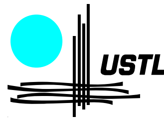


Numéro d'ordre : 4108

Université des Sciences et Technologies de Lille

Thèse
présentée pour obtenir le titre de
docteur spécialité Informatique
par
Éric Piel

Ordonnancement de systèmes parallèles temps-réel
De la modélisation à la mise en œuvre par l'ingénierie dirigée par les modèles

Thèse soutenue le 14 décembre 2007, devant la commission d'examen formée de :

Laurence Duchien	Professeur LIFL	**Présidente**
Frédéric Pétrot	Professeur INPG-TIMA	**Rapporteur**
François Verdier	Maître de conférence ENSEA	**Rapporteur**
Michel Riveill	Professeur UNSA	**Examinateur**
Sébastien Gérard	Chef de projet CEA	**Examinateur**
Jean-Luc Dekeyser	Professeur LIFL	**Directeur**
Philippe Marquet	Maître de conférence LIFL	**Co-Directeur**

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
LIFL - UMR 8022 - Cité Scientifique, Bât. M3 - 59655 Villeneuve d'Ascq Cedex

Remerciements

Ce manuscrit est l'achèvement de trois années de travaux de thèse, cette étape qui correspond à la fois à la fin de la période d'apprentissage et au début du travail de chercheur. Profitons en pour rappeler que malgré l'aspect relativement direct et lié des solutions présentées dans ce manuscrit, le travail de chercheur passe nécessairement par des recherches qui s'avèrent a posteriori inutiles ; ce manuscrit ne mentionne pas un certain nombre de travaux qui n'ont pas abouti ou qui sont devenus obsolètes suite à d'autres travaux. Mon plus grand espoir est que les propositions présentées ici pourront être utiles à d'autres, en servant de base à des produits industriels ou à d'autres propositions scientifiques plus avancées.

Je tiens à exprimer ma profonde gratitude envers Jean-Luc Dekeyser et Philippe Marquet pour m'avoir encadré, conseillé et soutenu tout au long de ma thèse et pour m'avoir guidé et aidé lors de l'écriture de ce manuscrit. Je remercie également les rapporteurs de cette thèse, Frédéric Pétrot et François Verdier ainsi que les examinateurs, Laurence Duchien, Michel Riveill, et Sébastien Gérard, pour avoir lu et commenté ce manuscrit et aussi pour leur remarques, questions et recommandations lors de ma présentation. Je tiens en outre à saluer l'INRIA et la région Nord-Pas de Calais pour avoir financé mes recherches au cours de ces trois années.

Les travaux présentés font tous partie des projets Gaspard ou ARTiS, et ils n'auraient pas pu avoir lieu sans le travail fournit par tous les autres contributeurs de ces deux projets. J'en profite pour exprimer toute ma sympathie et gratitude à l'ensemble des membres de l'équipe WEST/DaRT pour leur aide, leur collaboration, leur solidarité et leur amitié, dont en particulier et dans le désordre : Julien S., Christophe, Julien T., Sébastien, Rabie, Huafeng, Imran, Anne, Myriam, Pierre, Abdoulaye, Calin, Antoine, Asma, Adolf, César, Lossan, Ouassila, Ashish, Arnaud, et Karine. Je tiens d'autant plus à exprimer ma reconnaissance envers Anne, Myriam, Asma, et Julien T. pour avoir participé à la relecture de ce manuscrit.

Enfin, je souhaiterais remercier Sachiyo et l'ensemble de ma famille pour m'avoir soutenu si souvent au cours de ma thèse.

Table des matières

Table des matières 3

1 Introduction 7
- 1.1 Contexte et problématique . . . 7
- 1.2 Contributions . . . 8
- 1.3 Plan . . . 9

I Modélisation et simulation de MPSoC 11

2 Contexte et problématique 15
- 2.1 Ingénierie dirigée par les modèles . . . 15
  - 2.1.1 Principes et concepts fondamentaux de l'IDM . . . 17
  - 2.1.2 Modélisation et méta-modélisation . . . 20
  - 2.1.3 Transformation de modèles . . . 24
  - 2.1.4 Conclusion . . . 26
- 2.2 Systèmes sur puces . . . 26
  - 2.2.1 Organisation et usage des SoC . . . 27
  - 2.2.2 Conception des SoC . . . 28
  - 2.2.3 Co-simulation logiciel-matériel . . . 31
  - 2.2.4 Défis actuels . . . 34
  - 2.2.5 Conclusion . . . 35
- 2.3 Gaspard, un outil pour la co-modélisation de SoC . . . 35
  - 2.3.1 Le paquetage *Component* . . . 37
  - 2.3.2 Le paquetage *Factorization* . . . 38
  - 2.3.3 Le paquetage *Application* . . . 42
  - 2.3.4 Le paquetage *HardwareArchitecture* . . . 45
  - 2.3.5 Le paquetage *Association* . . . 45
  - 2.3.6 Synthèse . . . 47

3 Un méta-modèle pour l'expression exécutable de MPSoC 49
- 3.1 Allocation répétitive et hiérarchique . . . 49
  - 3.1.1 Allocation répétitive . . . 50
  - 3.1.2 Association hiérarchique . . . 56
- 3.2 Le méta-modèle de déploiement . . . 64
  - 3.2.1 Aperçu du paquetage *Deployment* . . . 65
  - 3.2.2 Résumé des concepts . . . 67

3.2.3 Notion d'*Implementor* . . . 68
3.2.4 Distinction *Software*/*Hardware* . . . 73
3.2.5 Notion de *PortImplementation* . . . 76
3.2.6 Notion d'*ImplementedByConnector* . . . 78
3.2.7 Notions de *Characterizable* et *Specializable* . . . 79
3.2.8 GaspardLib, une bibliothèque de composants pour la modélisation de SoC . . . 83
3.2.9 Synthèse et conclusion : le déploiement d'IP . . . 86
3.3 Conclusion . . . 88

4 Simulation à haut niveau d'abstraction 89
4.1 Simulation au motif près . . . 90
4.1.1 Spécificités des systèmes simulés . . . 90
4.1.2 Bases de la simulation au motif près . . . 93
4.1.3 Avantages et inconvénients de l'approche . . . 95
4.2 Un modèle d'exécution du modèle de calcul Gaspard . . . 96
4.2.1 Projections sur différents modèles d'exécution . . . 96
4.2.2 Flux de tableaux . . . 99
4.2.3 Implémentation des *tilers* . . . 101
4.2.4 Synchronisation des tâches . . . 102
4.2.5 Ordonnancement sur un seul processeur . . . 103
4.2.6 Ordonnancement dynamique . . . 104
4.2.7 Synthèse et conclusion . . . 106
4.3 Implémentation de la simulation en SystemC . . . 107
4.3.1 Simulation en SystemC . . . 107
4.3.2 Un ordonnanceur intégré au simulateur . . . 108
4.3.3 Le mécanisme de synchronisation . . . 113
4.3.4 Lecture et écriture des données . . . 116
4.3.5 Estimation du temps d'exécution . . . 117
4.3.6 Synthèse . . . 119
4.4 Conclusion . . . 120

5 Usage de l'IDM pour la compilation de SoC 121
5.1 Méta-modèles intermédiaires . . . 122
5.1.1 Aperçu global du méta-modèle *Polyhedron* . . . 123
5.1.2 Les composants applicatifs . . . 123
5.1.3 Expression de la répétition des tâches . . . 125
5.1.4 Les tableaux de données . . . 127
5.1.5 Le déploiement simplifié . . . 129
5.1.6 Le méta-modèle *Loop* . . . 130
5.1.7 Synthèse . . . 131
5.2 Du méta-modèle *Gaspard* vers le méta-modèle *Polyhedron* . . . 131
5.2.1 Moteur de transformations modèle à modèle . . . 132
5.2.2 Génération de polyèdres pour exprimer la répétition . . . 134
5.2.3 Découpage de l'arborescence des tâches . . . 136
5.2.4 Placement des tableaux de données . . . 137
5.2.5 Simplification du déploiement . . . 138

5.2.6 Synthèse 139
5.3 Génération de code SystemC 139
5.3.1 Moteur de transformation modèle-vers-texte 140
5.3.2 Génération du cœur du processeur 141
5.3.3 Génération des activités 142
5.3.4 Création d'un Makefile 144
5.3.5 Synthèse 146
5.4 Synthèse et conclusion 146

6 Étude de cas et validation expérimentale 149
6.1 Encodeur H.263 sur MPSoC 150
6.1.1 Application 150
6.1.2 Architecture matérielle 154
6.1.3 Association 155
6.2 Génération de code à l'aide de l'environnement Gaspard 159
6.3 Exploration du domaine de conception 160
6.3.1 Variation du nombre de processeurs 161
6.3.2 Variation du nombre de bancs mémoire 164
6.4 Synthèse 165

Conclusion de la première partie 167

II Ordonnancement dynamique de systèmes temps-réel parallèles 169

7 Calcul haute-performance et temps-réel 171
7.1 Multiprocesseur et approches temps-réel 172
7.2 Temps-réel et Linux 173
7.2.1 Le système standard GNU/Linux 173
7.2.2 Approche co-noyau 174
7.2.3 Approche multiprocesseur asymétrique 175
7.3 Ordonnancement multiprocesseur dans le noyau Linux 176
7.3.1 Ordonnancement des tâches 176
7.3.2 Équilibrage de charge 177

8 ARTiS : Un ordonnanceur temps-réel asymétrique 179
8.1 Partitionnement des processeurs et des processus 180
8.2 Mécanisme de migration 181
8.3 Politique d'équilibrage de charge 182
8.4 Mécanismes de communications asymétriques 182
8.5 Synthèse 182

9 Mise en œuvre 183
9.1 Migration dans ARTiS 183
9.1.1 Déclenchement de la migration 184
9.1.2 Déroulement de la migration d'une tâche 185
9.1.3 RT-FIFO : des FIFO sans verrou 186
9.2 Équilibrage de charge 186

9.2.1 Contraintes spécifiques à ARTiS . . . 187
9.2.2 Pondération de la longueur de la file d'exécution . . . 188
9.2.3 Suppression des verrous inter-processeurs . . . 190
9.2.4 Estimation de la prochaine tentative de migration . . . 190
9.2.5 Association du type de tâche au type de processeur . . . 192
9.3 Déploiement d'applications temps-réel . . . 192
9.3.1 Exemple de déploiement d'application . . . 193
9.3.2 Configuration du système . . . 194
9.3.3 Identification des tâches temps-réel . . . 195
9.4 Synthèse de l'implémentation . . . 196

10 Validation Expérimentale 197
10.1 Mesure de latence d'interruption . . . 198
10.1.1 Méthode de mesure . . . 198
10.1.2 Types de latence d'interruption . . . 199
10.1.3 Condition des mesures . . . 199
10.1.4 Latences observées . . . 200
10.2 Variation du temps d'exécution . . . 201
10.2.1 Méthode de mesure . . . 201
10.2.2 Condition des mesures . . . 201
10.2.3 Temps d'exécution observés . . . 202
10.3 Observation de l'équilibrage de charge . . . 202
10.3.1 lbμ, un outil de validation d'équilibrage de charge . . . 203
10.3.2 Conditions des mesures . . . 204
10.3.3 Observation des scénarios de test . . . 204
10.4 Synthèse de la validation expérimentale . . . 207

Conclusion de la deuxième partie 209

Conclusion générale 213

11 Conclusion 213
11.1 Résumé . . . 213
11.2 Perspectives . . . 215

Bibliographie personnelle 219

Bibliographie 221

III Annexes 229

A Déploiement 231

Résumé/Abstract 234

Chapitre 1

Introduction

1.1 Contexte et problématique 7
1.2 Contributions 8
1.3 Plan 9

1.1 Contexte et problématique

Les systèmes électroniques poursuivent une intense évolution. À la base de cette évolution correspond l'augmentation cadencée de la finesse de gravure. Actuellement l'industrie peut travailler avec une finesse de 65nm, mais déjà la technologie à 45nm apparaît. La loi de Moore s'applique toujours, tous les deux ans il est possible de placer deux fois plus de transistors sur une puce électronique. Cette capacité à placer de plus en plus de composants sur une puce est à l'origine des systèmes sur puce (SoC, pour *System-on-Chip*) : la puce n'est pas dédiée à un type de composant particulier mais au contraire contient tous les composants qui forment un système informatique. Processeur, mémoire, réseau d'interconnexion, circuits analogiques, circuits pour les ondes radio... l'éventail de type de composants qui peuvent former le système est énorme. Les SoC sont particulièrement exploités dans l'informatique embarquée en raison des avantages qu'ils confèrent en terme d'espace physique, de consommation d'énergie et aussi, lors de la production à grande échelle, en terme de coûts. Actuellement, pour utiliser le grand nombre de transistors disponible pour le calcul, il n'est plus possible d'avoir un seul processeur. En effet, la séquentialité imposée des instructions force à élever la fréquence et indirectement le voltage à des niveaux incompatibles avec les contraintes physiques du système. Il est nécessaire d'introduire l'usage des multiprocesseurs et d'employer le parallélisme de plus en plus massif [19].

Parallèlement à cette évolution, les applications de l'informatique embarquée se sont étendues et surtout suivent une tendance exigeant de plus en plus de puissance de calcul. Les principaux usages des systèmes embarqués se situent dans les transports (contrôleurs pour les avions, les voitures...), dans les systèmes de détection (radars, sonars...), et dans le multimédia (lecteurs vidéo, téléphones...). La plupart de ces applications correspondent à du traitement de signal : une suite de calculs appliqués sur un flux de données généralement multidimensionnel. Ces traitements sont en essence parallèles et répétitifs. Au fur et à mesure, les nouvelles applications doivent traiter plus de signaux, sur lesquels plus de calculs doivent être appliqués, plus rapidement. Cette demande croissante en puissance de calcul se marie

parfaitement à la mise à disposition de puces électroniques ayant des capacités de plus en plus grandes.

Parfaitement ? Oui, mais dans le sillage de ces augmentations, la complexité de développement, elle aussi, augmente [57]. S'il faut concevoir plus de composants sur une puce ou si une application doit fournir plus de fonctionnalités, alors le développement est plus long ou plus coûteux. Ainsi sans amélioration des méthodes ou des outils de conception, les systèmes de taille toujours croissante sont de plus en plus difficiles à gérer. Cette complexité est exacerbée par l'association de l'application sur le matériel, la quantité d'associations potentielles est une combinaison des tailles de l'application et du matériel. L'espace de conception, qui représente l'ensemble des décisions techniques que peut prendre l'équipe élaborant le SoC, est donc de plus en plus difficile à explorer.

Dans l'organisation actuelle de l'industrie, la hausse de la complexité doit être contre-balancée par une hausse de la productivité des concepteurs. En effet, c'est tout d'abord un moyen d'éviter l'augmentation du coût de développement. Par ailleurs, même si le coût n'était pas une contrainte, la taille d'une équipe de conception ne peut pas être augmentée sans fin. Il arrive un moment où le partage du travail ne fait plus gagner de temps. Hors, ce temps de conception est un enjeu vis-à-vis du temps de mise sur le marché : il faut être capable de réaliser un type de produit *le premier* car, sur le marché de l'électronique, c'est la seule manière de tirer un profit confortable grâce au prix de vente plus élevé tant qu'il n'y pas de concurrence et à la position dominante du marché acquise [55].

Actuellement, nous sommes donc face à un besoin de concevoir plus efficacement les SoC. Pour maîtriser la complexité, il faut pouvoir réutiliser au mieux les composants ou les blocs de composants déjà existants, qu'ils aient été produits précédemment en interne, ou qu'ils soient mis à disposition par un vendeur extérieur. La gestion du parallélisme et de la répétitivité dans le système est un point clef qu'il faut pouvoir traiter spécifiquement. Outre l'étape de spécification, l'étape de vérification doit tout autant être plus productive. Il est donc également important de faciliter au maximum le débogage et le test du système.

1.2 Contributions

C'est dans ce contexte d'amélioration primordiale de la productivité pour les systèmes embarqués parallèles, que se situent les travaux de cette thèse. Une des lignes directrices suivies par ces travaux est le recours à l'Ingénierie Dirigée par les Modèles (IDM) pour organiser la conception et la génération de SoC. L'IDM doit permettre de bénéficier à la fois d'une approche orientée modèle, permettant d'abstraire et de simplifier la représentation des systèmes, et d'une chaîne complète de compilation capable d'exploiter directement les modèles. Un des objectifs de ces travaux était de faire évoluer l'environnement de développement de SoC Gaspard développé par l'équipe. Ainsi, les différentes contributions ont été mises en œuvre dans cet environnement préexistant, permettant en particulier de compléter son flot de conception.

Au niveau de la modélisation, nous introduirons une **sémantique complète capable de représenter de manière compacte le placement** d'une application sur une architecture en prenant en compte à la fois le parallélisme et la hiérarchie de l'application et le parallélisme et la hiérarchie de l'architecture matérielle. Comme nous le verrons, c'est une pièce essentielle pour faciliter la conception de systèmes massivement parallèles. Nous introduirons également un nouvel **ensemble de notions destinées au déploiement d'IP**. Ces notions autorisent la

génération complète du code à partir des modèles. Elles ont également pour but d'améliorer la productivité en simplifiant la réutilisation de composants aussi bien matériels qu'applicatifs, et en permettant d'associer une fonctionnalité à un composant indépendamment de la cible de compilation (la fonctionnalité reste valable quelque soit le langage de programmation ou le niveau d'abstraction ciblé).

Afin de pouvoir parcourir l'espace de conception le plus tôt possible au cours du développement et afin d'accélérer ce parcours, nous proposerons un **haut niveau d'abstraction pour la co-simulation logiciel-matériel**. Il repose en partie sur le fait que l'application est exprimée dans les modèles à un haut niveau d'abstraction. Une version spéciale de l'application est compilée directement pour le processeur hôte de la simulation, ce qui permet de réduire le temps de simulation tout en permettant au développeur de lier directement les résultats de la simulation aux concepts manipulés dans les modèles.

Nous présenterons également la **mise en œuvre de transformations de modèle destinées à la compilation de SoC multiprocesseurs** (MPSoC) vers une simulation SystemC au niveau d'abstraction que nous venons tout juste de mentionner. Cela sera l'occasion de valider l'idée d'un flot de conception basé sur l'IDM. Ce sera également l'opportunité de mettre en avant les améliorations sur la productivité que peuvent avoir les transformations de modèles lorsqu'elles sont utilisées pour la compilation.

Afin de prendre en compte les contraintes temps-réel des systèmes embarqués tout en bénéficiant de la puissance de calcul offerte par les systèmes multiprocesseurs tels que les MPSoC, nous proposerons une approche d'**ordonnancement de tâches temps-réel basé sur une asymétrie imposée entre les processeurs**. Cet ordonnancement dynamique garanti sur certains processeurs les propriétés temps-réel en déplaçant automatiquement toute tâche qui pourrait mettre en danger ces propriétés vers un autre processeur. En outre, nous introduirons une **modification du noyau Linux utilisant cette approche**. Nous montrerons l'efficacité de cette proposition qui permet de développer des applications temps-réel aussi simplement que des applications Linux usuelles aussi bien du point de vue de la programmation qu'au niveau du débogage.

1.3 Plan

Ce mémoire est composé de deux grandes parties. La première partie traitera de l'usage de l'IDM dans le cadre de la conception de SoC en prenant appuis sur les bases mises à disposition par l'environnement Gaspard.

Dans le chapitre 2, nous présenterons le contexte de cette partie en mettant l'accent particulièrement sur l'IDM, la co-conception et la co-simulation de SoC, et l'environnement Gaspard.

Nos propositions concernant les méta-modèles seront présentées dans le chapitre 3. Elles ont à la fois pour but de permettre la génération de code complète depuis un modèle de SoC et de faciliter la modélisation de systèmes complexes.

Dans le chapitre 4, nous commencerons par exposer les bases d'un niveau de simulation à haut niveau d'abstraction. Par la suite une projection du modèle de calcul régissant l'application dans Gaspard vers un modèle d'exécution adapté aux SoC multiprocesseurs sera proposée. Nous détaillerons alors comment un modèle applicatif de Gaspard peut être simulé au niveau d'abstraction précédemment proposé.

Le chapitre 5 présentera la chaîne de compilation à base de transformations de modèles

que nous avons mise en place. Après avoir défini les méta-modèles intermédiaires de la chaîne, nous détaillerons les transformations permettant le passage du méta-modèle utilisé pour la conception des MPSoC vers un méta-modèle plus proche du code final. Puis une seconde transformation ayant pour but de produire du code SystemC pour la co-simulation à haut niveau sera présentée.

Une étude de cas d'une application d'encodage vidéo placée sur un SoC à base de plusieurs processeurs MIPS sera faite dans le chapitre 6. Non seulement cela permettra de donner un aperçu du flot de conception de SoC dans l'environnement Gaspard mais aussi validera les différentes propositions faites précédemment.

La seconde partie exposera les travaux permettant de faire cohabiter sur un même système les garanties temps-réel et la puissance de calcul procurée par une architecture multiprocesseur. Elle débutera par le chapitre 7 où seront présentées les approches existantes mêlant temps-réel et multiprocesseur. Ce sera aussi l'occasion de détailler le fonctionnement de l'ordonnancement et l'équilibrage de charge dans le noyau Linux.

Dans le chapitre 8, nous proposerons ARTiS, une technique d'ordonnancement asymétrique capable de mixer garanties temps-réel et multiprocesseurs.

Le chapitre 9 présentera l'implémentation de cette proposition dans le noyau Linux. Nous y détaillerons les modifications sur l'ordonnanceur et sur l'équilibrage de charge. Nous verrons également l'interface proposée permettant la mise en œuvre d'applications temps-réel et gourmandes en ressources de calcul.

Une validation expérimentale de cette implémentation sera exposée dans le chapitre 10. En particulier nous verrons des tests sur la latence d'interruption, la variation du temps d'exécution, et sur les modifications des politiques d'équilibrage de charge.

Enfin, dans le chapitre 11, nous conclurons ce mémoire et présenterons quelques perspectives.

Première partie

Modélisation et simulation de MPSoC

Dans cette partie nous nous intéresserons à l'usage de la modélisation et des outils de transformation de modèles comme aide à la conception de SoC, plus précisément les SoC parallèles destinés au traitement de signal systématique. Le but est de faciliter les différentes étapes de développement du système, depuis la description initiale à l'interprétation des résultats de simulation. Après avoir détaillé le contexte dans lequel ces travaux se situent, nous exposerons les contributions permettant de modéliser un MPSoC avec suffisamment d'informations pour être capable de produire une version exécutable, puis nous proposerons une simulation à haut niveau d'abstraction adapté aux concepts modélisés et favorisant la rapidité de simulation. Nous exposerons par la suite l'usage de transformations de modèles dans le but de générer une simulation du MPSoC et nous présenterons un exemple mettant en valeur l'usage et l'intérêt des différentes contributions.

Chapitre 2

Contexte et problématique

2.1 Ingénierie dirigée par les modèles **15**
 2.1.1 Principes et concepts fondamentaux de l'IDM 17
 2.1.2 Modélisation et méta-modélisation 20
 2.1.3 Transformation de modèles 24
 2.1.4 Conclusion 26
2.2 Systèmes sur puces **26**
 2.2.1 Organisation et usage des SoC 27
 2.2.2 Conception des SoC 28
 2.2.3 Co-simulation logiciel-matériel 31
 2.2.4 Défis actuels 34
 2.2.5 Conclusion 35
2.3 Gaspard, un outil pour la co-modélisation de SoC **35**
 2.3.1 Le paquetage *Component* 37
 2.3.2 Le paquetage *Factorization* 38
 2.3.3 Le paquetage *Application* 42
 2.3.4 Le paquetage *HardwareArchitecture* 45
 2.3.5 Le paquetage *Association* 45
 2.3.6 Synthèse 47

Avant d'aborder les contributions de cette thèse, nous allons détailler le contexte de ces travaux. En premier lieu, nous présenterons l'ingénierie dirigée par les modèles, les avantages qu'elle doit apporter et les principes sur lesquels elle repose. En second lieu, nous brosserons un aperçu des systèmes sur puce en mettant l'accent sur la conception et la simulation de ces systèmes. L'un des buts de cette thèse était la mise en œuvre du flot de conception de l'environnement Gaspard. Dans la dernière partie de ce chapitre, nous présenterons cet environnement de co-conception de SoC, tel qu'il était au commencement de cette thèse.

2.1 Ingénierie dirigée par les modèles

Tandis que la puissance matérielle s'accroît, les utilisateurs attendent des systèmes informatiques qu'ils puissent traiter des problèmes d'autant plus complexes. Pour répondre à cette attente, les logiciels deviennent eux aussi de plus en plus complexes. Afin de pouvoir suivre

l'augmentation constante en complexité des systèmes logiciels depuis leur introduction dans les années 50, la conception logicielle a grandement évolué. Elle a évolué selon plusieurs axes :

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

Au nombre des évolutions dans le développement des logiciels, on peut citer l'apparition de plusieurs méthodes de modélisation : Merise [95] (1970), SSADM (Structured Systems Analysis and Design Methodology) [41] (1980), UML (Unified Modeling Language) [78] (1995), etc. Ces méthodes permettent d'appréhender le concept de *modèle* en informatique. Elles proposent des concepts et une notation permettant de décrire le système à concevoir. En général, à chaque étape du cycle de vie du système, un ensemble de documents constitués de diagrammes permettent aux concepteurs, décisionnaires, développeurs, utilisateurs, etc. de partager leur perception du système. Ces méthodes de modélisation ont été parfois critiquées [43] pour leur lourdeur et leur manque de souplesse face à l'évolution rapide du logiciel. Elles ont conduit à la notion que Favre et al. nomment *modèle contemplatif* – un modèle qui sert essentiellement à communiquer et comprendre, mais reste passif par rapport à la production. Ainsi, après un demi-siècle de pratique et d'évolution, constate-t-on aujourd'hui que le processus de production de logiciels est toujours centré sur le code.

Ces méthodes ont néanmoins des atouts importants. L'abstraction apportée par la modélisation permet à l'utilisateur d'ignorer les détails d'implémentation et les techniques employées, mettant en avant la structure globale du système. Ceci permet de traiter les systèmes grands et complexes beaucoup plus efficacement. Un autre avantage de la modélisation est la possibilité de représenter une même partie par différentes vues, permettant de séparer le système par aspects et de l'appréhender par vues métiers. Ce sont des approches que l'on retrouve également dans la *programmation par aspects* [61] et les *langages dédiés* [70] (DSL).

L'Ingénierie Dirigée par les Modèles [83] (IDM) vient pallier la déficience des méthodes traditionnelles de modélisation en partant du constat suivant : « *Les modèles contemplatifs sont interprétés par l'homme alors que la préoccupation première de l'informaticien est de produire des artefacts interprétables par la machine* » [43, p. 22]. Elle œuvre à fournir un cadre de développement logiciel dans lequel les modèles passent de l'état contemplatif à l'état productif et deviennent les éléments de première classe dans le processus de développement des logiciels. Elle vise ainsi à améliorer la portabilité et la maintenabilité en autorisant la séparation des concepts, en particulier la séparation de la technologie des concepts métiers, à augmenter la productivité et la qualité en favorisant la réutilisation à la fois des outils et de patrons de conception déjà éprouvés. En pratique, pour un projet cela permet d'accélérer le développement et de réduire les coûts de conception.

Dans cette section nous reviendrons tout d'abord sur les principes et concepts fondamentaux sur lesquels se base l'IDM. Par la suite nous détaillerons la notion qui confère à cette méthode la propriété d'être productive : les transformations de modèles.

2.1.1 Principes et concepts fondamentaux de l'IDM

Avant tout, l'IDM est une méthodologie de conception de système, c'est-à-dire un ensemble de règles (de méthodes) à suivre dans la manière de concevoir un système informatique. Notons que cette méthodologie est encore récente et est encore l'objet de propositions. En particulier, on peut la retrouver, avec de légères variations, sous de nombreux noms (anglais) tels que *Model-Driven Engineering* (MDE), *Model-Driven Architecture* (MDA), *Model-Driven Development* (MDD), *Model Integrated Computing* (MIC), *Model-Driven Software Development* (MDSD), etc. Nous ne présenterons pas ici les nuances subtiles que peuvent impliquer les différentes dénominations mais au contraire nous insisterons sur l'essence commune de toutes les propositions qui forme un socle de principes et de concepts.

2.1.1.1 Les modèles, citoyens première classe

Le principe de base de l'approche est que l'ensemble des informations est représenté sous forme de *modèle*. En soi, le modèle est une abstraction de la réalité. En associant des symboles à des concepts réels et en indiquant les relations entre les différents concepts, on représente la réalité d'un point de vue donné. Ce principe est le même que les autres méthodes de modélisation précédemment citées. L'IDM se démarque par l'usage *systématique* des modèles tout au long du développement logiciel. En particulier, le code source n'est qu'un dérivé équivalent d'un modèle, obtenu en phase finale du développement.

Il est important de noter qu'en soi la notion de modèle apportée par l'IDM n'a absolument rien de novateur. Comme l'a fait remarquer Favre [42], on peut considérer que la notion remonte à plusieurs millénaires. L'alphabet cunéiforme ougaritique (3400 av. J.-C.) introduisait déjà la même notion, définissant un ensemble de représentations abstraites (les caractères) et de règles (la prononciation) pour permettre l'expression de la réalité (la parole). Plus récemment, et plus proche de l'informatique, les langages de programmation, les bases de données relationnelles, le web sémantique, reposent tous sur cette même idée d'ensembles de concepts prédéfinis et reliés qui, une fois interprétés, représentent une certaine réalité. Les modèles ne sont pas nouveaux, ils sont au contraire une manière de penser que l'homme utilise depuis la nuit des temps. L'IDM propose d'utiliser cette manière de penser pour la conception de systèmes en la formalisant et en l'adaptant au traitement informatique.

2.1.1.2 Interprétation des modèles par les machines

Pour qu'un modèle soit interprétable par une machine, il faut que l'expression dans laquelle il est représenté ait été préalablement formellement définie. C'est ce que la notion de *méta-modèle* apporte. Dans l'IDM, un méta-modèle permet de spécifier un langage d'expression de modèles. Il définit les concepts et les relations entre ces concepts qui sont disponibles lors de l'écriture de modèles. Il définit la syntaxe des modèles.

Un modèle qui est écrit selon un méta-modèle donné est dit *conforme à* ce méta-modèle. Cette relation est analogue à un texte et la grammaire de la langue. Comme son nom l'indique, un méta-modèle est lui-même un modèle. C'est-à-dire qu'il est lui aussi conforme à un méta-modèle. Bien sûr, pour comprendre un modèle il serait particulièrement peu pratique d'avoir à connaître une succession infinie de méta-modèles, chacun conforme à un autre de niveau d'abstraction plus élevé ! La solution formelle est d'avoir un méta-modèle qui soit conforme à lui-même, c'est-à-dire qui puisse être exprimé uniquement à l'aide des concepts qu'il définit. Ce méta-modèle récursif est le plus haut qu'il soit nécessaire de connaître. Actuellement, les

méta-modèles couramment utilisés sont Ecore [37] et MOF [77] (Meta-Object Facility). Très semblables, ils définissent par exemple les concepts d'objet, de classe, d'héritage, d'association, etc.

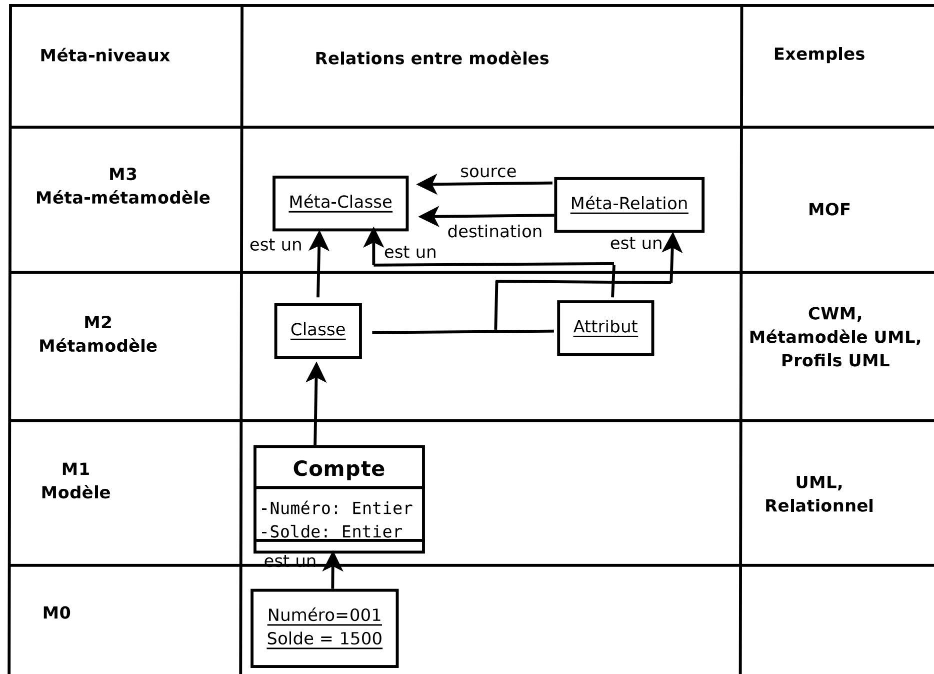


FIG. 2.1: Les différents niveaux de modélisation.

La figure 2.1 représente la relation entre modèle et méta-modèle. M0 est directement la représentation de la réalité (un programme informatique). Dans cet exemple, les variables ont des valeurs qui leurs sont affectées. Le niveau M1 est le plus bas des niveaux d'abstraction, c'est celui que le développeur manipule. Dans cet exemple, on y trouve la déclaration des variables vues dans M0 et la notion de *Compte* qui contient ces variables. Le modèle de niveau M1 est conforme au méta-modèle de niveau M2. C'est à ce niveau que l'on définit les concepts manipulés par le développeur. Un méta-modèle particulièrement connu est UML (sur lequel nous reviendrons par la suite). Les contributions de cette thèse concernant la modélisation se situent également à ce niveau. Dans l'exemple on voit que *Compte* est une *Classe* tandis que les déclarations de variables sont des *Attributs* contenus dans cette *Classe*. Enfin, un méta-modèle, au niveau M2 est conforme à un méta-méta-modèle, au niveau M3. Ce dernier est conforme à lui-même. En suivant l'exemple, les concepts de *Classe* et d'*Attributs* sont des *Méta-Classes*, tandis que la notion de contenance est une *Méta-Relation*. Ce méta-méta-modèle se décrivant lui-même, *Méta-Classe* et *Méta-Relation* sont des *Méta-Classes* et les relations *source* et *destination* les liant sont des *Méta-Relations*.

Si de manière formelle le méta-modèle le plus haut est conforme à lui-même, en pratique pour qu'un ordinateur puisse interpréter les modèles il faut décrire la sémantique de ce méta-modèle explicitement. Une fois la signification de tous les concepts codée, la machine sera capable de lire n'importe quel modèle qui est conforme directement ou indirectement à ce méta-modèle récursif. Au passage, il convient de bien noter qu'un méta-modèle ne contient

que des informations syntaxiques (structurelles) à propos du modèle, formellement aucune sémantique n'est portée. Le modèle ne prend vraiment du sens qu'à travers son interprétation, soit par l'utilisateur à l'aide d'une documentation des concepts adjointe au méta-modèle, soit par la machine lors de la transformation de modèles. Le méta-modèle du plus haut niveau est donc particulier parce que c'est le seul dont le sens est décrit explicitement dans la machine.

2.1.1.3 Composition des vues

Ce qui rend les modèles capables de représenter facilement des systèmes complexes est en grande partie la possibilité de séparer différents aspects du système en différentes vues. Cela permet d'exprimer tous les détails techniques requis tout en permettant aussi de voir le système selon une vue globale et simplifiée. C'est aussi la possibilité de découper différentes parties du modèle selon les relations qui relient des concepts afin d'avoir des vues plus simples représentant un état ou une fonction. Par exemple, dans nos modèles de MPSoC nous avons toujours une vue dédiée à la structure globale des composants de l'application et une vue spécifique au déploiement des composants élémentaires sur des fonctions mathématiques.

La séparation des vues permet également de développer le système selon les aspects métiers. Plusieurs personnes avec des compétences différentes peuvent développer ensemble le système sans qu'elles n'aient à comprendre en détail le domaine d'application des autres. De plus, un modèle peut être la composition de plusieurs modèles, chacun conforme à un méta-modèle différent adapté à une abstraction particulière avec le formalisme le plus approprié au métier. Ce qui rend cette composition possible c'est que tous les méta-modèles sont exprimés dans un langage commun unificateur : ils sont tous conformes au méta-modèle du niveau supérieur.

2.1.1.4 Productivité

Ce qui distingue vraiment l'IDM des autres approches par modélisation est que le modèle est directement utilisé pour produire les résultats du développement. Dans les autres approches, le code reste l'élément clef qui contient l'ensemble des informations décrivant le résultat (obtenu après compilation). Le passage entre modèle et code source nécessitant toujours une intervention humaine pour ajouter de l'information. L'ingénierie dirigée par les modèles préconise que l'ensemble des informations nécessaires à la génération du système soit décrit dans les modèles.

Un modèle est productif soit parce qu'il est directement exécutable par une machine, soit parce qu'il permet de produire des artefacts exécutables. Le second cas suppose qu'il faut aussi avoir les outils capables d'interpréter le modèle et de générer à partir de celui-ci les artefacts. C'est ce qu'apportent les *transformations de modèles*. Une transformation prend en entrée un (ou plusieurs) modèle conforme à un méta-modèle et génère un (ou plusieurs) nouveau modèle conforme à un autre méta-modèle. Pour une transformation donnée les méta-modèles d'entrée et de sortie sont fixes. La figure 2.2 schématise ce fonctionnement : une transformation est définie par un ensemble de règles, les méta-modèles d'entrée et de sortie sont propres à une transformation, tandis que les modèles peuvent être tout modèle conforme aux méta-modèles.

Une transformation de modèles peut ainsi être, en fonction de son usage, l'équivalent d'un convertisseur ou d'un compilateur. Ce sont les transformations qui donnent réellement

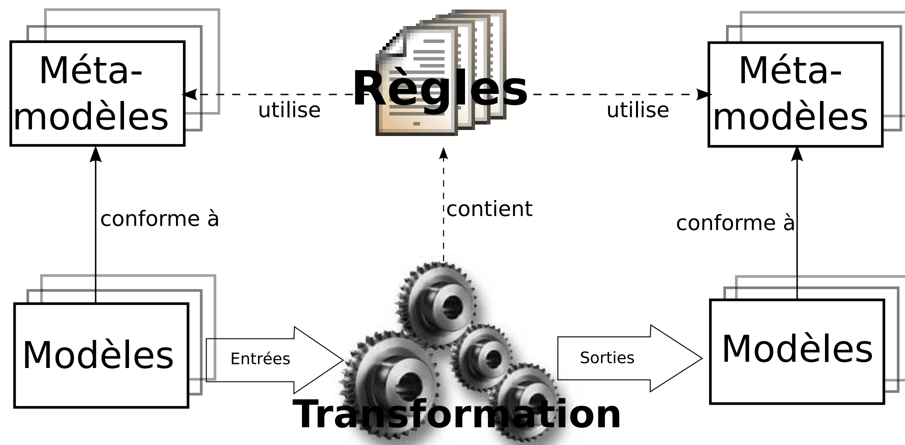


FIG. 2.2: Le développement du logiciel est productif dès la phase de modélisation à l'aide des transformations de modèles. Une transformation est un ensemble de règles définies pour des méta-modèles d'entrée et de sortie.

du sens au modèle, surtout celles qui font passer le modèle d'un niveau d'abstraction donné vers un niveau plus concret, en interprétant les concepts et les relations d'une certaine façon.

À partir du moment où le modèle peut être exécutable la notion de *plate-forme d'exécution* apparaît. Dans certaines variantes de l'IDM, et en particulier selon l'approche MDA, la plate-forme est elle-même décrite par un modèle. Le but est de permettre aux concepteurs de complètement séparer l'application de la plate-forme sur laquelle elle s'exécutera. Cela permet donc une portabilité maximale du modèle d'application, il est entièrement indépendant du système sur lequel il s'exécutera. Lors d'une transformation adéquate qui prend en entrée à la fois le modèle de plate-forme et le modèle du système, la mise en œuvre du modèle du système est restreinte selon l'espace technique de la plate-forme. Dans ces variantes, le modèle du système est alors dit *indépendant de la plate-forme* (Platform Independant Model, PIM), tandis que la version restreinte pour la plate-forme cible est dite *spécifique à la plate-forme* (Platform Specific Model, PSM). Nous verrons dans les chapitres suivants la similarité de cette approche avec le flot de conception que nous avons décidé d'utiliser.

2.1.2 Modélisation et méta-modélisation

Comme nous l'avons vu précédemment, les modèles permettent de contenir toute l'information. Cependant il ne faut pas oublier qu'avant tout l'avantage du modèle est qu'il puisse être représenté graphiquement, c'est ce qui le rend facilement accessible et c'est entre autres ainsi qu'il peut aider à gérer un système complexe (par rapport à une représentation textuelle). Bien entendu, un modèle pourrait aussi être représenté sous forme textuelle, pour peu qu'une équivalence ait été définie.

Pour donner un exemple de représentation graphique, nous allons brièvement expliquer la forme graphique d'un modèle conforme à MOF. C'est aussi l'occasion de se familiariser avec la représentation des méta-modèles qui seront proposés dans ce mémoire, tous étant conformes à MOF. La figure 2.3 décrit trois classes (les boîtes) de type « metaclass ». La classe

de gauche est nommée *Model* et contient 0 ou plus instance de la classe du bas via un lien nommé *connector* et 0 ou plus instance de la classe du haut via un lien nommé *element*. La classe du bas est nommée *Connection* et est associée à deux instances de la classe *Element* via deux liens nommés respectivement *source* et *target*. La classe du haut, nommée *Element* peut contenir 0 ou plus instance d'elle-même. Elle contient aussi trois attributs de type `String`. Notons qu'en MOF, il y a deux sortes de flèches différentes, l'une avec un losange à sa base, l'autre sans. La première sorte, comme *connector*, indique la notion de contenance (et implicitement permet la création d'instance), tandis que la seconde, comme *source*, indique la notion de référence.

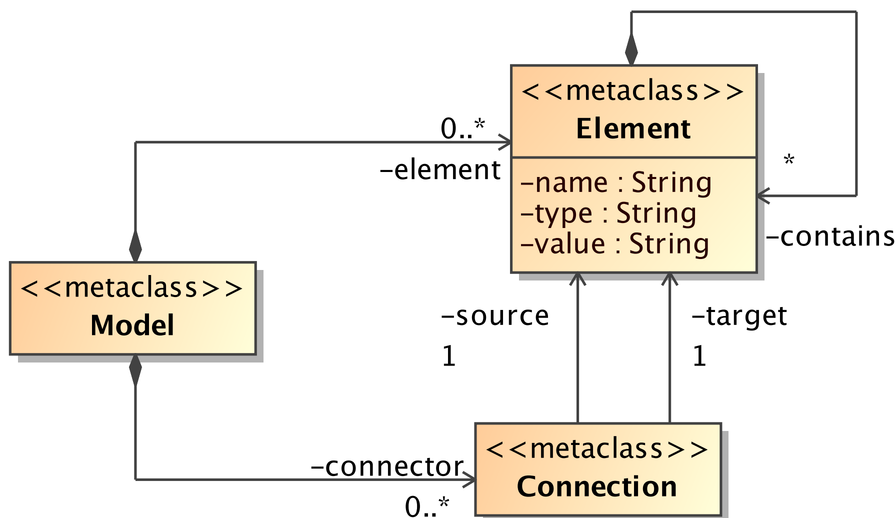


FIG. 2.3: Représentation graphique d'un modèle conforme à MOF. Le méta-modèle obtenu est très simple mais a un très large pouvoir d'expression, si large que dans la réalité il est peu utile.

Vue comme un méta-modèle, c'est-à-dire en y associant une certaine sémantique, la figure 2.3 permet de définir des modèles qui ont pour racine un *Model* contenant un certain nombre d'*Elements*. Chacun de ces éléments peut contenir un attribut (avec son nom, son type et sa valeur) et d'autres sous-éléments. La racine peut aussi contenir des connecteurs (*Connection*) qui relient deux *Elements* entre eux. C'est un méta-modèle très basique, avec seulement trois concepts. Il a néanmoins un très grand pouvoir d'expression puisque pratiquement n'importe quelle information peut être représentée selon ce schéma. Cependant, comme nous allons le voir, pour un méta-modèle la généricité n'est pas forcément un avantage.

Le rapport entre syntaxe (forme) et sémantique (signification) est tout aussi ambigu que dans un langage textuel. Comme l'a décrit Hofstadter [53], en soi la forme n'implique aucune signification particulière mais il n'existe en général qu'un petit ensemble de significations potentielles auxquelles la forme est adaptée. Pour que la forme soit adaptée à une signification donnée il faut au moins que *tout* ce qui a du sens selon les concepts apportés par la signification soit exprimable par la forme. Il est encore mieux, que la réciproque soit vraie : que *tout* ce qui est exprimable ait du sens. La liaison entre la forme et la signification, qui permet de passer

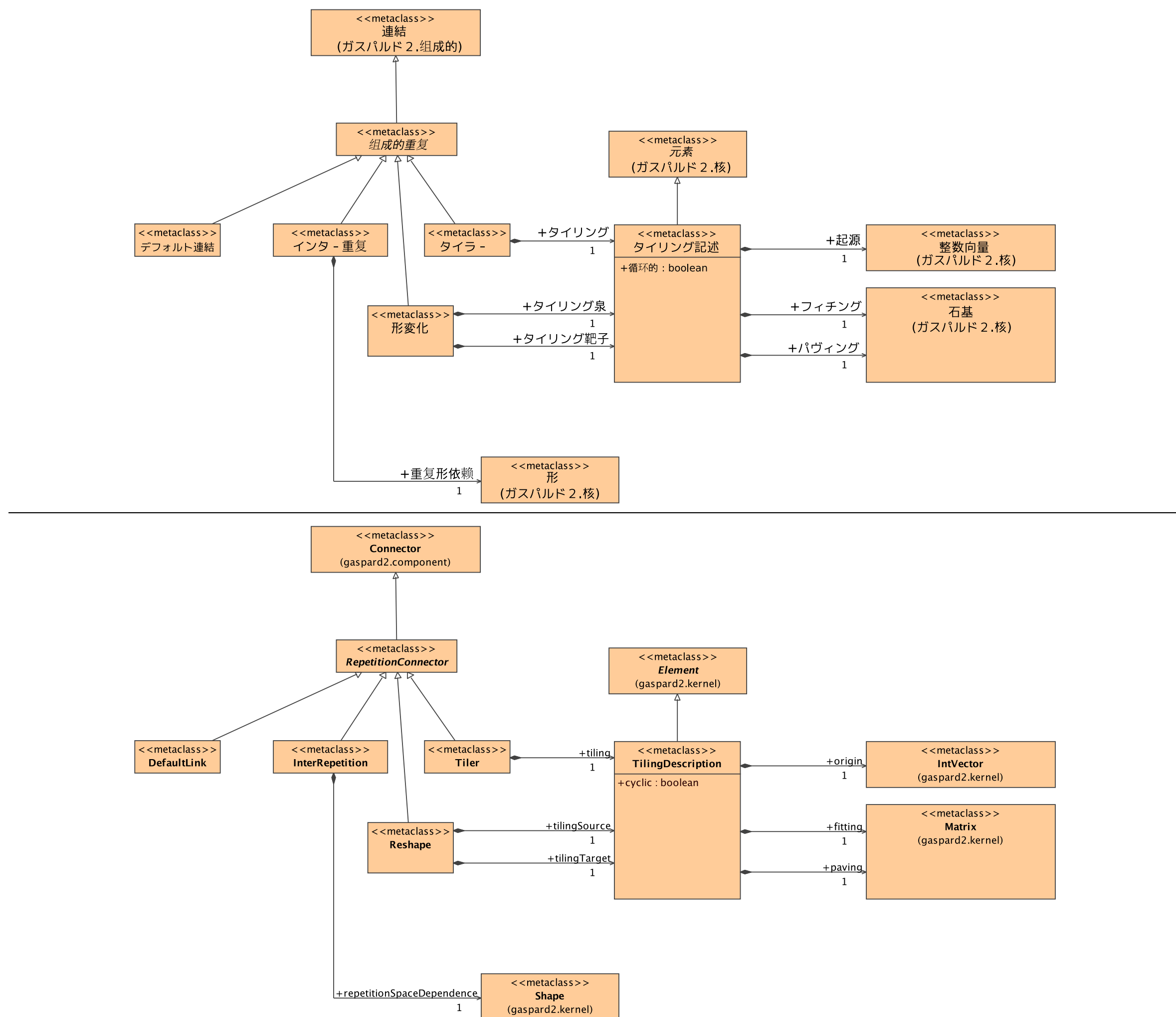


FIG. 2.4: Deux méta-modèles parfaitement équivalents. Pour un lecteur occidental il est cependant bien plus difficile d'associer une sémantique à la version japonaise car il y a beaucoup moins d'informations pour l'aider à rapprocher la forme à des concepts qu'il connaît déjà.

de l'une vers l'autre, est alors un *isomorphisme*, c'est-à-dire que chaque élément de la forme est associé à un concept différent de la signification.

Un méta-modèle définit seulement la forme et qu'il doit être accompagné d'une documentation, écrite en langage humain, qui définit l'isomorphisme. Avec cette documentation, tout ce que l'on peut vouloir exprimer est représentable à l'aide d'un modèle conforme au méta-modèle. Ce modèle, lorsqu'il sera lu, pourra être directement associé à ce que l'on souhaitait exprimer lors de son écriture. Remarquons aussi que le méta-modèle contient volontairement des informations qui guident le lecteur vers la signification. La figure 2.4 souligne l'importance du choix des noms pour aider à comprendre la sémantique d'un méta-modèle. Néanmoins, d'un point de vue formel, tel qu'interprété par une machine, les deux

modèles sont équivalents. Dans la pratique, plus l'isomorphisme est évident et implicite, plus il sera facile pour l'utilisateur de créer et de manipuler des modèles.

Lorsque l'on crée un méta-modèle on recherche une syntaxe qui représente le plus précisément possible l'ensemble des significations que l'on souhaite pouvoir exprimer. Un bon méta-modèle doit permettre la représentation de tout ce qui est sémantiquement correct tout en restreignant au maximum la possibilité d'exprimer des choses incohérentes. De manière générale il est préférable que l'utilisateur ne puisse pas représenter quelque chose d'incohérent, même si cela est souvent difficile à empêcher totalement.

2.1.2.1 UML et la méta-modélisation

UML [78] (Unified Modeling Language) est sans aucun doute le méta-modèle le plus utilisé au monde. Standardisé par l'OMG, il a pour but de permettre la modélisation de n'importe quel système informatique… champ pour le moins assez vaste. Sa force est de permettre de schématiser pratiquement tout et qu'un grand nombre de gens savent le lire (au moins en partie). Bien qu'il ait été créé par le même organisme qui a proposé MDA, une variante de l'IDM, il n'est pas directement utilisable pour faire de l'IDM. Son principal inconvénient est de ne pas permettre de produire directement un système car sa sémantique n'est pas suffisamment précise (malgré les tentatives d'éclaircissement [86]). En particulier, il existe un grand nombre de points de variation sémantique, où la signification est laissée à l'appréciation de l'utilisateur. L'IDM est bien plus jeune qu'UML, on peut la voir comme une généralisation de l'usage d'UML.

À l'heure actuelle, lorsqu'on utilise UML dans le cadre de l'IDM c'est souvent via son mécanisme interne de méta-modélisation appelé *profil*. Un profil est une collection d'extensions et, éventuellement, de restrictions qui décrivent un domaine particulier. Une extension est appelée *stéréotype*, elle spécialise une ou plusieurs classes UML et peut contenir des attributs supplémentaires appelés *tagged values*. Avec un profil, on a accès essentiellement au même pouvoir d'expression qu'avec l'utilisation de MOF mais en plus on a accès à tous les concepts déjà disponibles dans UML.

L'usage d'un profil permet de se baser sur un certain nombre de concepts et de représentations graphiques qui sont largement répandus grâce à la notoriété d'UML. On se sert en général du profil pour introduire les concepts manquants pour les systèmes que l'on cherche à représenter et pour fixer les points de variation sémantique. Un utilisateur connaissant UML qui aborde un modèle utilisant un profil peut directement comprendre une partie du modèle. L'autre avantage de l'usage des profils UML pour l'IDM est qu'il existe déjà de nombreux outils pour manipuler les modèles UML, en particulier il est directement possible de modéliser à l'aide d'une représentation graphique. Ce sont d'ailleurs les raisons pour lesquelles le méta-modèle de plus haut niveau que nous utilisons dans nos travaux ici présentés est également disponible sous forme de profil UML. Cela permet aux concepteurs de travailler avec des outils standards et de se baser sur des notions répandues. À l'aide d'une transformation de modèles, on peut passer du modèle conforme au profil UML à un modèle conforme au méta-modèle équivalent (sur lequel les autres transformations de modèles se basent).

2.1.3 Transformation de modèles

Nous l'avons déjà mentionné, ce qui distingue vraiment l'IDM des autres approches basées sur les modèles, est que le modèle serait directement productif. Les transformations de modèles permettent cette productivité. D'un point de vue utilisateur, une transformation prend un ou plusieurs modèles en entrée et génère un ou plusieurs modèles en sortie (cf figure 2.2). Aucune restriction ne s'applique au traitement effectué pour passer des entrées aux sorties, si ce n'est qu'il ne doit pas avoir besoin d'autres informations. Les transformations peuvent donc trouver leur place dans un grand nombre d'usages. Avant d'aborder les différents types de transformations existants, notons ce qui les distingue de programmes classiques.

La spécificité des transformations est que les entrées et les sorties sont exprimées *explicitement*, sous forme de méta-modèles. Un des avantages que cela apporte est que l'espace de travail d'une transformation est ainsi clair et formellement défini. Cela contraste avec les programmes classiques où les entrées et sorties sont définies implicitement dans leur code par le programmeur. De plus, l'enchaînement de plusieurs transformations est facilité parce qu'il est simple de détecter que la sortie d'une transformation n'est pas entièrement conforme au méta-modèle d'entrée de la transformation suivante, ce qui peut vite être difficile à vérifier lors de l'évolution d'une chaîne de programme. Notons cependant qu'il n'est pas possible d'avoir exclusivement des entrées et des sorties sous forme de modèle : notre monde n'est pas un modèle ! Il existe donc des transformations spéciales qui font le passage entre une représentation sous forme de modèle et une représentation non modélisée (par exemple du texte). Usuellement, ces transformations servent à générer les artefacts d'implémentation à partir du résultat d'une chaîne de transformations. Elles sont appelées transformations *modèle vers code*. Normalement, comme entre le modèle d'entrée et le texte obtenu il n'y a pas de différence sémantique, on peut aussi parler de la transformation comme un *pretty-printer*.

2.1.3.1 Expression de transformations de modèles

L'IDM préconise l'expression de transformations de modèles selon une approche ambitieuse : la décomposition en règles. Le principe de base de cette approche consiste à définir la transformation comme un ensemble de règles. On pourrait résumer l'utilité des règles par la maxime « diviser pour mieux régner ». Si nous avons attribué l'adjectif *ambitieuse* à cette approche, c'est qu'à la fois elle doit permettre de résoudre aisément le problème de la complexité, et qu'il n'est jamais facile d'obtenir un ensemble cohérent uniquement par l'assemblage de petits bouts. Les règles peuvent être écrites en langage impératif, où l'on indique *comment* faire la transformation, ou en langage déclaratif, où l'on indique *quoi* obtenir à partir d'entrées données. Dans l'IDM, c'est souvent sous forme déclarative que les règles sont exprimées. Dans ce cas, chaque règle a un patron d'entrée et un patron de sortie. Un patron (*pattern* en anglais) est une structure générique qui peut correspondre à un fragment de modèle. Le patron d'entrée définit tous les fragments du modèle source sur lesquels la règle doit être appliquée. Le patron de sortie quant à lui définit la structure que suivra chaque fragment généré par la règle. Ces patrons sont conformes au méta-modèle d'entrée ou de sortie mais certaines parties sont remplacées par des *variables de liaison* qui peuvent correspondre à n'importe quelle valeur d'un modèle. Pour établir la liaison entre ces deux patrons, une partie *logique* [30] spécifie le lien entre les valeurs des classes du patron de sortie et celles du patron d'entrée. En particulier, elle définit les calculs nécessaires à l'obtention des

variables du patron de sortie à partir des variables de liaisons du patron d'entrée. La figure 2.5 est un exemple de représentation de règle : la partie gauche définit le patron d'entrée, la partie droite définit le patron de sortie, la partie centrale définit les calculs nécessaires (via l'appel d'autres règles). Ici, la chaîne de caractères `ieq` est calculée à l'aide de l'appel à la règle `dist2phWraper` et des valeurs `ts`, `tt`, `ps` et `rs`.

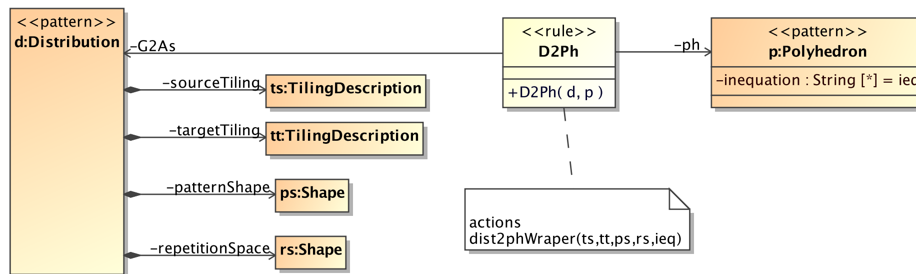


FIG. 2.5: Exemple de règle de transformation (définie avec le profil TrML [40]). Au centre se trouve la partie d'action, séparant le patron d'entrée à gauche du patron de sortie à droite.

Si l'on souhaite suivre l'IDM de manière orthodoxe, une transformation est un système en soi et l'IDM préconise la représentation des systèmes sous forme de modèle, les transformations se doivent donc d'être écrites elles-mêmes sous forme de modèle. L'approche par langage déclaratif se prête d'ailleurs très bien à une modélisation graphique. Les patrons sont en effet d'autant plus lisibles qu'ils sont représentés de manière similaire aux modèles.

Actuellement, il n'y a pas de consensus sur les outils ou les méta-modèles à utiliser pour écrire et exécuter une transformation de modèles. Il existe de nombreuses propositions aussi bien par des chercheurs que par des industriels (Kermeta [97, 74], ATL [59], ModelMorf [34], TrML [40]...) mais aucune n'est plébiscitée par la communauté. Citons quand même QVT [79] (Query, View, Transform) qui a été récemment standardisé. C'est un méta-modèle de description de transformations comportant trois langages : deux déclaratifs et un impératif. Certains outils sont capables d'exécuter la partie impérative mais aucun ne supporte entièrement la partie déclarative (la plus novatrice et intéressante), ce qui rend ce standard encore peu utilisé. Avec un point de vue très pragmatique, il existe aussi EMF [37] (Eclipse Modeling Framework) qui est une bibliothèque Java pour créer et modifier des modèles. Stable et répandue, elle est de trop bas niveau pour représenter directement les règles mais peut devenir la plate-forme d'exécution de transformations décrites à plus hauts niveaux.

2.1.3.2 Types de transformations

Tom Mens et al. [69] ont publié une taxonomie des transformations de modèles qui permet de situer les différents critères de variation des transformations. On peut noter en particulier la notion de transformation *exogène* et *endogène* : si les modèles d'entrée et de sortie sont conformes au même méta-modèle alors la transformation est endogène (comme par exemple l'optimisation d'un système), autrement la transformation est exogène (comme par exemple la traduction d'un langage vers un autre). Ils font aussi la distinction entre les transformations dites horizontales qui génèrent un modèle au même niveau d'abstraction que celui d'entrée et celles qui passent d'un niveau d'abstraction à un autre (par exemple la génération de code).

Czarnecki et Helsen [30] ont aussi publié une taxonomie, mais plus orientée sur les différences du langage d'écriture de transformation. Ils mettent en avant par exemple le type des variables utilisées, la manière de décrire un patron, s'il est possible de réduire le domaine de travail à un sous-ensemble des méta-modèles d'entrée et de sortie, ou encore l'ordre dans lequel sont exécutées les règles de transformation.

Il est intéressant de mentionner que, comme le font les deux taxonomies proposées, l'on peut aussi différencier les transformations unidirectionnelles à celles bidirectionnelles. On trouve dans la littérature [100, 92] un certain nombre de travaux pour pouvoir intervertir modèles d'entrée et de sortie automatiquement. Ceci est rendu possible en partie grâce à l'usage d'un langage déclaratif pour l'écriture des règles de transformation, que nous allons détailler dans la section suivante. Le principe d'une transformation bidirectionnelle est que les *mêmes* règles peuvent être utilisées pour passer d'un méta-modèle A vers un méta-modèle B que du méta-modèle B vers le méta-modèle A. Cette lecture à double sens de la transformation est rendue possible par un moteur d'exécution spécial. Bien sûr cela nécessite également que la transformation n'effectue que des actions qui puissent être inversées (par exemple une somme de deux nombres ne l'est pas).

Les transformations de modèles peuvent aussi proposer la traçabilité des modifications. La notion de *trace* est basée sur l'idée de garder l'information permettant d'associer à chaque élément du modèle de sortie les éléments du modèle d'entrée qui ont induit sa création [12]. Il existe de nombreux usages d'une telle information, par exemple, dans une chaîne de compilation, cela permet le débogage de l'implémentation tout en fournissant à l'utilisateur une vue dans le modèle de conception. Encore une fois ces informations peuvent être générées directement par le moteur d'exécution de la transformation, le développeur de la transformation n'ayant ainsi pas à ce soucier de cet aspect.

2.1.4 Conclusion

Nous avons montré les intérêts de l'ingénierie dirigée par les modèles. Elle permet de gérer la complexité lors de la conception et la maintenance de larges systèmes, ce qui en pratique permet d'accélérer le développement et/ou de réduire les coûts de conception. L'approche préconise l'usage systématique des modèles pour la représentation d'informations. Les modèles, outre le fait qu'ils se prêtent bien à une représentation graphique, permettent de facilement présenter en vues subjectives l'information adaptées à l'utilisateur (représentation métier) ou à l'étape de développement (vue globale/détaillée). De plus, l'IDM favorise une approche plus formelle en forçant à expliciter la syntaxe de tous les modèles ainsi que l'espace d'entrée et de sortie des transformations. Ces transformations peuvent être décrites dans un langage de description déclaratif, qui est à la fois décomposé en règle et graphique.

Après avoir détaillé les fondements l'IDM comme approche de conception, dans la section suivante nous allons brosser un aperçu du contexte d'application de cette thèse : les systèmes sur puces multiprocesseurs.

2.2 Systèmes sur puces

Avec l'évolution incessante de la technologie des semi-conducteurs, il est désormais possible de placer des centaines de millions de transistors sur une seule puce de silicium [58]. Ainsi, un système complexe (avec unités de calcul, mémoires, convertisseurs numérique/a-

nalogique, etc.) peut être intégré entièrement sur une telle puce. Ces systèmes sont alors nommés *systèmes sur puces* ou *SoC* (System-on-Chip).

2.2.1 Organisation et usage des SoC

Un SoC est l'équivalent d'un ordinateur complet placé sur une puce. Grâce à la technologie des semi-conducteurs, il peut contenir pratiquement n'importe quel type de composant :
- des processeurs (des microprocesseurs, des DSP...),
- des mémoires (RAM, ROM, Flash...),
- des éléments de connexion inter-composants (bus, crossbar...),
- des interfaces numériques (PCI, USB, ethernet...),
- des convertisseurs analogiques/numériques,
- des éléments reconfigurables (tels que les FPGA).

La figure 2.6 présente un exemple de SoC avec ses nombreux composants reliés entre eux.

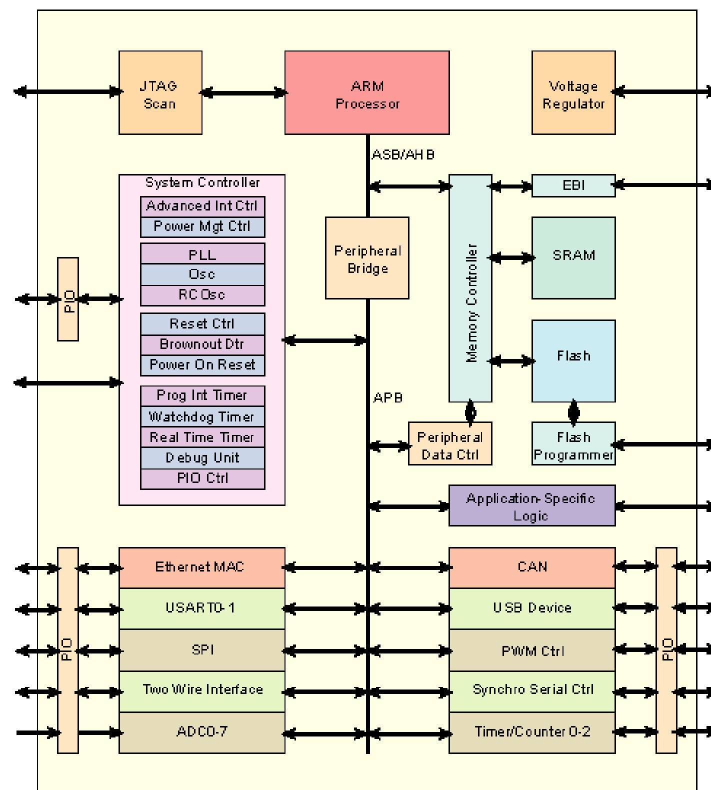


FIG. 2.6: Exemple de système sur puce, à base d'un processeur ARM et de nombreuses interfaces de communications. Image provenant de Wikipedia.

La spécificité des SoC est qu'ils sont conçus pour une application particulière. Contrairement aux ordinateurs communs qui peuvent s'adapter à de très nombreuses et diverses tâches, les SoC sont ciblés pour une tâche en particulier. La partie matérielle est optimisée pour ne contenir que les composants nécessaires à cela, et elle peut avoir des accélérateurs matériels pour une fonction de calcul spécifique qui serait trop longue à exécuter sur le microprocesseur embarqué (FFT, DCT, encryptage...). La partie logicielle est écrite en tenant

compte des spécificités particulières de la partie matérielle (vitesse d'exécution, disponibilité mémoire, accélérateurs matériels. . .). En général, il y a peu de couches logicielles, entre autres le système d'exploitation est minimaliste ou inexistant. Cette spécialisation a un coût : pour chaque nouvel usage, il faut concevoir un nouveau SoC.

En plus du coût de conception, la fabrication de la puce requiert la création d'un *masque*, dont le prix peut atteindre l'ordre des millions d'euros. Cela implique qu'il est rarement possible de faire du prototypage : le premier jet doit être le bon. La prévention des bogues doit être particulièrement stricte. Dans la phase de conception, la partie de vérification joue donc un rôle extrêmement important.

Par rapport à l'utilisation de composants sur des puces séparées et disposées sur une carte électronique, l'usage de SoC apporte de nombreux avantages. D'un point de vue physique, le SoC permet de réduire l'espace utilisé, ce qui est particulièrement recherché lors de la conception des appareils électroniques grand public (téléphone portable, lecteur DVD portable. . .). Les composants étant beaucoup plus proches des uns des autres, la performance du réseau de connexion inter-composant s'en trouve largement améliorée (car sa fréquence peut être augmentée). L'usage d'une seule puce permet aussi de réduire la consommation énergétique, ce qui est un point important dans tous les systèmes embarqués. Enfin, sur de gros volumes de fabrication, les SoC sont meilleur marché que l'assemblage sur carte PCB, grâce au coût d'intégration plus faible. En raison de ces avantages, aujourd'hui on retrouve les SoC dans presque tous les systèmes embarqués : les appareils électroniques grand public, les radars, les appareils de contrôle dans les transports. . .

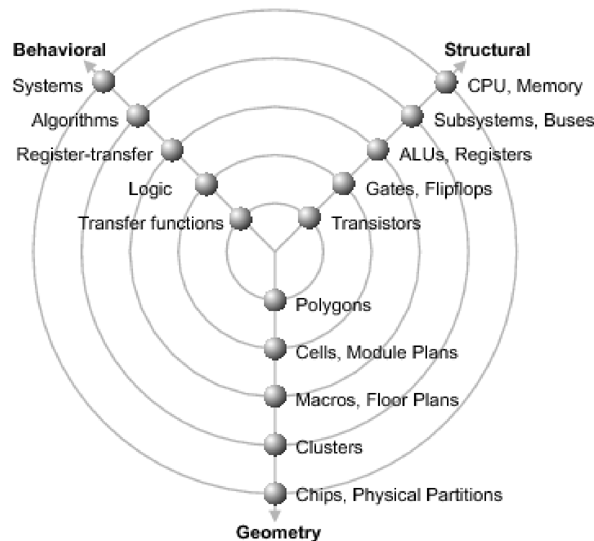


FIG. 2.7: Schéma d'organisation de raffinement de la conception en Y par Gajski et Kuhn [47]. Les trois axes correspondent aux vues structurelle, comportementale et géométrique du système.

2.2.2 Conception des SoC

Chaque SoC étant spécifique à une application donnée, la conception joue un rôle primordial puisqu'il est nécessaire de repasser par cette phase à chaque nouvel usage. C'est à ce

niveau que les plus importantes difficultés se trouvent, et tout particulièrement que se situent les possibilités d'augmentation de la productivité.

Gajski et Kuhn [47] ont proposé le modèle en Y («*Y chart*») pour représenter les différentes étapes de la conception de SoC (figure 2.7). Chacun des trois axes correspond à une vue différente du système : structurelle (la description électronique), comportementale (la description fonctionnelle), et géométrique (le résultat physique). Plus on s'approche du centre, plus les descriptions sont précises. Les outils de développement aident au passage d'un axe à un autre, avec comme but final une représentation géométrique précise.

Un schéma d'organisation usuelle de la conception de SoC s'inspire de ce modèle en Y. Comme on peut le voir sur la figure 2.8, les parties matérielles et logicielles sont développées parallèlement. Pendant qu'un groupe des concepteurs travaille sur l'aspect comportemental de l'application, un autre groupe conçoit l'architecture matérielle de puissance suffisante pour une exécution sans ralentissement. Les deux équipes ont des compétences différentes, adaptées à leur domaine de travail. La conception en parallèle a surtout l'avantage d'économiser en temps : au lieu d'attendre que l'architecture soit terminée pour commencer la conception de la partie logicielle, on conçoit dès le début la partie logicielle en restant le plus possible indépendant du matériel. Durant cette phase de conception, afin d'augmenter la productivité, les développeurs ont recourt à la réutilisation de composants déjà existants. Nommées *IP block* ou simplement *IP* dans le monde matériel et plutôt *fonction de bibliothèque* dans le monde logiciel, ces briques de base permettent de factoriser grandement les travaux. Ils correspondent à des fonctions plus ou moins couramment utilisées. Ils peuvent provenir des développements internes précédents, ou bien être achetés auprès d'autres entreprises.

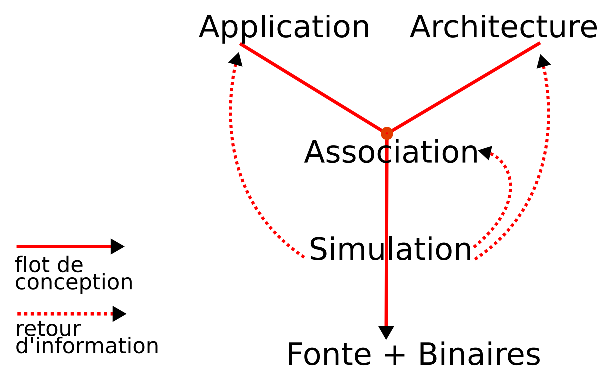


FIG. 2.8: Schéma d'organisation de la conception en Y. Les parties logicielles et matérielles sont développées parallèlement. À partir des informations de simulation, les premières étapes peuvent être réitérées.

Après que le logiciel et le matériel ont chacun atteint un niveau de développement suffisant, vient l'étape d'association qui consiste à placer l'application sur l'architecture. Il s'agit de spécifier quelle tâche sera exécutée par quel processeur, éventuellement choisir d'utiliser un DSP ou un accélérateur matériel pour certaines tâches particulières. Il faut également spécifier l'allocation des données et des instructions sur les différentes mémoires disponibles. Cette association est à la fois topologique (placement) et temporelle (ordonnancement). De mauvaises décisions à cette étape peuvent avoir un rôle catastrophique sur les performances du système. Les points les plus décisifs sont la capacité de calcul des processeurs par rapport

aux demandes des tâches et la localité des données : il s'agit de maîtriser la distance des communications nécessaires entre la mémoire contenant les données et le processeur exécutant la tâche qui les manipule.

Une fois l'association achevée, la description du système est complète. Il est alors possible de la vérifier. Plusieurs critères doivent être observés :

- **fonctionnel** : s'assurer que le SoC répond toujours correctement aux entrées, tel que défini dans le cahier des charges (au préalable il a déjà été vérifié que le logiciel et le matériel indépendamment passent le critère fonctionnel),
- **temps d'exécution** : s'assurer que par rapport aux entrées, les sorties correspondantes sont générées dans un temps *toujours* compris dans un intervalle prédéfini (réponse ni trop lente ni trop rapide),
- **physique** : s'assurer que le comportement physique du SoC est compatible avec son usage, tel que la consommation électrique ou la température de fonctionnement.

Une première approche pour s'assurer de ces critères est la vérification formelle. Elle consiste à vérifier mathématiquement un critère particulier à partir de la description précise de l'architecture et du logiciel. Extrêmement efficace, cette approche est cependant souvent difficile à mettre en œuvre car il est rare de pouvoir décrire avec suffisamment de détails chacun des composants logiciels et matériels. Par exemple, les mécanismes de cache des processeurs ou bien les politiques de gestion des contentions sur les réseaux de communication sont réputés être très difficile à décrire formellement.

Une seconde approche est la co-simulation (*co-* soulignant le fait qu'à la fois le logiciel et le matériel sont testés). Elle consiste à simuler l'ensemble du système et de fournir au développeur autant d'information que possible sur le déroulement de l'exécution. Si suffisamment de détails sont fournis au simulateur, il peut être capable d'indiquer en plus des résultats fonctionnels, le temps d'exécution, la consommation, etc. La simulation permet aussi d'observer le déroulement de l'exécution au cours du temps, y compris les états internes des composants à chaque instant. Si le système ne correspond pas complètement aux attentes, ce sont ces informations qui vont permettre d'aiguiller les concepteurs vers les points à modifier dans la partie logicielle, la partie matérielle, ou bien au niveau de l'association.

Il existe plusieurs niveaux d'abstraction pour la simulation. Chaque niveau offre un compromis différent entre la vitesse de simulation et la précision des résultats obtenus. La simulation peut aller d'une simulation plus rapide que la vitesse réelle retournant des informations grossières à une simulation prenant plusieurs heures pour une seconde simulée mais avec des observations disponibles pour chaque cycle d'horloge. Au cours de la conception on fait usage de plusieurs niveaux d'abstraction. Au fur et à mesure que le système est validé, on passe d'une simulation rapide à une simulation plus précise pour affiner les corrections.

C'est en général seulement à la suite de nombreuses itérations entre la modification du système et la simulation que l'on obtient un système convenable. Les informations de simulation sont utilisées pour corriger ou améliorer les parties logicielle, matérielle et l'association. En particulier, lorsque l'on fait évoluer le matériel pour trouver l'architecture la mieux adaptée, on parle d'*exploration d'architecture*. Une fois le système validé, vient l'étape finale de création du masque de la puce, qui permettra la fonte. La version binaire du logiciel est ensuite ajoutée en mémoire ROM ou Flash.

2.2.3 Co-simulation logiciel-matériel

La co-simulation permet à la fois de valider le système et d'observer le comportement des composants et de leur interaction tout au long de l'exécution de la simulation. Le but d'une co-simulation logiciel-matériel est de tester le plus tôt possible dans une chaîne de conception l'interaction entre le logiciel et le matériel [49]. En particulier elle doit permettre de ne pas avoir à réaliser un prototype physique de l'architecture matérielle pour valider le logiciel sur ce matériel. Après avoir détaillé les différents niveaux d'abstraction de simulation usuellement utilisés, nous présenterons différentes méthodes proposées dans la littérature pour accélérer la simulation logiciel.

2.2.3.1 Différents niveaux d'abstraction pour la simulation

Lors de la conception d'un SoC, différents niveaux d'abstraction de simulation sont utilisés. Dans un premier temps, on recherche une simulation rapide. On utilise alors une abstraction élevée mais les informations obtenues (vitesse d'exécution, consommation. . .) sont peu précises. Puis, alors que la description du système est raffinée, des simulations plus précises sont employées pour estimer le SoC de façon plus exacte. Chaque niveau d'abstraction est un compromis entre précision des résultats et vitesse de simulation. Augmenter le niveau d'abstraction d'une simulation correspond à définir un ensemble de simplifications qui suppriment des détails du fonctionnement réel du système tout en gardant le comportement global le plus réaliste possible. Actuellement, il n'existe pas d'échelle officielle et universellement reconnue spécifiant un nombre déterminé de niveaux d'abstraction. Chaque outil de simulation, voire chaque équipe, définit son propre ensemble de niveaux d'abstraction. Néanmoins, d'un point de vue global, il est possible de les regrouper en quelques niveaux différents.

Il existe plusieurs propositions d'ensemble de niveaux. Entre autres, Donlin [35] a proposé un ensemble de niveaux, dont la terminologie est maintenant souvent reprise, en particulier dans le cadre de simulations SystemC. Il définit cinq niveaux d'abstraction :

- **ALG** (algorithmique) Seul le comportement du système est simulé.
- **CP** et **CPT** (Processus Communicants) Le comportement du système est découpé en tâches et il est possible de quantifier les communications entre elles. L'architecture ne joue pas de rôle à l'exception de la prise en compte du parallélisme pour la découpe en tâches. Éventuellement, des annotations temporelles sont spécifiées (CPT).
- **PV** et **PVT** (Vue du Programmeur) L'architecture est représentée sous forme de composants. En particulier les composants de communication sont utilisés comme des mécanismes de transport et les politiques d'arbitrage sont appliquées. De plus, tous les registres des composants sont représentés. Éventuellement, des annotations temporelles sont spécifiées (PVT), plus précises qu'en CPT, elles sont suffisamment fines pour, par exemple, observer le coût des contentions lors de transactions.
- **CA** (Précis au cycle près) Parfois également nommé CABA, ce niveau permet de capter les détails micro-architecturaux, y compris avec une précision au bit près pour les interfaces. Il contient toujours des informations temporelles, précises au cycle d'exécution près.
- **RTL** (Niveau des transferts registre) L'implémentation et l'architecture sont complètement spécifiées. Pour chaque composant, le comportement à l'intérieur d'un cycle est simulé.

Les niveaux CP/CPT, PV/PVT, et CA font tous partie du niveau générique TLM (modéli-

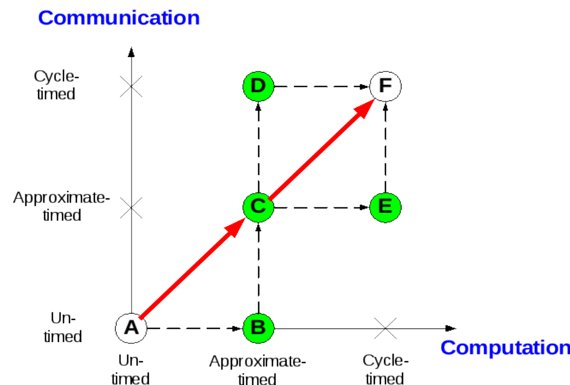


FIG. 2.9: Exemple de différents niveaux d'abstraction, qui se distinguent selon le niveau d'abstraction des communications et des calculs. Les flèches indiquent les différentes étapes possibles de raffinement.

sation au niveau des transferts). Par ailleurs, notons que pour préciser un niveau d'abstraction, il est courant de dissocier la précision de simulation des communications et celle des calculs, comme l'ont fait Cai et Gajski [24] et tel que représenté dans la figure 2.9. Dans ce schéma, les nœuds correspondent à différents niveaux d'abstraction. Ils sont placés selon deux axes : la précision de simulation des communications et celle des calculs. Les flèches symbolisent différents flots possibles de raffinement. Tous mènent à F, le niveau d'abstraction le plus bas et le plus précis.

2.2.3.2 Co-simulation avec abstraction de la partie logicielle

Habituellement, lorsque l'on parle de niveau d'abstraction de la simulation, on désigne uniquement l'abstraction utilisée pour la partie matérielle. Quel que soit le niveau d'abstraction, le logiciel est toujours représenté avec la précision la plus fine, en langage machine du processeur cible. Ainsi, au préalable à la simulation, le logiciel est compilé, puis au début de la simulation le résultat de la compilation est placé dans les mémoires simulées. Les composants processeurs interprètent alors chaque instruction, exprimant ainsi le comportement de la partie logicielle. Un certain nombre de travaux ont cherché à éviter de simuler l'application à une telle précision. Le principal objectif étant d'accélérer la simulation (en évitant la coûteuse étape d'interprétation de chaque instruction). Un deuxième objectif parfois recherché est une meilleure observation du comportement de l'application : il est toujours difficile pour un programmeur de faire le lien entre une suite d'instructions et les concepts manipulés dans le code de l'application.

Une manière d'abstraire l'application est de ne simuler qu'un ensemble d'échantillons de l'exécution du programme. Lors d'une simulation initiale le comportement complet de l'application est enregistré : il correspond principalement aux différents accès mémoires et à l'état du processeur (oisif ou non). Lors des autres simulations, seul un certain nombre d'échantillons sont rejoués. Le résultat de la simulation peut être interpolé à partir des résultats de simulation de chaque échantillon. L'inconvénient de cette méthode est qu'il est difficile de déterminer précisément quelles sont les phases les plus représentatives de

l'application et la part que chaque phase représente dans l'exécution globale. Certaines approches sélectionnent les échantillons aléatoirement [99] mais pour améliorer la précision d'autres approches cherchent à sélectionner les échantillons plus représentatifs en déterminant les *phases* du programme [87].

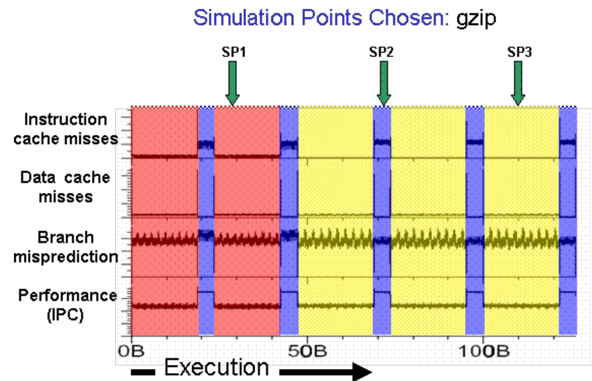


FIG. 2.10: Exemple de sélection de phases de l'application dans l'outil SimPoint. Les trois phases simulées seront SP1, SP2, et SP3.

Par exemple, l'outil SimPoint [51] utilise une pré-simulation et un algorithme de groupage pour détecter les phases les plus significatives et mesurer leur importance sur l'exécution totale. Un exemple de sélection est présenté figure 2.10 : l'application est découpée en trois phases différentes et seulement un échantillon correspondant à chaque phase sera simulé (les échantillons SP1, SP2, et SP3). Ekman et Stenström [39] ont proposé une piste d'extension à cette méthode pour simuler une architecture multiprocesseur. Un des problèmes de cette approche est la difficulté à replacer l'ensemble du matériel dans un état équivalent à celui dans lequel il se trouvait au début de l'échantillon simulé. Par ailleurs si cette approche permet d'estimer les propriétés non-fonctionnelles du système (temps d'exécution, consommation. . .), elle ne permet pas de valider le comportement du système ni de déboguer les composants. Mais surtout, la grande difficulté vient de la sélection des échantillons représentatifs, en particulier lors de communications entre les tâches sur les différents processeurs.

Une autre approche consiste à ne pas utiliser un simulateur d'instructions pour le processeur mais à exécuter directement le logiciel sur un processeur physique. La plupart des approches proposées reposent sur une organisation de la co-simulation en utilisant d'un côté un simulateur de matériel et d'un autre côté un simulateur du logiciel, qui se charge de contrôler l'exécution de l'application. Dans le simulateur de matériel, les processeurs sont remplacés par des composants connectés au simulateur logiciel. La connexion permet de transmettre des informations dans les deux sens : accès mémoire, interruptions matérielles. . .

Benini et al. [18] ont proposé une implémentation reposant sur ce type d'approche en utilisant GDB, un débogueur très répandu, pour contrôler l'application qui est compilée pour le processeur hôte de la simulation. Des accélérations jusqu'à un facteur de 10 ont été mesurées (pour la partie logicielle). Fummi et al.[46] ont présenté l'usage de cette approche en compilant et exécutant l'application sur le processeur cible. Pour cela ils utilisent une carte de développement et contrôlent l'exécution depuis l'ordinateur de simulation via le débogueur. À intervalles réguliers, une synchronisation du temps simulé est réalisée entre les différents simulateurs.

Lorsque le logiciel doit s'exécuter par dessus une couche de système d'exploitation, Honda et al. [54, 96] ont montré la possibilité d'ajouter un niveau d'abstraction supplémentaire dans la simulation logicielle. Le simulateur logiciel simule alors le processeur *et* le système d'exploitation, qui dans leur étude était µITRON[1]. Il est ainsi possible d'exécuter séparément les différentes tâches et d'accélérer la simulation. Un des autres avantages est de pouvoir observer et déboguer les tâches avec la panoplie d'outils disponibles sur la plate-forme de simulation. Cette méthode a également été utilisée par Taillard [94] avec Linux, mais la faible précision concernant les accès mémoires n'a pas permis d'estimer des informations précises sur l'architecture, telles que les contentions. En effet, l'application étant directement compilée depuis le code source en C ou en C++, il est très difficile d'extraire des informations complète sur l'organisation des accès mémoire.

Après avoir détaillé l'usage, les principes de conception, et les méthodes de validation des SoC, nous présentons les défis auxquels l'industrie fait face actuellement.

2.2.4 Défis actuels

Alors que les avancées technologiques des semi-conducteurs permettent toujours d'accroître la densité des transistors sur une puce (selon la loi de Moore), la demande d'applications nécessitant toujours plus de puissance de calcul se poursuit. Alors que ces deux aspects concordent, l'industrie des SoC peine à suivre en raison des défis que ces augmentations poussent à relever, en particulier vis-à-vis de la conception. Le groupe de l'ITRS [57] (International Technology Roadmap for Semiconductors) souligne depuis plusieurs années les nouvelles difficultés auxquelles l'industrie fait face.

Complexité du silicium Le nombre croissant de technologies différentes qui peuvent être mises sur une puce (logique, mémoire, partie analogique, partie reconfigurable, composant électro-mécanique...) demande beaucoup de compétence aux ingénieurs pour gérer leur intégration. L'augmentation de la taille ajoute des contraintes supplémentaires car les délais de transmission entre les composants rendent difficile la propagation de l'horloge sur l'ensemble du SoC sans réduire la fréquence. Par ailleurs, le silicium impose qu'une fois fondue la puce ne sera plus modifiée, il y a donc une nécessité de certification très poussée pour détecter le maximum d'erreurs avant la phase de fabrication.

Complexité des systèmes Tandis que les besoins applicatifs des SoC sont de plus en plus complexes, la réduction de la finesse de gravure demande en même temps de connaître plus d'informations sur le système. Les difficultés de traiter des applications de grande taille sont d'autant plus présentes au niveau de la partie logicielle. À l'heure actuelle, près de 80% du coût la conception revient au logiciel. Afin de pouvoir gérer la complexité supplémentaire, les pistes sont d'une part d'améliorer les approches de conceptions pour pouvoir utiliser mieux la hiérarchie et élever le niveau d'abstraction et d'autre part de créer des outils prenant mieux en compte les détails de l'électronique (tels que les interférences, les délais de transmission...).

Productivité Alors que la taille et la complexité de l'application augmentent, il faut être capable de maîtriser les coûts et les temps de développement. De plus, dans le domaine des systèmes embarqués, l'innovation prime et donc le marché est très sensible aux délais. Pour

[1] http://www.sakamura-lab.org/TRON/ITRON/SPEC/mitron4-e.html

un SoC, il faut compter entre six mois et un an entre l'idée originale, telle que la publication d'un nouveau standard, et la mise sur le marché. C'est ce qui est appelé le *temps de mise sur le marché* (TTM pour *time-to-market* en anglais). Il faut pouvoir garder ce TTM court. Cependant, la taille d'une équipe d'ingénieurs pour le travail sur un SoC est limitée : plus il y a de personnes et plus il faut découper les tâches et gérer les inter-actions entre les travaux de chacun. Pour cela, la productivité des ingénieurs doit être améliorée, en général grâce à l'introduction de nouvelles méthodes de travail et d'outils automatisant certaines tâches. À ce propos, l'ITRS propose un certain nombre de points clefs qui permettraient d'augmenter la productivité : la réutilisation, la vérification et le test, l'optimisation dirigée par le coût, des plates-formes de conception et d'implémentation sûres, des processus de gestion de la conception.

2.2.5 Conclusion

Les systèmes sur puces, qui sont des ordinateurs complets à l'intérieur d'une seule puce électronique, offrent de nombreux avantages par rapport à la fabrication classique sur carte. Ces avantages sont principalement matériels : espace physique réduit, faible consommation, performances améliorées, etc. Ainsi retrouve-t-on les SoC principalement dans les systèmes embarqués. Fonctionnellement, ils sont moins souples que les ordinateurs génériques : pour chaque nouvelle application il faut recommencer la conception, et une fois le SoC fondu très peu d'erreurs peuvent être corrigées. Lors de la conception, les développements du matériel et du logiciel sont simultanés, effectués par des équipes de compétence différente, ensuite vient une phase d'association de la partie comportementale sur l'architecture. Après le développement, une très importante phase itérative de vérification et de simulation commence. Ce n'est qu'une fois le système entièrement validé, tant sur les aspects fonctionnels que physiques, que le masque peut être généré.

À l'heure actuelle, alors qu'il est possible de placer toujours plus de transistors sur une puce et qu'en même temps les applications demandent toujours plus de puissance, c'est principalement la productivité des équipes de conception qui est mise à l'épreuve. Comme nous l'avons vu précédemment, L'IDM vise à améliorer la productivité des développeurs par la mise en place d'un certain nombre de méthodes. Dans la section suivante nous allons introduire Gaspard, un outil qui a pour but d'appliquer l'usage de l'IDM à la co-modélisation de SoC.

2.3 Gaspard, un outil pour la co-modélisation de SoC

L'augmentation de la quantité de transistors disponibles dans un SoC entraîne une augmentation similaire de la puissance de calcul potentielle. S'il a été longtemps possible de grossir directement la taille du processeur et de hausser la fréquence d'exécution, on assiste ces dernières années à un tournant important dans le style d'architecture matérielle. Historiquement restreint aux super-ordinateurs, le parallélisme est devenu nécessaire pour pouvoir continuer à accroître la puissance, entre autres en raison des exigences en terme de consommation d'énergie et de température [19], mais aussi en raison de la difficulté croissante à traiter efficacement une seule suite d'instructions avec toujours plus de transistors. Actuellement, l'usage de systèmes multiprocesseurs est de plus en plus évident.

Gaspard [32] est un environnement de développement qui doit permettre la co-modélisation de SoC dans le cadre de ce nouveau contexte. Gaspard est développé au sein

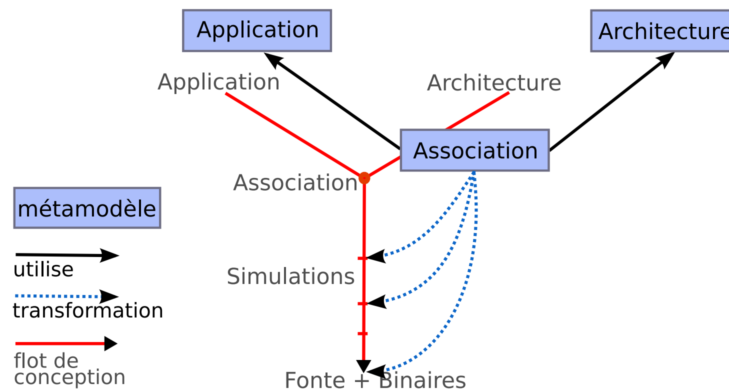


FIG. 2.11: Schéma du flot de conception en utilisant l'environnement Gaspard. L'architecture, l'application, et l'association sont spécifiées sous forme de modèles. Des transformations doivent permettre la génération de code correspondant aux différentes étapes de la création d'un SoC.

de l'équipe DaRT et il est orienté pour les applications de traitement de signal intensif. Ce type d'application est parmi ceux qui nécessitent le plus de puissance de calcul et se prête généralement bien à la parallélisation des traitements. Le traitement de signal est aussi parmi les types d'applications les plus utilisés dans les systèmes embarqués. Le principe de Gaspard est d'exploiter l'IDM pour co-modéliser un SoC multiprocesseur, voire massivement parallèle, destiné au traitement de signal intensif. Gaspard doit permettre au concepteur de SoC de modéliser indépendamment le comportement applicatif et l'architecture matérielle. En suivant le flot de conception en Y, le concepteur peut ensuite spécifier une association de l'application sur l'architecture. À partir de ces informations, des transformations de modèles permettent de générer des simulations à des niveaux de plus en plus précis, et après un certain nombre de cycles de correction par le concepteur, générer le code du SoC complet, à la fois matériel et logiciel. La figure 2.11 illustre cette organisation du développement, et fait le rapprochement avec le flot de conception en Y.

Au début de ces travaux de thèse, seul le méta-modèle permettant à l'utilisateur de spécifier le MPSoC était défini, les transformations étant encore uniquement des perspectives. Ce méta-modèle a pour origine les travaux d'Arnaud Cuccuru [29] lors de son doctorat dans l'équipe. Il est en partie inspiré de standards OMG tels que SPT (Schedulability, Performance, and Time) [81] pour représenter l'architecture matérielle, SysML (System Modeling Language) [80] pour la mise en œuvre de l'association, et UML pour l'approche composant. Il est également basé sur le langage Array-OL (Array Oriented Language) [33, 21] dédié aux applications de traitement de signal intensif, pour les notions de factorisation qui permettent de représenter de manière compacte les systèmes répétitifs réguliers. Notons au passage que le méta-modèle *Gaspard* est lui-même en partie à la base du tout jeune standard MARTE (Modeling and Analysis of Real-Time and Embedded systems) [84].

Dans la pratique, le méta-modèle *Gaspard* n'est pas directement utilisé par l'utilisateur, c'est le profil UML équivalent qui est manipulé. Cela permet d'utiliser les outils standard de modélisation graphique. Une transformation de modèles est employée pour passer du modèle conforme au profil UML au modèle conforme au méta-modèle. Nous ne détaillons ici que le méta-modèle, le profil étant pratiquement identique à l'exception de certains concepts

déjà disponibles dans UML. Nous allons présenter les cinq paquetages composant le méta-modèle permettant de modéliser respectivement les notions de composant, de factorisation, d'application, d'architecture matérielle, et d'association. Le lecteur intéressé pourra trouver des approfondissements sur les concepts présentés dans le rapport technique dédié à ce sujet [17] et auquel j'ai contribué, au même titre que la plupart des membres de l'équipe.

2.3.1 Le paquetage *Component*

L'une des bases du méta-modèle est le paquetage *Component*, il définit l'approche composant sous-jacente. Il permet la construction de l'application et de l'architecture sous une forme structurelle. Respectant les mêmes concepts qu'UML, il permet la mise en œuvre des mécanismes :

- **d'encapsulation**, pour rendre la définition d'un composant indépendante de l'environnement dans lequel le composant est utilisé. Une frontière claire est tracée entre l'intérieur du composant (son implémentation), et la façon dont il perçoit son environnement, ou la façon dont il est perçu par son environnement.
- **de composition et d'assemblage**, pour permettre de structurer hiérarchiquement la définition d'un composant. Un composant peut être composé d'autres composants offrant des fonctionnalités plus simples. L'assemblage de ces composants permet d'implémenter des comportements plus complexes.

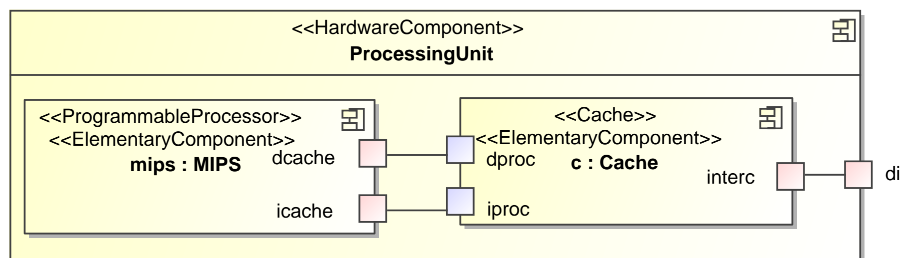


FIG. 2.12: Exemple de composant composé en utilisant le profil UML de Gaspard. Le composant *ProcessingUnit* est constitué de deux sous-composants, instanciés sous le nom de *c* et *mips*. Les ports et les connecteurs permettent d'indiquer l'organisation des instances pour répondre au service proposé par le composant.

Le concept de *Component* constitue l'élément de base réutilisable manipulé dans le contexte d'un modèle *Gaspard*. Un composant est défini par un ensemble d'interfaces exposées via des *Ports*, et par une structure constituée d'un assemblage d'*Instances* via des *Connectors*. Une *Instance* d'un composant B contenue dans un composant A indique que le composant A fait appel au composant B pour son implémentation. Les composants qui ne contiennent aucune instance sont des *ElementaryComponents*. Cela introduit la notion de « boîte noire », niveau de précision à partir duquel le modèle ne contient pas d'information, où l'implémentation n'a pas d'incidence sur le comportement global du SoC. La figure 2.12 illustre ces concepts à l'aide d'un exemple d'une unité de calcul *ProcessingUnit* définie par deux composants élémentaires : un processeur (*MIPS*) et un cache (*Cache*). Au passage notons que les différents exemples qui illustrent cette section sont tous dérivés du modèle complet de SoC présenté dans le chapitre 6.

De plus, mentionnons que les ports peuvent être *In* ou *Out*[2] , indiquant que l'interface ne fait respectivement que recevoir ou émettre des données. Les connexions doivent tenir compte de ces différences : les connecteurs de délégation (entre un composant et une instance) ne peuvent connecter que des ports ayant la même orientation, et au contraire, les connecteurs d'assemblage (entre deux instances) ne peuvent connecter que des ports dont l'orientation est opposée.

2.3.2 Le paquetage *Factorization*

Le paquetage *Factorization* est l'un des plus grands attraits du méta-modèle *Gaspard*. C'est grâce aux concepts qu'il introduit que le méta-modèle est particulièrement bien adapté aux systèmes massivement parallèles réguliers : il permet de représenter sous forme compacte un ensemble de composants identiques et répétés. Il met à disposition du développeur de MPSoC les concepts définis initialement dans Array-OL mais en les généralisant afin de pouvoir non seulement les utiliser sur l'application, mais également sur l'architecture et l'association.

2.3.2.1 La notion de *Shape*

Le concept de *Shape* peut être spécifié sur une instance ou sur un port. Il permet d'indiquer la multiplicité de l'élément, qui représente alors une collection de n éléments. Cette collection est organisée sous la forme d'un tableau multidimensionnel. Une *Shape* est un vecteur d'entiers strictement positifs qui représente le nombre d'éléments dans chaque dimension du tableau. Ainsi, il est par exemple possible de représenter 100 instances de composants sous la forme d'un tableau de 5×20 instances en spécifiant une *Shape* de $(5, 20)$. De plus, dans une *Shape* une dimension peut être spécifiée comme étant illimitée à l'aide du caractère ~. Dans ce cas, le tableau représenté contient alors une collection d'éléments ayant la forme des autres dimensions répétée à l'infini. Si cela n'a pas de sens dans le contexte de l'architecture matérielle, c'est utile dans le contexte du comportement applicatif pour représenter un traitement répété infiniment sur le temps.

2.3.2.2 La notion de *Tiler*

Le second aspect du paquetage *Factorization* concerne la manière d'ajouter des informations topologiques entre les entités répétées. En d'autres termes, une fois capable d'exprimer sous forme compacte les éléments, il faut pouvoir exprimer des liens entre ces éléments également sous une forme compacte.

L'idée générale de la notion de *Tiler* est d'identifier des sous-tableaux de points, nommés *motifs*, à l'intérieur d'un tableau (défini par une *Shape*), puis de définir la correspondance entre les points d'un motif et ceux du tableau. Ces motifs sont des tableaux multidimensionnels, par conséquent décrits à l'aide de la notion de *Shape* similairement aux autres tableaux. Nous appelons *tuile* l'ensemble des points du tableau qui correspondent au motif. Ces tuiles sont constituées de points régulièrement espacés, et les tuiles elles-mêmes sont régulièrement espacées. La description de l'espacement régulier des points d'une tuile est appelée *ajustage* (*fitting* en anglais), et la description de l'espacement régulier des tuiles dans le tableau est nommée *pavage* (*paving* en anglais). La description complète du positionnement des tuiles sur

[2]Lorsque les ports n'ont pas de direction particulière spécifiée, ils sont alors capables traiter les données dans les deux directions.

un tableau nécessite la description des dimensions du motif, l'ajustage, le pavage, une *origine* (qui est un point dans le tableau) et un *espace de répétition*. L'espace de répétition spécifie le nombre de tuiles. Il est également défini à l'aide d'une *Shape*.

Pour un motif donné, l'identification d'une tuile s'effectue à partir d'un *élément de référence* (**ref**) situé dans le tableau. La matrice d'ajustage est utilisée pour calculer la position des autres éléments de la tuile par rapport à **ref**. Cette matrice a autant de lignes que le tableau a de dimensions, et autant de colonnes que le motif a de dimensions. Les coordonnées des éléments de la tuile (e_i) sont construites comme la somme des coordonnées de **ref** et la multiplication de la matrice d'ajustage par les coordonnées i de l'élément dans le motif, comme suit :

$$\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \mathbf{ref} + F \cdot \mathbf{i} \mod \mathbf{s}_{\text{array}} \tag{2.1}$$

$S_{pattern}$ est la *Shape* du motif, S_{array} la *Shape* du tableau, et F la matrice d'ajustage.

Un exemple est présenté en figure 2.13, une tuile (à gauche) correspondant à un motif (à droite) de 3×2 éléments (spécifié par la *Shape* $S_{pattern}$) est positionnée sur un tableau de 6×4 éléments (spécifié par la *Shape* S_{array}). La tuile est construite d'après la matrice d'ajustage (F). Cette matrice est de taille 2×2 car le tableau a deux dimensions et le motif a également deux dimensions. Le premier vecteur de la matrice, $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$, indique que pour trouver l'élément suivant de la tuile lorsque l'on parcourt le motif le long de la première dimension, il faut à chaque fois se décaler de deux points horizontalement dans le tableau. Le second vecteur de la matrice d'ajustage, $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$, indique que pour trouver l'élément suivant de la tuile lorsque l'on parcourt le motif le long de la seconde dimension, il faut à chaque fois se décaler d'un point horizontalement et d'un point verticalement dans le tableau.

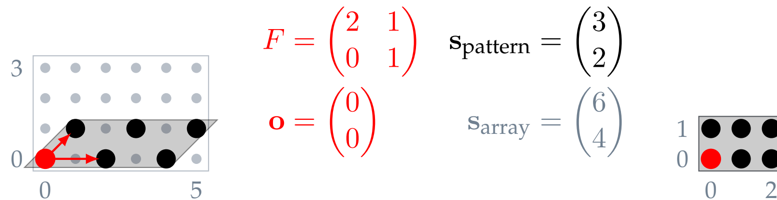


FIG. 2.13: À gauche, une tuile alignée sur l'abscisse et décalée de 1 le long de l'ordonnée du tableau. Au centre se trouvent la matrice d'ajustage (F), le vecteur d'origine (o), la *Shape* du motif ($S_{pattern}$), et la *Shape* du tableau (S_{array}). À droite, le motif correspondant à la tuile est dessiné.

Pour chaque répétition, le pavage permet de spécifier la position de l'élément de référence **ref** utilisé pour placer la tuile dans le tableau. Le placement de ces éléments de référence est fait de manière similaire à la construction de la tuile. La matrice de pavage a autant de lignes que le tableau a de dimensions, et autant de colonnes que l'espace de répétition a de dimensions. La position de l'élément de référence de la répétition de base est donnée par l'origine o. Les coordonnées $\mathbf{ref}_r$ de l'élément de référence pour une répétition r donnée sont calculées en faisant la somme de l'origine et la multiplication de la matrice de pavage par r, comme suit :

$$\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \mathbf{ref_r} = \mathbf{o} + P \cdot \mathbf{r} \mod \mathbf{s}_{\text{array}} \tag{2.2}$$

P est la matrice de pavage et $S_{repetition}$ la *Shape* de l'espace de répétition.

Pour résumer, l'ajustage décrit les coordonnées des points de la tuile dans le tableau relativement à un point de référence, le pavage décrit l'ensemble des points de référence des tuiles relativement à l'origine. Le processus de parcours du tableau par les tuiles est décrit par un *Tiler* possédant trois attributs : `origin` (un vecteur d'entiers), `fitting` (une matrice d'entiers), et `paving` (une matrice d'entiers). Les points de la tuile d'indice **r** dans l'espace de répétition sont énumérés comme suit. À partir d'un point donné d'indice **i** dans le motif, et pour un tableau ayant une taille s_{array}, les coordonnées du point correspondant dans le tableau sont :

$$\mathrm{o} + (\mathrm{P} \quad \mathrm{F}) \times \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \bmod \mathrm{s_{array}} \,. \tag{2.3}$$

Cette formule garantie que :
- les points de la tuile sont régulièrement espacés car ils sont construits depuis le point de référence de la tuile par combinaison linéaire des vecteurs colonne de la matrice d'ajustage ;
- les points de référence des tuiles sont régulièrement espacés car ils sont construits depuis l'origine par combinaison linéaire des vecteurs colonne de la matrice de pavage ;
- tous les points de la tuile sont des points du tableau : les calculs sont effectués modulo la taille du tableau.

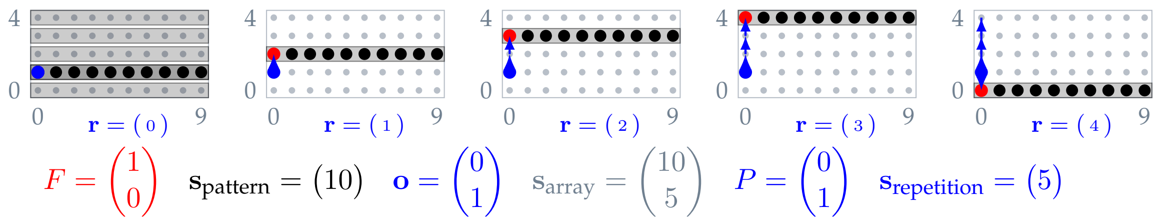


FIG. 2.14: Chacune des tuiles correspondant à une répétition de l'espace de répétition, indexée par **r**. Les vecteurs de pavage sont dessinés depuis l'origine **o** et indiquent comment les coordonnées de l'élément référence de la tuile courante sont calculées. Dans cet exemple, le tableau est ainsi parcouru ligne par ligne en commençant depuis la deuxième ligne.

L'utilisation du pavage pour parcourir le tableau complet en utilisant une tuile est montrée figure 2.14. Notons d'abord qu'à chaque répétition, la forme de la tuile reste identique : c'est une ligne de dix points, comme spécifié par la matrice d'ajustage F et la *Shape* du motif $S_{pattern}$. Entre chaque répétition, seul le point de référence varie. Le déplacement de ce point est spécifié par la matrice de pavage P. Cette matrice est de dimension 2×1 car le tableau a deux dimensions et l'espace de répétition en a une. L'unique vecteur qu'elle contient indique qu'à chaque répétition il faut décaler le point de référence d'un point verticalement. Le vecteur origine o fait démarrer ce point de référence à la deuxième ligne. En raison des propriétés modulo du *Tiler*, la dernière répétition ne lit non pas en dehors du tableau mais la première ligne de celui-ci.

Dans le méta-modèle *Gaspard*, lorsqu'un *Tiler* est utilisé sous la forme de connecteur, il est toujours utilisé comme connecteur de délégation, entre le port d'un composant et le port d'une instance contenue dans ce composant. Il est utilisable quelque soit l'orientation des ports. La taille du tableau du *Tiler* correspond à la *Shape* du port du composant. La taille du

motif correspond à la *Shape* du port de l'instance. L'espace de répétition est spécifié par la *Shape* de l'instance.

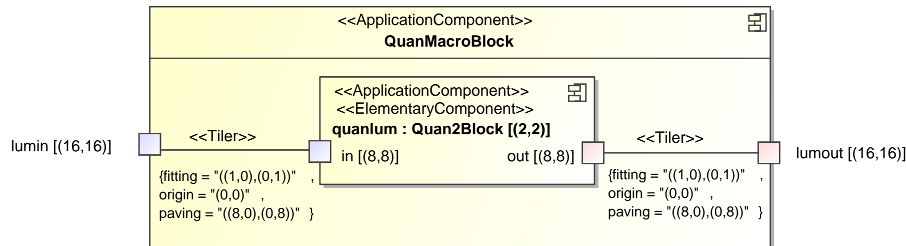


FIG. 2.15: Exemple d'usage de *Tilers*. Deux *Tilers* relient les éléments des ports de *quanlum* répété 2×2 fois à chaque unique élément des ports de *QuanMacroBlock*.

Un exemple d'usage de *Tiler* est donné figure 2.15. Les deux *Tilers* sont identiques, ils permettent chacun de lier les éléments d'un port des 4 instances de *Quan2Block* aux éléments d'un port du composant *QuanMacroBlock*. L'espace de répétition des *Tilers* est défini par la *Shape* de l'instance de *Quan2Block*, $(2, 2)$. Pour un *Tiler* donné, la *Shape* du port de l'instance, $(8, 8)$, est la forme du motif : le motif est un carré de 64 éléments. La *Shape* du port du composant *QuanMacroBlock*, $(16, 16)$, correspond à la forme du tableau. Les attributs `origin`, `fitting`, `paving` correspondent respectivement à l'origine, à l'ajustage, et au pavage. L'ajustage permet d'identifier la forme de la tuile dans le tableau. Étant une matrice identité, la tuile a la même forme que le motif : lors d'un décalage le long de la première dimension dans le motif, le même décalage est appliqué dans le tableau, de même pour la seconde dimension. Le point d'indice $[0, 0]$ du motif correspond donc au point de référence. La position du point de référence est calculé à partir de la matrice de pavage qui vaut $\begin{pmatrix} 8 & 0 \\ 0 & 8 \end{pmatrix}$ et de l'indice de la répétition courante. Le point de référence débute au point $[0, 0]$ dans le tableau (car l'origine est nulle) et vaut pour les 3 autres répétitions $[8, 0]$, $[0, 8]$, et $[8, 8]$. Ainsi, le tableau d'éléments contenu dans le port de *QuanMacroBlock* est découpé en quatre carrés adjacents qui sont répartis entre les quatre répétitions de *Quan2Block*.

2.3.2.3 La notion de *Reshape*

Le concept de *Reshape* a été proposé afin d'aider la tâche du concepteur lorsque des éléments de deux tableaux aux formes différentes sont liés un à un (par exemple dans le contexte de l'application, cela peut arriver en cas de « corner-turns »). Sémantiquement, un *Reshape* est équivalent à deux *Tilers* mis bout à bout : le premier *Tiler* rassemble dans un motif intermédiaire les éléments par motifs depuis le tableau d'entrée et le second *Tiler* répartit les éléments du motif intermédiaire sous une nouvelle forme dans le tableau de sortie. Ainsi, en plus des deux *Tilers*, le *Reshape* définit une *patternShape* qui correspond à la forme du motif intermédiaire, et un *repetitionSpace* qui correspond à l'espace de répétition parcouru pour remplir le tableau de sortie.

2.3.2.4 Les notions d'*InterRepetition* et de *DefaultLink*

Enfin, nous ne mentionnons que brièvement deux concepts non nécessaires à la compréhension du reste de ce mémoire : *InterRepetition* et *DefaultLink*. Le premier permet la connexion d'éléments faisant partie du même tableau, ce qui correspond à des dépendances de données inter-itération dans le contexte de répétition de tâches, ou à la communication entre différents éléments d'une grille de composants matériels dans le contexte de l'architecture. Avec ce type de connexion, il est souvent nécessaire de pouvoir exprimer les connexions irrégulières qui peuvent se trouver sur les bords du tableau d'éléments, le concept de *DefaultLink* est utilisé pour ce faire.

2.3.3 Le paquetage *Application*

Le paquetage *Application* n'introduit qu'une classe importante : *ApplicationComponent*. Elle étend la notion de *Component* pour indiquer explicitement qu'un composant représente une partie de l'application et qu'il faut donc y associer une sémantique propre. Cette sémantique permet de représenter une application selon un langage de flot de données[3] en s'appuyant sur les concepts des paquetages *Component* et *Factorization*. Le langage utilisé est largement inspiré d'Array-OL, dont le modèle de calcul a été défini de manière complète [21]. Nous allons présenter ici les grandes lignes de la sémantique, mais avant entendons-nous bien : lorsque l'on parle ici d'*application*, on ne parle pas spécialement de logiciel mais de la notion plus générale correspondant au comportement du système en fonction des entrées. En particulier, il est tout a fait envisageable d'implémenter tout ou partie de l'application sous la forme de matériel.

2.3.3.1 Les principes

Le but premier du langage défini par le paquetage *Application* est de permettre l'expression d'applications de traitement de signal intensif multidimensionnel. Ces applications travaillent sur des tableaux multidimensionnels. La complexité de ces applications ne provient pas des fonctions élémentaires qui les constituent, mais de la combinaison de ces fonctions et de la manière par laquelle les fonctions accèdent aux tableaux de données intermédiaires. En effet, typiquement ces fonctions élémentaires sont des fonctions connues (transformée de Fourrier, produit scalaire...) ou disponibles dans des bibliothèques de fonctions. Afin d'obtenir les meilleures performances de l'application, il est nécessaire d'utiliser au maximum le parallélisme potentiel de la tâche, qui pourra alors être corrélé avec celui du matériel.

Ainsi, le langage suit un ensemble de principes de base :
- tout le parallélisme potentiel est exprimé ;
- les dépendances de données sont explicites ;
- les tableaux sont à assignation unique, c'est-à-dire que chaque élément n'est écrit qu'une seule fois (mais peut être lu plusieurs fois) ;
- les dimensions temporelles[4] et spatiales sont traitées identiquement dans les tableaux.

L'application est définie par un graphe de dépendances où les nœuds correspondent aux tâches et les arêtes correspondent aux dépendances. Ce graphe est hiérarchique : tout nœud

[3]Plus exactement, la sémantique correspond à un langage fonctionnel du premier ordre manipulant des tableaux multidimensionnels dont il existe une projection directe vers un langage de flot de données.

[4]L'utilisateur peut souhaiter représenter le temps à l'aide de plusieurs dimensions, par exemple heures et minutes.

peut contenir lui même un graphe de dépendance. Formellement, une application est un ensemble de *tâches* connectées via des *ports*. Dans le méta-modèle *Gaspard* les tâches sont représentées par des *instances* de *ApplicationComponents* et (sans surprise) les ports par des *Ports*. Notons bien que dans un modèle, un composant ne correspond pas à une tâche, mais seulement à sa structure interne.

Les tâches peuvent être assimilées à des fonctions mathématiques qui prennent en entrée les données lues par les ports d'entrée et retournent les données écrites sur les ports de sortie. Il existe trois types de tâches :

– **une tâche élémentaire**, représentée par une instance d'*ElementaryComponent*, est atomique, elle correspond à une fonction boîte noire.
– **une tâche composée**, représentée par l'instance d'un *Component* contenant plusieurs instances, correspond à un graphe de dépendance.
– **une tâche répétée**, représentée par une instance de *Component* sur laquelle une *Shape* est spécifiée. Tous les connecteurs partant de ses ports doivent être des *Tilers*.

Les données échangées entre les tâches sont des tableaux. Ces tableaux sont multidimensionnels et peuvent avoir une dimension infinie. Leur taille est donc représentée par une *Shape*. Chaque port est caractérisé par la *Shape* et le type d'élément du tableau de données auquel il permet d'accéder. Le langage est à assignation unique, dans la pratique cela signifie que les éléments des tableaux ne sont pas des *variables* mais des *valeurs*. L'écriture d'une autre valeur correspond forcément à l'écriture à un indice différent. Le temps est donc représenté par une (ou plusieurs) dimension des tableaux de données. Par exemple, pour représenter une vidéo, un tableau tridimensionnel est nécessaire : deux dimensions pour l'image et une dimension pour le temps.

2.3.3.2 Parallélisme de tâches

Le parallélisme de tâches est représenté par les tâches composées. Chaque description interne de composant correspond à un graphe acyclique orienté. Chaque nœud du graphe (une instance) correspond à une tâche et chaque arête (un connecteur d'assemblage orienté du port de sortie vers le port d'entrée) correspond à une dépendance reliant deux ports conformes (même taille et même type). Il n'y a pas de contrainte imposée entre les tailles et les types des ports d'entrée d'une tâche et ses ports de sortie : une tâche pourrait lire deux tableaux bidimensionnels et produire un tableau tridimensionnel.

Par exemple, la figure 2.16 présente l'intérieur d'une tâche composée. Les connecteurs d'assemblage, qui sont orientés depuis les ports de sortie (en rouge, dont les noms se terminent par *out*) vers les ports d'entrée (en bleu, dont les noms se terminent par *in*), forcent une exécution de l'instance *softdct* avant toutes les autres et de *coding* après toutes les autres. Les deux tâches *quantmb* et *quant2mb* peuvent être exécutées en parallèle.

Chaque exécution d'une tâche lit chacun de ses tableaux d'entrée et écrit dans chacun de ses tableaux de sorties. Il n'est pas possible de lire plus d'un tableau par port d'entrée pour produire un tableau en sortie : ce n'est pas un graphe de flot de donnée, mais bien un graphe de dépendance. Ainsi, il est possible d'ordonnancer les tâches à l'aide de ce graphe. Tout le parallélisme de tâches potentiel est exprimé : si deux tâches ont chacune toutes leurs dépendances complétées (tous leurs tableaux d'entrée sont disponibles), elles peuvent être exécutées simultanément. Cependant il n'est pas possible de connaître le parallélisme de données puisqu'au niveau du graphe de tâches aucune information n'est indiquée concernant l'ordre d'accès aux données.

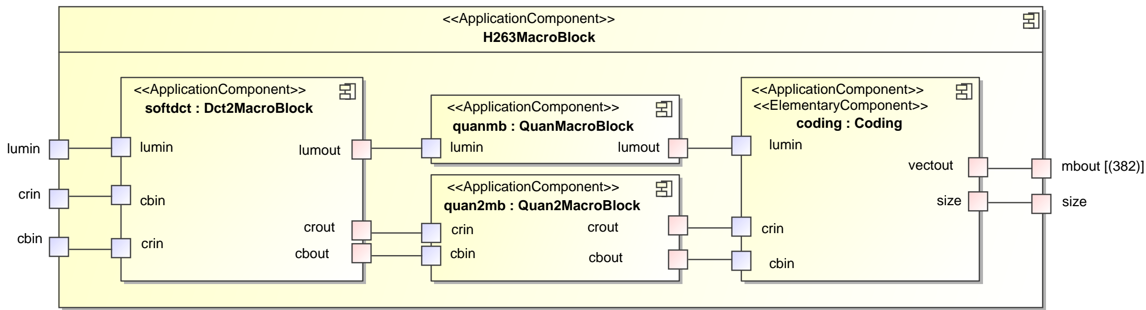


FIG. 2.16: Exemple de tâche composée représentée à l'aide du profil UML *Gaspard*. Elle est constituée de quatre sous-tâches. *coding* a deux dépendances sur *quant2mb* et une sur *quantmb*. *quant2mb* et *quantmb* ont chacune des dépendances sur *softdct*.

2.3.3.3 Parallélisme de données

Le parallélisme de données est spécifié autour d'une tâche répétée. L'hypothèse de base est que chaque répétition de tâche répétée *est indépendante*. C'est-à-dire que les répétitions peuvent être exécutées dans n'importe quel ordre, y compris en parallèle[5]. Par ailleurs, autour d'une tâche répétée, tous les connecteurs doivent être des *Tilers*[6]. Ces *Tilers* découpent les tableaux représentés par les ports du composant supérieur en motifs, représentés par les ports de la tâche. La tâche répétée n'accède donc qu'à des sous-tableaux des tableaux du composant. Notons également qu'à partir des spécifications du *Tiler* il est possible pour chaque tableau de vérifier formellement que l'ensemble des répétitions d'une tâche n'écrivent qu'une seule fois chaque élément.

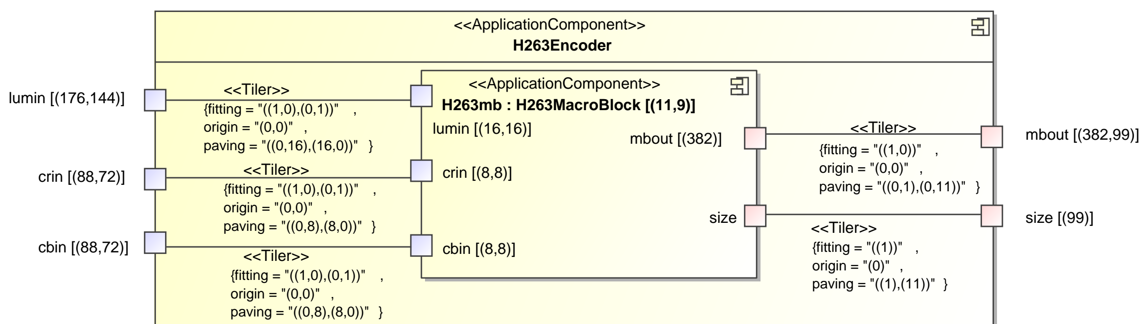


FIG. 2.17: Exemple de tâche répétée. Le composant *H263Encoder* contient une tâche répétée. Les 11×9 répétitions de la sous-tâche *H263mb* sont potentiellement exécutables simultanément. Chaque répétition travaille sur une partie différente des tableaux d'entrée et de sortie, définie par les *Tilers*.

[5]C'est la raison pour laquelle nous utilisons le terme de *répétition* et non pas d'*itération* qui est connoté d'une sémantique de séquentialité.

[6]Dans la pratique ils peuvent être également des connecteurs simples, dans ce cas ils sont considérés comme des *Tilers* qui ne lisent qu'un seul motif, de la même taille que le tableau.

La figure 2.17 illustre par un exemple la construction d'une tâche répétée. Le composant *H263Encoder* contient une tâche : la sous-tâche *H263mb* est répétée 11×9 et cinq *Tilers* spécifient comment sont lus ou écrits les motifs depuis les tableaux de données de la tâche. Rappelons que tous les *Tilers* ont le même espace de répétition, qui correspond à la répétition de la tâche, ici il est donc de 11×9.

Ainsi, cette sémantique du paquetage *Application* permet de représenter les dépendances entre les tâches et les répétitions de tâches. À partir de ces informations il est possible de déterminer pour n'importe quel élément des tableaux de sortie, les éléments des tableaux d'entrée dont ils dépendent. À partir de ces dépendances, on peut déduire un ordre d'exécution des tâches.

2.3.4 Le paquetage *HardwareArchitecture*

Le paquetage *HardwareArchitecture* permet au concepteur de définir la composition matérielle du SoC. La représentation est faite à un haut niveau d'abstraction en représentant à l'aide des notions du paquetage *Component* la structure et la hiérarchie du matériel. Les ports et les connecteurs représentent les liaisons directes entre les composants. Chaque connecteur est équivalent à une connexion de transaction et donc correspond à un ensemble de fils. Il existe quatre principaux types de composants matériels : *Processor* (les processeurs), *Memory* (les mémoires), *Communication* (les réseaux d'interconnexion), et *IO* (les périphériques d'entrée/sortie). Chacun de ces types est décomposé en un ensemble de types plus précis. Les types disponibles permettent de représenter globalement l'architecture SoC avec suffisamment de détails pour pouvoir associer l'application. Le fonctionnement exact de l'architecture est contenu dans les boîtes noires que sont les composants matériels élémentaires, et dans le cas où une partie de l'application est placée sur un accélérateur matériel, par les composants de l'application. Un exemple de modèle d'architecture matérielle est présenté figure 2.18. Le composant *HardwArchit* correspond à la globalité de la partie matérielle du SoC. Il est composé de quatre instances de composants.

Les mécanismes de factorisation peuvent être appliqués aux composants matériels. Ils permettent de représenter une topologie régulière. Par exemple, une grille de processeur peut être spécifiée rapidement et de manière compacte (voir l'exemple figure 3.2 p. 52). De même, un réseau d'interconnexion oméga [31] peut être facilement modélisé (comme démontré dans [85]).

2.3.5 Le paquetage *Association*

Le dernier paquetage du méta-modèle, *Association*, permet à l'utilisateur d'établir un lien entre une application et une architecture matérielle utilisée comme support d'exécution. L'allocation consiste à définir un placement des différents éléments de l'application sur les éléments de l'architecture. *TaskAllocation* permet de spécifier l'allocation d'une tâche sur une ressource de calcul (un processeur, un accélérateur matériel). C'est l'occasion de bien rappeler que l'application modélisée correspond à la fonctionnalité du système et n'est donc pas restreinte au logiciel : si une tâche est allouée sur un accélérateur matériel alors la fonctionnalité à laquelle elle correspond sera implémentée sous forme matérielle. *DataAllocation* est utilisée pour indiquer l'allocation d'un tableau de données sur une mémoire. Enfin, une *CommunicationAllocation* spécifie comment synchroniser deux mémoires. Elle n'est utile que lorsque des tâches partagent des données mais ont les tableaux de données alloués sur

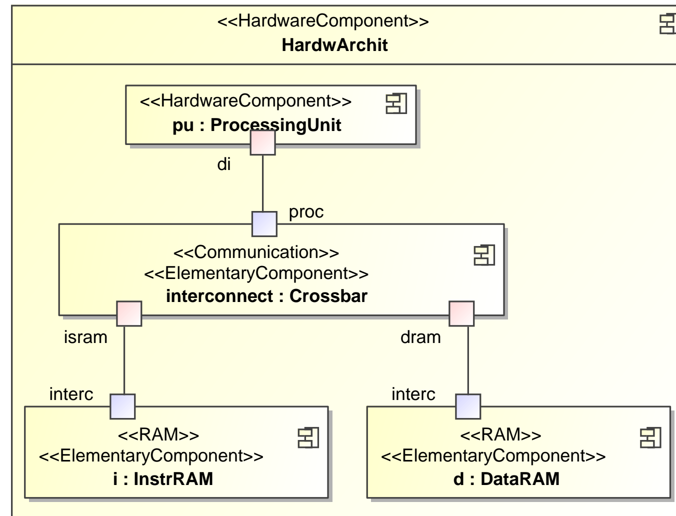


FIG. 2.18: Exemple de composant matériel. Le composant *HardwArchit* est composé de deux mémoires *d* et *i*, d'un composant processeur *pu*, et d'un réseau d'interconnexion les reliant.

différentes mémoires. La copie des données est alors effectuée selon la route indiquée par *CommunicationAllocation*. Notons que l'allocation représente le placement *spatial* de l'application sur l'architecture. Le placement temporel (ordonnancement des tâches, réutilisation de la mémoire pour différentes données. . .) n'est pas précisé, c'est au compilateur que reviennent ces décisions. La figure 2.19 présente un exemple simple d'association où une tâche (et donc toutes ses sous-tâches) est placée sur une unité de calcul alors que les tableaux de données sont placés sur une mémoire.

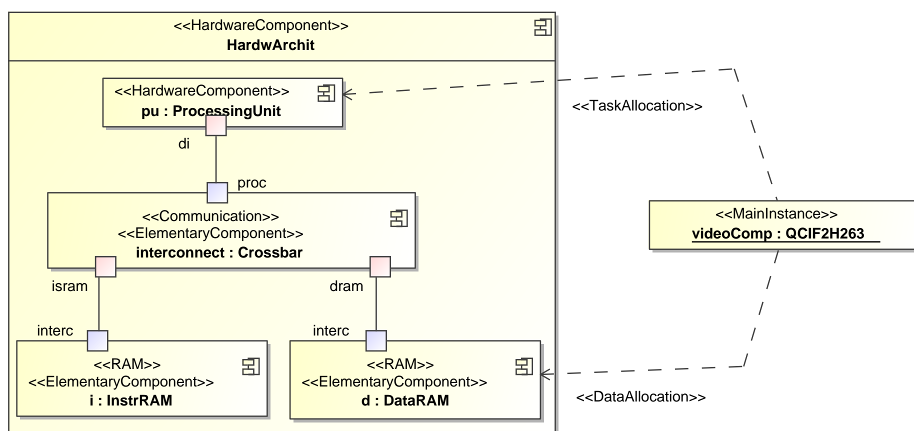


FIG. 2.19: Allocation de l'instance d'application *videoComp* sur l'architecture *HardwArchit*. Les tâches contenues dans cette application seront toutes exécutées sur l'unité de calcul *pu* tandis que toutes les données seront placées sur la mémoire vive *d*.

De nouveau, les mécanismes de factorisation jouent un rôle important ici : via la notion de *Distribution*, similaire à un *Reshape*, il est possible de distribuer une tâche répétée sur plusieurs processeurs ou d'indiquer la répartition d'un tableau de données sur un groupe de mémoires. Nous verrons plus en détail ce mécanisme dans le chapitre suivant, où nous pointerons certaines limitations du méta-modèle et proposerons des solutions.

2.3.6 Synthèse

Dans cette section nous avons présenté l'environnement Gaspard qui a pour objectif de permettre le développement de MPSoC dédiés au traitement de signal intensif en exploitant l'IDM. Au début de cette thèse, les transformations de modèles pour obtenir une simulation ou la réalisation complète n'étaient pas disponibles, seul le méta-modèle permettant de représenter un SoC avait déjà été défini. Il est constitué de cinq paquetages. Un paquetage introduit les notions de composant, un deuxième permet de représenter sous forme compacte un système répétitif régulier, y compris les liens qu'il existe entre les composants répétés. Les trois autres paquetages permettent de représenter l'application, l'architecture matérielle et le placement de l'application sur l'architecture. Ainsi, le SoC peut être développé en suivant l'organisation de co-conception usuelle.

Les travaux présentés dans cette première partie du mémoire se basent sur ce point de départ qu'est le méta-modèle *Gaspard* et vont consister à étendre l'environnement pour être capable de générer une co-simulation logiciel-matériel.

Chapitre 3

Un méta-modèle pour l'expression exécutable de MPSoC

3.1 Allocation répétitive et hiérarchique **49**
3.1.1 Allocation répétitive 50
3.1.2 Association hiérarchique 56
3.2 Le méta-modèle de déploiement **64**
3.2.1 Aperçu du paquetage *Deployment* 65
3.2.2 Résumé des concepts 67
3.2.3 Notion d'*Implementor* 68
3.2.4 Distinction *Software*/*Hardware* 73
3.2.5 Notion de *PortImplementation* 76
3.2.6 Notion d'*ImplementedByConnector* 78
3.2.7 Notions de *Characterizable* et *Specializable* 79
3.2.8 GaspardLib, une bibliothèque de composants pour la modélisation de SoC 83
3.2.9 Synthèse et conclusion : le déploiement d'IP 86
3.3 Conclusion **88**

Le méta-modèle de l'environnement Gaspard présenté précédemment permet de modéliser un SoC multiprocesseur. Cependant le méta-modèle n'est pas suffisant pour pouvoir générer un code *complet* fonctionnel pour obtenir une co-simulation, comme nous le présenterons dans le chapitre 5, ou bien pour obtenir les fichiers nécessaires à la création de la puce finale. Certaines informations nécessaires à la génération de code ne peuvent pas être exprimées avec le méta-modèle.

Dans ce chapitre, nous proposons une évolution du méta-modèle Gaspard afin, premièrement, de compléter la sémantique autour de l'allocation et de la hiérarchie et, deuxièmement, de permettre de faire le lien entre une tâche élémentaire et le code source d'une fonction implémentant cette tâche.

3.1 Allocation répétitive et hiérarchique

Vis-à-vis des performances d'un système parallèle, il est nécessaire d'obtenir un bon équilibrage de la charge entre les processeurs. En général, il faut pour cela distribuer équi-

tablement les répétitions de l'application sur les processeurs. Cependant, en fonction de la distribution, la localité des données peut très largement varier. En cas de faible localité, les communications requises lors des calculs augmentent et peuvent réduire considérablement le gain de performances apporté par le parallélisme. La distribution optimale d'une tâche dépend de l'organisation des accès aux données, de la distribution des données, et de la distribution des autres tâches. Il est donc nécessaire d'avoir des concepts capables d'exprimer avec précision les distributions.

Le mécanisme d'allocation, présenté dans la section 2.3 permet d'associer une partie de l'application à une partie de l'architecture matérielle. Cependant, tel quel, aucune sémantique dans le méta-modèle ne permet de représenter une hiérarchie de composants applicatifs placée sur une hiérarchie de composants matériels. Comme nous allons le voir, cela est particulièrement limitant lorsque les composants sont répétés. Avant d'entrer en détail dans la notion de hiérarchie, intéressons-nous d'abord à l'usage de répétition au niveau de l'allocation.

3.1.1 Allocation répétitive

L'un des points forts du méta-modèle Gaspard est de pouvoir représenter de manière compacte les systèmes possédant une structure répétitive régulière. Comme décrit dans la section 2.3, en premier lieu, le concept de « shape » permet de spécifier le nombre de répétitions d'un composant. Pour construire les liens entre composants répétés, le concept de « tiler » permet de partager un tableau d'éléments parmi un ensemble de composants répétés ; le concept de « reshape » repose sur le concept précédent et permet de lier directement des composants entre eux.

3.1.1.1 Distribution

L'usage du concept de « reshape » conjointement à celui de l'« allocation » s'appelle une « distribution ». Une distribution indique le placement d'un composant applicatif répété sur un ensemble de composants matériels répétés. C'est un concept essentiel pour tirer partie du parallélisme présent dans les modèles d'application et d'architecture représentés sous forme compacte. Elle permet de paralléliser une tâche répétée sur un groupe de processeurs identiques ou de distribuer un tableau de données sur un ensemble de bancs mémoire. Une description formelle de la distribution ainsi que des exemples détaillés ont été présentés dans [3].

La distribution de tâches (« TaskAllocation » et « Distribution ») indique pour chaque répétition de la tâche quel processeur l'exécutera. La distribution de données (« DataAllocation » et « Distribution ») indique quels éléments d'un tableau sont contenus sur quelle répétition d'un composant mémoire. Cette distribution ne permet pas d'indiquer l'ordre de stockage des données (par ligne, par colonne. . .).

Les répétitions de composants logiciels et matériels peuvent être multidimensionnelles et il n'est pas requit qu'elles aient le même nombre de dimensions. Par exemple, il est tout à fait possible de placer une répétition de $32 \times 6 \times 9$ tâches sur 8 processeurs. Notons aussi qu'une répétition de composants logiciels peut être associée à *plusieurs* répétitions du composant matériel, la répétition du composant logiciel est répliquée. Cela peut être utile par exemple pour réduire les communications lorsque plusieurs processeurs ont besoin de la même donnée : les données sont dupliquées sur chacune des mémoires locales aux processeurs.

La notion de placement de données et de tâches sur une architecture parallèle est aussi présente dans le domaine du calcul haute performance. Entre autres, le langage HPF [52] (High Performance Fortran) propose des techniques pour exprimer la distribution. Il existe une directive de compilation HPF qui spécifie au compilateur la manière d'attribuer des données à des processeurs (et indirectement aux mémoires les plus proches de chacun d'eux). Cette directive associe à chaque donnée un *propriétaire*. Lors de calculs, la règle du « propriétaire calcule »[1] est appliquée : le processeur exécute tous les calculs qui produisent les données dont il est propriétaire.

Avec le méta-modèle Gaspard, la distribution n'utilise pas de règle particulière pour lier calculs et données. Le développeur est libre de spécifier comme il l'entend l'allocation des données et des tâches, indépendamment. Au niveau du modèle d'association, il n'y a pas de restriction entre le placement des données sur une mémoire et le placement des tâches qui utilisent ces données sur des processeurs. Cependant en fonction de la cible de transformation, certaines distributions peuvent être interdites. Par exemple, dans notre

[1] En anglais : *the owner-computes rule*

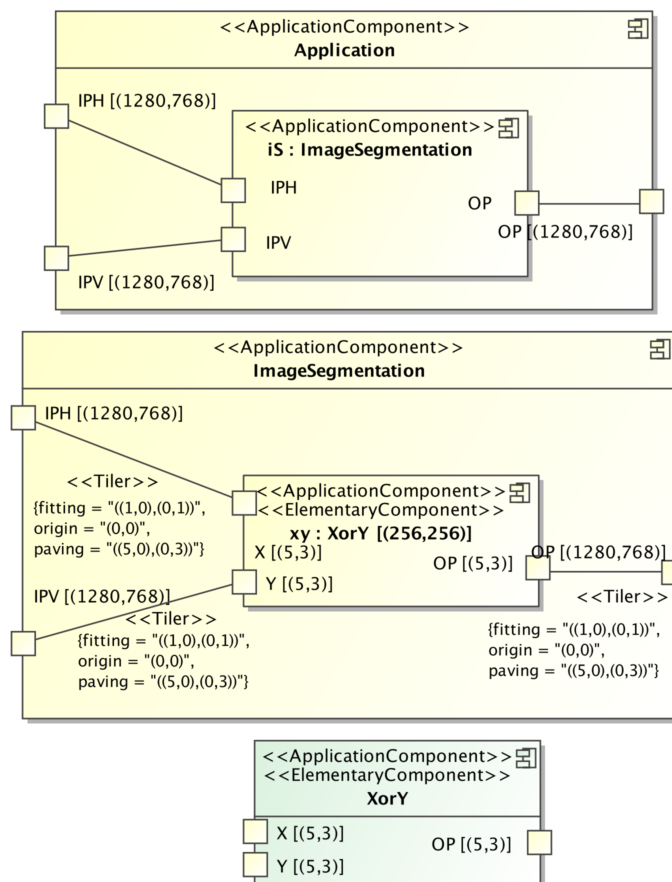


FIG. 3.1: L'application fictive du SoC qui va être distribuée. Une tâche lisant 2 images et générant une image et qui est composée de 256×256 répétitions de *XorY*.

chaîne de transformation vers une simulation en SystemC, il est obligatoire que toutes les données utilisées par une tâche se situent sur des mémoires accessibles directement par le processeur sur lequel est allouée cette tâche. C'est une restriction qui pourrait être levée en ajoutant automatiquement dans la transformation des tâches placées sur des processeurs intermédiaires ayant pour rôle uniquement la transmission de données. Bien sûr, pour des raisons de performance, il est habituel de placer les données dans les mémoires les plus proches des processeurs les utilisant, afin de favoriser au mieux la localité des données.

Comme nous l'avons vu précédemment, une « distribution » est composée de deux « tilers », d'une *patternShape* et d'un *repetitionSpace*. Chaque élément du tableau de sortie (l'architecture) peut être associé à zéro, un ou plusieurs éléments du tableau d'entrée (l'application). Cette syntaxe permet d'exprimer toutes les distributions classiques, qui sont entre autres disponibles dans HPF : bloc, cyclique, cyclique-k. Nous allons illustrer cette expressivité par des exemples pour chacune de ces distributions.

3.1.1.2 Exemples de distributions

Dans cette section nous allons associer de différentes manières une partie d'une application de traitement d'image à une architecture composée en grille. L'application est modélisée dans la figure 3.1. Elle contient une instance *iS* de *ImageSegmentation* qui lit deux images de 1280×768 pixels et génère une autre image de la même taille. Cette tâche est décomposée en une répétition de 256×256 instances de *XorY*. Chaque répétition travaille sur un petit motif de 5×3 pixels. C'est cette répétition que l'on va distribuer sur l'architecture matérielle.

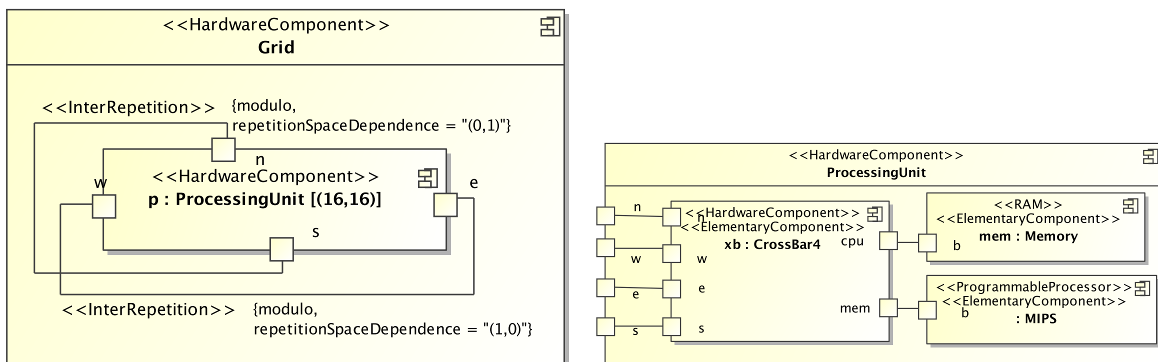


FIG. 3.2: L'architecture fictive du SoC sur laquelle l'application *ImageSegmentation* va être distribuée. C'est une grille de 16×16 unités de calcul.

L'architecture modélisée figure 3.2 est un SoC massivement multiprocesseur composé d'une grille de 16×16 unités de calcul (appelées *ProcessingUnit*). À l'aide de connexions « InterRepetition », chaque instance de l'unité de calcul est reliée à quatre autres instances via les ports *n*, *e*, *s*, et *w*, ce qui forme une grille torique. Une *ProcessingUnit* est composée d'un processeur, d'une mémoire et d'un crossbar pour assurer les communications. Par la suite, nous donnerons des équivalences des distributions en directive HPF. `P` est la variable représentant les processeurs (virtuels), déclarée en HPF ainsi :

```
!HPF$ PROCESSORS P(16,16)
```

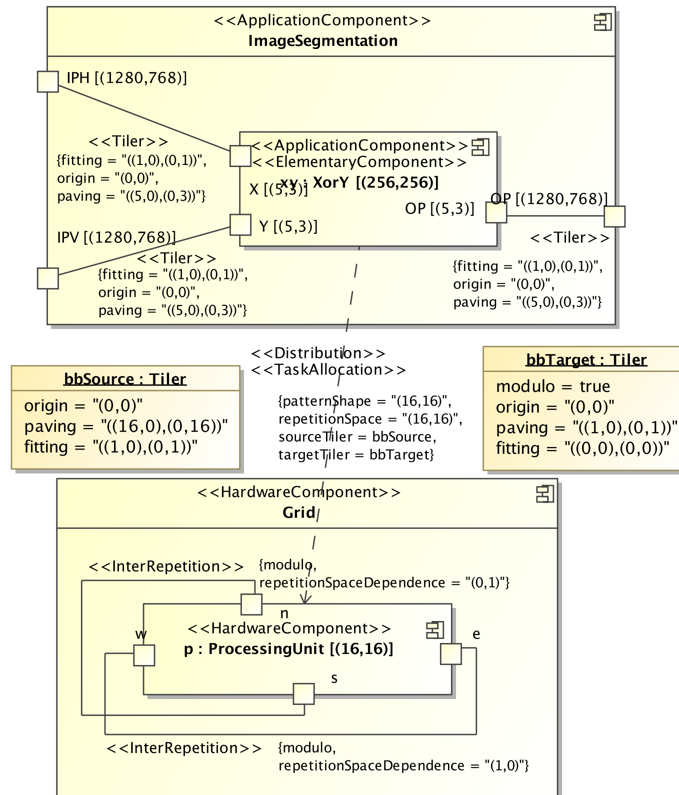


FIG. 3.3: Distribution par bloc de {16,16} des {256,256} répétitions de l'instance d'application *xy* sur les {16,16} répétitions de l'instance d'architecture *p*.

Distribution de tâches par bloc Tout d'abord, la figure 3.3 présente une distribution par bloc le long des deux dimensions de l'instance d'application *xy*. Chaque processeur va traiter un bloc de {16,16} répétitions de la tâche, c'est ce qu'indique la *patternShape*. La disposition des blocs est sans recouvrement entre eux, spécifié par le *paving* du *sourceTiler*. Le *repetitionSpace* spécifie que les blocs recouvrent l'ensemble du tableau de tâches. Ainsi, chaque répétition est exécutée une fois et une seule. Autrement dit, le processeur $[i, j]$ exécutera les 256 instances allant de $[i \times 16, j \times 16]$ à $[i \times 16 + 15, j \times 16 + 15]$. En considérant que HPF puisse directement spécifier le placement de tâche, après avoir déclaré la variable `xy` la directive équivalente à la distribution dans ce langage serait :

```
!HPF$ DISTRIBUTE xy(BLOCK,BLOCK) ONTO P
```

Distribution de tâches cyclique La figure 3.4 présente une distribution de la même application sur la même architecture, mais cette fois-ci avec une distribution [cyclique, cyclique]. Chaque bloc de 16×16 répétitions de *xy* est distribué sur la grille entière d'unités de calcul. La *patternShape* vide permet de spécifier que chaque motif ne contient qu'un seul élément (mathématiquement, un point a 0 dimension), ainsi pour chaque bloc de 16×16 il y a une seule répétition par unité de calcul. C'est grâce à la propriété de modulo des tilers (et en particulier du *targetTiler* ici) que toutes les tâches sont exécutées sur les processeurs : par

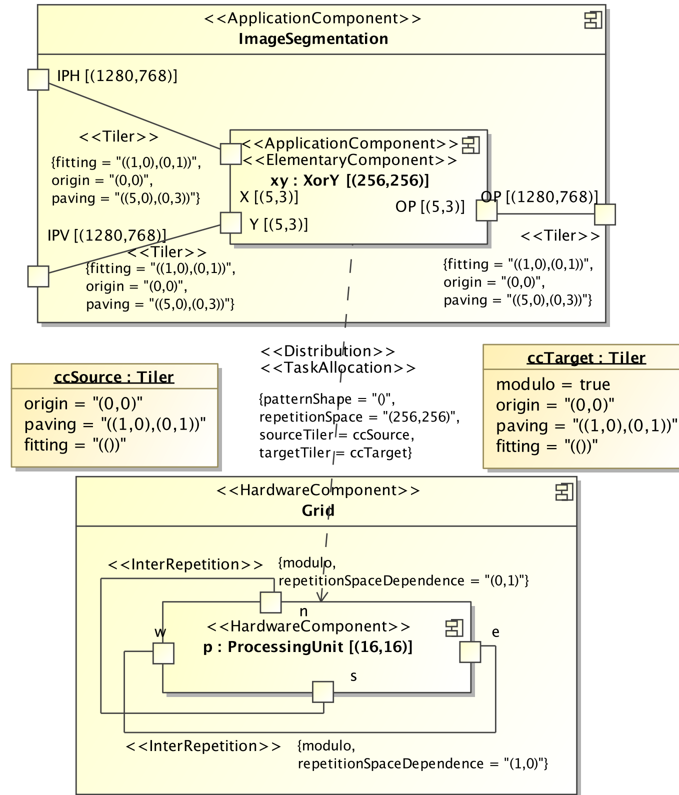


FIG. 3.4: Distribution cyclique des {256,256} répétitions de *xy* sur les {16,16} répétitions de *p*.

exemple la tâche $[16, 18]$ est placé sur le processeur $[0, 2]$. En d'autres termes, le processeur $[i, j]$ exécutera les 256 instances $[p \times 16 + i, q \times 16 + j]$ avec p et q des entiers compris entre 0 et 255. La spécification équivalente HPF serait :

```
!HPF$ DISTRIBUTE xy(CYCLIC,CYCLIC) ONTO P
```

Distribution de tâches cyclique-k En reprenant toujours la même application et la même architecture, la figure 3.5 présente cette fois une distribution cyclique de bloc 2 le long de l'abscisse et cyclique de bloc 8 le long de l'ordonnée. La taille des blocs est spécifiée par la *patternShape*, et le *paving* du *sourceTiler* indique qu'il n'y a pas de recouvrement entre blocs. Le *repetitionSpace* de {128,32} permet de parcourir l'ensemble des tâches, tout en parcourant de manière cyclique les processeurs. Ainsi, le processeur $[i, j]$ exécutera les 256 instances contenues dans les blocs allant de $[(p \times 16 + i) \times 2, (q \times 16 + j) \times 8]$ à $[(p \times 16 + i) \times 2 + 1, (q \times 16 + j) \times 8 + 7]$ avec p entier entre 0 et 127 et q entier entre 0 et 31. La spécification équivalente HPF serait :

```
!HPF$ DISTRIBUTE xy(CYCLIC(8),CYCLIC(2)) ONTO P
```

Distribution de données par bloc avec réplication Il est possible aussi de distribuer les données, en utilisant *DataAllocation*. Pour obtenir de bonnes performances de la part du SoC, il faut s'assurer de la localité des données, c'est-à-dire faire en sorte que les données soient

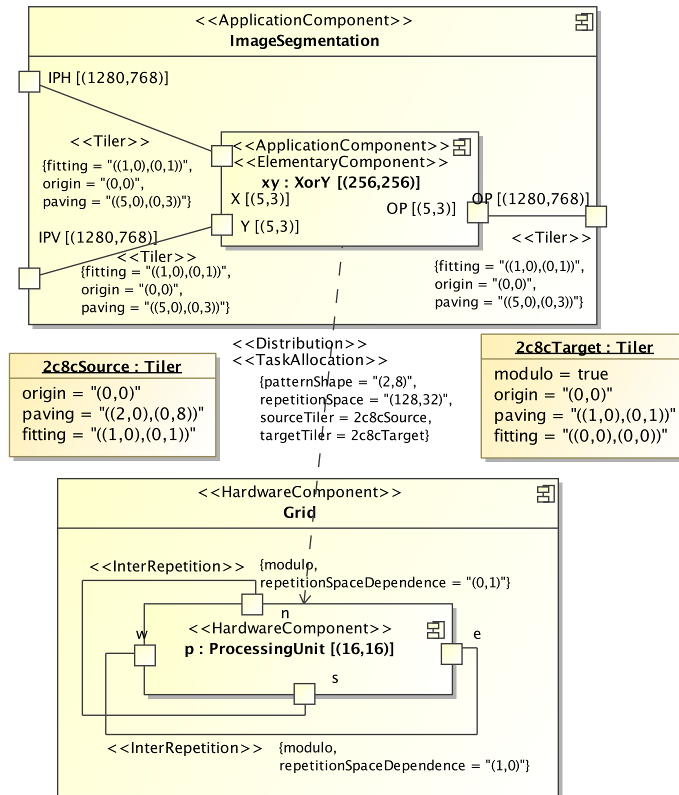


FIG. 3.5: Distribution cyclique-k des {256,256} répétitions de *xy* par blocs de {2,8} sur les {16,16} répétitions de *p*.

proches des tâches qui les utilisent. Afin d'illustrer la distribution avec réplication, nous allons considérer que la tâche *XorY* travaille sur des blocs de 5×3 pixels mais, pour les calculs, elle nécessite aussi un pixel supplémentaire autour des motifs d'entrée. C'est-à-dire que chaque instance utilise réellement des motifs de 7×5 pixels. Dans le domaine du traitement de signal, ce type de recouvrement de données est très courant. En considérant *XorY* distribuée par bloc, comme dans le premier exemple, nous proposons ici une distribution des données qui favorise la localité.

La figure 3.6 présente la distribution du tableau *IVH*, de taille d'une image complète, sur l'ensemble de la grille d'unités de calcul. Sur chaque unité de calcul est alloué un bloc de {82,50} pixels, chaque bloc étant décalé de {80,48} par rapport à ses voisins. Les bords des blocs sont ainsi répliqués sur plusieurs mémoires, ce qui permet que lors de l'exécution des tâches tous les accès mémoires se situent sur la mémoire locale. Notons que le tableau est considéré torique, par conséquent les premiers blocs partagent les bords avec les derniers blocs correspondants, c'est pour cela que l'*origin* pointe vers le dernier point du tableau.

La notion de *distribution* permet une grande expressivité de l'association de l'application sur l'architecture tout en gardant une syntaxe compacte. Néanmoins, telle quelle, la sémantique de l'association était restreinte pour les modèles complexes. Nous proposons ici une extension de cette sémantique pour permettre le placement à différents niveaux de hiérarchie.

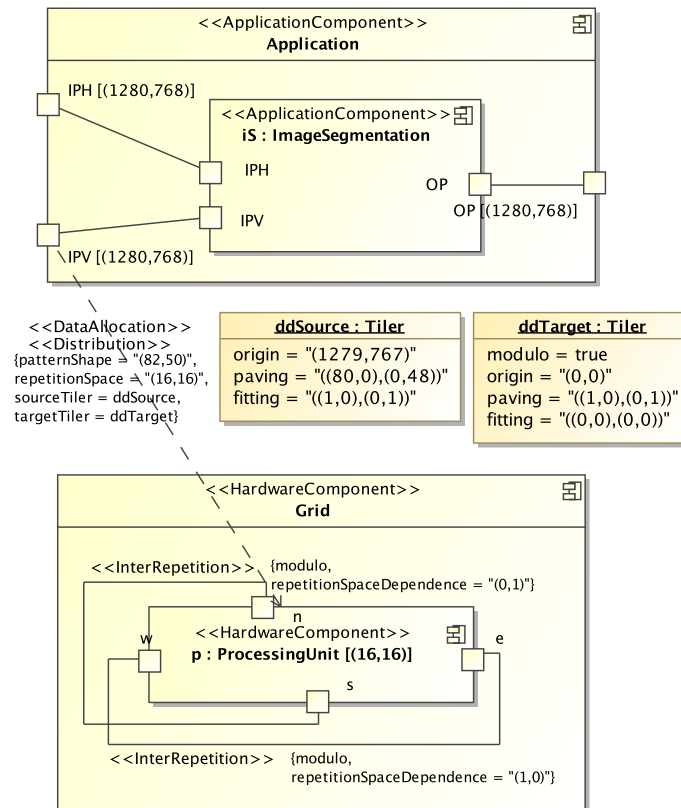


FIG. 3.6: Distribution des données du tableau *IVH* par bloc de {82,50} avec réplication des bords de 1 élément.

3.1.2 Association hiérarchique

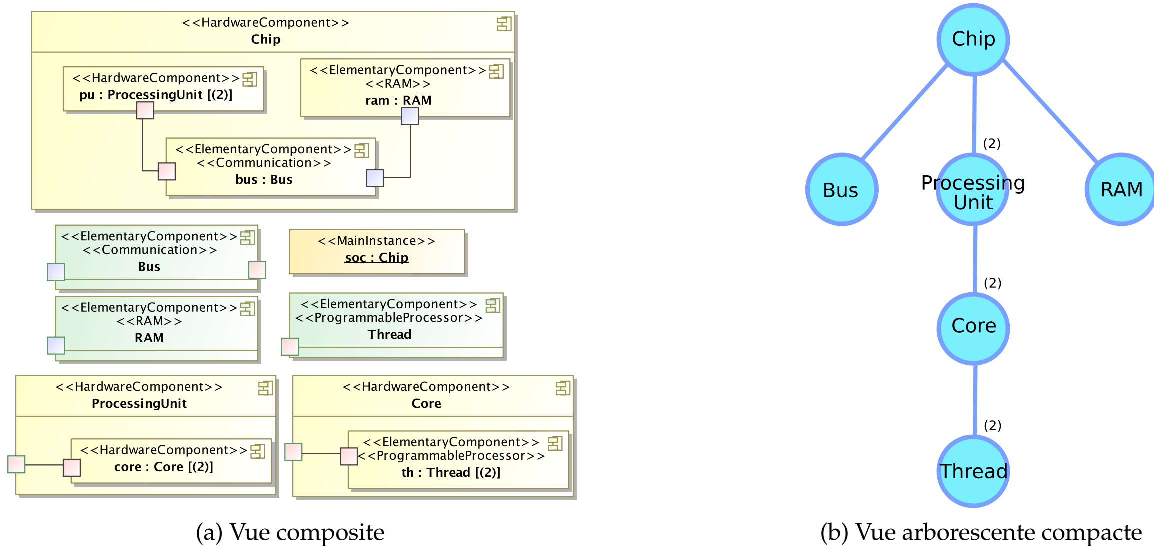
Alors que les modèles complexes nécessitent intensivement l'usage de la notion de composants, et donc implicitement de hiérarchie, et que cela peut être facilement spécifié dans les modèles d'application et d'architecture, aucune sémantique n'était précisée dans le méta-modèle Gaspard sur la gestion de la hiérarchie de composants lors de l'association. Afin de pallier ce manque et ainsi pouvoir implémenter une transformation de modèles vers une co-simulation logiciel/matériel complète, nous proposons ici une évolution du méta-modèle qui permet de représenter les placements à différents niveaux de hiérarchie.

3.1.2.1 Hiérarchie dans les modèles

Dans Gaspard, les modèles d'application et d'architecture ont une représentation structurelle. Cette structure définit principalement les connexions entre composants ainsi que les instances qui constituent un composant. Cette notion d'instances et de composants est la même que dans UML, avec une vue duale boîte noire/boîte blanche. Premièrement, le composant peut être vu simplement comme une boîte noire, il fournit une ou plusieurs fonctionnalités et respecte un certain nombre de propriétés (espace mémoire utilisé, temps d'exécution, emplacement physique sur la puce. . .). Il peut être également vu plus en détail

par sa structure interne, comme une boîte blanche : un ensemble structuré d'instances de composants (une instance représentant un usage d'un composant) qui fournit les fonctionnalités. Chacune des instances respecte les mêmes propriétés que le composant mais avec des contraintes plus fortes, de manière à ce que pour chaque propriété l'assemblage des instances soit équivalent au composant lui-même. Les modèles définissent une hiérarchie de composants, avec à la racine l'instance principale du modèle (stéréotypée « MainInstance »), chaque niveau inférieur étant défini par les instances contenues dans le composant.

Nous proposons une représentation arborescente et simplifiée du modèle pour représenter la hiérarchie sous forme plus compacte. Le modèle est représenté par un arbre, chaque nœud correspondant à un composant. La racine de l'arbre représente l'instance principale du modèle et le composant auquel elle correspond. Chaque branche d'un nœud indique une instance qui constitue le composant. Une branche peut avoir une *shape* qui indique le nombre de répétitions de l'instance. Cette vue a l'avantage de la compacité et représente très clairement toute la hiérarchie d'un modèle. Cependant par rapport à la vue composite UML, un grand nombre d'informations disparaissent, en particulier les connexions entre les composants et les tableaux de données utilisés en entrée et en sortie de chaque composant. Ces deux vues sont complémentaires.

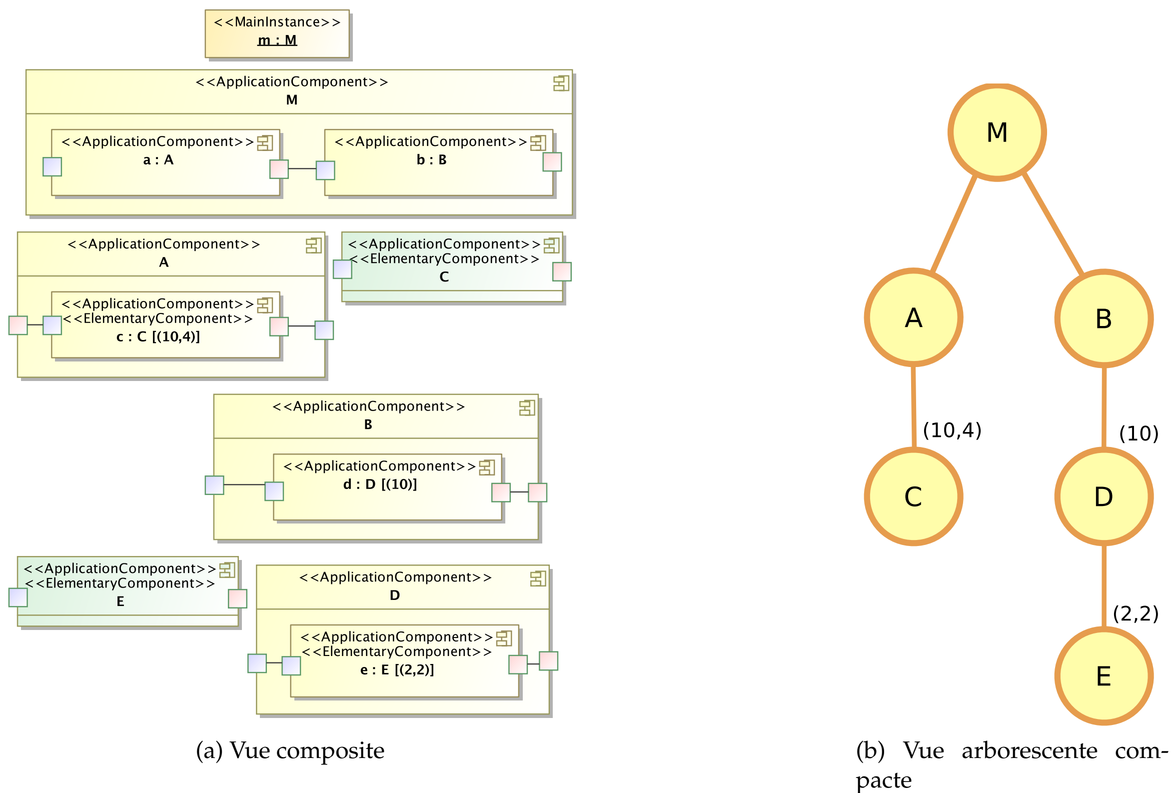


(a) Vue composite (b) Vue arborescente compacte

FIG. 3.7: Architecture possédant une hiérarchie dans la structure des processeurs.

Par exemple, la figure 3.7a présente un modèle d'architecture avec une mémoire, un bus et deux unités de calcul. Ces unités de calcul (*ProcessingUnit*) sont chacune composée de deux cœurs (*Core*), eux-même contenant deux fils d'exécution (*Thread*). Au total, il y a donc 8 fils d'exécution. Cette hiérarchie est représentée sous une forme arborescente compacte dans la figure 3.7b. On y distingue particulièrement bien les trois niveaux de hiérarchie des processeurs.

Un autre exemple, mais cette fois-ci avec un modèle d'application, est donné avec la figure 3.8a. L'application M est composée d'une instance de A qui passe le tableau résultat à une instance de B. Le composant A est composé de {10,4} répétitions de C. Quant au composant B il est constitué de 10 répétitions de D. Cette dernière tâche fait elle-même appel



(a) Vue composite

(b) Vue arborescente compacte

FIG. 3.8: Une application Gaspard.

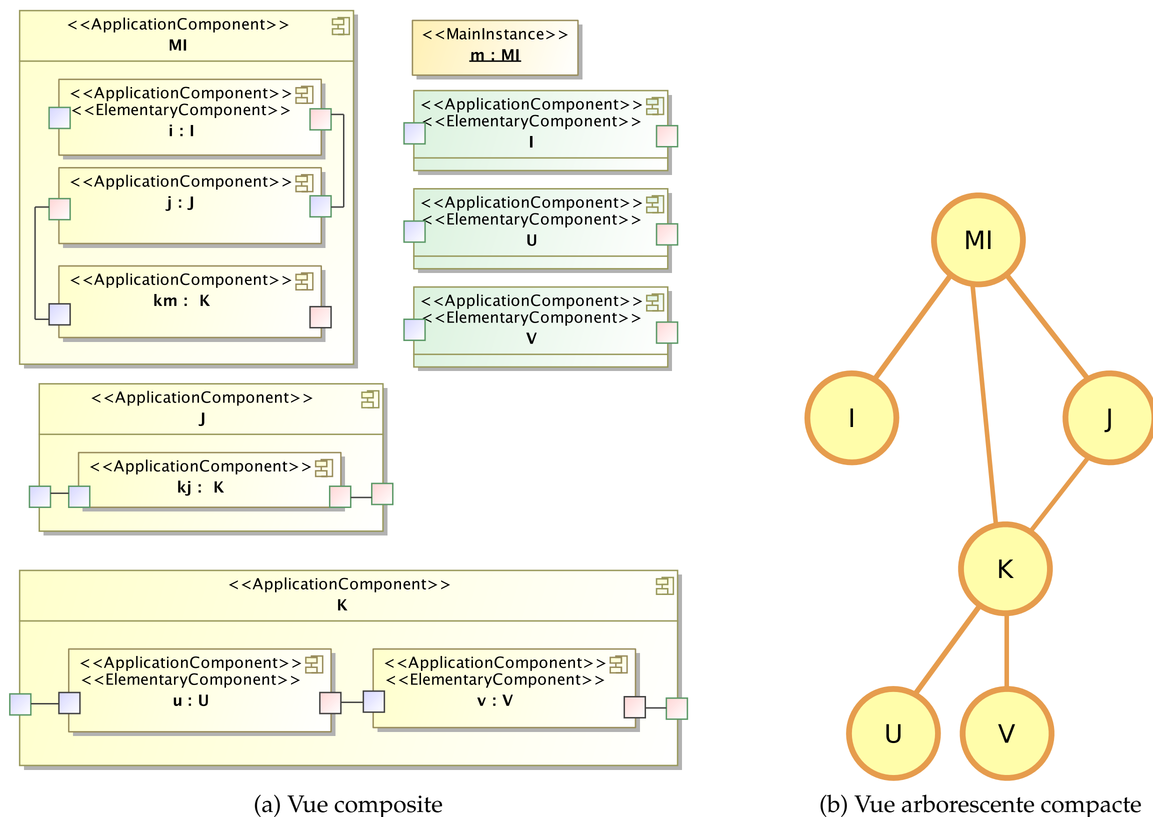
à {2, 2} répétitions de la tâche *E*. Notons que pour simplifier nous n'avons pas indiqué la taille des différents tableaux ni le type de connexion exact. La figure 3.8b représente cette application sous forme arborescente.

3.1.2.2 Nécessité de plus d'expressivité

Deux types de difficultés apparaissent lorsque l'on cherche à associer une application à une architecture alors qu'au moins l'une d'entre elles possède une hiérarchie de plus d'un niveau.

La répétition D'une part, des problèmes d'expressivité surviennent lorsque plusieurs niveaux d'arborescence sont répétés (ont une *shape*) à la fois dans l'application et dans l'architecture. Pour le placement de l'application sur l'architecture, il faut pouvoir indiquer pour chaque niveau de hiérarchie de chacun des modèles le lien entre les indices de répétitions d'une tâche et les indices de répétition d'un composant matériel. Par exemple, reprenons l'architecture et l'application présentées précédemment dans les figures 3.7b et 3.8b, comment répartir les 10×4 répétitions de la tâche *C* sur les deux *Threads* d'un *Core* d'une *ProcessingUnit* ? Comment placer les 2×2répétitions de la tâche *E* de manière à utiliser uniformément les 4 *Threads* de l'autre *ProcessingUnit* ? Ou bien comment spécifier qu'il faut les distribuer sur l'ensemble des *Threads* de l'architecture ?

Si aucune des instances de l'architecture ni de l'application n'était répétée, il serait facile de spécifier l'allocation, par exemple en plaçant chacune des tâches élémentaires (les feuilles de l'arborescence) sur un processeur élémentaire. De même, si toutes les répétitions étaient déroulées, c'est-à-dire chaque répétition d'instance représentée par un nœud différent, alors on pourrait aussi spécifier le placement. Bien sûr, dans la pratique, cette représentation non-compacte n'est pas envisageable : on ne souhaite pas que pour cette simple application l'utilisateur ait à dessiner 80 allocations ! C'est l'utilisation de la représentation compacte des répétitions qui amènent le besoin de spécifier le placement sur plusieurs niveaux de hiérarchie en même temps. Nous présenterons par la suite la sémantique que nous avons définie pour permettre cette expression.



(a) Vue composite

(b) Vue arborescente compacte

FIG. 3.9: Une application Gaspard où un composant est instancié plusieurs fois. Le composant *K* est utilisé par le composant *MI* et aussi par *J*.

Multiples instances d'un composant La seconde difficulté dans l'association concerne l'allocation sur une sous-hiérarchie d'un composant instancié *plusieurs fois* dans le modèle. Cette situation est illustrée à l'aide de l'application de la figure 3.9a, dont la vue arborescente est présentée dans la figure 3.9b. Le composant *K* est utilisé par le composant *MI* et aussi par *J*. Placer les différentes instances de ce composant sur différents processeurs d'une architecture telle que celle figure 3.10 ne pose pas de problème : on place une allocation par instance. Par contre il n'est pas possible de placer différemment chacune des instances *u* et *v* qui composent

K. Comment faire la distinction entre l'instance *km::u* et *kj::u* alors que l'origine de l'allocation est *u* ?

3.1.2.3 Notion de composant appliquée à l'association

Pour répondre à ces difficultés, nous proposons de compléter la sémantique de l'allocation en suivant le paradigme de composant : un composant peut contenir des sous-composants, ces sous-composants n'ont pas d'effet en dehors du composant, et toutes les propriétés qui s'appliquent au composant s'appliquent également aux sous-composants. Lorsqu'une tâche est allouée sur un composant matériel, alors implicitement toutes ses sous-tâches sont allouées soit sur ce composant ou bien sur un de ses sous-composants.

Remarquons que nous présentons l'usage de la hiérarchie sur l'allocation principalement dans le cadre du placement de tâches. En ce qui concerne le placement de données, le mécanisme peut tout à fait être appliqué de façon identique. Par exemple, la spécification d'une *DataAllocation* depuis le composant vers une mémoire indique que tous les tableaux de ce composant seront alloués sur cette mémoire, ainsi que tous les tableaux de ses sous-composants.

Nous définissons la sémantique de l'allocation à l'aide d'une règle principale et de deux règles secondaires qui permettent de traiter certains cas spéciaux.

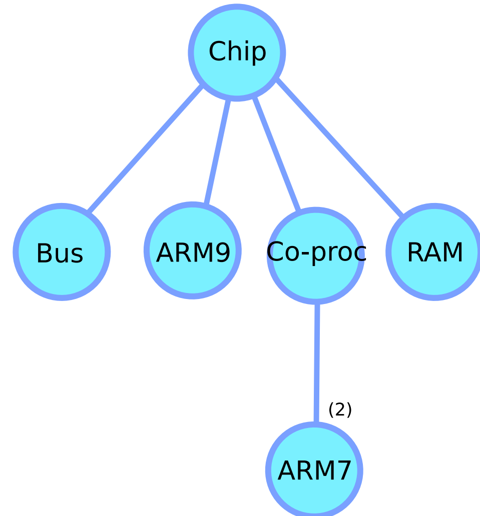


FIG. 3.10: Vue arborescente compacte d'une architecture à trois processeurs. Aux côtés d'un ARM9 se trouve un « co-processeur » composé de deux ARM7.

Règle principale de l'allocation Toute sous-tâche d'une tâche *A* allouée sur un composant matériel *M* peut être également allouée à condition que ce soit sur un sous-composant de *M*.

Règle du contexte De plus, une instance de tâche peut avoir plusieurs allocations. Ces allocations sont dépendantes du contexte. Seules les allocations qui sont valides selon la règle principale vis-à-vis des allocations des tâches parentes sont prises en compte. Cette sémantique permet de distinguer les différents placements possibles lorsqu'un composant logiciel est instancié plusieurs fois.

Pour un contexte donné, il peut encore y avoir plusieurs allocations. Si elles désignent une *succession* de composants matériels le long d'une arborescence (un composant, un de ses sous-composants, un sous-sous-composant. . .), cela signifie que cette tâche est placée sur l'instance matérielle la plus basse de la succession. Seule l'instance matérielle située *dans le contexte* de la succession de composants exécutera cette tâche. Cette interprétation permet de lever le doute du placement lorsqu'un composant matériel est instancié plusieurs fois.

Règle pour la répétition Si une tâche répétée (dont la *shape* n'est pas nulle) est allouée à l'aide d'une succession d'allocations, seule l'allocation qui pointe vers le composant matériel le plus bas peut être une distribution. Cette distribution indique donc le placement réel de la

tâche répétée, tandis que les autres allocations permettent de préciser le contexte. En effet, les distributions travaillent sur l'*ensemble des répétitions* de la tâche, cela n'aurait pas de sens si ces répétitions étaient déjà distribuées en des sous-ensembles. Notons que cela n'interdit donc pas qu'une tâche sans répétition puisse avoir plusieurs distributions le long d'une succession d'allocations. Dans ce cas, les distributions permettent de préciser pour quelles répétitions de composants matériels la tâche sera exécutée, il n'y a pas d'ambiguïté possible.

3.1.2.4 Exemples

Pour donner des exemples d'usage de cette sémantique, nous allons reprendre les applications et l'architecture que nous avons précédemment présentées. Dans ces exemples, l'allocation est représentée par une flèche rouge en pointillé reliant l'instance d'application placée à l'instance d'architecture. L'allocation simple possède un bout en forme de triangle, tandis que le bout de la distribution est en forme de losange. Par convention, dans les figures, l'architecture est à droite en bleu tandis que l'application est à gauche en jaune.

Pour placer les répétitions des différents niveaux de hiérarchie de l'application présentée en figure 3.8b, l'extension de l'association permet de distribuer chaque niveau de l'application sur un niveau différent de l'architecture. Dans la figure 3.11, les distributions de *A* et *B* indiquent sur quelle *ProcessingUnit* toutes leurs sous-tâches seront placées. Avec la tâche *A*, on exploite la notion de succession d'allocations pour spécifier aussi sur quel *Core* seront placées les sous-tâches. La tâche *C* est distribuée sur les deux *Threads* d'un *Core*. Le placement des tâches *D* et *E* montre la distribution à plusieurs niveaux. Les calculs seront distribués sur les *Threads* et les *Cores* en fonction respectivement des indices de répétitions de *E* et *D*.

Pour placer les instances de manières différentes dans l'application présentée en figure 3.9b, la notion de contexte dans l'association permet d'indiquer le placement que dans certaines conditions. L'allocation de toute l'application sur l'architecture est présentée dans la figure 3.12. Les tâches *I* et *J* sont respectivement placées sur *ARM9* et *Co-proc*. La tâche *MI::K* est placée sur *ARM9*, mais pas la tâche *MI::J::K* car dans le contexte de *J* le placement sur l'*ARM9* n'est pas possible. Similairement, les tâches *U* et *V* sont distribuées sur l'*ARM7* uniquement lorsqu'elles appartiennent à l'arborescence de *J*. Elles sont donc exécutées sur *ARM9* lorsqu'elles sont instanciées dans *MI::K*.

3.1.2.5 Complétude de l'association

On peut définir une règle qui assure de la propriété de complétude de l'association des tâches du modèle d'application sur le modèle d'architecture : chaque tâche élémentaire doit être associée à au moins un processeur élémentaire, soit directement par une allocation, soit indirectement par l'allocation d'une des tâches supérieure dans l'arborescence. Ainsi, si cette règle est respectée, cela signifie que pour chaque feuille de l'arborescence de l'application, qui correspond à du code de calcul, il y a au moins un processeur qui se chargera de son exécution. Les tâches supérieures, qui correspondent à des nids de boucles, sont toutes implicitement exécutées lors de l'exécution de chacune de leurs sous-tâches. Par conséquent, si une association respecte cette règle, il est garanti que l'ensemble de l'application sera exécuté.

La figure 3.13 présente une association avec plusieurs erreurs. La tâche *E* n'est pas directement placée et sa tâche supérieure *D* n'est pas placée sur des processeurs élémentaires, donc il n'est pas possible de déterminer quel processeur doit exécuter quelle partie de l'application :

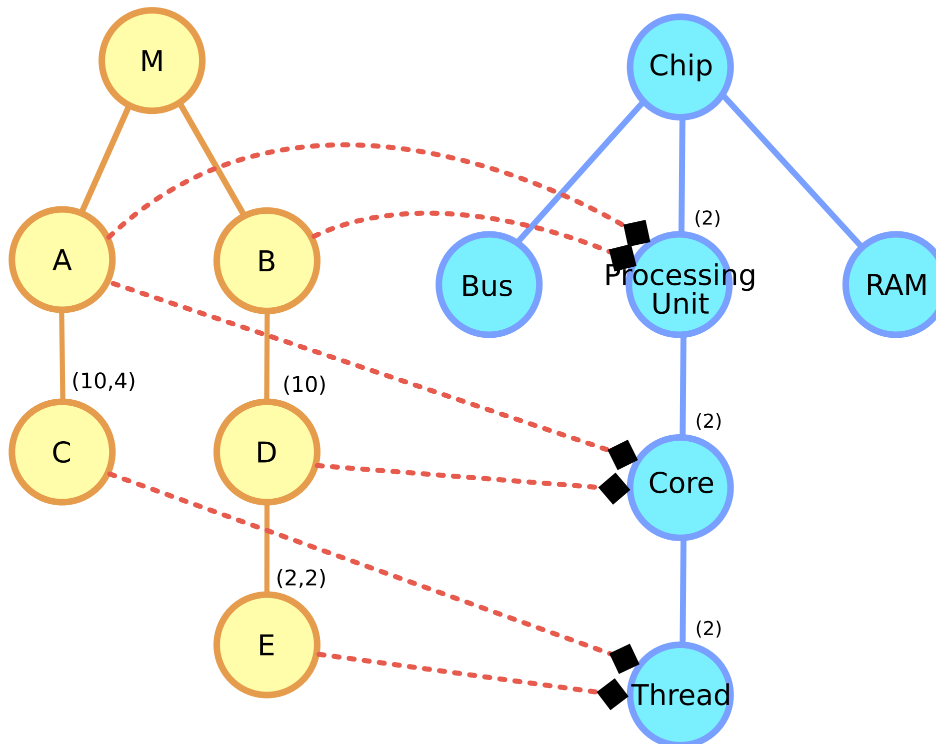


FIG. 3.11: L'application est entièrement distribuée sur l'architecture, en utilisant la hiérarchie des allocations.

l'association n'est pas complète. Par ailleurs, la tâche répétée *D* est distribuée sur plusieurs niveaux. Cette construction est interdite, car une fois le tableau des 10 tâches distribué sur les 2 *ProcessingUnits*, la taille du tableau de tâches à distribuer sur les 2 *Cores* n'est pas explicite (ni même forcément identique en fonction de la *ProcessingUnit*).

Enfin, notons que *C* est distribuée sur plusieurs processeurs élémentaires, donc cette partie est correcte. Étant donné qu'aucune allocation n'est définie pour les composants supérieurs, cette allocation indique que chaque *Core* de chaque *ProcessingUnit* exécutera les mêmes tâches. Chaque répétition de la tâche sera donc exécutée 4 fois, répliquée sur chacun des *Cores*.

3.1.2.6 Synthèse et conclusion sur les mécanismes d'allocation

Nous avons tout d'abord exploré la palette d'expressions que permet la notion de distribution dans Gaspard. Elle peut être aussi bien appliquée aux tâches qu'aux tableaux de données et autorise la représentation compacte d'un SoC, dans le prolongement des mécanismes de factorisation. Toutes les directives de placement de HPF, une extension spéciale de Fortran pour le calcul parallèle, peuvent être exprimées par la « Distribution ». Zéro, un, ou plusieurs composants logiciels peuvent être placés de manière régulière sur zéro, un, ou plusieurs composants matériels.

Nous avons vu que bien que très expressive au niveau du composant, cette notion a une expressivité limitée lors du placement de l'ensemble du SoC, en particulier vis-à-vis de la hiérarchie. Dans ce but, une extension de la sémantique a été proposée. En grande partie,

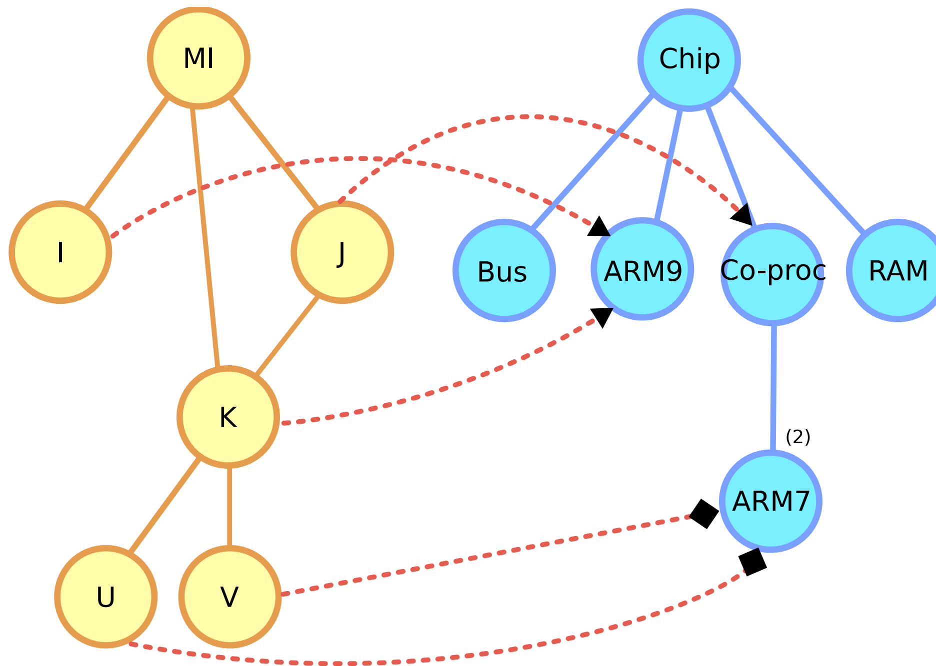


FIG. 3.12: L'application est entièrement distribuée sur l'architecture, en utilisant la hiérarchie des allocations et le fait que certaines allocations ne sont pas prise en compte en fonction du contexte.

cette extension apporte un sens particulier à des points de variation sémantique (c'est-à-dire à des configurations syntaxiques dont le sens était interprétable de différentes manières selon l'utilisateur).

Notons qu'il eut été possible d'utiliser une approche où l'allocation contient explicitement le chemin complet de la hiérarchie pour laquelle elle est valide (par exemple en utilisant un attribut supplémentaire *path*). Il avait aussi été envisagé que lors de la distribution, ce n'est pas uniquement la *shape* des deux composants impliqués qui soit prise en compte mais la concaténation de toutes les *shapes* des composants supérieurs. Ces solutions permettaient d'éviter certains cas pour lesquels la solution proposée nécessite de restructurer le modèle d'application ou d'architecture. Néanmoins, ces solutions impliquent qu'à un niveau donné il faille connaître tous les niveaux supérieurs, ce qui est complètement contraire à l'approche composant : on ne doit pas avoir à écrire un composant en fonction du contexte dans lequel il est exécuté, il doit pouvoir être autonome. En fait ce type d'approche n'est pas bien adapté aux systèmes complexes, car plus la complexité augmente, plus les informations à donner par l'utilisateur devront être complexes.

Après avoir vu les évolutions que nous avons apportées pour représenter complètement l'association d'un SoC, nous allons maintenant aborder une seconde partie qu'il manquait au méta-modèle Gaspard pour la génération automatique d'un SoC : comment faire les liens entre les composants élémentaires symbolisant des boîtes noires et le code existant qui implémente ces boîtes noires.

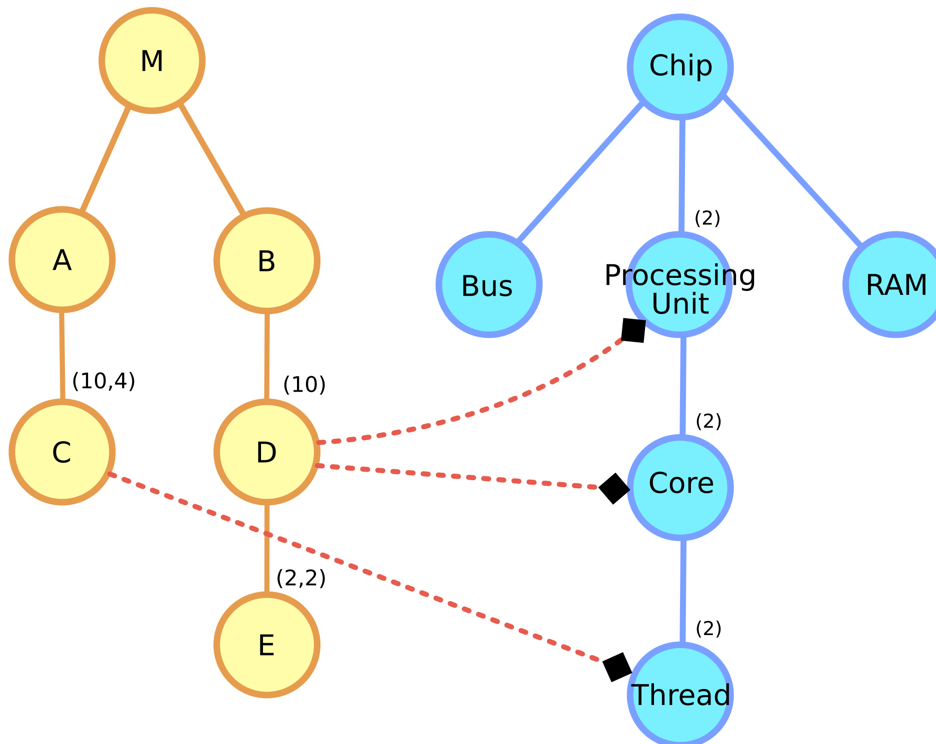


FIG. 3.13: Cette association contient plusieurs erreurs.

3.2 Le méta-modèle de déploiement

Le méta-modèle Gaspard tel que nous l'avons présenté permet de modéliser une application et une architecture à l'aide d'une représentation par composants. Tout le système est détaillé jusqu'au grain le plus fin : le composant élémentaire (« ElementaryComponent »). Ce type de composant est vu comme une boîte noire, c'est-à-dire que l'on ne sait pas comment les opérations se déroulent à l'intérieur, on ne connaît que les entrées et les sorties. En fait, pour la conception, connaître leur intérieur n'apporterait rien, ce sont simplement des briques de base à utiliser telles quelles. Les composants élémentaires correspondent à une implémentation déjà existante sous forme de code. Pour les composants matériels ce code peut être du VHDL, du SystemC, etc. et pour les composants logiciels on peut s'attendre à du code en assembleur, en C, etc. On nomme ces implémentations de briques de base *IP* (Intellectual Property). Ce terme est plus généralement employé dans le domaine du matériel, mais ici pour des raisons de clarté nous utiliserons ce terme également pour leurs équivalents logiciels plus souvent mentionnés sous le terme de *fonction de bibliothèque*.

Bien que pour la conception ces informations ne soient pas utiles, si l'on souhaite générer entièrement une exécution sur SoC à partir du modèle, il est important que le concepteur puisse associer avec précision dès le plus haut niveau de modélisation chaque composant élémentaire à sa concrétisation (l'IP). Ce sont des informations nécessaires pour que le modèle de MPSoC soit *exécutable*, c'est-à-dire qu'il soit possible de produire à partir du modèle non pas juste le squelette du code mais le code au complet directement compilable et exécutable. Pour répondre à ce besoin, nous proposons ici un ensemble de concepts pour exprimer les

informations liant la modélisation à haut niveau d'abstraction au code des IP. L'ensemble de ces concepts étend le méta-modèle Gaspard via l'introduction d'un nouveau paquetage nommé *Deployment*.

Comme le reste du méta-modèle Gaspard, ce paquetage est disponible à la fois sous forme de méta-modèle (utilisé par les transformations de modèles) et sous forme de profil UML (utilisé pour la conception à l'aide d'outils standards UML). Ces deux formes sont pratiquement identiques. La différence principale étant que sur certains points, le profil tire partie de concepts déjà existants dans UML. Par conséquent, nous présentons ici seulement le méta-modèle avec néanmoins, afin de bénéficier d'une représentation graphique, les exemples modélisés à l'aide du profil UML. Le lecteur intéressé par les détails du profil pourra se référer à l'annexe A.

Dans cette section, après avoir esquissé l'usage de ce nouveau paquetage dans sa globalité, nous détaillerons les différentes notions qui le composent. Ensuite, une bibliothèque de composants logiciels et matériels reposant exclusivement sur ces concepts sera présentée, permettant la validation du méta-modèle.

3.2.1 Aperçu du paquetage *Deployment*

Le but principal de ce paquetage est de donner les moyens au concepteur de lier chacune des briques de base que sont les composants élémentaires à un IP concrétisant la fonctionnalité requise. Plus exactement, il faut pouvoir fournir suffisamment d'informations pour que l'intégration du code de ces IP puisse être faite automatiquement, à la fois lors de la génération de code et lors de la compilation. C'est-à-dire qu'il faut permettre d'associer un IP à chaque composant, mais aussi connaître précisément le format des interfaces d'entrée et de sortie et la manière avec laquelle l'IP doit être appelé. Cependant, nous ne nous sommes pas limités à cette propriété lors de la création de ce paquetage : afin de faciliter la phase de conception nous avons tenu à ce qu'il réponde aussi à trois autres propriétés, que nous allons détailler.

Premièrement, afin de correspondre au mieux à la manière de travailler dans l'industrie et de factoriser l'écriture du code d'un IP, la description de l'IP doit autoriser dans le code de l'IP l'usage de paramètres qui ne sont spécifiés qu'au moment du déploiement d'un composant élémentaire sur l'IP. Un peu à l'identique de l'idée de *patron*, un code doit pouvoir être générique et spécialisé en fonction du besoin. Par exemple, le type de donnée sur lequel l'IP travaille doit pouvoir être un `int` ou un `float`, le code doit être indépendant de la taille des tableaux d'entrée et de sortie, le compromis entre précision du résultat et vitesse de calcul choisi, etc.

Deuxièmement, afin de convenir à la conception de SoC, il faut que le paquetage permette facilement la *réutilisation* de composants. Comme nous l'avons vu précédemment dans le chapitre introductif, la réutilisation d'IP logiciels et matériels est extrêmement importante pour la productivité. Lors du développement d'un nouveau SoC, la plupart des briques de base ne sont pas écrites spécialement mais proviennent soit de développements précédents en interne, soit de l'achat auprès d'autres entreprises qui se sont chargées de valider l'IP. Le paquetage *Deployment* doit permettre la création de bibliothèques d'IP. Nous avons prêté une grande attention lors de l'écriture de cette extension à ce que tout d'abord les IP puissent être décrits dans un modèle indépendant du modèle de SoC mais aussi à ce que l'usage d'un IP provenant d'une bibliothèque soit aussi simple et intuitif que possible. En particulier, les évolutions de la bibliothèque et du modèle de SoC doivent être indépendantes et la spécification de valeurs précises pour un composant de bibliothèque décrit par un patron ne nécessite pas

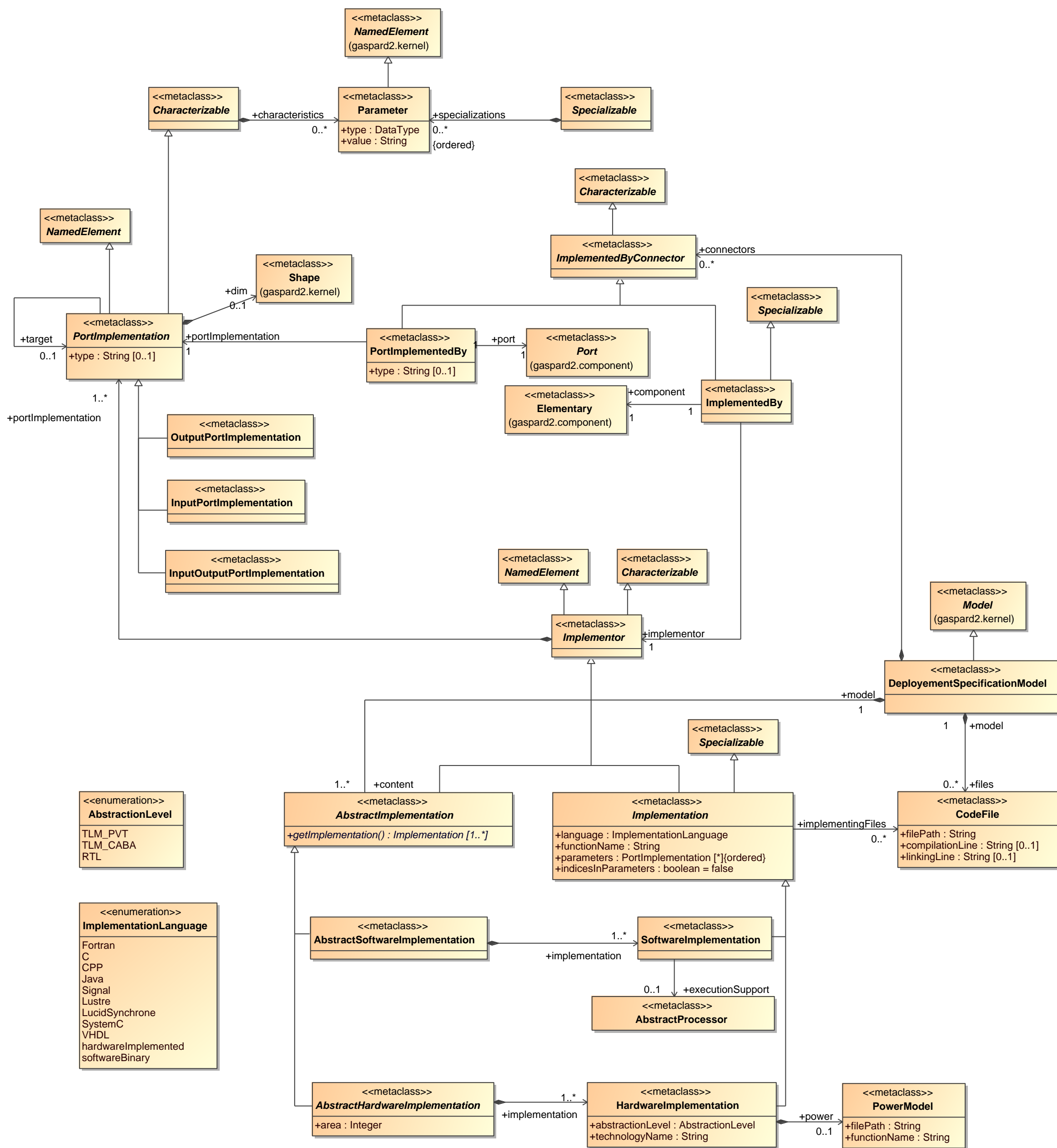


FIG. 3.14: Vue globale du paquetage *Deployment*. La racine est la classe *DeploymentSpecificationModel*. Chaque partie de cette figure est reprise par la suite dans des vues plus détaillées.

la modification de la bibliothèque. De plus, le déploiement d'un composant élémentaire sur un IP peut se faire par le simple ajout d'une dépendance (c'est-à-dire graphiquement une flèche).

Troisièmement, un composant élémentaire lié à un IP est une abstraction de la fonctionnalité de cet IP. Cependant cet IP est spécifique à une cible de compilation donnée. Par exemple, en ciblant une simulation SystemC de niveau TLM, il est préférable d'utiliser des composants

matériels liés à des IP SystemC TLM. Mais si le même modèle est transformé en ciblant une génération plus bas niveau en VHDL et qu'il existe une version VHDL de cet IP, il sera nettement préférable de choisir celle-ci. Plus important, l'IP peut aussi devoir changer en fonction du placement : par exemple si un composant élémentaire logiciel est placé sur un processeur programmable, alors l'IP sera implémenté en assembleur, en C, etc. par contre si le placement est effectué sur un accélérateur matériel, l'IP sera implémenté en SystemC, en VHDL, etc. Il n'est pas souhaitable que la partie *Deployment* d'un modèle de SoC nécessite une modification à chaque fois que l'association est modifiée ou que le concepteur souhaite générer le SoC vers une autre cible. Le paquetage doit donc proposer une abstraction pour que différents IP ayant la même fonctionnalité puissent être liés à un composant élémentaire et que celui adapté à l'association et à la cible courante soit sélectionné.

La figure 3.14 présente une vue globale du paquetage *Deployment* dans le méta-modèle Gaspard. Cette vue donne un premier aperçu des différentes classes et des liens entre-elles. Par la suite, nous allons revenir sur chacun des groupes de classes en fonction des concepts qu'elles portent. Remarquons que la racine du paquetage est la classe *DeploymentSpecificationModel*, toutes les classes du paquetage sont directement ou indirectement reliées par un lien de composition avec cette classe. Un certain nombre de classes ne font pas partie du paquetage ; elles sont définies dans d'autres paquetages de Gaspard, telles que *Model*, *Port*, *Shape*, etc. Ces classes sont les différents « points d'ancrage » des concepts du déploiement sur le reste du méta-modèle.

Il existe aussi une partie supplémentaire dans le paquetage qui concerne la description des propriétés non fonctionnelle des composants matériels. La classe *PowerModel* en fait aussi partie. Une vue globale est présentée en annexe dans la figure A.1. Nous ne la traiterons que brièvement ici, elle est décrite en détail dans la thèse de Rabie Ben Atitallah [16].

3.2.2 Résumé des concepts

La notion d'*Implementor* permet de décrire comment un IP peut être appelé depuis le code généré et comment le compiler. La description est faite à trois « niveaux » afin de donner de la souplesse lors de la modélisation et d'éviter au maximum d'avoir à spécifier plusieurs fois la même information. Le niveau le plus bas, exprimé à l'aide de la classe *CodeFile*, correspond à la description d'un fichier. Le niveau intermédiaire, exprimé à l'aide de la classe *Implementation*, correspond à la description d'un IP pour une cible particulière. Le niveau supérieur, exprimé à l'aide de la classe *AbstractImplementation*, correspond à une fonctionnalité indépendante de la cible. Ce dernier niveau permet de générer à partir du même modèle plusieurs cibles de compilation, comme nous l'avons mis en avant dans [1].

Chaque fonctionnalité peut correspondre soit à un IP d'application (dans le sens de comportement du système), soit à un IP matériel. Ainsi, il existe les quatre classes *AbstractSoftwareImplementation*, *SoftwareImplementation*, *AbstractHardwareImplementation*, et *HardwareImplementation*. Un composant applicatif ne peut être déployé que sur les deux premières, tandis qu'un composant matériel ne peut l'être que sur les deux dernières.

La notion de *PortImplementation* permet d'identifier pour chacun des ports d'un composant élémentaire la correspondance avec une interface de communication de l'IP et de définir explicitement le format de données utilisé.

Les connecteurs *ImplementedBy* et *PortImplementedBy* font la liaison entre les définitions d'IP et le modèle de SoC. Elles autorisent la construction de bibliothèques d'IP : les implémentations et les fichiers de code sont définis dans un modèle, depuis d'autres modèles les

composants élémentaires référencent ces implémentations sans les affecter.

Enfin notons que la plupart des concepts portent des attributs `characteristics` et `specializations`. Via une `characteristic`, le concepteur peut passer une information particulière à une *transformation*, cela permet d'éviter de surcharger le méta-modèle par des détails utiles seulement pour quelques cibles de compilation. Une `specialization` est transmise directement au code de l'IP. Lors de la compilation, le code peut utiliser ces informations pour obtenir une version *spécialisée* de l'IP, ainsi il est possible de créer des IP génériques.

Les sections suivantes décrivent en détail chaque concept composant le méta-modèle et chaque attribut qu'il contient. Cela a pour but de documenter entièrement le paquetage, ce qui pourra servir par la suite de référence à l'équipe. Le lecteur souhaitant obtenir uniquement une vue globale des travaux de thèse pourra passer à la section 3.2.8 présentant un exemple d'utilisation du méta-modèle.

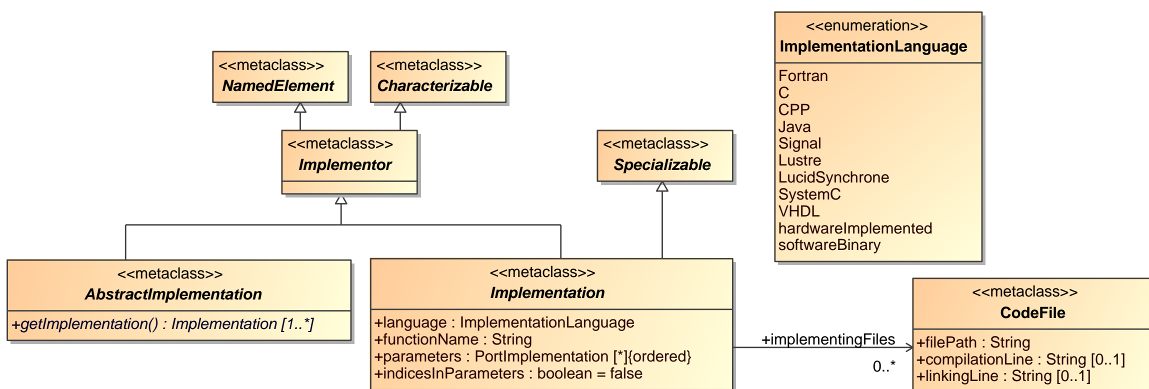


FIG. 3.15: Vue locale du paquetage *Deployment* autour de la notion d'*Implementor*

3.2.3 Notion d'*Implementor*

La notion d'*Implementor* est le cœur du paquetage *Deployment*. Une vue locale du méta-modèle autour de cette notion est présentée dans la figure 3.15. Cette notion permet de décrire comment un IP peut être appelé depuis le code généré et comment le compiler. La description se fait à l'aide de trois « niveaux » afin de donner de la souplesse lors de la modélisation et d'éviter au maximum d'avoir à spécifier plusieurs fois la même information. Le niveau le plus bas, exprimé à l'aide de la classe *CodeFile*, correspond à la description d'un fichier. Le niveau intermédiaire, exprimé à l'aide de la classe *Implementation*, correspond à la description d'un IP pour une cible particulière. Le niveau supérieur, exprimé à l'aide de la classe *AbstractImplementation*, correspond à une fonctionnalité indépendante de la cible. Nous commencerons par détailler l'usage et la sémantique de ces classes puis viendra un exemple d'utilisation.

3.2.3.1 *CodeFile*

Le but du paquetage *Deployment* est qu'une fois la génération de code du SoC achevée, ce code puisse être compilé avec tous les fichiers sources des IP utilisés. Il existe donc une notion dans le paquetage pour indiquer les informations concernant chacun des fichiers. La classe *CodeFile* permet de représenter un, et un seul, fichier. Il possède un attribut obligatoire `filePath` qui contient le chemin où trouver ce fichier (y compris le nom du fichier lui-même et son extension) dans le système de fichiers de l'ordinateur sur lequel se fait la génération de code. Afin de rendre le chemin indépendant de l'ordinateur sur lequel le modèle est lu, s'il est relatif le chemin débute du répertoire utilisateur.

La classe possède deux autres attributs, optionnels, `compilationLine` et `linkingLine`. Ils permettent d'indiquer, si nécessaire, des options particulières pour respectivement compiler et lier le fichier. Dans la plupart des cas en fonction uniquement du langage de programmation utilisé (défini dans *Implementation*) il est possible d'inférer la bonne manière de compiler et de lier. Cependant il peut arriver qu'un fichier nécessite des options spéciales pour obtenir l'IP voulu. Par exemple, lors de l'utilisation d'une bibliothèque mathématique optimisée en C++, les options de compilation pour obtenir le code le plus performant ont dû être explicitées afin de guider le compilateur. L'attribut `linkingLine`, surtout utilisé lors de la génération du logiciel, peut permettre entre autres de spécifier des bibliothèques standard sur lesquelles repose le code de l'IP.

3.2.3.2 *Implementation*

La classe *Implementation* permet de représenter directement un IP, qu'il soit logiciel ou matériel. Via la référence `implementingFiles`, les fichiers nécessaires à implémenter l'IP sont listés. Il n'y a pas de correspondance directe *Implementation*/*CodeFile*. Une *Implementation* peut nécessiter plusieurs *CodeFiles* (par exemple en C++ il est courant d'avoir pour chaque objet un ficher de code et un fichier de déclaration) tout comme plusieurs *Implementations* peuvent partager un même *CodeFile* (par exemple, un ensemble de fonctions similaires d'une bibliothèque qui sont regroupées dans un même fichier).

Les quatre attributs de la classe permettent de spécifier comment faire le lien entre le code généré du SoC et le code de l'IP. Cette information est l'un des points clefs du déploiement. En fonction du langage de programmation de l'IP, le type d'information requis peut varier légèrement. De manière générale, il faut entre autres savoir le nom de la fonction à appeler, l'ordre dans lequel les entrées et les sorties doivent être passées ou bien quel nom de paramètre utiliser, à quoi correspond la valeur de retour de la fonction, quel est le format des données, si les arguments sont passés par valeur ou par référence. En outre, il peut être aussi nécessaire d'appeler une fonction d'initialisation avant le premier appel, ou encore passer des paramètres supplémentaires qui sont indépendants des données. Si le code généré et le code de l'IP sont dans des langages différents (par exemple en C et en C++, ou VHDL et Verilog), d'autres informations encore peuvent être nécessaires, telles que le type de passage des valeurs (ordre sur la pile, etc.).

Pour décrire le protocole d'appel à une fonction d'IP, deux approches sont possibles :

- La mise en place d'un mécanisme qui permet de décrire tous les protocoles que l'on peut rencontrer. Avec cette approche, l'intégration de n'importe quel nouvel IP ne nécessite que de spécifier correctement chaque attribut. Mais un très grand nombre d'attributs est requis et il est très difficile de s'assurer qu'une fois le méta-modèle fixé

on ne rencontrera pas un protocole d'appel qui a besoin d'autres attributs pour être décrit. Pour mettre en relief ces difficultés, remarquons que les 72 pages du standard OMG *Interface Definition Language* [76] ne permettent que de définir des interfaces de fonctions logicielles, pas des IP matériels.
– La mise en place d'un mécanisme qui permet de décrire un nombre restreint de variantes, couplé à un protocole fixe. L'intégration d'un IP peut requérir l'écriture d'un adaptateur entre ce protocole fixé et le protocole utilisé par l'IP. L'avantage est que le nombre d'attributs dans le méta-modèle est très limité, donc plus simple à spécifier. Surtout, cela assure que n'importe quel IP puisse être intégré. D'une part tous les usages de composants élémentaires Gaspard sont spécifiables via les attributs, et d'autre part, tous les IP sont intégrables (à condition d'écrire un adaptateur).

Nous avons retenu la seconde approche, afin de pouvoir utiliser n'importe quel type d'IP sans avoir à étendre le méta-modèle (et implicitement les transformations). Le protocole pour appeler l'IP dépend du langage dans lequel il est codé. Ce langage est indiqué par l'attribut `language` qui prend une valeur de l'énumération *ImplementationLanguage*. La plupart des valeurs de cette énumération sont directement compréhensibles (Fortran, VHDL...). La valeur `softwareBinary` ne peut s'appliquer qu'à un IP logiciel placé sur un processeur et signifie que le code est déjà compilé, cela implique donc que seule l'étape d'édition de liens est nécessaire. Dans la même veine mais pour les IP logiciels placés sur un accélérateur matériel, la valeur `hardwareImplemented` indique que le composant matériel est déjà existant (par exemple un IP de calcul de FFT), et qu'il n'y a donc à générer que les appels à la tâche placée sur ce composant, pas le composant lui-même.

La manière usuelle d'appeler une fonction déjà écrite varie suivant chaque langage. De plus, le type d'information à passer est différent selon que ce soit un IP logiciel ou matériel. En théorie, il faut donc définir un protocole d'appel par langage, qui est respecté par l'IP, et auquel le code généré devra se conformer quel que soit le langage de programmation ciblé. Dans la pratique, la tâche est bien moins ardue qu'il pourrait en avoir l'air. En effet, dans tous les langages logiciels que nous souhaitons supporter (C, C++, Fortran...) il est possible d'utiliser le protocole d'appel standard C. Nous définissons donc ce protocole d'appel pour tous les IP logiciels écrits dans un langage qui le supporte.

Le protocole pour les langages logiciels définit tout d'abord que chaque instance de tâche Gaspard correspond à un appel à l'IP. L'IP correspond à une fonction qui retourne un entier et dont le nom est défini par l'attribut `functionName`. Si la valeur de retour est nulle, l'exécution s'est bien passée, autrement une erreur s'est produite et donc la sortie est indéfinie. Chaque motif d'entrée est passé à l'aide de l'adresse mémoire où se trouve un tableau de taille fixe linéarisé (en effet, dans Gaspard, la taille des tableaux est connue à la compilation). De même pour les motifs de sortie, chaque paramètre correspond à l'adresse à laquelle écrire, où la mémoire a déjà été allouée mais sans être initialisée à une valeur particulière. L'ordre dans lequel les paramètres sont appelés est celui de la liste ordonnée de ports `parameters`. Cette information est nécessaire car lors de la modélisation, les ports, qui représentent les motifs, ne sont pas ordonnés (le contraire aurait d'ailleurs peu de sens dans le cas d'une représentation graphique). Le type des paramètres est dérivé de celui des ports. Ainsi, un IP dont le langage est `C`, le nom de fonction est `convolution` et l'ordre des paramètres est `in1, in2, out1`, chacun étant un tableau de `float` d'une dimension, va correspondre à une fonction déclarée ainsi : `int convolution(float *in1, float *in2, float *out1);`

Le protocole pour les langages matériels est relativement différent mais repose sur les mêmes informations. Le nom du composant est défini par l'attribut `functionName`. Si le

langage nécessite d'un ordre pour les ports, il est donné à l'aide de `parameters`, autrement chaque port du code de l'IP doit être nommé identiquement au *PortImplementation* correspondant. Pour les implémentations précises au bit prêt, par exemple en VHDL ou en Verilog, chaque composant, possède implicitement un port `reset` et un port `clk` qui correspondent respectivement à la remise à zéro et à l'horloge. Même si le composant n'utilise pas l'un de ces ports, ils doivent être présents sur son interface. En SystemC, l'instanciation de l'IP se fait par l'usage (conventionnel) d'un `new`, et le premier argument est une chaîne de caractères correspondant au nom du composant élémentaire qui est déployé. Cet argument n'est pas fonctionnellement nécessaire mais permet de faciliter l'observation de la simulation. Par exemple, pour instancier un composant élémentaire nommé « RAM » déployé sur un IP SystemC qui a pour attribut `functionName` *StaticRam* on utilisera une syntaxe du type : `StaticRam *instanceRAM = new StaticRam("RAM");`

Si l'attribut booléen `indicesInParameters` est vrai alors la fonction appelée recevra une information supplémentaire : l'indice de répétition courant de la tâche. Dans ce cas, chaque appel correspondant à une tâche Gaspard répétée ou chaque instance de composant matériel recevra un indice différent compris entre 0 et le nombre de répétitions (non inclus) pour chaque dimension. Cela peut être utile à certaines fonctions dont les calculs dépendent de l'indice. Cela peut aussi être utile à l'IP matériel d'un processeur qui requière cette information pour pouvoir la transmettre au logiciel afin que ce dernier sache sur quel instance de processeur il s'exécute (le calcul exécuté étant différent suivant l'instance). Tous les indices depuis la racine du modèle sont transmis, concaténés les uns après les autres, partant de la racine et descendant jusqu'au composant élémentaire. Tout comme les entrées et sorties, le passage de cette information est dépendant du langage. Par exemple, pour les descriptions logiciels, nous avons défini que cette information est transmise par l'ajout de deux paramètres supplémentaires à l'appel de fonction. Le premier paramètre est un entier qui indique le nombre de dimensions de la répétition, tandis que le second est un vecteur d'entiers correspondant à chacun des indices. L'appel de fonction est de la forme : `int function_name(`*...entrées et sorties...*`, int dim, int *indices)`. En SystemC, ces deux variables sont transmises comme arguments au constructeur de la classe.

3.2.3.3 *AbstractImplementation*

Une *Implémentation* représente un IP pour une cible (un langage) donnée. Si dans le modèle l'on déploie directement un composant élémentaire sur cet IP, alors on perd en abstraction parce qu'il ne sera possible de générer entièrement le SoC que pour les cibles compatibles avec cet IP. Le modèle devient dépendant de la cible. Le principal désavantage est d'obliger l'utilisateur à modifier le modèle en fonction des transformations qu'il choisit d'appliquer. Pour éviter cela, la notion d'*AbstractImplementation* permet de regrouper un ensemble d'IP qui ont la même fonctionnalité. Via le lien de composition `implementation`, une *AbstractImplementation* peut contenir plusieurs *Implementations*. Chacune de ces *Implementations* doit être équivalente. Elles doivent donc avoir le même nombre de ports d'entrée et de sortie et chacun des ports doit avoir la même sémantique et la même `shape`. L'*AbstractImplementation* doit elle aussi avoir la même interface que toutes les implémentations qu'elle contient. Seules les propriétés non-fonctionnelles des implémentations peuvent varier. Outre le langage de programmation de l'IP, on peut imaginer que des implémentations varient selon leur temps d'exécution, la quantité de mémoire temporaire nécessaire, la précision des calculs, la consommation énergétique, le niveau d'abstraction, le type de processeur pour lequel elles sont

optimisées, etc. Ces propriétés peuvent être spécifiées soit par des attributs supplémentaires propres aux IP logiciels ou matériels comme nous le verrons dans la section suivante ou bien à l'aide de *characteristics* comme nous le verrons dans la section 3.2.7.

Lors du déploiement d'un composant élémentaire, le concepteur de SoC peut lier le composant à l'*AbstractImplementation* en se souciant uniquement de la fonctionnalité que le composant doit avoir, sans devoir spécifier un IP précis. C'est dans une transformation de modèles que le choix entre les différentes implémentations sera fait. La transformation décide celle qui semble la plus adaptée au code qui va être généré. Il n'y a pas de contraintes sur la manière avec laquelle le choix est effectué. Actuellement dans la chaîne de transformation de Gaspard, l'*Implementation* qui a le même langage et le même niveau d'abstraction que la cible est choisie et s'il en existe plusieurs, la première est choisie. Naturellement, des algorithmes plus complexes pourraient être développés, tels la sélection de la moins gourmande en mémoire, ou de la plus performante tout en gardant la consommation d'énergie totale sous un seuil donné, etc. Notons que pour des raisons de simplicité de compréhension et pour favoriser l'usage des *AbstractImplementations*, une *Implementation* doit nécessairement faire partie d'une *AbstractImplementation*, même si elle est seule. Ainsi, si plus tard une autre implémentation est ajoutée, les modèles qui utilisent l'*AbstractImplementation* bénéficieront directement de cet ajout.

3.2.3.4 *Implementor*

La classe *Implementor* joue un rôle mineur. Elle est une généralisation de *AbstractImplementation* et *Implementation*. Comme nous le verrons par la suite dans la section 3.2.6, son usage est d'autoriser le déploiement d'un composant élémentaire directement sur une *Implementation*, au lieu de le déployer sur une *AbstractImplementation*. Elle hérite de la classe *NamedElement*, utilisée dans tout le méta-modèle Gaspard pour indiquer un élément qui peut être nommé. Ce nom n'a pas de signification particulière, il est uniquement utile à l'utilisateur comme documentation. Nous reviendrons plus tard sur le fait qu'il hérite de la méta-classe *Characterizable*.

3.2.3.5 Exemple

La figure 3.16 présente un exemple d'usage des différentes classes que nous venons d'introduire. Cette représentation graphique est basée sur le profil UML correspondant au méta-modèle Gaspard. Dans cet exemple les classes utilisées sont *AbstractSoftwareImplementation* et *SoftwareImplementation*, nous verrons dans la section suivante ce que *Software* implique, mais pour l'instant nous pouvons les considérer comme respectivement une *AbstractImplementation* et une *Implementation*. L'*AbstractImplementation* décrit une fonctionnalité : la convolution (fonction mathématique souvent utilisée dans le domaine du traitement du signal). Cette fonctionnalité est disponible via trois *Implementations* différentes. Celle du haut, `vsipl-convolution`, est codée en C et pour l'utiliser il faudra appeler la fonction `wrap_vsipl_convolution_f()` avec comme argument les deux motifs d'entrée et le motif de sortie. Pour la compiler, il faut compiler un fichier défini par le *CodeFile* qui lui est associée : `GaspardLib/Software/convolution/wrap_vsipl_convolution.c`. En plus des options de compilation par défaut, il faut passer au compilateur `-O2` et pour lors de l'édition de liens il faut utiliser `-lVU -lvsipl -lm` (qui permet de lier le programme avec entre autres la bibliothèque VSIPL). Le type d'information est le même pour les deux autres

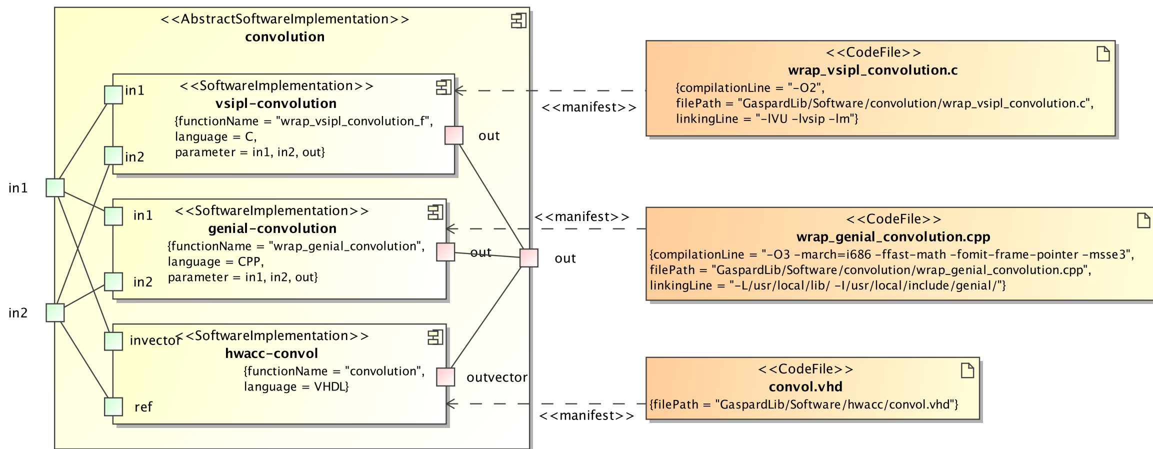


FIG. 3.16: Exemple d'usage des notions d'*Implementor* représenté à l'aide du profil UML. La convolution est une fonction mathématique qui prend deux vecteurs et en produit un. Ici l'usage de cette fonction est disponible en C, C++, et VHDL, chacune de ces implémentations est définie dans un fichier de code source.

implémentations, l'une en C++ et l'autre en VHDL. Enfin, on peut remarquer que chacune des *Implementations* possède une interface similaire à celle de l'*AbstractImplementation*.

3.2.4 Distinction *Software/Hardware*

Orthogonalement aux deux notions *AbstractImplementation* et *Implementation*, il existe dans le méta-modèle les notions *Software* et *Hardware* qui permettent de spécifier respectivement si une implémentation est logicielle ou matérielle. Une vue locale du méta-modèle autour de ces notions est présentée dans la figure 3.17. Un composant élémentaire du modèle d'application ne peut être déployé que sur une implémentation de type *Software*, tandis qu'un composant élémentaire du modèle d'architecture ne peut être déployé que sur une implémentation de type *Hardware*. La combinaison de ces deux distinctions produit quatre classes : *AbstractSoftwareImplementation*, *AbstractHardwareImplementation*, *SoftwareImplementation*, et *HardwareImplementation*. La première ne peut contenir que des implémentations de type *SoftwareImplementation*.

3.2.4.1 *AbstractHardwareImplementation*

Similairement, *AbstractHardwareImplementation* ne peut contenir que des implémentations de type *HardwareImplementation*. C'est une classe abstraite car le type exact de l'implémentation doit être indiqué : processeur, mémoire, réseau de communication... Pour chaque type il existe des attributs particuliers permettant de décrire avec précision le composant afin d'aider à la génération automatique de l'architecture. Par exemple, l'attribut `area` permet d'indiquer la surface utilisée sur la puce par l'IP. La liste complète est disponible dans la figure A.1, nous ne détaillerons cependant pas plus la signification de chaque classe. On pourra se reporter à la thèse de Rabie Ben-Atitallah pour de plus amples informations. Tous les IP représentés par les *HardwareImplementations* contenues dans une *AbstractHardwareImplementation* doivent

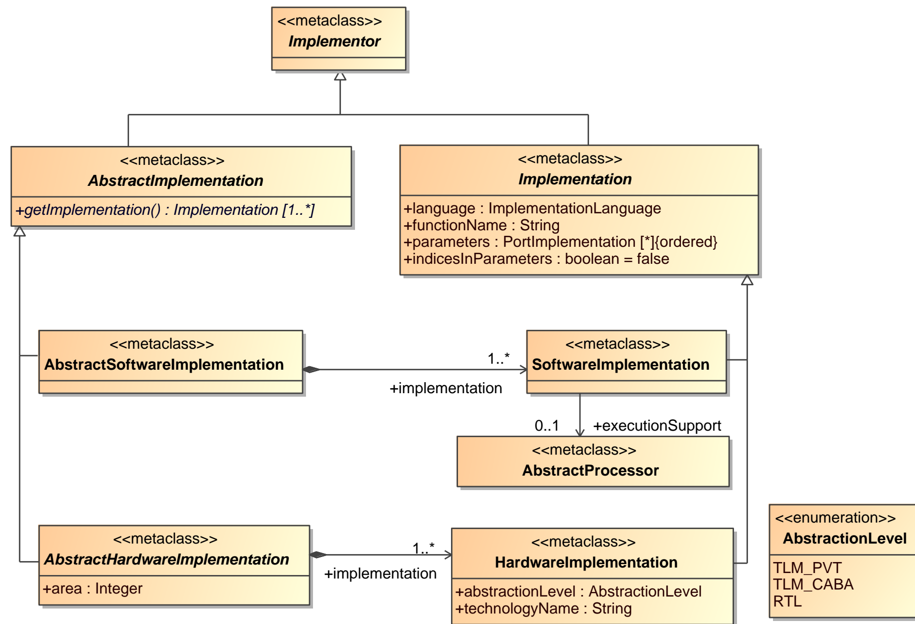


FIG. 3.17: Vue locale du paquetage *Deployment* des distinctions *Software*/*Hardware*.

avoir les mêmes fonctionnalités. Ils doivent tous correspondre à des réalisations physiques qui possèdent exactement le même comportement. Seul le langage dans lequel il est écrit, le niveau d'abstraction, la technologie peuvent varier. Par exemple si c'est une mémoire, toutes les implémentations doivent avoir la même taille, la même largeur de mot, la même vitesse de rafraîchissement. . .

3.2.4.2 *SoftwareImplementation*

La classe *SoftwareImplementation* hérite de *Implementation*, et a donc pour but de représenter une implémentation particulière d'un IP. Elle correspond à un composant élémentaire de l'application. Peu importe la forme sous laquelle l'IP est finalement généré, tous les IP qui correspondent à une tâche Gaspard sont représentés par cette classe. Par exemple, un IP qui effectue une transformée de Fourrier sera représenté par une *SoftwareImplementation* même s'il est écrit en VHDL ou en Verilog (pour être placé sur un accélérateur matériel). En effet, l'information de la forme sous laquelle va être le composant est déjà convoyée par l'association : une tâche peut être placée sur un processeur programmable, un processeur spécialisé (DSP) ou un accélérateur matériel. La classe possède un attribut supplémentaire `executionSupport` qui référence un processeur abstrait (c'est-à-dire un ensemble d'IP de processeurs ayant les mêmes caractéristiques). Il permet d'indiquer que cette implémentation n'est valide que pour un processeur donné. Bien qu'en général ce ne soit pas le cas, cela peut être nécessaire par exemple si l'IP est codé en assembleur (et donc adapté uniquement à un processeur).

3.2.4.3 *HardwareImplementation*

La classe *HardwareImplementation* permet de représenter un IP correspondant à un composant élémentaire du modèle d'architecture. Deux attributs particuliers permettent de différencier l'usage de l'IP par rapport aux autres. `abstractionLevel` est une énumération qui indique le niveau d'abstraction dans lequel est codé l'IP. Les trois valeurs possibles sont `TLMPVT`, `TLMCABA` et `RTL`. Dans le cadre de Gaspard, cela permet de choisir l'IP le plus adapté au niveau d'abstraction de la simulation que l'on souhaite générer. Les travaux effectués au sein de l'équipe par Lossan Bondé [20] permettent l'interopérabilité entre les niveaux d'abstraction, utile lorsque certains composants d'un modèle sont déployés sur des IP qui ne sont pas disponibles à un niveau d'abstraction donné, la simulation utilise alors une implémentation de ces IP ayant un autre niveau d'abstraction. L'attribut `technologyName` spécifie la finesse de gravure de l'implémentation physique que l'IP représente. Ce peut être par exemple `90nm`, `45nm`, etc. La technologie d'implémentation peut en effet faire varier la consommation d'énergie. Lorsqu'il n'est pas spécifié, l'IP est considéré comme pouvant représenter une implémentation à une finesse de gravure quelconque. Une *HardwareImplementation* peut également contenir un *PowerModel*. Présenté en détail dans la thèse de Rabie Ben Atitallah [16], il permet de spécifier un modèle de consommation d'énergie qui sera utilisé lors de la simulation de l'architecture.

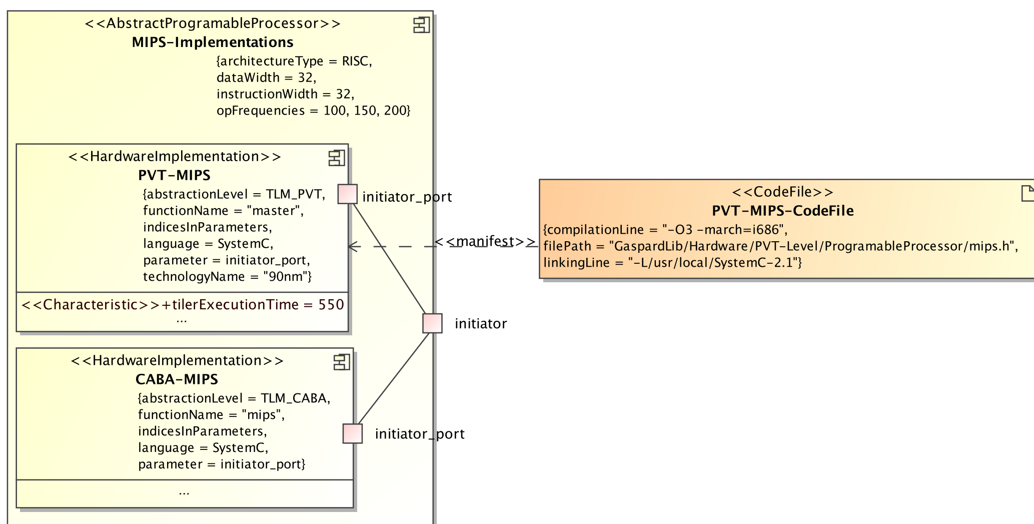


FIG. 3.18: Exemple d'utilisation de *HardwareImplementation* et *AbstractHardwareImplementation* à l'aide du profil UML. Deux implémentations du même processeur MIPS sont disponibles, leurs niveaux d'abstraction diffèrent.

3.2.4.4 Exemple

Dans la figure 3.16 l'usage des classes *AbstractSoftwareImplementation* et *SoftwareImplementation* représente des IP sur lesquels des composants élémentaires applicatifs peuvent être déployés. Parmi les trois implémentations disponibles, deux seront générées comme logiciel tandis que la troisième (en VHDL) sera générée sous forme de matériel. Néanmoins,

elles représentent toutes la même fonctionnalité applicative, une convolution. La figure 3.18 montre l'usage des classes *AbstractHardwareImplementation* et *HardwareImplementation*. La classe *AbstractProgramableProcessor* est une spécialisation d'*AbstractHardwareImplementation*. `MIPS-Implementations` correspond à un ensemble d'IP qui représente le même processeur physique (un MIPS 32 bits qui peut être cadencé à 100, 150 ou 200 MHz). Les deux implémentations disponibles, `PVT-MIPS` et `CABA-MIPS`, sont toutes les deux codées en SystemC. Elles diffèrent principalement sur le niveau d'abstraction. La première propose un niveau PVT, tandis que la seconde propose un niveau CABA. Selon le niveau d'abstraction de la simulation que l'utilisateur souhaitera générer l'une ou l'autre sera sélectionnée pour simuler le processeur MIPS. L'attribut `indicesInParameters` est activé car les IP ont besoin de connaître le numéro de répétition du processeur, information utile au logiciel qui s'exécute sur les processeurs.

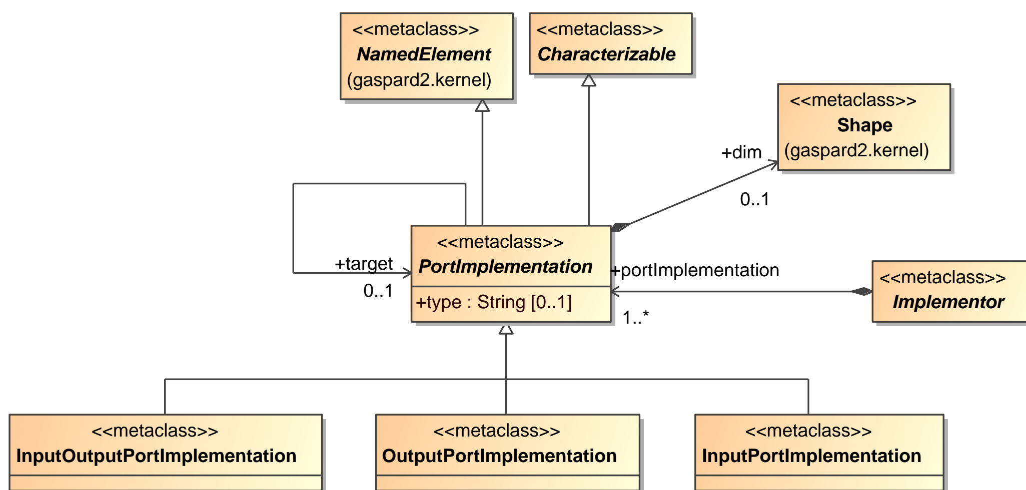


FIG. 3.19: Vue locale du paquetage *Deployment* autour de la notion de *portImplementation*.

3.2.5 Notion de *PortImplementation*

L'information la plus importante sur l'IP pour générer le code qui va lier cet IP au reste du SoC est le format de ses interfaces. Cette interface rassemble l'ensemble des ports des composants Gaspard, via lesquels toute connexion avec le reste du SoC est effectuée. La notion de *PortImplementation* permet d'identifier pour chacun des ports d'un composant élémentaire la correspondance avec une interface de communication de l'IP. La figure 3.19 présente la vue locale de la notion de *portImplementation*

3.2.5.1 *PortImplementation*

Les *PortImplementations* appartiennent à un *Implementor*, c'est-à-dire à une implémentation ou à une implémentation abstraite. En pratique ceci signifie que d'un point de vue extérieur une implémentation abstraite est similaire à une implémentation. Ainsi on peut connecter un composant élémentaire à une *AbstractImplementation* en la considérant totalement équivalente

à l'implémentation de l'IP. La classe *PortImplementation* est en réalité utilisable sous trois formes différentes : *OutputPortImplementation*, *InputPortImplementation*, et *InputOutputPortImplementation*. La différence entre ces trois formes est similaire à celle entre les trois formes de port Gaspard. La première forme indique une interface qui émet des données, la seconde indique une interface qui reçoit des données et enfin la troisième correspond à une interface qui peut tout aussi bien émettre que recevoir les données. Dans le cadre d'IP matériels, un port émetteur doit se comprendre comme initiateur d'une communication tandis qu'un port récepteur correspond à une interface esclave.

L'attribut `target` permet à une *PortImplementation* contenue dans une *Implementation* de référencer le *PortImplementation* équivalent de l'*AbstractImplementation* englobante. Cette référence est doit être obligatoirement présente, elle indique quel port de l'implémentation abstraite correspond à quel port de chacune des implémentations contenues. Implicitement, les ports des *AbstractImplementation* ont toujours cet attribut vide.

Une *PortImplementation* porte aussi un nom (car elle hérite de la classe *NamedElement*). En fonction du langage dans lequel est codé l'IP, le nom a une importance plus ou moins grande. Dans les langages où les ports ou paramètres d'une fonction sont spécifiés par nom, comme en VHDL, cet attribut est utilisé lors de la génération du code pour obtenir l'appel correct. En revanche, dans les langages tels que le C, où seul l'ordre des paramètres compte lors d'un appel de fonction, cet attribut joue uniquement un rôle de documentation pour l'utilisateur. Les noms des ports des *AbstractImplementation* jouent toujours un rôle uniquement de documentation.

Enfin, avec les attributs *type* et *dim*, on peut spécifier le type et la taille du port. Dans le cadre d'un IP logiciel, si le port a une taille spécifiée cela indique que l'interface prend un tableau d'élément. Au contraire dans le cas d'un IP matériel, si le port a une taille alors cela signifie que le port est répété selon la forme spécifiée, chaque répétition du port correspondant à une connexion, autrement dit cela correspond à un tableau de ports et non pas à un port sur lequel un tableau d'éléments est transmis. Le type du port est spécifié par une chaîne de caractères. Cette chaîne est écrite selon le langage de programmation de l'IP. Elle est utilisée telle quelle lors de la génération de code. Une autre solution aurait été d'utiliser, et d'éventuellement étendre, les notions de type du méta-modèle Gaspard. Cette solution n'a pas été retenue car elle oblige d'une part l'utilisateur à entrer les concepts qui n'ont pas d'utilité à être détaillés actuellement et d'autre part que chacune des transformations de génération de code soit complétée pour qu'elle puisse générer le code équivalent aux concepts. À terme, le besoin de pouvoir interpréter le type par les transformations (par exemple pour autoriser l'interopérabilité entre langages) pourrait imposer la seconde solution.

Éventuellement, un port d'une *AbstractImplementation* peut avoir les attributs *type* ou *dim* spécifiés. Dans ce cas, il est obligatoire que *tous* les ports qui lui sont liés aient la même valeur pour l'attribut en question. Cela peut être utile pour forcer toutes les implémentations à respecter un type d'interface donné.

Dans la section suivante nous détaillerons le fait que la classe *PortImplementation* hérite de *Characterizable*. Avant cela, regardons l'usage de *PortImplementation* sur un exemple.

3.2.5.2 Exemple

La figure 3.20 contient la définition selon le profil UML Gaspard d'un IP logiciel qui calcule la valeur absolue des pixels d'une image en niveaux de gris. Comme on peut le voir par les spécifications de l'*InputPortImplementation* et de l'*OutputPortImplementation* de

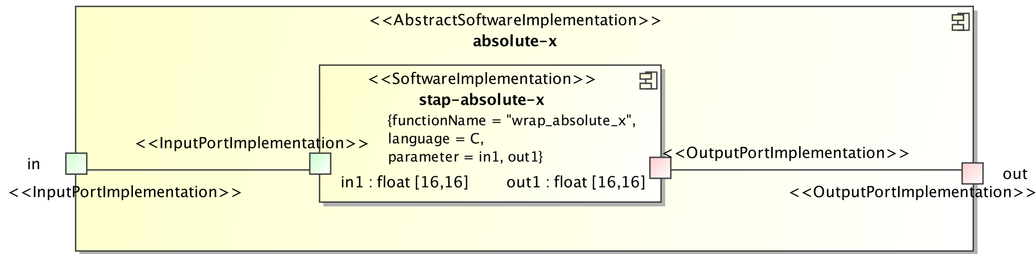


FIG. 3.20: Exemple d'usage de *PortImplementation* exprimé avec le profil UML. Les ports de l'implémentation sont détaillés et reliés aux ports de l'implémentation abstraite.

`stap-absolute-x`, la fonction décrite prend en entrée un tableau de taille 16×16 d'éléments de type `float` (c'est-à-dire un nombre en virgule flottante en C) et produit un tableau de même type en sortie. Chacun de ces ports est connecté au port correspondant de l'implémentation abstraite. Dans la figure 3.16 précédente, on peut observer l'usage des connexions lorsqu'il existe plusieurs *Implementations*. Chaque port des implémentations est connecté à un, et un seul, port différent de l'implémentation abstraite.

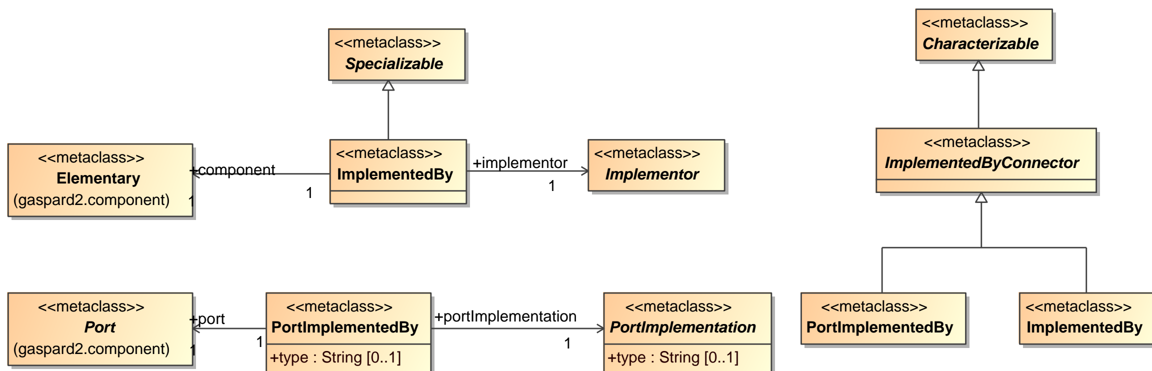


FIG. 3.21: Vue locale du paquetage *Deployment* autour de la notion d'*ImplementedBy*.

3.2.6 Notion d'*ImplementedByConnector*

Jusqu'à présent nous avons vu les mécanismes permettant la description d'IP et de leur interface. La notion d'*ImplementedByConnector* permet de faire le lien entre cette description et les composants élémentaires. Il existe deux types de connecteurs. Un *ImplementedBy* permet de lier le composant à l'implémentation tandis qu'un *PortImplementedBy* permet d'associer un port du composant à un port de l'implémentation. Ces deux classes, représentées figure 3.21 sont contenues dans un *DeploymentSpecificationModel*, la racine du paquetage. C'est-à-dire qu'elles sont indépendantes des autres classes, ce qui autorise la construction de bibliothèques d'IP : les implémentations et les fichiers de code sont définis dans le modèle de manière auto-suffisante, puis on peut modifier indépendamment les *ImplementedByConnectors* qui font

la liaison entre ces définitions d'IP et le modèle de SoC. Notons au passage que dans un même modèle on peut parfaitement envisager plusieurs composants élémentaires déployés sur le même IP.

3.2.6.1 *ImplementedBy*

La classe *ImplementedBy* référence un *Elementary* (composant élémentaire) et un *Implementor*. Elle fait office de connecteur. Le composant élémentaire est alors considéré comme *déployé* sur l'*Implementor* : à la génération de code la boîte noire que représente le composant sera remplacée par le code de l'IP correspondant à l'implémentation. Un composant élémentaire ne peut être référencé que par un seul *ImplementedBy*, il n'y aurait pas de sens qu'il soit déployé plusieurs fois sur différents IP. Usuellement, ce connecteur référence une *AbstractImplementation*, c'est-à-dire une fonctionnalité. Cependant, si l'utilisateur souhaite forcer la sélection d'une *Implementation* en particulier, il peut le faire en faisant pointer ce connecteur directement vers celle-ci. C'est pour autoriser ce genre de construction que la classe référence un *Implementor*.

3.2.6.2 *PortImplementedBy*

De même, les *PortImplementedBys* référencent un *Port* et un *PortImplementation*. Le *Port* doit appartenir à un composant élémentaire et doit avoir la même direction que le *PortImplementation* (*In*, *Out*, ou *InOut*). Le *PortImplementation* doit appartenir à une *AbstractImplementation*. *Port* et *PortImplementation* ont chacun un `type` et une `shape`. Si un de ces attributs est spécifié sur les deux ports alors ils doivent être identiques, le modèle reste cohérent. Si l'attribut d'un seul des ports est spécifié alors implicitement l'autre port aura le même type ou la même forme. Dans le cas où ni *Port* ni *PortImplementation* n'ont de type de défini, alors il faut que le type soit spécifié dans l'attribut `type` du connecteur.

Lors du déploiement d'un composant élémentaire, ce composant doit être associé à une implémentation ou une implémentation abstraite et chacun de ses ports doit être relié à un *PortImplementation* différent de l'implémentation abstraite. Même si le composant est déployé sur une implémentation (pour forcer sa sélection), les ports doivent être déployés sur les ports de l'implémentation abstraite. Cette règle a simplement pour but de faciliter le changement d'implémentation : il suffit de modifier une référence.

3.2.6.3 Exemple

La figure 3.22 présente un exemple d'usage de ces connecteurs. La tâche élémentaire `Multiplication` est déployée sur l'implémentation abstraite `mult`. Pour cela, un connecteur *ImplementedBy* relie la tâche à l'implémentation abstraite tandis que trois connecteurs *PortImplementedBys* relient les ports de la tâche aux ports de l'implémentation. La direction et le type des ports sont respectés.

3.2.7 Notions de *Characterizable* et *Specializable*

Nous allons maintenant aborder deux notions relativement semblables qui ne sont pas nécessaires pour décrire un IP mais introduites afin de pouvoir modéliser avec plus de souplesse et plus rapidement les IP. Une vue locale du méta-modèle autour de ces notions est présentée dans la figure 3.23. *Characterizable* et *Specializable* sont utilisées de la même

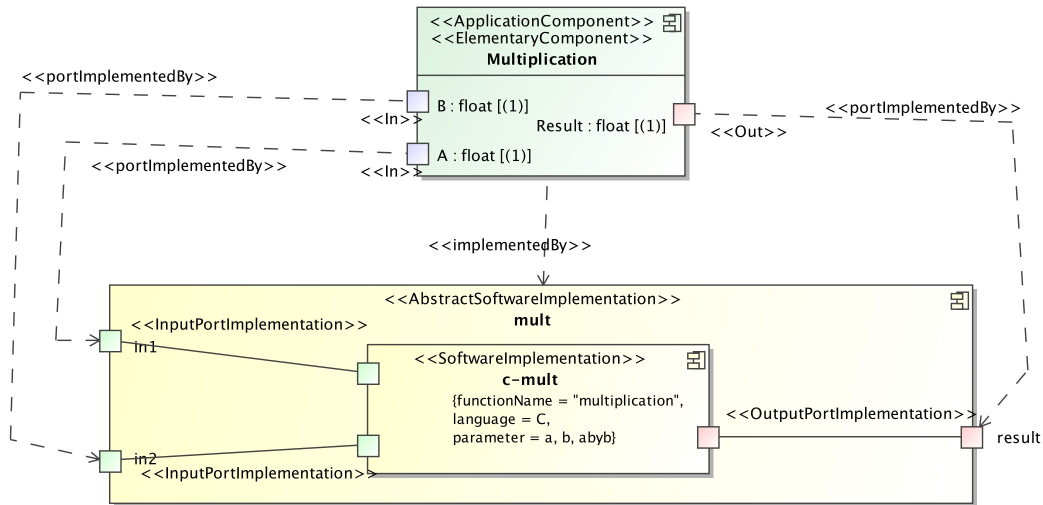


FIG. 3.22: Exemple de déploiement d'une tâche élémentaire à l'aide de *PortImplementedBys* et d'*ImplementedBy* selon le profil UML.

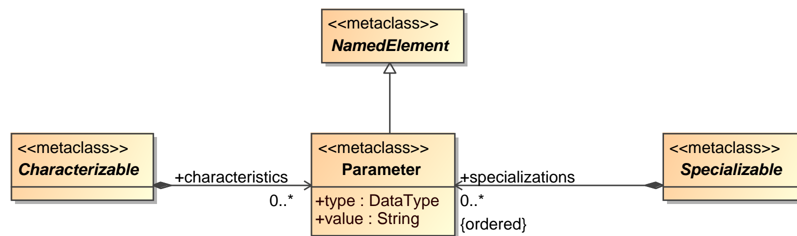


FIG. 3.23: Vue locale du paquetage *Deployment* autour des notions *Characterizable* et *Specializable*.

manière dans le méta-modèle : une classe qui hérite de l'une d'entres-elles est capable de contenir respectivement des `characteristics` ou des `specializations`. Ce sont des paramètres spéciaux que le concepteur de SoC va pouvoir utiliser pour passer des informations supplémentaires directement aux transformations ou au compilateur.

3.2.7.1 *Parameter*

Les deux classes *Characterizable* et *Specializable* peuvent contenir 0 ou plus *Parameter*. Un *Parameter* est similaire à la notion d'*attribut* en MOF. Avec les trois champs `name`, `type`, et `value` il permet au concepteur de SoC de créer ses propres attributs, ou paramètre. Le nom du paramètre permet de le différencier des autres. Le champ `type` indique bien sûr le type de la valeur de ce paramètre, ce ne peut être qu'un type simple défini dans le méta-modèle, tel que entier, booléen, chaîne de caractères… Enfin, la valeur est entrée sous forme de chaîne de caractères, ce qui permet d'entrer n'importe quel type de valeur.

3.2.7.2 *Characterizable*

Via la classe *Characterizable*, on donne la possibilité au concepteur de créer des paramètres appelés `characteristics`. Un tel paramètre a pour but de passer une information particulière à une *transformation*. Selon la présence et la valeur d'une `characteristic` précisée sur une classe, une transformation peut varier le modèle de sortie généré. Ce sont des attributs qui auraient pu être directement placés sur une classe du méta-modèle Gaspard mais qui n'ont d'utilité que lors de la génération d'une cible particulière, ou bien dont l'usage n'avait pas été envisagé lors de l'écriture du méta-modèle. Par exemple, la transformation que nous avons écrite qui génère une simulation à haut niveau d'abstraction, accepte une `characteristic` spéciale nommée `tilerExecutionTime` dont la valeur qui indique le temps d'exécution d'un calcul d'adresse lorsqu'elle est placée sur un processeur. Cette information n'a pas d'utilité pour les simulations de niveau d'abstraction plus bas ni pour les autres transformations, et ne correspond vraiment pas à la spécification typique d'un processeur.

Les classes *Implementation* et *PortImplementation* pourraient foisonner d'attributs pour décrire en détail tous les IP possibles en fonction de chaque cible de transformation possible, mais il n'y aurait que très peu d'usage de ces informations et l'introduction d'une nouvelle transformation pourrait nécessiter l'ajout d'attributs supplémentaires au méta-modèle. En déplaçant la déclaration de ces informations dans la transformation qui va en faire usage, le méta-modèle reste simple et on assouplit la modélisation des IP en évitant que les descriptions autorisées soient fixées dans le méta-modèle. Notons que la documentation de ces informations (leur nom, leur type et leur signification) n'est cependant pas moins présente. La documentation d'une `characteristic` est simplement déplacée du méta-modèle vers la documentation de la transformation qui la prend en charge.

3.2.7.3 *Specializable*

Via la classe *Specializable*, on donne la possibilité au concepteur de créer des paramètres appelés `specializations`. Tous les paramètres qui concernent une implémentation, y compris ceux placés sur les ports, sont transmis directement au code de l'IP. Lors de la compilation, le code peut utiliser ces informations pour obtenir une version *spécialisée* de l'IP. Typiquement ce genre de paramètre va permettre d'utiliser le même code d'IP pour des implémentations qui diffèrent selon le type des données qui sont traitées (float, int, double...) ou selon la taille des motifs de calcul ou du nombre de ports. Ainsi la description d'une implémentation peut être généralisée, et sa spécialisation est sélectionnée par le concepteur lors du déploiement du composant élémentaire, en fonction des besoins du SoC.

Cela introduit une forme simple de *patron*, où un seul code d'IP peut être utilisé pour obtenir plusieurs variantes. Bien que l'on puisse modéliser les IP sans cette notion, c'est un concept utile afin d'éviter de devoir modéliser un grand nombre d'implémentations très similaires. Ainsi il est possible de créer une bibliothèque d'IP génériques : lors de son utilisation avec un modèle de SoC le concepteur va adapter les implémentations au besoin des composants élémentaires en indiquant la valeur prise par chacune des `specializations`.

La manière avec laquelle les `specializations` sont passées au code de l'IP varie en fonction du langage de programmation. Pour chaque langage possible dans Gaspard, il y a une manière définie. En C, cela ce fait par l'utilisation des variables du préprocesseur. Par exemple, une `specialization` nommée *TYPEIN* et ayant pour valeur *int* sera équivalent à avoir ajouté au début du code de l'IP `#define TYPEIN int`. En VHDL, cela ce fait par

l'utilisation de generics. En C++, le passage de l'information utilise le mécanisme de template interne au langage. Pour des cas tels que celui-ci, où l'ordre de passage des valeurs importe, les `specializations` sont ordonnées (indiqué par `{ordered}` dans le méta-modèle).

3.2.7.4 Impact sur les autres classes

Dans le paquetage *Deployment*, *Specializable* et *Characterizable* ont été placées de manière à généraliser à la fois les concepts pour décrire les IP et les connecteurs *ImplementedBy* qui pointent vers ces concepts. *Characterizable* généralise *Implementation*, *AbstractImplementation*, *PortImplementation* et aussi *ImplementedBy* et *PortImplementedBy*. L'ajout de caractéristiques pour les transformations peut être utile à la fois sur implémentation, sur un port, ou sur une implémentation abstraite. Dans ce dernier cas, toutes les implémentations contenues dans l'implémentation abstraite peuvent être considérées comme ayant aussi la caractéristique. Quant à *Specializable*, elle généralise *Implementation* et aussi *ImplementedBy* (qui relie un composant élémentaire à une *Implementation*). En effet, la spécialisation s'applique au code lui-même, il n'y a donc pas de sens de spécialiser un port ou une implémentation abstraite. Dans le cas où plusieurs fichiers de code sont associés à l'implémentation, la spécialisation est appliquée à tous les fichiers. L'idée de généraliser aussi les connecteurs de déploiement est d'autoriser l'utilisation des paramètres aussi lors de l'usage d'une bibliothèque. Cela permet la création d'un côté d'un modèle contenant une bibliothèque d'IP dans leur version générique et d'un autre côté un modèle de SoC utilisant ces IP en les paramétrant via les connecteurs de déploiement.

Pour éviter les incohérences, un même paramètre ne peut pas être défini à plusieurs niveaux en même temps (par exemple à la fois sur une implémentation et sur un connecteur d'implémentation). Néanmoins, il est possible d'ajouter un paramètre sans lui assigner de valeur (en spécifiant juste son nom et son type). Dans ce cas le paramètre ne n'est pas considéré du tout par les transformations et il a uniquement un usage de documentation pour indiquer à l'utilisateur de la bibliothèque d'IP qu'une implémentation ou un port *pourrait* avoir ce paramètre de spécifié, via la redéfinition du paramètre sur un connecteur.

3.2.7.5 Exemple

La figure 3.24 présente un exemple d'utilisation des `specializations` et `characteristics`. La tâche élémentaire `SumVector` est déployée sur l'implémentation abstraite `vector-add` qui ne contient qu'une implémentation, `c-vadd`. La fonctionnalité de l'implémentation est de sommer les éléments d'un vecteur. L'implémentation a une `characteristic` nommée `executionTime` qui a pour valeur 1650. Cette information sera utilisée par la transformation générant une simulation à haut niveau d'abstraction pour simuler approximativement le temps d'exécution d'un appel à l'IP (la valeur est en nanosecondes pour un processeur fixé). Éventuellement, cette information pourrait être aussi utilisée par d'autres transformations. Par exemple s'il existe plusieurs implémentations compatibles avec la cible on pourrait imaginer que la transformation qui choisit la meilleure implémentation utilise cette information pour sélectionner la plus rapide. Il y a aussi une `characteristic` sur le port d'entrée, elle est spécifiée uniquement via le connecteur *PortImplementedBy*.

L'implémentation possède également une `specialization`, nommée `VECTOR_SIZE`, mais dont la valeur n'est pas spécifiée. Elle n'a pour but que de permettre au concepteur du SoC de se souvenir lors du déploiement que cette `specialization` est disponible. C'est

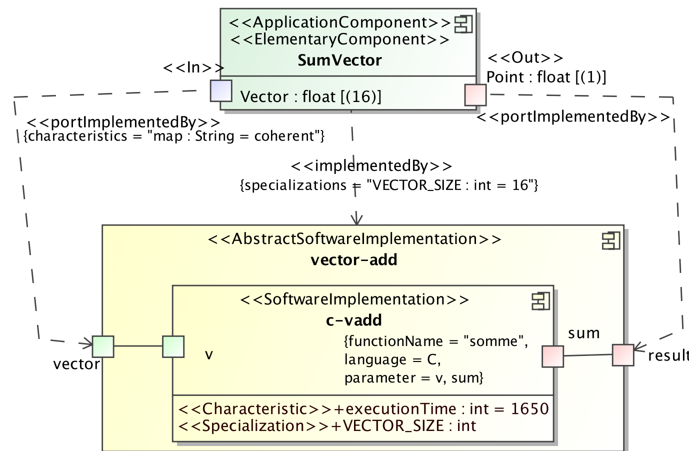


FIG. 3.24: Exemple d'utilisation des notions de *Characterizable* et *Specializable* dans une vue selon le profil UML. Une tâche élémentaire est déployée sur une implémentation. Plusieurs `specializations` et `characteristics` sont spécifiées, au niveau de l'implémentation et au niveau des connecteurs.

ainsi, qu'on la retrouve sur le connecteur *ImplementedBy*. Elle a, cette fois ci, sa valeur de fixée à 16. Le code de l'IP est en effet générique pour faire la somme des éléments d'un vecteur de n'importe quelle taille. L'entrée de la tâche élémentaire étant de 16 nombres, cette spécialisation permet d'adapter l'IP à la tâche.

3.2.8 GaspardLib, une bibliothèque de composants pour la modélisation de SoC

Nous venons de balayer l'ensemble des notions du paquetage *Deployment*. L'une des contraintes du paquetage était la possibilité de créer une bibliothèque d'IP indépendamment d'un modèle de SoC. Avec le partenariat de deux étudiants de Master et Rabie Ben-Atitallah, nous avons commencé le développement d'une telle bibliothèque. Nommée *GaspardLib*, elle contient des IP logiciels et matériels dont les fonctions sont orientées vers le traitement du signal et les systèmes embarqués. Lors de la modélisation de SoC, le concepteur va pouvoir piocher parmi ces briques de base pour construire à la fois l'architecture et l'application. C'est aussi l'occasion de valider ce nouveau paquetage, d'une part en créant la bibliothèque et en l'utilisant dans différents modèles de SoC et d'autre part en s'assurant qu'il est possible de décrire des IP déjà existants et utilisés faisant partie de bibliothèques largement diffusées.

3.2.8.1 IP logiciels

Les IP logiciels proviennent de trois sources différentes. Il y en a qui proviennent de la bibliothèque C++ GENIAL [63] (GENeric Image Array Library), de la bibliothèque C VSIPL [27] (Vector Signal Image Processing Library) et d'un ensemble de fonctions écrites en C spécialement dans l'optique d'être utilisées pour Array-OL par un partenaire industriel pour le traitement d'image. À ce jour, au total 33 IP ont été ajoutés à la bibliothèque. Les fonctionnalités disponibles sont par exemple FFT (sous différentes variantes), DCT, FIR,

gradient d'une image en niveaux de gris, etc. Lorsque la même fonctionnalité était disponible via plusieurs implémentations, correspondant chacune à la fonction d'une bibliothèque différente, le mécanisme d'*AbstractImplementation* a été utilisé.

L'ajout d'un IP se passe en deux étapes. La première consiste à créer un adaptateur autour de l'appel de l'IP pour qu'il puisse être compatible avec le protocole d'appel standard de Gaspard. La seconde étape consiste à modéliser l'interface de l'adaptateur créé à l'aide du paquetage *Deployment* du méta-modèle. Lorsque l'IP complète une *AbstractImplementation* qui contient déjà d'autres *Implementations*, il faut également s'assurer que toutes les fonctions retournent *exactement* les mêmes sorties pour des entrées identiques.

Par exemple, pour l'utilisation de la fonction de FFT mono-dimensionnelle de VSIPL, nous avons commencé par l'écriture d'un adaptateur. Une FFT mono-dimensionnelle prend en entrée un vecteur de nombres réels et retourne un vecteur de nombres complexes. Telle quelle, la fonction de VSIPL ne peut pas être utilisée dans Gaspard. Il faut convertir les données entre le format qu'utilise Gaspard, c'est-à-dire une représentation linéaire des tableaux multi-dimensionnels selon la norme C, et le format de la bibliothèque qui peut être optimisé pour le type de calculs réalisés ou qui peut contenir plus d'informations (telle la taille du tableau). Dans notre cas, il faut convertir le vecteur d'entrée en un `block` puis le `block` en une `view` pour pouvoir appeler la fonction. Comme c'est le cas ici, l'adaptateur peut aussi avoir à initialiser la fonction. Le code suivant présente le code de l'adaptateur finalement réalisé.

```
1   #include <vsip.h>
    #include <VU.h>
    // LENGTH is size of the input vector
    #ifndef LENGTH
5   #define LENGTH 128
    #endif

    struct dbl_cplx{
            double re;
10          double imag;
    };

    void wrap_vsipl_fft_d(double * in_1, struct dbl_cplx * out)
    {
15          vsip_fft_d * rcc;
            vsip_block_d * block;
            vsip_vview_d * view;
            vsip_cvview_d  *yout;
            int i,j,k;
20          vsip_cscalar_d z;

            vsip_init(NULL); // library initialisation

            // fft structure with all elements
25          rcc = vsip_rcfftop_create_d(LENGTH, 1.0, 1, VSIP_ALG_TIME);
            block = vsip_blockbind_d(in_1, LENGTH, VSIP_MEM_NONE); // data block in parameter
            view = vsip_vbind_d(block, 0, 1, LENGTH); // conversion block -> view
            // window create with complex
            yout = vsip_cvcreate_d((LENGTH/2)+1, VSIP_MEM_NONE);
30
            vsip_rcfftop_d(rcc, view, yout); // Call to the FFT!

            for(j=0; j<(LENGTH/2)+1; j++) // we copy the result of fft ...
            {
35                  z = vsip_cvget_d(yout,j);
                    out[j].re = vsip_real_d(z);
                    out[j].imag = vsip_imag_d(z);
            }

40          for(k=1; k<(LENGTH/2)+2; k++,j++)// and create symetry around LENGTH
            {
                    out[j].re = out[j-2*k].re;
                    out[j].imag = -out[j-2*k].imag;
            }
45
            vsip_finalize(NULL);
```

```
}
```

L'appel aux fonctions `vsip_init()` et `vsip_rcfftop_create_d()` permet d'initialiser la fonction de calcul de FFT. Les fonctions `vsip_blockbind_d()` et `vsip_vbind_d()` se chargent de la conversion du vecteur d'entrée. La fonction `vsip_cvcreate_d()` prépare l'espace temporaire pour sauvegarder le résultat. C'est la fonction `vsip_rcfftop_d()` qui effectue les calculs de la FFT. La première boucle permet de convertir le résultat dans le format de Gaspard. La seconde boucle est particulière, elle se charge de rendre la sortie identique à la sortie de l'IP correspondant de la bibliothèque GENIAL. Enfin, l'appel à `vsip_finalize()` permet de clore l'usage de la bibliothèque VSIPL.

On peut également noter la présence de la constante de préprocesseur `LENGTH`. Elle permet de généraliser la fonction à un vecteur en entrée de n'importe quelle longueur. Si lors de la compilation elle est définie alors la fonction utilisera cette valeur, sinon par défaut la longueur est de 128 éléments.

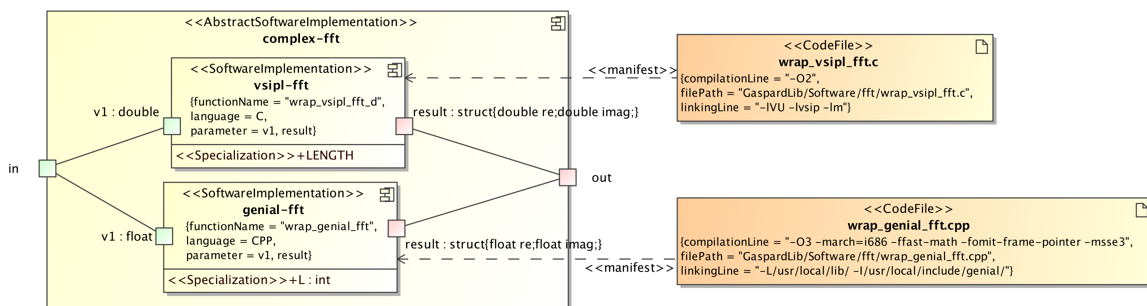


FIG. 3.25: Exemple d'IP logiciel de la bibliothèque GaspardLib. La fonctionnalité de calcul de FFT est disponible via l'usage soit de la bibliothèque GENIAL soit de la bibliothèque VSIPL.

La figure 3.25 présente la modélisation de la fonctionnalité de FFT dans GaspardLib. L'*AbstractSoftwareImplementation* indique la fonctionnalité dans son ensemble : c'est un IP logiciel avec une entrée et une sortie. Elle contient deux *SoftwareImplementations* qui correspondent à la fonctionnalité dans les bibliothèques VSIPL et GENIAL. On trouve ainsi la description à l'adaptateur que nous venons de détailler. Codé en C, il est appelé `wrap_vsipl_fft_d` et peut être spécialisé à l'aide de la spécialisation `LENGTH`. Les types des entrées et des sorties sont spécifiés mais pas leurs dimensions. Ceci permet au concepteur de SoC de décider lui-même de la taille des données sur lesquelles travailler. Il faudra néanmoins qu'il se charge de la cohérence entre la taille des ports du composant élémentaire déployé et la spécialisation `LENGTH`. Le *CodeFile* indique l'emplacement du fichier contenant l'adaptateur. On considère ici que l'utilisateur s'est déjà chargé de compiler la bibliothèque VSIPL. L'attribut `linkingLine` permettra lors de la liaison du code logiciel d'incorporer la bibliothèque.

3.2.8.2 IP matériels

Les IP matériels ajoutés à la bibliothèque GaspardLib proviennent soit de ceux proposés dans soclib [89] soit du développement en interne pour les composants de plus haut niveau d'abstraction (PVT). Il y a actuellement une dizaine de composants disponibles. Les composants permettent de générer une architecture complète, et donc comprennent mémoire,

processeurs, bus mais aussi des composants d'entrée-sortie. Pour l'instant ils sont orientés pour la génération de simulation en SystemC.

Par exemple, pour l'ajout de la fonctionnalité d'un processeur MIPS, nous avions deux versions du composant, à des niveaux d'abstraction différents. Le composant en PVT a été développé en interne en SystemC. Il est constitué de deux fichiers sources. Le premier, `mips.cc` contient le code tandis que le second, `mips.h` contient la définition de la classe SystemC. Il est tel quel :

```
1  #ifndef MASTER_HEADER
   #define MASTER_HEADER

   #include <systemc.h>
5  #include "bus_types.h"
   #include "basic_TLM_initiator_port.h"
   using basic_protocol::basic_timed_initiator_port;

   class master : public sc_module
10 {
   public:
           master( sc_module_name module_name, int index_dim = 0, int *index = NULL);
           SC_HAS_PROCESS( master );
           basic_timed_initiator_port<ADDRESS_TYPE,DATA_TYPE> *initiator_port;
15
   private:
           void run();
   };

20 #endif
```

La classe s'appelle `master` et le constructeur (du même nom) prend un argument qui est le nom de l'instance indiqué dans le modèle du SoC. Cet argument est utile pour obtenir des rapports de performance et un débogage lisible par l'utilisateur. Les deux autres arguments du constructeur permettent de passer à l'instance son indice lorsque le composant est répété. L'appel à `SC_HAS_PROCESS()` est spécifique à SystemC et indique que cette classe représente un composant SystemC. La variable `initiator_port` représente le port par lequel le composant sera connecté. Enfin la méthode privée `run()` est utilisée par SystemC et correspond à l'exécution du composant.

La modélisation de ce composant dans GaspardLib a déjà été présentée dans la figure 3.18. La figure 3.26 présente de nouveau la fonctionnalité MIPS de GaspardLib mais avec une vue plus détaillée. On y voit la fonctionnalité générique appelée `MIPS-implementations` qui est stéréotypée « AbstractProgramableProcessor » pour indiquer que les IP décrits définissent un processeur. À l'intérieur de cette implémentation abstraite, on trouve les implémentations en CABA (nommée `CABA-MIPS`) et en PVT (nommée `PVT-MIPS`). Pour cette dernière, on retrouve le nom de la classe `master` dans l'attribut `functionName` ainsi que le nom du port et son type comme *OutputPortImplementation* puisque le composant est initiateur sur ce port. Le *CodeFile* associé spécifie le fichier `mips.h`, il sera utilisé pour faire la déclaration du composant dans le code généré. Seul le fichier `mips.h` est mentionné, en effet le déploiement en SystemC compile implicitement le fichier `.cc` correspondant au `.h`, il n'est donc pas nécessaire de l'ajouter. Les caractéristiques indiquent à la transformation des informations à propos du temps d'exécution du composant et des tâches sur ce composant afin qu'elle puisse générer une simulation avec une estimation précise du temps.

3.2.9 Synthèse et conclusion : le déploiement d'IP

À travers cette section nous avons détaillé le paquetage *Deployment*. Il complète le méta-modèle Gaspard afin de donner la possibilité au concepteur de SoC de décrire l'usage des IP

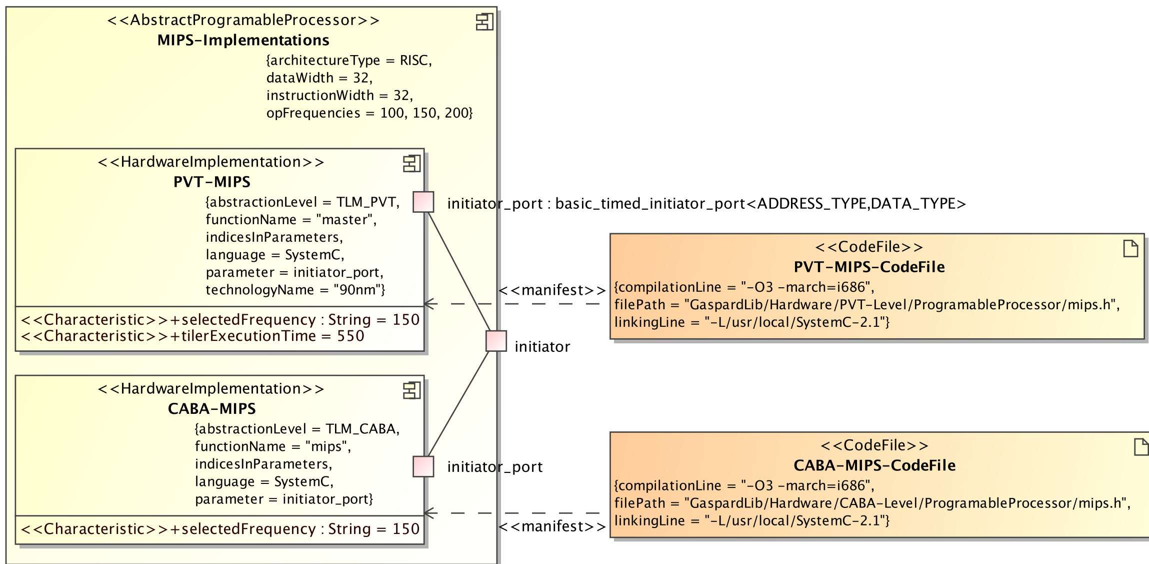


FIG. 3.26: Exemple d'IP matériel de la bibliothèque GaspardLib. La fonctionnalité de MIPS est disponible aux niveaux d'abstraction CABA et PVT.

sur lesquels sont basées les boîtes noires que sont les composants élémentaires. C'est à l'aide de ces descriptions que les transformations de modèles seront en mesure de générer un code complet à partir du modèle de SoC. Ainsi le modèle est rendu *productif* puisque le concepteur n'a plus besoin de modifier le résultat des transformations.

Les concepts du paquetage permettent de décrire chaque fichier concerné par un IP et l'interface de cet IP. Afin de garder le méta-modèle simple, l'usage de l'IP doit correspondre un à protocole fixé par le méta-modèle Gaspard, ce qui peut éventuellement nécessiter un adaptateur pour les cas où l'IP a des attentes qui ne peuvent être directement décrites. À la fois les IP matériels et logiciels peuvent être modélisés, via les mêmes concepts.

En outre, des mécanismes ont été introduits pour faciliter la modélisation et l'évolution du SoC. Tout d'abord, le concept d'abstraction permet de capturer le fait qu'une fonctionnalité puisse être rendue disponible par plusieurs IP qui diffèrent sur certaines propriétés. L'IP le plus adapté peut être automatiquement sélectionné en fonction de l'association ou de la cible de génération. De plus, les connecteurs d'implémentation permettent de séparer les IP du reste du modèle, laissant la possibilité de les placer à part dans une bibliothèque comme nous l'avons montré. Le concept de caractéristique permet de ne pas avoir à étendre le méta-modèle lorsqu'une transformation nécessite des informations spécifiques. Enfin, les spécialisations permettent d'introduire une certaine généricité dans un IP, qui ne peut être fixée que par le concepteur du SoC au moment du déploiement.

Tout cela permet d'avoir un mécanisme simple et efficace pour faire les liens entre les briques de base du modèle et le code source des IP correspondants. On peut néanmoins regretter le protocole fixe d'appel des IP qui force parfois la présence d'adaptateurs réduisant potentiellement la performance. À l'aide d'un système de description plus complet des types, il pourrait être possible de détecter que plusieurs IP liés partagent le même format de donnée et donc ne nécessitent pas d'adaptateur pour les communications entre eux.

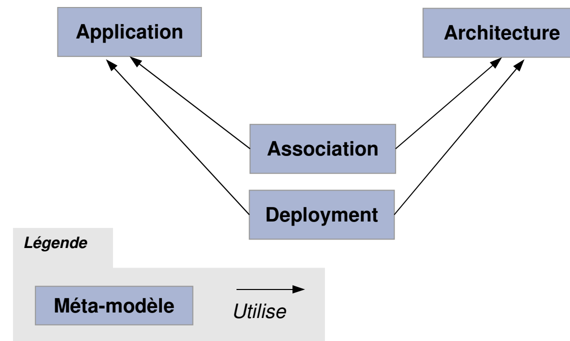


FIG. 3.27: Les paquetages du méta-modèle Gaspard et les relations entre eux.

3.3 Conclusion

Dans ce chapitre, les extensions apportées au méta-modèle original de Gaspard ont été présentées. D'une part, nous avons étendu l'association, principalement par le raffinement de la sémantique afin de pouvoir traiter les modèles complexes, et d'autre part nous avons ajouté l'information de déploiement afin de définir le lien entre les composants élémentaires du modèle et le code existant. Une fois étendu, le méta-modèle Gaspard est suffisamment expressif pour permettre la génération complète du SoC à partir du modèle. La figure 3.27 synthétise les relations entre les différents paquetages de Gaspard, notons que chacun correspond à une étape différente de la modélisation de SoC. Pour nos travaux, le but était de pouvoir générer une simulation à haut niveau d'abstraction. Avant de présenter la génération à l'aide des transformations de modèles, nous allons introduire dans le chapitre suivant le niveau d'abstraction que nous visons.

Chapitre 4

Simulation à haut niveau d'abstraction

4.1 Simulation au motif près **90**
4.1.1 Spécificités des systèmes simulés 90
4.1.2 Bases de la simulation au motif près 93
4.1.3 Avantages et inconvénients de l'approche 95
4.2 Un modèle d'exécution du modèle de calcul Gaspard **96**
4.2.1 Projections sur différents modèles d'exécution 96
4.2.2 Flux de tableaux 99
4.2.3 Implémentation des *tilers* 101
4.2.4 Synchronisation des tâches 102
4.2.5 Ordonnancement sur un seul processeur 103
4.2.6 Ordonnancement dynamique 104
4.2.7 Synthèse et conclusion 106
4.3 Implémentation de la simulation en SystemC **107**
4.3.1 Simulation en SystemC 107
4.3.2 Un ordonnanceur intégré au simulateur 108
4.3.3 Le mécanisme de synchronisation 113
4.3.4 Lecture et écriture des données 116
4.3.5 Estimation du temps d'exécution 117
4.3.6 Synthèse 119
4.4 Conclusion **120**

Dans la conception des SoC, la validation est une étape prépondérante, étant donné qu'il est très difficile de corriger des erreurs après la fonte de la puce. Les approches formelles peuvent être utilisées pour la validation, mais il est difficile de représenter entièrement le système avec suffisamment de précision et de surcroît le temps de calcul de la vérification peut s'avérer prohibitif. Une alternative courante est alors d'effectuer la validation par simulation. Il existe plusieurs niveaux d'abstraction pour la simulation. Chaque niveau offre un compromis différent entre vitesse de simulation et précision des résultats. Comme nous l'avons vu dans la section 2.2, tout le monde s'accordera à distinguer au moins quatre niveaux de co-simulation : RTL (à partir duquel le SoC complet peut être créé), CABA (précis au cycle et au bit), PVT (où les échanges entre composants sont découpés en transactions), et CPT (où seul le comportement fonctionnel est simulé, en découpant les tâches selon l'association). Ce dernier niveau ne permet pas de simuler le matériel. Pour les trois premiers niveaux, alors

que le matériel est simulé à des précisions différentes, le logiciel est toujours exécuté à la précision la plus fine, en langage machine de la cible.

Une manière d'abstraire l'application est de détecter les *phases* du programme [87], de ne simuler qu'une instance de chaque phase et d'en déduire des résultats pour l'exécution totale. C'est ce que proposent des outils comme SimPoint [51]. L'inconvénient de cette méthode est la difficulté à déterminer précisément toutes les différentes phases de l'application. Ici, nous proposons d'abstraire l'application *par construction*, en ne générant que la partie liée au comportement significatif de l'application, à savoir : les accès mémoires.

L'élévation du niveau d'abstraction de la simulation tire partie de deux points importants du contexte de la thèse. Premièrement, l'usage de modèles à haut niveau d'abstraction pour représenter l'application, indépendant d'une cible donnée, autorise la génération d'un code logiciel directement optimisé pour la simulation. Deuxièmement, Gaspard ne cible que les systèmes de traitement de signal intensif, qui exhibent des propriétés particulières. Le but de ce niveau d'abstraction est de permettre la prise de décision sur l'association et l'architecture le plus tôt possible lors du développement. Le résultat d'une simulation devra donc permettre d'observer le temps d'exécution, le mouvement des données sur les différents réseaux d'interconnexion et spécialement les conflits d'accès, et éventuellement le résultat du calcul des tâches.

Dans ce chapitre, après avoir détaillé ce nouveau niveau d'abstraction, nous présenterons un modèle d'exécution des applications Gaspard et son implémentation à ce niveau d'abstraction.

4.1 Simulation au motif près

Nous introduisons ici la simulation *au motif près* (PA, pour *pattern accurate* en anglais). L'idée sous-jacente est de réaliser une co-simulation logiciel-matériel qui tire partie du fait que le SoC est décrit à un niveau d'abstraction très élevé. Au lieu de simuler instruction par instruction l'application, un code spécifique à la simulation est généré depuis le modèle. Outre le fait que ce code est compilé pour le processeur hôte, ce qui permet déjà une large amélioration de la performance, seules les parties significatives pour les résultats de la simulation sont complètement simulées. En particulier, les accès mémoires aux *motifs des tableaux* sont représentés fidèlement, tandis que les accès aux instructions, les mécanismes d'ordonnancement entre les tâches et même les traitements de données effectués par les tâches élémentaires sont omis ou simplifiés. Nous allons tout d'abord exposer les spécificités de l'environnement qui permettent l'abstraction supplémentaire de cette simulation puis nous dépeindrons les bases de la simulation et enfin nous discuterons des avantages et des inconvénients de cette approche.

4.1.1 Spécificités des systèmes simulés

Le but est de produire une co-simulation à partir d'un modèle de SoC conforme au métamodèle Gaspard. Notre proposition repose sur certaines spécificités des modèles développés dans Gaspard. Afin de clairement définir les limites et les hypothèses sur lesquelles repose la simulation nous allons parcourir les différentes propriétés d'un modèle et de l'environnement et mettre en avant leurs incidences en terme de simulation.

4.1.1.1 Traitement de signal systématique

L'environnement de développement Gaspard vise la modélisation de SoC pour le traitement de signal intensif et même plus spécifiquement pour le traitement de signal systématique. Dans ce type d'application, le code est utilisé pour effectuer le même traitement répétitivement sur des données différentes. Le même code est donc utilisé pour un très grand nombre de données, contrairement à des applications plus usuelles où l'on pourrait trouver le même ordre de grandeur d'instructions que de données traitées. Notons au passage que c'est une caractéristique commune avec les applications de calcul scientifique. À l'aide d'une simulation CABA nous avons observé les communications sur le bus d'un SoC quadri-processeur ayant les caches de faible taille et exécutant un encodeur H263. Sur cette application relativement représentative du traitement de signal systématique seuls 0,7% des transferts sur le bus correspondent aux instructions. Dans ces conditions, nous nous permettons de définir l'hypothèse que dans le système à simuler, les transferts d'instructions sont négligeables par rapport aux transferts de données.

Le traitement systématique, un sous-ensemble du traitement intensif, a pour spécificité d'effectuer toujours les mêmes calculs, quelles que soient les données traitées. Peu importe les entrées, les tâches sont toujours appelées dans le même ordre et lisent les mêmes tableaux dans le même ordre. Dans son état actuel, le modèle d'application Gaspard ne permet que de spécifier un graphe statique des flux de données, un modèle respecte donc forcément le principe du traitement systématique. Néanmoins il serait possible pour une tâche élémentaire (ou plutôt l'IP sur lequel elle est déployée) de ne pas suivre le principe du traitement systématique. Par exemple, le calcul pourrait être plus rapide pour certaines valeurs d'entrée que pour d'autres. Cela n'a cependant qu'un effet limité sur l'exécution globale puisqu'il ne peut pas influencer ni l'ordre d'exécution ni les accès mémoires. Nous posons donc l'hypothèse que l'exécution du SoC est la même quelque soit la valeur des données traitées.

4.1.1.2 Modélisation à haut niveau

Dans la co-simulation usuelle le logiciel qui est exécuté sur les processeurs est donné au simulateur soit sous la forme de code binaire, déjà compilé et lié, soit sous forme de sources en langages bas niveau tels que C. Une fois le logiciel sous cette forme, il n'est pas possible (tout du moins *irréaliste*) d'obtenir une version simplifiée de l'exécution ni de retrouver quel accès en lecture ou en écriture correspond à quelle répétition de quelle tâche. Dans l'environnement Gaspard, l'application est décrite explicitement en terme de transfert de donnée. Plus exactement, il y a une claire séparation entre la phase de lecture ou d'écriture d'un motif de travail dans un tableau et le calcul lui-même effectué dans la tâche élémentaire. Le calcul d'adresse, en fonction de l'indice de répétition, est explicite.

Pour la simulation au motif près, nous appliquons l'hypothèse que toutes les tâches élémentaires sont déployées sur des implémentations abstraites qui, en plus d'être compatibles avec le processeur cible, permettent aussi l'exécution de la fonctionnalité sur le processeur hôte de la simulation. Pour vérifier cette hypothèse, l'implémentation abstraite doit contenir une implémentation dédiée à chacun des types de processeurs. Alternativement, elle doit posséder au moins une implémentation d'IP codée dans un langage compilable indépendamment du type de processeur, par exemple écrite en C ou en C++. Contrairement à des versions écrites en assembleur, les versions dans un langage suffisamment élevé tel que le C peuvent

être compilés pour n'importe quel type de processeur[1].

Nous faisons une deuxième hypothèse que, lors de l'exécution sur le SoC, les données locales à la fonction de l'IP sont de tailles suffisamment restreintes pour qu'elles restent entièrement dans la mémoire locale du processeur (les caches). Autrement dit, il faut que tous les accès mémoire externes au processeur soient décrits par les *Tilers* de l'application. Dans l'optique que les tâches élémentaires sont des *briques de bases*, cela n'est pas très contraignant puisque les calculs sont directs et donc ne nécessitent que peu de stockage temporaire. Cette contrainte peut être difficile à vérifier parce qu'elle dépend de l'association de l'application sur l'architecture : selon la taille et le type de cache utilisé, les données temporaires peuvent être ou ne pas être transmisses à des mémoires extérieures au processeur.

4.1.1.3 Les résultats de la simulation

L'augmentation du niveau d'abstraction vise avant tout à réduire le temps de simulation. Cela permet d'exécuter un plus grand nombre de simulations et/ou d'obtenir plus rapidement un retour après la modification d'une partie du modèle de SoC. La simulation doit guider l'utilisateur parmi l'ensemble des possibilités de conception. Il y a donc intérêt à simuler un large éventail de possibilités et à les comparer afin de trouver le meilleur choix. Cette exploration de l'espace de conception peut faire varier les modèles d'architecture et d'association mais aussi celui de l'application. L'application reste fonctionnellement la même, puisque c'est pour celle-ci que le SoC est conçu, mais les algorithmes sur lesquels elle se base peuvent changer. Donc, avant tout, la simulation doit produire des résultats concernant les propriétés non-fonctionnelles du SoC telles que le temps d'exécution, la consommation énergétique, dissipation thermique, etc. Pour une analyse fine, ces résultats doivent pouvoir aussi être décomposés par composant et le long du temps.

La simulation doit aussi aider à trouver les goulots d'étranglement, les parties de l'architecture qui ralentissent le reste du SoC parce qu'elles sont trop chargées. En particulier, il faut pouvoir observer la quantité de données échangées sur les réseaux d'interconnexion et savoir à quels moments apparaissent des conflits d'accès (la tentative d'accès simultanés au même chemin de donnée). Ce type d'information permet d'ajuster l'association, surtout vis-à-vis de la localité des données comparée à celle des tâches.

Le résultat fonctionnel, les valeurs de sortie obtenues en fonction des entrées, est moins important. Non pas qu'il ne soit pas primordial que l'application effectue les bons calculs, mais une vérification purement fonctionnelle peut être faite en amont, uniquement à partir du modèle d'application. La simulation de l'application seule peut être exécutée à la fois plus rapidement et plus tôt au cours du développement, donc au moment de la co-simulation logiciel-matériel cet aspect est déjà validé et moins important. Il y a néanmoins certaines motivations pour vérifier le côté fonctionnel telles que la détection d'erreurs dans les composants matériels, la validation de code dépendant du temps (par exemple le code des contrôleurs de périphérique d'entrées/sorties), ou la vérification des outils de compilation (qui évoluent souvent afin de suivre les avancées de l'industrie).

Enfin, notons que le but de la simulation est souvent de savoir qu'elle est l'association ou l'architecture *la plus* efficace parmi un panel de possibilités. Autrement dit, l'information

[1]De manière générale, la portabilité d'un programme complet entre différentes cibles de compilation est difficile même lorsque le langage est de haut niveau. Néanmoins la portabilité est pratiquement parfaite pour une fonction dont toutes les entrée et sorties sont passées en arguments et qui utilise le langage pour décrire des calculs mathématiques, comme c'est le cas des tâches élémentaires.

nécessaire n'a pas besoin d'être absolue, « telle association met tant de temps à l'exécution », mais elle doit plutôt être relative, « telle association est plus rapide que telle autre ». L'attente au niveau de la précision des résultats est différente. Par exemple, pour effectuer une comparaison il vaut mieux que toutes les consommations d'énergie mesurées soient toutes optimistes de 50% plutôt que les résultats sur certaines configurations soient pessimistes de 10% et sur d'autres configurations soient optimistes de 10%. Il peut donc être préférable d'avoir une relativement grande imprécision sur les valeurs mais selon une fonction croissante que d'avoir une faible imprécision avec une distribution aléatoire.

4.1.2 Bases de la simulation au motif près

Notre proposition se base sur le principe de l'exécution logicielle abstraite, en contraste de l'exécution à l'instruction près de la co-simulation usuelle. Dans le modèle applicatif Gaspard, chaque tâche prend en entrée des motifs d'éléments et génère d'autres motifs en sortie, ces motifs sont lus ou écrits dans des tableaux selon les informations des *Tilers*. La partie logicielle est simulée avec la lecture ou l'écriture de ces motifs comme grain le plus fin. Cette granularité correspond également à l'exécution d'une répétition d'une tâche élémentaire. Le calcul d'adresse étant explicite dans le modèle (il est spécifié par les *Tilers*), il possible de traduire chaque accès à un motif par un enchaînement de requêtes à la mémoire.

L'ordonnancement et la synchronisation des tâches se font directement dans le simulateur. Selon le graphe de dépendance de donnée spécifié dans le modèle d'application, le composant processeur de la simulation est chargé d'exécuter les répétitions de chaque tâche dans un ordre correct. La synchronisation des tâches, assurant la lecture d'un tableau uniquement après l'écriture complète de celui-ci, est aussi gérée par le simulateur, y compris entre plusieurs processeurs. Alors que dans la réalité la synchronisation repose sur le transfert de donnée, à ce niveau de simulation, la quantité que cela représente sur les réseaux d'interconnexion face aux données est considérée comme négligeable.

L'exécution des tâches élémentaires peut être traitée de deux manières différentes. Premièrement, il est possible d'exécuter la tâche directement sur le processeur hôte (le processeur qui exécute la simulation). Pour chaque tâche, l'IP sur lequel elle est déployée est compilé non pas pour le processeur cible mais pour le processeur hôte. Deuxièmement, il est possible de ne pas exécuter du tout les tâches élémentaires ! Si les résultats fonctionnels ne sont pas nécessaires et étant donné que l'ordre et la quantité des calculs ne varient pas en fonction des valeurs, alors seuls la structure de l'application et les accès mémoires sont suffisants pour représenter l'exécution de l'application. Le temps d'exécution d'un appel à cet IP est considéré constant et connu au préalable par le simulateur. Ce temps est soit mesuré automatiquement à l'aide d'une simulation bas niveau ou fournit pas l'utilisateur. De plus, le fait que les manipulations de données internes à la tâche restent circonscrites au cache du processeur assure que la simulation ne sous-estime pas les accès mémoire.

Concernant la simulation de la partie matérielle, nous proposons de garder la même abstraction que celle de la simulation PVT (le plus haut niveau actuel couramment utilisé) : la granularité est au niveau de la transaction. La différence par rapport à une simulation PVT se situe au niveau des composants processeur. Alors qu'en PVT les processeurs sont des simulateurs de jeu d'instruction (ISS), dans la simulation au motif près le composant processeur est pratiquement indépendant de l'IP qu'il représente et il contient la structure de l'application et les calculs d'adresse. Notons que l'utilisateur a le choix de garder les caches dans la simulation ou bien de les simuler comme un tout avec le processeur.

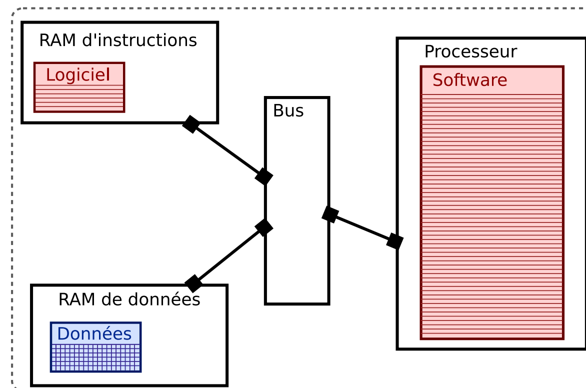


FIG. 4.1: Schéma d'une co-simulation au niveau PVT. Les instructions et les données du logiciel n'ont pour le simulateur aucun sens particulier. Le composant processeur interprète les instructions une par une.

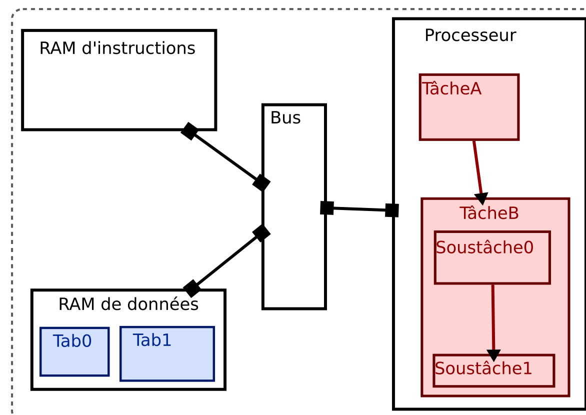


FIG. 4.2: Schéma d'une co-simulation au niveau PA. Le simulateur est capable de prendre en compte l'application ainsi que les tableaux de donnée comme tels. Les observations que l'on peut tirer de la simulation sont plus complètes vis-à-vis de l'application ce qui permet au développeur de mieux interpréter les résultats.

Les figures 4.1 et 4.2 schématisent les différences entre une simulation PVT et PA. Dans le premier type de simulation, les informations concernant le logiciel sont pratiquement complètement effacées. Les tâches et les tableaux ne sont vus que comme des ensembles d'instructions et de données. Dans la simulation au motif près, les composants matériels restent identiques à l'exception des processeurs qui sont remplacés par une simulation de l'application. L'ordonnancement des tâches ainsi que la synchronisation entre-elles sont gérés par le simulateur. Comme dans une simulation au niveau PVT, les accès en écriture et en lecture des données sont entièrement réalisés, par contre, étant donné que l'information de haut niveau est encore présente, ils peuvent être en plus associés à un motif précis d'un tableau requis par une répétition d'une tâche. En outre, si l'utilisateur n'a pas besoin du résultat fonctionnel, la simulation PA peut désactiver l'appel aux IP de l'application, ne

simulant alors que leur temps d'exécution et les accès mémoires requis.

4.1.3 Avantages et inconvénients de l'approche

L'avantage principal de ce niveau plus élevé d'abstraction est l'accélération de la simulation, ce que nous présenterons plus en détail dans le chapitre 6. Par rapport à PVT, tous les composants s'exécutent à la même vitesse sauf les composants processeur qui s'exécutent beaucoup plus rapidement. Cela vient en partie du fait que l'application est directement compilée pour le processeur hôte et donc il n'y a pas à payer la lourde pénalité en terme de calculs que représente la traduction d'instruction. L'accélération provient aussi du fait que les calculs des tâches élémentaires peuvent être entièrement éliminés. Dans le cadre du développement de SoC multiprocesseur, où les processeurs représentent une très grande partie du SoC, cette amélioration de la vitesse de simulation des processeurs est particulièrement influente sur le temps de simulation total. Par exemple, nous verrons par la suite que la simulation logicielle est accélérée au point de représenter un coût négligeable en durée par rapport au temps de simulation du matériel.

Un second avantage très important est le rapprochement facile entre les observations auxquelles la simulation donne accès et le modèle de SoC. Grâce au fait que la lecture des motifs est explicite, il est possible d'associer les transactions échangées sur les réseaux d'interconnexion à un motif et une répétition particuliers d'une tâche. Avec ces informations on peut déterminer précisément quelle tâche ou quel tableau est impliqué dans l'apparition d'un goulot d'étranglement. Ainsi ces observations facilitent la correction et la validation de l'association.

Lorsque la simulation exécute les tâches élémentaires, permettant d'obtenir un résultat fonctionnel, il est possible d'utiliser les outils conventionnels de débogage sur le code des IP. Dans une simulation classique, les instructions du logiciel sont interprétées par l'ISS, cela rend difficile l'intégration à la simulation d'un débogueur complet, qui en outre est spécifique à chaque IP processeur présent dans la simulation. Comme mis en avant par Honda *et al.* [54], lorsque la simulation exécute le code directement sur le processeur hôte, alors le concepteur a la possibilité d'utiliser tous les outils de débogage disponibles pour sa station de travail sur ce code. En plus de jouir d'une mise en place plus simple, cette solution apporte de l'efficacité car le développeur utilise des outils dont il a l'habitude et qui généralement sont plus riches en fonctionnalités et convivialité.

À ce niveau de simulation, il n'y a pas d'IP de processeur nécessaires. Ils sont remplacés par le code abstrait de l'application. Cela permet de réaliser une co-simulation logiciel-matériel même si tous les IP de processeur ne sont pas disponibles. En particulier, cela peut permettre d'effectuer les premières co-simulations plus tôt dans le processus de développement, avant que les IP de processeur ne soient acquis. Ainsi le temps de développement peut être réduit.

Un niveau de simulation correspond toujours à un compromis entre performance et précision. Ainsi, en contre-partie des avantages que nous venons de parcourir, la simulation a une précision réduite par rapport aux simulations de plus bas niveau. En particulier vis-à-vis de l'estimation temporelle, la simulation instruction par instruction assure une bien meilleure précision face à la somme des temps d'exécution des tâches élémentaires et des calculs d'adresse. Nous reviendrons plus en détail sur l'aspect de la précision dans le chapitre 6 concernant la validation expérimentale.

Après avoir abordé les motivations et les principes de fonctionnement de la simulation, dans la section suivante nous allons porter notre attention sur la sémantique exacte que

nous allons associer à la spécification du SoC. L'implémentation d'un modèle d'architecture matérielle Gaspard est complètement définie (tout du moins dans une simulation à haut niveau) : les informations précisées dans le modèle sont suffisantes pour spécifier la génération de code sans avoir de choix techniques à faire. Par contre, le modèle de calcul du modèle d'application Gaspard, bien qu'il soit déterministe, n'est pas lié à un modèle d'exécution particulier, il laisse un certain nombre de libertés sur la manière d'implémenter un modèle. Dans la section suivante, nous allons présenter différentes implémentations possibles, puis nous en proposerons une qui s'adapte bien aux MPSoC.

4.2 Un modèle d'exécution du modèle de calcul Gaspard

Dans la section 2.3, nous avons détaillé le modèle de calcul (MoC) des applications Gaspard. De manière intuitive, on peut le résumer ainsi : chaque répétition d'une tâche est exécutée une fois tous les motifs en entrée disponibles ; les motifs de sorties produits sont transmis à toutes les tâches dépendantes de ceux-ci. Ce MoC, indépendant de l'architecture ou de l'association, définit une sémantique suffisamment précise pour être déterministe : en respectant les contraintes posées, pour une application donnée, toute exécution mènera au même résultat.

Dans le but de générer une simulation la plus réaliste possible du SoC il faut que la manière précise avec laquelle les tâches vont être exécutées soit identique entre la simulation et la réalisation finale du SoC. Ici, notre but est donc de proposer une spécification précise du modèle d'*exécution* des applications Gaspard. Avant tout, il définit la manière selon laquelle sera implémentée l'application dans la réalisation finale. Il doit donc convenir à une exécution sur MPSoC, en particulier il faut tenir compte des faibles ressources mémoires et du fait que la couche du système d'exploitation est très légère, voire inexistante. Ce modèle d'exécution sera également celui suivi pour la compilation de l'application vers les cibles permettant la simulation du MPSoC.

Après avoir examiné différents modèles d'exécution qui ont déjà été proposés nous définirons précisément celui que nous suivrons. Celui ci se base sur l'usage de FIFO avec plusieurs producteurs et plusieurs consommateurs pour la transmission des tableaux entre tâches. Les accès mémoires imbriqués à différents niveaux hiérarchiques sont fusionnés afin de minimiser l'usage des mémoires temporaires. Afin d'assurer la validité des tableaux lus, les tâches distribuées sur plusieurs processeurs sont synchronisées, tandis que celles exécutées sur un seul processeur sont ordonnancées séquentiellement. Enfin, dans le but de permettre le recouvrement d'exécution de tâches, les tâches distribuées sont ordonnancées dynamiquement.

4.2.1 Projections sur différents modèles d'exécution

Il existe un grand nombre de possibilités pour projeter le modèle d'application sur un modèle d'exécution. On peut commencer par citer deux projections « naïves » sur ces types de modèle d'exécution :

- **Exécution séquentielle**, à chaque niveau de hiérarchie les tâches sont exécutées les unes après les autres selon un ordre respectant des dépendances. Pour chaque tâche, les répétitions sont exécutées itérativement. Cependant ce modèle d'exécution ne permet pas le parallélisme, ce qui lui soustrait tout de suite un usage réel.

– **Exécution parallèle intégrale**, chaque répétition d'une tâche étant indépendante, toutes les répétitions d'une tâche sont exécutées en parallèle. Chaque tâche peut être exécutée dès que toutes les tâches dont elle dépend sont terminées. Ce modèle d'exécution introduit en revanche trop de parallélisme pour les architectures qui ne possèdent pas assez d'unités d'exécution.

4.2.1.1 Projection sur les KPN

Dans l'équipe, Abdelkader Amar a proposé [14] une projection vers le modèle de calcul des réseaux de processus de Kahn (KPN, pour *Kanh Process Network*) définis par Kahn [60]. Ce modèle de calcul possède déjà de nombreuses projections vers des modèles d'exécution. Les KPN permettent facilement de représenter des applications concurrentes. Chaque processus communique avec les autres à travers des FIFO unidirectionnelles. Chacune des FIFO a un seul producteur et un seul consommateur. De plus, si la lecture d'une FIFO peut être bloquante (lorsque celle-ci est vide), l'écriture ne l'est jamais : les FIFO sont infinies. Les KPN sont déterministes, les résultats ne dépendent que du réseau et pas de l'ordonnancement.

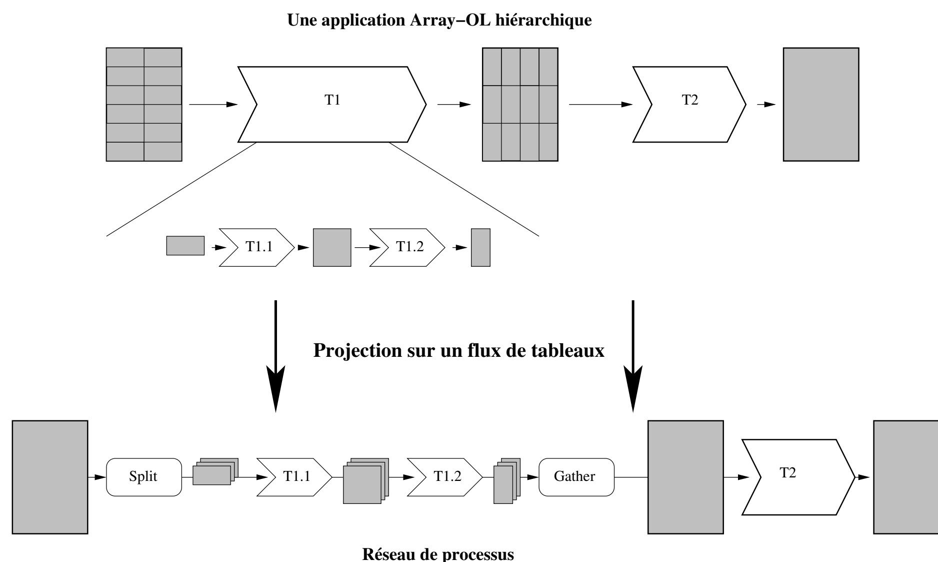


FIG. 4.3: Projection d'une application Gaspard sur KPN. Les carrés gris représentent les tableaux transmis entre les tâches/processus représentés par des formes blanches.

Une implémentation des KPN ne peut cependant pas être exacte puisqu'il faut forcément utiliser des FIFO bornées. Parks [82] a proposé un ordonnancement capable de supporter les FIFO bornées tout en assurant que les applications infinies ne soient jamais arrêtées à condition que soit possible, c'est-à-dire à condition qu'aucun processus ne génère de données plus rapidement qu'un autre processus ne les lit. Le modèle d'exécution des KPN utilisé par Amar est distribué et repose sur CORBA.

Pour projeter le modèle de calcul des applications Gaspard sur les KPN, chaque tâche est implémentée par un processus. Ainsi, chaque processus se charge de l'exécution de toutes les répétitions d'une tâche et des *tilers* associés. Cela implique que la granularité de la communication se fait au tableau (et non pas au motif près). La figure 4.3 met en avant le

traitement de la hiérarchie, qui n'existe pas dans les KPN. Les applications sont mises à plat entièrement jusqu'aux tâches élémentaires. Deux processus supplémentaires sont ajoutés à chaque aplatissement de tâche composée, ils correspondent aux *tilers* d'entrée et de sortie et sont placés avant et après l'enchaînement des sous-tâches, comme indiqué dans la figure (les processus *Split* et *Gather*).

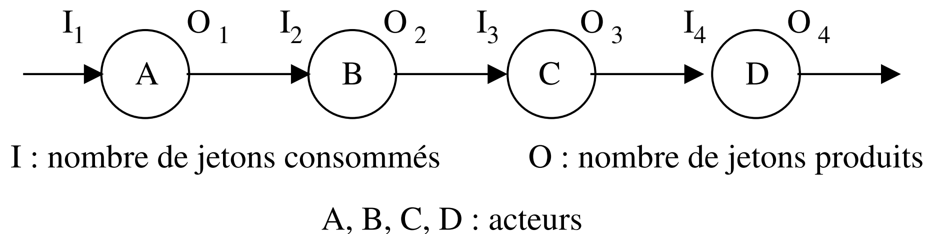


FIG. 4.4: Un graphe d'application SDF.

4.2.1.2 Projection sur SDF

Philippe Dumont, lors de ses travaux de thèse dans l'équipe, a proposé dans sa thèse [36, p. 103] de projeter les applications Gaspard sur le modèle de calcul proposé par Lee et dénommé SDF [65] (Synchronous DataFlow). SDF est un langage à flot de données synchrones, dont une implémentation est disponible dans l'environnement PTOLEMY. Une application SDF est représentée par un graphe orienté acyclique dont les nœuds représentent les calculs et les arêtes représentent les données produites et consommées par ces nœuds. Un exemple de graphe d'application est présenté dans la figure 4.4. Chaque nœud, appelé *acteur* produit et consomme un nombre fixe de données, appelées *jetons*. Le graphe est hiérarchique, c'est-à-dire que chaque acteur peut être lui-même une application SDF.

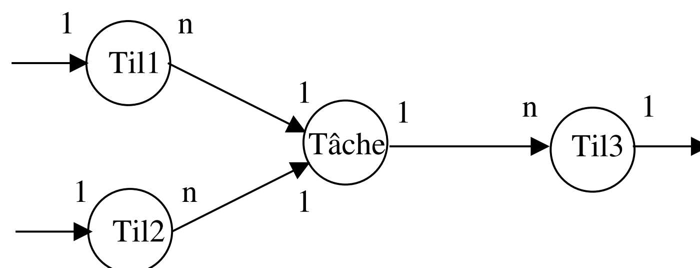


FIG. 4.5: Schéma de projection d'une tâche Gaspard sur un graphe d'application SDF. La tâche consomme deux motifs et en produit un autre.

L'intérêt principal de SDF est que les applications peuvent être ordonnancées statiquement et consomment une quantité de mémoire bornée. La projection d'une application Gaspard vers un modèle SDF se fait selon deux points. Premièrement, le graphe des tâches composées, qui représentent le parallélisme de données, est directement transformé en un graphe SDF avec chaque instance représentée par un nœud et chaque connexion représentée par une arête avec un seul jeton en entrée et en sortie, puisque chaque tableau n'est produit ou

consommé qu'une seule fois. Deuxièmement, chaque tâche contenant une instance répétée n fois est transformée en un graphe avec l'instance représentée par un nœud consommant et produisant un jeton par arête. Comme on peut le voir sur le schéma de la figure 4.5, chaque *tiler* d'entrée est transformé en un nœud consommant un jeton (le tableau) produisant n jetons qui sont transmis au nœud de l'instance. De même, chaque *tiler* de sortie est transformé en un nœud consommant n jetons originaires du nœud de l'instance et produisant un jeton, le tableau de sortie. Ainsi, le nœud de l'instance est exécuté n fois pour chaque tableau produit. Remarquons que les *tilers* ne peuvent pas être représentés par des arêtes comme dans le modèle d'application Gaspard car ils effectuent des calculs.

L'implémentation a le désavantage de ne pas permettre l'exécution concurrente de plusieurs répétitions d'une même tâche, car la façon de représenter les *tilers* par un nœud force à séquentialiser l'exécution de l'instance répétée. L'exécution des différentes instances d'une tâche composée peut néanmoins être pipelinée (un nœud producteur et un nœud consommateur peuvent être exécutés parallèlement sur des données qui correspondent à des temps de calcul différents).

Jusqu'à présent les projections que nous avons vues ne prennent en compte explicitement que l'application. L'implémentation peut, selon les cas, plus ou moins bien s'adapter à la plate-forme d'exécution. En nous inspirant de ces projections, nous allons proposer un modèle d'exécution qui puisse suivre les contraintes posées par le concepteur concernant l'association de l'application sur l'architecture matérielle. Notre objectif est également d'obtenir une implémentation efficace, aussi bien en temps d'exécution qu'en terme de consommation mémoire.

4.2.2 Flux de tableaux

Dans le modèle de calcul Gaspard, les dépendances entre les répétitions de tâche correspondent à la production et la consommation des motifs sur lesquels la tâche travaille. Cela représente la granularité la plus fine possible du flux de données de l'implémentation. C'est avec cette granularité que le maximum de parallélisme peut être obtenu : dès que sont disponibles tous les motifs d'entrée d'un élément de l'ensemble des répétitions, la répétition est exécutée. Néanmoins cette granularité engendre un certain nombre de difficultés dans l'implémentation. Tout d'abord, les motifs produits par une tâche ne sont pas forcément les mêmes que ceux consommés par la tâche suivante. Il faut dans ce cas fusionner un ensemble de répétitions de la tâche productrice et un ensemble de répétitions de la tâche consommatrice. S'il est toujours possible de trouver de tels ensembles, ils peuvent être parfois aussi grands que l'ensemble des répétitions, autrement dit, le motif intermédiaire est de la même taille que le tableau. C'est en particulier ce qui arrive lors d'un *corner turn* où le tableau est accédé en colonne d'un côté et en ligne de l'autre côté.

Une autre difficulté qu'engendre l'usage d'un flux de motifs est qu'au niveau de l'implémentation un grand nombre de synchronisations est nécessaire pour indiquer la production et la consommation de chaque motif. Cela crée un surcoût important en terme de communication et de calcul à l'exécution.

Afin de réduire la complexité de l'implémentation et d'éviter les surcoûts liés à la trop fine granularité des synchronisations, nous proposons un modèle d'exécution des applications Gaspard selon un flux de tableaux. L'ensemble des répétitions d'une tâche attend que tous les tableaux d'entrée soient disponibles avant de s'exécuter.

L'utilisation des tableaux comme granularité du flux interdit l'usage de tableaux infinis

entre deux tâches, cela reviendrait à attendre indéfiniment le calcul de toutes les répétitions de la première tâche ! Seul le niveau de hiérarchie le plus haut peut contenir une répétition infinie, avec les tâches communiquant uniquement à travers des tableaux de taille bornée. Néanmoins cette contrainte n'est pas très forte car Philippe Dumont a démontré *par construction* dans sa thèse [36, p. 101] qu'il est toujours possible de transformer automatiquement toute application en une application dont la forme respecte cette contrainte.

Le passage de tableaux entre deux tâches peut s'effectuer à travers des FIFO. Si l'utilisateur souhaite minimiser l'usage de la mémoire, les FIFO peuvent être bornées à un seul tableau : après qu'une tâche ait produit les tableaux de sortie, elle ne peut être exécutée de nouveau qu'une fois que toutes les tâches utilisant ces tableaux ont elles-mêmes été exécutées. L'augmentation de la taille des FIFO à deux tableaux permet dans un cas où plus de deux tâches se suivent d'améliorer le parallélisme en autorisant chaque tâche du pipeline à s'exécuter en même temps (plutôt qu'une sur deux lorsque la FIFO ne contient qu'un seul tableau). C'est ce qui est usuellement appelé le mécanisme de *flip-flop*. Dans le cadre du traitement de signal systématique, il est rarement nécessaire d'utiliser des FIFO plus grandes.

Notons aussi que l'implémentation est optimisée lorsque plusieurs tâches utilisent le même tableau comme entrée, comme le tableau produit par la tâche B dans la figure 4.6. Chaque tableau produit n'est écrit qu'une seule fois en mémoire, quel que soit le nombre de consommateurs. Toutes les tâches consommatrices de ce tableau lisent donc dans le même emplacement mémoire. Lorsqu'une tâche produit un tableau et plusieurs tâches le lisent, il y a une FIFO par connexion, qui sert de synchronisation mais un seul espace mémoire utilisé pour stocker les valeurs du tableau. Cet espace mémoire est libre uniquement après que toutes les FIFO aient été lues.

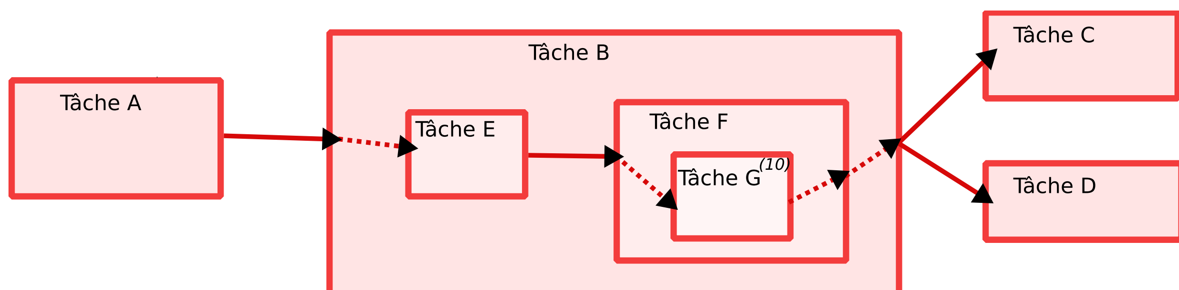


FIG. 4.6: Exemple d'application Gaspard. Les tâches sont représentées par les rectangles, la transmission de tableau entre deux tâches est représentée par une flèche. Le passage de tableau entre une tâche et une sous-tâche, correspondant à un *tiler*, est représenté par une flèche en pointillé.

Concernant l'allocation mémoire, elle est faite statiquement selon les contraintes posées par le modèle d'association. Chaque tableau se voit attribuer un espace mémoire propre correspondant à sa taille totale. De manière à pouvoir traiter l'exécution concurrente de plusieurs répétitions d'une même tâche, il faut que le tableau soit multiplié autant de fois qu'il peut y avoir d'usages concurrents. Sachant que les tâches Gaspard ne peuvent pas être récursives[2] et que dans ce modèle d'exécution il n'y a pas de préemption de tâche, le nombre maximum d'usages concurrents est réduit au nombre de processeurs sur lesquels s'exécutent

[2]Car il n'est pas possible d'exprimer de condition d'arrêt.

les tâches qui l'utilisent. Ainsi, l'espace attribué à chaque tableau est multiplié par le nombre de processeurs qui y font un usage indépendant. De plus, lorsque le tableau est distribué sur plusieurs bancs mémoire, l'espace mémoire attribué varie pour chaque banc. Pour un banc donné, l'espace alloué correspond à la quantité d'éléments qui y seront stockés.

4.2.3 Implémentation des *tilers*

La hiérarchie de l'application est conservée dans l'implémentation. À chaque niveau de hiérarchie, il y a des *tilers* qui se chargent chacun de découper un tableau en motifs et de les transmettre au niveau inférieur. Les tableaux correspondent soit à un motif du niveau supérieur, soit à un passage de données entre deux tâches. Dans l'exemple présenté figure 4.6, les passages de données entre deux tâches sont représentés par une flèche en ligne continue tandis que les accès à des motifs du niveau supérieur sont représentés par des flèches en pointillé. Bien sûr, au niveau de hiérarchie le plus haut, tous les tableaux correspondent à des échanges de données entre tâches puisqu'il n'y a pas de niveau supérieur. Ce type de tableau est stocké de la manière que nous venons de voir dans la section précédente. Par contre les tableaux qui correspondent à un motif du niveau supérieur ne sont que des représentations temporaires.

Les *tilers* se chargent de deux choses :

– le calcul d'adresse, qui est plus exactement un changement de coordonnée entre le couple indice de répétition-coordonnée dans le motif et les coordonnées dans le tableau.
– l'accès aux données, qui selon le type de *tiler* consiste en la lecture ou en l'écriture des données entre la mémoire locale du processeur et la mémoire où est situé le tableau.

En raison de ce deuxième point, nous avons décidé d'implémenter les *tilers* d'une tâche toujours exécutés de manière indissociable avec chaque répétition de la tâche. Bien qu'il serait envisageable de paralléliser le calcul d'adresse sur un autre processeur que celui exécutant la tâche elle-même, la recopie en mémoire locale serait complexe, voir impossible selon les architectures. Par conséquent, avant et après chaque appel à une fonction d'un IP, il y a un calcul d'adresse puis un transfert entre les éléments du tableau en mémoire distante et les éléments du motif disponible sur le processeur. Lorsqu'un connecteur simple est utilisé pour passer un tableau à une sous-tâche (la sous-tâche utilise directement le tableau) la sémantique est la même que celle d'un *tiler* mais sans calcul d'adresse. Par conséquent, ces connecteurs sont implémentés de la même manière, exécutés juste avant ou juste après chaque répétition de tâche.

Lorsque le motif est lu depuis un tableau qui correspond lui-même à un motif au niveau supérieur, la création du motif intermédiaire est inutile puisque cela ne serait qu'une recopie du tableau supérieur. Afin d'éviter cette recopie et de réduire l'utilisation de la mémoire, ces motifs intermédiaires ne sont jamais créés. À la place, le calcul d'adresse par le *tiler* le plus bas dans la hiérarchie est modifié pour prendre en compte également le calcul d'adresse du *tiler* supérieur. S'il y a plus d'un niveau de hiérarchie, cela est fait récursivement. L'opération pour fusionner deux *tilers* en un est appelée *court-circuit* et a été mathématiquement validée dans la thèse de Julien Soula [90, p. 42]. L'idée est de fusionner les deux « changements de repère » que sont chaque *tiler* en un seul changement de repère qui utilise à la fois les indices de la répétition du niveau bas et ceux du niveau haut. Cela fonctionne dans tous les cas sauf lorsque le *tiler* du niveau inférieur utilise un modulo. Ce cas, qui dans la pratique est rare, peut être résolu en étendant la notion de modulo des *tilers* pour autoriser des valeurs différentes selon les dimensions ou bien simplement en exécutant séquentiellement le calcul de chaque

tiler. Ainsi, seules les tâches élémentaires sont entourées de *tilers*, les niveaux hiérarchiques supérieurs étant uniquement représentés par le parcours des indices de répétitions.

Concernant le calcul d'adresse à partir des coordonnées dans le tableau supérieur deux cas sont possibles selon que le tableau a été alloué directement sur une mémoire ou bien distribué sur un ensemble de bancs mémoire. Dans le cas le plus simple, lors d'une allocation directe, les coordonnées obtenues par le calcul de *tiler* sont linéarisées[3], multipliées par la taille d'un élément de tableau et sommées à l'adresse de base du tableau en mémoire (qui est calculée depuis une adresse fixe et le numéro de l'élément en cours de traitement dans la FIFO). Dans le cas où le tableau est utilisé à l'intérieur d'une tâche distribuée, l'espace mémoire attribué est multiplié selon le nombre de processeurs. Dans ce cas, l'adresse de base du tableau est calculée en fonction du numéro du processeur sur lequel le *tiler* s'exécute.

Lorsque le tableau est distribué, un calcul préliminaire doit être fait pour connaître l'adresse de base, qui varie selon le banc mémoire qui doit être accéder. Ce calcul est constitué par l'exécution successive des deux *tilers* sur lesquels repose la distribution. Les coordonnées obtenues indiquent le banc mémoire où se trouve l'élément recherché. En fonction du banc, l'adresse de base du tableau est récupérée depuis une table pré-calculée. La linéarisation est effectuée en ne prenant en compte que les éléments alloués sur le banc. Dans le cas, le plus usuel, où une dimension du tableau est distribuée par bloc sur les différents bancs mémoire cette étape de linéarisation est identique à celle d'une allocation directe.

4.2.4 Synchronisation des tâches

Les tâches sont exécutées sur les processeurs spécifiés par l'association. Lorsqu'une tâche est distribuée, les répétitions sont exécutées sur plusieurs processeurs en parallèle. Sur chaque processeur, l'implémentation de la tâche consiste en l'exécution *séquentielle* de chacune des répétitions qui ont été placées sur le processeur. En effet, il n'y a pas d'intérêt à paralléliser les répétitions sur un processeur donné puisque de toute façon un seul fil d'exécution est disponible.

Pour une tâche donnée chaque répétition se charge de produire une partie de chaque tableau de sortie. Si cette tâche est distribuée sur plusieurs processeurs alors, même si dans le modèle d'application une tâche est toujours producteur unique d'un tableau, dans l'implémentation le tableau est produit en partie depuis chacun des processeurs. De même, pour chaque consommateur, même si une FIFO est attribuée par connexion et non pas par tableau, il peut y avoir autant de lecteurs de la FIFO que de processeurs. Par conséquent, des synchronisations sont nécessaires pour pouvoir gérer pour chaque connexion plusieurs producteurs et plusieurs consommateurs. Le fonctionnement des synchronisations est défini ainsi :

- Chaque consommateur attend que l'ensemble des motifs ait été produit par les producteurs qui en ont la charge et qui sont répartis sur les différents processeurs.
- Si une FIFO est pleine, les producteurs se mettent en attente avant même de commencer la lecture des motifs d'enter (parce que l'on ne permet aux tâches de se mettre en attente uniquement en dehors d'un cycle complet de parcours des répétitions).

[3] Par exemple l'adresse de l'élément $[i, j]$ ayant une taille T, contenu dans un tableau de dimension $[M, N]$ dont l'adresse de base en mémoire est B a pour adresse mémoire : $B + (i + Mj)T$.

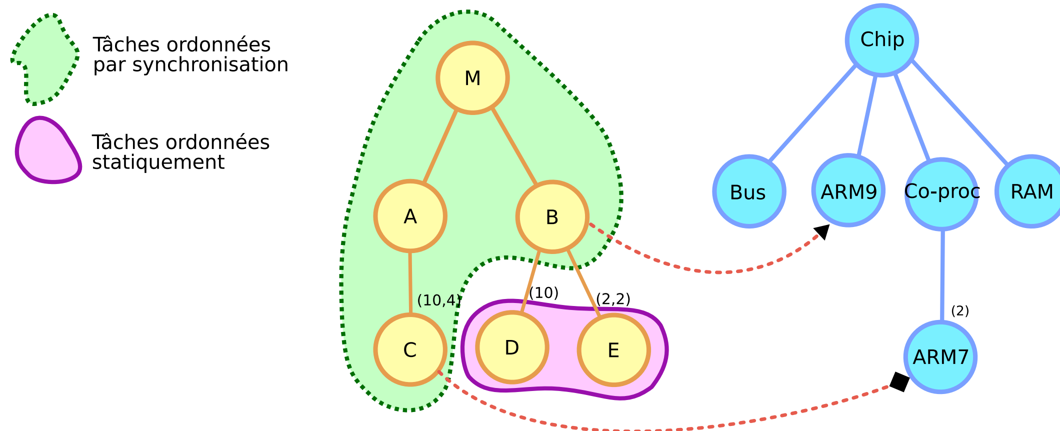


FIG. 4.7: Exemple de différence de l'ordonnancement en fonction de l'association. Dans cette vue compacte arborescente, la tâche B est placée sur l'ARM9 tandis que la tâche C est distribuée sur les deux ARM7. Ces tâches et tous les niveaux supérieurs sont ordonnancés en utilisant la synchronisation. Les niveaux plus bas, c'est-à-dire ici les tâches D et E sont ordonnancées statiquement juste en utilisant l'aspect séquentiel du processeur.

4.2.5 Ordonnancement sur un seul processeur

Cette synchronisation des tâches, nécessaire pour assurer la cohérence des données, n'est utile que lorsque les tâches sont placées sur plusieurs processeurs. En effet, lorsque toutes les tâches sont exécutées sur un seul et même processeur, il est possible de les ordonnancer statiquement selon un ordre respectant les dépendances de données. Ainsi, pour l'optimisation de l'exécution, seuls les plus hauts niveaux de hiérarchie de l'application sont ordonnés grâce aux synchronisations. À partir du niveau où, selon l'association spécifiée par l'utilisateur, un composant contient tous ses sous-composants placés sur le même processeur, toute la sous-hiérarchie est exécutée séquentiellement. Par exemple, dans la figure 4.7, tous les sous-composants de B (les tâches D et E) sont placés sur le même processeur (l'ARM9). Les échanges de données entre ces tâches sont donc forcément entre le même processeur, ce qui permet d'ordonnancer ces tâches statiquement. Les autres tâches, qui peuvent avoir des dépendances entre plusieurs processeurs utilisent le mécanisme de synchronisation. Notons que si C avait eu des sous-composants, ils auraient été ordonnancés statiquement, puisque chaque exécution de C n'est faite que sur un processeur.

Le calcul de l'ordre d'une exécution séquentielle est effectué statiquement, c'est-à-dire à la génération de code ou à la compilation. Il doit assurer qu'un ordre de manière à ce que toutes les entrées d'une tâche aient déjà été produites avant que la tâche ne soit exécutée. Cela peut se faire directement en utilisant les connexions entre tâches comme dépendance. Si un composant est composé, alors il est remplacé par l'exécution de chaque sous-composant, l'ensemble est ordonnancé statiquement de la même manière.

4.2.6 Ordonnancement dynamique

4.2.6.1 Problèmes de recouvrement liés à l'ordonnancement statique

Plusieurs possibilités existent pour traiter sur chaque processeur l'ordonnancement des tâches synchronisées entre plusieurs processeurs. La solution la plus simple est d'effectuer un ordonnancement statique séquentiel identique à celui utilisé pour les tâches exécutées sur un seul processeur. Dans ce cas le graphe de dépendance entre les tâches utilisé pour calculer l'ordre est celui de toute l'application, y compris les tâches qui ne sont pas exécutées sur le processeur. De cette liste ordonnée de tâches, commune à tous les processeurs, on peut produire pour chaque processeur la séquence ordonnée des tâches qui s'exécuteront sur le processeur en question. Avant et après chaque tâche sont placées les synchronisations. Cet ordonnancement permet une exécution correcte de l'application, placée sur l'architecture comme spécifié dans l'association.

Cependant cet ordonnancement est limité en terme de performance. À partir du moment où le temps d'exécution de chaque tâche, ou le temps de chaque accès à la mémoire, ne sont pas connus précisément à l'avance, il n'est pas possible de prévoir l'ordonnancement optimal sur une architecture multiprocesseur. En fonction de l'avancement de l'exécution sur un processeur par rapport aux autres processeurs, la réorganisation de l'ordre des tâches peut accélérer l'exécution globale. Cet ordre spécifique à chaque processeur, voire à une phase de l'exécution, est impossible à prévoir statiquement. En effet, cela nécessite de connaître le temps d'exécution de chaque répétition sur chaque processeur, mais aussi de connaître le temps des communications y compris les moments des éventuels conflits d'accès. Ces informations nécessitent une connaissance très précise de l'architecture. D'autant plus que les données d'entrée elles-mêmes peuvent influencer ces durées.

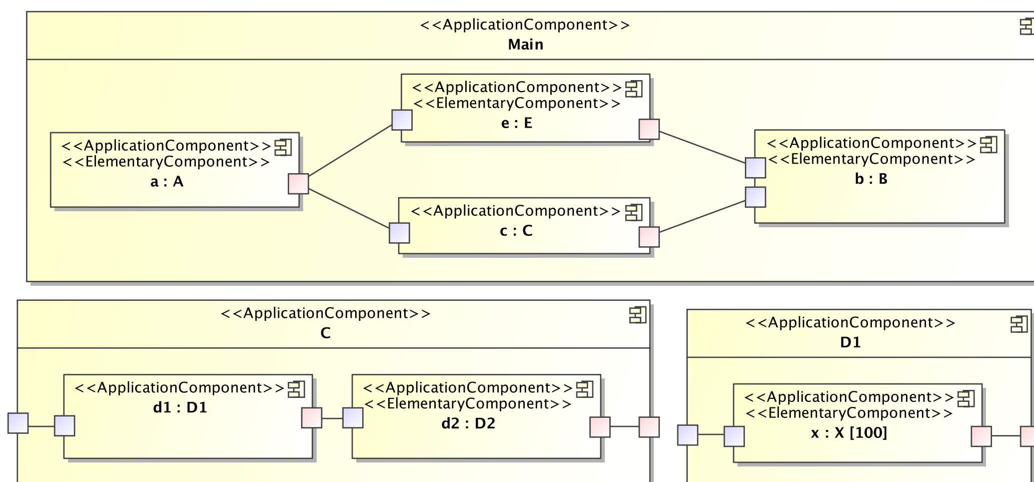


FIG. 4.8: Vue composite d'une application Gaspard. Les tâches *e* et *c* consomme le même tableau produit par *a* et produisent chacune un tableau consommé par *b*. La tâche *c* est composée d'une tâche répétée *d1* et d'une tâche élémentaire *d2*.

Commençons par mettre en avant des difficultés que l'on peut rencontrer avec un ordonnancement statique qui sont très facilement traitées par un ordonnancement dyna-

mique. Observons l'exécution de l'application présentée en figure 4.8. L'application est placée sur une architecture à deux processeurs identiques *P0* et *P1*. Les tâches *a*, *e* et *b* sont placées sur *P0*, tandis que *d1* est distribuée sur les deux processeurs et *d2* est placée sur *P1*. L'ordonnancement statique résultant est comme suit :

P0	P1
a c :d1 :x e b	 c :d1 :x c :d2

Les tâches *c* et *e* étant exécutées séquentiellement, il n'est pas possible d'exécuter *e* en même temps que *d2* parce qu'il est nécessaire d'attendre la fin complète de *c* avant d'exécuter *e*. Pour pallier ce problème avec un ordonnancement statique il faudrait aplatir toute la hiérarchie de tâches et à partir du graphe produire ainsi des synchronisations moins contraignantes. En utilisant un ordonnancement dynamique, il est possible de faire chevaucher l'exécution des tâches *e* et *d2* sans modification des synchronisations :

P0	P1
a c :d1 :x e b	 c :d1 :x c :d2

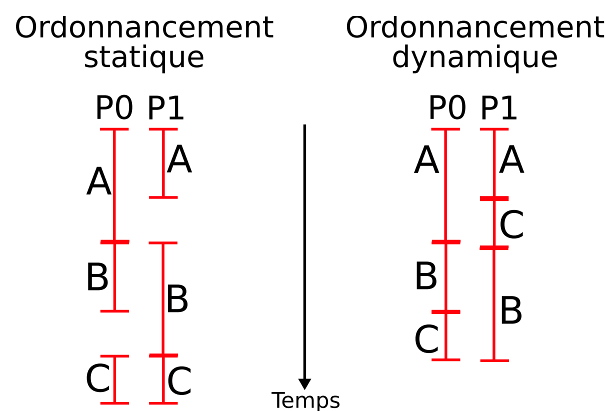


FIG. 4.9: Chronogramme de l'ordonnancement de trois tâches *A*, *B* et *C* sur deux processeurs. *B* est dépendante de *A* et donc doit attendre la fin de son exécution pour être ordonnancée. Sur chaque processeur, une seule tâche ne s'exécute à la fois. En raison du comportement du matériel, le temps d'exécution d'une tâche varie d'un processeur à l'autre.

Ce problème survient lorsque la structure de l'application rend difficile un ordonnancement statique optimal. Dans d'autres cas, l'ordonnanceur statique est purement incapable de donner un ordre optimal. C'est par exemple ce qui peut arriver lors du manque de

connaissances sur le comportement du matériel. La figure 4.9 met en valeur ce cas dans un exemple d'ordonnancement d'une application composée d'une tâche *A* produisant un tableau consommé par une tâche *B* et d'une troisième tâche *C* indépendante. Toutes ces tâches sont parallèles, et peuvent donc être exécutées sur les deux processeurs, *P0* et *P1*, qui sont contenus dans l'architecture matérielle. En raison du fonctionnement de l'architecture (par exemple des conflits d'accès à la mémoire), la tâche *A* est un peu plus rapide à s'exécuter sur *P1* que sur *P0*, tandis qu'à l'inverse *B* est plus rapide sur *P0* que sur *P1*. Alors que l'ordre statique d'exécution des tâches *A-B-C* empêche les processeurs d'être constamment utilisés, l'ordonnancement dynamique permet d'utiliser pleinement la puissance offerte par les processeurs. Dans cet exemple l'ordonnancement dynamique offre ainsi une exécution plus rapide de l'application.

4.2.6.2 Un ordonnancement dynamique coopératif

L'ordonnancement que nous proposons est non-préemptif : il n'interrompt pas une tâche en cours d'exécution pour une autre. Le choix de la tâche suivante à exécuter se fait uniquement après que l'ensemble des répétitions dont la tâche a la charge sur le processeur ait été traité, en d'autres termes : après la complétion de la tâche courante. La préemption sur les tâches aurait un surcoût en terme de performance, dû à l'exécution régulière de l'ordonnanceur pendant le traitement des tâches. Avec l'ordonnancement coopératif des tâches, pour les cas où l'ordonnancement statique est suffisant pour obtenir une exécution optimale, le surcoût de l'ordonnanceur est minimisé. De plus la préemption des tâches n'apporterait pas de gains de performance puisque l'ordonnancement coopératif assure déjà que dès que le processeur est libre, une autre tâche est ordonnancée (s'il en existe une de disponible).

Il faut noter que sur certaines architectures matérielles, il serait envisageable d'utiliser l'ordonnancement dynamique pour permettre une optimisation supplémentaire consistant à recouvrir dans le temps les accès aux mémoires par des calculs. Cette technique est très courante dans le cadre du calcul haute performance, elle repose sur l'usage parallèle des réseaux de communication entre nœuds de calcul pour lire ou écrire les données et des processeurs pour exécuter le traitement sur d'autres données. Cependant dans les architectures SoC usuelles, cette optimisation n'est pas possible car le processeur, qui a la mémoire locale gérée par un mécanisme de cache, est bloqué lors d'accès mémoires. Cette optimisation nécessite par exemple une architecture où le logiciel est capable de superviser les données en mémoire locale (type *scratchpad*) et où les accès à la mémoire distante peuvent être délégués (via un DMA). De plus, ce mécanisme nécessite une plus grande mémoire locale afin de pouvoir stocker temporairement les données en cours de transfert. Pour ces raisons, nous avons décidé de ne pas traiter ce type d'optimisation dans le cadre de cette thèse.

4.2.7 Synthèse et conclusion

La définition de la projection du modèle de calcul des applications Gaspard sur un modèle d'exécution permet de préciser en détail le comportement final de l'application. Deux projections avaient déjà été définies mais vers d'autres modèles de calcul. En reprenant un certain nombre de choix pris pour ces projections, notamment la granularité de la synchronisation au tableau près (et non pas au motif près) nous avons proposé une projection qui puisse s'accorder avec les contraintes des SoC en général, et surtout permette de respecter l'association définie par le concepteur. Certains choix techniques ont été orientés par la minimisation de l'usage mémoire et la maximisation des performances.

L'exécution d'applications Gaspard selon cette projection repose sur la manipulation des données et des synchronisations avec la granularité du tableau. L'exécution des *tilers*, qui effectuent les calculs d'adresse et les accès mémoire, est couplée à la tâche à laquelle ils sont liés. Les tableaux intermédiaires introduits par la hiérarchie sont évités. Afin de favoriser les performances, les synchronisations ne sont utilisées qu'entre les tâches réparties sur plusieurs processeurs. Un ordonnancement statique est utilisé pour les tâches entièrement placées sur un seul processeur, tandis qu'un ordonnancement dynamique coopératif est utilisé pour augmenter les performances des tâches synchronisées alors que le comportement exact de l'architecture matérielle est inconnu.

Après avoir défini le fonctionnement de la simulation au motif près, et avoir spécifié le modèle d'exécution des applications Gaspard, il est alors possible de définir, à la croisée de ces deux spécifications, une simulation au motif près de ces applications. Ainsi, dans la section suivante nous allons porter notre attention sur l'implémentation d'une telle simulation.

4.3 Implémentation de la simulation en SystemC

L'implémentation des modèles de SoC Gaspard en une co-simulation SystemC au niveau PA est décomposée entre la partie logicielle et la partie matérielle.

L'implémentation de la partie matérielle de la simulation PA est similaire à celle de simulation PVT. Les composants élémentaires sont remplacés par des instances des IP sur lesquels ils sont déployés. Le déroulement des composants répétés est fait lors de la phase d'initialisation de la simulation. Durant cette phase les connexions inter-composants sont également créées selon les spécifications du modèle. Les communications entre composants sont au niveau de la transaction et contiennent en plus des données usuelles des informations temporelles permettant l'estimation du temps d'exécution. On pourra retrouver les détails de l'implémentation dans la thèse de Rabie Ben Atitallah [16].

La partie logicielle de la simulation est novatrice : elle se repose sur une abstraction de l'application plutôt que sur l'application entièrement compilée. Dans cette section nous allons présenter comment il est possible de simuler une application Gaspard en implémentant une abstraction du modèle d'exécution Gaspard proposé dans la section précédente. Tout d'abord, nous présenterons l'intégration du composant processeur dans le reste de la simulation, ensuite nous détaillerons l'implémentation de l'ordonnanceur dynamique, puis nous aborderons le mécanisme de synchronisation intégré au simulateur, suivra la description des techniques employées pour la simulation des accès aux données et enfin nous présenterons la gestion du temps à ce niveau de simulation.

4.3.1 Simulation en SystemC

La cible que nous visons est une simulation en SystemC. Afin de réduire les difficultés de développement et de simplifier l'usage de ce niveau de simulation, nous proposons de garder le même type de simulation des composants matériels qu'au niveau PVT. Cela offre en particulier l'avantage à l'utilisateur de ne pas devoir fournir des IP spéciaux pour chaque composant matériel, au contraire l'ensemble des IP au niveau d'abstraction PVT peut être réutilisé. Selon la méthode de simulation au niveau PA, les IP de processeur sont remplacés par un code correspondant à la simulation logicielle. Vis-à-vis des autres composants, ce composant doit se comporter identiquement à son alter ego du niveau PVT.

Dans l'implémentation que nous proposons chaque instance de processeur est simulée par un code spécifique. Néanmoins les processeurs répétés se partagent le même code. Dans ce cas, la distinction entre les différentes répétitions se fait à l'aide d'arguments passés lors de l'initialisation de la simulation, représentant l'indice multi-dimensionnel. Respectant l'organisation usuelle d'un composant de processeur SystemC au niveau TLM, ce code contient en plus d'une fonction d'initialisation, une fonction d'exécution et un port par lequel les accès mémoire sont effectués.

Lorsqu'une simulation de la partie fonctionnelle est requise, les IP de chaque composant élémentaire doivent être appelés afin d'effectuer les traitements de données et ainsi d'obtenir les sorties réelles. À cette fin, l'ensemble des *CodeFiles* des IP est compilé pour le processeur hôte (celui sur lequel la simulation va s'exécuter) et lié au reste du simulateur. Les IP nécessitent donc d'être compatibles avec le processeur hôte, soit parce qu'ils sont écrits dans un langage tel que C ou C++, soit parce qu'il existe une implémentation compilée pour le jeu d'instruction du processeur hôte en plus de celle pour le processeur cible. Si cela n'est pas le cas, alors la partie fonctionnelle de la simulation ne sera pas disponible. Les appels vers la fonction de l'IP sont effectués en respectant le protocole défini dans la section 3.2.

Lorsque l'application ne traite pas de tableau infini, la simulation doit pouvoir s'arrêter dès que l'application est terminée, dès qu'il n'y a plus de calculs à faire. Pour cela, chaque processeur est chargé de détecter qu'il n'exécute plus aucune tâche, c'est-à-dire que toutes les répétitions de toutes les tâches ont été traitées. Une fois que l'ensemble des processeurs a terminé, c'est la simulation complète qui se termine.

4.3.2 Un ordonnanceur intégré au simulateur

Comme nous l'avons vu, en ce qui concerne l'ordonnancement et les synchronisations, le modèle d'exécution des tâches Gaspard est décomposé en deux, selon les niveaux de hiérarchie. À partir du niveau de hiérarchie où toutes les tâches sont exécutées sur le même processeur, tous les niveaux inférieurs sont ordonnancés statiquement. Les niveaux supérieurs sont ordonnancés dynamiquement.

4.3.2.1 Granularité d'une activité

Chaque tâche du niveau le plus bas de l'ordonnancement dynamique est appelée *activité*. Une activité est construite récursivement en partant de cette *tâche d'origine*. On y retrouve toute la hiérarchie contenue dans la tâche. Si une tâche contient des sous-tâches, alors chaque sous-tâche est écrite, dans un ordre respectant les dépendances de données. L'écriture d'une tâche contenant une instance répétée correspond à un nid de boucles qui permet de parcourir l'ensemble des répétitions. Quant aux tâches élémentaires, elles sont obtenues en générant d'une part le code correspondant aux *tilers* avec court-circuit et d'autre part l'appel de la fonction de l'IP. Ainsi le code interne à la tâche d'origine *B* de la figure 4.10 d'une activité ressemble à ce pseudo-code :

```
1 taskB() {
      // sous-tache D
      for (i=0; i< 10; i++) {
          // tache elementaire
5         tilers d entree
          D_ip();
          tilers de sortie
```

```
        }
        // sous-tache E
10      for (j=0; j<2; j++) {
            for (k=0; k<2; k++) {
                // tache elementaire
                tilers d entree
                E_ip();
15              tilers de sortie
            }
        }
```

En plus de ce code interne qui correspond aux niveaux bas de la hiérarchie, chaque activité contient une copie de toute la hiérarchie supérieure à la tâche d'origine. En d'autres termes, dans l'arborescence de l'application, chaque nœud faisant partie du chemin entre la racine et la tâche d'origine de l'activité est ajouté à l'activité. Ces nœuds correspondent forcément à une tâche composée ou répétitive (puisque toute tâche élémentaire ordonnancée dynamiquement est une tâche d'origine d'une activité). Chaque nœud inclus dans l'activité se traduit par la création des nids de boucles parcourant les répétitions du nœud et par l'ajout des synchronisations pour chacune des connexions correspondant à un passage de tableau de données.

Lorsqu'une activité est exécutée en parallèle sur plusieurs processeurs, les répétitions parcourues doivent être seulement celles spécifiées dans le modèle d'association, par conséquent les nids de boucles dépendent de l'indice du processeur. Ainsi chaque activité contient tout le code englobant la tâche d'origine et tout le code qui est contenu dans cette tâche d'origine. Elle peut donc s'exécuter indépendamment des autres activités, avec lesquelles elle n'est liée que par les synchronisations. Sous forme de pseudo-code, nous notons `for(a/P)` une boucle qui parcourt une dimension des indices de répétitions à l'aide de la variable `a` en fonction du processeur `P`. La structure globale du code de l'activité *C* de la figure 4.10 ressemble à cela :

```
1   activityC() {
        // nids de boucle de la racine M (repetition infinie)
        for (;;) {
            // nids de boucle de la tache A (vide)
5           {
                // synchronisations debut (avec B)

                // lecture des tableaux entree (aucun)

10              // nids de boucle de la tache C distribuee sur ARM7
                for (i/ARM7) {
                    taskC();
                }
                // ecriture tableau de sortie (1)
15              ...
                // synchronisations fin (avec B)
                ...
            }
        }
20  }
```

Reprenons l'exemple de la figure 4.7 page 103. Dans la figure 4.10, sont représentées les deux activités obtenues à partir de l'application originale. Chaque activité correspond à une

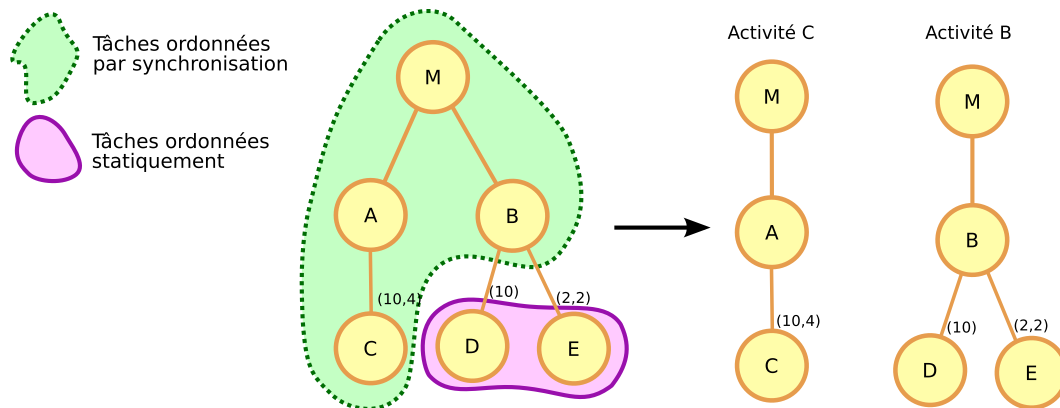


FIG. 4.10: Représentation du modèle présenté figure 4.7 sous forme arborescente des tâches sont incluses dans une activité selon le type d'ordonnancement qui va être appliqué. Cet ordonnancement est dépendant du placement sur l'architecture matérielle.

feuille de l'arbre contenu dans la partie ordonnancée par les synchronisations, ici les tâches d'origine sont donc *B* et *C*. Pour chaque tâche d'origine, toute la hiérarchie supérieure est recopiée, ainsi l'activité *C* contient en plus de *C* les boucles et les synchronisations correspondantes à *A* et *M*. De même, l'activité *B* contient les boucles les synchronisations correspondantes à *M*. Les tâches ordonnées statiquement sont copiées dans l'activité correspondante à la tâche d'origine qui les contient, ainsi *D* et *E* sont copiés dans l'activité *B*. Le processeur ARM9 exécutera donc une activité, *B*, tandis que chacun des deux processeurs ARM7 exécutera une activité, *C*, qui correspond au parcours de la moitié des répétitions de la tâche *C*.

4.3.2.2 Implémentation des contextes d'exécution

Chaque activité est exécutée par l'ordonnanceur dynamique intégré au composant processeur. Dans notre implémentation, nous nous sommes basés sur l'API de threads POSIX[4] pour pouvoir séparer l'exécution de chaque activité à l'intérieur du même composant processeur écrit en SystemC. Par rapport à créer notre propre bibliothèque de gestion minimale des contextes d'exécution, l'avantage d'utiliser ce standard pour contenir les activités est la portabilité. Cela permet d'avoir le même code du simulateur quelque soit l'architecture ou le système d'exploitation de l'ordinateur hôte, du moment que le standard POSIX est supporté (ce qui le cas dans l'extrême majorité des cas). Attention, il faut bien noter que bien que les threads soient habituellement utilisés pour exécuter plusieurs tâches *en même temps*, ce n'est pas l'usage que nous en faisons dans le simulateur. Ils ne servent que comme support pour séparer les contextes d'exécution des différentes activités, une seule activité est exécutée à la fois pour chaque composant processeur. Notons au passage que cette technique est très proche de celle qu'utilise SystemC pour créer les threads de chaque composant de la simulation, qui exécute également chaque thread un par un.

Le fonctionnement du simulateur d'ordonnancement repose sur le concept de maître/esclave. Dans ce patron de conception de systèmes parallèles, le thread maître a pour rôle de répartir le travail parmi les esclaves. Chaque fois qu'un esclave termine sa partie du travail, il

[4] http://www.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html

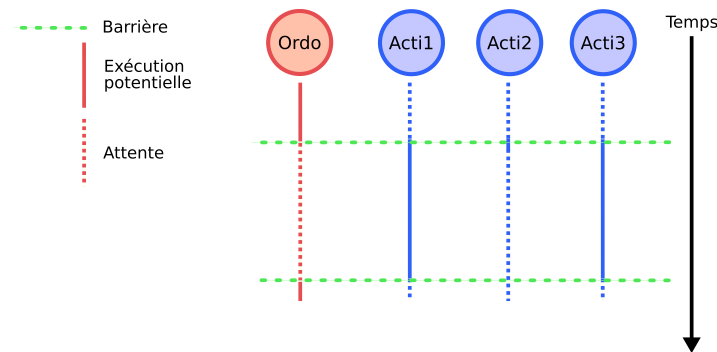


FIG. 4.11: Exemple de cycle d'ordonnancement utilisant le concept de maître/esclave. À l'aide de barrières l'ordonnanceur gère l'exécution d'un ensemble d'activités. Dans le cycle présenté les activités 1 et 3 peuvent exécuter un traitement. Leur exécution est possible en même temps, mais elles gèrent entre-elles la séquentialité d'exécution réelle. L'activité 2 ne peut pas s'exécuter en raison de dépendance de données non satisfaite.

recontacte le maître pour recevoir une autre partie du travail à traiter. Nous utilisons ici le maître comme ordonnanceur, et chaque esclave correspond à une activité. Le contact entre maître et esclave se fait à l'aide de barrières.

Un schéma de déroulement d'un cycle d'ordonnancement est présenté figure 4.11. Au départ toutes les activités sont prêtes à être exécutées. Dès qu'une activité a une dépendance de données non satisfaite elle se met en attente sur une barrière partagée avec le maître. Cependant le maître, l'ordonnanceur, ne relâche pas immédiatement la barrière, laissant le thread bloqué. Ainsi seules les activités qui ont toutes leurs dépendances de satisfaites s'exécutent. Une fois qu'une activité a généré un tableau de données, elle se synchronise avec les autres tâches et se met en attente en contactant le maître. Lorsque toutes les tâches sont en attente du maître cela signifie que chaque activité est soit en attente de réception d'un tableau, soit en attente de fin de transmission d'un tableau. Le thread maître laisse alors la main aux autres composants matériels pour avancer dans la simulation. Cela correspond à l'état oisif du processeur. Après quelques cycles simulés, lorsque SystemC rend la main au composant processeur, le maître relâche la barrière ce qui a pour effet de débloquer tous les threads esclaves et ainsi le cycle reprend.

Parmi les activités qui ne sont pas en attente de données, chaque tâche partage le processeur l'une après l'autre. L'ordonnancement est coopératif, les tâches décident elles-mêmes quand passer la main à une autre tâche. Selon le modèle d'exécution, ce changement de tâche s'effectue à chaque fois que la tâche d'origine de l'activité a parcouru toutes les répétitions. Pour simuler ce comportement de partage du processeur, un verrou d'exclusion mutuelle, *mutex*, est défini au sein de chaque composant processeur. Pour qu'une activité puisse s'exécuter elle doit acquérir ce mutex. Dans chaque activité se trouvent un unique accès au mutex et un unique relâchement du mutex, placés autour de l'appel vers la tâche d'origine. Ce partage de ressource processeur joue le rôle d'ordonnanceur coopératif, les tâches s'exécutant séquentiellement. L'ordre d'exécution n'est pas explicite, mais il correspond à un ordre possible d'ordonnancement de l'application réelle.

Une fois qu'une activité a parcouru l'ensemble des répétitions qui lui étaient attribuées, cette partie de l'application est terminée. Dans ce cas, le thread esclave associé est lui aussi

terminé. Lorsque le maître ne possède plus aucun esclave, c'est le signe qu'il n'y a plus de tâches de l'application à exécuter sur le processeur. Le maître se charge de retourner cette information au reste du simulateur, qui pourra ainsi stopper la simulation dès que toutes les tâches de tous les processeurs auront fini leur traitement.

Nous avons créé une petite bibliothèque de fonctions pour permettre au composant processeur de gérer les activités selon le mécanisme que nous venons de présenter. Cette bibliothèque met à disposition un objet `abar` qui est partagé par tous les threads d'un composant processeur. En plus d'une méthode d'initialisation qui prend en argument le nombre de threads esclaves, la bibliothèque met à disposition quatre autres méthodes. La méthode `master()` est appelé par le thread maître pour attendre tous les threads esclaves. La méthode `slave()` met un esclave en attente et ne retourne que lorsque le maître a relancé les esclaves. La méthode `release()` permet au maître de relancer tous les esclaves d'un seul coup et aussi de savoir le nombre d'esclaves qui continuent à s'exécuter. Enfin la méthode `bye()` permet à un thread esclave d'indiquer sa terminaison. À l'aide de cette bibliothèque le code du simulateur correspondant à l'ordonnancement comporte une quinzaine de lignes seulement. Le code du thread maître, qui est le thread original créé par SystemC et depuis lequel les autres threads sont créés correspond à ce code :

```
1  // creation de l'objet abar pour 4 threads
   int slaves_left = 4;
   abar *ab = new abar(slaves_left);
   // creation des 4 threads
5  struct tlmpa_activity act_X = { index, &rw_mutex, ab, &activityX, ltimer, port_rw};
   pthread_create( &thread, NULL, &thread_func, &act_X );
   :
   :
   while (slaves_left) {
10          // attend les esclaves
           ab->master();
           // Laisse le reste du simulateur s'executer
           ltimer->sync();
           // relache les esclaves
15         slaves_left = ab->release();
   }

   // fin du processeur
   return;
```

Voici le code d'un thread esclave :

```
1  void* thread_func( void *vptr_args )
   {
           struct tlmpa_activity* a = (struct tlmpa_activity *)vptr_args;
           (a->activity)(a);
5          a->ab->bye();
           return NULL;
   }
```

4.3.2.3 Vers un niveau d'abstraction plus élevé

Dans l'implémentation que nous venons de décrire les tâches s'exécutent séquentiellement sur chaque processeur bien qu'elles soient chacune dans un thread séparé. Une extension

possible de la simulation serait de s'affranchir de cette séquentialité en autorisant l'exécution simultanée de multiples activités d'un processeur. Techniquement, cela peut se faire très aisément en réduisant les sections non-partagées protégées par le mutex du processeur. Au lieu de placer d'acquérir le mutex pour toute l'exécution d'une activité, il peut être prit uniquement lors d'accès mémoire. Cette extension permettrait un gain de performance lorsque le simulateur s'exécute sur un ordinateur multiprocesseur et que la bibliothèque de thread supporte le déploiement sur plusieurs processeurs : le traitement de plusieurs tâches peut être fait en parallèle. Cependant, bien que le nombre d'accès mémoire et le temps de calcul puisse être estimés avec la même précision qu'offre le niveau de simulation PA, l'ordre des accès mémoires pourrait ne plus être assuré, plusieurs tâches pouvant s'enchevêtrer. Indirectement, cela signifie que la détection des conflits d'accès sur les réseaux d'interconnexion perd en précision, et donc le temps d'exécution simulé également. Ce niveau d'abstraction serait encore plus proche du niveau de simulation fonctionnelle.

4.3.3 Le mécanisme de synchronisation

Pour traiter les dépendances de données, il faut que les tâches puissent attendre qu'un ou plusieurs tableaux soit disponibles, et aussi qu'elles puissent attendre qu'un ou plusieurs tableaux produits aient été entièrement lus avant de récrire par-dessus. Ces attentes sont assurées par le mécanisme de synchronisation. Dans la simulation au motif près, ce mécanisme est intégré directement au simulateur ce qui permet de retourner plus d'informations concernant le comportement des tâches tout en accélérant la simulation. Notons que cela ne serait pas possible sans la modélisation explicite de ces dépendances dans le modèle de SoC.

Concernant la projection vers un modèle d'exécution, nous avons indiqué que chaque dépendance correspond à une FIFO. Cela permet d'implémenter les attentes de tableau aussi bien du côté consommateur que du côté producteur. Il y a une FIFO par connexion, et non pas par tableau. Cela signifie que si un tableau est lu par plusieurs tâches, il y aura autant de FIFO que de consommateurs (ce n'est important que dans ce sens, puisque dans le modèle d'application Gaspard, un tableau ne peut pas provenir de plusieurs tâches). Néanmoins, dès lors qu'une tâche est distribuée sur plusieurs processeurs, chaque répétition de la tâche se partage la lecture ou l'écriture du tableau, ce qui implique que chaque FIFO peut avoir plusieurs producteurs et plusieurs consommateurs. Le partage est légèrement particulier puisqu'un élément est considéré produit ou consommé qu'une fois que *tous* les producteurs, respectivement les consommateurs, ont traité cet élément. Ainsi, avec une telle FIFO au départ tous les producteurs peuvent écrire dans le tableau, mais aucun consommateur ne peut y accéder. Au moment où le dernier producteur a terminé d'écrire sa partie, tous les consommateurs peuvent commencer à accéder au tableau. Similairement, lorsque la FIFO est pleine, les producteurs doivent attendre que chacun des consommateurs ait terminé la lecture du premier tableau.

Toutes les activités sont synchronisées par ce mécanisme. Pour chaque tâche au-dessus de la tâche d'origine d'une activité (les niveaux ordonnancés dynamiquement), deux appels aux primitives de synchronisation sont placés par tableau traité. Les tableaux qui correspondent à un motif de tableau dans la tâche supérieure ne font pas l'objet de synchronisations (puisque la tâche supérieure s'est déjà synchronisée sur le tableau englobant). Seuls les tableaux qui correspondent explicitement à un passage de données entre deux tâches du même niveau hiérarchique nécessitent des synchronisations. Dans l'exemple précédent figure 4.10, la tâche A transmet un tableau à la tâche B, de même pour la tâche D qui produit un tableau consommé

par *E*. Dans cette application il n'y aura donc qu'une seule connexion qui requiert des synchronisations, celle entre *A* et *B*, les autres connexions sont soit entre des tâches ordonnancées statiquement soit entre une tâche et une sous-tâche.

4.3.3.1 Primitives de synchronisation

L'implémentation de ce mécanisme de synchronisation est effectuée par la mise en place de quatre primitives correspondant au début de lecture d'un tableau (`startReading()`), au début d'écriture d'un tableau (`startWriting()`), à la fin de lecture d'un tableau(`finishReading()`), et à la fin d'écriture d'un tableau(`finishWriting()`). Ces primitives, qui prennent en argument un objet correspondant à la FIFO, sont placées au tout début et à la toute fin d'une tâche, en dehors des appels vers les sous-tâches et des boucles. Lorsqu'une tâche manipule plusieurs tableaux ou qu'un tableau est envoyé à plusieurs tâches, l'ordre d'appel des primitives pour chaque connexion est indifférent. Les deux primitives de début sont bloquantes. Si la FIFO est vide ou si elle n'est pas terminée de consommer, ces primitives se chargent de mettre l'activité en attente, via la fonction d'ordonnancement `slave()`. Les deux primitives de fin ne sont jamais bloquantes, elles servent à relancer les tâches en attente. Il existe une autre fonction qui permet d'initialiser une FIFO. Elle prend deux paramètres en entrée qui correspondent au nombre de producteurs et de consommateurs.

Voici un exemple d'utilisation des primitives pour la synchronisation de trois producteurs et deux consommateurs :

```
1  // initialisation de la FIFO
   gasp_sync sync_X(2, 3);

   // L'activite de la tache productrice
5  void activity_prod(struct tlmpa_activity* a)
   {
       // nids de boucle des taches superieures
       for (a/P) {
           for (b/P) {
10             // debut d'ecriture
               sync_X.startWriting(a->ab);
               // nids de boucle d'autres noeuds intermediaires
               for (d/P) {
                   ...
15                 taskX()
                   ...
               }
               // fin d'ecriture du tableau
               sync_X.startWriting(a->ab);
20         }
       }
   }

   // L'activite de la tache consommatrice
25 void activity_cons(struct tlmpa_activity* a)
   {
       // nids de boucle des taches superieures
       for (a/P) {
```

```
        for (c/P) {
30          // debut de lecture
            sync_X.startReading(a->ab);
            // nids de boucle d'autres noeuds intermediaires
            for (e/P) {
                ...
35              taskX()
                ...
            }
            // fin de lecture du tableau
            sync_X.finishReading(a->ab);
40      }
    }
}
```

4.3.3.2 Mise en œuvre des synchronisations

L'implémentation de ces primitives est disponible sous la forme d'une classe nommée `gasp_sync` représentant une FIFO et ayant chacune des primitives comme méthode. Les synchronisations pour le cas d'une FIFO vide et pour le cas d'une FIFO pleine sont mises en œuvre de manière symétrique. Au total, quatre variables partagées sont utilisées. `p` et `c` contiennent respectivement le nombre de producteurs et de consommateurs en cours d'exécution. `pp` et `pc` indiquent la phase du dernier tableau disponible respectivement en écriture et en lecture. En plus de ces variables, chaque thread contient une variable `pos` qui indique sur quelle phase de tableau il travaille. Au démarrage cette dernière variable vaut 1 pour tous les threads : ils doivent tous travailler sur le premier tableau. `p` contient le nombre maximal de producteurs, tandis que `c` est nul : tous les producteurs s'exécutent et aucun consommateur. `pp` vaut 1 et `pc` vaut 0 : le premier tableau est disponible en écriture, mais pas encore en lecture (seul le tableau imaginaire 0 est disponible).

Les consommateurs ne peuvent lire le tableau que lorsque leur phase correspond à la phase du tableau disponible en lecture : tant que leur variable `pos` est différente de `pc` ils restent en attente. Les producteurs, qui ont la contrainte similaire avec `pp`, peuvent s'exécuter dès le départ. Lorsqu'un producteur a terminé son écriture du tableau il décrémente `p`, le nombre de producteurs en cours, et tente de travailler sur le tableau suivant en incrémentant son numéro de tableau ; il se retrouve automatiquement bloqué. Si le nombre de producteurs devient nul alors le nombre de consommateurs en cours d'exécution est mis au maximum de consommateurs et `pc`, la phase du tableau disponible en lecture, est inversée. Ceci débloque les consommateurs. Ensuite, un déroulement symétrique se passe avec les producteurs en attente. Ce mécanisme permet l'exécution successive des producteurs et des consommateurs nécessaire aux tâches Gaspard.

La figure 4.12 présente un exemple de déroulement des synchronisations avec trois producteurs et deux consommateurs. On y retrouve la mise en attente des consommateurs au départ, pendant que tous les producteurs s'exécutent. Au fut et à mesure que les producteurs achèvent leur écriture, la variable `p` est décrémentée, ce qui permet au dernier producteur de déclencher la synchronisation. Puis c'est au tour des consommateurs de s'exécuter en travaillant sur le premier tableau. On peut remarquer également que chaque fois qu'un thread tente de travailler sur le tableau suivant, il doit commencer par attendre sa disponibilité.

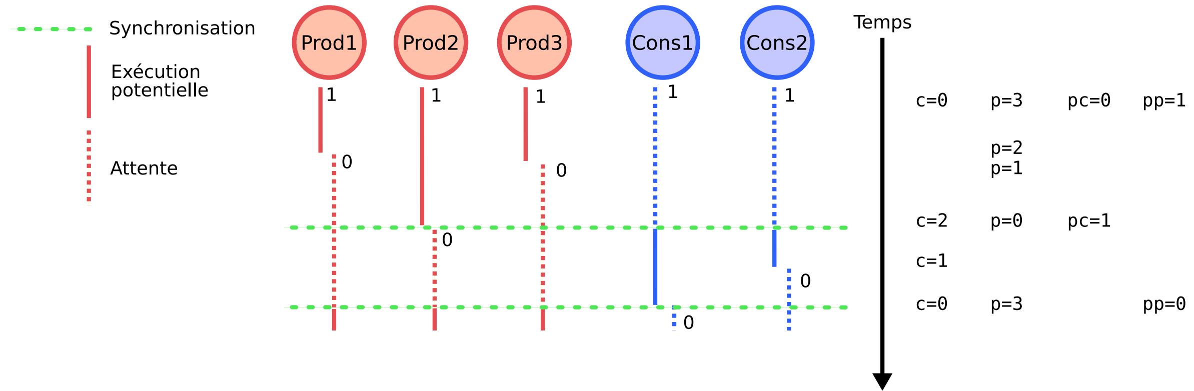


FIG. 4.12: Schéma du déroulement du mécanisme de synchronisation avec 3 producteurs et 2 consommateurs. Chaque diagramme vertical représente le déroulement d'un thread, avec la valeur de la variable locale `pos` indiqué le long des lignes. Les lignes horizontales représentent les instants de synchronisation. À droite sont indiquées les valeurs des variables au cours du temps.

4.3.4 Lecture et écriture des données

Vis-à-vis des composants d'interconnexion et de mémoire, les accès aux données sont identiques à ceux d'une simulation PVT. Par contre, en ce qui concerne le composant processeur, alors que dans la simulation PVT les accès correspondent au décodage de certaines instructions contenant l'adresse explicitement indiquée, au niveau PA les accès aux données sont exprimés à l'intérieur du code des *tilers* et se réfèrent à l'élément souhaité par une série de coordonnées d'un point dans un tableau. De plus, en raison du découpage en activités, les requêtes peuvent émaner de n'importe quel thread du composant processeur. Il faut donc que le simulateur du processeur puisse faire passer tous les accès par le seul port qui le connecte à la mémoire et qu'il soit aussi capable de faire le lien entre cette représentation abstraite de l'adresse et l'adresse linéarisée dans l'espace d'adressage de l'architecture. Cette correspondance doit pouvoir être faite dans les deux sens afin de pouvoir présenter dans le journal d'exécution généré par le simulateur un ensemble cohérent d'informations sur le déroulement de la simulation au motif près.

Pour centraliser les accès mémoires en un seul point, le port de communication du processeur, les méthodes `read()` et `write()` sont mises à disposition des *tilers* par le processeur. Elles se transmettent les requêtes dans le bon format. Comme nous le verrons dans la section suivante, elles ont surtout pour rôle d'assurer la synchronisation temporelle du processeur avec le reste du simulateur lors de chacune de ces requêtes. Par ailleurs, le fait que plusieurs threads s'exécutent à l'intérieur du composant processeur signifie qu'en théorie un thread pourrait tenter d'accéder à la mémoire pendant qu'un autre thread est déjà en cours d'accès. Cette situation ne doit pas arriver car d'une part dans la réalité les requêtes mémoire sont séquentialisées et d'autre part le simulateur SystemC n'est pas capable de traiter les deux communications en parallèle (il n'a pas connaissance des différents threads du processeur). Pour ces raisons, les appels à `read()` et `write()` par les threads doivent *toujours* être protégés par le mutex de partage du processeur. Ce mutex assure la séquentialité des accès mémoire depuis le processeur. Étant donné que les *tilers* sont autour des tâches

élémentaires et que l'ordonnancement statique d'une activité est toujours protégé par le mutex de partage du processeur pour éviter l'entrelacement des tâches, il n'est pas nécessaire d'acquérir explicitement le mutex lors des accès mémoires depuis les *tilers*.

La conversion entre les coordonnés d'un élément dans le tableau et l'adresse mémoire se fait selon la linéarisation que nous avons décrite dans le modèle d'exécution. En plus de cela, lors de l'initialisation de la simulation, les composants processeurs listent pour chaque tableau utilisé par les tâches sa taille (multi-dimensionnelle), son adresse de base en mémoire et le nombre d'octets que prend chaque élément. Avec ces informations, il est possible d'associer une requête sur le bus à un élément de tableau particulier. Notons que lorsqu'un tableau est distribué le calcul d'adresse doit également intégrer les deux *tilers* de la distribution pour connaître l'adresse de base du banc mémoire. Dans l'implémentation actuelle, pour pouvoir déterminer simplement l'élément à partir d'une adresse dans ce cas, chaque banc mémoire contient l'espace pour tout le tableau et la linéarisation s'effectue de manière identique au cas où un tableau est directement alloué. Cela nécessite plus d'espace mémoire mais dans le cadre de la simulation ceci n'est pas une contrainte et n'affecte pas les observations.

Par ailleurs, notons également que lorsque la simulation exécute réellement les IP, qui sont compilés pour le processeur hôte, le type de donnée des éléments de tableau doit lui aussi être codé selon le processeur hôte afin de pouvoir faire fonctionner les fonctions correctement. Cela peut être un problème si la taille en octet d'un élément est différente entre le processeur hôte et le processeur cible. Par exemple si l'élément est de type `long` et que la simulation est faite sur une architecture IA-64 (64 bits), mais que l'architecture cible est un MIPS 32 bits, chaque requête mémoire nécessitera deux transmissions au lieu d'une seule dans la réalité. Dans ce type de situation, il faut désactiver l'exécution des IP logiciels lors de la simulation afin de pouvoir observer le nombre correct de transmissions sur les réseaux d'interconnexion.

4.3.5 Estimation du temps d'exécution

Jusqu'à présent nous avons vu les techniques mises en œuvre afin d'obtenir une simulation aussi proche que possible du modèle d'exécution des applications Gaspard. Nous allons ici aborder un aspect différent de la simulation : l'estimation du temps d'exécution. Bien sûr, connaître le temps d'exécution de l'application peut être en soi très important pour le concepteur de SoC, par exemple pour pouvoir juger si le système est capable de supporter une cadence de données en entrée. Néanmoins l'estimation temporelle joue également un grand rôle pour pouvoir garder chaque composant synchronisé par rapport aux autres au fur et à mesure de la simulation. Ce synchronisme entre l'avancement dans la simulation de chaque composant est primordial pour estimer les goulots d'étranglement de l'architecture et les conflits d'accès sur les réseaux d'interconnexion.

De manière schématique, la gestion temps dans le simulateur SystemC repose sur l'annonce par chaque composant de l'intervalle de temps simulé depuis l'annonce précédente. Au fur et à mesure, le simulateur ordonnance les différents composants de manière à ce que chacun ait écoulé le même temps simulé. Un composant peut appeler la fonction SystemC `wait()` avec le temps écoulé en argument. Il sera alors mis en attente jusqu'à ce que tous les autres composants aient rejoint ou dépassé le temps total simulé du composant.

Comparé à la simulation au niveau PVT utilisant un ISS, qui après l'interprétation de chaque instruction annonce l'intervalle de temps correspondant écoulé, l'estimation temporelle du processeur au niveau PA est nécessairement plus grossière puisque le traitement d'un motif peut correspondre à plusieurs centaines ou milliers d'instructions. Le modèle possède

deux informations spéciales pour aider à la simulation : le temps d'exécution d'un appel sur le processeur cible pour chaque IP, et le temps d'exécution du calcul d'une dimension d'un calcul d'adresse sur le processeur cible.

La mise en œuvre de la simulation PA consiste à compter localement dans le processeur le temps écoulé à l'aide de ces informations et à ne synchroniser le processeur avec le reste des composants uniquement lorsque cela est absolument nécessaire : avant les transactions avec les autres composants. En effet, puisque les autres composants n'influencent pas les activités s'exécutant sur un processeur tant qu'il n'y a pas d'accès mémoire ou de synchronisations de FIFO et que les activités n'influencent pas les autres composants en dehors de ces deux actions, synchroniser le temps de simulation plus fréquemment ne conduirait qu'à ralentir la simulation sans fournir plus de précision.

Dans le code du simulateur de processeur, l'estimation du temps est prise en charge par l'ajout de code en quatre points :

- **Le calcul des *tilers***, à chaque calcul d'une dimension de *tiler* d'entrée ou de sortie le compteur de temps local au processeur est incrémenté selon la valeur spécifiée dans le modèle.
- **L'appel à un IP**, qu'il soit réel ou juste simulé, à chaque fois qu'une fonction d'un IP est appelée, le compteur local du processeur est incrémenté selon la valeur spécifiée pour cet IP (et en fonction du processeur cible simulé).
- **L'état oisif du processeur**, lorsque toutes les activités d'un processeur sont en attente, le processeur laisse un petit laps de temps aux autres composants avant de relancer les activités pour voir si une FIFO est prête. Pour une simulation la plus précise possible, il peut correspondre au temps d'exécution d'une instruction sur le processeur cible mais cela tend à ralentir énormément la simulation (car cela entraîne autant de changements de contexte que dans une simulation PVT). Plus cette valeur est grande, plus rapide est la simulation et moins elle devient précise. Nous avons décidé de mettre la même valeur que le calcul d'un *tiler*, qui correspond au temps minimum pour qu'une autre activité puisse déclencher une synchronisation. Au retour de l'attente le compteur local du processeur doit être incrémenté d'autant.[5]
- **Les accès mémoire**, avant chaque lecture ou écriture le processeur doit informer le simulateur du temps écoulé depuis la dernière synchronisation temporelle. Ensuite la main est transmise aux autres composants pendant que le processeur attend la réponse de sa requête. Lorsque la réponse à la requête est reçue, le temps écoulé durant l'attente est fourni. Ce temps doit être ajouté au temps local du processeur.

Nous avons développé une classe nommée `gasp_time` qui se charge d'exposer une interface simple pour la gestion du temps qui vient d'être spécifiée. Chaque processeur possède une instance de cette classe. En plus de la méthode d'initialisation qui se charge de mettre à zéro le compteur de temps local, quatre méthodes sont disponibles. Pour incrémenter le compteur local, il faut utiliser la méthode `add()`. Elle prend en argument le nombre de nanosecondes à incrémenter au compteur. L'addition est protégée par un mutex. La méthode `sync()` se charge de synchroniser le reste de la simulation avec le composant processeur. Au retour de cette méthode tous les composants ont avancé au moins jusqu'au même temps que le processeur. La méthode spéciale `add_nosync()` est identique à `add()` mais le temps ajouté ne sera pas utilisé dans lors de la prochaine synchronisation. Elle est utilisée lors des

[5]Théoriquement il serait possible de mettre en attente le processeur indéfiniment jusqu'à ce qu'une synchronisation de dépendance soit exécutée mais nous n'avons pas réussi à implémenter cette technique en SystemC.

accès mémoires, qui se chargent eux-mêmes de synchroniser le processeur. Enfin, `get()` permet simplement de lire le compteur local pour retourner des informations à l'utilisateur.

Voilà ce à quoi peut ressembler l'usage de la classe :

```
1  gasp_time ltimer;
   taskX() {
         // sous-tache A
         for (i) {
5           for (j) {
                 // sous-sous-tache B
                 for (k) {

                       //tilers d'entree
10                     for (Ti) {
                           calcul d'une dimension
                           ltimer(TIME_TILER)
                       }
                       read()
15                     // tache elementaire M
                       M()
                       ltimer.add(TIME_M)
                       //tilers de sortie
                       for (To) {
20                         calcul d'une dimension
                           ltimer(TIME_TILER)
                       }
                       write()
                 }
25          }
         }
   :
   :
   void read(const ADDRESS_TYPE &address, struct tlmpa_activity* a, unsigned int &data)
30 {
           int time = 0;
           a->ltimer->sync();
           a->c->read(address, data, time);
           a->ltimer->add_nosync(time);
35 }
```

4.3.6 Synthèse

Au cours de cette section nous avons parcouru les différents aspects sur lesquels repose l'implémentation en SystemC de la simulation au motif près d'un modèle de SoC Gaspard. Avec les mécanismes mis en place la simulation bénéficie du haut niveau d'abstraction dans lequel est modélisée l'application pour à la fois accélérer le temps d'exécution et permettre à l'utilisateur l'observation du comportement complet du SoC selon des concepts proches de ceux manipulés dans la modélisation.

Alors que les composants matériels sont simulés identiquement à une simulation PVT, les composants processeurs sont particuliers car ils ne se chargent pas de décoder les instructions mais ils *sont* l'application. Chaque processeur est composé de plusieurs threads nommés

activité qui correspondent chacun à une tâche du niveau hiérarchique le plus bas parmi les tâches ordonnancées dynamiquement. À l'aide d'une exclusion mutuelle représentant le partage du processeur, les activités sont ordonnancées de manière coopérative. Le mécanisme de synchronisation entre les tâches de différents processeurs repose sur l'usage de variables partagées à l'intérieur même du simulateur. Les accès mémoire requièrent surtout la conversion dans les deux sens entre la représentation des éléments dans les coordonnées d'un tableau et leur adresse linéarisée dans l'espace d'adressage de l'architecture matérielle. La simulation du temps utilise un compteur local au processeur synchronisé avec le reste des composants aux seuls moments correspondants aux accès mémoires et aux synchronisations de tâches.

4.4 Conclusion

Dans ce chapitre nous avons défini un nouveau niveau de simulation, plus abstrait que ceux actuellement existant. Il repose sur un certain nombre d'hypothèses qui sont vérifiées dans le cadre du traitement de signal systématique et de la modélisation à haut niveau. En parallèle nous avons défini un modèle d'exécution pour les applications Gaspard qui peut convenir à l'usage sur SoC multiprocesseurs. La convergence de ces deux propositions a permis le développement de techniques de simulation à haut niveau des SoC modélisés dans Gaspard. La validation de ces techniques sera développée dans le chapitre 6, entièrement dédié à ce sujet, mais avant nous allons voir comment l'IDM permet de passer d'un modèle abstrait du SoC à une telle simulation.

Chapitre 5

Usage de l'IDM pour la compilation de SoC

5.1 Méta-modèles intermédiaires **122**
5.1.1 Aperçu global du méta-modèle *Polyhedron* 123
5.1.2 Les composants applicatifs 123
5.1.3 Expression de la répétition des tâches 125
5.1.4 Les tableaux de données 127
5.1.5 Le déploiement simplifié 129
5.1.6 Le méta-modèle *Loop* 130
5.1.7 Synthèse 131
5.2 Du méta-modèle *Gaspard* vers le méta-modèle *Polyhedron* **131**
5.2.1 Moteur de transformations modèle à modèle 132
5.2.2 Génération de polyèdres pour exprimer la répétition 134
5.2.3 Découpage de l'arborescence des tâches 136
5.2.4 Placement des tableaux de données 137
5.2.5 Simplification du déploiement 138
5.2.6 Synthèse 139
5.3 Génération de code SystemC **139**
5.3.1 Moteur de transformation modèle-vers-texte 140
5.3.2 Génération du cœur du processeur 141
5.3.3 Génération des activités 142
5.3.4 Création d'un Makefile 144
5.3.5 Synthèse 146
5.4 Synthèse et conclusion **146**

Au cours des deux chapitres précédents, nous avons présenté d'abord une évolution du méta-modèle Gaspard afin qu'il puisse contenir toutes les informations nécessaires à la génération d'un code *complet* puis nous avons présenté ce à quoi doit correspondre ce code : la simulation haut niveau du SoC. Dans ce chapitre, nous allons maintenant nous attacher à l'obtention de ce code de manière automatique. Cette étape plus traditionnellement appelée *compilation*, est basée sur une partie fondamentale de l'IDM : les transformations de modèles.

Le passage du modèle au code ne se fait pas en une seule étape. Il y a une très grande différence entre le modèle d'origine (le SoC) et le résultat final (le code du simulateur). Afin

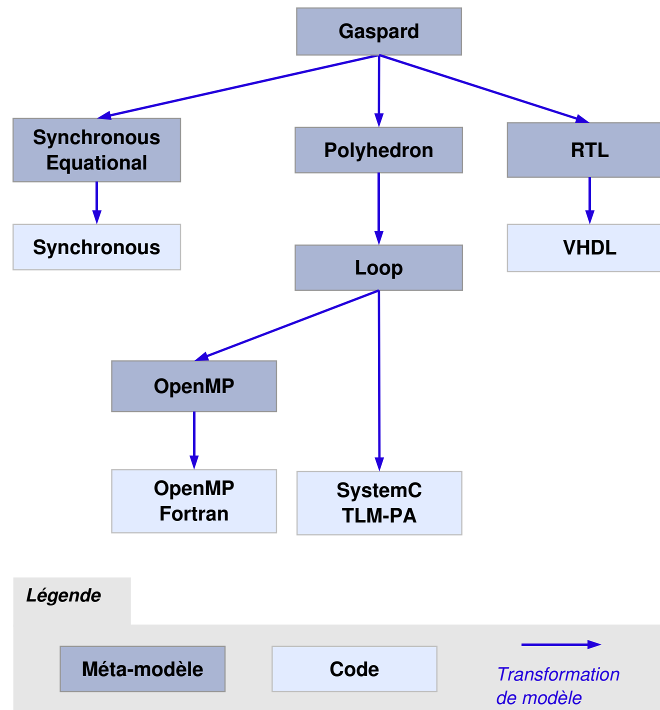


FIG. 5.1: La chaîne de compilation Gaspard. Seule la partie allant du méta-modèle Gaspard vers SystemC/PA est présentée dans ce mémoire.

de diviser les difficultés du passage de l'un à l'autre, nous avons introduit des modèles intermédiaires entre lesquels les transformations sont moins complexes. Ainsi pour parvenir à la cible de compilation, c'est une chaîne de transformations qui est utilisée. La figure 5.1 donne un aperçu de cette chaîne. Comme nous l'avons décrit dans [2], pour arriver au code SystemC/PA à partir d'un modèle Gaspard, trois transformations sont appliquées, passant par un modèle *Polyhedron* puis un modèle *Loop*. Cette figure met particulièrement bien en valeur un des avantages de l'IDM : la réutilisation des transformations. Le projet développé dans l'équipe vise d'autres cibles que SystemC, telles qu'une validation formelle en langage synchrone, une version fonctionnelle en OpenMP et une pour l'obtention d'accélérateurs matériels. Les générations vers ces cibles ont des parties communes. À l'aide des méta-modèles intermédiaires, il est très facile *d'assembler* les transformations les unes après les autres, ce qui permet de réutiliser les premières transformations.

Après avoir détaillé les méta-modèles intermédiaires *Polyhedron* et *Loop* nous présenterons les parties de la chaîne de transformations développées dans le cadre de cette thèse ; à savoir le passage de Gaspard à *Polyhedron* et de *Loop* vers le code SystemC.

5.1 Méta-modèles intermédiaires

Théoriquement, il serait possible d'écrire une transformation qui permette de passer directement du modèle de SoC Gaspard au code de la simulation en SystemC. Outre le fait

de ne pas respecter une recommandation de l'IDM de ne générer du code qu'à partir d'un modèle très proche, cela serait techniquement difficile et surtout il faudrait repartir de zéro pour chaque cible de compilation. Nous avons spécifié les méta-modèles intermédiaires de manière à factoriser au mieux les transformations. Tout ce qui doit être fait quelque soit la cible (OpenMP, VHDL, ou SystemC) est effectué par les transformations amenant au méta-modèle *Loop*. Ce méta-modèle a donc pour but d'être aussi proche que possible du résultat tout en restant indépendant de la plate-forme cible.

Afin de diviser les difficultés de développement, le méta-modèle *Polyhedron* est une étape intermédiaire entre les méta-modèles *Gaspard* et *Loop*. Comme nous allons le voir, ces deux méta-modèles intermédiaires sont en substance très proches l'un de l'autre. La division a été principalement introduite pour séparer une règle faisant appel à un outil externe du reste des règles de transformation car les appels externes relèvent de techniques différentes pour le développement et la maintenance.

Après avoir expliqué les caractéristiques globales du méta-modèle *Polyhedron* nous détaillerons chacun des concepts nouveaux par rapport au méta-modèle *Gaspard*. Puis nous présenterons brièvement le méta-modèle *Loop*.

5.1.1 Aperçu global du méta-modèle *Polyhedron*

La première chose à noter concernant le méta-modèle *Polyhedron* est qu'il est dérivé du méta-modèle *Gaspard*. Nous l'avons construit de manière à ce qu'il puisse être équivalent à un modèle de SoC Gaspard mais sans utiliser un certain nombre de mécanismes présents uniquement pour faciliter la tâche de modélisation. Ainsi, *Polyhedron* peut être globalement compris comme le méta-modèle *Gaspard* sans allocation et avec un déploiement grandement simplifié. Au lieu d'avoir des concepts pour indiquer le placement des tâches et des tableaux, dans ce méta-modèle les tableaux de données sont contenus dans les mémoires sur lesquelles ils sont alloués et les tâches sont contenues dans les processeurs sur lesquels elles doivent être exécutées. Afin de représenter les répétitions des tâches distribuées sur plusieurs processeurs fidèlement, le concept mathématique de polyèdre entier est utilisé. De cette caractéristique dérive le nom du méta-modèle. Comme nous le verrons plus tard, à partir d'une telle représentation, la transformation vers un modèle exécutable est nettement plus directe.

La figure 5.2 représente de manière globale une grande partie de ce méta-modèle intermédiaire. Les différents groupes de concepts seront au fur et à mesure en détail. Pour ne pas complexifier davantage la vue, les concepts relatifs aux types, au matériel et aux connecteurs ne sont pas tous inclus. Notons aussi l'absence de tous les concepts d'allocation, qui ne font pas partie de ce méta-modèle.

5.1.2 Les composants applicatifs

La figure 5.3, un extrait de la figure 5.2, présente les différentes classes utilisées pour modéliser l'application du SoC. De manière identique au méta-modèle *Gaspard*, la modélisation repose sur la notion de *Component*/*ComponentInstance* : un composant, qui correspond à une tâche, peut contenir plusieurs instances de composants, qui correspondent à l'usage de tâches. Cette structure est partagée avec les composants matériels. Par rapport au méta-modèle *Gaspard*, il n'existe que deux types de composants applicatifs : les composants *ElementaryTask* contiennent exclusivement un IP, les composants *LoopComponent* contiennent des

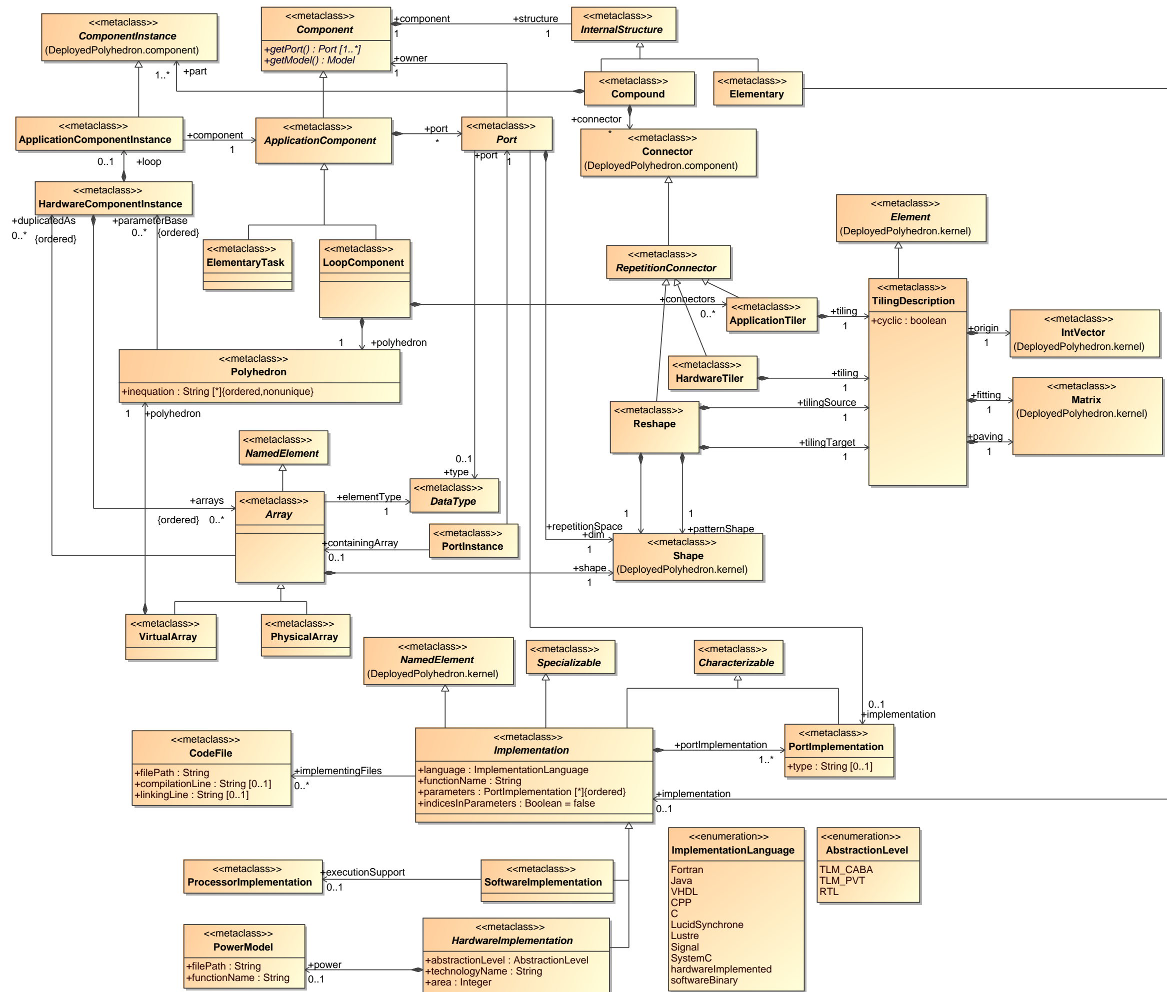


FIG. 5.2: Vue globale d'une grande partie du méta-modèle *Polyhedron*. Le figures suivantes reprennent chacune une partie de cette vue.

sous-composants mais aussi la répétition. En apparence proche des composants *Compound* de *Gaspard* ils diffèrent nettement par le fait que l'espace de répétition n'est pas associé à une instance mais à la structure interne du composant. Cette structure est plus proche du code généré parce que tous les *tilers* englobant l'instance dépendent également de cet espace de répétition.

De plus, également dans le but d'une meilleure adéquation avec le code cible, les connecteurs se distinguent selon le fait qu'ils sont utilisés pour les accès mémoire ou bien pour représenter les dépendances de données entre composants. Les *ApplicationTilers* représentent les accès mémoire par les tâches, même si l'accès est direct et ne nécessite pas de calcul d'adresse. Ils sont donc placés entre chaque *Port* d'un *LoopComponent* et un *PortInstance* d'une instance de composant. Les dépendances de données entre les tâches sont quant à elles représentées par des *Connectors*. Ils relient deux *PortInstances* entre eux. Par ailleurs, ils ne sont pas directement contenus dans le *LoopComponent* mais uniquement dans la structure

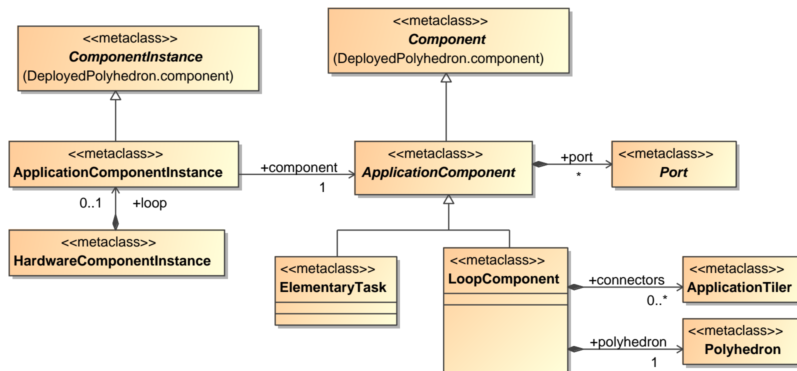


FIG. 5.3: Les concepts permettant de représenter l'application. Par rapport au méta-modèle *Gaspard* on peut noter surtout le fait qu'une instance de matériel (correspondant au processeur) contient directement des tâches et que l'espace de répétition n'est plus représenté par une *Shape* mais par un *Polyhedron*.

interne de celui-ci, qui est également chargée de contenir les instances de composants. Notons que s'il existe une dépendance de données entre des tâches qui ne sont pas sur le même processeur et donc sont contenues dans des composants distincts, elle est aussi représentée par un *Connector*, placé dans la structure interne de l'un des composants.

Les *HardwareComponentInstance* de processeurs contiennent une instance de composant applicatif. Cette instance indique la racine de l'arborescence de tâches qui s'exécuteront sur ce processeur. Ceci constitue une grande différence par rapport au méta-modèle *Gaspard* : implicitement cela permet de ne pas avoir de modèle d'association. Avec le méta-modèle *Gaspard* l'application est une grande arborescence. À l'inverse, dans le méta-modèle *Polyhedron*, chaque instance de processeur (éventuellement répétée) contient sa propre arborescence de tâches (équivalente à une partie de l'application globale). Cette représentation se rapproche de la cible en permettant de spécifier pour chaque processeur un code différent à exécuter.

L'expression de l'espace de répétition n'est pas modélisée par une *Shape* (un vecteur d'entiers) comme c'est le cas dans le méta-modèle *Gaspard* mais par un *Polyhedron*. Nous allons détailler cette représentation dans la section suivante.

5.1.3 Expression de la répétition des tâches

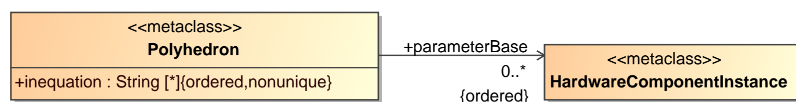


FIG. 5.4: Représentation d'un polyèdre entier paramétré par les indices de répétition de composants matériels. Cela permet de représenter fidèlement l'espace de répétition d'une tâche distribuée.

Le fait que les tâches soient placées directement sur un processeur influence énormément

la manière d'exprimer l'espace de répétition des tâches distribuées. Dans le méta-modèle *Gaspard* elles sont représentées par une *Shape*, qui spécifie un espace de répétition dont les limites sont parallèles aux axes, et par une *Distribution*, qui associe une partie de l'espace à chaque processeur. Cependant, si la distribution est une notion commode à manipuler par le concepteur, elle n'est pas facile à utiliser lors de l'exécution du code. En effet, pour connaître le placement décrit par une *Distribution* en utilisant directement les informations spécifiées, il faut parcourir complètement l'espace de répétition et au fur et à mesure noter les liens entre processeurs et indice de répétition de tâche. Afin d'être plus proche du code généré, il faut pouvoir représenter explicitement les indices de répétition à exécuter sur un processeur donné. Pour cela nous proposons l'usage de polyèdres entiers, paramétrés par les indices du processeur. La figure 5.4 présente les concepts utilisés.

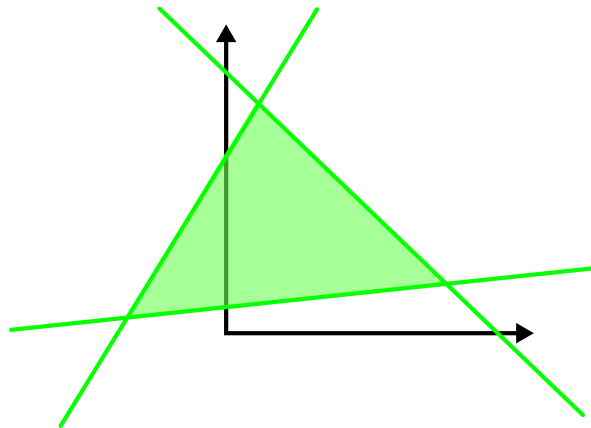
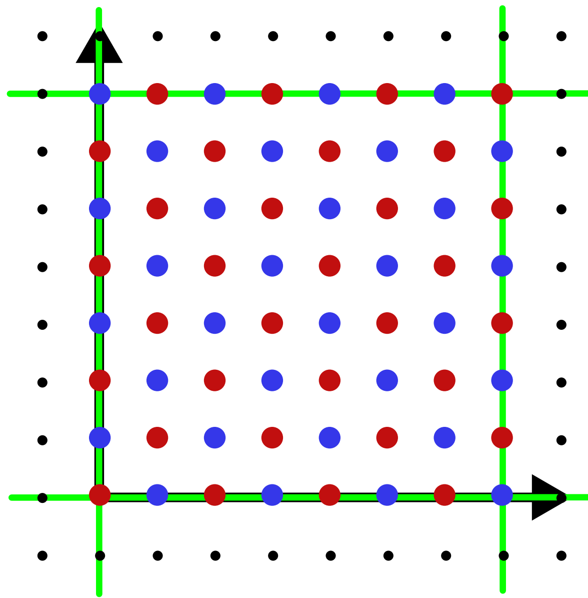


FIG. 5.5: Exemple de polyèdre, en vert clair, défini par trois inéquations affines, en vert.

Avant d'aller plus loin, expliquons ce que nous entendons par *polyèdre entier paramétré*. Mathématiquement, un polyèdre est un ensemble convexe fermé délimité par des inéquations affines. Cette définition est indépendante du nombre de dimensions de l'espace dans lequel le polyèdre est défini. Dans un espace bidimensionnel (un plan), les inéquations sont des droites définissant un demi-plan. La figure 5.5 montre un exemple d'un tel polyèdre, définissant un espace triangulaire. Similairement, dans un espace tridimensionnel les inéquations sont des plans. Un polyèdre entier est l'ensemble des points situés à des coordonnées entières qui vérifient toutes les inéquations du polyèdre. Lorsque le polyèdre entier est en plus *paramétré*, un certain nombre d'équations dépendantes de variables supplémentaires appelées *paramètres* sont ajoutées aux inéquations. Cela réduit l'ensemble des points du polyèdre à un sous-ensemble dépendant du paramètre. La figure 5.6a présente un exemple de tel polyèdre. Le polyèdre entier qui correspond à tous les points présents dans le carré délimité par les inéquations est séparé en deux ensembles (points rouges et bleus), selon la valeur du paramètre. Il est exprimable mathématiquement par la formule figure 5.6b

Pour exprimer l'espace de répétition d'une tâche à l'aide d'un polyèdre nous définissons d'une part que les coordonnées de chaque point du polyèdre correspondent aux indices de répétition de la tâche et d'autre part que les paramètres varient selon les indices de répétition des composants matériels sur lesquels elle est placée. La classe *Polyhedron* permet de modéliser le polyèdre : l'attribut `inequation` contient des chaînes de caractères, chacune représentant



(a) Le polyèdre est défini par quatre inéquations en vert et des équations affines. Le paramètre ne peut avoir que deux valeurs, les points correspondant à la première valeur sont en bleu, tandis que ceux en rouge correspondent à la seconde valeur.

$$\begin{cases} p_0 \leq 0, 1 - p_0 \leq 0 \\ -2mh_0 - p_0 + x_0 + x_1 = 0 \\ x_0 \leq 0, 7 - x_0 \leq 0 \\ x_1 \leq 0, 7 - x_1 \leq 0 \end{cases}$$

(b) Le polyèdre est défini par cet ensemble d'équations et d'inéquations. Les variables p sont les paramètres, la variable mh sert à représenter le modulo, et les variables x sont les inconnues, c'est-à-dire les indices de l'espace de répétition.

FIG. 5.6: Exemple de polyèdre entier paramétré, sous forme géométrique et sous forme de système d'équations et d'inéquations.

une inéquation ou une équation du polyèdre. Un certain nombre de contraintes sont imposées sur la syntaxe de la chaîne de caractères. Une seule équation ou inéquation peut être exprimée par chaîne, les variables dont le nom commence par un *x* sont les variables de l'espace de répétitions, les variables dont le nom commence par un *p* sont les variables des paramètres.

Il eut été possible de modéliser plus finement le polyèdre en proposant des classes pour chaque concept d'une équation, néanmoins, cela aurait complexifié nettement le modèle sans apporter d'avantage puisque, comme nous le verrons par la suite, le polyèdre est traité d'un seul bloc par une application externe. L'attribut `parameterBase` permet, dans le cas où la tâche est distribuée selon plusieurs niveaux de hiérarchie de l'architecture, d'indiquer quelle variable de paramètre dans les équations correspond à quel composant répété de l'architecture. Le premier indice du premier composant correspondra à p_0, l'indice suivant à $p_1 \ldots$ Cela est nécessaire car seuls les niveaux de hiérarchie auxquels la tâche a été explicitement distribuée influencent le placement des répétitions.

Toutes les distributions de tâche répétées peuvent être représentées sous forme polyédrique (et qu'il n'y a par conséquent pas de perte d'information). La démonstration sera faite par construction dans la section suivante, traitant de la transformation de modèles *Gaspard* en modèles *Polyhedron*.

5.1.4 Les tableaux de données

Similairement aux tâches, les tableaux de données sont explicitement placés sur les instances de mémoire comme illustré dans la figure 5.7 Avec le méta-modèle *Gaspard* les données

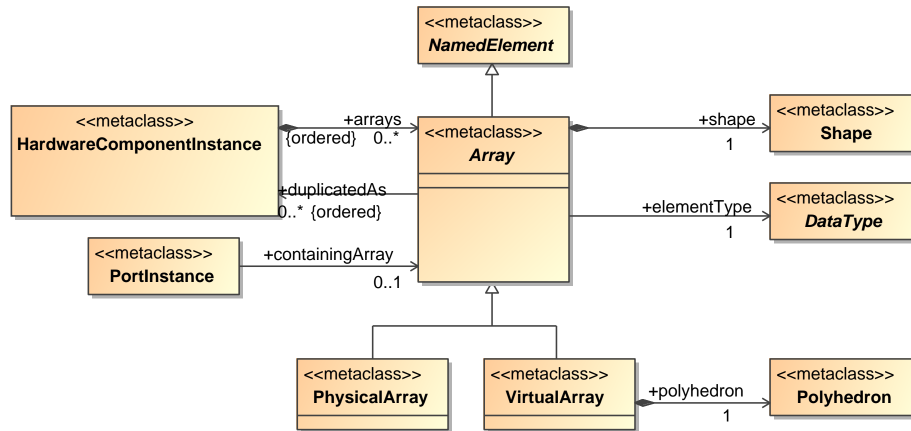


FIG. 5.7: Concepts nécessaires à la représentation des tableaux de données et à leur distribution sur des bancs mémoire.

sont implicitement exprimées par la taille des ports des tâches et par l'allocation de ces ports sur des mémoires. Les concepts du méta-modèle *Polyhedron* permettent de représenter ces données telles qu'elles seront utilisées dans le code généré. Ainsi une *HardwareComponentInstance* d'un composant matériel mémoire peut contenir des *Arrays* via l'attribut `arrays`. Cet attribut est ordonné, cela permet de situer précisément un tableau à l'intérieur d'une mémoire. En prenant en compte le nombre, la taille et le type de chacun des tableaux le précédant parmi l'ensemble des tableaux, il est possible de déterminer son adresse dans la mémoire.

Les informations de taille et de type sont associées à un *Array* par les attributs `shape` et `elementType`. Ils doivent être identiques à la taille et au type des ports des tâches qui utilisent le tableau. Cette notion d'utilisation est indiquée par l'attribut `containingArray` d'une *PortInstance*. L'accès à des données à travers la *PortInstance* correspond à la lecture ou l'écriture dans le tableau représenté par l'*Array*. Comme nous l'avons vu dans la définition du modèle d'exécution des tâches Gaspard, toutes les *PortInstances* ne correspondent pas à un accès mémoire : lorsqu'elles sont liées à un *Port* d'un composant englobant elles ne font que déléguer l'accès vers une autre *PortInstance*. Pour cette raison, les *PortInstances* qui ne sont connectées que par des *ApplicationTilers* ne nécessitent pas d'avoir un `containingArray`.

L'attribut `duplicatedAs` correspond à une notion complètement implicite dans le méta-modèle *Gaspard* : il indique le nombre de processeurs qui peuvent avoir besoin en même temps du tableau (pour transmettre des données différentes). En effet, chaque tableau correspond à la transmission de données entre deux tâches. Si les tâches sont distribuées sur plusieurs processeurs, il peut y avoir lieu en même temps autant de transmissions que de processeurs. Il est donc nécessaire dans ce cas que le tableau soit dupliqué selon le processeur. `duplicatedAs` fonctionne comme l'attribut `parameterBase` : il désigne une liste ordonnée de composants matériels répétés. La concaténation des *shapes* de chaque composant correspond à la duplication complète du tableau.

Si le tableau est alloué sans distribution, alors il doit être du type *PhysicalArray*. Lorsque l'on représente la distribution d'un tableau sur plusieurs répétitions d'une mémoire, il faut utiliser un *VirtualArray*. Similairement aux tâches, ces tableaux possèdent un attribut

`polyhedron` permettant de déterminer à partir de ses coordonnées les répétitions de la mémoire qui vont contenir l'élément du tableau. L'usage diffère légèrement puisque les paramètres du polyèdre sont les coordonnées de l'élément du tableau et que les points parcourus par le polyèdre correspondent aux indices de répétition de la mémoire. En conséquence, à moins que les données ne soient dupliquées, le polyèdre ne contiendra qu'un seul point.

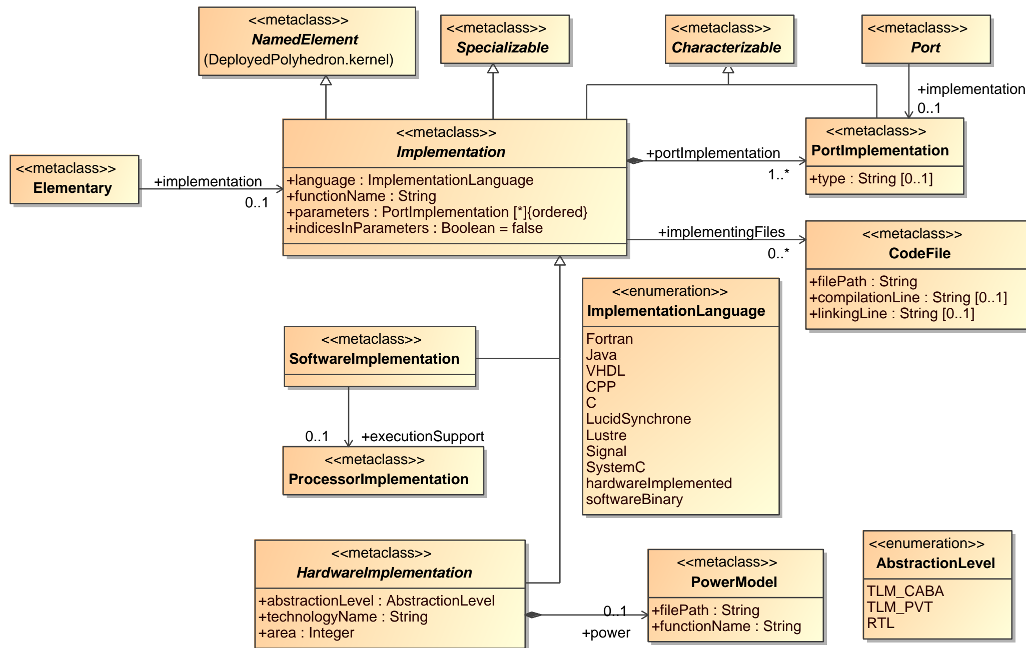


FIG. 5.8: Les concepts de déploiement dans le méta-modèle *Polyhedron*. On note principalement l'absence de la notion d'implémentation abstraite. Un composant élémentaire ne peut correspondre qu'à un seul IP.

5.1.5 Le déploiement simplifié

La dernière partie qui distingue les méta-modèles *Polyhedron* et *Gaspard* est la partie contenant les concepts de déploiement présentée dans la figure 5.8. Une des contraintes principales du déploiement, présentée dans le chapitre 3, est la nécessité d'associer plusieurs implémentations différentes à une fonctionnalité. Cela permet de sélectionner l'implémentation la plus adaptée à la cible de compilation. À partir du moment où une chaîne de transformations est exécutée, la cible est connue, il est donc possible de faire la sélection et de ne garder que la meilleure implémentation. Ainsi le méta-modèle *Polyhedron* permet de lier directement une implémentation donnée à un composant. Principalement, la notion de *AbstractImplementation* disparaît par rapport au méta-modèle *Gaspard*.

De plus, il devrait être possible de créer des bibliothèques de composants indépendantes des modèles de SoC, d'où la présence des concepts d'*ImplementedBy* et de *PortImplementedBy*. Ceux-ci pouvant porter des `characteristics` et des `specializations` pour paramétrer un IP uniquement au moment de la conception du SoC. Au cours de la chaîne de transformation, toutes les bibliothèques de composants peuvent être intégrées au modèle de SoC,

ces mécanismes d'indirection sont alors inutiles. Les paramètres peuvent être tous placés directement sur les implémentations.

Les concepts de déploiement sont donc relativement simples, comparés au méta-modèle *Gaspard*, avec seulement deux niveaux. En outre, les composants élémentaires, c'est-à-dire ceux ayant une structure interne *Elementary*, sont liés directement à une *Implementation* via l'attribut `implementation`, tout comme les *Ports* sont liés directement à un *PortImplementation*. Notons que les composants élémentaires doivent être obligatoirement liés à une implémentation (sans quoi il ne sera pas possible de générer complètement le code).

Selon le type du composant, l'implémentation est soit une *SoftwareImplementation* soit une *HardwareImplementation*. Chaque implémentation peut correspondre à un ou plusieurs *CodeFiles*.

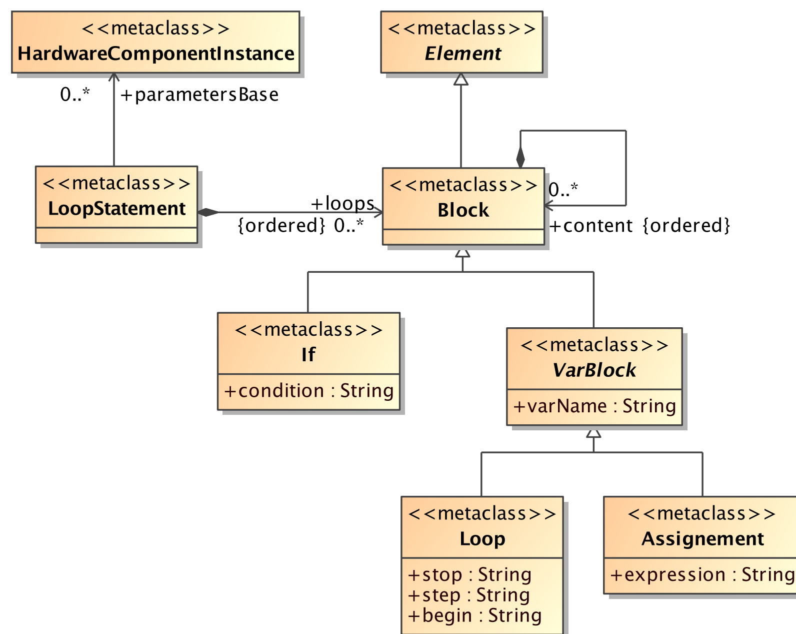


FIG. 5.9: Les concepts utilisés par le méta-modèle *Loop* pour représenter la structure de code correspondant à un polyèdre.

5.1.6 Le méta-modèle *Loop*

Le méta-modèle *Loop* est le second méta-modèle intermédiaire. Développé par Julien Taillard, doctorant dans l'équipe, il est très proche du méta-modèle *Polyhedron*. L'unique différence porte sur la représentation de la répétition des tâches. À la place d'utiliser un polyèdre, la répétition est représentée par un *LoopStatement*. Il correspond à la structure du pseudo-code qui, pour un indice de processeur donné, parcourt tous les indices de répétitions de la tâche associés. Cette représentation est ce qu'il y a de plus proche du code généré final. Dans notre chaîne de transformations, le traitement qui permet de passer d'une représentation à l'aide d'un polyèdre à une représentation sous forme de pseudo-code est effectué par un outil externe (CLooG [15]). Ceci est la principale raison de la présence de ces deux méta-

modèles intermédiaires très proches : l'appel externe peut être complètement découplé du reste des transformations.

La figure 5.9 présente le détail d'un *LoopStatement*. Il est constitué d'une suite ordonnée de *Blocks*, eux-mêmes pouvant contenir une suite ordonnée de *Blocks*. Un *Block* représente une instruction du pseudo-code, qui peut être soit un *If* (une condition), un *For* (une boucle), ou un *Assignement* (l'affectation d'une valeur à une variable). Chaque attribut de ces *Blocks* contient une expression arithmétique ou une expression de comparaison en pseudo-code. Les variables utilisées sont les mêmes que dans les polyèdres du méta-modèle *Polyhedron*. Chaque feuille de l'arborescence des *Blocks* correspondent à des points du code où les variables `x` représentent l'indice d'une répétition de la tâche.

5.1.7 Synthèse

Dans cette section nous avons présenté les deux méta-modèles intermédiaires utilisés dans la chaîne de transformations. Ils représentent de manière explicite la syntaxe des entrées et sorties qui vont être manipulées par les différentes transformations. Ils ont été définis de manière à permettre la représentation du SoC sous une forme aussi proche que possible des cibles de compilation tout en gardant suffisamment d'abstraction pour être compatibles avec chacune d'entre-elles. Ainsi la partie de compilation commune à toutes les cibles a pu être factorisée en une séquence de deux transformations de modèles.

Le méta-modèle *Polyhedron* est basé sur le méta-modèle *Gaspard* mais supprime la notion d'association. À la place les tâches sont représentées directement sur les processeurs et les tableaux de données sont spécifiés sur les mémoires. Pour représenter sans perte d'information la distribution des données ou des tâches, le concept de polyèdre entier paramétré est employé. En plus de cela, les notions de déploiement ont été largement simplifiées, de manière à ce que chaque composant élémentaire soit lié directement à l'IP qui sera utilisé lors de la génération de code. Quant au méta-modèle *Loop*, il est très proche de cette représentation mais troque la notion de polyèdres par une arborescence représentant un nid de boucles qui parcourent les indices de répétitions des tâches ou des mémoires selon la distribution définie par le concepteur.

Lors du développement, les deux méta-modèles intermédiaires ont été spécifiés une fois le méta-modèle *Gaspard* complété et les cibles de compilation déterminées mais *avant* l'écriture des transformations. Au cours de l'écriture des transformations ils ont été raffinés, principalement pour corriger quelques erreurs de modélisation. Cela n'a néanmoins pas empêché de développer en parallèle les différentes transformations de modèle, à la fois celles permettant de passer d'un modèle *Gaspard* à un modèle *Loop* que celles permettant de générer un modèle cible. Dans les deux sections suivantes nous allons détailler les différentes transformations allant du modèle de SoC au code de simulation en SystemC.

5.2 Du méta-modèle *Gaspard* vers le méta-modèle *Polyhedron*

La première transformation de la chaîne de compilation permet de passer du modèle de SoC, dessiné par le concepteur, à un modèle intermédiaire *Polyhedron* équivalent. Cette transformation, développée dans le cadre de cette thèse, suit les recommandations de l'IDM. Nous allons la détailler dans cette section. Tout d'abord, nous présenterons l'infrastructure de développement spécialement dédiée à la définition de transformations sous forme de règles sur laquelle nous nous sommes basés. Par la suite nous détaillerons les algorithmes

mis en œuvre pour obtenir un modèle *Polyhedron* équivalent au modèle d'entrée, à savoir : l'expression des répétitions sous forme de polyèdres, le découpage de l'arbre d'application selon l'association, le placement des tableaux de données et la simplification du déploiement.

5.2.1 Moteur de transformations modèle à modèle

Pour générer un modèle à partir d'un autre modèle, l'IDM préconise l'usage d'une *transformation de modèles*. En plus d'avoir ses modèles d'entrée et de sortie explicitement conforment à des méta-modèles spécifiques, une transformation de modèles se distingue par le fait de n'être constituée que par un ensemble de *règles de transformation*. Chaque règle est relativement indépendante des autres. Pour pouvoir exécuter les règles, il faut un moteur de transformations. Comme nous l'avons vu dans le chapitre 2, une règle est composée d'un patron d'entrée, d'un patron de sortie, et une partie *logique* établissant la liaison entre ces deux patrons.

Les constats issus de l'utilisation des moteurs de transformations existants ont amené l'équipe à proposer MoMoTE (Model to Model Transformation Engine), afin de satisfaire aux exigences des différents concepteurs des transformations de l'environnement Gaspard. Bien que d'aspect rudimentaire par rapport à d'autres moteurs, MoMoTE a l'avantage de permettre de mélanger les approches de programmation impérative et déclarative. En particulier, dans la partie logique d'une règle il est possible de faire appel à des bibliothèques ou des programmes extérieurs, vus comme des boîtes noires.

MoMoTE est utilisé comme une API Java, et repose sur les bibliothèques Ecore et EMFT Query [37] pour manipuler les modèles et exprimer les patrons d'entrée et de sortie des règles. Ce moteur permet de transformer N modèles d'entrée en M modèles de sortie. Chaque règle est représentée par une classe. Un avantage sur les autres moteurs, MoMoTE autorise d'appliquer le mécanisme d'héritage sur les règles, ce qui permet la factorisation des points communs de différentes règles.

Une règle est définie à l'aide de cinq méthodes qui sont appelées par le moteur de transformations aux différentes phases de l'exécution :
– le constructeur
– `getCondition()`
– `create()`
– `process()`
– `processReferences()`

Nous allons détailler la sémantique de chacune de ces méthodes en les illustrant avec une règle simple. Rappelons que dans la section précédente nous avions décrit la partie matérielle du méta-modèle *Polyhedron* comme étant identique à celle du méta-modèle *Gaspard*. La partie de la transformation se chargeant de la représentation architecture matérielle est donc simplement une conversion un à un des concepts. Ainsi la règle permettant de générer une instance de composant matériel est écrite de la manière suivante :

```
1  public class GaspardHI2PolyhedronHI extends Rule{

       public GaspardHI2PolyhedronHI()
       {
5          super();
           setQueryFromContext(true);
           addRule(DeployedpolyhedronPackage.eINSTANCE.getComponentInstance_Dim(),
                   new Gaspard2Shape2PolyhedronShape());
           addRule(DeployedpolyhedronPackage.eINSTANCE.getComponentInstance_PortInstance(),
10                 new Gaspard2PortInstance2PolyhedronPortInstance());
       }
```

```
    protected EObjectCondition getCondition(EObject srcElementContext)
    {
15      return new EObjectTypeRelationCondition(
                Gaspard2Package.eINSTANCE.getHardwareComponentInstance());
    }

    protected EObject create(EObject srcElement)
20  {
        return DeployedpolyhedronFactory.eINSTANCE.createHardwareComponentInstance();
    }

    protected void process(EObject srcElement, EObject tgtElement)
25  {
        HardwareComponentInstance hci=(HardwareComponentInstance) srcElement;
        fr.lifl.west.gaspard2.metamodel.deployedpolyhedron.HardwareComponentInstance shci=
          (fr.lifl.west.gaspard2.metamodel.deployedpolyhedron.HardwareComponentInstance) tgtElement;
        shci.setName(hci.getName());
30  }

    protected void processReferences(EObject srcElement, EObject tgtElement)
    {
        HardwareComponentInstance hci=(HardwareComponentInstance) srcElement;
35      fr.lifl.west.gaspard2.metamodel.deployedpolyhedron.HardwareComponentInstance shci=
          (fr.lifl.west.gaspard2.metamodel.deployedpolyhedron.HardwareComponentInstance) tgtElement;
        shci.setComponent(((fr.lifl.west.gaspard2.metamodel.deployedpolyhedron.HardwareComponent)
          getTransformation().getGlobalRefs().get(hci.getComponent())));

40  }
}
```

Le constructeur , ici nommé `GaspardHI2PolyhedronHI()`, indique si la règle doit être appliquée sur tout le modèle ou uniquement sur les éléments traités par la règle supérieure (avec `setQueryFromContext()`). En plus, il déclare les sous-règles à l'aide la méthode `addRule()` qui prend en premier paramètre l'emplacement auquel ajouter l'élément créé par la règle spécifiée dans le second paramètre.

`getCondition()` spécifie le patron d'entrée de la règle à l'aide d'une condition EMFT Query (utilisant une approche similaire aux requêtes SQL). Dans l'exemple, la condition ne sélectionne que les éléments de type *HardwareComponentInstance* du méta-modèle *Gaspard*.

`create()` permet de créer l'élément racine du patron de sortie de la règle. Ici, la règle crée une *HardwareComponentInstance* du méta-modèle *Polyhedron*. C'est un des désavantages de la version actuelle de MoMoTE, seul l'élément racine du patron de sortie est déclaré explicitement. Si la sortie contient plus d'éléments, ils doivent être créés « manuellement » au cours de l'appel à l'une des deux méthodes suivantes.

`process()` correspond à la partie logique de la règle. Elle est appelée pour chaque partie du modèle validant la condition donnée précédemment. Elle peut inclure autant de calculs nécessaires pour générer la sortie à partir des informations contenues dans l'élément d'entrée courant. Dans l'exemple présenté, la règle lit le nom de l'instance Gaspard et le recopie tel quel comme nom de l'instance Polyhedron.

`processReferences()` correspond à la partie logique de la règle traitant des références. Elle est appelée exactement de la même manière que `process()`, mais *après* que toutes les règles de la transformation aient déjà été exécutées une fois. À ce moment de l'exécution, tous les éléments du modèle de sortie sont créés, ce qui permet de créer les références entre eux. Dans l'exemple, la méthode recherche le composant du modèle *Gaspard* référencé dans

l'attribut `component`. À l'aide de `getGlobalRefs()`, le composant équivalent dans le modèle *Polyhedron* est retrouvé et une référence vers celui-ci est ajoutée à l'instance courante.

Pour conclure cette rapide présentation du moteur de transformations, notons qu'au cours du développement des évolutions ont été apportées à la fois au méta-modèle d'entrée et au méta-modèle de sortie. La mise à jour de la transformation a toujours été aisée, en particulier grâce au fait qu'il soit très facile de faire le lien entre la modification d'une partie du méta-modèle et les règles impliquées par cette modification.

5.2.2 Génération de polyèdres pour exprimer la répétition

Une des différences notables entre le méta-modèle *Gaspard* et celui *Polyhedron* est le changement de représentation de la répétition des tâches. Dans le méta-modèle intermédiaire, l'usage de polyèdres entiers permet d'exprimer l'espace de répétition même lorsque la tâche est distribuée sur plusieurs processeurs.

5.2.2.1 Construction mathématique

La génération du polyèdre se fait dans la règle transformant les instances de composants applicatifs. À partir d'une instance, il est possible de retrouver sa *Shape*, son espace de répétition, et la *Distribution* qui est appliquée. Depuis cette *Distribution*, on peut retrouver directement l'ensemble de processeurs sur lesquels la tâche est placée et indirectement on retrouve sa *Shape*. Ce sont toutes les informations nécessaires à la création du polyèdre. Pour résumer, voici les informations disponibles :

- sourceTiler (O_{sw}, P_{sw}, F_{sw})
- repetitionSpace, patternShape
- targetTiler (O_{hw}, P_{hw}, F_{hw})
- espace de répétition de l'application (m_{sw})
- espace de répétition des processeurs (m_{hw})

L'usage du polyèdre doit permettre de déterminer les points $\overrightarrow{T_p}$, faisant partie d'espace de répétition de l'application en fonction de l'indice du processeur $\overrightarrow{P}$, appartenant à l'espace de répétition des processeurs.

Une fois ces données posées, il suffit de fusionner l'ensemble des définitions concernant ces concepts : les définitions des *Tilers* (cf formule 2.3, p. 40), la définition de l'espace de répétition de la distribution, celle de la forme du motif, et enfin les définitions des espaces de répétitions de l'application et des processeurs. On obtient alors le système d'équations et d'inéquations suivant :

$$\begin{cases} \overrightarrow{T_p} + m_{sw}.\overrightarrow{Q1} = O_{sw} + P_{sw}.\overrightarrow{r} + F_{sw}.\overrightarrow{i} \\ \overrightarrow{P} + m_{hw}.\overrightarrow{Q2} = O_{hw} + P_{hw}.\overrightarrow{r} + F_{hw}.\overrightarrow{i} \\ 0 \leq \overrightarrow{r} < repetitionSpace \\ 0 \leq \overrightarrow{i} < patternShape \\ 0 \leq \overrightarrow{P} < m_{sw} \\ 0 \leq \overrightarrow{T_p} < m_{hw} \end{cases}$$

Où l'on a introduit deux variables supplémentaires $Q1$ et $Q2$ qui permettent de représenter la notion de modulo des *Tilers*. Aucune contrainte ne portant sur ces variables, elles peuvent prendre n'importe quelles valeurs entières.

Toutes ces équations et inéquations sont affines. Une fois répliquées pour représenter chaque dimension des vecteurs, on obtient le polyèdre recherché. Notons au passage, que cette construction permet de démontrer qu'il est possible de trouver un polyèdre pour n'importe quelle *Distribution*.

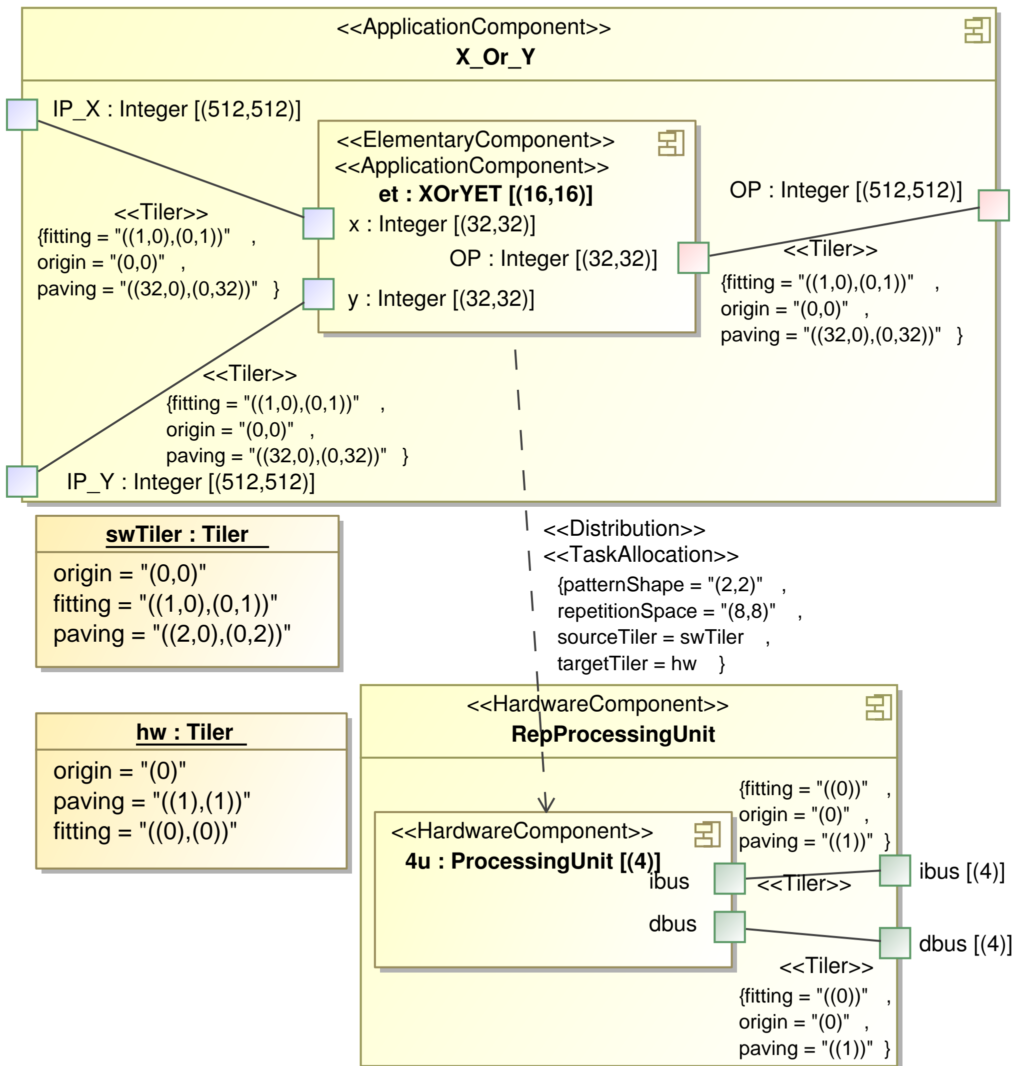


FIG. 5.10: Exemple de distribution d'une tâche répétée 16×16 sur quatre processeurs.

5.2.2.2 Exemple de construction à partir d'une distribution

Pour mettre en pratique cette construction de polyèdre, nous allons calculer le polyèdre correspondant à l'espace de répétition de la tâche distribuée telle que dans la figure 5.10. La distribution de cette tâche répétée 16×16 répartie également les nombres de tâches sur chacun des quatre processeurs. La figure 5.11 illustre cette distribution en associant une couleur différente à chaque indice de tâche selon le processeur attribué.

En construisant le polyèdre selon la méthode définie précédemment on obtient ce système :

$$\begin{cases} p_0 \leq 0, 3 - p_0 \leq 0 \\ -4 * mh_0 - p_0 + 1 * q_0 + 1 * q_1 + 0 * d_0 + 0 * d_1 + 0 = 0 \\ -16 * ms_0 - x_0 + 2 * q_0 + 0 * q_1 + 1 * d_0 + 0 * d_1 + 0 = 0 \\ -16 * ms_1 - x_1 + 0 * q_0 + 2 * q_1 + 0 * d_0 + 1 * d_1 + 0 = 0 \\ q_0 \leq 0, 7 - q_0 \leq 0 \\ q_1 \leq 0, 7 - q_1 \leq 0 \\ d_0 \leq 0, 1 - d_0 \leq 0 \\ d_1 \leq 0, 1 - d_1 \leq 0 \\ x_0 \leq 0, 15 - x_0 \leq 0 \\ x_1 \leq 0, 15 - x_1 \leq 0 \end{cases}$$

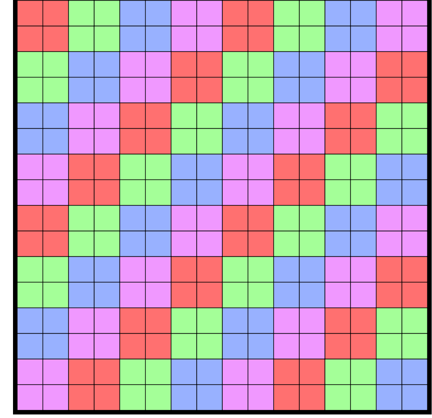


FIG. 5.11: Exemple de distribution d'une tâche répétée 16×16 sur quatre processeurs. Chaque carré représente une répétition de la tâche, sa couleur correspond au processeur sur lequel elle est allouée.

5.2.2.3 Polyèdre lors d'une allocation simple

Nous venons de voir le fonctionnement de la règle de transformation lorsque l'allocation est une distribution. Lorsqu'une allocation simple est utilisée ou qu'aucune allocation n'est placée sur le composant applicatif, le polyèdre généré doit simplement parcourir l'ensemble de l'espace de répétition de la tâche (défini par sa *Shape*). Dans ce cas le polyèdre est défini par un système d'inéquation (trivial) de la forme : $0 \leq \overrightarrow{T_p} < m_{hw}$.

5.2.3 Découpage de l'arborescence des tâches

L'introduction des polyèdres est requise par la suppression des concepts d'association. Le deuxième effet de cette suppression est que dans un modèle *Polyhedron* l'arbre d'application est partagé entre les différents processeurs. Cette différence est illustrée dans la figure 5.12.

La règle correspondant à cette partie de la transformation est exécutée en utilisant comme entrée l'application déjà générée dans le modèle *Polyhedron*. La règle ne modifie que la structure du modèle. L'algorithme de cette partie de la transformation commence par repérer chaque tâche d'origine d'une branche : la tâche la plus basse d'une branche de l'arborescence qui soit explicitement allouée sur un processeur. Pour chaque tâche d'origine trouvée, la règle de transformation parcourt le chemin allant de la racine de l'application à la tâche d'origine. Chaque nœud rencontré est recopié sur le processeur où est allouée la tâche d'origine. Si un nœud est déjà présent sur le processeur, parce qu'une autre tâche partage une partie du chemin, alors il est directement utilisé, sans faire une nouvelle copie. Ainsi, au final chaque processeur contient une arborescence, correspondant à une partie de l'arbre d'application. Une fois l'algorithme terminé, les éléments d'origine sont supprimés. Remarquons que les tâches en dessous de la tâche d'origine ne sont pas recopiées ni supprimées, elles sont directement utilisées depuis la copie de la tâche d'origine.

Cette partie de la transformation est agrémentée d'un ensemble de fonctions qui ne génèrent aucun élément : elles ont pour rôle unique de valider l'association spécifiée par l'utilisateur et de produire selon les erreurs des messages idoines.

Après que toute l'application ait été déplacée sur les différents processeurs, la génération des connecteurs est effectuée. Les connecteurs simples, qui représentent la dépendance de

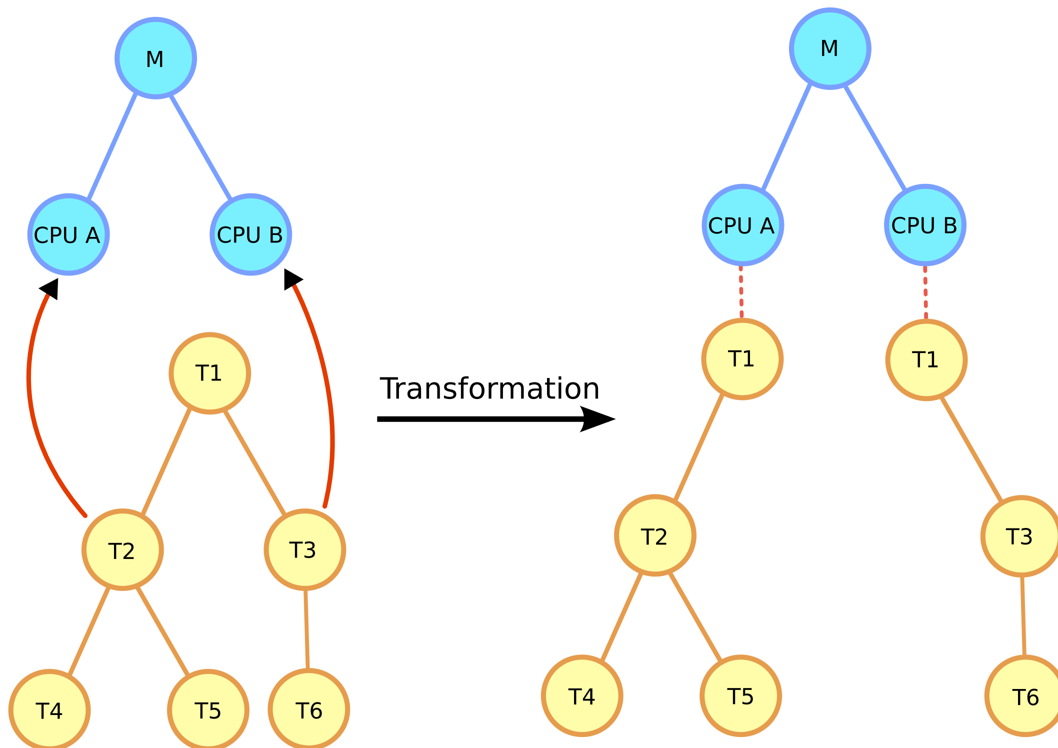


FIG. 5.12: Effets de la transformation sur la structure arborescente de l'application. Chaque processeur ne contient que les tâches qui y ont été allouées.

données entre tâches, sont créés entre les mêmes ports que dans le modèle d'entrée. Si un connecteur relie une tâche A à une tâche B qui est présente sur plusieurs processeurs dans le modèle *Polyhedron*, alors le connecteur est dupliqué de manière à relier chaque tâche A à l'ensemble des tâches B. Concernant les *Tilers*, ils sont générés selon le modèle d'entrée afin de relier les ports d'une sous-tâche à ceux de la tâche contenante. Ils ne relient que tâches et sous-tâches qui sont sur le même processeur.

Cette partie de la transformation qui se charge du découpage de l'arborescence est implémentée en deux phases. D'abord un ensemble de règles se charge de générer à partir de chaque élément de l'application Gaspard, un élément équivalent dans le modèle *Polyhedron*. Puis, le découpage de l'arbre à proprement dit est effectué en modifiant le modèle *Polyhedron* déjà créé. Actuellement ces deux phases sont effectuées au cours la même transformation. Néanmoins a posteriori, il semble qu'il aurait été plus approprié d'implémenter la deuxième phase explicitement dans une seconde transformation de type endogène (ayant les modèles d'entrée et de sortie conforment au même méta-modèle). Cette évolution permettra de séparer finement la seconde phase en plusieurs règles et sous-règles.

5.2.4 Placement des tableaux de données

Le troisième et dernier effet de la suppression de l'association sur le méta-modèle *Polyhedron* est la présence explicite de tableaux de données sur les instances de mémoires. Dans le méta-modèle *Gaspard* il n'existe pas de concepts propres aux tableaux, ils sont implicitement

présents entre les instances de composants applicatifs reliées par des connecteurs.

La règle générant les tableaux de données est appliquée à chaque instance de port connectée à une autre instance de port de composant applicatif. Si une allocation a été spécifiée spécialement pour l'instance, alors un tableau est directement créé sur l'instance de mémoire indiquée[1]. Autrement, chaque instance de port correspondant au même tableau (y compris dans le cas où il existe plusieurs lecteurs) est inspectée deux fois. La première inspection recherche si un tableau a déjà été créé, auquel cas ce tableau est réutilisé. La seconde inspection recherche si une allocation de donnée a été spécifiée, auquel cas elle est suivie lors de la création du tableau. En dernier ressort, une allocation de donnée est recherchée sur chacune des tâches parents.

Ainsi, à moins que plusieurs allocations différentes ne soient expressément spécifiées dans l'association, un seul tableau est créé par groupe de connecteurs correspondant à une même sortie. Notons que lors de la création du tableau, si la tâche contenant le connecteur est une tâche d'origine de distribution ou plus bas, c'est-à-dire qu'elle pourra être exécutée sur plusieurs processeurs en même temps, alors la référence `duplicatedAs` du tableau est associée aux processeurs pointés par la distribution de tâche.

5.2.5 Simplification du déploiement

Dans le modèle *Gaspard*, afin de donner de la souplesse à la modélisation et de permettre les bibliothèques d'IP, il est possible de placer les `characteristics` et les `specializations` sur de nombreuses classes différentes, bien qu'elles n'affectent réellement que les *Implementations* et les *PortImplementations*. Plus précisément, les paramètres placés sur un *ImplementedBy* ou une *AbstractImplementation* influent sur une *Implementation*, et ceux placés sur un *PortImplementedBy* influent sur une *PortImplementation*. Pour pouvoir prendre en compte toutes ces informations lors de la génération d'une implémentation ou d'une implémentation de port dans le modèle *Polyhedron*, la transformation génère ces éléments respectivement à partir des *ImplementedBy* et *PortImplementedBy*. Les différents niveaux sont parcourus et l'ensemble des paramètres trouvés est généré sur l'*Implementation* ou la *PortImplementation* du modèle *Polyhedron*.

Lorsque plusieurs implémentations sont disponibles à l'intérieur d'une même implémentation abstraite, une fonction de sélection est appelée afin de choisir l'implémentation la plus adaptée à la cible de compilation. Actuellement, seul le langage de l'implémentation est pris en compte pour déterminer l'adéquation, mais bien entendu il est envisageable de perfectionner cette fonction pour tenir compte de plus de critères (tels le temps d'exécution, la consommation. . .).

Le fait de ne générer les implémentations qu'à partir des *ImplementedBy* et *PortImplementedBy* a par ailleurs l'avantage de ne produire dans le modèle de sortie que les implémentations réellement utilisées par les composants élémentaires du SoC. Même si la présence d'implémentations non utilisées provenant des bibliothèques d'IP n'empêcherait pas la compilation vers la cible, elle gênerait à la lecture du modèle et pourrait augmenter le temps d'exécution des transformations.

[1]Plus exactement, le tableau est créé sur l'instance *équivalente* dans le modèle *Polyhedron* à l'instance mémoire indiquée.

5.2.6 Synthèse

Nous venons de décrire les grands traits la transformation entre le méta-modèle *Gaspard* et le méta-modèle *Polyhedron*. Basée sur le moteur de transformation MoMoTE, elle est composée de 59 règles, chacune chargée de générer un type d'élément du modèle de sortie. Alors que le modèle d'entrée est fortement séparé entre les parties correspondant aux paquetages d'application, d'association, d'architecture matérielle, et de déploiement, le modèle de sortie forme réellement un tout qui va être ensuite beaucoup plus facile de transformer pour obtenir un code représentant le SoC complet.

La validation de cette transformation a été faite de manière expérimentale, au cours du développement. Trois modèles de MPSoC plus ou moins complexes ont été utilisés pour la tester. Dans la pratique, similairement au développement d'un compilateur, l'étape de validation s'est déroulée par de nombreux cycles de test et de correction. Les différents paliers d'avancement de la validation comprennent :

- l'exécution complète de la transformation sur les différents modèles de test (sans interruption due à une erreur dans le code).
- la vérification de la conformité du modèle de sortie avec son méta-modèle. En effet, certains attributs ou certaines références requises selon le méta-modèle peuvent être manquants.
- la vérification manuelle du modèle de sortie par rapport aux attentes à partir du modèle d'entrée.
- la vérification que la transformation suivante fonctionne avec le modèle de sortie, ou en d'autres termes que la sémantique associée au méta-modèle de sortie soit bien la même.
- la vérification que la chaîne de transformations complète fonctionne, en particulier il s'agit de s'assurer que *toute* l'information contenue dans le modèle de départ ait bien été traitée.

À l'heure actuelle, la transformation a été utilisée sur un grand nombre de modèles, puisqu'elle est un des points clefs de la chaîne de transformations de l'environnement Gaspard. À l'exception de quelques restrictions connues sur le modèle d'entrée (tel que la gestion des connecteurs *Reshapes* dans l'application), il n'y a pas de problèmes connus.

Dans la section suivante nous nous intéresserons à la seconde transformation importante permettant de générer une simulation SystemC du modèle de SoC.

5.3 Génération de code SystemC

La chaîne de transformations menant à la génération de code SystemC passe d'abord par la transformation entre les méta-modèles *Polyhedron* et *Loop*. Nous ne présentons pas en détail cette transformation, d'une part parce qu'elle a été développée dans le cadre des travaux de thèse de Julien Taillard et d'autre part parce qu'elle est d'un point de vue théorique particulièrement simple : tous les éléments sont recopiés à l'exception des polyèdres qui sont transformés en nids de boucle par l'appel à un outil externe (CLooG).

Considérant que le code SystemC est suffisamment proche de la représentation du modèle *Loop*, une transformation directe est utilisée pour passer de l'un à l'autre. Cette transformation est de type modèle-vers-texte. La partie de cette transformation générant le code des composants matériels a été réalisée par Rabie Ben Atitallah. Chaque composant matériel mène à la création d'une classe C++. Dans le cas d'un composant composé, la classe contient l'instanciation de tous les sous composants (éventuellement répétés) et des connexions entre

eux. Dans le cas d'un composant élémentaire, la classe contient un appel à l'IP du composant.

Après avoir détaillé le fonctionnement d'une transformation modèle-vers-texte nous présenterons les principes sur lesquels repose la génération de code de la simulation de l'application au niveau PA.

5.3.1 Moteur de transformation modèle-vers-texte

Selon les recommandations de l'IDM, pour effectuer la transformation d'un modèle en texte il faut que le modèle soit aussi proche que possible des concepts présents dans le texte généré. La relation entre un élément du modèle et une partie du texte doit être autant que possible de type un-vers-un. L'idée portée par cette contrainte est que la transformation ne doit pas modifier les concepts mais uniquement les traduire *littéralement* de la représentation abstraite à une représentation concrète, manipulable directement par les outils usuels.

La plupart des moteurs de transformations existants supportent cette sorte de transformation, probablement parce que le fait qu'elle soit de type un-vers-un la rend facile à implémenter. L'équipe a choisi JET [38] (Java Emitter Templates) : le pendant de EMFT Query (utilisé comme couche de base par MoMoTE) pour la transformation modèle-vers-texte. Outre sa facile intégration avec l'environnement Eclipse, il permet de mixer la notion de patron, commune à la plupart des moteurs de transformations modèle-vers-texte, au langage Java, ce qui lui confère une grande possibilité d'expression. Nous n'utilisons pas directement JET, mais via l'utilisation d'une couche supplémentaire qui effectue le lien avec Ecore, la technologie de modèle d'Eclipse. Développée par l'équipe, cette couche est nommée MoCodE (Model to Code Engine), elle est spécialement dédiée à la génération de code source depuis des modèles.

Le principe de base de MoCodE est d'associer à chaque type d'élément du méta-modèle d'entrée un patron de code source. Au départ de l'exécution, seule la racine du modèle est transformée. C'est au code présent dans le patron de parcourir le modèle et de solliciter la transformation des éléments souhaitée (à l'aide de la fonction `generate()`). Ainsi, au fur et à mesure, les patrons parcourent chacun un petit bout du modèle et appellent d'autres patrons. Le moteur de transformation détermine le patron correspondant à un élément en recherchant le patron ayant le même nom que le type de l'élément transformé. S'il n'existe pas de patron spécifique au type, alors le moteur remonte l'héritage de classe jusqu'à trouver un patron existant. Par exemple si la génération d'un élément *SoftwareImplementation* est demandée, MoCodE recherchera un patron nommé `SoftwareImplementation`, puis cherchera un patron correspondant à son père, nommé `Implementation` et enfin à son grand-père, `NamedElement`. De plus, l'utilisateur identifie les types d'élément qui génèrent leur propre fichier. Pour les autres types, la chaîne de caractères obtenue par l'exécution du patron sur un élément donné est insérée dans le fichier depuis lequel la génération a été appelée.

L'écriture du patron est inspirée des technologies web PHP[2] et JSP[3] : par défaut le texte présent dans le patron est directement écrit dans le fichier de sortie. Il existe des délimiteurs spéciaux (`<%` et `%>`) qui indiquent l'appel à du code Java. Lors de l'exécution, le texte généré par le code est inséré à l'emplacement du code. De plus, il est également possible d'ajouter à la transformation des bibliothèques de fonctions purement en Java qui pourront être appelées depuis n'importe quel patron.

Lors de la génération de code, il est souvent nécessaire de produire différents types de fichiers : les fichiers de code, les fichiers d'en-tête, les fichiers de compilation (tels que le

[2] http://php.net/manual/
[3] http://java.sun.com/products/jsp/

Makefile), etc. MoCodE permet de traiter chaque type de fichier comme une génération de code séparée, il suffit pour cela de placer les patrons dans des paquetages différents. Pour chaque paquetage, c'est-à-dire chaque type de fichiers sortie, la génération est entièrement relancée en partant de la racine du modèle.

Par exemple, voici le patron permettant de générer le code correspondant à l'élément *If* du méta-modèle *Loop* (présenté dans la section sur les méta-modèles intermédiaires) :

```
1  <%@ jet package="generated.appli_src"
       class="If"
       imports="java.util.* org.eclipse.emf.ecore.EObject fr.lifl.west.mocode.engine.*
                fr.lifl.west.gaspard2.metamodel.deployedloop.* tools.*"
5  %>
   <%
   fr.lifl.west.mocode.engine.Engine engine =
       ((fr.lifl.west.mocode.engine.Argument) argument).getEngine();
   fr.lifl.west.gaspard2.metamodel.deployedloop.If element =
10     (fr.lifl.west.gaspard2.metamodel.deployedloop.If)
           ((fr.lifl.west.mocode.engine.Argument) argument).getElement();
   ApplicationTrace AT = (ApplicationTrace)
       ((fr.lifl.west.mocode.engine.Argument) argument).getUserArgument();
   %>
15     // Block If
       if ( <%=LoopsVariables.convertVarNames(AT.getVariables(), element.getCondition())%> )
       {
   <%
   List<fr.lifl.west.gaspard2.metamodel.deployedloop.Block> sub_block_list =
20     (List<fr.lifl.west.gaspard2.metamodel.deployedloop.Block>) element.getContent();
   // Generate the nested loops inside this block
   for (fr.lifl.west.gaspard2.metamodel.deployedloop.Block objBlock : sub_block_list) {
   %>
           <%=engine.generate(objBlock, AT)%>
25 <%
   }
   :
   :
   %>
30     } // End block If
```

Dans cet exemple, les 5 premières lignes indiquent que le nom du patron est If, il permet donc de traiter les composants de type *If*, et fait partie du paquetage generated.appli_src qui correspond à la génération de fichiers .cc. Les trois lignes suivantes permettent de récupérer les arguments passés au patron. Le premier argument est toujours l'élément sur lequel le patron travaille.

Viennent ensuite les quelques lignes de codes générées par le patron correspondant directement à la notion de *If* en SystemC. À l'intérieur des parenthèses du if (ligne 16) vient se greffer la condition, lue depuis l'élément traité. La fonction convertVarNames() est une méthode appartement à une bibliothèque Java auxiliaire à la transformation. Elle convertit le pseudo-code contenu dans le modèle en code SystemC.

Enfin, dans la partie entre accolades du if (lignes 18 à 29) est présent le code permettant de parcourir chaque sous-bloc de l'élément traité. Chacun des sous-blocs sera inséré à cet emplacement. Notons que même si la fonction generate() est appelée identiquement quelque soit le type de bloc, le moteur de transformations exécutera un patron différent selon qu'il est un *Loop*, un *If*, ou un *Assignement*.

Nous allons maintenant présenter les différentes parties de la transformation de modèles *Polyhedron* en codes de simulation SystemC.

5.3.2 Génération du cœur du processeur

Tout le code s'exécutant sur un processeur est contenu dans une seule classe, c'est-à-dire dans un fichier source et un fichier d'en-tête. Cette classe contient l'interface SystemC du

processeur, chaque activité exécutée par le processeur, et l'ordonnanceur dynamique qui correspond également à la méthode que le simulateur SystemC appelle durant la simulation. Ce dernier est le cœur du processeur, il organise l'exécution globale de l'application.

Ainsi, dans la transformation le patron correspondant à la racine du modèle énumère chaque élément *ProgrammableProcessor* du modèle *Loop* et appelle la génération de code. Le patron de processeur, qui est spécifié comme étant à l'origine d'un nouveau fichier, définit les bases de la classe processeur. Une partie du code généré est invariable : le constructeur de la classe, les méthodes d'aide aux activités pour les accès en mémoire, les méthodes d'aide aux activités pour connaître l'indice de répétition du processeur. Le code de l'ordonnanceur est également invariable à l'exception de la partie se chargeant de créer et de lancer un thread pour chaque activité contenue dans le processeur.

Le patron de processeur se charge également d'appeler la transformation de chaque IP utilisé par les tâches élémentaires. Cette transformation ne se charge pas d'introduire le code source de l'IP mais uniquement d'insérer la déclaration d'appel C conforme au protocole d'appel à cet IP : le nom de la fonction à appeler et le type de chaque paramètre. Elle est calculée à partir des informations de déploiement contenues dans une *SoftwareImplementation*.

En plus de cela, le patron génère également la création de chacune des synchronisations utilisées par les tâches du niveau supérieur d'une activité. Ces synchronisations, des variables de type `gasp_sync`, n'appartiennent pas au processeur : elles sont définies globales au simulateur de manière à ce que tous les processeurs puissent y avoir accès.

Enfin, le patron appelle la création de chaque activité. Cependant, le modèle d'entrée ne contient pas explicitement les informations sur les activités : il ne contient qu'une seule arborescence de tâches par processeur. Le patron fait donc appel à une méthode spéciale capable de retrouver chaque tâche d'origine d'une activité. Pour chaque tâche d'origine, la génération de la tâche racine est appelée avec un argument contenant le chemin de la racine à la tâche d'origine. Comme nous allons le voir dans la section suivante, chacun de ces appels va insérer le code complet d'une activité.

5.3.3 Génération des activités

La génération d'une activité se fait par l'appel récursif au patron traitant les *ApplicationComponents*. Générant d'abord la tâche racine, qui représente les répétitions les plus globales, l'appel au patron permet de générer les nids de boucles jusqu'à l'utilisation des IP. Comme nous l'avons vu dans le chapitre exposant le modèle d'exécution, une activité est décomposée entre une partie supérieure ordonnancée dynamiquement et synchronisée entre processeurs et une partie inférieure contenant un ensemble de tâches ordonnancées statiquement. Au plus bas niveau d'une activité se trouvent les appels aux IP. Nous allons voir successivement la création de ces différents niveaux qui sont illustrés par la figure 5.13. Notons que nous ne présentons ici que la génération du fichier `.cc`. Le fichier d'en-tête `.h`, créé via un autre paquetage de patrons, est pratiquement entièrement invariable.

5.3.3.1 Code du niveau supérieur

Lors de la génération du niveau supérieur de l'activité, seules les tâches parents de la tâche d'origine sont parcourues. Pour connaître le chemin à suivre, le patron de composant applicatif prend en argument la liste des composants depuis le composant en cours de

génération jusqu'à la tâche d'origine. Lors de l'appel pour générer le code du composant contenu, la liste est tronquée puis passée en argument. Lorsque la liste ne contient plus que la tâche d'origine, ou bien est complètement vide alors le patron génère le code pour le niveau inférieur, ce que nous détaillerons juste après.

Lors de la génération d'un composant compris dans le niveau supérieur, trois points sont effectués. Tout d'abord, pour chaque connecteur simple, qui correspond à la transmission d'un tableau, une synchronisation est placée. Selon que le port correspondant est *In* ou *Out*, la synchronisation est en lecture ou en écriture. Puis, pour chaque tiler que la tâche utilise, un court-circuit avec le tiler supérieur est calculé. C'est-à-dire que les deux tilers sont fusionnés de manière à obtenir un nouveau tiler pouvant directement désigner les coordonnées des points dans le tableau de données correspondant à un motif d'une répétition de la tâche. Aucun code n'est généré pour ce nouveau tiler, il est simplement transmis à la tâche inférieure, qui pourra de nouveau appliquer un court-circuit dessus.

En dernier lieu vient la génération des boucles de l'espace de répétition de la tâche. Le nid de boucle est obtenu en exécutant récursivement le patron correspondant au type exact de *Block*. Au dernier niveau de boucle, dans le bloc qui ne contient pas de sous-blocs, la génération de la tâche inférieure est appelée. L'étape de génération de code pour un composant applicatif est alors de nouveau exécutée, jusqu'à arriver au niveau inférieur de l'activité.

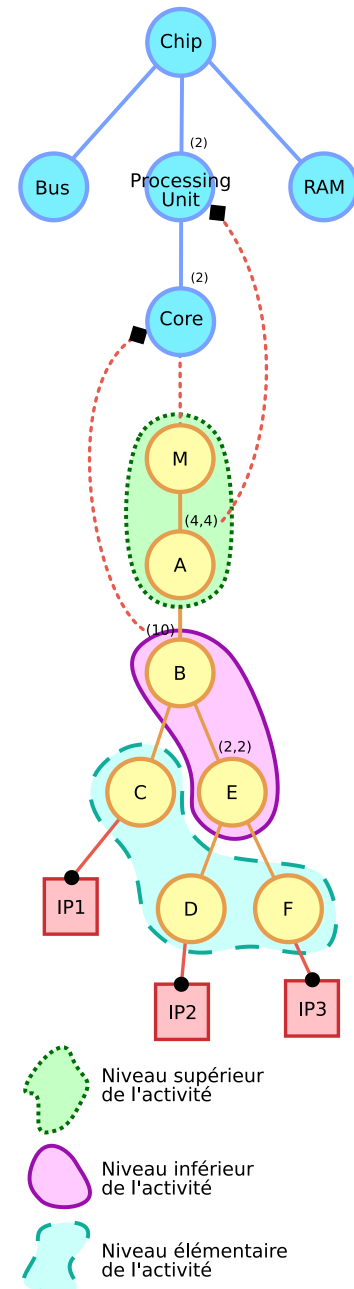


FIG. 5.13: Les différents niveaux d'une activité. Au plus haut se trouve l'architecture. Le niveau supérieur de l'activité est compris entre la racine de l'arborescence et la tâche précédant la tâche d'origine. Le niveau inférieur est compris entre la tâche d'origine et les feuilles non-inclus. Le niveau des tâches élémentaires contient les feuilles.

5.3.3.2 Code du niveau inférieur

À partir du moment où la génération a atteint la tâche d'origine de l'activité, ce qui est détecté grâce à l'argument contenant le chemin jusqu'à cette tâche passée au patron, le code produit est légèrement différent. En particulier, à partir de ce niveau toutes les sous-tâches d'une tâche sont générées.

Ainsi, au niveau inférieur de la hiérarchie, le patron commence par calculer les courts-circuits des tilers, puis génère les nids de boucles correspondant à l'espace de répétition. Cette partie de la génération est similaire à celle du niveau supérieur excepté l'absence de création des synchronisations.

Au dernier niveau de boucle, la génération est plus complexe car il faut appeler chacune des sous-

tâches dans un ordre qui respecte les dépendances de données. Le graphe des sous-tâche est transmit à un ordonnanceur statique. Cet ordonnanceur créé la liste des tâches ordonnées en commençant par sélectionner toutes les tâches qui n'ont aucune dépendance en entrée puis il sélectionne au fur et à mesure les tâches dont les dépendances en entrée sont disponibles. Une fois cette liste de tâche calculée, le patron du bloc appelle la génération de code pour chacun des composants de la liste dans l'ordre. Cela assure une exécution séquentielle des sous-tâches respectant les dépendances de données et conforme à l'exécution simulée.

5.3.3.3 Code du niveau élémentaire

La génération récursive de l'activité s'arrête lorsque la tâche générée est élémentaire. Ce niveau correspond à un patron spécial dédié aux *ElementaryTasks*. Dans une telle tâche il n'y a jamais de répétitions, il n'y a donc pas de boucles à générer. Le patron consiste donc à créer les tilers d'entrée, créer l'appel à l'IP sur lequel la tâche est déployée, puis à créer les tilers de sortie.

La création des tilers est déléguée au patron spécifique à ce type d'élément. La distinction entre un tiler d'entrée (qui lit les motifs de données depuis la mémoire) et un tiler de sortie (qui écrit les motifs dans la mémoire) est faite en regardant si le type des ports auxquels le tiler est connecté est *In* ou *Out*. Le code du tiler généré correspond au tiler calculé par les différents courts-circuits effectués le long de la descente des tâches de l'activité : il accède donc directement au tableau en mémoire, sans transfert intermédiaire, quel que soit le nombre de tâches intermédiaires.

Le code généré du tiler est constitué d'abord de la déclaration sous forme de constantes des caractéristiques du tiler (origine, pavage, ajustage). Puis le calcul des coordonnées de chaque point du motif dans le tableau est inséré. Associé à ce calcul le compteur de temps du processeur est mis à jour en fonction du nombre de dimensions du tiler est des caractéristiques indiquées sur l'IP du processeur simulé. Ce calcul est suivi de la linéarisation des coordonnées en adresses mémoire. La linéarisation utilise les informations stockées dans l'élément *Array* référencé par le port auquel est connecté le tiler. En fonction de la taille des FIFO utilisées, de la taille des tableaux précédents dans la mémoire, et du nombre de duplications des tableaux, l'adresse de base du tableau est calculée. Enfin, une lecture ou une écriture, selon le type du tiler, est effectuée pour chaque point du motif en utilisant les fonctions d'accès à la mémoire mises à disposition par la classe du processeur. Ce sont ces fonctions qui se chargent de journaliser les transferts de données effectués sur le processeur.

Une fois chaque tiler d'entrée produit (dont l'ordre n'importe pas), l'appel à l'IP est généré. Cela utilise les informations de déploiement contenues dans l'implémentation. Selon le déploiement des ports, les motifs d'entrée et de sortie sont passés dans un ordre précis comme paramètre de la fonction. Puis le temps d'exécution de la fonction, également spécifié par les concepts de déploiement, est ajouté au compteur de temps du processeur. Bien sûr, lorsque l'utilisateur a choisi de ne pas simuler l'aspect fonctionnel de l'application, l'appel à l'IP n'est pas généré, seul le compteur de temps est incrémenté.

Après cela les tilers de sorties sont générés, terminant ainsi le code de l'activité.

5.3.4 Création d'un Makefile

La génération du simulateur ne s'arrête pas à la génération de tous les fichiers sources, il est nécessaire de permettre à l'utilisateur de facilement compiler et exécuter le simulateur.

La transformation que nous avons réalisée génère donc, en plus des fichiers sources, un fichier *Makefile*. Ce fichier, usuel dans le cadre de développement UNIX, permet d'automatiser entièrement la compilation et l'édition de liens des fichiers du simulateur. Après l'exécution de la commande, l'utilisateur peut lancer la simulation en exécutant le programme généré appelé `TLMRun`.

Voici un exemple de fichier généré (et légèrement simplifié pour faciliter la lecture) :

```
1  sensor.o : SPECIALIZATION =-DFILENAME=\"Gen_IM.pgm\" -DWIDTH=100\
              -DHEIGHT=100 -DDESTINATIONADDRESS=0x00001000
   sensor.o : CFLAGS_CODE =-O3 -g -march=i686
   c-vadd.o : SPECIALIZATION =-DVECTOR_SIZE=64
5  c-vadd.o : CFLAGS_CODE =
   LDFLAGS_CODE = -lnetpbm

   ## Library directories
   LIBS = -L$(SYSTEMC)/lib-$(TARGET_ARCH) -L$(TLM_PA_LIB)
10 CFLAGS = -g -Wall $(INCDIR) $(CFLAGS_CODE) $(SPECIALIZATION)
   LDFLAGS = $(LDFLAGS_CODE) -lsystemc  -lpthread

   ## All the source files in the directory need to be compiled
   SRCS = $(wildcard ./*.cc) $(wildcard ./*.c)
15 OBJS = $(patsubst %.c,%.o,$(SRCS:%.cc=%.o))

   all:    Make.dep TLMrun

   %.o: %.c
20         $(GX) $(CFLAGS) -c $< -o $@

   %.o: %.cc
           $(GXX) $(CFLAGS) -c $< -o $@

25 TLMrun: $(OBJS)
           $(GXX) -o TLMrun $(LIBS) $(OBJS) $(LDFLAGS)
```

Dans la génération, une grande partie du fichier *Makefile* est statique : les commandes permettant de compiler chaque type de fichier supporté (actuellement SystemC, C++, et C), les commandes pour effectuer l'édition de lien, et la liste des fichiers à compiler (tous les fichiers de code source présents dans le répertoire). Dans l'exemple présenté ci-dessus, cela correspond aux définitions entre la huitième ligne et la fin.

Le reste du fichier généré (compris entre les lignes 1 et 7 dans l'exemple) permet de compiler les fichiers sources des IP. Pour chaque fichier, l'attribut `compilationLine` du *CodeFile* le représentant est utilisé pour compléter les arguments passés au compilateur. Ce mécanisme est également utilisé pour transmettre au code les `specializations` contenues dans l'implémentation correspondante au fichier. Elles sont chacune représentées sous la forme d'une variable du préprocesseur et spécifiées sous la forme d'un argument pour le compilateur. Enfin, les `linkingLines` de tous les *CodeFiles* du modèle sont concaténées et passées en argument de l'éditeur de lien.

5.3.5 Synthèse

La transformation d'un modèle *Loop* vers du code SystemC de simulation au motif près repose sur une technologie de génération de code à l'aide de patrons. Un patron, qui correspond à une classe du méta-modèle source, contient le texte tel qu'il doit être produit et, dans des sections délimitées, du code Java permettant de contrôler la génération.

Chaque composant matériel est transformé en une classe, c'est-à-dire deux fichiers sources. Les composants de type processeur sont pris en charge par un patron particulier qui insère à l'intérieur du fichier le code de la partie de l'application allouée sur le processeur. Chaque activité est générée en appelant récursivement les patrons pour les composants d'application et les blocs de boucles. En outre, un fichier `Makefile` est également créé, permettant ainsi d'automatiser complètement la compilation du code du simulateur.

Similairement à la transformation d'un modèle *Gaspard* vers un modèle *Polyhedron*, la validation de cette transformation s'est faite par son application sur un certain de nombre de modèles de SoC. Les différentes étapes la progression de la validation ont été la vérification manuelle que le code généré est complet, la compilation sans erreur du simulateur, l'exécution de la simulation jusqu'à la fin du traitement de l'application et enfin la vérification des résultats fonctionnels et du journal d'exécution produit par le simulateur.

5.4 Synthèse et conclusion

Dans ce chapitre nous avons présenté l'usage des transformations de modèles et leur implémentation dans nos travaux. En partant du modèle Gaspard avec les évolutions proposées dans le chapitre 3, les transformations successives permettent d'obtenir une simulation du MPSoC au niveau d'abstraction proposé dans le chapitre 4. La compilation du modèle est faite en passant à travers trois transformations successives qui ont chacune pour but de rapprocher les concepts abstraits facilement manipulables par le concepteur à des concepts toujours plus proche du code de simulation.

Les deux premières transformations sont communes aux autres chaînes de transformations de l'environnement Gaspard. Elles sont reliées par le méta-modèle intermédiaire *Polyhedron*. Ce méta-modèle n'a ni été défini pour être facilement utilisable par le concepteur ni été spécifié de manière à être équivalent à du code, mais il a été défini pour permettre la compilation d'une partie du modèle *Gaspard* sans pour autant imposer des concepts propres à une cible de compilation. Ainsi cette première transformation de la chaîne se charge principalement de supprimer l'expression explicite de l'association en plaçant directement les tâches sur les processeurs et les tableaux de données sur les mémoires, et de simplifier le déploiement en reliant directement les composants élémentaires à l'implémentation d'IP la plus adaptée. Après la seconde transformation le MPSoC est exprimé par un modèle *Loop*. À partir de là, une dernière transformation, de modèle-vers-texte, est capable de générer l'ensemble de fichiers complets, directement compilables pour produire le simulateur exécutable.

En prenant du recul sur le travail d'implémentation effectué, il est intéressant de faire quelques remarques sur l'usage des transformations de modèles dans le cadre de l'IDM et plus spécifiquement pour la compilation de SoC. Concernant le flot de développement, l'usage de langages déclaratifs s'est avéré particulièrement efficace pour le développement de ce « compilateur ». Le fait que les deux moteurs de transformations utilisés permettent de mixer cette approche avec un langage impératif usuel (Java) a favorisé d'autant plus la flexibilité de programmation. Chaque règle de transformation étant clairement séparée et

définie par ses entrées et sorties, la conception et la maintenance sont simplifiées et la qualité de la transformation peut être plus aisément assurée. De même, à plus gros grain, étant donné que les modèles d'entrée et de sortie des transformations sont clairement identifiés par les méta-modèles, il est facile d'ajouter ou de remplacer une transformation à l'intérieur d'une chaîne de transformations. Dans le cadre spécifique des SoC, où la technologie évolue très vite, et où les améliorations du compilateur peuvent avoir un grand impact, cette flexibilité dans le développement et surtout dans l'évolution du compilateur est un atout important.

Concernant l'outillage, les transformations de modèles souffrent encore d'une technologie trop jeune. Aucun outil n'a encore complètement implémenté le seul standard pour l'écriture de transformations : QVT. S'il existe de nombreux moteurs de transformations, aucun n'est encore pleinement mature. Le fait que l'équipe se soit résolue à développer ses propres moteurs (MoMoTE et MoCodE) en est un écho. Par ailleurs, lors du développement un problème particulièrement notable a été que même si deux méta-modèles sont très proches, ils doivent être spécifiés entièrement séparément, ce qui contraint à devoir écrire de nombreuses règles de un-vers-un. Ce fut le cas en particulier dans la transformation de *Polyhedron* à *Loop*. Une amélioration de la méthodologie serait de pouvoir spécifier un méta-modèle intermédiaire uniquement par la redéfinition d'un sous-ensemble des paquetages d'un méta-modèle de référence. De manière similaire, la transformation permettant de passer du méta-modèle de référence au méta-modèle intermédiaire ne définirait que les règles transformant le sous-ensemble des paquetages. Cela éviterait d'avoir à coder explicitement chaque règle identité et pourrait permettre de fusionner plusieurs transformations facilement (tant qu'elles ne travaillent pas sur les mêmes concepts).

En soi, cette chaîne de transformations permet déjà de démontrer un premier point qui était loin d'être évident au début de ces travaux : l'usage de l'IDM est applicable à la co-conception de SoC et il est possible d'obtenir automatiquement le code complet du SoC à partir d'une description exclusivement sous forme de modèles. Dans le chapitre suivant nous allons, à l'aide d'une étude de cas, valider le fonctionnement de la transformation et l'intérêt d'une simulation au niveau d'abstraction PA.

Chapitre 6

Étude de cas et validation expérimentale

6.1 Encodeur H.263 sur MPSoC **150**

6.1.1 Application 150

6.1.2 Architecture matérielle 154

6.1.3 Association 155

6.2 Génération de code à l'aide de l'environnement Gaspard **159**

6.3 Exploration du domaine de conception **160**

6.3.1 Variation du nombre de processeurs 161

6.3.2 Variation du nombre de bancs mémoire 164

6.4 Synthèse **165**

Dans les chapitres précédents nous avons présenté différentes contributions de cette thèse. Au fur et à mesure, des exemples ciblés sur quelques notions particulières ont illustré ou validé en partie ces contributions. Nous allons ici exposer une étude de cas à l'aide d'un exemple plus complet, et ainsi valider de manière expérimentale les différentes contributions et leur intégration dans le processus complet, de la modélisation du SoC jusqu'à la simulation de celui-ci. C'est également l'occasion pour le lecteur de se faire une idée plus précise sur l'organisation globale de la co-conception de SoC à l'aide de l'environnement Gaspard tel qu'il est à l'heure actuelle.

Nous allons dans un premier temps présenter le modèle de SoC développé, un encodeur vidéo H.263 placé sur une architecture multiprocesseur MIPS. Puis nous présenterons rapidement sa transformation en un code SystemC grâce aux transformations de modèles développées lors de cette thèse. Enfin, nous comparerons les résultats de la simulation avec ceux de simulations plus bas niveau dans le but de valider la simulation. Nous ferons également varier certains paramètres dans le modèle de SoC, afin de montrer une utilisation possible de Gaspard dans le cadre d'exploration de l'espace de conception.

6.1 Encodeur H.263 sur MPSoC

6.1.1 Application

Dans cette étude de cas nous avons choisi comme application un encodeur H.263 parce qu'elle est typique des applications visées par Gaspard : c'est une application gourmande en puissance de calcul effectuant un traitement d'image. Le standard H.263 [28] permet de compresser un flux vidéo. Il a été développé pour la transmission de la vidéo sur des lignes à très bas débits pour les applications de visiophonie, les systèmes de surveillance sans fil, etc. Chaque image de la vidéo est traitée séparément. L'implémentation que nous visons convertit une vidéo au format QCIF en dans le format compressé H.263. L'application est composée de trois tâches exécutées séquentiellement :

- **La transformée en cosinus discrète** (DCT, pour *Discrete Cosine Transform*) permet d'éliminer la redondance de données et de transformer les données du domaine spatial en une représentation fréquentielle.
- **La quantification** consiste à diviser chaque coefficient de la DCT par un pas de quantification et mettre les coefficients non significatifs à zéro.
- **Le codage** permet d'encoder les macro-blocs traités en attribuant à chaque coefficient DCT quantifié un mot binaire dont la longueur est d'autant plus courte que le coefficient est fréquent. La méthode de codage utilisée est celle de Huffman [56].

La taille d'une image d'une séquence QCIF est de 176×144 pixels. Dans l'algorithme d'encodage, les données manipulées sont structurées sous forme de macro-blocs qui représentent un espace de 16×16 pixels d'une image vidéo. Le format de données du macro-bloc est le YCbCr qui contient trois composantes : luminance (Y), chrominance bleu (Cb), et chrominance rouge (Cr). Les blocs de luminance décrivent l'intensité des pixels tandis que les blocs de chrominance contiennent des informations sur les couleurs des pixels. La couleur verte n'est pas explicitement codée, elle peut être dérivée à partir des valeurs des trois composantes. Un macro-bloc contient 6 blocs de 8×8 valeurs : 4 blocs contiennent les valeurs de la luminance, un bloc contient les valeurs de la chrominance bleu et un bloc contient les valeurs de la chrominance rouge.

La figure 6.1 présente l'algorithme tel que nous l'avons modélisé à l'aide du profil Gaspard. On peut voir chaque niveau de hiérarchie en parcourant le modèle de haut en bas. *VideoSequence* est le composant de base, il contient juste une répétition de 400 *QCIF2H263*. Cela permet de traiter 400 images d'une vidéo. Ce nombre correspond à la séquence vidéo de référence que nous utilisons pour les simulations. Pour la génération finale du SoC le concepteur placerait là une répétition infinie (représentée par le caractère ~). Comme à chaque fois pour les tâches de base, elle ne contient aucun port, étant donné qu'elle ne nécessite pas d'être connectée à d'autres tâches : les entrées et sortie se font via des sous-tâches dédiées.

QCIF2H263 est le composant qui traite une image, il est composé de trois sous-tâches. La tâche *QCIFReader* se charge de lire une séquence vidéo au format QCIF et produit trois tableaux, chacun correspondant à une composante YCbCr. Dans le cadre de la simulation, nous déploierons cette tâche sur une fonction lisant un fichier, mais dans la réalité cette tâche sera chargée de l'acquisition des données depuis une caméra. Quant à la tâche *CompressFileSave*, elle se charge de sauvegarder le flux compressé. Le format utilisé est choisi pour accommoder la restriction posée par le modèle de calcul : il oblige à connaître à l'avance toutes les tailles des tableaux de données. Les données sont transmises via un tableau dont la première dimension est de 384 éléments, tandis qu'un second tableau spécifie le nombre réel d'éléments utilisés (si

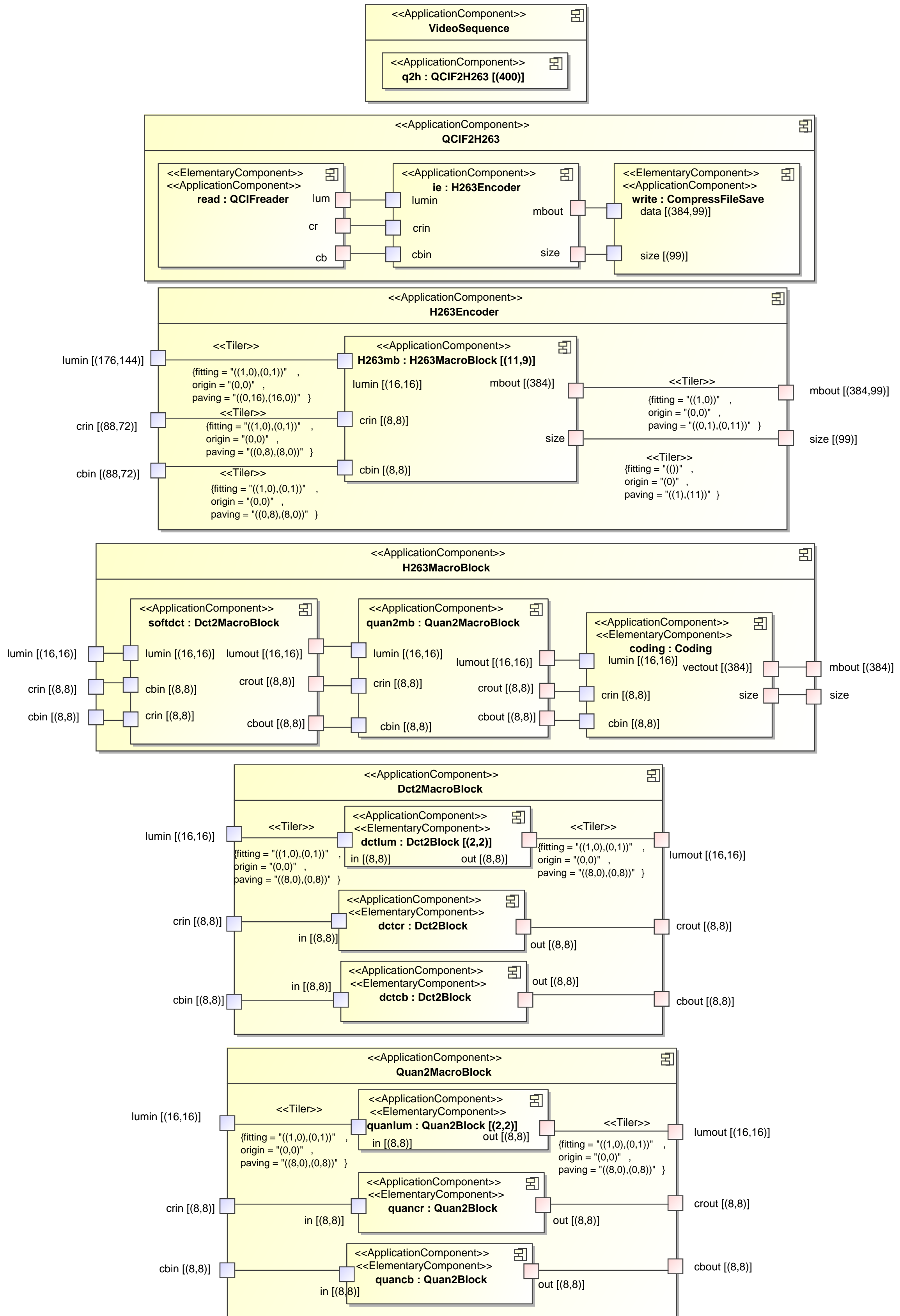


FIG. 6.1: Vue principale de l'implémentation de l'encodeur H.263 en modèle Gaspard.

la compression a été efficace, largement inférieur à 384). 384 correspond à la taille maximale d'éléments que peut générer l'encodeur à partir d'un macro-bloc ($8 \times 8 \times 6$).

Entre ces deux tâches la tâche d'encodage *H263Encoder* est appelée. Elle contient réellement tout l'algorithme pour encoder une image. C'est une tâche répétitive qui a pour sous-tâche *H263MacroBlock*. La répétition consiste à travailler sur chacun des 11×9 macro-blocs composant l'image. Comme on peut le voir par la taille des ports d'entrée de *H263MacroBlock*, un macro-bloc correspond à 16×16 pixels de la luminance et 8×8 pixels des chrominances. La découpe de l'image d'entrée se fait à l'aide des *Tilers*. Par exemple, le *Tiler* pour la chrominance bleu a un ajustage de `((1,0),(0,1))` indiquant une tuile compacte : lorsque l'on avance d'un élément dans la première dimension du motif, on avance également d'un élément dans la première dimension du tableau et lorsque l'on avance d'un élément dans la seconde dimension du motif, on avance d'un élément dans la seconde dimension du tableau. Le pavage est de `((8,0),(0,8))` : pour lire le motif suivant dans la première dimension de la répétition il faudra se décaler de 8 pixels dans la première dimension du tableau, de même pour la seconde dimension. Cela est illustré figure 6.2. Les motifs ayant une taille de 8×8, les tuiles dans le tableau sont bord à bord. Enfin, l'espace de répétition est de 11×9, on va donc lire l'ensemble des 88×72 éléments du tableau.

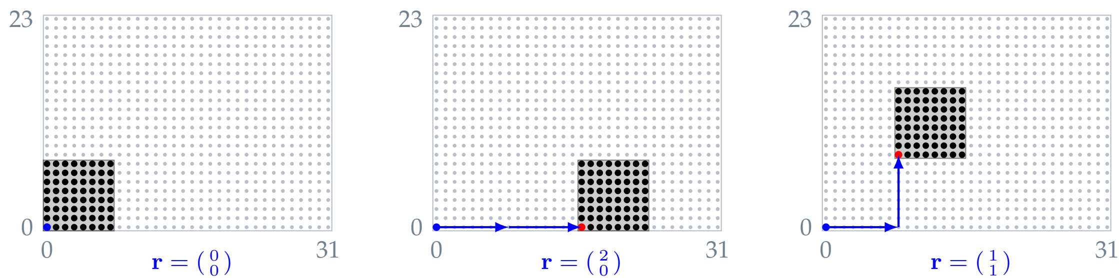


FIG. 6.2: Vue partielle du tableau des éléments de chrominance. Chaque lecture de motif correspond à une tuile compacte de 8×8 éléments du tableau. Ici sont représentés les éléments lus pour trois répétitions de tâche différentes.

Pour l'écriture du flot compressé, les deux dimensions de l'espace de répétitions sont linéarisées. Ainsi le *Tiler* entre les ports *size* a un pavage de `((1),(11))` indiquant que pour la première dimension de la répétition il faut placer chacun des 11 éléments lus les uns après les autres et que pour la seconde dimension de la répétition il faut se décaler de 11 éléments à chaque fois, écrivant ainsi chaque bloc de 11 éléments l'un après l'autre.

Le composant *H263MacroBlock* correspond à l'algorithme tel que nous l'avons décrit plus haut : un appel à la DCT (*Dct2MacroBlock*), puis à la quantification (*Quan2MacroBlock*), et au codage (*Coding*). Cette dernière tâche est une tâche élémentaire, nous l'avons déployé sur un adaptateur appelant une fonction de code de Huffman.

Le composant *Dct2MacroBlock* se charge d'appeler pour chaque bloc de 8×8 éléments la tâche élémentaire *Dct2Block* qui effectue une DCT bi-dimensionnelle. Ainsi, pour traiter la luminance, des *Tilers* permettent de découper le bloc de 16×16 en quatre motifs puis de reconstituer le bloc complet. Similairement, le composant *Quan2MacroBlock* appelle pour chaque bloc de 8×8 éléments la tâche élémentaire *Quan2Block* qui effectue une quantification.

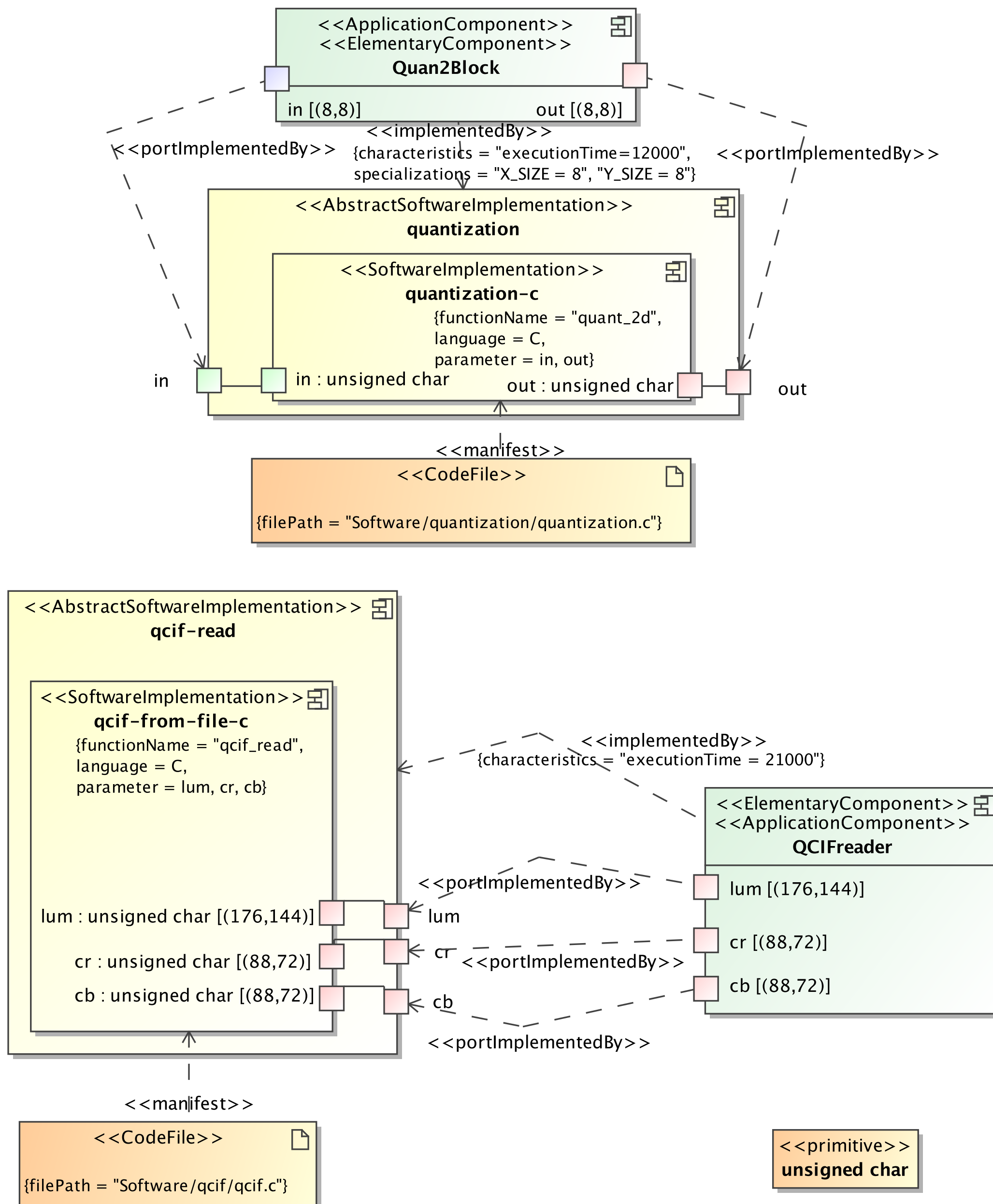


FIG. 6.3: Vue du déploiement des tâches élémentaires *Quan2Block* et *QCIFReader*. En bas à droite est présenté le type des données traitées par les fonctions : `unsigned char`.

Les fonctions auxquelles correspondent les tâches élémentaires n'étant pas disponibles dans une bibliothèque de composants (telle GaspardLib), nous avons décrit entièrement le déploiement de chaque tâche élémentaire directement dans le modèle. La figure 6.3 présente une partie du déploiement des tâches élémentaires. *Quan2Block* est déployée sur l'implémentation abstraite *quantization*, qui ne contient qu'une seule implémentation d'IP : *quantization-c*. Cette implémentation utilise un fichier (`quantization.c`) et fait appel à la fonction `quant_2d()`.

Le type des ports de l'implémentation défini le type des données traitées, il est `unsigned char` pour l'entrée comme pour la sortie. Remarquons que ce type de donnée est un élément explicite du modèle, on retrouve sa définition comme type *primitive* en bas à droite dans la figure. Enfin, il faut également noter que la dépendance *ImplementedBy* porte deux spécialisations : `X_SIZE` et `Y_SIZE`. Elles sont utilisées par la `quant_2d()` pour connaître la taille des tableaux de donnée, elle peut être adaptée à n'importe quelle taille de données. La caractéristique `executionTime` qui permettra à la simulation PA de simuler le temps de traitement a été mesurée manuellement dans une simulation du processeur au niveau CABA.

Similairement, la tâche élémentaire *QCIFReader* est déployée sur l'implémentation abstraite *qcif-read*. Comme le standard QCIF correspond à une taille d'image spécifique de 176×144 pixels, l'implémentation d'IP contenue, *qcif-from-file-c* ne permet pas au concepteur de spécifier la taille des données qu'elle génère : chaque port a une taille fixée.

6.1.2 Architecture matérielle

L'architecture matérielle modélisée est relativement simple. Présentée figure 6.4, elle a pour composant principal *HardwArchit*. Ce composant est composé de deux mémoires, un réseau d'interconnexion et quatre processeurs MIPS. Le réseau d'interconnexion est un crossbar modélisé à l'aide de deux ports multiples. L'un de ces ports est de type *In* et permet de connecter les composants ayant des ports initiateurs de transaction (tels que les processeurs). En fonction du nombre de composants à brancher le concepteur peut faire varier sa multiplicité. Similairement, l'autre port est de type *Out*, auquel on peut connecter les composants ayant des ports esclaves lors des transactions.

C'est sur ce second port que sont connectées les deux mémoires. L'une des mémoires est destinée aux données et l'autre aux instructions mais même si les noms choisis sont explicites, vis-à-vis du code généré ce n'est pas dans le modèle d'architecture que cette sémantique est spécifiée, elle sera spécifiée lors de l'association. Les deux composants sont élémentaires, ils sont déployés sur une implémentation abstraite de RAM disponible dans la bibliothèque de composant GaspardLib. Pour relier ces composants sur un port différent du réseau d'interconnexion, nous utilisons deux *Reshapes* qui indiquent comment distribuer le port d'une mémoire sur le port *slave* de multiplicité 2. Ainsi les deux *Reshapes* sont pratiquement identiques : ils sélectionnent le port de la mémoire grâce au *Tiler tin* puis le place dans le tableau du port *slave* grâce à l'autre *Tiler*. Pour la mémoire *i*, l'origine du *Tiler* est 0, tandis que pour *d* l'origine est 1. Les mémoires sont ainsi connectées respectivement au port *slave* d'indice 0 et 1.

Le composant *MultiMips* a une interface avec un port de multiplicité 4, il est donc possible de le connecter directement au réseau d'interconnexion. Ce composant est un composant répétitif : il contient un sous-composant *ProcessingUnit* répété quatre fois. C'est ainsi que l'on spécifie que l'architecture est quadriprocesseur. Le composant *ProcessingUnit* est lui même composé de deux sous-composants. Il contient un processeur nommé *mips* et un cache nommé *c* qui fait l'interface entre le processeur et l'extérieur de l'unité de calcul.

Même si les types des composants élémentaires indiquent leur fonction générique (processeur, réseau de communication, RAM) et que les noms des composants élémentaires ont été choisis de manière à refléter au mieux la fonctionnalité des composants, pour la transformation ils ne signifient rien tant qu'ils n'ont pas été déployés sur un IP. La figure 6.5 présente le déploiement des composants *Cache* et *MIPS*. Nous avons pu utiliser des IP définis dans la bibliothèque GaspardLib (présentée dans le chapitre 3). Ainsi, après importation de cette bibliothèque dans le modèle, *Cache* a été déployé sur une implémentation abstraite

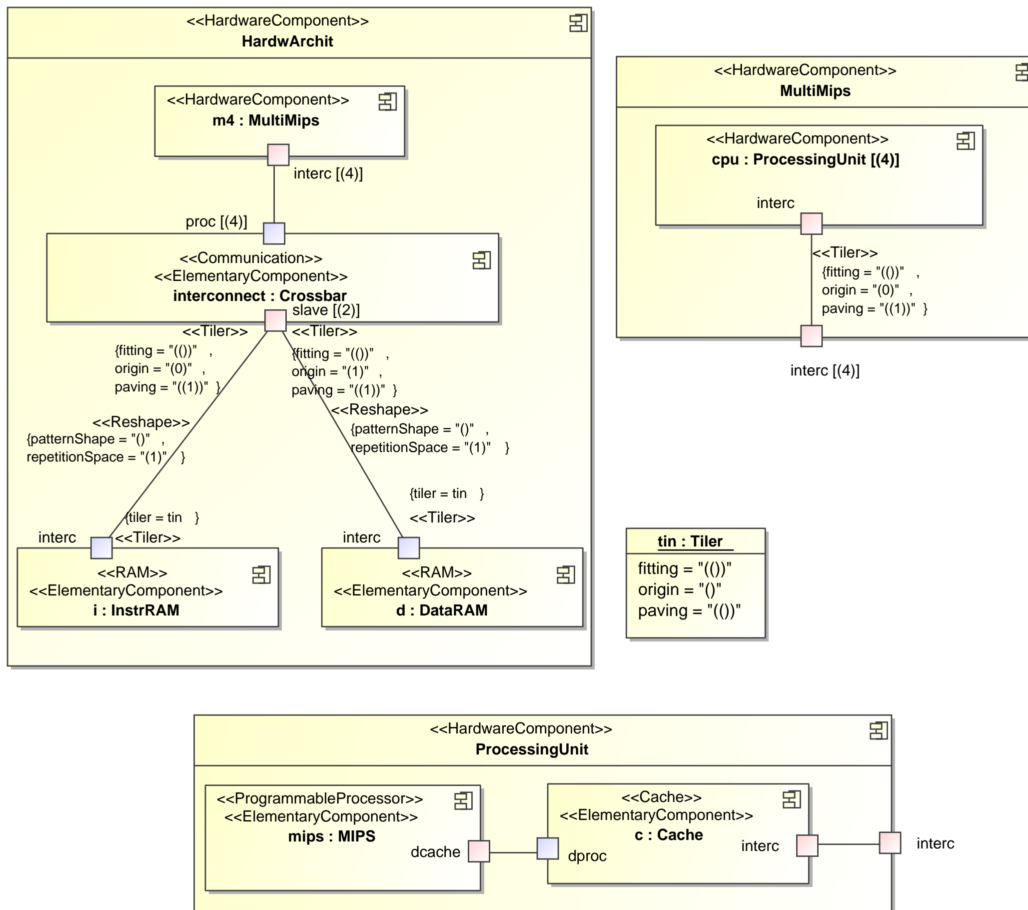


FIG. 6.4: Vue principale de l'architecture matérielle du MPSoC en modèle Gaspard.

d'un cache de 32 Ko. *MIPS* a été déployé sur une implémentation abstraite de processeur MIPS. La caractéristique `selectedFrequency=100` a été ajoutée afin de choisir la vitesse d'exécution du processeur. Notons que dans le cadre de la génération vers une simulation PA, l'implémentation du processeur n'est utile que pour connaître l'interface des ports, les IP ne sont pas utilisés à ce niveau d'abstraction.

6.1.3 Association

Une fois les modèles d'architecture matérielle et d'application définis, pour obtenir le modèle de SoC au complet il faut définir l'association. La figure 6.6 présente l'association que nous avons retenue. Notons tout d'abord les deux instances de type *MainInstance* qui représentent le composant principal d'architecture matérielle et la tâche principale de l'application.

videoComp, la racine de l'application, est placée via une *DataAllocation* sur l'instance *d* de l'architecture, ce qui indique que tous les tableaux de l'application seront contenus dans cette mémoire. La racine de l'application est également placée via une *TaskAllocation* sur le composant *MultiMips*, indiquant que toute l'application sera exécutée sur cette partie de

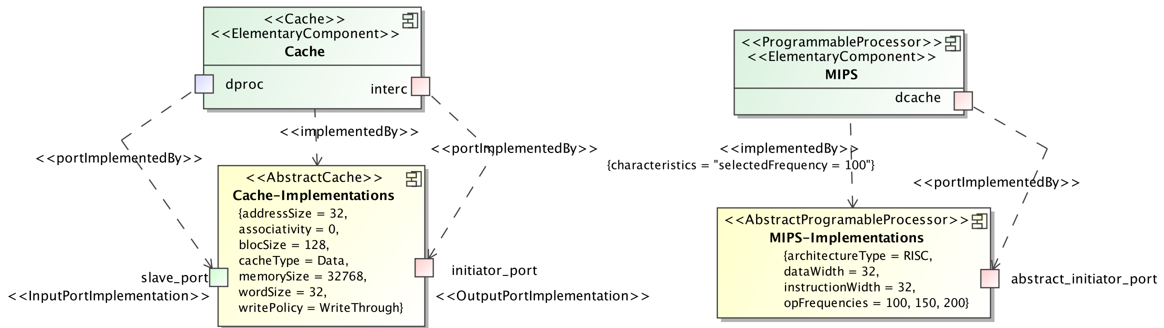


FIG. 6.5: Vue du déploiement des composants *Cache* et *MIPS*.

l'architecture.

Pour spécifier le placement précis sur les quatre processeurs des différentes tâches qui composent l'application, nous avons représenté la structure interne des composants applicatifs *QCIF2H263* et *H263Encoder* ainsi que celle du composant *MultiMips*. Les deux tâches élémentaires correspondant aux composants *CompressFileSave* et *QCIFreader* sont placées respectivement sur le troisième et quatrième processeur. Nous les avons mises sur des processeurs différents dans le but de répartir la charge de calcul le plus équitablement possible, même si en réalité ces tâches ne représentent que très peu de calculs par rapport à la tâche d'encodage.

Pour ce placement nous utilisons des *Distributions*, puisqu'elles sont nécessaires pour toute l'allocation qui met en jeu au moins une instance répétée. Les deux distributions sont très similaires : elles ont un espace de répétition de `(1)`, une forme de motif de `()` (un tableau de zéro dimension, un seul point), utilisent en entrée le *Tiler* `one-point`. Elles ne diffèrent que par le *Tiler* de sortie qui est `one-third` ou `one-forth`. Ces tâches ne sont pas répétées, donc elles correspondent à des tableaux de zéro dimension (de forme `()`). Le fonctionnement des distributions consiste à prendre un élément dans le tableau contenant les tâches, il n'en existe de toute façon qu'un seul, à l'aide du *Tiler* `one-point` puis à l'aide du second *Tiler* à le placer sur un élément du tableau de processeur (parmi les quatre disponibles). Le numéro du processeur est spécifié à de l'origine du *Tiler*, qui vaut `(2)` pour `on-third` et `(3)` pour `on-forth`. L'espace de répétition des distributions valant 1, tout cela n'est fait qu'une seule fois.

La dernière, et plus importante, distribution permet de répartir l'encodage sur les différents processeurs. Nous avons choisi de le faire à grain fin en répartissant le traitement de chaque image sur les quatre processeurs. Il aurait été possible de pipeliner le traitement de la séquence vidéo en assignant à chaque processeur le traitement d'une image complète : le premier processeur encode entièrement la première image, le deuxième processeur encode la deuxième image, etc. puis le premier processeur encode la cinquième image, le deuxième processeur encode la sixième image… Cela aurait le désavantage de quadrupler la latence entre l'entrée d'une image et la sortie. Et puis surtout, avouons-le, cela aurait été moins intéressant pour cette étude de cas.

Pour répartir le traitement de l'image sur les quatre processeurs nous utilisons le parallélisme de données spécifié dans *H263Encoder* : chaque image de 176×144 pixels est en

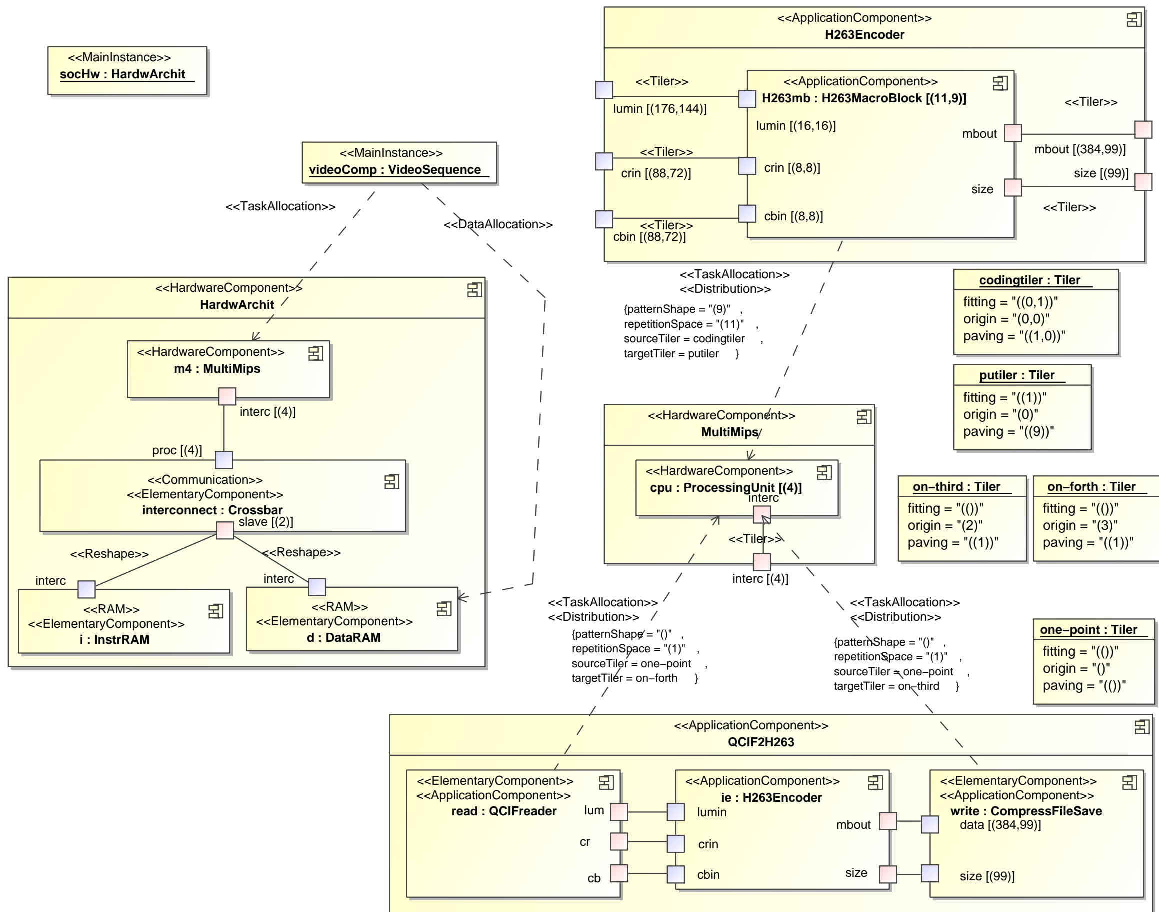


FIG. 6.6: Vue principale de l'association de l'application H.263 sur l'architecture quadriprocesseur.

fait traitée sous la forme de 11×9 macro-blocs. Nous attribuons à chaque processeur une partie de ces 99 sous-tâches, encodant chacune un macro-bloc. 99 n'étant pas divisible par 4, la répartition n'est pas parfaitement homogène : chaque processeur se voit attribuer 25 sous-tâches, à l'exception du quatrième processeur qui n'en a que 24. De manière informelle, la distribution consiste à faire comme si on linéarisait le tableau de 11×9 tâches sur un tableau de 99 éléments, mais en réalité le tableau de processeurs ne contient que 4 éléments et grâce à l'usage du modulo, tous les éléments placés après le quatrième processeur sont de nouveau placés à partir du premier processeur.

Cette distribution est schématisée en figure 6.7, où l'on voit les deux premières répétitions de la distribution. Un motif de 9 éléments est lu puis placé sur les quatre processeurs. Le premier processeur reçoit les éléments 1, 5, et 9. Le second processeur reçoit les éléments 2 et 6, etc. Lors de la répétition du deuxième motif, il est lu depuis la deuxième ligne et écrit dans les processeurs à partir de l'indice 9. 9 modulo 4 vaut 2, le deuxième processeur reçoit donc le premier élément, ainsi que le 5 et le 9, le premier processeur reçoit les éléments 4 et 8, etc.

Sur le modèle on voit cette distribution définie par une `patternShape` de 9, c'est pour

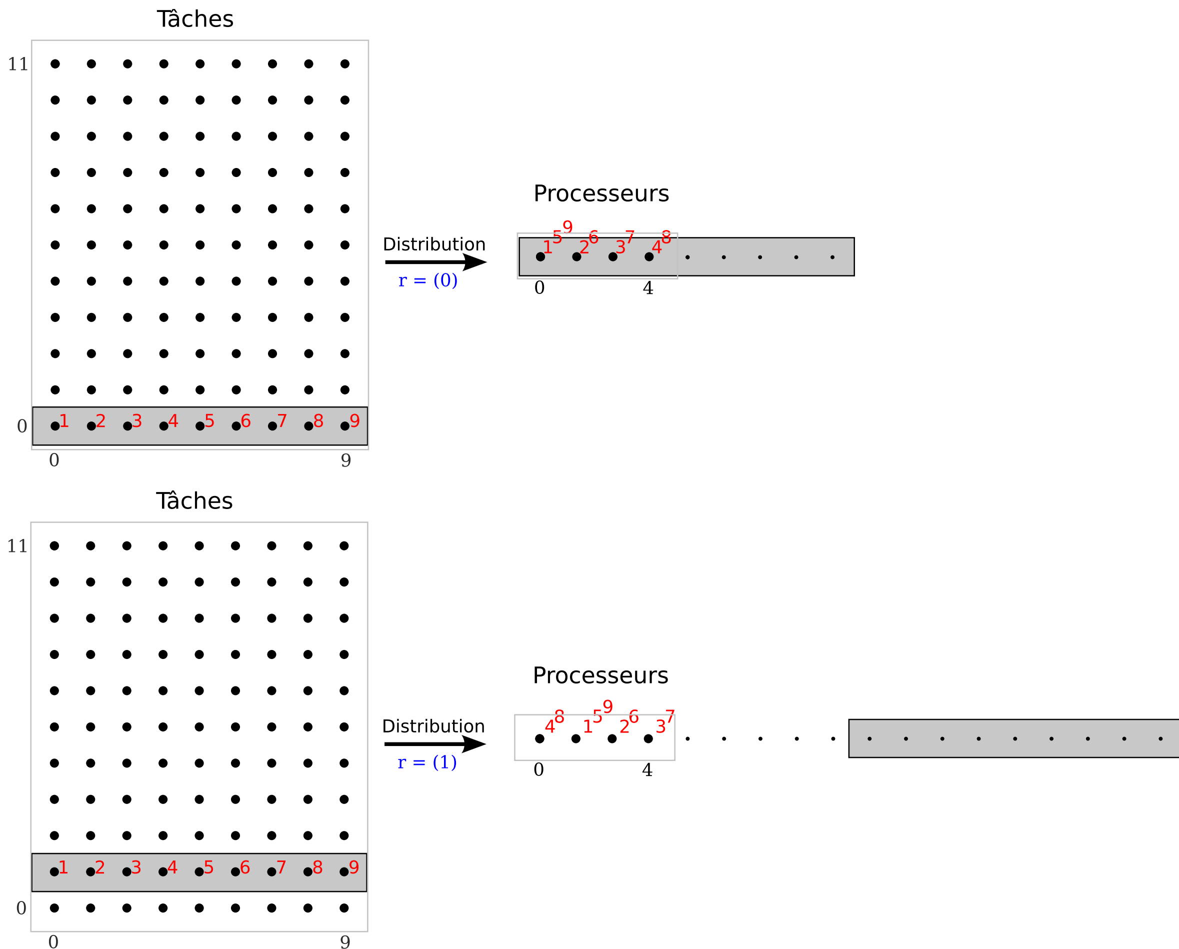


FIG. 6.7: Vue partielle de la distribution des 11×9 tâches sur les 4 processeurs. La distribution utilise un motif de 9 éléments (en gris) et le place sur les 4 processeurs en faisant jouer le modulo. Les 11 répétitions permettent de lire tout le tableau de tâche, les tâches étant placées les unes après les autres sur les processeurs.

cela que le motif lu est de 9 éléments, et un `repetitionSpace` de 11, c'est ce qui indique que le processus va être appliqué 11 fois. Le *Tiler* `codingtiler` permet d'une part de définir la lecture du motif dans le tableau de tâches, en se décalant d'un élément le long de la deuxième dimension, et le déplacement du motif à chaque répétition, en se décalant d'un élément le long de la première dimension. Le *Tiler* `putiler` spécifie le placement de ces motifs dans le tableau de processeurs. Il indique d'une part que le motif est écrit en se décalant d'un élément le long de la seule dimension, et d'autre part que chaque motif est écrit l'un après l'autre, c'est-à-dire en se décalant de 9 éléments.

Remarquons au passage, que telle quelle la distribution peut s'adapter à n'importe quel nombre de processeurs : elle va répartir le plus équitablement possible les 99 tâches sur tous les processeurs. Cela est utile car lors de l'exploration d'architecture nous allons faire varier ce nombre de 4 à 16.

6.2 Génération de code à l'aide de l'environnement Gaspard

Actuellement la modélisation du SoC se fait via la création d'un modèle UML en utilisant le profil Gaspard. Même si en théorie tout éditeur UML pourrait convenir, dans la pratique il est nécessaire d'utiliser un éditeur capable de sauvegarder le fichier contenant le modèle dans le format UML Ecore utilisé par la chaîne de transformations Gaspard. Ainsi, nous utilisons l'outil MagicDraw pour la modélisation, à partir duquel nous exportons le modèle vers le format idoine. Il est également possible d'utiliser le plugin Eclipse « Papyrus », qui manipule directement le format UML Ecore.

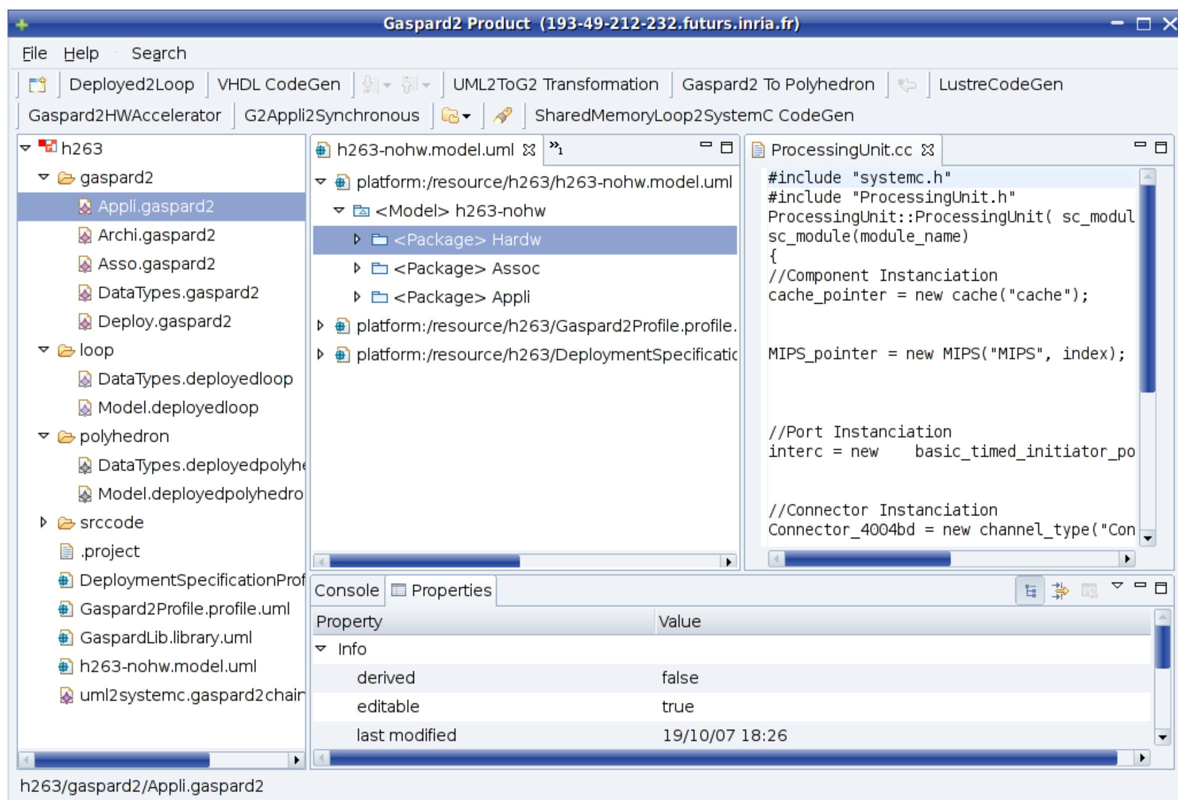


FIG. 6.8: Capture d'écran de l'environnement Gaspard. Le panneau de gauche présente les différents modèles utilisés dans la chaîne de transformation : UML, Gaspard, Polyhedron, Loop, et le code source SystemC/PA. Le panneau de droite présente le contenu du modèle UML et le contenu d'un fichier SystemC généré. Les boutons de la barre d'outils permettent d'appliquer une transformation spécifique à un modèle.

À partir des fichiers du modèle, il faut exécuter une des chaînes de transformations de Gaspard. Cela s'effectue via l'environnement Gaspard, dont une capture d'écran est présentée en figure 6.8. Dans cet environnement basé sur Eclipse, il faut créer un projet, y insérer le modèle puis choisir l'une des chaînes de compilation disponibles. Il y en existe actuellement quatre dont les cibles sont : langage Synchrone, VHDL, OpenMP/Fortran, et SystemC/PA. Dans cette étude de cas nous choisissons cette dernière. L'exécution successive des différentes transformations est alors effectuée automatiquement. Au fur et à mesure les

modèles intermédiaires sont générés. Sur la capture d'écran, ils sont visibles dans le panneau gauche (UML, Gaspard, Polyhedron, Loop, et le code source SystemC/PA). Le panneau droit permet d'afficher le contenu d'un modèle sous une forme arborescente textuelle ou les fichiers de code générés.

À la fin de l'exécution de la chaîne vers SystemC/PA, on obtient un répertoire contenant l'ensemble des fichiers nécessaires à la compilation du simulateur. Dans cet exemple nous avons obtenu une quinzaine de fichiers. Une fois ces fichiers générés, le concepteur peut compiler le simulateur en appelant `make` dans le répertoire puis exécuter la simulation à l'aide de la commande `./TLMrun`. La simulation produit un fichier journal qui contient le détail de l'exécution.

6.3 Exploration du domaine de conception

Afin de démontrer l'usage du flot de conception pour l'exploration de l'espace de conception nous avons fait successivement varier deux paramètres du modèle. Premièrement nous avons fait varier le nombre de processeurs. Nous verrons qu'il existe une limite à partir de laquelle les contentions sur le réseau d'interconnexion ralentissent l'exécution. Ensuite, pour pallier ces contentions nous avons fait varier le nombre de bancs mémoire et observé les effets sur les contentions.

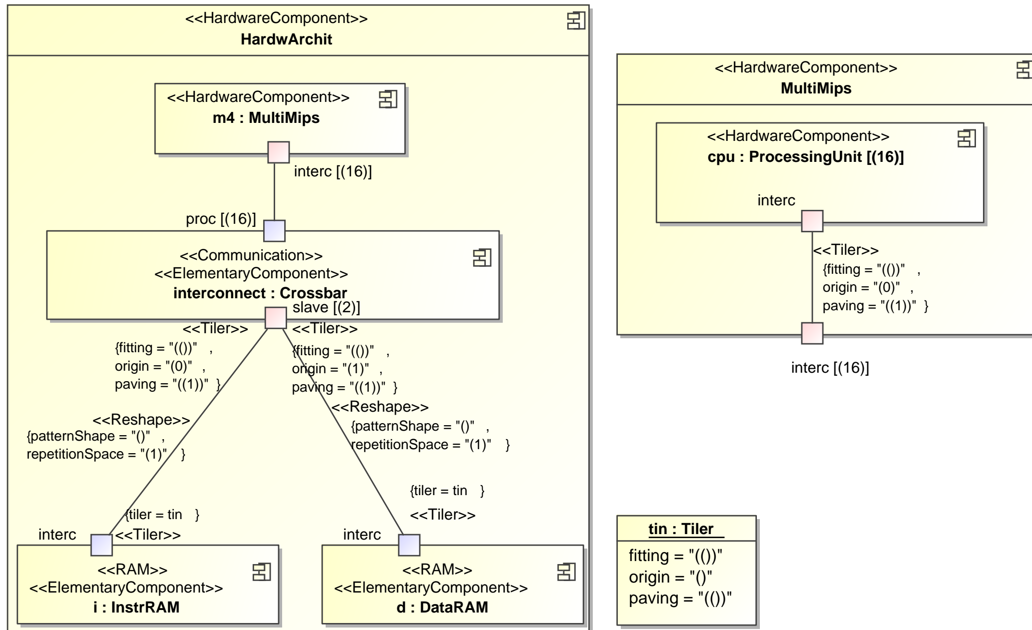


FIG. 6.9: Vue principale de l'architecture matérielle du MPSoC en modèle Gaspard avec 16 processeurs.

6.3.1 Variation du nombre de processeurs

Nous avons changé le nombre de processeurs contenus dans l'architecture de 4 (tel que nous l'avons présenté précédemment) à 8, 12, et 16. Dans le modèle cette modification est mineure, elle consiste à changer la multiplicité du composant *ProcessingUnit* et des ports des composants *MultiMips* et *Crossbar*. La figure 6.9 présente la vue de la partie architecture du nouveau modèle avec 16 processeurs. L'association n'a pas besoin d'être modifiée car nous l'avons écrite de manière à ce qu'elle puisse s'adapter à n'importe quel nombre de processeurs (tant que la répétition de *ProcessingUnit* reste d'une seule dimension).

Une fois le modèle modifié, il suffit de le ré-exporter vers Eclipse, d'exécuter de nouveau la chaîne de transformations sur le projet, de recompiler la simulation, et de l'exécuter pour obtenir les nouveaux résultats.

Afin de vérifier les résultats de simulation PA, nous avons comparé chaque configuration à la même configuration simulée au niveau d'abstraction CABA. La simulation en CABA, développée dans le cadre des travaux de thèse de Rabie Ben Atitallah, n'a pas été générée automatiquement (l'environnement Gaspard ne dispose pas encore d'une chaîne de transformations vers cette cible), elle a été écrite manuellement (la partie matérielle tout comme la partie logicielle). À ce niveau d'abstraction, la simulation est effectuée au cycle et au bit près, ainsi la précision des résultats tels que le temps d'exécution est théoriquement très proche de la réalité.

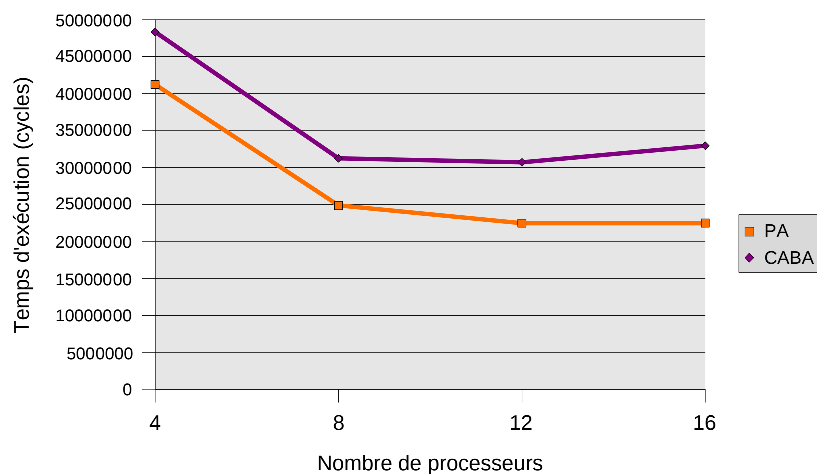


FIG. 6.10: Graphe du temps d'exécution simulé (en cycle) en fonction du nombre de processeurs pour les niveaux d'abstraction PA et CABA.

La figure 6.10 présente sous forme de graphe les différents temps d'exécution[1] obtenus par simulation selon le niveau d'abstraction et le nombre de processeurs. Notons que les résultats correspondent à la simulation de l'encodage d'une seule image de la séquence vidéo.

[1]Attention, nous employons ici le vocabulaire usuel utilisé dans le cadre de simulations : *temps d'exécution* correspond au temps d'exécution de l'application simulée sur le matériel simulé, tandis que *temps de simulation* correspond à la durée d'exécution du simulateur sur la machine hôte.

En ce qui concerne la précision des informations de temps d'exécution fournies par la simulation PA, on peut noter entre 15% et 30% de sous-estimation. Cela peut être expliqué, du moins en partie, au fait que le cache utilisé a une politique dite « write-through » qui consiste à transmettre chaque écriture à la mémoire. La mise en œuvre de ce type de politique est plus simple, mais implique une occupation du réseau d'interconnexion par toutes les écritures, y compris celles correspondantes aux données temporaires pour le parcours des boucles et le calcul interne aux fonctions des IP. Ce type d'écriture n'est pas simulé dans PA, ce qui entraîne moins de contentions sur le bus, et donc un temps d'exécution plus court. Néanmoins, ce cas est parmi les plus défavorables dans lequel peut se trouver la simulation. Le niveau d'abstraction PA est donc capable de produire des estimations du temps d'exécution convenables dans tous les cas.

Lors de l'exploration de l'espace de conception, la valeur relative est plus importante que la valeur absolue : il faut pouvoir classer les configurations entre elles pour choisir les meilleures. Dans notre exemple, par rapport à CABA, la simulation PA mène aux mêmes conclusions : augmenter le nombre de processeurs diminue le temps d'exécution jusqu'à 12 processeurs. Avec 16 processeurs, la tendance est inversée et le temps d'exécution augmente (en PA, l'exécution sur 12 processeurs prend 22,46 millions de cycles tandis qu'avec 16 processeurs elle prend 22,48 millions de cycles). Ainsi, malgré une sous-estimation en valeur absolue, la simulation PA permet de prendre les bonnes décisions vis-à-vis de l'exploration.

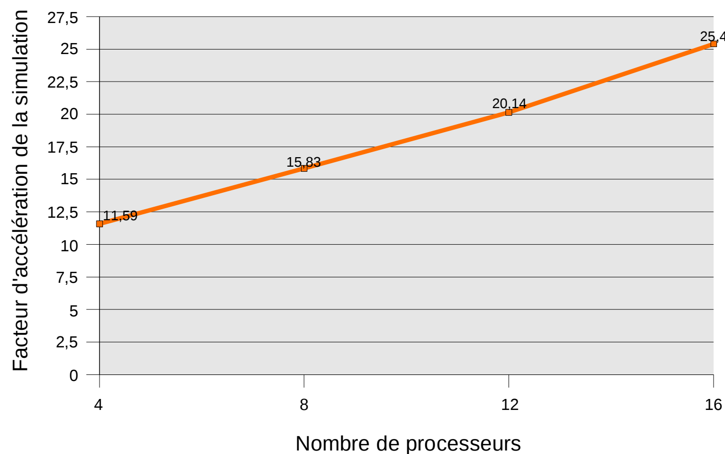


FIG. 6.11: Graphe du facteur d'accélération entre le temps de simulation PA et celui CABA en fonction du nombre de processeurs.

La figure 6.11 présente le facteur d'accélération obtenu entre une simulation PA et une simulation CABA en fonction du nombre de processeurs. Il évolue entre 11 pour 4 processeurs et 25 pour 16 processeurs. Pour donner une idée des valeurs absolues, la simulation CABA avec 16 processeurs a pris 381s tandis que la simulation PA a pris 15s. Cela démontre donc le réel avantage de simuler à ce plus haut niveau d'abstraction.

Le fait que le facteur soit plus important lorsqu'il y a plus de processeurs met en valeur la particularité de PA à accélérer la simulation des composants processeurs. Concernant le facteur d'accélération, il est intéressant de comparer le niveau PA avec le niveau d'abstraction

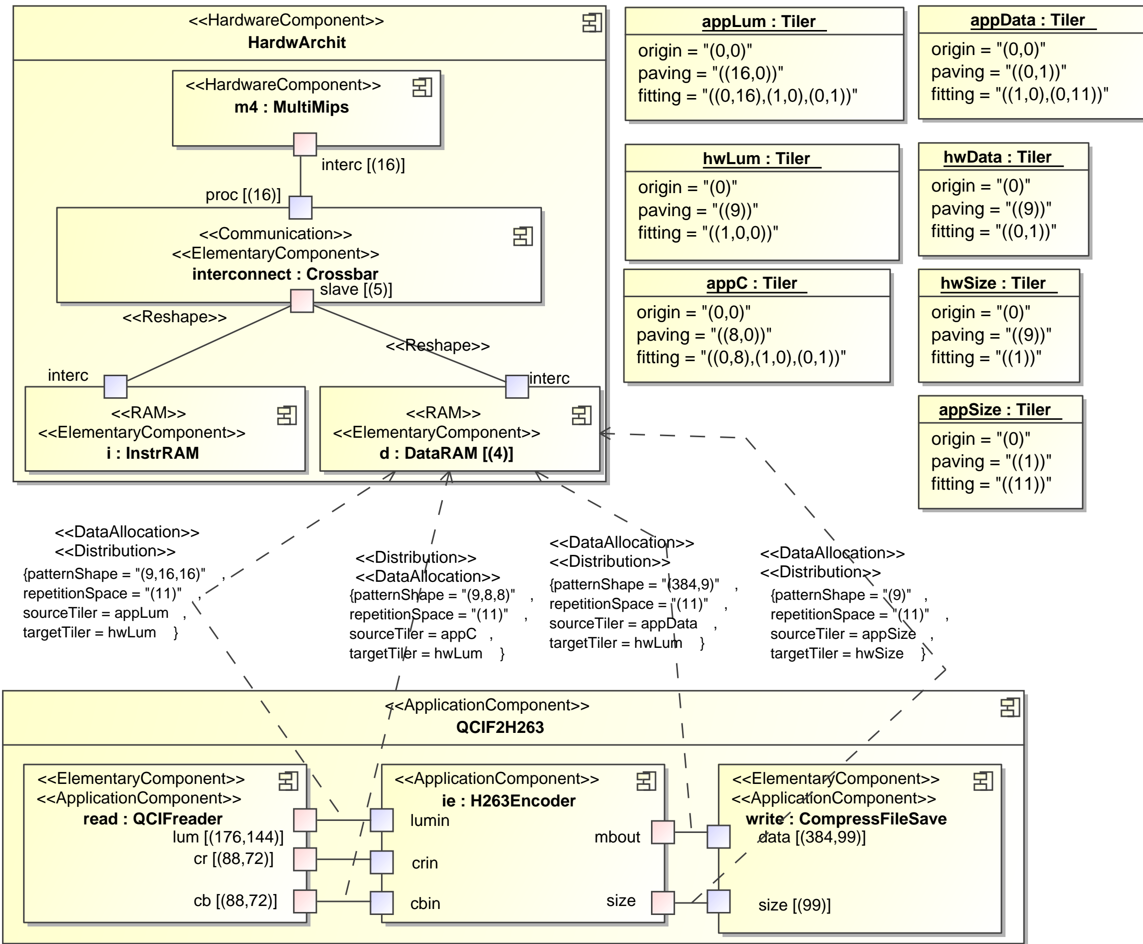


FIG. 6.12: Vue partielle de la distribution des tableaux intermédiaires de *QCIF2H263* sur les 4 bancs mémoires. La distribution suit la même organisation que la distribution des tâches afin que chaque processeur ne fasse des accès qu'à un seul banc mémoire.

sur lequel il est basé : PVT. À partir de l'architecture générée pour la simulation PA, de l'ISS MIPS au niveau PVT et du code logiciel pour la simulation CABA, nous avons assemblé manuellement une simulation PVT de la configuration à 16 processeurs. La simulation en PVT a pris 47s, dont 6,7% du temps a été dédié directement à la simulation des processeurs. Dans la simulation PA, qui a pris 15s, seulement 0,9% du temps a été dédié à la simulation des processeurs, soit un facteur d'accélération environ de 25 sur cette partie. À ce niveau d'abstraction le temps de simulation des processeurs est pratiquement négligeable. Par ailleurs, remarquons que même si uniquement les composants processeurs sont modifiés par rapport à PVT, la simulation PA réduit également le coût du noyau du simulateur SystemC car il évite l'ordonnancement entre chaque instruction simulée.

En faisant varier le nombre de processeurs, nous avons noté que, contre-intuitivement, passer de 12 processeurs à 16 diminue les performances. Nous allons tenter de pallier ce phénomène en faisant varier le nombre de bancs mémoire.

6.3.2 Variation du nombre de bancs mémoire

À partir des résultats de la simulation PA, il est possible d'observer les contentions d'accès à la mémoire. Remarquant que ces contentions étaient particulièrement élevées pour la configuration avec 16 processeurs, nous avons décidé d'augmenter le nombre de bancs mémoire, pour le faire passer de 1 à 2 puis 4. Dans le modèle, ces changements se traduisent d'abord au niveau du modèle d'architecture par l'ajout d'une multiplicité sur le composant *DataRAM* et l'augmentation du nombre de ports sur le crossbar. Pour être effectif ce changement doit être également répercuté sur les allocations de données dans le modèle d'association. Dans le modèle original les tableaux n'étaient pas distribués (puisqu'il n'y avait qu'une seule mémoire), la distribution des données a dû être spécifiée. Nous avons réparti chacun des cinq tableaux intermédiaires de *QCIF2H263* de manière similaire à la distribution de *H263MacroBlock* sur les processeurs. Ainsi, lors de l'encodage, chaque processeur ne travaille que sur un banc de mémoire. Identiquement à la distribution des tâches, la distribution de données s'adapte automatiquement au nombre de mémoires.

La figure 6.12 présente une vue partielle du modèle d'architecture modifié et de la distribution de données introduite. Pour garder la vue concise, seules quatre distributions sont représentées. Le fonctionnement des distributions de données est similaire à celui de la distribution de tâches schématisé figure 6.7 mais au lieu de prendre chaque tâche et de l'associer à un processeur, elle prend un *bloc d'éléments* et les associe tous à la même mémoire. Par exemple, pour la distribution du tableau transmit par le port *lum*, selon le *Tiler appLum* elle parcourt en 11 fois le tableau en prenant des colonnes de 9 blocs de 16×16 éléments —les blocs identiques à ceux lus par *H263MacroBlock*. Puis, selon le *Tiler hwLum* elle linéarise chaque l'ensemble obtenu des blocs sur les quatre mémoires —de la même manière que *H263MacroBlock* est placé sur les MIPS. Ainsi, le bloc de [0, 0] à [15, 15] est placé sur la première mémoire, le bloc de [0, 16] à [15, 31] est placé sur la deuxième mémoire, etc. En conséquence toutes les tâches sur le processeur 0 accéderont à la mémoire 0, de même pour les processeurs 4, 8, et 12.

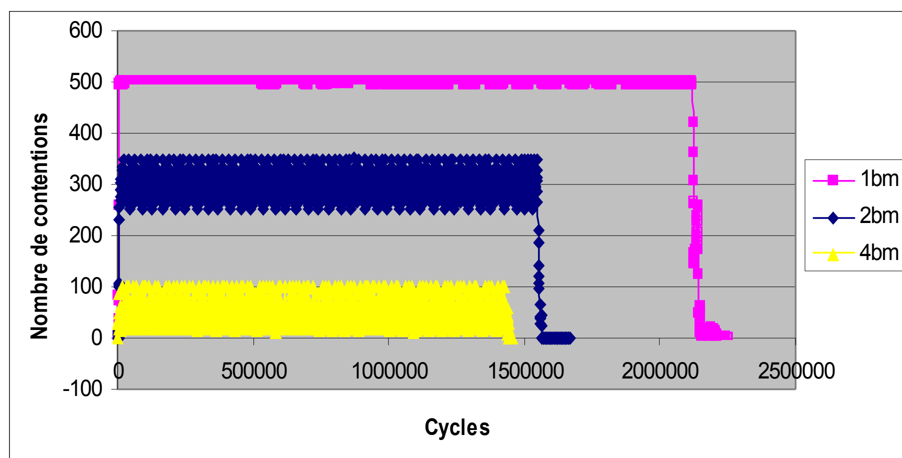
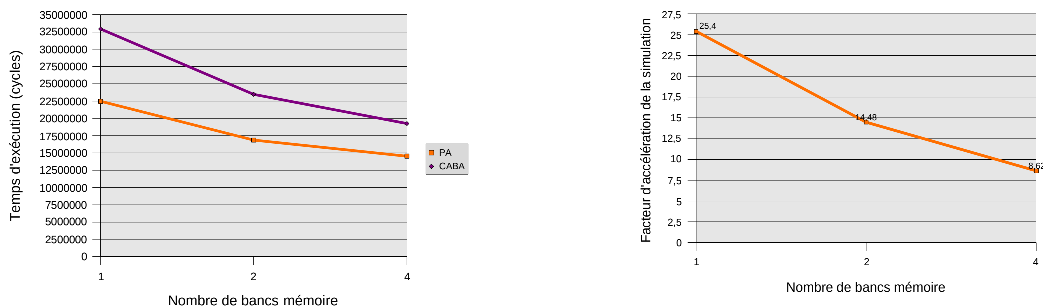


FIG. 6.13: Graphe des contentions sur le crossbar au cours de l'exécution simulée selon le nombre de bancs mémoire. Le nombre de contentions correspond à la quantité de contentions entre chaque période de 10µs.

Une fois la distribution définie, il est extrêmement aisé de tester le MPSoC avec différents

nombres de bancs mémoire : il ne suffit que de changer la multiplicité du composant mémoire et des ports. Les contentions d'accès à la mémoire de données mesurées dans la simulation PA sont représentées graphiquement dans figure 6.13. Chaque courbe correspond à un nombre de bancs mémoire différent. Chaque point d'une courbe correspond au nombre de contentions enregistrées par le crossbar pour accéder aux différents bancs dans un intervalle de 10µs. Il est clairement mis en évidence que l'augmentation du nombre de bancs mémoire permet de réduire les contentions : on passe d'une moyenne aux alentours de 500 contentions/10µs à moins de 100.



(a) Graphe du temps d'exécution simulé (en cycle)

(b) Graphe du facteur d'accélération du temps de simulation

FIG. 6.14: Comparaisons entre les simulations PA et celui CABA en fonction du nombre de bancs mémoire.

Les figures 6.14a et 6.14b présentent respectivement les graphes du temps d'exécution et du facteur d'accélération en fonction du nombre de bancs mémoire. On note de nouveau que la simulation PA permet de classer correctement les différentes configurations : plus le nombre de bancs mémoire augmente et plus l'exécution est rapide (cet effet est dû à la réduction des contentions). Par ailleurs, le facteur d'accélération du temps de simulation diminue lors de l'ajout des bancs mémoire. Cela s'explique simplement par le fait que proportionnellement la simulation des processeurs correspond à une moins grande partie de la simulation totale.

6.4 Synthèse

Dans ce chapitre une étude de cas de la modélisation d'un SoC a été exposée. Nous avons présenté la conception d'un MPSoC à base de processeurs MIPS et d'une mémoire partagée dédié à l'encodage H.263. D'une part cela fut l'occasion de montrer l'usage des propositions faites au chapitre 3 dans la modélisation complète d'un SoC. Ce fut également l'opportunité de démontrer le fonctionnement des transformations implémentées au cours de cette thèse et d'esquisser l'usage du flot de conception Gaspard. Nous avons vu que grâce à ce flot il est particulièrement facile de parcourir l'espace de conception : il suffit de modifier le modèle du MPSoC exprimé à l'aide de concepts de haut niveau et automatiquement la simulation peut être regénérée. Enfin, nous avons comparé la simulation au niveau PA à des simulations au niveau PVT et CABA. Des facteurs d'accélération respectivement jusqu'à 3 et 25 ont été mesurés. Ce gain de vitesse de simulation n'empêche pas une comparaison fiable des différentes alternatives qui peuplent l'espace de conception.

Conclusion de la première partie

Dans cette partie nous avons proposé des évolutions au méta-modèle Gaspard original afin de pouvoir modéliser entièrement un SoC, sans zones d'ombre sur la sémantique d'association ni sur la réalisation des composants élémentaires. Les mécanismes de déploiement introduits favorisent en plus l'emploi de bibliothèques d'IP et d'implémentations indépendantes de la cible de compilation.

Nous avons ensuite défini un haut niveau d'abstraction pour la co-simulation qui, contrairement aux niveaux d'abstraction usuels s'appliquant uniquement au matériel, permet de cacher les détails de l'application. À ce niveau, seuls l'agencement des accès aux données et les synchronisations entre les tâches sont pris en compte. Après avoir défini une projection du modèle de calcul des applications Gaspard vers un modèle d'exécution, nous avons spécifié la simulation de ce modèle d'exécution selon le niveau d'abstraction PA.

Par la suite nous avons présenté la chaîne de transformations créée qui permet de générer un code SystemC/PA directement à partir d'un modèle de SoC. Cette chaîne suit les recommandations de l'IDM. En particulier, des méta-modèles ont été définis pour caractériser les entrées et les sorties intermédiaires des transformations. La première transformation détaillée est commune à un ensemble de cibles de compilation, elle vise principalement à interpréter l'association, tandis que la seconde est spécifique à SystemC/PA et permet de générer le code de simulation du SoC.

Enfin, nous avons présenté l'usage des différentes propositions à travers un exemple de MPSoC dédié à l'encodage vidéo. Outre la validation globale du flot de conception, cela a mis en avant la vitesse d'exécution d'une simulation PA ayant malgré cela une précision suffisante qui favorise une exploration de l'espace de conception.

Deuxième partie

Ordonnancement dynamique de systèmes temps-réel parallèles

Chapitre 7

Calcul haute-performance et temps-réel

7.1 Multiprocesseur et approches temps-réel **172**
7.2 Temps-réel et Linux **173**
7.2.1 Le système standard GNU/Linux 173
7.2.2 Approche co-noyau 174
7.2.3 Approche multiprocesseur asymétrique 175
7.3 Ordonnancement multiprocesseur dans le noyau Linux **176**
7.3.1 Ordonnancement des tâches 176
7.3.2 Équilibrage de charge 177

Après avoir abordé, dans la partie précédente, la génération de MPSoC dans sa globalité, de la définition des outils de modélisation à la simulation, nous allons maintenant nous attacher à l'ordonnancement dynamique des tâches. Notre contribution constitue une nouvelle approche visant à faire co-habiter les propriétés temps-réel, dont on a besoin dans les systèmes embarqués, aux systèmes multiprocesseurs, dont on a besoin pour le traitement intensif de données. À partir de l'instant où l'on cherche à étendre le domaine d'application du traitement systématique au traitement intensif, il faut être capable de répondre aux évènements extérieurs de manière adéquate et dans un temps raisonnable. Ces besoins correspondent aux propriétés temps-réel.

Historiquement, les notions de calcul haute-performance (HPC, pour *High Performance Computing*) et de temps-réel ont souvent été considérées antinomiques, la seconde étant usuellement associée aux systèmes embarqués. De nos jours, cette stricte distinction ne tient plus et de nombreuses applications (radar, nœud de traitement des communications, environnement d'immersion virtuelle. . .) peuvent bénéficier de la conjonction de ces deux propriétés. À notre connaissance, il n'y a encore aucun système complètement mis au point capable de procurer simultanément les avantages des deux approches. Nous proposons ici une solution logicielle basée sur une architecture matérielle multiprocesseur faisant cohabiter ces deux notions.

Nous visons ici le temps-réel dit *dur*. C'est-à-dire la propriété du système à être capable de *toujours* répondre aux évènements extérieurs dans un temps borné. Il existe également le temps-réel dit *mou* qui ne vise à borner le temps de réponse que pour la *majorité* des cas, par exemple pour 99,999% des évènements reçus. Remarquons que la distinction temps-réel

dur/mou ne dépend ni de la borne ni de la vitesse d'exécution. On pourrait imaginer un système temps-réel dur de contrôle d'une centrale nucléaire qui doit réagir dans la seconde tout autant qu'un système d'enregistrement multimédia qui possède des capacités de temps-réel mou de réaction de l'ordre du millième de seconde. De plus, un système performant en terme de puissance de calcul n'est pas forcément plus prompt à réagir *à chaque fois* rapidement aux entrées du système. D'ailleurs, les évolutions actuelles sur la performance ont tendance à améliorer la vitesse d'exécution pour la plupart des situations tout en dégradant la vitesse dans des cas relativement rares, en opposition aux attentes du temps-réel dur.

Les travaux présentés dans cette partie ont été effectués dans le cadre du projet européen Hyades [4]. Ce projet, mêlant industriels et académiques, a débuté dans le contexte que nous venons de présenter. Spécifiquement, deux applications, proposées par les partenaires industriels, ont été ciblées. La première est une plate-forme de diffusion de flux multimédia MPEG4, avec une gestion dynamique de la qualité de service propre à chaque destinataire du flux. La seconde application est un environnement d'immersion virtuelle permettant à l'utilisateur d'agir sur des objets virtuels via une interface haptique qui permet aussi de ressentir la force nécessaire à déplacer les objets. Pour être réaliste le retour de force par rapport au mouvement de l'utilisateur doit se faire dans un temps borné très court, et nécessite une grande puissance de calcul pour détecter les collisions entre objets et simuler les forces physiques.

7.1 Multiprocesseur et approches temps-réel

L'usage de multiprocesseurs symétriques (SMP, pour *Symmetric Multi-Processors*) pour faire face au besoin de puissance de calcul est une solution efficace et rependue. Elle a déjà été testée dans le contexte du temps-réel [13]. Pour tirer partie pleinement d'une architecture SMP, le système d'exploitation doit mettre en œuvre un mécanisme de mémoire partagée, de migration et d'équilibrage de charge entre processeurs, ainsi que les différents moyens de communication entre tâches. La complexité d'un tel système d'exploitation requise par le parallélisme le fait de suite ressembler plus à un système d'exploitation généraliste (GPOS, pour *General Purpose Operating System*) qu'à un système dédié tel que les systèmes d'exploitation temps-réel (RTOS, pour *Real-Time Operating System*). Un RTOS visant une architecture SMP doit implémenter tous ces mécanismes et soigneusement considérer les interférences avec les contraintes temps-réel dur. Cette complexité contribue largement au fait que la plupart des RTOS soient monoprocesseurs seulement.

Dans leur état de l'art des RTOS actuels, Stankovic et Rajkumar [91] décrivent une taxonomie complète de ce type de système d'exploitation. Ceux développés en entier seulement pour un projet sont des *espèces en danger*, principalement à cause de la complexité à implémenter toutes les fonctionnalités requises de nos jours par les développeurs d'applications. La gestion d'une architecture SMP fait partie de ces difficultés. Dans notre situation, le travail d'ingénierie nécessaire pour produire un système à la fois temps-réel et adapté aux SMP serait trop coûteux en temps et en argent.

Une autre approche est de se baser sur un système d'exploitation spécialement écrit pour être réutilisable, à partir duquel le concepteur peut choisir un ensemble de fonctionnalités qui seront utiles pour le matériel ciblé. RTEMS [75, 93] en est un exemple. Cet RTOS libre supporte les architectures multiprocesseurs. Néanmoins, le support SMP est limité, entre autres les tâches ne peuvent pas être déplacées d'un processeur à un autre, leur placement est

fixé lors du développement.

Les noyaux « de recherche » sont des petits systèmes d'exploitation conçus afin de présenter un ou plusieurs nouveaux paradigmes permettant de répondre à un problème donné. Bien que cela puisse être une approche intéressante soit lorsque les solutions disponibles sont vraiment très pauvres soit parce que le paradigme est beaucoup plus simple à utiliser ou à comprendre, il est rarement efficace de nos jours de forcer les utilisateurs à complètement re-considérer l'organisation du système sur lequel doivent reposer leurs applications (par exemple en fournissant une API complètement novatrice ou en introduisant des nouveaux concepts pour représenter les briques de base de la programmation).

La dernière approche que nous présentons consiste à ajouter une extension temps-réel à un GPOS. Cela a l'avantage de fournir aux utilisateurs toutes les fonctionnalités des systèmes généralistes, y compris les facilités de développement de logiciel. Dans la section suivante nous allons voir plus en détails les différentes alternatives de cette approche en utilisant Linux comme GPOS de base.

7.2 Temps-réel et Linux

Le noyau Linux est capable de gérer efficacement les plates-formes SMP, mais n'a jamais été développé dans le but d'être un RTOS. Même s'il existe un ordonnancement dédié au temps-réel, via les politiques d'ordonnancement FIFO et round-robin, seules des tâches temps-réel mou sont supportées, aucune garantie de temps de réaction n'est assurée. McKenney [67] a décrit en détail le grand nombre de solutions qui ont fleuri ces dernières années autour du noyau Linux pour lui permettre d'exhiber les propriétés d'un système temps-réel. Cette multitude de propositions trouve sa source dans l'adoption par les développeurs de ce jeune système d'exploitation, attirant les utilisateurs de temps-réel par ses nombreux avantages tels la facilité de développement et la liberté d'extension du système. En plus des RTOS dédiés qui ont été adaptés pour supporter l'API Linux, les approches varient d'une séparation pratiquement totale entre le noyau original et la partie temps-réel à un mélange complet des deux aspects.

7.2.1 Le système standard GNU/Linux

Apparue récemment (dans la version 2.6 du noyau), une option disponible à la compilation permet la « préemptivité du noyau ». Proposée initialement par MontaVista, vendeur d'un Linux pour systèmes embarqués, cette option [71, 72] permet le réordonnancement d'une tâche à l'intérieur même du noyau. Par conséquent, lorsqu'une interruption matérielle est reçue, il n'est plus nécessaire d'attendre que l'on ait quitté l'espace noyau pour changer de contexte et donner la main à la tâche associée à l'interruption. Pour pouvoir assurer l'atomicité nécessaire de certaines sections, la fonction `preempt_disable()` permet d'inhiber temporairement la propriété de préemptivité (tandis que `preempt_enable()` la réactive).

La préemptivité du noyau améliore effectivement les latences utilisateur (le temps que met une tâche à être réordonnancée après qu'une interruption à laquelle elle est associée soit déclenchée). Cependant les garanties restent dans l'optique de temps-réel mou : la très grande majorité des latences sont courtes mais il y en a parfois qui sont beaucoup plus longues. Ces garanties sont suffisantes pour les tâches de traitement multimédia ciblées lors de l'intégration dans le noyau, mais le temps-réel dur ne peut être assuré.

Actuellement Ingo Molnar et Steven Rostedt continuent à travailler dans cette direction en développant un patch[1] pour le noyau Linux qui cible des latences à la fois plus faibles et plus constantes. Ce projet nommé « preempt-rt » tente de permettre la préemption de n'importe quelle partie du noyau, y comprit les sections critiques et les gestionnaires d'interruption. Pour cela des techniques spéciales ont été introduites telles que les RCU [68] pour accéder aux données partagées ou bien l'usage de threads pour l'exécution des gestionnaires d'interruption. L'inconvénient de cette approche est la légère perte en performance, et surtout les grandes difficultés techniques d'implémentation et de maintenance. Le code du noyau a dû être entièrement parcouru pour l'adapter à ces modifications. Il est très difficile d'assurer qu'il ne manque pas, par exemple dans un pilote de périphérique, d'autres adaptations. Le meilleur moyen de mettre en perspective ces difficultés est de noter qu'après plus de trois ans de travail par plusieurs ingénieurs à temps complet, le projet n'est pas encore complété.

Pour obtenir des garanties plus fortes et de manière plus simple plusieurs solutions impliquant le noyau Linux ont déjà été proposées, elles peuvent être regroupées en deux approches clairement distinctes.

7.2.2 Approche co-noyau

La première de ces approches se base sur la présence simultanée d'un second noyau spécialisé dans le traitement temps-réel. Les projets RTAI [25] et RTLinux [44, 101] proposent de telles solutions. I-Pipe [48] est un nouveau projet qui permet de mettre en place facilement un tel co-noyau. De manière générale l'implantation de tels systèmes consiste en un petit noyau (souvent nommé micro-noyau) qui met à disposition les services temps-réel et qui ordonnance le noyau Linux comme une tâche de priorité faible, lorsqu'aucune tâche temps-réel n'est éligible. Le fonctionnement se fait par une virtualisation des interruptions, qui ne sont jamais véritablement masquées. Cela permet de préempter le noyau Linux à n'importe quel instant. L'architecture est représentée par la figure 7.1.

Le micro-noyau est suffisamment compact et possède un nombre de chemins d'exécution suffisamment petit pour qu'il soit techniquement possible de *prouver* le temps de réponse à une interruption comme étant borné. Cette preuve est impossible à obtenir sur le noyau Linux en raison de sa trop grande complexité. Les latences obtenues sur un tel système sont particulièrement bonnes, de l'ordre de la dizaine de microsecondes sur une architecture ordinaire. L'inconvénient majeur cependant est que le modèle de programmation est *dual*, les tâches temps-réel n'ont pas accès aux fonctions de Linux et doivent se restreindre à l'API limitée proposée par le micro-noyau. Inversement, les tâches Linux n'ont aucun moyen de bénéficier des garanties temps-réel. Des mécanismes plus ou moins aisés à utiliser ont été réalisés pour permettre la communication et l'interaction entre tâches Linux et tâches temps-réel du micro-noyau, mais aucun ne peut effacer les contraintes de programmation. Par exemple, dans RTLinux le mécanisme qui permet ce type de communication est appelé LXRT, via une API spécifique il autorise une tâche à avoir un thread temps-réel et un thread dans l'espace Linux, ils peuvent alors communiquer à l'aide de la mémoire partagée.

La seconde approche pour procurer à Linux des propriétés temps-réel ne possède pas cet inconvénient.

[1] http://www.kernel.org/pub/linux/kernel/projects/rt/

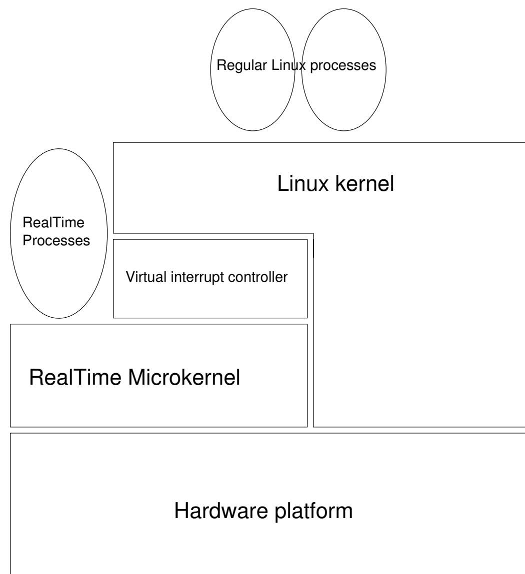


FIG. 7.1: Architecture d'un système co-noyau.

7.2.3 Approche multiprocesseur asymétrique

Cette approche exploite l'architecture SMP et introduit la notion de processeur « protégé » (*shielded*), c'est-à-dire le principe de multiprocesseur asymétrique. Sur une machine multiprocesseur il existe alors deux types de processeurs : ceux spécialisés pour le temps-réel qui n'exécutent que des tâches temps-réel et ceux qui exécutent toutes les tâches non temps-réel. En plus, les processeurs temps-réel sont délestés des traitements d'interruption qui ne sont pas associées directement à une tâche temps-réel. Même si le mélange des termes SMP et asymétrique peut paraître antinomique, cette solution reste basée sur le modèle SMP de Linux et le concept d'asymétrie est introduit par dessus à l'aide de modifications très minimes. Des systèmes tels que CCC RedHawk Linux [23, 22], maintenant proposé par Novell [2], ou SGI REACT/pro pour IRIX [88] utilisent cette approche et peuvent annoncer des latences inférieures à la milliseconde.

Dans ce modèle le programmeur n'a à faire aucune distinction entre une tâche temps-réel et une tâche standard, tout se fait à la configuration du système. À l'aide de mesures nous avons vérifié les faibles latences d'interruption procurées par cette solution. Cependant, contrairement à l'approche co-noyau, il est impossible de certifier un temps de réponse théorique maximum étant donné la complexité du noyau Linux sur lequel les tâches sont ordonnancées. En particulier, les tâches temps-réel se doivent de ne pas appeler des fonctions susceptibles de perturber la réactivité de l'ordonnanceur (par exemple l'écriture sur le disque dur). En plus, et c'est l'une des restrictions des plus importantes, comme seules les tâches temps-réel peuvent être exécutées sur les processeurs temps-réel, de la ressource CPU est gâchée dès que ces tâches n'utilisent pas toute la puissance disponible du processeur.

La solution ARTiS que nous proposons étend cette approche en autorisant également les tâches standard à être exécutées sur les processeurs protégés tant qu'elles ne compromettent pas les propriétés temps-réel. Avant de voir plus en détail notre proposition, nous allons

[2] http://www.novell.com/products/realtime/

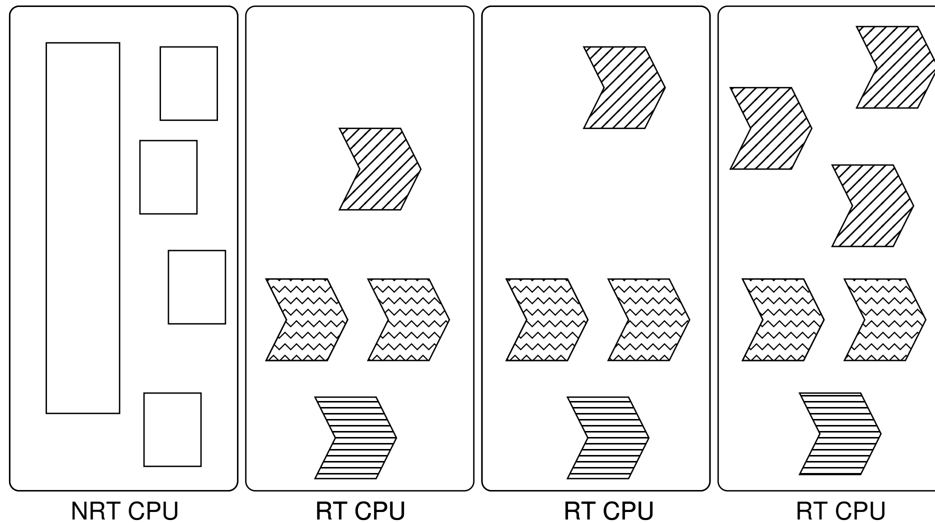


FIG. 7.2: Schéma d'une architecture basée sur l'asymétrie entre les processeurs pour garantir le temps-réel.

détailler le fonctionnement de l'ordonnancement dans Linux.

7.3 Ordonnancement multiprocesseur dans le noyau Linux

7.3.1 Ordonnancement des tâches

Dans Linux, comme dans tous les systèmes POSIX, l'ordonnancement des tâches est préemptif : le système d'exploitation peut décider d'interrompre une tâche en cours d'exécution pour en exécuter une autre. Dans Linux ce réordonnancement est effectué avec une fréquence fixe, par exemple sur IA-64 cette fréquence est de 1024Hz [73]. À chaque tic d'ordonnancement, ainsi qu'à chaque fois qu'un pilote de périphérique détecte un besoin, la fonction `schedule()` est appelée. Elle détermine la meilleure tâche à ordonnancer et réalise le changement de contexte entre la tâche précédente et la tâche suivante.

Une priorité et une politique d'ordonnancement sont associées à chaque tâche. Plus la priorité (représentée par un nombre entier) est élevée[3], plus une quantité importante de temps de calcul sera attribuée à la tâche. Il existe trois politiques d'ordonnancement possibles pour une tâche :

- `SCHED_OTHER` (ordonnancement normal) fait en sorte de répartir équitablement le temps du processeur entre les tâches tout en procurant plus de temps aux tâches ayant une priorité élevée qu'à celles ayant une priorité basse.
- `SCHED_FIFO` (ordonnancement First In/First Out) est utilisé dans le cadre du temps-réel, il exécute chaque tâche les unes après les autres sans jamais interrompre une tâche en cours d'exécution. En d'autres termes, c'est un ordonnancement non-préemptif. Seules les tâches partageant la priorité la plus élevée sont sélectionnées, et donc les tâches à priorité inférieure ne s'exécutent jamais tant qu'une tâche à plus haute priorité est prête.

[3]Notons que, contre-intuitivement, une priorité élevée correspond à une petite valeur et une priorité faible correspond à une grande valeur.

– `SCHED_RR` (ordonnancement *round-robin*) est aussi utilisé dans le cadre du temps-réel, il partage le temps processeur en des quantums de temps identiques pour chaque tâche de la priorité la plus élevée. Les tâches qui n'ont pas la priorité la plus élevée ne se voient allouées aucun quantum de temps.

L'ordonnanceur recherche d'abord une tâche ordonnancée selon la politique `SCHED_FIFO` ou `SCHED_RR`, en commençant par les niveaux de priorité les plus élevés. Si aucune tâche n'est disponible alors il choisit parmi les tâches ordonnancées selon la politique `SCHED_OTHER` la tâche qui n'a pas été exécutée depuis le plus longtemps.

Comme nous l'avons mentionné, l'ordonnanceur est préemptif, c'est-à-dire que ce n'est pas la tâche qui décide de rendre la main mais c'est le système qui décide d'interrompre une tâche lors de son exécution. Du point de vue de la tâche, un changement de contexte peut arriver n'importe quand. Néanmoins, d'un point de vue global au système, le changement de contexte ne peut pas toujours être immédiatement effectué : il y existe un certain nombre de restrictions lorsque le noyau lui-même est en cours d'exécution, ce qui arrive soit suite à un appel système par une tâche, soit suite à une interruption matérielle. En effet, il existe dans le noyau des *sections critiques* qui correspondent à des parties de code qui ne doivent pas être interrompues : dans le but de protéger l'intégrité des structures de données telles que les listes chaînées (une section critique correspond dans ce cas à un verrou) ou dans le but d'assurer le respect du protocole de communication lors du contrôle de périphériques. Ces sections critiques peuvent entraîner de longues périodes sans réordonnancement, c'est ce qui empêche le noyau de garantir des contraintes temps-réel.

Dans le noyau Linux 2.6 l'implémentation de l'ordonnanceur est nommée O(1) (d'après sa complexité). La *file d'exécution*, une structure de données qui contient une référence vers les tâches en attente d'exécution, est décomposée en deux parties : une file active qui contient la liste des tâches qui sont encore éligibles à un quantum de temps processeur et une file non-active qui liste les tâches ayant déjà consommé leur quota de temps imparti. Au fur et à mesure de l'ordonnancement, les tâches passent de la file active à la file non-active. Lorsque la file active devient vide, les deux files sont inversées et les tâches sont exécutées de nouveau au fur et à mesure. Chaque processeur d'une machine SMP possède sa propre file d'exécution indépendante des autres processeurs, cela améliore les performances de l'ordonnanceur par rapport à une seule file d'exécution partagée par tous les processeurs mais nécessite un mécanisme d'équilibrage de charge.

7.3.2 Équilibrage de charge

Afin de fournir le meilleur usage de l'ensemble des processeurs, il est nécessaire d'avoir un mécanisme explicite qui s'assure que chaque processeur reçoit approximativement la même charge de travail. On appelle ce mécanisme *l'équilibrage de charge*. L'idée générale est de détecter les cas déséquilibrés et alors de déplacer des tâches d'un processeur à un autre pour rééquilibrer la charge. Dans le noyau, l'implémentation d'un tel mécanisme doit répondre aux questions de savoir quand procéder à la détection, comment détecter le déséquilibre, comment choisir les tâches à migrer et aussi comment migrer une tâche entre deux files d'exécution [66]. Dans Linux, ce mécanisme est réparti, chaque processeur exécute le même algorithme d'équilibrage.

La détection du déséquilibre est faite par la fonction `load_balance()`. Cette fonction est appelée lorsque la file d'exécution du processeur est vide (le processeur est oisif), et également périodiquement (environ 5 fois par seconde). L'état de déséquilibre de charge du système est

détecté au début de `load_balance()` avec l'appel à la fonction `find_busiest_queue()`. Elle compare la taille de la plus grande file d'exécution du système à la file d'exécution du processeur local, si cette dernière est au moins 25% plus courte que la file la plus longue alors l'état de déséquilibre est décrété.

Dans cette situation, un verrou est pris sur la file d'exécution la plus longue, en plus du verrou sur la file locale déjà acquit. Le nombre de tâches à déplacer est calculé de manière à ce que les deux files soient de même longueur après l'équilibrage. Ce sont les tâches de la partie non-active de la file distante qui sont choisies en premier lieu, puis celles de la partie active. Les tâches les plus prioritaires sont sélectionnées en premier. Enfin il est vérifié que la tâche n'a pas été exécutée trop récemment (car alors il est probable qu'elle ait encore des données en mémoire cache et donc son attachement au processeur est encore fort). La migration est alors effectuée simplement en déplaçant la référence de chaque tâche choisie de la file d'exécution distante à la file locale. C'est ainsi que le mécanisme d'équilibrage se termine.

Nous venons de passer en revue les mécanismes utilisés par le noyau Linux pour l'ordonnancement de tâches. Ils prennent en compte les architectures multiprocesseurs. Par la suite nous allons tout d'abord présenter le fonctionnement d'ARTiS d'un point de vue théorique puis nous détaillerons l'implémentation comme extension au noyau Linux sur système multiprocesseur. Enfin nous présenterons la validation expérimentale du système final, en mettant l'accent sur trois aspects du système : les latences d'interruption, la variation du temps d'exécution et l'équilibrage de charge.

Chapitre 8

ARTiS : Un ordonnanceur temps-réel asymétrique

8.1 Partitionnement des processeurs et des processus **180**
8.2 Mécanisme de migration **181**
8.3 Politique d'équilibrage de charge **182**
8.4 Mécanismes de communications asymétriques **182**
8.5 Synthèse **182**

À partir du constat qu'il n'existe pas actuellement de systèmes capables de proposer à la fois calculs haute-performance et temps-réel sans requérir de la part du développeur de programmer et déboguer différemment les parties temps-réel et les parties à priorité moins élevée. Nous proposons ARTiS, une extension temps-réel pour les architectures multiprocesseurs. La méthode proposée à travers ARTiS combine les avantages à la fois des systèmes d'exploitation généralistes et des systèmes d'exploitation temps-réel en mettant en place par-dessus une architecture SMP un ordonnancement temps-réel asymétrique (**A**symmetric **R**eal-**Ti**me **S**cheduler). L'idée est de se baser sur la couche d'exploitation d'un GPOS pour prendre en charge les aspects liés à l'architecture multiprocesseur et d'introduire une couche supplémentaire assurant les contraintes temps-réel. L'objectif est de garantir à tous les processus l'accès aux services fournis par le GPOS, de conserver la possible utilisation de l'ensemble des ressources du SMP, tout en proposant un comportement apte pour le temps-réel dur [11, 6]. Le cœur de cette méthode est la nette distinction entre des processeurs temps-réel et non temps-réel ainsi que la migration automatique de toute tâche tentant de désactiver la préemption sur un processeur temps-réel.

Le modèle de programmation d'ARTiS permet l'usage de tâches temps-réel écrites dans l'espace utilisateur : le programmeur peut utiliser les API conventionnelles (POSIX, Linux...) pour développer son application. Les tâches temps-réel d'ARTiS sont temps-réel dans le sens où elles sont identifiées par une haute priorité d'ordonnancement et ne sont jamais perturbées par des activités non temps-réel. Le temps de réponse aux tâches temps-réel de plus haute priorité est borné (la valeur de cette borne dépend de la plate-forme d'exécution).

Un exemple d'architecture typique d'un système basé sur ARTiS est présenté figure 8.1. Dans ce chapitre, nous commencerons par détailler différents types de processeurs (RT/NRT) et différents types de tâches (RT/Standard) que l'on peut distinguer sur le schéma. Puis nous présenterons les mécanismes de migration et d'équilibrage de charge, représentés dans la

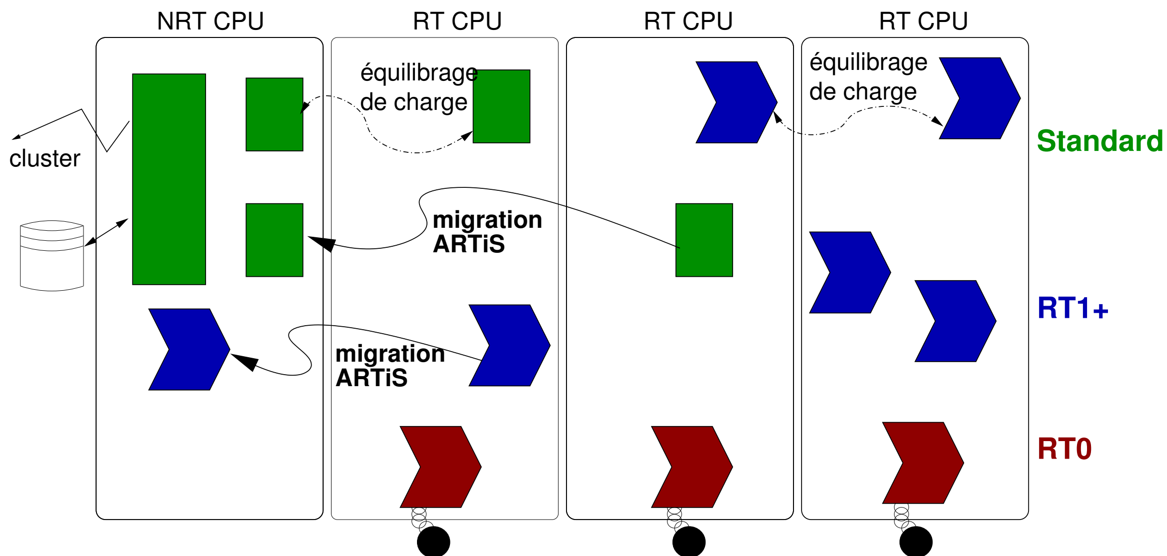


FIG. 8.1: Exemple d'usage typique d'un système basé sur ARTiS. L'application est séparée selon différents niveaux de priorité temps-réel. Les tâches sont déplacées par les mécanismes de migration et d'équilibrage de charge.

figure par les flèches. Enfin, nous parlerons des mécanismes de communication entre les tâches.

8.1 Partitionnement des processeurs et des processus

Les processeurs sont partitionnés en deux ensembles, un ensemble de CPU NRT (*Non Real-Time*) et un ensemble de CPU RT (*Real-Time*). Chacun possède une politique d'ordonnancement différente. Le but est d'assurer la meilleure latence d'interruption possible pour certaines tâches s'exécutant sur l'ensemble de processeurs RT.

Deux classes de tâches temps-réel sont définies. Ce sont toutes des tâches temps-réel du point de vue de la priorité d'ordonnancement, mais elles diffèrent dans leur association aux processeurs :

- une, ou plusieurs tâches temps-réel sont attachées à chacun des CPU RT. Elles sont nommées *RT0*. Ce sont les tâches temps-réel de plus haute priorité. Ces tâches ont la garantie que le processeur leur sera entièrement dédié tant que nécessaire. Ces tâches sont les seules autorisées à être non-préemptibles sur leur processeur. Cette propriété leur assure une latence minimale. Ces tâches RT0 sont les tâches temps-réel dur d'ARTiS. L'exécution de plus d'une tâche RT0 sur un même processeur est possible ; développeur se doit d'assurer la faisabilité de l'ordonnancement de ces tâches pour que les latences puissent être garanties.
- sur chacun des processeurs, d'autres tâches temps-réel peuvent s'exécuter, mais *uniquement* en mode préemptible. En fonction de leur priorité, ces tâches sont nommées RT1, RT2. . .ou RT99. De manière générale, ces tâches sont désignées par tâches *RT1+*, tâches temps-réel de priorité 1 et plus. Elles peuvent profiter efficacement des ressources si les tâches RT0 ne consomment pas tout le temps CPU. Pour assurer la meilleure latence aux

tâches RT0, les tâches RT1+ sont automatiquement migrées par l'ordonnanceur ARTiS sur un CPU NRT si elles sont sur le point de devenir non-préemptibles (quand elles appellent une des fonctions `preempt_disable()` ou `local_irq_disable()`). Les tâches RT1+ sont les tâches temps-réel mou d'ARTiS, équivalent au temps-réel garanti par le noyau de base du système d'exploitation généraliste. Elles n'ont pas de garanties temporelles fermes, mais leurs demandes sont prises en compte selon une politique du meilleur effort (*best effort*). Elles sont aussi le support des parties les plus calculatoires des applications visées.
– les autres tâches, non temps-réel, sont nommées tâches *standard* dans la terminologie ARTiS. Elles ne sont liées à aucune contrainte temporelle. Elles peuvent coexister avec des tâches temps-réel et sont éligibles par l'ordonnanceur tant qu'aucune tâche temps-réel ne requiert le processeur. Comme pour les tâches RT1+, les tâches standard sont automatiquement migrées vers un CPU NRT si elles tentent d'entrer dans une section de code non-préemptible.

Les CPU NRT exécutent principalement des tâches standard. Ils exécutent aussi les tâches RT1+ quand celles-ci sont non-préemptibles. Pour assurer l'équilibrage de charge du système, chacune de ces tâches peut être déplacée sur un processeur RT, mais seulement dans l'état préemptible. Quand une tâche RT1+ s'exécute sur un CPU NRT, elle conserve son niveau de priorité élevé par rapport aux tâches standard.

Ainsi ARTiS supporte trois niveaux d'exécution temps-réel : RT0, RT1+, et standard. Le niveau RT0 garantit de ne pas avoir à pâtir de latences élevées dues à l'exécution de sections non-préemptibles sur le même processeur. Les tâches RT1+ sont des tâches temps-réel mou mais peuvent bénéficier pleinement de l'architecture SMP, utile entre autres pour le calcul intensif. Enfin, les tâches standard sont exécutées sans affecter les CPU RT. Elles peuvent tirer partie des ressources de l'architecture SMP. Ce partitionnement est adapté à de larges applications construites à l'aide de plusieurs composants nécessitant différents niveaux de garantie temps-réel et de puissance processeur.

8.2 Mécanisme de migration

Le noyau Linux possède déjà un mécanisme de migration de tâche, utilisé pour l'équilibrage de charge, cependant pour ARTiS un mécanisme de migration spécifique a été défini. L'objectif de ce mécanisme est d'assurer une latence aussi faible que possible pour les tâches RT0. Toute tâche RT1+ ou standard s'exécutant sur un processeur RT est automatiquement migrée vers un processeur NRT quand elle tente de désactiver la préemption. Un changement fondamental par rapport au mécanisme original de Linux est la nécessité d'éviter la prise de verrous inter-processeurs. Ce type de verrous est extrêmement dangereux pour les propriétés temps-réel des CPU RT : il pourrait devoir attendre un CPU NRT, dont les propriétés sont beaucoup plus lâches. Pour effectivement migrer les tâches, les CPU RT et NRT doivent communiquer via des queues. Basé sur les travaux de Valois [98], ARTiS implémente un mécanisme de FIFO sans verrou pour un producteur et un consommateur assurant l'absence d'attentes actives de l'ordonnanceur ARTiS.

8.3 Politique d'équilibrage de charge

Une politique efficace d'équilibrage de charge assure la pleine exploitation d'un système SMP. Habituellement, l'équilibrage de la charge consiste à déplacer des tâches d'un processeur à un autre afin qu'aucun des processeurs ne soit oisif alors que des tâches sont en attente d'ordonnancement sur un autre processeur. Dans le cas d'ARTiS, les spécificités du système doivent être prises en compte. Les tâches RT0 ne doivent jamais migrer, par définition. Les tâches RT1+ devraient revenir sur les CPU RT plus rapidement, en priorité sur les tâches Linux : ces processeurs offrant des garanties temps-réel que les CPU NRT n'offrent pas. En vue de minimiser la latence sur les CPU RT et de fournir les meilleures performances au niveau du système global, nous avons proposé [9, 8] des algorithmes spécifiques et asymétriques d'équilibrage de charge.

8.4 Mécanismes de communications asymétriques

ARTiS met à disposition des mécanismes de communications asymétriques. Au sein des machines SMP, les tâches échangent des données par lecture/écriture dans la mémoire partagée. Pour assurer une cohérence à ces échanges, des sections critiques sont nécessaires. Ces sections critiques sont protégées d'accès concurrents simultanés par des mécanismes de verrous. Ce schéma de communication n'est pas adapté à notre cas particulier : un échange de données entre une tâche RT0 et une tâche RT1+ entraînerait une migration de la tâche RT1+ avant que celle-ci ne prenne le verrou afin de l'empêcher d'entrer dans un état non-préemptible sur un CPU RT. Un schéma de communication asymétrique doit autoriser la communication entre tâches sans impliquer de migration (qui est pénalisant pour la performance). Cela est effectué par l'usage de FIFO sans verrou dans un contexte un écrivain/un lecteur.

8.5 Synthèse

Nous venons de proposer un modèle de système capable de mixer propriétés temps-réel et calcul haute performance. L'approche est basée sur le partitionnement d'un système multiprocesseur entre processeurs RT, où les tâches sont garanties des propriétés temps-réel, et processeurs NRT, où tout code qui pourrait mettre en danger les propriétés temps-réel est exécuté. Ce partitionnement n'empêche pas l'usage de tous les processeurs pour les tâches de calcul, il implique uniquement qu'une tâche sera automatiquement et temporairement migrée si elle désactive la préemption. De plus, nous avons proposé des politiques d'équilibrage de charge spécifiques à cette asymétrie de manière à maintenir optimal l'usage de tous les processeurs.

Dans le chapitre suivant nous allons présenter une implémentation de ces mécanismes, réalisée en vue de prouver la faisabilité de nos propositions.

Chapitre 9

Mise en œuvre

9.1 Migration dans ARTiS **183**
9.1.1 Déclenchement de la migration 184
9.1.2 Déroulement de la migration d'une tâche 185
9.1.3 RT-FIFO : des FIFO sans verrou 186
9.2 Équilibrage de charge **186**
9.2.1 Contraintes spécifiques à ARTiS 187
9.2.2 Pondération de la longueur de la file d'exécution 188
9.2.3 Suppression des verrous inter-processeurs 190
9.2.4 Estimation de la prochaine tentative de migration 190
9.2.5 Association du type de tâche au type de processeur 192
9.3 Déploiement d'applications temps-réel **192**
9.3.1 Exemple de déploiement d'application 193
9.3.2 Configuration du système 194
9.3.3 Identification des tâches temps-réel 195
9.4 Synthèse de l'implémentation **196**

Le modèle ARTiS a été implémenté [7, 5] comme une modification du noyau Linux 2.6. Cette implémentation a été réalisée pour des architectures à base de processeurs IA-64 et x86. La partie dépendante de la plate-forme est extrêmement limitée, de l'ordre de quelques lignes de code. Par conséquent, le portage de cette implémentation à un autre type d'architecture déjà supportée par Linux devrait être relativement facile. Cette implémentation fonctionne sur des ordinateurs SMP mais également avec des processeurs multi-thread — ce qui permet aux ordinateurs avec uniquement un processeur, ayant une technologie multi-thread, de bénéficier de l'approche ARTiS pour garantir le temps-réel.

Après avoir détaillé le fonctionnement de la migration automatique et du nouvel équilibrage de charge dans le noyau, nous présenterons l'interface de configuration du système mise à disposition du développeur.

9.1 Migration dans ARTiS

La modification majeure, et la plus délicate techniquement, sur le noyau Linux concerne la migration automatique d'une tâche non RT0 sur le point d'entrer dans une section de code non-préemptible sur un processeur RT : une des bases du fonctionnement du modèle

ARTiS. Cette modification a été développée principalement par un autre membre de l'équipe, l'ingénieur de recherche Julien Soula. Par ailleurs, pour bénéficier pleinement de chaque processeur, les tâches doivent être déplacées d'un processeur à un autre en fonction de la charge de calcul. L'algorithme usuel d'équilibrage de charge inclus dans Linux a été étendu afin de traiter les aspects temps-réel d'ARTiS.

La migration d'ARTiS fait référence au mécanisme qui peut automatiquement migrer une tâche d'un processeur RT vers un processeur NRT lorsque cette tâche est sur le point d'entrer dans une section de code non-préemptible. Tel quel, le mécanisme requiert d'une part que le point d'entrée dans une section non-préemptible soit identifié, et d'autre part de déplacer la tâche du processeur RT vers le processeur NRT. Cette seconde partie se base sur une implémentation spécifique garantissant qu'un processeur RT n'attende jamais un verrou partagé avec un processeur NRT.

9.1.1 Déclenchement de la migration

Lorsqu'une tâche tente d'entrer dans une section critique, le mécanisme de migration d'ARTiS n'est pas systématique. Plusieurs conditions doivent être satisfaites avant d'autoriser la migration :

- la tâche est actuellement sur un processeur RT, puisque bien sûr sur les processeurs NRT l'entrée en section critique est toujours autorisée.
- la tâche a une priorité inférieure à RT0 : les tâches RT0 sont libres d'utiliser des appels systèmes utilisant des sections critiques (bien que cela soit découragé à cause des latences que cela pourrait introduire).
- la tâche n'est pas la tâche spéciale d'état oisif du processeur, puisque chaque processeur possède une telle tâche et qu'en aucun cas elle ne doit s'exécuter ailleurs.
- ce n'est pas un gestionnaire d'interruption qui tente d'entrer dans une section critique. Dans Linux les gestionnaires d'interruption sont exécutés dans le contexte de la tâche courante mais migrer cette tâche ne préviendrait pas la section critique. Nous verrons par la suite comment éviter l'exécution des gestionnaires d'interruption non primordiaux sur les processeurs RT.

Notons que dans l'implémentation, l'évaluation de ces conditions a été optimisée, de manière à ce que le surcoût engendré soit minimal (elle correspond au pire cas à la comparaison de quatre variables d'entiers).

La migration doit être déclenchée dès qu'une tâche requiert d'entrer dans un état où il ne sera plus possible de l'interrompre, permettant ainsi aux tâches RT0 de toujours être ordonnancées à temps. Cela ne peut arriver que dans l'espace noyau, à l'intérieur d'un appel système. Il aurait été possible de migrer la tâche à chaque appel système, autorisant son exécution sur un processeur RT exclusivement dans les parties de calcul dans l'espace utilisateur. Néanmoins tous les appels systèmes n'utilisent pas de section critique, nous avons donc préférer effectuer la détection à une granularité plus fine en identifiant les deux chemins menant à la désactivation de la préemption :

- la préemption est explicitement désactivée (les interruptions sont encore prises en compte mais le ré-ordonnancement de la tâche RT0 est impossible) cela correspond à un appel à la fonction `preempt_disable()`.
- les interruptions sont désactivées (plus aucune interruption n'est prise en compte), cela correspond à un appel à la fonction `local_irq_disable()`.

Une tâche qui appelle une de ces deux fonctions doit migrer vers un processeur NRT. Ces deux fonctions ont été modifiées pour inclure un appel à la fonction `artis_try_to_migrate()`. Cette fonction vérifie les conditions de migrations et, si la migration est autorisée, elle déclenche le mécanisme de migration en appelant `artis_request_for_migration()`.

De plus, il est possible de localement désactiver la migration à l'intérieur d'une section du noyau. C'est ainsi, par exemple, que nous avons protégé la fonction `schedule()` (l'ordonnanceur). Pour cela ARTiS met à disposition deux fonctions `artis_migration_disable()` et `artis_migration_enable()`. Elles activent et désactivent une variable globale spéciale qui est vérifiée en plus des conditions déjà présentées de la migration automatique.

9.1.2 Déroulement de la migration d'une tâche

Les verrous sont des mécanismes légers et faciles à utiliser lorsque plusieurs threads peuvent avoir des accès concurrents à une donnée. Malheureusement ce mécanisme ne présente aucun moyen de gérer la priorité ni la préemption. Par conséquent les verrous inter-processeurs mettent en danger les garanties de latence parce qu'un processeur NRT pourrait bloquer un processeur RT en attente sur un verrou. Ainsi, dans ARTiS, contrairement au mécanisme de migration original du noyau, un processeur ne doit pas prendre de verrou sur la file d'exécution des tâches du processeur destinataire de la tâche.

Notre implémentation profite du fait que l'ordonnanceur prend un verrou sur la file d'exécution propre au processeur pour accéder aux files d'exécution des processeurs émetteur et récepteur de la tâche sans prendre de verrous inter-processeurs. Les actions de suppression et d'ajout de la tâche sont exécutées par des processeurs différents, le lien entre eux étant obtenu à l'aide d'une FIFO intermédiaire spécifique à ARTiS, nommée RT-FIFO. Dans ARTiS, chaque processeur est connecté à chaque autre processeur par le biais d'une RT-FIFO différente (même si pour le mécanisme de migration présenté ici, seulement les chemins depuis des processeurs RT vers des processeurs NRT sont utiles).

Le déclenchement de la migration est effectué depuis le contexte d'exécution de la tâche à migrer par l'appel à `artis_request_for_migration()`. Cependant, en raison de la structure de donnée de l'ordonnanceur, une tâche ne peut pas se changer de file d'exécution elle-même : cela impliquerait que le même contexte de tâche soit utilisé en même temps par deux processeurs. Par conséquent la migration requiert l'aide d'une tâche tierce (c'est d'ailleurs ce qui est fait dans le noyau Linux original lorsqu'une tâche demande explicitement sa migration vers un autre processeur : l'appel système délègue le changement de processeur au thread noyau `kmigration`). Dans ARTiS, cette charge est assignée à la prochaine tâche à s'exécuter sur le processeur émetteur.

Au total, le processus de migration implique l'interaction de trois tâches : la tâche migrée, la prochaine tâche à s'exécuter sur le processeur émetteur et la tâche en cours d'exécution sur le processeur destinataire. Chacune de ces tâches est en charge d'une ou plusieurs étapes du processus de migration :

- **l'étape de requête** est effectuée par la tâche elle-même via l'appel à `artis_request_for_migration()`. Quand le besoin de migration est détecté, la tâche active un drapeau spécial pour indiquer l'entrée dans le processus de migration. En plus, pour éviter qu'au même moment la tâche soit migrée pour une autre raison (par exemple pour l'équilibrage de charge), l'affinité aux processeurs est figée au processeur local uniquement. Puis elle appelle l'ordonnanceur. ARTiS garantit que cette tâche quittera le

processeur et que son prochain ordonnancement se fera sur le processeur destinataire.
– **l'étape d'envoi** est effectuée par la tâche suivante ordonnancée. Après le changement de contexte, l'ordonnanceur détecte que la tâche « précédente » requiert une migration par la présence du drapeau de migration. Il fait alors appel à la fonction `artis_complete_migration()`. Cette fonction choisit le processeur NRT destinataire (le moins chargé), puis ajoute la tâche à la RT-FIFO spécifique au couple de processeurs et force un appel à l'ordonnanceur du processeur destinataire par l'envoi d'une interruption inter-processeur spéciale.
– **l'étape de réception** est effectuée sur le processeur destinataire. À chaque appel de l'ordonnanceur la fonction `artis_fetch_migration()` est utilisée pour vérifier l'état des RT-FIFO pointant vers le processeur local. Si l'une de ces FIFO n'est pas vide alors chacune des tâches présentes est retirée de la file puis ajoutée à la file d'exécution locale.
– **l'étape de réveil** est effectuée par la tâche migrée. Une fois réordonnancée (et donc au retour de l'appel à l'ordonnanceur), la tâche réinitialise le drapeau de migration et continue son exécution normale, qui commence par désactiver la préemption.

9.1.3 RT-FIFO : des FIFO sans verrou

La structure de donnée d'une RT-FIFO introduite dans ARTiS est caractérisée par le fait que les accès doivent être faits sans verrou : un processeur RT ne doit pas partager de verrou avec un processeur NRT.

Valois [98] a proposé un algorithme qui garantie que ni l'ajout ni la suppression dans une FIFO ne sont bloquantes. Cet algorithme est sans verrou, et lorsqu'en plus il n'y a qu'un seul producteur et un seul consommateur, il est également sans attente. Il se base sur l'usage d'une liste chaînée, à un bout s'effectuent les suppressions tandis qu'à l'autre bout s'effectuent les ajouts. La caractéristique principale de l'algorithme de Valois est que la liste n'est jamais vide :
– à l'initialisation, un nœud factice est introduit dans la structure ;
– à chaque fois qu'un nœud est retiré de la file, il reste dans la structure pour remplacer le nœud factice précédent.

L'algorithme utilise des nœuds qui contiennent la structure nécessaire pour le chaînage et une référence vers la donnée transmise (dans notre cas cette donnée est la structure du noyau associée à la tâche migrée). Dans l'implémentation proposée par Valois les nœuds sont alloués et libérés de la mémoire dynamiquement. Dans le contexte temps-réel, une allocation dynamique n'est pas possible car elle nécessite l'usage de verrous inter-processeurs ! La solution mise en place consiste à créer un nœud factice à la création de chaque tâche, en plus des nœuds factices créés pour chaque liste chaînée. Chaque tâche a ainsi un nœud de disponible pour sa migration (ce qui est suffisant car une tâche ne peut être migré que vers un seul processeur à la fois). Lorsqu'une tâche est retirée d'une RT-FIFO, son nœud associé reste dans la liste chaînée et la tâche acquiert comme nœud le précédent nœud factice.

9.2 Équilibrage de charge

Un mécanisme d'équilibrage de charge a pour but d'optimiser l'exploitation des processeurs simplement en déplaçant les tâches d'un processeur à un autre. Le but de ce type de mécanisme peut aussi être décrit comme étant la minimisation du temps d'exécution d'un

ensemble de tâches. Habituellement, cela est équivalent à maintenir la même charge de calcul sur chaque processeur.

Les caractéristiques d'un équilibrage de charge sont expliquées en détail par Cyril Fonlupt dans sa thèse [45]. On peut les résumer ainsi :

– politique de **mise à jour des informations** : détermine les mécanismes pour collecter et renouveler les statistiques du système. On peut distinguer les politiques qui sont à l'initiative du récepteur — les processeurs les moins chargés initialisent l'équilibrage de charge et prennent des tâches depuis les autres processeurs — des politiques à l'initiative de l'émetteur — les processeurs surchargés initialisent l'équilibrage de charge afin de se décharger d'un certain nombre de leurs tâches vers les autres processeurs — et des politiques mixtes — tous les processeurs sont susceptibles de détecter un déséquilibre ;
– politique de **déclenchement** : détermine le moment où la redistribution des tâches est effectuée ;
– politique de **sélection** : choisit les nœuds à ré-équilibrer ;
– politique de **désignation locale** : choisit les tâches qui seront déplacées.
– politique d'**appariement** : sélectionne le nœud destinataire d'une tâche à déplacer.

9.2.1 Contraintes spécifiques à ARTiS

Dans son état actuel, l'équilibrage de charge présent dans le noyau Linux fonctionne correctement, surtout dans les conditions d'usage de calcul intensif. Cependant avec l'introduction des contraintes d'ARTiS, son comportement n'est plus optimal. En particulier, l'asymétrie introduite entre les processeurs nécessite un équilibrage qui puisse tenir compte des affinités spécifiques entre processeurs et tâches. On peut noter quatre nouvelles contraintes principales.

Premièrement, un changement obligatoire par rapport au mécanisme d'équilibrage de charge original est la suppression des verrous inter-processeurs. Pour la même raison que dans le cas de la migration automatique, les verrous sont proscrits afin d'assurer les propriétés temps-réel des processeurs RT.

Deuxièmement, la migration automatique d'ARTiS peut impliquer l'exécution des tâches RT1+ sur des processeurs NRT. Cependant, les processeurs RT assurent des latences plus faibles, par conséquent dans le cadre d'une politique du meilleur effort les tâches RT1+ doivent être migrées vers un processeur RT dès que cela est possible (c'est-à-dire une fois la préemption de nouveau activée). C'est l'équilibrage de charge qui aura le devoir de migrer les tâches RT1+ des CPU NRT vers les CPU RT. Similairement, si un processeur RT est trop chargé, ce sont les tâches standard, celles qui tirent le moins de bénéfice à s'exécuter sur un tel processeur, qui doivent être privilégiées pour augmenter la charge d'un processeur NRT.

Troisièmement, le nombre de migrations automatiques doit être minimisé. Même si le mécanisme de migration est relativement peu coûteux en terme de performance grâce en particulier au fait qu'aucun verrou n'est utilisé, il tend à retarder l'exécution de la tâche migrée. Par conséquent, il est judicieux de réduire le nombre de situations où une tâche doit être migrée en privilégiant sur les processeurs RT la présence de tâches qui ne désactivent pas la préemption.

Quatrièmement, la mesure de la charge processeur doit pouvoir tenir compte du déséquilibre entre processeurs de la charge dédiée aux tâches temps-réel. Alors que l'équilibrage de charge original approxime la charge processeur par le nombre de tâches, cette mesure n'est pas réaliste si une grande partie du temps d'exécution d'un processeur est dédiée aux

tâches temps-réel. En effet, ces tâches ne partagent pas le processeur équitablement comme le feraient des tâches standard.

9.2.2 Pondération de la longueur de la file d'exécution

La politique d'association de Linux sélectionne le processeur qui va recevoir les tâches en choisissant le moins chargé. La charge est estimée d'après le nombre de tâches en attente d'exécution (la longueur de la file). Cette estimation est tout à fait raisonnable tant qu'il n'y a que des tâches standard qui s'exécutent car elles partagent le temps CPU équitablement et donc plus la file d'exécution est grande, moins il y a de temps accordé à chaque tâche.

Mais avant d'aller plus loin, détaillons ce que nous entendons par *équité*. Avec un ordonnancement équitable, sur une période donnée, les tâches ayant les mêmes priorités doivent se voir attribuer le même temps d'exécution. En corollaire, le temps accordé doit être fonction croissante de la priorité. Sur un processeur donné c'est uniquement l'ordonnanceur qui se charge d'assurer l'équité entre les tâches. Cependant il n'a pas les moyens d'ajuster le temps octroyé à chaque tâche en fonction de ce qui se passe sur les autres processeurs. C'est à l'équilibrage de charge d'assurer l'équité inter-processeur.

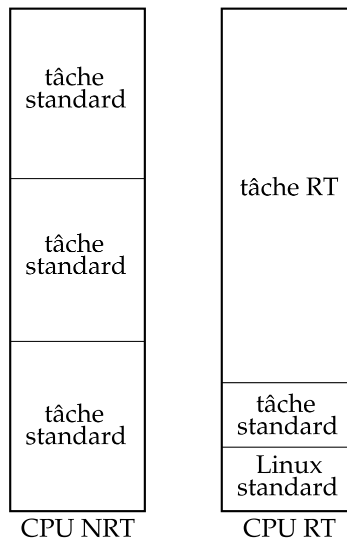


FIG. 9.1: Partage du temps processeur avec le mécanisme d'estimation de charge original de Linux. Le partage n'est pas équitable car la consommation de la tâche temps-réel est mal estimée.

Cette hypothèse n'est plus valide lorsqu'il y a un grand nombre de tâches temps-réel sur un processeur. Étant donné que ces tâches ont une priorité absolue sur les autres tâches, leur ordonnancement est non-préemptif, donc le temps CPU n'est pas partagé : la longueur de la file d'exécution n'est alors plus représentative de la puissance de calcul disponible sur un processeur. Par exemple, considérons un système bi-processeur. Si une tâche temps-réel consomme $3/4$ du temps processeur et qu'il y a 5 tâches standard également en cours d'exécution, alors l'implémentation actuelle dans Linux placera 3 tâches sur chaque processeur. Dans ce cas, certaines tâches standard disposeront de $1/3$ du temps processeur, tandis que les

autres (pourtant ayant la même priorité) ne disposeront que de $1/8$ du temps d'un processeur, comme illustré par la figure 9.1.

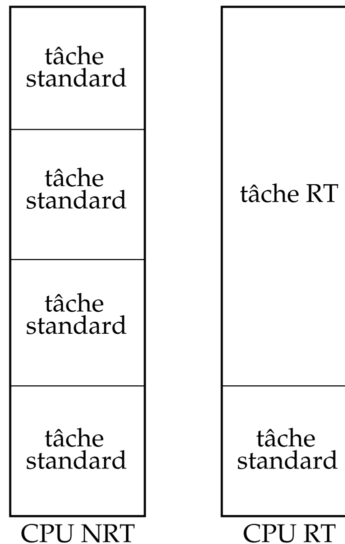


FIG. 9.2: Partage du temps processeur avec le mécanisme d'estimation de charge avec pondération de la longueur de la file d'exécution. Toutes les tâches standard disposent de la même quantité de temps CPU.

La solution que nous proposons est de mesurer la charge d'un processeur à l'aide de la formule $L \times \frac{1}{1-RT}$, où L est la longueur de la file d'exécution *sans* les tâches temps-réel et RT est la proportion de temps CPU consommé par les tâches temps-réel. Dans le scénario présenté, avec cette formule, les tâches standard retrouvent une équité vis-à-vis du temps CPU disponible, comme le montre la figure 9.2. Notons que ce type de scénario, où les processeurs n'exécutent pas tous autant de tâches temps-réel est extrêmement probable dans un système utilisant ARTiS en raison de l'asymétrie de distribution des tâches temps-réel.

Avec la formule d'estimation de la charge proposée, on pourra remarquer que tant qu'un processeur n'exécute aucune tâche standard, sa charge est considérée comme nulle, même si les tâches temps-réel consomment une grande partie de la puissance disponible. Dans le cas d'un soudain grand déséquilibre, cela peut mener à un équilibrage correct uniquement après que des tâches aient été déplacées vers un CPU RT puis déplacées de nouveau vers un CPU NRT. Néanmoins cette propriété offre le bénéfice d'éviter au CPU RT de perde du temps en restant oisif, ce qui serait encore plus gênant d'un point de vue de la performance globale du système.

Ainsi, l'implémentation requiert l'ajout de statistiques concernant le nombre de tâches RT exécutées sur chaque processeur, et aussi la mesure de la proportion RT. À un instant précis cette proportion est soit 1 (une tâche RT est en cours d'exécution) soit 0 (le processeur exécute une tâche standard ou est oisif). Pour obtenir une valeur plus proche de la valeur intuitivement recherchée, il est nécessaire de lisser la valeur sur une période donnée. Nous proposons d'utiliser un mécanisme similaire à celui déjà utilisé dans Linux pour la mesure de charge, nommé `CALC_LOAD()`. Avec ce mécanisme, les valeurs les plus récentes sont plus fortement pondérées. Les 500 dernières mesures sont prises en compte, ce qui correspond sur

les architectures que nous visons à 0,5 seconde.

9.2.3 Suppression des verrous inter-processeurs

Une des contraintes directes d'ARTiS est la nécessité d'éviter les verrous qui pourraient être partagés entre processeurs RT et NRT. L'implémentation originale ne requiert pas de verrous lors de la lecture de la charge des autres processeurs mais lors de la migration de tâches (d'un processeur chargé au processeur courant) un verrou inter-processeur est utilisé pour modifier atomiquement les deux files d'exécution.

L'usage des RT-FIFO permet de résoudre ce problème mais implique un certain nombre d'adaptations à effectuer sur le mécanisme d'équilibrage de charge. La version originale est basée sur l'initiative du récepteur, mais l'usage de FIFO se prête beaucoup plus directement à l'initiative de l'émetteur : le processeur choisit une tâche à transmettre, l'insère dans la FIFO, et plus tard le processeur destinataire la récupérera de manière asynchrone. En échangeant des messages à travers une RT-FIFO, il serait possible de garder l'initiative du récepteur, mais au prix d'une plus grande complexité et surtout de l'introduction de longs délais dans la migration.

Pour inverser la politique de mise à jour des informations la modification principale effectuée a porté sur la fonction `find_busiest_queue()` qui ne doit plus rechercher le processeur le plus chargé, mais le moins chargé. Nous avons implémenté une nouvelle fonction nommée `find_idlest_queue()`. Bien sûr, toutes les sous-fonctions ont dû être changées de manière similaire. Cela implique aussi qu'il n'y a plus d'intérêt à exécuter l'équilibrage de charge lorsque le processeur devient oisif, il est simplement exécuté à intervalles réguliers.

9.2.4 Estimation de la prochaine tentative de migration

Un mécanisme particulier a été introduit afin de permettre le retour des tâches RT1+ d'un CPU NRT vers un CPU RT de manière efficace. Typiquement, une application RT1+ peut appeler consécutivement plusieurs fonctions mettant en danger les propriétés temps-réel. Chacun de ces appels sera effectué sur un processeur NRT. Si l'équilibrage de charge migrait la tâche dès qu'un appel était terminé, il se produirait un effet *ping-pong* entre les deux processeurs, comme l'illustre la figure 9.3. Non seulement cela ralentirait l'exécution mais en plus la charge ne serait pas équilibrée.

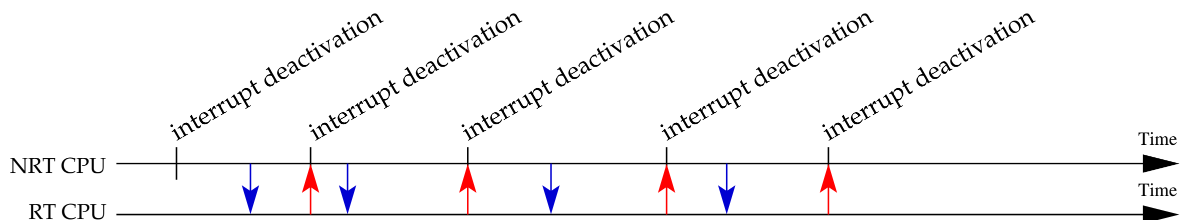


FIG. 9.3: Le problème dit de *ping-pong*. Une tâche s'exécutant sur un processeur NRT est migrée par l'équilibrage de charge vers un processeur RT moins chargé. En raison de la fréquente désactivation des interruptions, après chaque équilibrage, la tâche retourne sur le processeur NRT.

Pour répondre à ce problème, nous proposons une modification de la politique de sélection de tâches de manière à favoriser les tâches qui sont plus à même de rester longtemps sur le processeur RT. Par la simple observation des appels mettant en danger les propriétés temps-réel (c'est-à-dire une tentative de migration) par la tâche, il est possible d'estimer le moment de la prochaine tentative de migration. L'équilibrage de charge peu éviter de migrer les tâches dont le risque de nouvelle migration est élevé. Ce mécanisme est présenté dans la figure 9.4. Il existe une période (rectangle hachuré) située autour du temps estimé de prochaine migration pendant laquelle la migration est interdite.

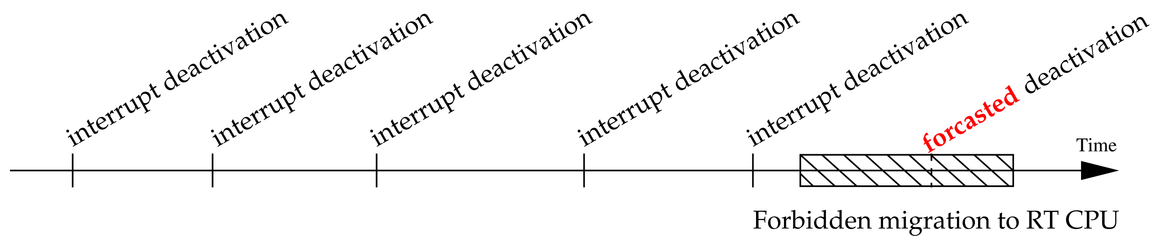


FIG. 9.4: Période de migration interdite (rectangle hachuré). Cette période est déduite de l'observation du comportement précédent de la tâche.

Lors de l'équilibrage de charge, au temps t, il existe deux possibilités :

- La migration suivante estimée est après t, si la migration va probablement arriver bientôt alors cette tâche ne doit pas être sélectionnée pour l'équilibrage. Au contraire, s'il est estimé que la migration a peu de chance d'arriver avant une durée suffisamment longue par rapport au temps de migration, la tâche peut être migrée vers un processeur RT.
- La migration suivante estimée est avant t, si la migration aurait dû se produire récemment alors elle est considérée comme ayant encore une forte chance d'arriver très prochainement et par conséquent la tâche n'est pas sélectionnée pour l'équilibrage. Si la migration avait été prévue considérablement avant, alors la tâche peut être migrée vers un processeur RT. La durée utilisée pour le temps est relative à la période mesurée de la tâche.

Ces deux conditions peuvent être représentées mathématiquement par les deux formules suivantes :

$$\begin{cases} t > (P_m + t_d) - C \\ t < (P_m + t_d) + K \times P_m \end{cases}$$

Où P_m représente la période moyenne entre deux blocages, t_d l'instant du dernier blocage et t l'instant courant. C est un temps de l'ordre de plusieurs tics d'ordonnancement, il correspond à la durée minimale pendant laquelle la tâche est assurée de rester sur le processeur NRT. Dans l'implémentation nous l'avons fixé à 2 tics, ce qui correspond approximativement au temps perdu par la tâche lors d'une double migration. K est un coefficient qui pourrait être situé typiquement entre 2 et 100, il dépend de la confidence portée à l'estimation de prochaine migration : plus l'erreur d'estimation peut être grande et plus le coefficient doit être grand. Nous l'avons fixé à 2 dans l'implémentation, estimant que les tâches temps-réel sont souvent

répétitives et privilégiant ainsi la politique du meilleur effort pour les latences d'interruption (qui nécessite de migrer aussi vite que possible les tâches vers un CPU RT).

Bien sûr l'implémentation de ce mécanisme de prédiction consiste en une, légère, modification de l'équilibrage de charge (la fonction `load_balance()`) mais elle consiste aussi en l'obtention des statistiques concernant les tentatives de migration. Ces statistiques sont stockées sous la forme de deux entiers pour chaque tâche. L'un des nombres correspond à la moyenne entre deux tentatives de migration pondérée sur les 100 dernières tentatives, l'autre nombre correspond à l'instant de la dernière tentative de migration. Chaque appel à `artis_try_to_migrate()` qui mènerait à la migration forcée de la tâche si elle était sur un processeur RT correspond à une *tentative de migration*.

9.2.5 Association du type de tâche au type de processeur

La politique de désignation locale (le mécanisme sélectionnant la tâche qui doit être déplacée) et la politique d'association (le mécanisme sélectionnant la destination de la tâche) ont été modifiées afin de refléter l'asymétrie d'ARTiS. Basée sur la fonction originale `load_balance()` et ses sous-fonctions, la fonction `load_balance_push()` a été mise en place pour permettre un équilibrage de charge à l'initiative de l'émetteur. Selon le type des processeurs émetteur et destinataire, cette fonction ne déplace que les tâches RT1+ ou toutes les tâches.

Les équilibrages de charge symétriques (NRT vers NRT et RT vers RT) n'ont pas subi de modifications, et donc les politiques restent les mêmes que dans la version de Linux originale.

Concernant l'équilibrage de charge d'un processeur RT vers NRT, la fonction `move_tasks_push()` qui sélectionne les tâches a été modifiée afin de déplacer en priorité les tâches NRT, devant les tâches RT1+ qui bénéficient d'une meilleure latence en restant sur le processeur RT. Bien sûr l'équilibrage de charge d'un processeur NRT vers RT doit se comporter à l'inverse, ce qui est la politique par défaut.

Un autre aspect important concerne la fréquence de l'équilibrage. La fonction `rebalance_tick()` a été étendue afin de pouvoir gérer des fréquences différentes selon les types de processeurs impliqués. En particulier, la migration des tâches RT1+ de NRT vers RT est déclenchée très fréquemment. La fréquence choisie a été déterminée expérimentalement, elle correspond à 4 fois la fréquence des autres déclenchements, de manière à réduire le temps passé par les tâches sur les CPU NRT tout en évitant un surcoût trop important lié à l'équilibrage. On pourrait également noter la suppression du déclenchement au moment où un processeur devient oisif (ne possède plus de tâches à exécuter) car cela n'apporterait pas de bénéfices dans le cadre d'un équilibrage à l'initiative de l'émetteur.

9.3 Déploiement d'applications temps-réel

Un avantage de l'implémentation d'ARTiS dans Linux est qu'elle repose sur l'API déjà existante. Pour bénéficier des apports d'ARTiS, il n'est pas nécessaire de recompiler une application écrite pour Linux. Néanmoins, l'utilisateur doit spécifier le partitionnement du système et la distribution des tâches sur les processeurs. La configuration du système consiste à définir le type de chaque processeur (NRT ou RT) ainsi que l'affinité des interruptions (quels processeurs prennent en charge une interruption donnée). La configuration de l'application consiste à définir la priorité de chaque tâche ainsi que leur affinité aux processeurs. Cette

configuration peut être spécifiée à l'aide d'une extension de l'API du noyau qui permet de manipuler directement ces concepts.

9.3.1 Exemple de déploiement d'application

Une application temps-réel sur un système multiprocesseur ne prend tout son sens que lorsque les contraintes temps-réel sont combinées à du calcul intensif. ARTiS est dédié à ce type d'application. Les contraintes temps-réel sont satisfaites par les tâches RT0. Les communications entre tâches RT0 et RT1+ permettent le calcul intensif avec transfert de données. Les tâches standard se chargent des services supplémentaires. L'équilibrage de charge assure le partitionnement dynamique des tâches sur les processeurs.

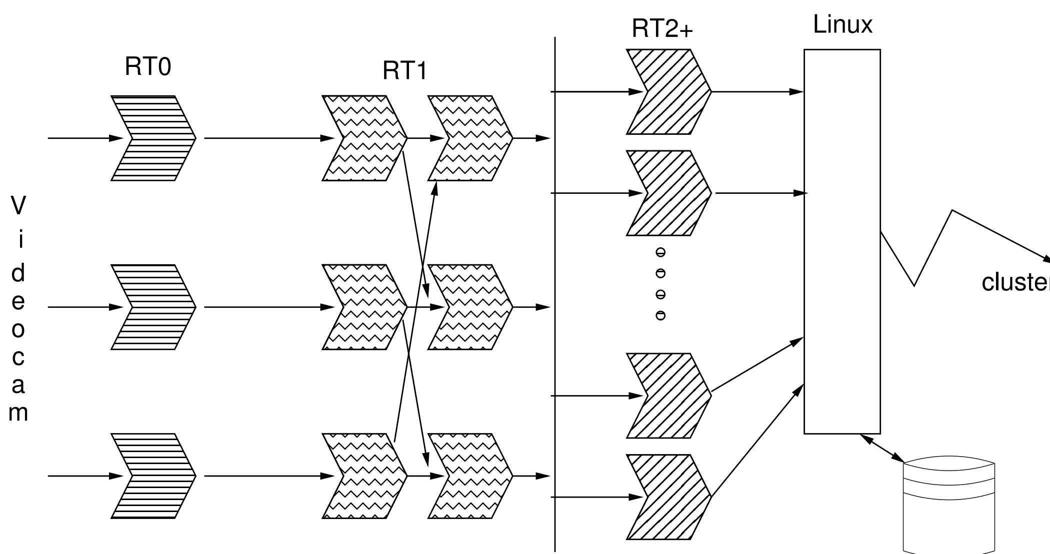


FIG. 9.5: Exemple d'architecture d'une application adaptée pour ARTiS.

L'implémentation d'une application sur ARTiS requiert l'identification de la priorité temps-réel de chacune des tâches de l'algorithme ainsi que le placement de ces tâches sur les processeurs du système SMP. Pour illustrer cela, prenons comme exemple un processus d'assurance qualité automatisé : la détection de défauts sur une chaîne de production s'exécutant sur un système SMP quadri-processeur. Trois des processeurs sont assignés à l'ensemble RT tandis que le dernier est assigné à l'ensemble NRT. Plusieurs tâches peuvent être identifiées, leurs communications et leurs placements sont synthétisés dans la figure 9.5 :

– Une caméra vidéo et/ou des capteurs reçoivent périodiquement des données. Jusqu'à trois tâches RT0 peuvent traiter en même temps les différentes acquisitions de donnée avec une latence compatible avec les contraintes temps-réel. Chacune de ces tâches est assignée à un processeur RT.
– Directement connecté à ces tâches, le traitement intensif des données, basé sur des structures de données régulières, peut être utilisé pour le traitement des images. Un nombre fixe de tâches RT1 est dédié à ce calcul de données parallèle (à la OpenMP). Elles doivent communiquer avec les tâches RT0 et aussi entre elles sans subir de migration superflue. Elles sont donc pratiquement liées à un processeur RT, mais peuvent

éventuellement être migrées de temps à autre vers le processeur NRT au cas où elles utiliseraient un appel système désactivant la préemption. Ce sont elles qui utilisent la majorité des ressources de calcul du système.
– L'identification des défauts peut être effectuée en utilisant des structures de données irrégulières : chaque traitement de défaut est géré spécialement. Un nombre dynamique de tâches RT2 est créé. Elles peuvent communiquer avec les tâches RT1 ainsi qu'entre-elles. Étant donné que le nombre de tâches est dynamique, l'équilibrage de charge est indispensable.
– Enfin, les défauts peuvent être comparés à une base de données locale ou distante afin de générer des statistiques... Cette étape ne requiert pas de traitement temps-réel, par conséquent peut être effectué par des tâches Linux standard. Elles sont principalement placées sur les processeurs NRT, mais peuvent également utiliser des processeurs RT s'ils sont oisifs.

La figure 9.6 présente un placement possible de ces tâches sur les deux ensembles de processeurs : RT et NRT.

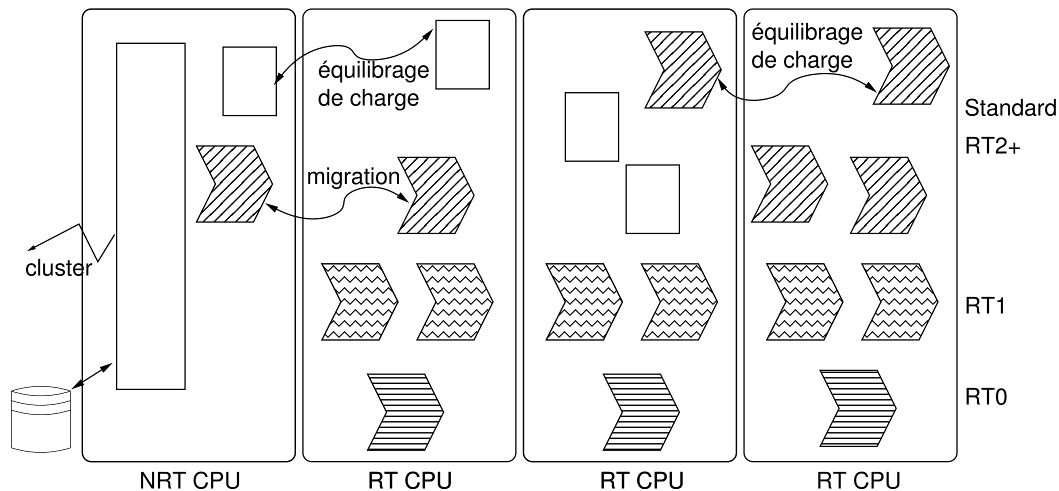


FIG. 9.6: Placement des tâches sur les processeurs RT et NRT.

9.3.2 Configuration du système

ARTiS est mis à disposition sous la forme d'un patch pour le noyau Linux. Une fois ce patch appliqué, la compilation du noyau et un redémarrage de la machine sont suffisants pour mettre en place un système ARTiS. Selon l'architecture matérielle exacte, il peut être cependant nécessaire également de désactiver certaines options d'économie d'énergie qui ont tendance à augmenter les latences d'interruption lorsque le système est oisif. Aucune installation de paquetage ni de recompilation d'application n'est nécessaire.

Une fois le système en marche, une configuration est requise pour indiquer au noyau le partitionnement de la machine. L'utilisateur peut spécifier quels sont les CPU RT et quels sont ceux NRT à l'aide d'une interface `/proc` (écriture dans un fichier virtuel). Afin de maintenir la cohérence du partitionnement, les interruptions doivent être redirigées selon leurs usages. De manière simplifiée, la règle à suivre est que toutes les interruptions doivent être prises

en charge par un processeur NRT, à l'exception de celles attendues par les tâches RT0 qui doivent alors être exclusivement transmises au processeur sur lequel la tâche s'exécute. Ces modifications peuvent être aisément automatisées par un script au démarrage. Par ailleurs, l'interface /proc permet d'activer et de désactiver dynamiquement ARTiS ainsi que d'afficher quelques statistiques concernant la migration de tâches.

9.3.3 Identification des tâches temps-réel

Les tâches RT0 sont identifiées au sein du noyau par le fait que ce sont des tâches Linux ordonnancées avec la politique d'ordonnancement FIFO (SCHED_FIFO) avec la priorité maximale. Pour configurer une tâche ainsi, les fonctions POSIX sched_setscheduler(), sched_setparam(), et sched_get_priority_max() sont utilisées. De plus, une tâche RT0 doit également être liée à un processeur RT. Cela est effectué à l'aide de la fonction Linux sched_setaffinity(). L'ensemble des processeurs sur lesquels la tâche est autorisée ne doit comprendre qu'un seul processeur, de type RT.

En raison de la nature d'ARTiS, la seule contrainte supplémentaire par rapport à l'usage de ces fonctions est l'ordre des appels à sched_setscheduler() et sched_setaffinity() : la priorité doit toujours être modifiée avant le changement d'affinité. Si les appels à ces fonctions sont inversés, la tâche ne sera pas RT0 parce que dans l'implémentation actuelle de Linux le changement de priorité d'une tâche implique la désactivation de la préemption. Cette désactivation force la migration de la tâche vers un processeur NRT. Comme l'affinité qui vient juste d'être spécifiée ne contient qu'un processeur RT, ARTiS la modifie automatiquement pour autoriser l'exécution de la tâche également sur le processeur NRT ! Notons que si un utilisateur ne souhaite pas, ou ne peut pas, recompiler son application pour l'adapter à cette nécessité, il est également possible de donner la priorité RT0 à une tâche depuis une autre tâche. Par exemple les outils Linux chrt et taskset permettent cela.

```
unsigned int rt_cpu;
struct sched_param schedp;

/* lock the address space of the process */
if (mlockall(MCL_CURRENT|MCL_FUTURE) != 0)
        perror(...);

/* set the scheduling policy */
memset(&schedp, 0, sizeof(struct sched_param));
schedp.sched_priority = sched_get_priority_max(SCHED_FIFO);

if (sched_setscheduler(0, SCHED_FIFO, &schedp) != 0)
        perror(...);

/* bound the process to the rt_cpu CPU */
if (sched_setaffinity(0, sizeof(unsigned long), 0x1UL << rt_cpu ) == -1)
        perror(...);
```

FIG. 9.7: Identification d'une tâche RT0

La figure 9.7 présente le code typique d'identification d'une tâche pour devenir RT0. Afin de simplifier cette partie d'initialisation au développeur, ARTiS fournit une bibliothèque ba-

sique pour manipuler ces propriétés explicitement. Ainsi, il existe entre autres deux fonctions pour activer et désactiver la priorité RT0 d'une tâche :

```
int artis_enter_rt0 (pid_t pid, int rt_cpu);
int artis_leave_rt0 (pid_t pid);
```

Les tâches RT1+ sont toutes des tâches associées à une politique d'ordonnancement FIFO ou *round-robin* (SCHED_FIFO ou SCHED_RR). La priorité de ces tâches est définie par la priorité POSIX. La bibliothèque ARTiS met à disposition des fonctions pour identifier ces tâches :

```
int artis_enter_rt1plus(pid_t pid, int policy, int priority);
int artis_leave_rt1plus(pid_t pid);
```

De nouveau, si le code application ne doit pas être modifié, une priorité RT1+ peut être assignée à une tâche par un outil indépendant.

Les tâches standard, c'est-à-dire les tâches qui ne sont pas temps-réel, sont toutes ordonnancées selon la politique par défaut de Linux (SCHED_OTHER). L'affinité d'une tâche qui n'a pas une priorité RT0 doit inclure un processeur NRT, sans quoi ARTiS y ajoutera automatiquement un processeur NRT lors de l'entrée dans une section de code non-préemptible. De plus, l'affinité processeur d'une tâche RT1+ devrait toujours contenir au moins un processeur RT.

9.4 Synthèse de l'implémentation

L'implémentation d'ARTiS est disponible, elle est basée sur le noyau Linux 2.6 et écrite pour les architectures IA-64 et x86. Un mécanisme de migration automatique a été ajouté : cela a consisté principalement à ajouter la détection d'entrée et de sortie des sections critiques et à mettre en place l'usage d'une FIFO sans verrou pour le transfert des tâches entre chaque processeur. Nous avons également adapté l'équilibrage de charge de manière à prendre en compte l'asymétrie. Nous avons proposé plusieurs améliorations dont en particulier une mesure de la charge prenant mieux en compte les tâches temps-réel, l'usage de mécanismes de migration sans verrous inter-processeurs, un mécanisme pour éviter que la migration automatique ne s'interpose à l'équilibrage, et la mise en place d'une sélection de tâche considérant les affinités entre les types de tâches et les types de processeurs. Par ailleurs, une interface a été développée de manière à permettre à l'utilisateur de dynamiquement modifier les paramètres d'ARTiS. L'API suit autant que possible l'API standard POSIX et le cas échéant elle suit l'interface traditionnelle de Linux. Il n'est pas nécessaire de recompiler une application pour bénéficier des propriétés temps-réel. La configuration du système est effectuée en spécifiant les priorités de chaque tâche et le partitionnement des processeurs et des interruptions.

Dans le chapitre suivant nous allons présenter les procédures suivies pour valider le comportement de cette implémentation.

Chapitre 10

Validation Expérimentale

10.1 Mesure de latence d'interruption **198**
10.1.1 Méthode de mesure 198
10.1.2 Types de latence d'interruption 199
10.1.3 Condition des mesures 199
10.1.4 Latences observées 200
10.2 Variation du temps d'exécution **201**
10.2.1 Méthode de mesure 201
10.2.2 Condition des mesures 201
10.2.3 Temps d'exécution observés 202
10.3 Observation de l'équilibrage de charge **202**
10.3.1 lbμ, un outil de validation d'équilibrage de charge 203
10.3.2 Conditions des mesures 204
10.3.3 Observation des scénarios de test 204
10.4 Synthèse de la validation expérimentale **207**

Au cours de l'implémentation d'ARTiS nous avons effectué trois types d'observation afin de valider l'approche tant sur le plan théorique que pratique [10]. Ces observations, qui permettent de vérifier les avantages mais aussi de noter certains désavantages d'ARTiS, comprennent : les latences d'interruption, la variation de temps d'exécution, et l'efficacité de l'équilibrage de charge.

Avant de présenter plus en détail les mesures, remarquons que sur les architectures matérielles actuelles de nombreuses optimisations de performance favorisent les performances globales au déficit du temps d'exécution de cas spéciaux. Comme l'ont montré McGuire et Zhou [50], cela entraîne des temps d'exécution extrêmement variables, contraires à ce qui est recherché pour le temps-réel dur. En particulier, les mécanismes de cache et de prédiction de branchement peuvent introduire de grandes latences lors de conditions inhabituelles, qui peuvent passer aisément inaperçues lors de mesures expérimentales. Nous avons tenté de tenir compte au mieux de ces difficultés pour obtenir les résultats présentés ici. Entre autres, des charges spécifiquement dédiées à déclencher des chargements de ligne de cache ont été implémentées.

10.1 Mesure de latence d'interruption

Le premier type d'évaluation que nous présentons concerne la latence de réaction aux interruptions, le temps écoulé entre un évènement matériel reporté au processeur et l'exécution par le processeur de la routine spécifique traitant cet évènement. Dans un contexte temps-réel dur, il est important de minimiser les latences les plus longues, afin de pouvoir assurer un temps de réponse *borné* aussi faible que possible. Nous avons distingué deux types de latences, l'une associée au noyau et la seconde associée aux tâches utilisateur.

Même si une analyse théorique du temps d'exécution du pire cas (WCET pour Worst Case Execution Time) serait préférable car elle permettrait de définir avec assurance un temps de réponse maximum, sa complexité est telle qu'il ne nous a pas été possible de l'effectuer. Les difficultés d'une telle analyse ont été mises en évidence par l'étude du WCET de RTEMS (restreint à un seul processeur) par Colin et Puaut [26]. Dans l'implémentation d'ARTiS, en plus des difficultés rencontrées par les auteurs, il faudrait modéliser l'architecture IA-64, simplifier et annoter le code source du noyau Linux et prendre en compte le parallélisme à la fois au niveau matériel et logiciel. À l'heure actuelle probablement personne n'est à même de traiter une tâche si colossale. Quoi qu'il en soit, à cause des simplifications de modèles nécessaires, ce type d'étude a une tendance à largement surestimer la borne maximale par rapport à la réalité, ce qui réduit l'intérêt des résultats.

10.1.1 Méthode de mesure

Les tests consistent à mesurer le temps écoulé entre la génération matérielle d'une interruption et l'exécution du code concernant cette interruption. Le protocole expérimental mis en place a été écrit en tentant de rester aussi près que possible des mécanismes habituels utilisés par les tâches temps-réel. La tâche de mesure configure un composant matériel de manière à ce qu'il génère une interruption à un temps connu précis, puis elle se met en attente de cette interruption. Lorsque l'interruption est reçue par le système, la tâche est réveillée et elle note le temps actuel. Puis le cycle recommence avec la mesure suivante. Cet enchaînement est particulièrement typique des applications temps-réel, qui attendent un évènement matériel, traitent les nouvelles données, envoient l'information et se remettent en attente. Il y a quatre instants différents associés à une interruption donnée, ils correspondent à différents emplacements dans du code exécuté (figure 10.1) :

- t'_0, programmation de l'interruption ;
- t_0, émission de l'interruption (instant défini par la tâche de mesure) ;
- t_1, entrée dans le gestionnaire d'interruptions spécifique à cette interruption ;
- t_2, entrée dans la tâche utilisateur temps-réel.

Les mesures ont été effectuées sur un quadri-processeur Itanium II 1,3GHz. Le noyau Linux utilisé est une version 2.6.11 instrumentée. Le compteur sur lequel s'appuient toutes les mesures est l'`itc` (un registre du processeur qui compte les cycles d'horloge) et les interruptions sont générées avec une précision au cycle près à l'aide du PMU (une unité interne à chaque processeur [73]).

Même lors de fortes charges du système, des états qui entraînent de longues latences sont extrêmement rarement exhibés. Pour cela, un grand nombre de mesures est nécessaire. Chaque test a duré 8 heures, soit environ 300 millions de mesures. Avec des tests d'une telle durée, nous avons pu vérifier la reproductibilité des résultats d'une campagne de mesures à l'autre.

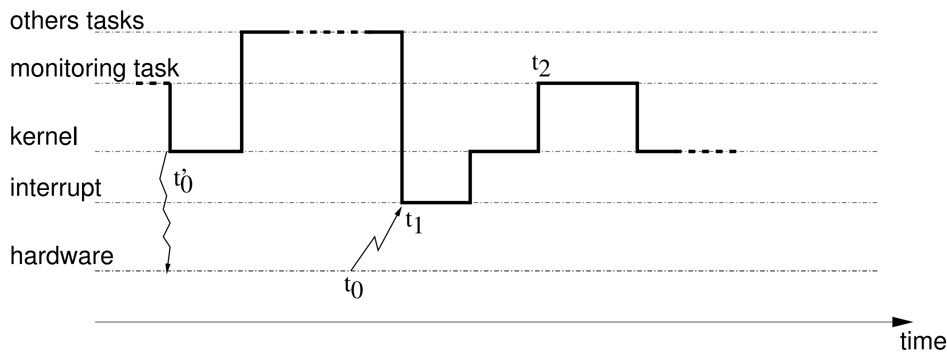


FIG. 10.1: Chronogramme des différentes mesures associées à une interruption.

10.1.2 Types de latence d'interruption

À partir des trois points de mesure, deux valeurs intéressantes peuvent être calculées. Leur intérêt provient du fait qu'il est facilement possible de les associer à des techniques de programmation usuelles. Ces deux types de latence sont :

- **la latence noyau**, $t_1 - t_0$, correspond au temps écoulé entre la génération de l'interruption et l'entrée dans le gestionnaire d'interruptions. Cette latence correspond à celle que subirait une tâche temps-réel écrite en tant que module noyau.
- **la latence utilisateur**, $t_2 - t_0$, correspond au temps écoulé entre la génération de l'interruption et l'exécution du code associé dans la tâche utilisateur. Cela correspond à la latence que subirait une application temps-réel écrite entièrement en espace utilisateur. L'application reçoit les notifications à l'aide d'un appel système bloquant (un `read()`), méthode de vérification qui s'est avérée conduire aux plus faibles latences lors de nos tests préliminaires.

Les tâches temps-réel qui s'exécutent dans l'espace utilisateur ont été développées en utilisant l'interface traditionnelle et standard POSIX. Ceci est un des avantages principaux qu'ARTiS procure. Par conséquent, dans le contexte d'ARTiS, il s'agit de focaliser études et analyse sur ces latences.

10.1.3 Condition des mesures

Les mesures ont été conduites sur quatre configurations de système différentes. Ces configurations ont été sélectionnées pour leur pertinence vis-à-vis des latences. Tout d'abord, le noyau standard Linux a été mesuré avec et sans charge. Puis, un noyau similaire mais avec la préemption activée a été mesuré. Cette option du noyau autorise les tâches à être réordonnancées même si l'exécution courante est dans l'espace noyau, elle est censée améliorer les latences. Enfin, l'implémentation d'ARTiS a été mesurée. Seul le premier noyau est présenté sans charge car les résultats pour les deux autres noyaux se sont avérés extrêmement similaires.

Lors des mesures, la charge système consiste à occuper le système à l'aide d'une utilisation intensive des processeurs et du déclenchement d'un certain nombre d'interruptions dans le but de maximiser la probabilité de verrous inter-processeurs et de désactivations de préemption. Pour cela, cinq programmes correspondants à cinq charges différentes furent

utilisés :

- **Charge de calcul**, une tâche exécutant une boucle infinie sans jamais faire d'appel système est attachée à chacun des processeurs, simulant ainsi une tâche de calcul.
- **Charge d'entrée/sortie**, le programme `iodisk` lit et écrit continuellement sur le disque.
- **Charge réseau**, le programme `ionet` inonde la carte réseau de requêtes à l'aide d'échos/réponses `ICMP`.
- **Charge verrous**, le programme `ioctl` appelle la fonction `ioctl()` qui contient des accès au verrou noyau global (*big kernel lock*).
- **Charge en défauts de cache**, le programme `cachemiss` génère des accès en dehors du cache avec grande fréquence. Cela ajoute des latences, le gestionnaire d'interruption et la tâche RT0 sont exécutés avec un cache « froid », impliquant de très nombreux accès à la mémoire globale.

10.1.4 Latences observées

Le tableau 10.1 résume les mesures des différentes configurations testées. Deux valeurs sont associées à chaque type de latence (noyau et utilisateur). La valeur « maximum » correspond à la latence maximale détectée au cours des 8 heures. La seconde valeur correspond à la latence maximale des 99,999% plus petites latences, dans notre cas les 3000 valeurs les plus élevées ne sont pas comptées. C'est un indicateur couramment utilisé pour caractériser les systèmes temps-réel mous.

		Noyau		Utilisateur	
Configurations		99,999%	Max.	99,999%	Max.
Linux standard	oisif	1µs	6µs	5µs	78µs
Linux standard	chargé	6µs	63µs	731µs	> 49000µs
Linux avec préemption	chargé	4µs	60µs	258µs	1155µs
ARTiS	chargé	8µs	43µs	18µs	104µs

TAB. 10.1: Latences noyau et utilisateur observées pour chaque configuration.

L'étude de la configuration non-chargée donne quelques points de comparaison avec les autres configurations. Notons tout d'abord en comparant avec un noyau standard chargé que la charge a une légère influence sur les latences noyau, elle a surtout une influence énorme sur les latences utilisateur : la latence maximale est supérieure de trois ordres de grandeur ! Ceci met bien en évidence le comportement original de Linux, développé sans porter attention aux contraintes temps-réel. Néanmoins cela ne s'oppose aucunement aux performances du système, concernant la moyenne des latences utilisateur les quatre configurations ont exhibé des valeurs d'environ 4µs.

L'option du noyau pour la préemption n'influe pas sur les latences noyau. Ceci était attendu puisque les modifications apportées ne visent qu'à permettre l'ordonnancement plus rapidement, du côté noyau rien ne permet de répondre aux interruptions plus efficacement. Par contre, on peut noter une amélioration considérable des latences utilisateur : 99,999% des latences sont en dessous de 258µs au lieu de 731µs. Cette amélioration est encore plus nette vis-à-vis de la latence maximale qui a été réduite par un facteur de plus de 40. Cette option

est donc efficace, même s'il reste fortement avantageux en terme de temps de réaction de placer le code temps-réel directement dans le noyau.

Avec la configuration ARTiS, les deux types de latences sont significativement réduits. En particulier, la latence utilisateur maximale observée a été de 104µs, ce qui est du même ordre de grandeur que celle observée dans le noyau. La réduction de latence au niveau noyau s'explique par le fait que le processeur RT ne traite que l'interruption associée au temps-réel, il n'est donc jamais occupé à traiter une autre interruption.

Notons que dans le cadre du projet Hyades pour lequel ARTiS a été implémenté sur Linux, les contraintes temps-réel dur sur la machine de test étaient de 300µs. La configuration ARTiS est nécessaire pour assurer cette contrainte tout en bénéficiant des nombreux avantages de flexibilité, de compatibilité et de qualité qu'offre le développement de tâches temps-réel dans l'espace utilisateur.

10.2 Variation du temps d'exécution

La deuxième évaluation d'ARTiS que nous présentons concerne la stabilité du temps d'exécution. Brièvement, cela repose sur l'exécution multiple d'un même traitement de calcul, puis sur la comparaison des temps passés par chaque exécution. Dans un contexte temps-réel, où il faut pouvoir assurer le temps de réponse à un évènement matériel dans un temps borné, cette mesure est complémentaire à la latence d'interruption. À partir du moment où la tâche temps-réel est exécutée, elle doit être interrompue le moins souvent possible.

10.2.1 Méthode de mesure

Les tests ont consisté à mesurer le temps d'exécution d'une routine effectuant un million de divisions entières, durant environ 10ms. Cette durée a été choisie car elle est approximativement du même ordre de grandeur que le calcul le plus long nécessaire par une tâche temps-réel qui pourrait bénéficier d'ARTiS[1]. Il est intéressant de noter que cela correspond à environ 10 quantums d'ordonnancement. Les mesures furent effectuées un million de fois. Chaque mesure consiste à mesurer le temps d'exécution en comparant le compteur de cycles avant et après l'appel à la routine de calcul.

10.2.2 Condition des mesures

Tous les tests furent exécutés sur le même ordinateur que les mesures de latence, en se basant sur le noyau Linux 2.6.12. Les tests ont été exécutés 8 fois, correspondant à chaque combinaison des trois aspects suivants :

– **La priorité**, la tâche peut être une tâche standard (priorité 0 de l'ordonnancement `SCHED_OTHER`) ou une tâche temps-réel (priorité maximale avec l'ordonnancement `SCHED_FIFO`, équivalent à RT0 dans ARTiS).
– **La charge**, la machine peut être soit dans l'état oisif (sans aucune charge), soit hautement chargée (les mêmes charges que pour les mesures précédentes).
– **L'usage d'ARTiS**, le noyau peut être compilé avec ou sans ARTiS.

[1] Les tâches s'exécutant plus longtemps ont implicitement des contraintes de latence d'interruption également plus longues, suffisamment longues pour s'accommoder des latences maximales exposées par le noyau Linux standard vu dans la section précédente.

10.2.3 Temps d'exécution observés

Le tableau 10.2 résume les mesures des différentes configurations. Nous appelons T_{min} la plus petite durée d'exécution mesurée parmi l'ensemble des configurations. Elle devrait être égale ou très proche du temps d'exécution minimum possible sur ce système. Pour ces mesures nous avons observé T_{min} à 9269µs. Chacun des tests, qui duraient environ 3 heures, est résumé par deux nombres. Le premier est la différence de temps entre T_{min} et le temps maximal enregistré. Le second, indiqué entre parenthèses, représente le pourcentage que le temps supplémentaire de la plus longue exécution par rapport à T_{min}.

Noyau	**Charge**	**Priorité standard**	**priorité RT0**
Linux standard	oisif	0,5926ms (6,39%)	8,30µs (0,08%)
Linux standard	chargé	516,1ms (5567,49%)	20,61µs (0,22%)
ARTiS	oisif	5,675ms (61,22%)	27,07µs (0,29%)
ARTiS	chargé	659,7ms (7116,43%)	21,61µs (0,23%)

TAB. 10.2: Différence des temps d'exécution entre T_{min} et le maximum observé pour la configuration.

Tout d'abord, les résultats montrent que la charge, comme on peut s'y attendre, est un facteur affectant énormément le temps d'exécution, en particulier sur les tâches à priorité standard. Lorsque la tâche est ordonnancée avec une priorité temps-réel, la variation de temps est pratiquement nulle (avec seulement une variation maximale de 0,22% observée sur un noyau standard). Cela provient du fait qu'aucune autre tâche ne peut être ordonnancée tant que la tâche s'exécute. Les seuls ralentissements sont dus à la gestion d'interruption ou à des threads noyau.

Concernant ARTiS, les résultats démontrent qu'il n'influence pratiquement pas les variations du temps d'exécution. Pour les tâches RT0, il est possible de noter néanmoins une légère augmentation de la variation (de 0,07 point, sur le système oisif). En effet, alors que ARTiS permet de largement améliorer les latences d'interruption, le mécanisme ne vise pas d'amélioration pour les variations de temps d'exécution (qui sont de toute façon minimales sur le noyau Linux). La légère augmentation des variations par rapport au noyau standard s'explique par le surcoût introduit dans l'ordonnanceur et l'équilibrage de charge. Nous ne voyons pas de raison au fait que le système oisif ait exhibé la plus forte variation. Quoi qu'il en soit, la variation de temps d'exécution d'une tâche RT0 est très faible, 27,07µs au maximum, comparé au temps de latence de 104µs. Dans le cadre du projet Hyades, la somme de ces deux temps reste largement inférieure à la latence de temps de réponse maximale fixée (de 300µs).

10.3 Observation de l'équilibrage de charge

La dernière évaluation que nous présentons concerne l'équilibrage de charge. À notre connaissance, il n'existe pas de programme disponible dédié à l'estimation de l'équilibrage de charge. Bien sûr un mauvais équilibrage de charge peut mener à de moins bonnes performances, donc une évaluation de performance, type de mesure pour lequel il existe pléthore d'outils, permettrait une analyse préliminaire. La comparaison peut s'effectuer sur le temps (extérieur) passé durant l'exécution de l'ensemble des tâches. Cependant cette approche a

plusieurs limitations. Tout particulièrement, rien n'assure que la performance soit vraiment dépendante de l'ordonnanceur et de l'équilibrage de charge, peut-être est-ce d'autres mécanismes qui ralentissent la tâche. De plus, l'ensemble de tâches testé est forcément très restreint par rapport à l'immense espace de combinaison de tâches que peut être amené à traiter l'équilibrage de charge. Un autre problème relève de la complexité introduite dans le code du programme d'évaluation entraînant une difficulté pour reproduire fidèlement un type de charge. Enfin, lors de l'implémentation de nouvelles politiques d'équilibrage de charge (ou d'ordonnancement), l'affinage de l'algorithme requiert souvent d'observer le comportement avec une configuration précise des tâches (correspondant à la politique modifiée) qu'il peut être très difficile de reproduire uniquement avec un outil de mesure de performance.

Durant l'implémentation d'ARTiS, il a été nécessaire de confirmer qu'une modification de l'équilibrage de charge avait bien les effets escomptés. Un outil particulier a été écrit dans ce but. Après avoir présenté cet outil et son usage, nous détaillerons les différentes configurations de tâches testées puis nous présenterons les résultats obtenus.

10.3.1 lbμ, un outil de validation d'équilibrage de charge

10.3.1.1 Scénarios

lbµ est un outil concentré sur l'exécution d'un ensemble de tâches de manière aussi reproductible que possible. L'ensemble de tâches utilisé pour une mesure est appelé *scénario*. Il est écrit en vue d'observer et d'analyser un comportement spécifique de l'équilibrage de charge. Les tâches du scénario ne sont pas de vraies tâches dans le sens où elles ne font que simuler le comportement de tâches selon certains paramètres. Grâce à cela leur comportement est très similaire d'une exécution à une autre. Le même scénario peut être rejoué autant de fois que nécessaire avec différentes politiques d'équilibrage.

Un scénario est écrit en définissant les propriétés de chaque tâche qui va être exécutée. Les propriétés possibles comprennent le type et la priorité d'ordonnancement, l'affinité processeur, la quantité de calculs (sous forme d'un nombre de boucles à effectuer), la fréquence et la durée d'attentes, et la fréquence d'un appel système désactivant la préemption (ce qui n'est réellement significatif que lorsqu'ARTiS est utilisé). La figure 10.2 présente la définition d'un scénario. Toutes les tâches sont démarrées en même temps, au début des mesures. Bien entendu, un seul scénario n'est pas suffisant pour évaluer un équilibrage de charge dans sa globalité. Pour tester entièrement un équilibrage, il est nécessaire d'utiliser un ensemble de scénarios mettant en jeu chacun des différents aspects qui peuvent influencer les politiques d'équilibrage.

10.3.1.2 Propriétés observables

Comme résultat d'une exécution l'utilisateur obtiendra des informations à propos du comportement et du placement des tâches. Si le besoin se fait ressentir, il est facile d'afficher d'autres informations en instrumentant le code de lbµ. Le type d'information disponible n'est restreint que par les informations mises à dispositions par le noyau Linux. Pour nos expériences, les informations collectées par lbµ pour chaque tâche étaient :

- le temps global d'exécution de la tâche (en comptant les moments non-ordonnancés, c'est-à-dire la durée entre le début et la fin de l'exécution) ;
- le pourcentage de temps passé sur chaque processeur (cela représente le placement) ;

```
# une tache standard
{
        cpu_mask = 0xffff
        loop = 10000000
}
# une tache RT0
{
        cpu_mask = 0x2
        sched = FIFO
        priority = 99
        loop = 110000000
        sloop = 4000
        sleep = 1000000
}
```

FIG. 10.2: Extrait de la définition d'un scénario de tâches pour lbµ. Deux tâches seulement sont définies, une tâche standard effectuant uniquement des calculs et une tâche RT0 qui utilise un pourcentage fixe d'un processeur.

- le nombre de changements de contexte que la tâche a effectués (en détaillant ceux forcés à cause de préemption de ceux voulus par la tâche).

En plus de ces informations propres à chaque tâche, après la terminaison de chaque tâche, la quantité de temps que chaque processeur a passé oisif est sauvegardée.

10.3.2 Conditions des mesures

Cette évaluation a été effectuée sur le même système que les deux précédentes. Plusieurs scénarios ont été écrits, chacun ciblant une configuration de tâches différente et spécifique dans laquelle le nouvel équilibrage de charge devrait mieux se comporter. Trois configurations du noyau ont été testées. L'une est basée sur le noyau original (Linux 2.6.12), une autre utilise ARTiS mais en gardant l'équilibrage de charge original, enfin la troisième est basée sur ARTiS complet. La deuxième configuration est utilisée uniquement pour ces tests, dans le but de mettre en évidence l'influence du mécanisme de migration et de l'équilibrage de charge ARTiS. Étant donné l'usage de l'équilibrage original du noyau, il se peut que lors de la migration de tâches entre CPU NRT et CPU RT les verrous inter-processeurs employés bloquent la préemption pour une longue durée. Cette configuration, bien qu'utilisant une partie d'ARTiS, ne peut donc pas garantir les contraintes temps-réel.

10.3.3 Observation des scénarios de test

Pour chaque politique modifiée dans l'équilibrage de charge pour ARTiS, nous présentons les résultats de l'exécution d'un scénario adapté à la politique.

10.3.3.1 Pondération de la longueur de la file d'exécution

Afin de vérifier l'amélioration apportée par la nouvelle implémentation d'estimation de la charge générée par une tâche temps-réel, un scénario avec 13 tâches standard et 3 tâches RT0 a été développé. Chacune des tâches RT0 est liée à un processeur RT différent et consomme

environ 90% de la puissance d'un processeur. Les tâches Linux n'effectuent que du calcul. Les systèmes avec ARTiS ont été configurés de manière a avoir trois processeurs RT et un processeur NRT. Dans ce scénario, l'équilibrage de charge optimal théorique, qui procure à la fois performance et équité entre les tâches, serait de placer 10 tâches standard sur le processeur NRT, et une tâche RT0 ainsi qu'une tâche standard sur chaque processeur RT. Ainsi, cela permettrait à chaque tâche standard de bénéficier de 10% du temps d'un processeur. Le calcul à effectuer par les tâches RT0 était suffisamment long pour que toutes les tâches standard soient terminées avant que les tâches RT0 ne se terminent : dans toutes les configurations les tâches RT0 ont mis 617s pour s'exécuter.

	Temps oisif CPU (μs)				**Durée tâches standard (s)**		
Configurations	NRT	RT	RT	RT	moy	δ	max
Linux standard	0	0	0	0	315	97	448
ARTiS équilibrage original	0	0	0	0	313	96	439
ARTiS nouvel équilibrage	0	1,6	1,8	1,1	392	25	439

TAB. 10.3: Résultats de l'exécution de 13 tâches standard et 3 tâches RT0 pour trois configurations du système, avec trois processeurs RT. Le temps oisif est celui observé à la fin de la neuvième tâche. La moyenne, l'écart type et le maximum de la durée d'exécution des tâches standard sont indiqués.

Le tableau 10.3 résume les résultats de l'exécution du scénario. Le temps oisif passé par chaque processeur observé à la fin de la neuvième tâche standard est indiqué. La fin de cette tâche correspond au moment où il n'y a plus assez de tâches pour occuper tous les processeurs, c'est donc le dernier instant où, si la puissance processeur est parfaitement utilisé, ces temps devraient être nuls.

Pour bien interpréter les résultats, il convient de faire quelques remarques. De manière non intuitive, l'équité entre les tâches tend à faire *augmenter* le temps d'exécution moyen, sans influencer la puissance disponible. En effet, puisque toutes les tâches commencent en même temps et que tous les processeurs, avec un équilibrage équitable ou non, sont entièrement occupés, seul le plus long temps d'exécution correspond à la performance de calcul. Meilleure est l'équité, plus le temps d'exécution des tâches tend à être identique au plus long, ce qui fait augmenter la moyenne.

Les observations avec le noyau Linux standard et avec noyau utilisant ARTiS sans l'équilibrage amélioré sont très similaires. Il y a une grande iniquité entre les 13 tâches pourtant identiques : l'écart type du temps d'exécution est près de 100s, par rapport à une moyenne d'environ 314s. Certaines des tâches ont mis plus de deux fois plus de temps avant de se terminer. Ceci a été confirmé par l'observation durant les premières minutes de l'exécution du scénario : tous les processeurs exécutent 4 tâches, temps-réel ou standard.

Les observations de la troisième configuration, celle utilisant le mécanisme de pondération de la longueur de la file d'exécution de l'ordonnanceur, montrent un écart type de l'exécution des tâches seulement de 25s. Toutes les tâches mettent approximativement le même temps à se terminer : la même puissance processeur a été équitablement attribuée à chaque tâche standard. Cela a été confirmé durant l'exécution, le processeur NRT exécutait 10 tâches pendant que les processeurs RT n'en exécutaient que 2 (une tâche RT0 et une tâche standard).

La moyenne du temps d'exécution est supérieure aux deux autres configurations (392s

au lieu d'environ 314s), ce n'est pas un signe de ralentissement mais uniquement le fait que plus de tâches aient mit pratiquement autant de temps que la tâche la plus lente. D'ailleurs, ce temps maximum d'exécution est similaire aux deux premières configurations. On pourrait néanmoins noter que les processeurs ont passé un petit peu de temps oisif, minimal mais significatif face aux deux premières configurations. Cela est causé par le fait que, sur un processeur RT, lorsqu'une tâche se termine, il n'y a plus aucune tâche à exécuter et le processeur doit attendre qu'une autre tâche lui soit transmise. Cela n'arrive pas lorsque trois tâches standard s'exécutent en même temps sur le processeur RT. C'est une petite perte en puissance processeur qui peut être considérée comme le « coût » de l'équité.

10.3.3.2 Estimation de la prochaine tentative de migration

Le mécanisme d'estimation de la prochaine tentative de migration doit favoriser les tâches mettant fréquemment en danger les propriétés temps-réel sur les processeurs NRT, afin d'éviter le problème de ping-pong entre les processeurs RT et NRT. Le scénario que nous avons mis en place pour valider ce mécanisme est basé sur l'exécution de deux ensembles de 8 tâches standard. Un ensemble est constitué de tâches faisant un appel système prenant un verrou très fréquemment (environ toutes les 200µs), il est appelé MF (Migrant Fréquemment). L'autre ensemble est constitué de tâches prenant un verrou beaucoup plus rarement (environ 1000 fois moins souvent), il est appelé MR (Migrant Rarement). Les processeurs sont partitionnés en deux CPU NRT et deux CPU RT. Un équilibrage de charge idéal ordonnancerait l'ensemble MF sur les processeurs NRT et l'ensemble MR sur les processeurs RT (en ne migrant les tâches sur les processeurs NRT uniquement le temps de la prise de verrou).

	Temps oisif CPU (µs)				**Chgt ctxt**	**Chgt ctxt**
Configurations	NRT	NRT	RT	RT	MR	MF
Linux standard	0	0	0	0	575	535
ARTiS équil. orig.	0	0	150	195	400	7200
ARTiS nouvel équil.	0	0	130	142	310	845

	Rép. des tâches MR (%)				**Rép. des tâches MF (%)**			
Configurations	NRT	NRT	RT	RT	NRT	NRT	RT	RT
Linux standard	25	37.5	37.5	0	25	12.5	12.5	50
ARTiS équil. orig.	10	53	18	19	63	21	15	1
ARTiS nouvel équil.	15	50	20	15	68	31	1	0

TAB. 10.4: Résultats pour trois configurations différentes de l'exécution de 8 tâches prenant fréquemment un verrou (MF) et de 8 tâches prenant rarement un verrou (MR) sur un système avec deux processeurs NRT et deux processeurs RT. Sont indiqués : le temps oisif passé par chaque processeur avant la fin de la première tâche, le nombre moyen de changements de contexte, et la répartition des tâches sur les processeurs.

Le tableau 10.4 résume les résultats. Sur un noyau standard, les deux types de tâches sont placés de manière similaire. Chaque tâche utilise 12,5% de n'importe quel processeur. Cela est attendu puisque le type de processeur ne joue aucun rôle et toutes les tâches sont considérées similaires (car la prise de verrou n'influence pas l'ordonnancement). La différence

de répartition des ensembles entre les processeurs est due uniquement au hasard lors du choix des tâches à déplacer. Au total, il y avait donc quatre tâches par processeur.

Avec ARTiS, dans les deux configurations, les tâches ne pouvaient pas s'exécuter tout le temps sur le processeur RT. Chaque processeur a été un petit peu oisif, même avant la fin de la première tâche. Cela implique une perte de puissance de calcul. Cet effet provient de la migration automatique nécessaire à garantir les propriétés temps-réel. Alors qu'avec l'équilibrage de charge original les tâches MF ont passé environ 16% de leur temps sur un processeur RT, avec le nouvel équilibrage, elles n'y ont passé que 1% de leur temps. Cela confirme que l'estimation de prochaine tentative de migration est bien capable de différencier les tâches tentant de migrer fréquemment de celles qui ne le font pas et que cette information permet d'influencer l'équilibrage. Dans chaque configuration les tâches MR ont passé environ 35% de leur temps sur les processeurs RT. Dans la configuration avec le nouvel équilibrage, on aurait pu s'attendre à ce que les tâches MR passent plus de temps sur les processeurs RT, cette inefficacité est le résultat des performances réduites de la politique de déclenchement à l'initiative de l'émetteur par rapport à celles avec l'initiative du récepteur (qui peut aller chercher une autre tâche dès que sa file d'exécution est vide). Dans le sens où le nouvel équilibrage est obligatoire pour éviter de verrous inter-processeur, on peut considérer que la prise en compte de l'asymétrie permet de compenser la politique à l'initiative de l'émetteur.

L'effet du nouveau mécanisme est clairement observable au niveau du nombre de changements de contexte forcés. Chaque migration entre processeurs implique un changement de contexte. Avec le mécanisme original, il y a eu environ 7200 changements de contexte par tâche MF, c'est l'effet « ping-pong ». L'évolution de l'équilibrage permet d'éviter cet effet, ainsi les tâches subissent 10 fois moins de changements de contexte.

10.4 Synthèse de la validation expérimentale

La validation de l'implémentation actuelle d'ARTiS a été faite expérimentalement par l'observation des trois principaux aspects concernant le comportement du système temps-réel. Nous avons vérifié l'impact de la solution sur l'amélioration des latences d'interruption par rapport au noyau standard qui permet de qualifier ARTiS de système d'exploitation temps-réel. Outre le fait que les latences ont été divisées par un facteur de plus de 10, les latences de l'espace utilisateur sont du même ordre de grandeur que celles de l'espace noyau. Nous avons également vérifié que l'exécution de la tâche temps-réel, une fois ordonnancée, est aussi bornée dans le temps. Enfin, plusieurs scénarios de charge ont été définis dans le but de tester l'équilibrage face aux difficultés supplémentaires introduites par l'asymétrie des processeurs. Les politiques d'équilibrage se sont montrées conformes à leur comportement théorique, permettant au système de garder une grande puissance de calcul même une fois l'asymétrie introduite dans le système. Nous avons également proposé un nouvel outil, lbµ, spécialement dédié à la reproduction et l'observation de charge.

Conclusion de la deuxième partie

Dans cette partie nous avons proposé un modèle de système à même de fournit des propriétés temps-réel et des hautes-performances de calcul à la fois. L'approche est basée sur le partitionnement du système multiprocesseur entre processeurs RT, où certaines tâches sont protégées de délais lors du traitement des évènements matériels, et processeur NRT, où tout code qui pourrait introduire des délais est exécuté. Ce partitionnement n'exclue pas un équilibrage de charge des tâches sur l'ensemble de la machine, il implique seulement que certaines tâches soient automatiquement migrées lorsqu'elles sont sur le point d'entrer dans des sections non-préemptibles. En outre, nous avons proposé des politiques d'équilibrage de charge qui prennent en compte l'asymétrie afin de maintenir un usage efficace de la puissance de calcul disponible.

Une implémentation d'ARTiS est disponible, basée sur le noyau Linux et écrite pour les architectures x86 et IA-64. En plus des mécanismes de migration automatique et d'équilibrage de charge, une interface a été développée pour permettre à l'utilisateur de dynamiquement paramétrer le système. La configuration du système est faite en spécifiant d'une part la priorité de chaque tâche et d'autre part en partitionnant les processeurs et les interruptions.

La validation de l'implémentation a été réalisée en observant les trois principaux aspects du système. Une très nette amélioration des latences d'interruption a été confirmée, rendant les latences utilisateur pratiquement aussi faibles que celles observables dans le noyau. La variation de temps d'exécution d'une tâche temps-réel est pratiquement nulle, similairement au noyau standard. Les nouvelles politiques d'équilibrage de charge ont été confirmées comme respectant leur comportement théorique, permettant de garder une large capacité de calcul même après l'introduction de l'asymétrie dans le système.

Conclusion générale

Chapitre 11

Conclusion

11.1 Résumé **213**
11.2 Perspectives **215**

11.1 Résumé

Les travaux de cette thèse s'inscrivent dans le cadre du développement de systèmes sur puce multiprocesseurs dédiés à des applications de traitement de signal intensif. Nous avons d'abord exposé les difficultés de la conception de SoC liées principalement à l'augmentation constante de la complexité des puces et de la complexité des applications visées. Nous avons présenté les bases de l'ingénierie dirigée par les modèles puis le flot de conception usuel des SoC : l'application et l'architecture sont développées simultanément puis l'association est réalisée, à partir de laquelle est possible de faire des co-simulations à des niveaux de plus en plus détaillés tout en corrigeant et raffinant les descriptions et implémentations du logiciel et du matériel. Nous avons également présenté le méta-modèle Gaspard qui permet la représentation des SoC parallèles sous forme compacte et autorise une modélisation qui suit le flot de conception usuel en proposant des concepts particuliers pour l'application, l'architecture, et l'association.

Les premières contributions présentées consistent à étendre le méta-modèle Gaspard pour pouvoir concevoir les modèles dits *exécutables*, c'est-à-dire des modèles contenant toutes les informations requises pour obtenir un code complet du SoC. Nous avons introduit une sémantique utilisant la notion de composant pour la distribution d'applications parallèles et hiérarchiques sur des architectures elles aussi parallèles et hiérarchiques. Puis nous avons proposé une évolution du méta-modèle pour relier les boîtes noires que sont les composants élémentaires aux codes des IP. Outre le fait de permettre aux transformations de modèles de générer plus qu'un simple squelette du code, cette seconde proposition favorise la productivité du concepteur en autorisant l'usage de bibliothèques de composants, en permettant de retarder la spécialisation du code des IP à l'étape de déploiement des composants et en assurant la représentation de fonctionnalités indépendantes de la cible de compilation.

Par la suite, nous avons proposé un haut niveau d'abstraction pour la co-simulation dit *au motif près* (PA). À ce niveau, l'application de traitement de signal est représentée uniquement par les accès aux données et l'ordonnancement entre les tâches. Cette abstraction ne prend pas en compte la lecture des instructions et même les appels aux fonctions des IP peuvent être

évités. Nous avons spécifié la projection du modèle de calcul des applications Gaspard sur un modèle d'exécution. Pour optimiser les performances, un ordonnancement dynamique des tâches est effectué aux plus hauts niveaux de hiérarchie de l'application, ceux distribués sur plusieurs processeurs. Les tâches à grain fin sont ordonnancées statiquement. Puis nous avons mis en place une simulation au niveau d'abstraction PA en SystemC qui respecte le modèle d'exécution précédemment défini. Chaque processus SystemC représentant un processeur ordonnance plusieurs threads, ce mécanisme simule l'ordonnancement dynamique.

Dans le chapitre suivant nous avons présenté la chaîne de compilation mise en place dans Gaspard pour générer une simulation SystemC à partir d'un modèle de MPSoC. Cette chaîne respecte les recommandations de l'IDM, elle est constituée de plusieurs transformations qui prennent en entrée et en sortie des modèles conformes à des méta-modèles prédéfinis. Chaque transformation est construite par assemblage d'un ensemble de règles de transformation. La première transformation développée permet d'obtenir à partir d'un modèle Gaspard un modèle contenant toute l'information mais exprimée de manière plus proche du code. En particulier l'association est interprétée : l'application est découpée sur chaque processeur et les données sont représentées sur les mémoires. Pour refléter la distribution nous avons fait usage des polyèdres entiers. La seconde transformation détaillée génère le code SystemC, il contient à la fois la simulation matérielle et logicielle.

Après cela nous avons détaillé l'usage du flot complet de Gaspard et de la simulation au motif près avec un exemple d'encodeur H.263 placé sur un MPSoC à base de processeurs MIPS. Cela a permis de valider les transformations de modèles ainsi que la simulation à haut niveau d'abstraction. Cette étude de cas a aussi été l'occasion de présenter une exploration d'architecture où nous avons aisément fait varier le nombre de processeurs puis le nombre de bancs mémoire.

Dans la seconde partie de ce mémoire nous avons mis l'accent sur les propriétés temps-réel des systèmes multiprocesseurs. Nous avons commencé par exposer les difficultés techniques que pose le besoin conjoint de systèmes parallèles et d'applications temps-réel : la gestion d'un système multiprocesseur et les mécanismes nécessaires pour profiter pleinement de la puissance de calcul mise à disposition sont très difficiles à garantir sans effet sur l'exécution des tâches temps-réel.

Nous proposons une approche nommée ARTiS inspirée des systèmes utilisant des processeurs « protégés » mais qui autorise néanmoins à user de toute la puissance de calcul. En effet, les processeurs utilisés pour garantir les propriétés temps-réel peuvent exécuter toutes les tâches. C'est uniquement lorsqu'une tâche fait appel à une partie du noyau qui pourrait mettre en danger ces propriétés que cette tâche est migrée vers un autre processeur. Pour tenir compte de l'asymétrie introduite entre les processeurs, nous avons également proposé des politiques d'équilibrage de charge particulières qui prennent en compte le comportement des tâches pour les placer sur le type de processeur le mieux adapté tout en maximisant la puissance de calcul utilisée.

Cette proposition a été implémentée dans le noyau Linux puis validée expérimentalement concernant la latence de réponse aux interruptions, la variation du temps d'exécution, et les modifications des politiques d'équilibrage de charge. Nous avons ainsi mis en évidence que les tâches peuvent bénéficier de propriétés temps-réel dur en étant écrites et exécutées complètement usuellement. Cela est bénéfique au concepteur qui peut ainsi développer et déboguer l'application en utilisant les interfaces et les outils habituels.

11.2 Perspectives

Le travail effectué dans cette thèse peut être poursuivi le long de nombreuses directions, nous en présentons ici quelques-unes.

Introduction du contrôle dans les applications

Actuellement l'environnement Gaspard ne vise que les applications de traitement de signal systématique. Mais il existe déjà des travaux, en particulier ceux de Ouassila Labbani [62], qui visent à intégrer la notion de flot de contrôle dans au flot de données des applications Gaspard. Outre le fait que les tâches puissent être dynamiquement choisies parmi un ensemble d'alternatives, cela introduit un nouveau type de tâches dites *de contrôle* qui dirigent les différentes alternatives d'exécution selon les entrées et les états précédents.

Vis-à-vis du modèle d'exécution ces ajouts ne devraient pas introduire trop de changements. Principalement, l'arborescence de tâches peut être calculée sans prendre en compte les différentes alternatives, et c'est uniquement à l'exécution lors de l'appel depuis une tâche parente que le choix est pris. Néanmoins, selon la taille des FIFO de tableaux entre les tâches, il faut prendre un certain nombre de précautions pour ne répercuter ni trop tôt ni trop tard les directives de contrôle.

En ce qui concerne la génération de code de simulation, bien sûr l'introduction du contrôle entraînera la génération du code de chaque tâche alternative. Il ne sera plus possible de placer toute une hiérarchie de tâches dans une seule fonction mais il faudra remplacer chaque niveau contrôlé par un appel à une fonction différente. Par ailleurs lors de la simulation PA, les tâches de contrôle devront impérativement être exécutées car leurs calculs peuvent influencer le comportement des accès mémoires et des durées d'exécutions de certaines tâches.

Extension du modèle d'exécution et de la génération de code

Le modèle d'exécution que nous avons proposé ainsi que les transformations de modèles ne permettent pas de prendre en compte toutes les architectures et associations possibles. En particulier, deux types de répartitions intéressantes ne sont pas gérés par Gaspard actuellement : le placement des données sur des mémoires distribuées et l'allocation de tâches entre des processeurs et des accélérateurs matériels.

Pour l'instant, lors du placement des données, toutes les mémoires sont considérées partagées, c'est-à-dire que tous les processeurs peuvent accéder à n'importe quelle mémoire. Dans le cas des mémoires distribuées, il faut tenir compte du fait que, selon le processeur, certaines mémoires ne peuvent être atteintes directement. La route entre le processeur et la mémoire doit être calculée et éventuellement des tâches supplémentaires servant de relais doivent être placées sur les composants actifs intermédiaires.

Nous avons présenté dans cette thèse l'exécution de tâches sur de multiples processeurs, éventuellement répétés. Dans ses travaux, Sébastien Le Beux a présenté l'exécution de tâches sur des accélérateurs matériels [64]. Cependant il n'est pas encore possible d'allouer une partie de l'application sur des processeurs et une autre partie sur des accélérateurs matériels. La recherche dans cette direction devra proposer un modèle d'exécution capable de synchroniser les tâches entre elles, et définissant une gestion du flot de données qui soit efficace aussi bien lors du traitement sur un processeur que sur un accélérateur matériel (où chaque opération peut être pipelinée). Il faudra entre autres déterminer s'il est plus judicieux de voir

les accélérateurs matériels comme esclaves d'un processeur ou au contraire qu'ils traitent eux-mêmes les transferts de données et les synchronisations avec les processeurs.

Introduction des mécanismes d'inter-répétition

Une possibilité pour étendre le modèle de calcul des applications Gaspard est d'ajouter la notion de liens inter-répétitions. Grâce à ce concept les répétitions d'une tâche peuvent utiliser comme données d'entrée les données de sortie d'autres répétitions de la même tâche. Cela permet de représenter un certain nombre de fonctions simples telles que la sommation séquentielle, la convergence de calcul...

Dans le modèle de d'exécution, une inter-répétition peut être sous la forme de deux mécanismes complémentaires. D'une part, c'est un passage de tableaux de données intermédiaires, similaire à celui qui est effectué entre deux tâches différentes à l'exception que la taille de stockage des données ne peut pas être choisie arbitrairement mais doit dépendre du nombre de répétitions séparant la création et l'usage des données. D'autre part, une inter-répétition doit également être vue comme une dépendance d'ordre d'exécution : une répétition ne doit être exécutée qu'après que toutes les répétitions générant ses données d'entrée aient été exécutées. Pour exprimer cela, l'usage des polyèdres entiers pourrait être une piste. En effet, avec l'outil CLooG, le parcours des points d'un polyèdre peut être ordonné afin de suivre un « front de lecture » qui correspondrait aux dépendances d'ordre spécifiées par l'inter-répétition.

Automatisation de l'exploration de l'espace de conception

Nous avons vu que le flot de conception dans Gaspard facilite l'exploration de l'espace de conception. En effet, il est très facile de modifier des paramètres aussi bien dans l'architecture, dans l'application que dans l'association. Passer de 4 à 8 processeurs peut consister à changer uniquement un nombre. De plus, il est possible de cibler différents niveaux d'abstraction de la simulation sans avoir à modifier le modèle de SoC. Il est ainsi possible de parcourir très rapidement et grossièrement l'espace de conception en simulant les différentes possibilités au niveau d'abstraction PA, puis les meilleures solutions peuvent être plus précisément observées et sélectionnées en passant à un niveau de simulation plus bas.

L'étape suivante de l'environnement Gaspard consiste à automatiser complètement cette exploration. Cela implique l'ajout d'informations par le concepteur. D'une part, il faut spécifier quels sont les paramètres du modèle de SoC qui peuvent être modifiés pour parcourir l'espace de conception. Éventuellement, ces paramètres ne sont pas indépendants et donc doivent être modifiés conjointement selon une relation particulière. D'autre part, il faut également spécifier les critères multiples auxquels devra se conformer le SoC final (temps d'exécution de l'application, consommation d'énergie, taille physique, coût de fabrication...).

L'algorithme d'exploration doit alors être capable de prendre en compte et d'interpréter les résultats fournis par les simulations. À partir de ces résultats, les modifications à apporter sur le modèle pour converger vers un SoC respectant les critères doivent être déterminées. Une des pistes possibles pour sélectionner les paramètres à modifier est d'utiliser la traçabilité des transformations de modèles.

Extension des politiques d'ordonnancement temps-réel dans Linux

L'ordonnancement actuel des tâches temps-réel dans ARTiS est basé sur la très rudimentaire politique d'ordonnancement FIFO du noyau Linux, qui ne prend en compte que la priorité statique des tâches. Afin de pouvoir traiter correctement les cas où plusieurs tâches à haute priorité sont amenées à s'exécuter en même temps, l'usage de politiques plus sophistiquées est nécessaire.

Il serait avantageux d'ajouter les politiques d'ordonnancement temps-réel usuelles telles que EDF (*Earliest Deadline First*) ou RM (*Rate Monotonic*). Cette évolution requiert la définition plus complète du modèle de tâche dans le noyau, une extension des API classiques, et l'implémentation de ces nouvelles politiques d'ordonnancement. Les tâches de plus haute priorité pourraient alors être des tâches périodiques exécutant une boucle infinie. L'extension de l'API permettrait d'exprimer la prochaine échéance ou la fréquence des appels d'une tâche ainsi que la puissance de calcul nécessaire. Une organisation hiérarchique de l'ordonnanceur serait introduite : l'exécution de la tâche ayant présentement la plus haute priorité serait remplacée par un ordonnancement amélioré gérant l'ensemble des tâches à cette priorité.

Bibliographie personnelle

[1] Rabie Ben Atitallah, Éric Piel, Smail Niar, Philippe Marquet, and Jean-Luc Dekeyser. Multilevel MPSoC simulation using an MDE approach. In *IEEE International SoC Conference (SoCC 2007)*, Hsinchu, Taiwan, September 2007.

[2] Rabie Ben Atitallah, Éric Piel, Julien Taillard, Smail Niar, and Jean-Luc Dekeyser. From High Level MPSoC description to SystemC Code Generation. In *International Mod-Easy'07 Workshop in conjunction with Forum on specification and Design Languages (FDL'07)*, Barcelona, Spain, September 2007.

[3] Pierre Boulet, Philippe Marquet, Éric Piel, and Julien Taillard. Repetitive Allocation Modeling with MARTE. In *Forum on specification and design languages (FDL'07)*, Barcelona, Spain, September 2007. Papier invité, nom des auteurs par ordre alphabétique.

[4] ITEA Hyades Project. Linux for high performance and real-time computing on SMP systems. In *Sixth Realtime Linux Workshop*, Singapore, November 2004.

[5] Philippe Marquet, Éric Piel, Julien Soula, and Jean-Luc Dekeyser. Implementation of ARTiS, an asymmetric real-time extension of SMP Linux. In *Sixth Realtime Linux Workshop*, Singapore, November 2004.

[6] Philippe Marquet, Éric Piel, Julien Soula, and Jean-Luc Dekeyser. ARTiS, un système d'exploitation temps-réel asymétrique. In *4e édition de la Conférence Française sur les Systèmes d'Exploitation (CFSE'4) - ACM*, Le Croisic, France, April 2005.

[7] Éric Piel, Philippe Marquet, Julien Soula, Christophe Osuna, and Jean-Luc Dekeyser. ARTiS, an asymmetric real-time scheduler for Linux on multi-processor architectures. Research Report RR-5781, INRIA, France, December 2005.

[8] Éric Piel. *Équilibrage de charge pour systèmes temps-réel asymétriques sur multi-processeurs*. Mémoire de DEA, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, France, June 2004.

[9] Éric Piel, Philippe Marquet, Julien Soula, and Jean-Luc Dekeyser. Load-balancing for a real-time system based on asymmetric multi-processing. In *16th Euromicro Conference on Real-Time Systems, WIP session*, Catania, Italy, June 2004.

[10] Éric Piel, Philippe Marquet, Julien Soula, and Jean-Luc Dekeyser. Asymmetric scheduling and load balancing for real-time on Linux SMP. In *Workshop on Scheduling for Parallel Computing (SPC 2005)*, Poznan, Poland, September 2005. Lecture Notes in Computer Science vol. 3911. ©Springer-Verlag.

[11] Éric Piel, Philippe Marquet, Julien Soula, and Jean-Luc Dekeyser. Real-time systems for multi-processor architectures. In *14th International Workshop on Parallel and Distributed Real-Time Systems, In conjunction with IPDPS, 20th IEEE International Parallel and Distributed Processing Symposium*, Island of Rhodes, Greece, April 2006. IEEE Computer Society Press. Papier invité.

Bibliographie

[12] Netta Aizenbud-Resher, Richard F. Paige, Julia Rubin, Yael Shalam-Gafni, and Dimitrios S. Kolovos. Operational semantics for traceability. In *ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings*, 2005. 26

[13] Gregory E. Allen and Brian L. Evans. Real-time sonar beamforming on workstations using process networks and POSIX threads. *IEEE Transactions on Signal Processing*, pages 921–926, mars 2000.

[14] Abdelkader Amar, Pierre Boulet, Jean-Luc Dekeyser, and Frans Theeuwen. Distributed process networks using half FIFO queues in CORBA. In *ParCo'2003, Parallel Computing*, Dresden, Germany, septembre 2003.

[15] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, septembre 2004.

[16] Rabie Ben Atitallah. *Modèles et simulation de systèmes sur puce multiprocesseurs – Estimation des performances et de la consommation d'énergie*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, décembre 2007.

[17] Rabie Ben Atitallah, Pierre Boulet, Arnaud Cuccuru, Jean-Luc Dekeyser, Antoine Honoré, Ouassila Labbani, Sébastien Le Beux, Philippe Marquet, Éric Piel, Julien Taillard, and Huafeng Yu. Gaspard2 uml profile documentation. Rapport technique 0342, INRIA, septembre 2007. 37

[18] Luca Benini, Davide Bertozzi, Davide Bruni, Nicola Drago, Franco Fummi, and Massimo Poncino. Systemc cosimulation and emulation of multiprocessor soc designs. *Computer*, 36(4) :53–59, avril 2003. 33

[19] Ben Bennet. From dual-core to many-core, is the industry ready ? In *PPAM 2005, Sixth international conference on parallel processing and applied mathematics*, Poznan, Poland, septembre 2005. 7, 35

[20] Lossan Bondé. *Transformations de Modèles et Interopérabilité dans la Conception de Systèmes Hétérogènes sur Puce à Base d'IP*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, décembre 2006.

[21] Pierre Boulet. Array-OL revisited, multidimensional intensive signal processing specification. Rapport de recherche RR-6113, INRIA, février 2007. 36, 42

[22] Stephen Brosky. Symmetric multiprocessing and real-time in PowerMAX OS. White paper, Concurrent Computer Corporation, Fort Lauderdale, FL, 2002.

[23] Steve Brosky and Steve Rotolo. Shielded processors : Guaranteeing sub-millisecond response in standard Linux. In *Workshop on Parallel and Distributed Real-Time Systems, WPDRTS'03*, Nice, France, avril 2003.

[24] Lukai Cai and Daniel Gajski. Transaction level modeling : An overview. In *Hardware/-Software Codesign and System Synthesis*, pages 19–24, octobre, 2003. 32

[25] Pierre Cloutier, Paolo Montegazza, Steve Papacharalambous, Ian Soanes, Stuart Hughes, and Karim Yaghmour. DIAPM-RTAI position paper. In *Second Real Time Linux Workshop*, Orlando, FL, novembre 2000.

[26] Antoine Colin and Isabelle Puaut. Worst-case execution time analysis of the RTEMS Real-Time operating system. In *13th Euromicro Conference on Real-Time Systems (ECRTS'01)*, Delft, The Netherlands, juin 2001.

[27] VISPL Consortium. Vector signal image processing library, 2002. http://www.vsipl.org/.

[28] Guy Cote, B. Erol, M. Gallant, and F. Kossentini. H.263+ : video coding at low bit rates. *IEEE Trans. On Circuits And Systems For Video Technology*, novembre 1998.

[29] Arnaud Cuccuru. *Modélisation unifiée des aspects répétitifs dans la conception conjointe logicielle/matérielle des systèmes sur puce à hautes performances*. PhD thesis, Université des sciences et technologies de Lille, Laboratoire d'informatique fondamentale de Lille, novembre 2005. 36

[30] Krzysztof Czarnecki and Simon Helsen. Classification of model tranformation approaches. In *Proceeding of OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture*, 2003. 24, 26

[31] Lawrie D.A. Access and alignment of data in an array processor. *IEEE Trans. Comput.*, C-24(12) :1145–1155, décembre 1975. 45

[32] DaRT Team LIFL/INRIA, Lille, France. Graphical Array Specification for Parallel and Distributed Computing (GASPARD2). https://gforge.inria.fr/projects/gaspard2/, 2008. 35

[33] Alain Demeure, Anne Lafarge, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd, and Jean-Louis Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France, septembre 1995. 36

[34] Tata Research Development and Design Centre. Modelmorf – a model transformer. http://www.tcs-trddc.com/ModelMorf. 25

[35] Adam Donlin. Transaction level modeling : flows and use models. In *Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 75–80, Stockholm, Sweden, 2004. 31

[36] Philippe Dumont. *Spécification Multidimensionnelle pour le traitement du signal systématique*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, décembre 2005.

[37] Eclipse Consortium. EMF. http://www.eclipse.org/emf, 2007. 18, 25

[38] Eclipse Consortium. JET, Java Emitter Templates. http://www.eclipse.org/modeling/m2t/?project=jet, 2007.

[39] Magnus Ekman and Per Stenström. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, Texas, USA, mars 2005. 33

[40] Anne Etien, Cedric Dumoulin, and Emmanuel Renaux. Towards a unified notation to represent model transformation. Rapport de recherche RR-6187, INRIA, mai 2007. 25

[41] Malcon Eva. *SSADM Version 4 : A User's Guide*. McGraw-Hill Publishing Co, avril 1994. 16

[42] Jean-Marie Favre. Concepts fondamentaux de l'IDM. De l'ancienne egypte à l'ingénierie des langages. In *2èmes Journées sur l'Ingénierie Dirigée par les Modèles (IDM'06)*, Lille, France, 2006. 17

[43] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles, au-delà du MDA*. Hermès Science, Lavoisier, janvier 2006. 16

[44] Finite State Machine Labs, Inc. RealTime Linux (RTLinux). http://www.fsmlabs.com/.

[45] Cyril Fonlupt. *Distribution Dynamique de Données sur Machines SIMD*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, décembre 1994.

[46] Franco Fummi, Mirko Loghi, Stefano Martini, Marco Monguzzi, Giovanni Perbellini, and Massimo Poncino. Virtual hardware prototyping through timed hardware-software co-simulation. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'05)*, pages 798–803, Munich, Germany, mars 2005. 33

[47] D. D. Gajski and R. Kuhn. Guest editor introduction : New VLSI-tools. *IEEE Computer*, 16(12) :11–14, décembre 1983. 28, 29

[48] Philippe Gerum. I-pipe, 2005. http://www.adeos.org/.

[49] A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, and O. Yamamoto. A hardware-software co-simulator for embedded system design and debugging. In *ASP-DAC '95 : Proceedings of the 1995 conference on Asia Pacific design automation*, page 25, Makuhari, Japan, 1995. ACM Press. 31

[50] Nicholas Mc Guire and Qingguo Zhou. Benchmarking - cache issues. In *Seventh Real-Time Linux Workshop, RTLWS'05*, Lille, France, novembre 2005.

[51] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0 : Faster and more flexible program analysis. In *Workshop on Modeling, Benchmarking and Simulation*, Madison, Wisconsin, USA, juin 2005. 33

[52] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0. Rice University, Houston, TX, janvier 1997.

[53] Douglas R. Hofstadter. *Gödel, Escher, Bach : an Eternal Golden Braid*. Basic Books, New York, USA, 1979. 21

[54] Shinya Honda, Takayuki Wakabayashi, Hiroyuki Tomiyama, and Hiroaki Takada. RTOS-centric hardware/software cosimulator for embedded system design. In *Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'04)*, Stockholm, Sweden, septembre 2004. 34

[55] Jackson Hu. Will moore's law continue ? In *IEEE International SoC Conference (SoCC 2007)*, Hsinchu, Taiwan, septembre 2007. Keynote presentation. 8

[56] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9) :1098–1101, septembre 1952.

[57] ITRS, International Technology Roadmap for Semiconductors. Design, 2005 edition. http://www.itrs.net/, 2005. 8, 34

[58] ITRS, International Technology Roadmap for Semiconductors. Executive summary, 2005 edition. http://www.itrs.net/, 2005. 26

[59] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference : MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM*, Montego Bay, Jamaica, octobre 2005. 25

[60] 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.

[61] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997. 16

[62] Ouassila Labbani. *Modélisation à haut niveau du contrôle dans des applications de traitement systématique à parallélisme massif*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, novembre 2006.

[63] Patrick LAURENT. Generic image array library. http://www.ient.rwth-aachen.de/team/laurent/genial/genial.html, 2007.

[64] Sébastien Le Beux. *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, décembre 2007.

[65] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, janvier 1987.

[66] Robert Love. *Linux Kernel Development*. Sams Publishing, août 2003.

[67] Paul E. McKenney. Attempted summary of "RT patch acceptance" thread. Linux Kernel Mailing List, juillet 2005. http://lkml.org/lkml/2005/7/11/118.

[68] Paul E McKenney, Ingo Molnar, Dipankar Sarma, and Suparna Bhattacharya. Extending RCU for realtime and embedded workloads. In *Linux Symposium*, Ottawa, Canada, août 2006.

[69] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A taxonomy of model transformations. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. http://drops.dagstuhl.de/opus/volltexte/2005/11. 25

[70] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4) :316–344, 2005. 16

[71] Kevin Morgan. Linux for real-time systems : Strategies and solutions. White paper, MontaVista Software, Inc., 2001.

[72] Kevin Morgan. Preemptible Linux : A reality check. White paper, MontaVista Software, Inc., 2001.

[73] David Mosberger and Stéphane Eranian. *IA-64 Linux Kernel : Design and Implementation*. Prentice-Hall, 2002.

[74] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, octobre 2005. 25

[75] OAR Corporation. RTEMS home page. http://www.rtems.com/.

[76] Object Management Group, Inc., editor. *Common Object Request Broker Architecture (CORBA), Version 3.0*, chapter 3, IDL Syntax & Semantics. http://www.omg.org/cgi-bin/doc?formal/02-06-39, juin 2002.

[77] Object Management Group, Inc., editor. *MOF 2.0 Core Final Adopted Specification*. http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 2003. 18

[78] Object Management Group, Inc., editor. *UML 2 Infrastructure (Final Adopted Specifcation)*. http://www.omg.org/cgi-bin/doc?ptc/2003-09-15, septembre 2003. 16, 23

[79] Object Management Group, Inc. MOF Query / Views / Transformations. http://www.omg.org/docs/ptc/05-11-01.pdf, novembre 2005. OMG paper. 25

[80] Object Management Group, Inc., editor. *SysML v0.9*. http://www.omg.org/cgi-bin/doc?ad/05-01-03, janvier 2005. 36

[81] Object Management Group, Inc., editor. *(UML) Profile for Schedulability, Performance, and Time Version 1.1*. http://www.omg.org/technology/documents/formal/schedulability.htm, janvier 2005. 36

[82] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD Thesis, EECS Department, University of California, Berkeley, CA, décembre 1995.

[83] Planet MDE. Model Driven Engineering, 2007. http://planetmde.org. 16

[84] ProMarte partners. UML Profile for MARTE, Beta 1. http://www.omg.org/cgi-bin/doc?ptc/2007-08-04, août 2007. 36

[85] Imran Rafiq Quadri, Pierre Boulet, and Jean-Luc Dekeyser. Modeling of topologies of interconnection networks based on multidimensional multiplicity. Rapport de recherche RR-6201, INRIA, mai 2007. 45

[86] Bran V. Selic. On the semantic foundations of standard uml 2.0. In *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2004*, Bertinoro, Italy, septembre 2004. 23

[87] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6) :84–93, 2003. 33

[88] Sillicon Graphics, Inc. REACT : Real-time in IRIX. Rapport technique, Sillicon Graphics, Inc., Mountain View, CA, 1997.

[89] The SoCLib project : An open modelling and simulation platform for system on chip design. `http://soclib.lip6.fr/`.

[90] 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.

[91] John A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, 28(2-3) :237–253, novembre 2004.

[92] Perdita Stevens. Bidirectional model transformations. Presentation in Summer school on Generative and Transformational Techniques in Software Engineering 2007 (GTTSE'07), juillet 2007. `http://twiki.di.uminho.pt/twiki/pub/Events/GTTSE2007/TechnologyPresentations/stevens.pdf`. 26

[93] Till Straumann. Open source real-time operating systems overview. In *8th International Conference on Accelerator and Large Experimental Physics Control Systems*, San Jose, California, USA, novembre 2001.

[94] Julien Taillard. *Modélisation d'IP pour la simulation SystemC d'OS de type Linux embarqué*. Master thesis, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, juillet 2005. 34

[95] Hubert Tardieu, Arnold Rochfeld, and René Colletti. *La Méthode Merise : Principes et outils*. Editions d'Organisation, 1991. 16

[96] Hiroyuki TOMIYAMA, Shin ichiro CHIKADA, Shinya HONDA, and Hiroaki TAKADA. An rtos-based design and validation methodology for embedded systems. *IEICE Transactions on Info and Systems*, E88-D(9) :2205–2208, septembre 2005. 34

[97] INRIA Triskell. Kermeta. `http://www.kermeta.org/`. 25

[98] John D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, octobre 1994.

[99] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS : accelerating microarchitecture simulation via rigorous statistical sampling. In *30th Annual International Symposium on Computer Architecture*, San Diego, USA, juin 2003. 33

[100] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. A bidirectional transformation approach towards automatic model synchronization. In *Summer school on Generative and Transformational Techniques in Software Engineering 2007 (GTTSE'07)*, Braga, Portugal, juillet 2007. 26

[101] Victor Yodaiken. The RTLinux manifesto. In *Proc. of the 5th Linux Expo*, Raleigh, NC, mars 1999.

Troisième partie

Annexes

Annexe A

Déploiement

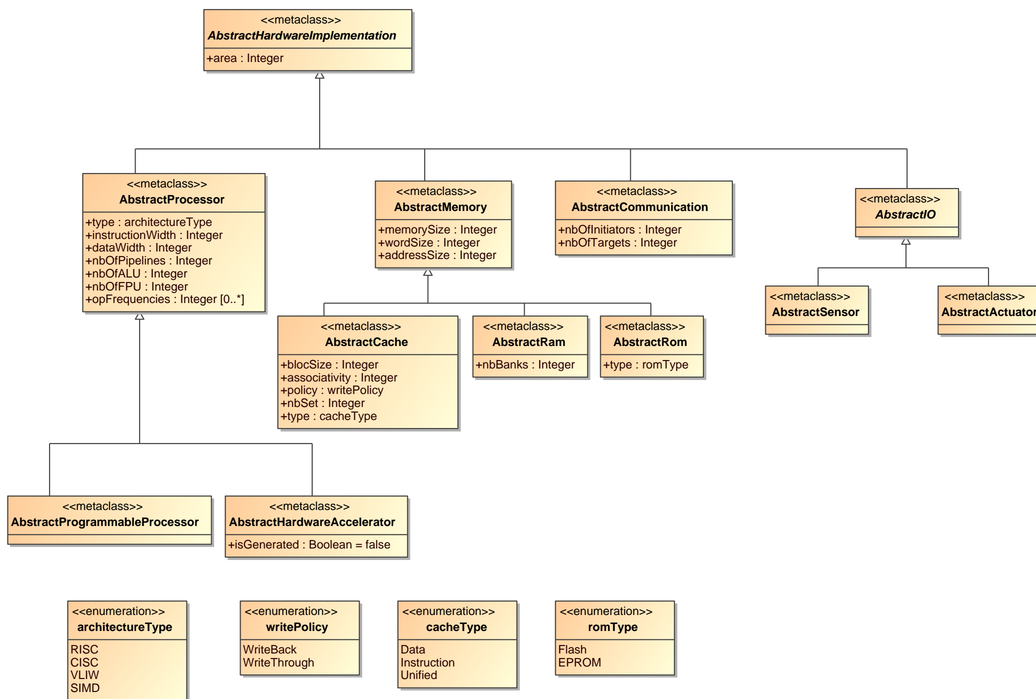


FIG. A.1: Vue globale de la partie matérielle du paquetage *Deployment*. Elle est décrite en détail dans la thèse de Rabie Ben Atitallah.

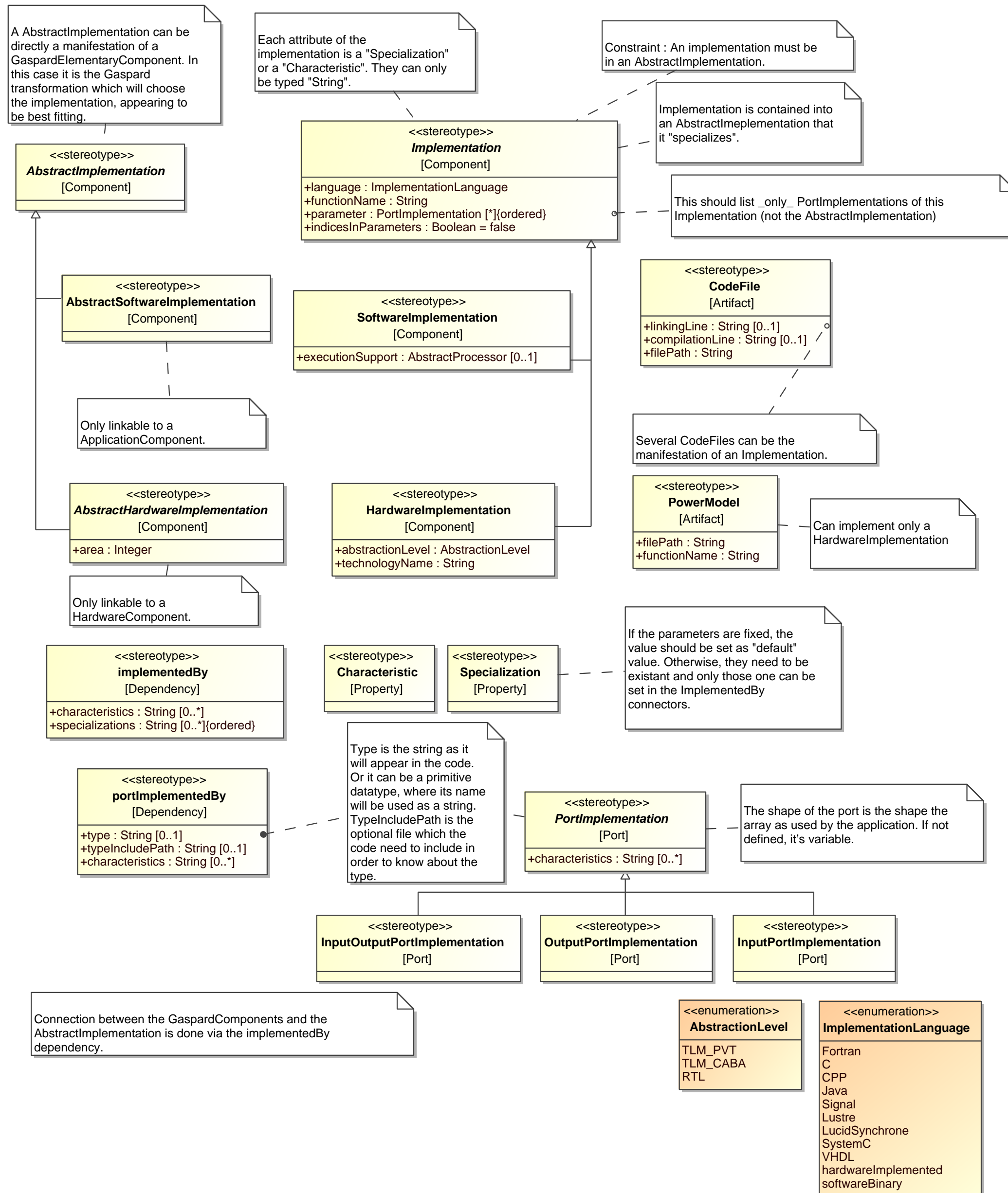


FIG. A.2: Vue globale du profil UML de déploiement. Il permet de représenter en UML les concepts du paquetage *Deployment* du méta-modèle *Gaspard* présenté à la section 3.2.

Ordonnancement de systèmes parallèles temps-réel
De la modélisation à la mise en œuvre par l'ingénierie dirigée par les modèles

Résumé : Les travaux de cette thèse s'inscrivent dans le cadre de l'ordonnancement de calculs à haute performance sur multi-processeurs à mémoire partagée. Ces travaux recouvrent deux parties distinctes.

Dans une première partie nous abordons le flot de conception de systèmes-sur-puce (SoC) multi-processeurs. Nous proposons des concepts supplémentaires pour la modélisation de ces systèmes en UML afin de compléter la sémantique propre au parallélisme, et de permettre le raffinement du modèle de SoC jusqu'à une spécification complète, en restant au plus haut niveau de conception. Par la suite nous présentons un nouveau niveau de simulation qui, en prenant en compte le particularisme du traitement intensif de données, abstrait le fonctionnement du SoC. En particulier, l'exécution et l'ordonnancement des tâches sont directement effectués sur la machine hôte. La simulation, implémentée en SystemC, permet de très rapidement évaluer les performances d'un placement de données et de tâches sur une architecture.

La compilation du modèle de SoC vers la simulation est faite à l'aide d'une chaîne de transformations de modèles. Nous détaillons les avantages apportés par l'usage de l'ingénierie dirigée par les modèles pour la conception de SoC. Enfin nous illustrons la mise en œuvre de l'ensemble du flot de conception développé au cours de la thèse avec un exemple d'exploration d'architecture.

Dans une seconde partie, nous proposons une approche permettant d'exploiter les systèmes multi-processeurs pour garantir les propriétés temps-réel d'applications parallèles. L'approche se base sur le fait que les tâches qui composent le système ont des priorités différentes, en particulier vis-à-vis du temps de réponse aux interruptions. L'introduction d'asymétrie parmi la liste d'ordonnancement des processeurs permet d'assurer de faibles latences pour les tâches à priorité temps-réel. Un équilibrage de charge adapté permet de maintenir toute la puissance potentielle des processeurs. Cette contribution est complétée d'une validation expérimentale basée sur une implémentation dans le noyau Linux.

Mots clefs : Parallélisme, multi-processeur, systèmes-sur-puce, systèmes temps-réel, traitement intensif de données, ingénierie dirigée par les modèles, UML, compilation, flot de conception.

Scheduling for real-time parallel systems
From modeling to implementation using model-driven engineering

Abstract: The works presented in this PhD thesis encompass the scheduling for high-performance computing on shared memory multi-processor systems. Those works cover two distinct parts.

In the first part, we treat the multi-processor systems-on-chip (SoC) conception flow. We propose additional concepts for modeling those systems in UML in order to complete the specific parallelism semantics, and to permit refining the SoC model up to a complete specification, while staying at the highest conception level. Later we present a new abstraction level for the simulation which, by taking into account the particularities of the intensive data processing, abstracts the SoC inner working. In particular, the execution and task scheduling are directly handled on the host computer. The simulation, implemented in SystemC, allows a very fast performance evaluation of a mapping of data and tasks on an hardware architecture.

The compilation of the SoC model towards the simulation is done through a model transformation chain. We detail the advantages brought by the usage of model-driven engineering for the SoC conception. Finally, we illustrate the implementation of the conception flow developed along the 3 years with an example of architecture exploration.

In the second part, we propose an approaching allowing the usage of multi-processor systems to guaranty real-time properties on parallel applications. The approach is based on the fact that the tasks composing the system have different priorities, in particular with respect to the interruption response time. The introduction of asymmetry inside the scheduling task queue of the processors permits to insure low latencies for the task with real-time priorities. The whole potential power of the processors is maintained thanks to an adapted load-balancing. This contribution is completed by an experimental validation based on an implementation on the Linux kernel.

Keywords: Parallelism, multi-processor, systems-on-chip, real-time systems, intensive signal processing, model-driven engineering, UML, compilation, design flow.