Le module de calcul matriciel
proc_one.h	Un module processeur
proc_two.h	Un module processeur
synchro_signal.h	Les signaux de synchronisation
synchro_table.h	La table de synchronisation
task.h	Le module de tâche élémentaire

TAB. 6.1 – Les fichiers générés

6.2.2 Le programme

Nous pouvons voir le résultat de l'application à la figure 6.4.

FIG. 6.4 – *A gauche, l'image originale d'une taille de 512x512 pixels, à droite l'image réduite à 256x256 pixels*

La figure de droite représente l'image source de notre application, celle de gauche montre le résultat du traitement de l'application. Le temps d'exécution de la simulation d'une telle application est d'environ une trentaine de seconde sur une machine de puissance moyenne (P4 2.4GHz, 512 Mo de RAM) et ce sur un système linux (une distribution linux gentoo).

6.2.3 Analyse de performance

Les évaluations

Pour cet exemple, j'ai choisi les valeurs suivantes pour chacun des paramétres de consommation.

Composant	**Valeur**
Processor1	5
Processor2	10
Memoire1	5
Memoire2	5
Memoire3	5
TilerIn	25
TilerOut	25
Scale	10
MemAcc	5

TAB. 6.2 – Paramètre de consommation

Les cinq premières valeurs du tableau 6.2 correspondent aux valeurs au repos de chacun des composants. Les valeurs TilerIn, TilerOut, Scale et MemAcc correspondent respective-

ment aux valeurs de calcul du tiler d'entrée, du tiler de sortie, de la fonction Scale et enfin d'un accès à une donnée en mémoire.

Le système

Entrons maintenant dans le détails de la simulation. La figure 6.5 montre une capture d'écran des traces obtenues lors de la simulation, visualisée grâce au logiciel GTKWave.

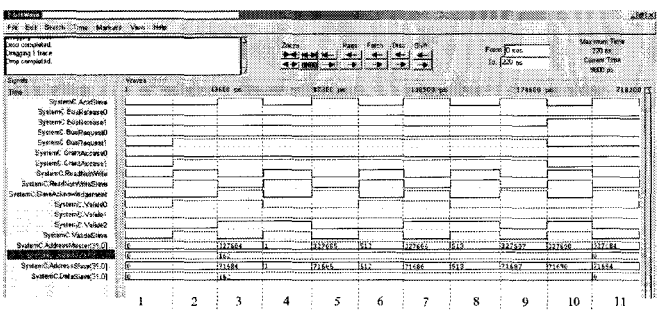


FIG. 6.5 – *Extrait de la trace d'exécution de l'application*

Dans la section 1, tous les signaux sont à leurs valeurs initiales. Par défaut, tous les signaux générés en utilisant SystemC™ sont mis à la valeur 0. Dans la section 2, le processeur 1 demande un accés au bus en placant le bit BusRequest à 1. Comme aucun temps de communication n'a été affecté au bus, la réponse est immédiate, ce qui permet au processeur de d'écrire immédiatement toutes les données sur les différents ports. Il demande l'accés à l'emplacement 0 de la mémoire, place les valeurs appropriés sur les ports ReadNotWrite, AddressMaster et ValideSlave. Instantanément, le bus convertit l'adresse recu sur le port AddressMaster et le converti pour selectionner le banc mémoire sur lesquel se trouve les données et valide ce banc mémoire (dans ce cas le banc numéroté 0). A la section 3, on voit que le processeur reçoit la donnée sur le port DataMaster (valeur 162 en décimal) et il reçoit également le signal AckSlave pour l'informer que la demande est terminée. Aussitôt, le processeur copie cette donnée à l'emplacement mémoire 327684, qui est l'emplacemement mémoire où est stocké le motif. L'adresse 327684 est convertie en l'adresse 71684 du banc mémoire numéro 2. Ces différents transfert de donnée continue ainsi jusqu'à obtention du motif complet. Puis la tâche déployé sur le deuxième processeur peut commencer ses traitement (non représenté sur la figure).

La figure 6.6 présente le graphe que l'on obtient aprés la simulation. Les données sont récupérées dans un fichier texte qui sont ensuite traité afin obtenir des données visulisable dans un graphe. En abcisse de ce graphe, nous avons le temps de simulation et en ordonnée la consommation du système.

A partir de ce graphique, il est possible de calculer un certain nombre de valeurs statistiques permettant de caractériser le placement de l'application sur l'architecture. Par exemple, il est possible de calculer la valeur moyenne de consommation. Cette valeur per-

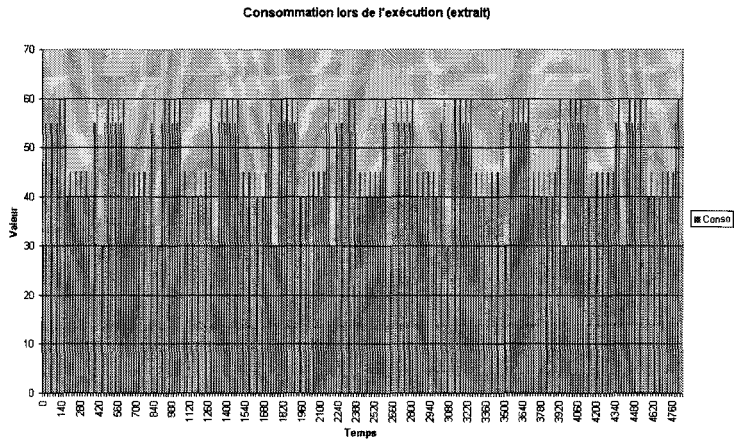


FIG. 6.6 – *Extrait du graphe de performance de l'application*

mettant de comparer deux placements différents entre eux. Il peut également être interessant de calculer les valeurs minimale et maximale de consommation. Ces valeurs permettant de comparer deux placement ayant des valeurs moyenne identique. En effet, dans le cas de système embarquè, il est interessant d'avoir le moins de variation de consommation, est donc de température.

Les composants

Comme pour le système, il est possible d'analyser la consommation par composant materiel de l'architecture. Dans les figures 6.7 et 6.8, nous pouvons voir le graphe de consommation pour le processeur contenant la fonction de calcul « Scale » et le graphe de consommation pour le processeur contenant les deux tilers.

Sur ces graphes, on peut remarquer que le premier processeur n'est en fonction que très peu de temps, alors que le second est très souvent en fonctionnement. Pour palier à cette différence, le concepteur peut, par exemple transférer l'exécution du tiler de sortie du second processeur sur le premier.

6.3 Conclusion

J'ai montré dans ce chapitre la simplicité d'application des différentes étapes de notre méthodologie sur un exemple de traitement de signal. En effet, le code a été généré de façon complétement automatique. L'analyse des résultats de la simulation, sous forme de graphiques produits automatiquement, permet une estimation rapide de la consommation en énergie engendrée par l'application, l'architecture et le placement choisi. Ces résultats étant obtenus en un temps court, il est possible à l'utilisateur de les utiliser pour une exploration de placements efficaces.

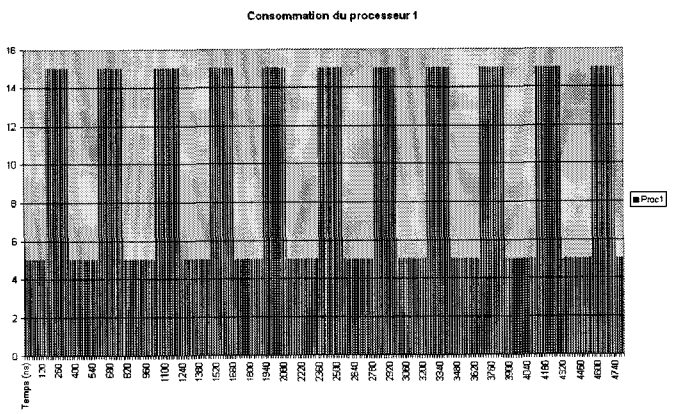


FIG. 6.7 – *Extrait du graphe de consommation pour le processeur contenant la fonction « Scale »*

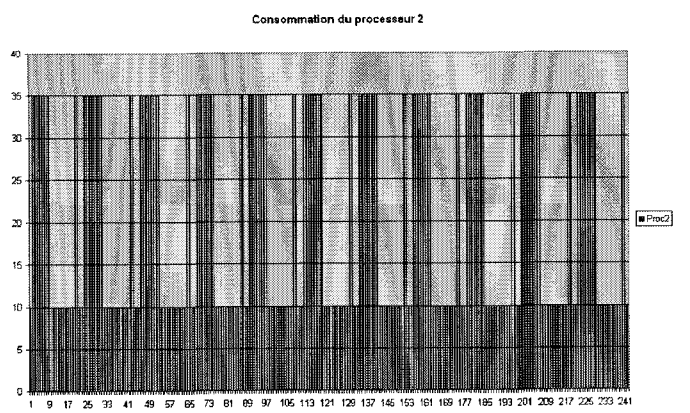


FIG. 6.8 – *Extrait du graphe de consommation pour le processeur contenant les deux tilers*

Chapitre 7

Conclusion

7.1 Bilan

Les systèmes embarqués sont les parties électroniques qui prennent place progressivement dans les objets usuels allant des téléphones mobiles aux voitures.

Récemment la demande pour les systèmes embarqués et le nombre de fonctionnalités souhaitées se sont fortement accrus tandis que les délais de conception requis diminuent. Des architectures multiprocesseurs hétérogénes semblent devenir la clé pour que les systèmes embarqués puissent supporter cette complexité. En paralléle, l'intégration a fait de grands progrés : il est maintenant possible d'intégrer sur une même puce plus de 100 millions de transistors. Il est dès lors possible d'intégrer complétement un système sur une seule puce.

La plupart de ces systèmes monopuce spécifiques à des applications modernes telles que le traitement d'image et les jeux, requièrent une grande capacité de mémoire, des unités de calcul de plus en plus puissantes et rapides, ainsi que des éléments d'interconnexion complexes. Ceci les rend donc sensiblement difficiles Ã concevoir directement à des bas niveaux d'abstraction dans des délais raisonnables. Cette complexité rend également la vérification de la conformité des différentes spécifications du système à tous les niveaux.

Cependant, de nos jours les concepteurs ne disposent d'aucun outil permettant de concevoir de tels systèmes de façon fiables et rapides. En effet, le concepteur d'aujourd'hui compte encore principalement sur sa propre expérience et son effort manuel pour écrire le code décrivant les systèmes, même à des niveaux d'abstraction très bas.

Ce travail se situe donc dans le cadre de la conception de systèmes multiprocesseurs monopuce spécifiques à des applications modernes. Il rentre dans le cadre du projet INRIA DaRT. Ce dernier consiste en le développement d'un environnement de conception de SoC, implémenté dans un outil appelé Gaspard.

Gaspard est un outil de conception basé sur les principes MDA. Il permet une flexibilité et réutilisation considérables moyennant les métamodèles Ã différentes étapes de conception.

Ce travail de thèse s'inscrit dans le cadre global de Gaspard, plus particulièrement aux parties : génération de code, simulation et analyse de performances à un haut niveau d'abstraction. Il propose un métamodèle SystemC au niveau TLM permettant de générer automatiquement du code simulable en utilisant un moteur de transformation. Des éléments permettants l'estimation de performances sont introduits dans les modèles, donnant ainsi aux concepteurs des mesures pertinentes utilisables pour une exploration d'architectures efficace.

Cette méthodologie a été validée avec une application de traitement d'images déployée sur une architecture bi processeurs.

Les principaux avantages de la méthodologie présentée dans ce travail sont :

- Transformation de modèles suivant l'approche MDA : ceci permet une grande réutilisation des modèles, mais également une économie considérable dans le temps de conception, et par conséquent dans le temps de mise sur le marché.
- L'automatisation : le code exécutable est généré automatiquement en utilisant un moteur générique de transformations modèle à modèle. Ceci décharge le concepteur totalement de l'écriture manuelle des modèles de bas niveau.
- Vérification : l'utilisation de l'approche MDA assure la cohérence des modèles au fur et à mesures des transformations et raffinements. Les modèles obtenus par le moteur de transformation sont ainsi valides par construction.
- Estimation de performances et de la consommation : le flot proposé permet non seulement d'intégrer aux modèles des estimation de paramètres divers, mais également d'observer les détails de son exécution et de suivre sa consommation après la simulation. Ceci peut facilement être utilisé de manière très rentable dans la phase pénible de l'exploration d'architectures.

Ses limitations sont principalement :

- SystemC est le seul langage supporté
- Le retour d'informations automatiquement vers le haut du flot de conception n'est pas supporté dans l'état actuel des travaux.

7.2 Perspectives

Une méthodologie de génération de code exécutable au niveau TLM et d'estimation de performances pour la conception de systèmes multiprocesseurs monopuce spécifiques a été proposée au cours de cette thèse. Toutefois, dans le but d'optimiser encore plus cette méthodologie et l'étendre pour être encore plus générale, certains axes de réflexion méritent d'être considérés. Sans prétendre être exhaustif, en voici une liste :

- Développement d'autres métamodèles supportant les principaux langages utilisés dans la conception de SoCs tels que VHDL, Verilog, SpecC, C# et Matlab.

- Application de la même méthodologie sur des niveaux d'abstraction encore plus bas comme CABA et RTL.
- Prise en compte des FPGAs.

Bibliographie

[]

[AAJ+01] Agrawal A, Bakshi A, Davis J, Eames B, Ledeczi A, Mohanty S, Mathur V, Neema S, Nordstrom G, Prasanna V, Raghavendra C, and Singh M. "milan : A model based integrated simulation framework for design of embedded systems". In *"Workshop on Languages, Compilers, and Tools for Embedded Systems"*, Snowbird, Utah, June 2001.

[ABB+03] EL Mustapha Aboulhamid, Mike Baird, Bishnupriya Bhattacharya, David Black, Dundar Dumlogal, Abhijit Ghoshand Andy Goodrich, Robert Graulich, Thorsten Groetker, Martin Jannsenand, Evan Lavelle, Kevin Kranen, Wolfgang Mueller, Kurt Schwartzand Adam Rose, Ray Ryan, Minoru Shoji, and Stuart Swan. *SystemC 2.0.1 Language Reference Manual.* 2003.

[ABD01] Abdelkader Amar, Pierre Boulet, and Jean-Luc Dekeyser. Assembling dynamic components for metacomputing using CORBA. In *Parallel Computing 2001*, Naples, Italy, septembre 2001. Lecture Notes in Computer Science.

[Ada01] Mickael D. Adams. The JPEG-2000 still image compression standard. Technical Report N2412, ISO/IEC JTC 1/SC 29/WG 1, septembre 2001. `http://www.jpeg.org/wg1n2412.pdf`.

[AK01] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures : A Dependence-based Approach.* Morgan Kaufmann Publishers, octobre 2001. `http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-286-0`.

[BDL+] Pierre BOULET, Jean-Luc DEKEYSER, Jean-Luc LEVAIRE, Philippe MARQUET, Julien SOULA, and Alain DEMEURE. Visual data-parallel programming for signal processing applications.

[BDL+01] Pierre Boulet, Jean-Luc Dekeyser, Jean-Luc Levaire, Philippe Marquet, Julien Soula, and Alain Demeure. Visual data-parallel programming for signal processing applications. In *9th Euromicro Workshop on Parallel and Distributed Processing, PDP 2001*, pages 105–112, Mantova, Italy, février 2001.

[BDSV98] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms : From parallelism extraction to code generation. *Parallel Computing*, 24(3-4) :421–444, mai 1998.

[BF98] Pierre Boulet and Paul Feautrier. Scanning polyhedra without DO-loops. In *PACT'98*, pages 4–11. IEEE Computer Society, 1998.

[Boa01] OMG Architecture Board. Model driven architecture (MDA). Technical Report ormsc/2001-07-01, OMG, 2001.

[Dar] http ://www-futurs.inria.fr.

[DDGV01] Alain Darte, Claude Diderich, Marc Gengler, and Frédéric Vivien. Scheduling the computations of a loop nest with respect to a given mapping. *Lecture Notes in Computer Science*, 1900, 2001.

[Dem01] Didier Demigny, editor. *Méthodes et architectures pour le TSI en temps réel.* Hermès Science Publications, 2001. ISBN 2-7462-0327-8.

[DG] Alain DEMEURE and Yannick Del GALLO. An array approach for signal processing design.

[dKES+00] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI : Application modeling for signal processing systems. In *37th Design Automation Conference*, Los Angeles, CA, juin 2000. ACM Press.

[DRV00] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization.* Birkhauser Boston, 2000. `http://www.birkhauser.com/detail.tpl?isbn=0817641491`.

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES99)*, pages 74–78, New York, mai 3–5 1999. ACM Press.

[GMS00] Manish Gupta, Sayak Mukhopadhyay, and Navin Sinha. Automatic parallelization of recursive procedures. *International Journal of Parallel Programming*, 28(6) :537–562, 2000.

[HLPJ99] W.M. Ho, A. Le Guennec, F. Pennaneac'h, and J.-M. Jézéquel. UMLaut : an extendible UML transformation framework. Rapport de recherche 3775, INRIA, 1999.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74 : Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland, août 1974.

[Kea99] Katarzyna Keahey. PARDIS : Programmer-level abstractions for metacomputing. *Future Generation Computer Systems*, 15(5–6) :637–647, octobre 1999.

[KM77] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77 : Proceedings of the IFIP Congress 77*, pages 993–998. North-Holland, 1977.

[LDN+01] Akos Ledeczi, James Davis, Sandeep Neema, Brandon Eames, Greg Nordstrom, Viktor Prasanna, C.S. Raghavendra, Amol Bakshi, Sumit Mohanty, Vaibhav

Mathur, and Mitali Singh. Overview of the model-based integrated simulation (milan) framework. Technical report, ISIS, 2001.

[Lee01] Edward A. Lee. *Overview of the Ptolemy Project.* University of California, Berkeley, mars 2001.

[LF97] Vincent Lefebvre and Paul Feautrier. Optimizing storage size for static control programs in automatic parallelizers. In *European Conference on Parallel Processing*, pages 356–363, 1997.

[Lim01] Amy Wingmui Lim. *Improving Parallelism and Data Locality with Affine Partitioning.* PhD thesis, Stanford University, septembre 2001.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data flow analysis and its use in array privatization. In ACM, editor, *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages : papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993*, pages 2–15, New York, NY, USA, 1993. ACM Press.

[Mar02] Philippe Le Marrec. *Cosimulation multiniveaux dans un flot de conception multilangages.* These, „ Grenoble, France, Jun 2002. in-french.

[Mef02] Samy Meftali. *Exploration d'architectures et allocation/affectation memoire dans les systemes multiprocesseurs monopuce.* PhD thesis, TIMA, UJF Grenoble, septembre 2002.

[ML02] Praveen K. Murthy and Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, juillet 2002.

[Mot00] Markus Mottl. Automating functional program transformation. Master's thesis, University of Edinburgh, septembre 2000. `http://www.ai.univie.ac.at/~markus/msc_thesis/`.

[MP02] Sumit Mohanty and Viktor K. Prasanna. "rapid system-level performance evaluation and optimization for application mapping onto soc architectures". In *"15th IEEE International ASIC/SOC Conference"*, New York, USA, 2002.

[Nic02] Gabriela Nicolescu. *Specification and validation for heterogeneous embedded systems.* PhD thesis, INPG, Nov 2002.

[Obj00] Object Management Group, Inc. MOF meta object facility, specification, version 1.3. `http://www.omg.org/cgi-bin/doc?formal/00-04-03`, janvier 2000.

[Obj01a] Object Management Group, Inc., editor. *Common Object Request Broker Architecture (CORBA), Version 2.6.* `http://www.omg.org/technology/documents/formal/corba_iiop.htm`, décembre 2001.

[Obj01b] Object Management Group, Inc. Data parallel processing final adopted specification. `http://cgi.omg.org/cgi-bin/doc?ptc/01-11-09.pdf`, novembre 2001.

[Obj01c] Object Management Group, Inc., editor. *Unified Modeling Language (UML), Version 1.4.* http://www.omg.org/technology/documents/formal/uml.htm, septembre 2001.

[Obj02a] Object Management Group, Inc., editor. *(UML) Profile for Schedulability, Performance, and Time Specification.* http://www.omg.org/cgi-bin/doc?ptc/2002-03-02/, mai 2002.

[Obj02b] Object Management Group, Inc., editor. *XML Metadata Interchange (XMI), Version 1.2.* http://www.omg.org/technology/documents/formal/xmi.htm, janvier 2002.

[Obj02c] Objecteering Software. Objecteering version 5.2. http://www.objecteering.com/, 2002.

[Ope02] Open SystemC Initiative. SystemC. http://www.systemc.org/, 2002.

[PaRH98] M. Pfaff and M. Schutti ans R. Hagelauer. Shardc : Coupling simulation and physical hardware to improve design validation. In *Design Automation and Test in Eorope Conference*, pages 129–133, Paris, France, Mar 1998.

[PPRS01] Cristóbal Pareja, Ricardo Peña, Fernando Rubio, and Clara Segura. Optimizing Eden by program transformation. In *2nd Scottish Functional Programming Workshop, St. Andrews 2000.* Intellect, 2001.

[PR98] Thierry Priol and Christophe René. Cobra : A CORBA-compliant programming environment for high-performance computing. In *Euro-Par'98*, number 1470 in Lecture Notes in Computer Science, pages 1114–1122, Southampton, UK, novembre 1998.

[Rat01] Rational. Rational Rose v2001 : Visual modeling, UML, object-oriented, component-based development with Rational Rose. http://www.rational.com/products/rose/index.jsp, 2001.

[Ree95] Hideki John Reekie. *Realtime Signal Processing : Dataflow, Visual, and Functional Programming.* PhD Thesis, School of Electrical Engineering, University of Technology, Sydney, Australia, septembre 1995.

[SL97] Yves Sorel and Christophe Lavarenne. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal (numéro spécial Adéquation Algorithme Architecture)*, 14(6) :569–578, 1997.

[SL00] Yves Sorel and Christophe Lavarenne. *SynDEx Documentation Index.* INRIA, 2000. http://www-rocq.inria.fr/syndex/doc/.

[Sor94] Yves Sorel. Massively parallel computing systems with real time constraints - the "Algorithm Architecture Adequation" methodology. In *Proceedings of the 1st International Conference on Massively Parallel Computing Systems*, pages 44–54, Los Alamitos, CA, USA, mai 1994. IEEE Computer Society Press.

[Sou01] Julien Soula. *Principe de Compilation d'un Langage de Traitement de Signal.* Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, décembre 2001.

[SR98] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. White paper, Rational Rose, 1998.

[SVD+02] Stuart Swan, Dirk Vermeersch, Dündar Dumlugöl, Peter Hardee, Takashi Hasegawa, Adam Rose, Marcello Coppola, Martin Janssen, Thorsten Grötker, Abhijit Ghosh, and Kevin Kranen. *Functional specification for systemc 2.0.* 2002.

[SW95] Manuel Serrano and Pierre Weis. Bigloo : A portable and optimizing compiler for strict functional languages. In *Static Analysis Symposium*, pages 366–381, 1995.

[Tel99] Telecommunication Standardization Sector of Itu. Specification and description language (SDL). `http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf`, 1999.

[Tel02] Telelogic. Tau generation 2. `http://www.taug2.com/`, 2002.

[TP01] Peng Tu and David Padua. Chapter 8. automatic array privatization. *Lecture Notes in Computer Science*, 1808, 2001.

[TVSA01] William Thies, Frederic Vivien, Jeffrey Sheldon, and Saman P. Amarasinghe. A unified framework for schedule and storage optimization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 232–242, 2001.

[VK01] Mathur V. and Prasanna V. K. "a hierarchical simulation framework for application development on system-on-chip architectures". In *"14th IEEE Int'l ASIC-SOC Conference"*, Washington DC, USA, September 2001.

[VSI01] VSI Alliance, editor. *Virtual Component Interface Standard Version 2.* `http://www.vsi.org/`, avril 2001.

[Wil92] D. S. Wile. Integrating syntaxes and their associated semantics. Technical Report RR-92-297, Univ. Southern California, California, USA, 1992.

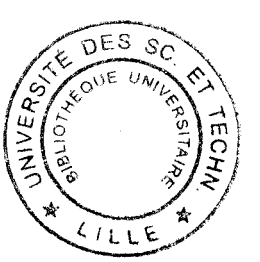
[ZJ93] P. Zave and M. Jackson. Conjunction as composition. In *On Software engineering and methodology*, volume 8, pages 379–411. ACM trans., Oct 1993.

[ZMS] Qiang Zhu, Akio Matsuda, and Minoru Shoji. An object-oriented design process for system-on-chip using UML.

Table des figures

Liste des tableaux

Liste des Algorithmes



Modélisation Unifiée des Aspects Répétitifs dans la Conception Conjointe Logicielle/Matérielle des Systèmes sur Puce à Hautes Performances

Thèse

présentée à

L'Université des Sciences et Technologies de Lille

Pour obtenir le titre de

Docteur en Informatique

par

Arnaud Cuccuru

Soutenue le 29 novembre 2005
devant la commission d'examen composée de :

Jean-Marc Geib,	Président,	LIFL
Jean-Luc Dekeyser,	Directeur,	LIFL
Jean-Marc Jezequel,	Rapporteur,	IRISA
François Terrier,	Rapporteur,	CEA
Pierre Boulet,	Examinateur,	LIFL
Robert de Simone,	Examinateur,	INRIA

Remerciements [1]

Au professeur Jean-Luc Dekeyser, pour m'avoir fait confiance alors que j'étais prédestiné à une carrière de comptable.

Au professeur Jean-Marc Geib, pour m'avoir fait l'honneur d'accepter de présider mon jury de thèse.

Aux professeurs Jean-Marc Jezequel et François Terrier, pour avoir accepter d'évaluer mes travaux, avec l'investissement en temps et en énergie que cela implique, et pour leurs précieux conseils et leurs remarques avisées.

A Robert de Simone, directeur de recherche INRIA, et au professeur Pierre Boulet, pour leur soutien, leurs conseils, et leur présence en tant qu'examinateurs.

A tous les membres de l'équipe WEST, passés (au moins à ceux que j'ai connu), présents (pour leur soutien, leurs idées, ou à défaut leur bonne humeur), et futurs (au moins à ceux qui liront cette thèse).

A tous mes proches, qui m'ont soutenu aussi bien matériellement que moralement, et m'ont permis de me remettre régulièrement en cause, en me posant des questions fréquentes et pertinentes sur mes travaux : "En quoi ça consiste une thèse ?", "Bon, je sais que je te l'ai déjà demandé, mais tu travailles sur quoi exactement ?", ou encore "Et où est-ce que ça va t'ammener ?". Pour répondre à cette dernière question, il est difficile d'éviter les lieux communs. Je serai tenté de répondre : "L'avenir nous le dira". Plus prosaiquement, je répondrais que dans le pire des cas, cette thèse m'ammènera au bureau de l'ANPE le plus proche de chez moi. Avec un peu de chance, on m'y proposera une formation de boucher ou d'entretien des espaces verts [2]...

A Thomson, pour avoir inventé le MO5.

Pour conclure la section des remerciements, il est d'usage de faire une dédicace. Parfois, le travail est dédié *"à celui ou celle qui n'a pas encore vu le jour au moment où ces lignes sont écrites"*, ou encore *"à toutes les filles que j'ai aimé avant"*. Mais après tout, ça n'est qu'une modeste thèse de doctorat en informatique, parmi les centaines qui doivent être soutenues chaque année dans le monde, une goutte de contribution dans l'océan de la science [3]. Je vais donc dédier cette thèse aux arbres qui vont se sacrifier pour permettre son impression [4], et conclure cette section par une citation écologique, à méditer...

Si vous êtes mordus par une vipère, sucez-vous le genou, ça fera marrer les écureuils.
Pierre Desproges

---

[1] **remerciements** n. m. pl. : Note de reconnaissance d'un auteur envers ceux qui l'ont aidé dans la rédaction de son ouvrage. [Office de la langue française, 1982]

[2] Il n'y a pas de sous métiers, l'essentiel étant de gagner sa vie honnêtement.

[3] Après la comptabilité, la poésie était ma deuxième passion...

[4] ... l'écologie la troisième.

Table des matières