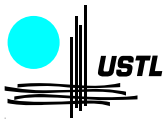
Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles

Sébastien Le Beux




Numéro d'ordre : 4105

Université des Sciences et Technologies de Lille

Thèse

présentée pour obtenir le grade de docteur
spécialité : Informatique
par

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 soutenue le 7 decembre 2007, devant la commission d'examen formée de :

El Mostapha Aboulhamid ...	Professeur Université de Montréal	**Rapporteur**
Tanguy Risset	Professeur INSA de Lyon	**Rapporteur**
Bernard Toursel	Professeur LIFL	**Examinateur**
Carlos Valderrama	Professeur Université de Mons	**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

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, El Mostapha Aboulhamid et Tanguy Risset ainsi que les examinateurs, Carlos Valderrama et Bernard Toursel, pour avoir lu et commenté ce manuscrit et aussi pour leur remarques, questions et recommandations lors de ma présentation.

Les travaux présentés font tous partie du projet Gaspard, et ils n'auraient pas pu avoir lieu sans le travail fournit par tous les autres contributeurs de ce projet. 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., Philippe D., Christophe, Julien T., Eric, Rabie, Huafeng, Imran, Anne, Myriam, Pierre, Frédéric, Abdoulaye, Calin, Antoine, Asma, Adolf, César, Lossan, Ouassila, Ashish, Arnaud et Karine. Je tiens d'autant plus à exprimer ma reconnaissance envers les relecteurs de ce manuscrit.

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

Table des matières

Introduction

Plus vite, plus haut, plus fort pourrait devenir la devise des applications de notre monde aujourd'hui. Détection radar, décodage audio et traitement vidéo sont des exemples de ces applications qui nous facilitent la vie et nous aident quotidiennement. Elles font partie du domaine d'application appelé *traitement de signal intensif* et sont caractérisées par un grand nombre de données traitées de façon régulière par des calculs répétitifs. En plus d'être gourmandes en puissance de calcul, ces applications sont souvent soumises à des contraintes de temps qu'il convient de respecter. Les performances qu'exige l'exécution de ces applications mettent à mal les systèmes informatiques et cela s'accentue avec les contraintes d'espace ou de consommation auxquelles ils sont soumis. Un exemple flagrant est le téléphone cellulaire qui doit rester compact et posséder une autonomie suffisante tout en offrant au consommateur les dernières innovations en matière de multimédia. Les automobiles ne sont pas en reste au niveau des contraintes d'espace avec l'intégration de systèmes électroniques complexes liés à la sécurité.

On parle de *système embarqué* lorsque des ressources informatiques sont soumises à de fortes contraintes d'intégration et de consommation d'énergie. Un système embarqué est composé d'unités de calcul, de mémoires pour le stockage de données, de moyens de communication, de périphériques, etc. Généralement, chacune de ces composantes est exploitée de manière à en tirer le bénéfice maximum pour l'exécution de l'application réalisée par le système embarqué. Avec l'évolution en puissance de calcul des applications, les systèmes embarqués sont amenés à accroître leur capacité de calcul, ce qui se traduit souvent par une parallélisation des applications et des ressources qui composent les systèmes embarqués. Le parallélisme augmente le nombre de calculs réalisés en même temps tout en limitant l'impact sur la consommation.

Parallèlement à la demande croissante en puissance de calcul et en miniaturisation, les innovations technologiques en matière de semiconducteurs garantissent une augmentation rapide du nombre de transistors que contiennent les puces. Cela permet la réalisation de systèmes complexes sur une seule puce, appelés systèmes sur puce ou System on Chip (SoC) en anglais. De par leur forte capacité d'intégration, les SoCs offrent de grandes économies en consommation d'énergie et en espace ainsi qu'un gain important en performance. Du fait de ces avantages, les SoCs sont donc souvent intégrés dans des systèmes embarqués afin de remplacer certaines ressources informatiques. Ces avantages ne vont pas sans des inconvénients majeurs que sont la complexité de conception des SoCs, le risque que le produit final ne corresponde pas à la spécification et le délai de production qui les rend parfois obsolètes avant même leur mise sur le marché. À ces problèmes, il convient d'ajouter, une fois de plus, la demande toujours croissante en puissance de calcul.

Nous identifions trois axes majeurs qui permettent de répondre aux problèmes rencontrés lors de l'utilisation de SoCs : les circuits reconfigurables, les accélérateurs matériels et

les méthodologies de conception.

1. Les circuits reconfigurables, telles des ardoises magiques, peuvent être reconfigurés à volonté. La couche reconfigurable positionnée entre les transistors et les ressources fonctionnelles offre de la flexibilité en contrepartie de performances moindres. L'intégration de circuits reconfigurables dans les SoCs laisse une marge d'erreur aux concepteurs en permettant l'ajout de nouvelles fonctionnalités après fabrication du SoC, la correction de ressources qui contiennent des erreurs, la prise en compte de standards non existants lors de la spécification, etc.
2. Un accélérateur matériel est un circuit spécialisé pour le *traitement du signal systématique*[1]. Il permet une parallélisation maximale des calculs nécessaires à l'exécution d'une application, il fournit par conséquent un support d'exécution optimal pour le traitement de tâches régulières et répétitives.
3. Les méthodologies de conception sont en quelque sorte le lien entre un SoC et son concepteur. Elles établissent les limites de compréhension du SoC que son concepteur a et, en définitive, la productivité de concepteur. Un exemple de méthodologie utilisée depuis quelques années déjà est la réutilisation d'IPs pour les ressources fonctionnelles d'un SoC.

Problématique

Nous identifions plusieurs problèmes liés à l'utilisation des solutions que sont les accélérateurs matériels et les circuits reconfigurables.

Les accélérateurs matériels sont en règle générale dédiés à l'exécution d'une application donnée. Par conséquent, chaque accélérateur doit être conçu de façon unique, entrainant un coût de conception élevé et un long délai de production. S'ajoute à ces inconvénients la multiplication des interventions humaines et, par conséquent, des risques d'erreurs dans le processus de conception. Ces risques sont plus élevés encore lorsque qu'un accélérateur est conçu dans un langage *HDL* au niveau de description *RTL*. C'est traditionnellement à ce niveau de description que sont conçus les circuits électroniques.

La complexité toujours croissante des accélérateurs et les contraintes de temps ont favorisé le développement d'outils censés faciliter leur conception. Majoritairement, l'objectif principal de ces outils est la génération automatique d'accélérateurs dans un langage HDL à partir de descriptions qualifiées d'*abstraites*, c'est-à-dire sans les détails d'implémentations du niveau de description RTL. On parle alors de *synthèse de haut niveau*. Il s'avère cependant que ces niveaux de description ne se révèlent pas suffisament élevés limitant ainsi les utilisations potentielles du code produit (simulations, vérifications, exécutions).

Par ailleurs, ces outils ne tirent pas suffisamment profit du parallélisme potentiel de l'application et des diverses possibilités d'exécution de ce parallélisme. La cause principale est l'inadaptation des langages de description des applications, la conséquence est une performance relative des accélérateurs générés.

En outre, les outils de génération d'accélérateurs sont généralement découplés des placements sur les architectures reconfigurables, pourtant excellents supports d'implémentation des accélérateurs. L'une des raisons est le manque de standardisation dans la représentation

[1] Le traitement du signal systématique est un sous-ensemble du traitement du signal intensif, il ne prend en charge que les calculs réguliers.

des architectures reconfigurables ainsi que le manque d'homogénéisation dans la représentation des placements.

La méthode de travail adoptée pour la résolution de chacun de ces points est primordiale. En effet, les problèmes rencontrés, de la spécification initiale des besoins jusqu'à l'aboutissement d'un projet, sont souvent liés à des aspects méthodologiques et aux difficultés de compréhension entre un client et un concepteur. La spécification initiale peut se résumer en un ensemble d'idées et de besoins liés à des habitudes ou à des opinions parfois mal conceptualisées par un client. L'aboutissement du projet dépend de l'interprétation par le concepteur de ces spécifications initiales et de sa capacité à capitaliser ou à réutiliser le travail déjà réalisé. Cette capacité de communication, de capitalisation et de réutilisation nécessite des formats standard et des outils adaptés à l'échange de l'information.

L'environnement Gaspard du projet INRIA DaRT

Nos travaux s'inscrivent dans le cadre du projet DaRT qui tend à développer un environnement de co-conception pour SoCs. Les applications traitées relèvent du traitement de signal intensif et nécessitent un grand nombre de calculs réalisés de façon régulière sur des tableaux de données multidimensionnels. Gaspard permet la représentation dans un cadre unifié de l'application, de l'architecture fonctionnelle, du placement de la première sur la seconde et du déploiement des IPs. Cette représentation est réalisée à un haut niveau d'abstraction, c'est-à-dire indépendamment des détails d'implémentation. À partir d'une telle représentation, différentes plateformes d'exécution sont ciblées par un processus de raffinement, dont la dernière étape correspond à la génération de code.

Cette démarche de modélisation, de raffinement, de génération de code s'inscrit dans un cadre méthodologique et technologique guidé par l'*Ingénierie Dirigée par les Modèles* (IDM). L'IDM favorise l'homogénéisation et l'unification des processus qui interviennent dans la conception de SoCs tout en permettant de travailler de façon indépendante sur les différents aspects des systèmes. L'application et l'architecture du SoC peuvent ainsi être développées de façon indépendante par les différentes équipes en charge du projet, tout en restant dans un même cadre de travail. L'utilisation de langages unifiés permet l'échange des informations et des travaux relatifs à la conception du SoC et cela indépendamment du domaine d'expertise des concepteurs.

Contributions

L'utilisation des accélérateurs matériels dans l'environnement de co-conception Gaspard est toutefois limitée à l'utilisation d'IPs existants en bibliothèques. Cela restreint fortement les possibilités d'exploitation de ces accélérateurs qui sont spécifiques à chaque application. Notre objectif est de concevoir ces accélérateurs à partir d'applications modélisées à un haut niveau d'abstraction, que nous appelons les applications Gaspard et peut se résumer par le développement d' « *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles* ».

Pour construire ce flot de conception, nous avons défini :

1. Une *exécution matérielle* qui spécifie le comportement des applications modélisées à un haut niveau d'abstraction lorsqu'elles sont exécutées par des accélérateurs matériels.

La spécification à un haut niveau d'abstraction de ces applications repose sur une représentation factorisée du parallélisme de données et sur des dépendances de données entre des tableaux multidimensionnels. Nous détaillons les conséquences de ce parallélisme dans la définition de l'exécution matérielle que nous proposons.

2. Un *métamodèle RTL* qui formalise selon les principes de l'IDM les concepts manipulés dans l'exécution matérielle des applications que nous venons de citer dans le paragraphe précédent. Une instance du métamodèle RTL, appelé *modèle*, correspond à la description d'un accélérateur matériel donné qui permet l'exécution d'une application Gaspard donnée. Le métamodèle RTL permet par ailleurs la description, dans un même cadre, des architectures reconfigurables telles les FPGAs et du placement des accélérateurs sur FPGA.
3. Une *transformation de modèles*, assimilée à la compilation, qui permet de générer un accélérateur depuis la modélisation à un haut niveau d'abstraction d'une application Gaspard. Cette transformation de modèles ajoute, entre autres, des informations d'implémentation relatives à l'exécution matérielle de l'application : elle *raffine* la modélisation de l'application.
4. Une *optimisation des accélérateurs* qui, à partir des informations de placement d'un accélérateur sur FPGA, modifie l'application modélisée à un haut niveau d'abstraction de manière à adapter l'accélérateur au FPGA.
5. Une *génération de code VHDL* depuis le métamodèle RTL afin de réutiliser les outils de synthèse FPGA et qui correspond, en IDM, à une *transformation modèle vers texte*. Parallèlement à cette génération de code VHDL, un script peut être généré si un placement sur FPGA est spécifié dans le modèle RTL.

Bien plus qu'un ensemble de contributions, c'est bien un flot de conception bâti à l'aide de l'IDM et articulé autour de chacune de ces contributions que nous proposons. Nous faisons ainsi face aux défis lancés par les applications et leurs continuelles exigences en réduisant les temps de conception (*plus vite*) d'accélérateurs matériels fortement parallélisés (*plus fort*) et générés depuis des hauts niveaux d'abstraction (*plus haut*).

Plan du document

Nos travaux fournissent des éléments de réponses aux différentes interrogations suivantes : Comment s'exécute matériellement une application modélisée à un haut niveau d'abstraction dans l'environnement Gaspard ? Comment générer des accélérateurs depuis ce haut niveau d'abstraction ? Comment optimiser les accélérateurs pour un FPGA donné ? Comment générer du code synthétisable pour ces accélérateurs ? Nous proposons des réponses à ces questions dans ce document dont le plan est le suivant :

Chapitre 1 : État de l'art Ce premier chapitre présente le contexte de nos travaux qui recouvrent l'accélération matérielle pour les applications de traitement de signal intensif et l'outillage proposé pour leur conception.

Chapitre 2 : Un flot de conception IDM permettant l'exécution matérielle d'applications Gaspard À partir du contexte étudié, nous dégageons la problématique, positionnons nos travaux et donnons les grandes lignes de nos contributions.

Chapitre 3 : Le métamodèle RTL pour la description des accélérateurs matériels Nous présentons l'exécution matérielle des applications Gaspard qui spécifie le comportement des accélérateurs matériels pour l'exécution de ces applications. Les concepts manipulés par les accélérateurs servent de base à la construction du métamodèle RTL qui permet par ailleurs la métamodélisation des FPGAs et du placement des accélérateurs sur ces FPGAs. Le métamodèle RTL est au cœur de nos contributions, il est le pivot de notre flot de conception.

Chapitre 4 : Génération de code depuis le métamodèle RTL Les concepts manipulés dans le métamodèle RTL sont associés à des éléments de syntaxe afin de permettre une génération de code. Nous présentons dans ce chapitre la génération de code VHDL synthétisable pour les accélérateurs et la génération de scripts de placements pour un outil de synthèse. Ces codes permettent l'implémentation des accélérateurs sur FPGA.

Chapitre 5 : Transformation de modèles vers le métamodèle RTL La compilation d'une application modélisée à un haut niveau d'abstraction et indépendamment de sa cible d'exécution en un accélérateur matériel est présentée dans ce chapitre. Cette étape, appelée *transformation de modèles* en IDM, est décomposée en règles de transformation que nous détaillons pour la compilation du parallélisme de données.

Chapitre 6 : Optimisation de l'implémentation des accélérateurs sur FPGA Ce chapitre présente un processus d'optimisation qui génère un accélérateur matériel adapté au FPGA sur lequel il est implémenté. Pour cela, nous proposons une heuristique d'optimisation itérative qui, à partir d'un accélérateur généré dans le métamodèle RTL, modifie la description à un haut niveau d'abstraction. Les principaux avantages de ce processus d'optimisation sont qu'il réutilise des fonctions de transformation de boucles existantes et qu'il s'opère avant la génération de code.

Chapitre 7 : Utilisation de notre flot pour la conception d'un système radar anti-collision Cette dernière contribution porte sur l'utilisation de notre flot de conception pour la sécurité dans le transport routier. Nous introduisons un système embarqué qui vise à prévenir les collisions et détaillons la modélisation à un haut niveau d'abstraction du sous-ensemble de ce système composé de calculs réguliers et systématiques. Nous présentons les résultats de l'implémentation sur FPGA de l'accélérateur généré depuis ce haut niveau d'abstraction. Cette étude de cas valide le bon fonctionnement de notre flot de conception.

Conclusion Nous concluons par le bilan des travaux effectués et détaillons les contributions apportées avant d'aborder quelques perspectives à nos travaux.

Chapitre 1

État de l'art

Contents

Ce premier chapitre familiarise le lecteur avec le contexte d'étude de nos travaux. La figure 1.1 illustre dans les ellipses chacun des points que nous abordons. Les relations entre ces points sont représentées par des flèches en traits pleins. La flèche en pointillés représente le cheminement que nous suivons tout au long de ce chapitre et qui nous conduit en dernier lieu à cette thèse (que nous positionnons dans le chapitre suivant).

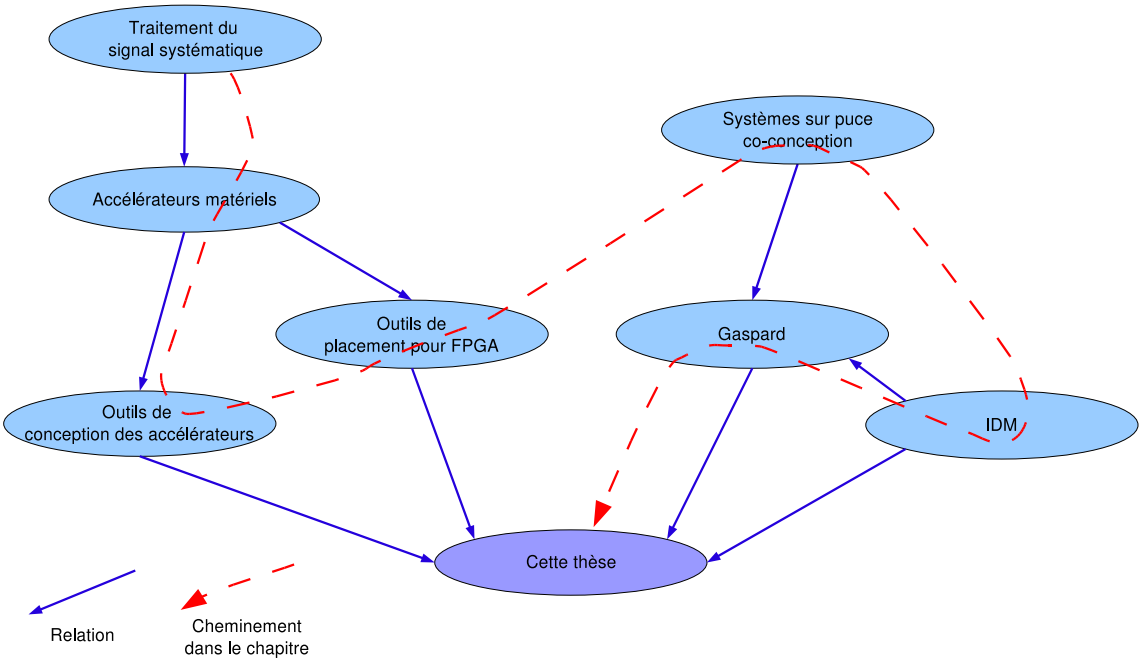


FIG. 1.1: Organisation du chapitre.

Nous débutons ce chapitre par une introduction au *traitement du signal systématique* et à ses spécificités en terme notamment de dépendances de données. Les applications de traitement du signal systématique peuvent tirer profit d'exécutions sur *accélérateurs matériels* car il est possible de personnaliser ces derniers en conséquence. Nous présentons alors des *outils de conception et des modèles de calcul* qui permettent de générer de tels accélérateurs (nous présentons différents outils dans la section 1.2 et accordons la section 1.3 entière au modèle de calcul ARRAY-OL). Les technologies de réalisation des accélérateurs sont alors présentées, nous abordons en particulier les architectures reconfigurables et les *outils de placement pour FPGA*.

Orthogonalement à ce premier contexte d'étude, les *systèmes sur puce* et la notion de *co-conception* sont introduits, nous mettons l'accent sur les défis lancés à ces outils de co-conception. Nous verrons alors que l'*Ingénierie Dirigée par les Modèles* (*IDM*) répond à certains de ces défis, mais constaterons le peu de travaux relatifs à l'IDM et à la génération d'accélérateurs matériels. L'environnement de co-conception *Gaspard* est introduit en dernier lieu, il est l'environnement dans lequel s'intègre nos travaux.

Au terme de ce chapitre, nous serons alors en mesure d'introduire « *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles* » qui est intégré dans l'environnement de co-conception Gaspard.

1.1 Traitement du signal systématique

Un *signal* est assimilé à un support véhiculant de l'information. Le *traitement du signal* (TS) est l'art de manipuler un signal afin d'en extraire cette information. Le *traitement du signal intensif* (TSI) est le sous-ensemble du traitement du signal le plus gourmand en calcul, dont la partie la plus régulière est le *traitement du signal systématique* (TSS). Ce dernier consiste à réaliser des calculs sur les signaux indépendamment de leur valeur dans le but d'en extraire des propriétés intéressantes (par exemple pour détecter la présence d'un obstacle dans le contexte d'un radar anti-collision ou de chercher une forme bien précise dans une image). Cette partie TSS précède, dans le temps, une étape plus irrégulière et dépendante de la valeur des données, le *traitement de données intensif* (TDI). Cette étape analyse le signal issu du traitement systématique et prend éventuellement des décisions (freinage d'urgence lors de la détection d'un obstacle ou analyse de façon plus détaillée une forme détectée dans une image).

Dans ces travaux, nous nous intéressons au traitement du signal systématique dont le positionnement dans le traitement du signal est représenté à la figure 1.2.

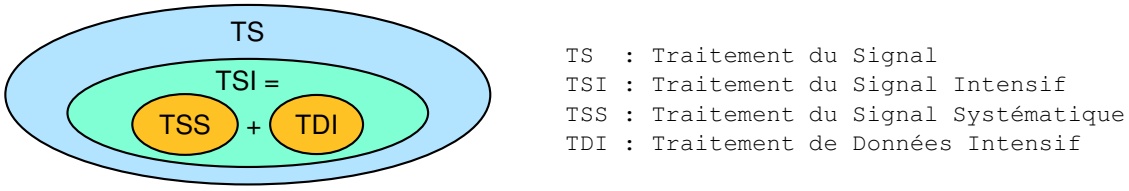


FIG. 1.2: Positionnement du TSS dans le domaine du TS.

1.1.1 Exemples d'applications

La décomposition du TSI en TSS et en TDI se retrouve dans de nombreuses applications :

- **Radar anti-collisions :** la prévention des collisions dans un système nécessite l'utilisation d'une antenne. Cette antenne renvoie un signal sous la forme d'un flux de données contenant des informations relatives à la présence d'obstacles, un écho. Ces informations sont masquées par du bruit (des parasites) et sont étalées sur le flux de données temporel. Une phase de pré-traitement qui relève du traitement du signal systématique, filtre ce signal de manière à en faire ressortir les caractéristiques intéressantes (la présence d'obstacles). L'étape suivante est plus irrégulière et consiste à analyser ce nouveau flux de données de manière à valider la présence d'un obstacle, à déclencher un freinage d'urgence où à le poursuivre si nécessaire [72]. Cette application est illustrée à la figure 1.3 ;
- **Traitement sonar :** l'analyse d'un environnement maritime est similaire à celui d'un environnement routier, mais nécessite toutefois une vision plus large de la scène. Dans le cas d'un sous-marin, ce n'est pas une antenne qui est utilisée mais plusieurs hydrophones répartis autour de ce sous-marin. La première phase de traitement est systématique et est constituée d'une FFT (qui ajoute une dimension fréquentielle aux données traitées). Le reste du traitement est dédié à la détection d'objets et à leur poursuite au cours du temps ;

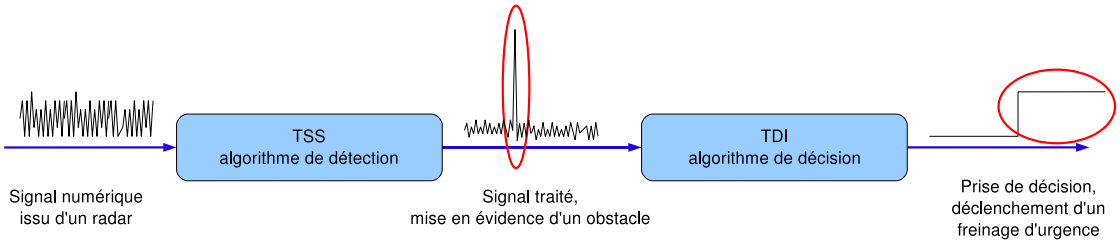


FIG. 1.3: Représentation de l'application radar anti-collision. La phase de traitement du signal systématique précède celle du traitement de données intensif.

– **Convertisseur** $16/9$ **-** $4/3$ **:** le passage d'un format télévisuel à un autre est décomposé en deux phases [79]. La première étape consiste à créer des pixels à partir du signal vidéo au format $16/9$ via une interpolation, il en résulte un nouveau signal vidéo. La seconde étape supprime certains pixels de ce signal de manière à ne conserver qu'une partie de l'information, cela permet le passage au format $4/3$. Cette application ne considère à aucun moment la valeur des pixels, il s'agit donc de traitement du signal systématique ;

D'autres exemples encore illustrent l'utilisation du TSS dans les applications de TSI : encodeur/décodeur JPEG-2000, récepteur de radio numérique, etc. Chacune de ces applications réalise des traitements réguliers sur les données, certaines exploitent le résultat de ces traitements pour prendre des décisions. Nous pouvons par ailleurs remarquer la diversité de la sémantique portée par les dimensions des tableaux de données manipulés par ces applications (temporel, spatial et fréquentiel) et leur nombre. Il s'agit là d'une caractéristique très spécifique aux applications de traitement du signal intensif qu'est la manipulation de tableaux *multidimensionnels*.

1.1.2 Spécificités du TSS

Chacune de ces applications montre que le traitement du signal systématique offre diverses caractéristiques intéressantes du point de vue des données et des calculs : la structuration en tableaux des données, la modification au sein des calculs de la taille des tableaux et du nombre de leur dimension, la présence de dimensions cycliques ou de taille infinie (pour représenter le temps par exemple) dans ces tableaux. Les calculs sont effectués indépendamment des valeurs des données et sont souvent les mêmes d'une application à une autre (produits scalaires, transformés de Fourier (FFT), etc.).

Ce bilan des caractéristiques du traitement du signal systématique montre que ce ne sont pas les calculs qui importent, mais que ce sont bien les interactions entre ces calculs (les dépendances de données) qui sont importantes. En terme de réalisation de l'exécution de ces applications, cela se traduit souvent par l'utilisation de *circuits spécialisés* ou d'*accélérateurs matériels* dont les chemins de données peuvent être personnalisés pour les interactions entre les calculs.

1.2 Les accélérateurs matériels pour l'exécution d'applications de traitement du signal intensif

Les possibilités de conception des accélérateurs sont infinies, ils offrent l'opportunité de personnaliser l'exécution d'une application dans un objectif de performance, de dimension, de consommation, etc. Pour ces raisons, les accélérateurs sont couramment utilisés pour les phases de traitement qui nécessitent un grand nombre de calculs réguliers.

1.2.1 Conception des accélérateurs au niveau RTL

La méthode traditionnelle consiste à décrire les accélérateurs au niveau RTL (Register Transfer Level) dans un langage de description matérielle, Hardware Description Language (HDL), comme VHDL [57] ou Verilog [118]. De nombreux travaux présentent de telles réalisations, elles sont souvent associées à des optimisations qui permettent d'augmenter les performances de l'accélérateur. Dans le cadre plus précis de l'utilisation d'accélérateurs matériels dans les systèmes composés de radars, Peled et Liu [95] ont proposé dans les années 1970 de placer dans une mémoire ROM les résultats possibles de la multiplication de deux nombres, évitant ainsi l'utilisation de multiplieurs coûteux en silicium . Plus récemment, Tessier et Burleson [117] dressaient en 1999 un état de l'art sur les techniques de conception des accélérateurs pour la réalisation de ces filtres.

La conception d'accélérateurs matériels au niveau RTL est toujours d'actualité, à l'image de Cardoso et Corte-Real [24] qui optimisent en 2005 un multiplieur accumulateur pour une application de détection de mouvement implémentée sur plusieurs circuits reconfigurables. Par ailleurs, nous avons proposé en 2006 [70] une implémentation dédiée d'un filtre utilisé dans le cadre d'un système anti-collision et qui nécessite une grande puissance de calcul. Nos travaux ont permis l'implémentation sur un FPGA donné de ce filtre, disponible actuellement sous la forme d'un IP.

La conception des accélérateurs au niveau RTL est liée à des risques d'erreurs car elle repose sur l'expertise d'un concepteur. De plus, les possibilités d'exploration d'architecture d'un accélérateur sont potentiellement freinées par une structure rigide définie par un concepteur. Les temps de conception sont longs car il est nécessaire de valider la bonne fonctionnalité de chaque sous-ensemble de l'accélérateur, puis de valider le fonctionnement de l'association de ces sous-ensembles, etc. Malgré ces difficultés de conception, les accélérateurs restent couramment utilisés dans le domaine du traitement de signal intensif. Cependant, plutôt que d'être directement décrit au niveau RTL, les accélérateurs sont conçus à l'aide d'outils.

1.2.2 Conception des accélérateurs assistée par des outils

Les possibilités de conception et la délicatesse des conceptions manuelles ont fait naître des outils qui tendent à générer des accélérateurs pour les applications de traitement du signal intensif. Nous présentons certains de ces outils dans cette section.

1.2.2.1 Environnement MMALPHA

L'environnement MMALPHA[1] est une interface pour le logiciel Mathematica[2] à partir duquel il est possible de manipuler des programmes dans le langage ALPHA.

Langage ALPHA ALPHA [100, 105, 75] est un langage à parallélisme de données et fonctionnel créé par l'équipe API à l'IRISA de Rennes. La première définition de ce langage a été proposée par Mauras en 1989 [82] et est fondée sur le formalisme des systèmes d'équations récurrentes [65], mais le langage ALPHA à évolué depuis. ALPHA est utilisé dans le cadre d'un outil pour la synthèse d'architectures VLSI[3] systoliques[4]. C'est un langage à parallélisme de données permettant la description de haut niveau d'algorithmes de calcul réguliers. Le formalisme qui sous-tend ALPHA est le *modèle polyédrique*. Celui-ci est aujourd'hui à la base de plusieurs méthodes de parallélisation automatique des boucles et de la synthèse de réseaux systoliques.

Programmation en ALPHA Il n'y a pas de notion de temps dans ALPHA : un programme décrit un ensemble de calculs dont les dépendances de données expriment l'ordre dans lequel ces calculs peuvent être réalisés. Cela implique qu'ALPHA soit un langage à *assignation unique* et qu'il existe potentiellement plusieurs ordonnancements pour l'exécution d'un programme. Un programme ALPHA est un *système*. Les variables sont des tableaux généralisés de forme quelconque. L'ensemble des indices du tableau est appelé *domaine* de la variable, il spécifie la forme de ce tableau. Les données manipulées sont multidimensionnelles et correspondent à des unions de polyèdres convexes. Par exemple le code suivant :

$$V : i, j | 1 \leq i \leq j; j \leq 4 \; of \; real;$$

déclare une variable V de type réel. Le domaine de cette variable est l'ensemble des points (i, j) dans le triangle suivant :

$$\begin{array}{cccc} V_{1,1} & V_{1,2} & V_{1,3} & V_{1,4} \\ & V_{2,2} & V_{2,3} & V_{2,4} \\ & & V_{3,3} & V_{3,4} \\ & & & V_{4,4} \end{array}$$

Langages ALPHA0et ALPHARD La génération des accélérateurs au niveau RTL est réalisée par le biais des langages ALPHA0 [37] et ALPHARD [74] qui sont des sous-ensembles de ALPHA. Un programme ALPHA0 est généré depuis un programme ALPHA. Il décrit le fonctionnement de l'architecture dans le temps (l'indice qui représente le temps est identifié, t par exemple). Pour la réalisation de cette architecture, chaque équation du système ALPHA est interprétée de manière à lui faire correspondre un élément architectural (connexion simple, registre, opérateur, etc.). Ainsi, l'équation ALPHA suivante :

$$A[t, p] = B[t, p-1] + C[t-1, p]$$

[1] http://www.irisa.fr/cosi/ALPHA/

[2] http://www.wolfram.com

[3] Very Large Scale Integration : circuits intégrés à forte densité d'intégration.

[4] Réseau de processeurs identiques connectés localement, seuls les processeurs sur les bords de l'architecture communiquent avec l'extérieur.

est interprétée par : la variable A du processeur[5] p à l'instant t prend la valeur de la variable B du processeur $p-1$ à l'instant t additionnée à la variable C du processeur p à l'instant $t-1$. En ALPHA0, cette équation est équivalente à un registre (pour le décalage temporel de la variable C), un connecteur simple (pour la connexion entre les processeurs p et $p-1$) et un additionneur.

Un programme ALPHARD est généré depuis un programme ALPHA0. Il décrit la même architecture que celle décrite dans le programme ALPHA0 dont il provient, mais introduit la structure du circuit. Chaque sous-système devient alors un composant de type *cellule* (pour l'exécution d'un calcul), *contrôleur* (pour le contrôle du comportement des cellules) ou *module* (hiérarchie utilisée pour l'assemblage de cellules). Un programme ALPHARD est suffisamment proche du niveau RTL du circuit pour en générer le code VHDL, ce qui permet de le synthétiser sur FPGA par exemple.

Bilan de ALPHA ALPHA est en mesure d'exprimer de façon compacte des formes de données complexes (un triangle par exemple). Cependant, en traitement du signal, les données sont souvent manipulées sous la forme de tableaux « simples » , la puissance de l'expression des formes de données dans ALPHA est donc rarement nécessaire. Par ailleurs, ALPHA ne gère pas les accès cycliques dans les tableaux de données, accès pourtant indispensables, par exemple, pour l'application du traitement sonar introduit précédemment. En outre, les programmes ALPHA manipulent des indices pour les accès aux données, ce qui implique que le programmeur calcule ces indices. Cela augmente considérablement le risque d'erreurs quant à l'expression des dépendances de données sur des tableaux multidimensionnels. Concernant la définition d'un équivalent RTL d'un programme ALPHA, ALPHARD ne prend pas en considération la quantité de ressources nécessaires à l'implémentation du circuit. Il est donc nécessaire de passer par un outil commercial pour obtenir un aperçu de la surface occupée sur FPGA par exemple.

CLOOG et l'extension CLOOGVHDL Bastoul [10] propose le plugin CLOOG (Chunky Loop Generator) pour la librairie de fonctions polyédriques POLYLIB [98]. CLOOG fournit un logiciel et une bibliothèque qui génère des boucles pour parcourir les points entiers d'un polyèdre, il est en mesure d'en générer le code C ou Fortran par exemple. CLOOG est actuellement étendu par Devos [36, 35] et l'extension de CLOOGVHDL qui permet la génération de code VHDL pour l'exécution matérielle de ces boucles.

Cependant, les problèmes rencontrés lors de la programmation avec ces logiciels restent les mêmes que ceux rencontrés avec ALPHA.

1.2.2.2 SDF : Synchronous Data Flow

SDF [76, 78] est un modèle de calcul à *flot de données* permettant la description d'application du traitement de signal. En SDF, les calculs d'une application sont représentés par des nœuds (*actor*) et les données (*tokens*) par des arcs. Une application décrite sous la forme d'un graphe acyclique orienté dont chaque nœud consomme et produit des données. La figure 1.4 illustre une application en SDF[6]. Les symboles associés aux entrées et aux sorties de chaque nœud (ici A, B, C et D) spécifient la quantité de jetons consommés ou produits par l'exécution d'un nœud.

[5] Les processeurs générés dans ALPHA peuvent être interprétés comme des unités de calcul d'un accélérateur.
[6] SDF est supporté par l'environnement PTOLEMY [77] qui en permet la représentation et la simulation.

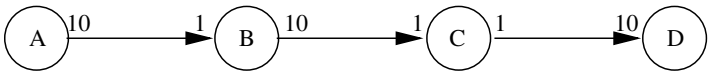


FIG. 1.4: Un graphe de tâches en SDF.

La représentation des dépendances de données en SDF est relativement intuitive, il est donc aisé d'appréhender et de concevoir une application. Cependant, SDF est limité à l'expression des dépendances de données sur une seule dimension, ce qui restreint considérablement le nombre d'applications modélisables. Pour pallier à ce défaut majeur, Edward Lee proposa en 2002 [88] l'extension MDSDF (MultiDimensional Synchronous Dataflow) qui permet de manipuler des flots de données multidimensionnelles et autorise ainsi la représentation de certaines applications de traitement d'images ou de traitement vidéo. La différence entre une représentation MDSDF et SDF provient des symboles qui sont, dans MDSDF, associés à chacune des dimensions des tableaux de données.

La figure 1.5 représente une tâche A qui produit un tableau bidimensionnel composé de 40×48 données. La tâche B découpe ce tableau par bloc de 8×8. Pour chaque exécution de la tâche A, la tâche B sera exécutée 5×6. En SDF, cette application peut être représentée par le biais d'une linéarisation des tableaux.

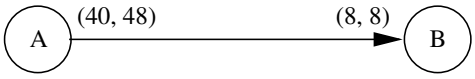


FIG. 1.5: Un graphe d'application en MDSDF avec un flot de données bidimensionnel.

Comparé à SDF, MDSDF permet la représentation d'applications qui manipulent des flots de données multidimensionnels. Il existe cependant des contraintes sur le nombre de dimensions qui ne peut évoluer au cours de l'application (il n'est donc pas possible de représenter la dimension fréquentielle créée par une FFT). Une seconde contrainte concerne la consommation et la production de données nécessairement parallèle aux axes. Afin de résoudre ce problème, Murthy et Lee [87] ont proposé l'extension GMDSDF (Generalized MultiDimensional Synchronous Dataflow), plus riche en expression que MDSDF. Cependant, la spécification des dépendances de données non parallèles aux axes ne peut s'appliquer sur des tableaux possédant plus de deux dimensions.

Des travaux menés par Williamson en 1998 [124] ont permis la génération de code VHDL depuis le modèle de calcul SDF. Cependant, ce modèle de calcul est le plus restrictif des trois (SDF, MDSDF et GMDSDF) car il est possible d'y manipuler qu'une seule dimension. Or, la gestion de tableaux multidimensionnels est indispensable pour le traitement du signal systématique.

En 2006, Filiba *et al.* [48] génère du code VHDL (depuis l'environnement PTOLEMY) de tâches applicatives (additionneurs, soustracteur, etc.). Cependant, la génération de code n'est pas fonctionnelle pour les flots de données, il n'est donc pas possible de réaliser matériellement les dépendances de données.

1.2.2.3 SYNDEX et SYNDEX-IC

SYNDEX [109] implémente la méthodologie « Adéquation Algorithme Architecture » et permet donc la représentation :

– d'applications sous la forme d'un graphe dirigé acyclique dans lequel les nœuds correspondent à des calculs et les arcs à des dépendances de données ;
– d'architectures multiprocesseurs hétérogènes et des interconnexions par des bus, des connexions point à point, etc. ;
– des caractéristiques de placement d'une application par rapport à une architecture (ou l'inverse) sous la forme de temps d'exécution de calculs ou temps de transfert de données par exemple (des contraintes de temps ou d'ordonnancement peuvent aussi être spécifiées).

Toutes ces informations servent à l'exploration des implémentations possibles d'une application donnée sur une architecture donnée en terme de placement et d'ordonnancement des différents calculs. Cette exploration s'appuie sur des heuristiques d'optimisation et il est possible de visualiser le résultat sous une forme graphique, ce qui en facilite l'analyse.

Les architectures supportées par SYNDEX sont des architectures de type multiprocesseurs hétérogènes. Afin d'ouvrir la voie à la génération d'accélérateurs matériels, des travaux ont permis le développement de SYNDEX-IC [63, 99].

SYNDEX-IC : une extension pour la génération d'accélérateurs matériels SYNDEX-IC [63] permet, à l'image de SYNDEX, la représentation d'applications sous la forme d'un graphe de tâches. Ce graphe peut être *factorisé* de manière à exprimer une répétition sur une tâche : il exprime son parallélisme de données potentiel. Les *frontières* de données avec les tâches factorisées sont décrites par des nœuds de factorisation (*factorization node*) *Fork*, *Join*, *Diffusion* et *Iterate*.

La figure 1.6 représente la factorisation par l'utilisateur d'un produit scalaire [64] à l'aide des nœuds de factorisation *Fork* et *Iterate*. Dans la représentation factorisée, des indices permettent de reconstruire les dépendances de données.

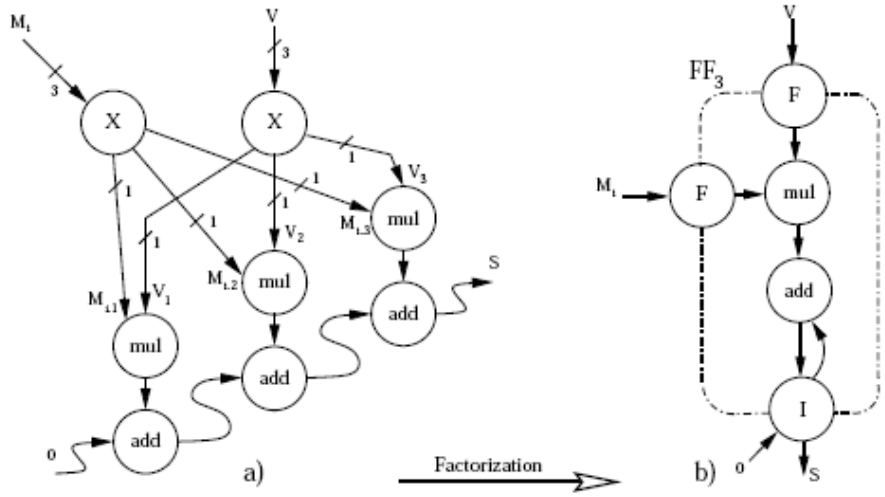


FIG. 1.6: Factorisation d'un graphe de tâches dans SYNDEX-IC et expression des dépendances de données sous la forme d'indices.

Optimisations L'heuristique utilisée pour l'optimisation tend à générer des circuits qui remplissent les contraintes temps réel de l'application tout en respectant la quantité de ressources allouée pour sa mise en œuvre (en nombre de CLB pour un FPGA par exemple). Pour cela, Dias *et al.* [38] introduisent la notion de défactorisation (*defactorization*) qui consiste à dérouler plus ou moins les boucles contenues dans un graphe factorisé. Plus le graphe est défactorisé, plus les latences à l'exécution sont faibles et plus les ressources consommées augmentent. Il s'agit donc de dérouler les boucles de manière à remplir les contraintes de temps d'exécution tout en essayant de minimiser la quantité de ressources nécessaire à l'implémentation du circuit à l'issu du processus d'optimisation.

Cette optimisation est intéressante car elle prend en considérations les ressources allouées sur FPGA. Cependant, seule la défactorisation est automatisée, il n'est donc pas possible de produire un circuit « plus petit » par le biais d'une automatisation de la factorisation par exemple.

Génération de code VHDL Afin de générer le code VHDL de l'accélérateur, des circuits matériels sont associés aux nœuds *Fork*, *Join* et *Iterate* du graphe de tâches et sont instanciés lors de la synthèse du chemin de données. La synthèse du chemin de contrôle ajoute des éléments de contrôle dans le chemin de données de manière à gérer les transferts de données entre les différentes tâches. De la même façon que pour le chemin de données, un équivalent matériel des frontières est proposé. Dans le cas de l'opérateur *Fork*, la frontière exprimée par cet opérateur est implémentée sous la forme d'un contrôleur qui gère les flux de données entrant et sortant de cette frontière.

Au final, le circuit généré possède plusieurs contrôleurs localisés (situés sur les frontières dans l'application) plutôt qu'un contrôleur centralisé. Cela permet de réduire la complexité du routage des signaux de contrôle, car les contrôleurs peuvent être placés à proximité de l'élément contrôlé.

Bilan de SYNDEX-IC SYNDEX-IC fournit une chaîne complète de génération d'accélérateurs matériels pour les applications de traitement de signal intensif. La représentation graphique des applications permet de mettre évidence aussi bien les dépendances de données « classiques » (sans parallélisme) que celles liées au parallélisme de données. Les frontières factorisées expriment sous une forme factorisée ce parallélisme et permettent le découpage d'un tableau en sous-tableaux par des indices. L'expression des dépendances de données dans le parallélisme de données est cependant restreinte à l'utilisation de ces indices, ce qui ne permet pas l'expression de dépendances de données plus complexes (motifs à trous ou recouvrement des données par exemple).

En ce qui concerne le processus d'optimisation, celui-ci explore les diverses possibilités de défactorisations d'un graphe d'application mais ne gère pas sa factorisation : il n'est pas possible de réduire le coût d'implémentation du circuit généré à partir de la description initiale du graphe de tâches.

1.2.2.4 MATLAB et SIMULINK

MATLAB [7] est un logiciel de calcul numérique et de visualisation graphique dont la plus grande partie des fonctions repose sur le calcul matriciel. Les fonctions spécialisées de

[7] http://www.mathworks.fr/products/matlab/

MATLAB (pour le traitement du signal notamment) en font un langage de programmation adapté aux besoins de l'ingénierie.

SIMULINK [8] est la couche graphique de MATLAB, il permet la représentation de systèmes et de fonctions mathématiques sous la forme d'un diagramme structurel. Ce diagramme permet de mettre en évidence les relations entre les différents éléments d'un système. Dans SIMULINK, le type de bloc appelé *Signal Processing Blockset* met à disposition des utilisateurs des fonctions plus ou moins avancées pour le traitement du signal.

SYNPLIFY DSP Balakrishnan et Eddington [9] modélisent les applications pour l'outil SYNPLIFY DSP de SYNPLICITY [81] à l'aide de SIMULINK. La modélisation d'une application peut nécessiter l'utilisation de blocs dédiés à des fonctionnalités particulières comme les filtres FIR. Il est possible de procéder à des optimisations de ces blocs par le biais, entre autres, de la fonction *folding* qui permet de réduire le coût d'implémentation d'un circuit en contrepartie d'une perte de la puissance de calcul. La fonction *folding* d'un bloc correspond à la séquentialisation de l'exécution du calcul réalisé par ce bloc et est paramétrable par le degré de séquentialisation choisi.

Le tableau 1.1 illustre les résultats d'implémentation sur un FPGA Virtex-4 d'un même bloc de calcul sans optimisation, puis avec des optimisations du type *folding* (avec des facteurs 4, 8 puis 16). La fréquence maximale du circuit diminue légèrement avec l'augmentation de *folding* car des ressources combinatoires sont introduites dans le chemin de données du circuit afin de gérer la séquentialisation des tâches. Le débit de sortie est exprimé en MS/s, MegaSample/second (mega-échantillons par seconde) et diminue fortement avec l'augmentation du *folding* : la séquentialisation des tâches a un impact direct sur le débit en sortie. En contrepartie, le nombre de ressources exploitées (ici les blocs multiplieurs) diminue avec le même facteur que *folding*.

TAB. 1.1: Impact de l'optimisation *folding* sur les circuits générés depuis l'outil Synplify DSP : exemple du bloc FIR [9]

Facteur *folding*	Frequence maximale estimée (MHz)	Débit maximal (MS/s)	Blocs DSP48
aucun	414	> 400	16
4	346	86	4
8	352	44	2
16	338	21	1

Génération de code VHDL et bilan Simulink met à disposition HDL CODER [107] pour la génération de code VHDL des applications. De la même façon, les extensions de Simulink par Synplify DSP permettent la génération de code VHDL des circuits éventuellement optimisés. L'expression des calculs sur des données multidimensionnelles et la possible génération de code étendent considérablement la portée d'utilisation de MATLAB et SIMULINK pour le traitement de signal intensif. Cependant, les dépendances de données sont exprimées à l'aide d'indices, les inconvénients liés à la manipulation de ces indices ont été évoqués précédemment.

[8] http://www.mathworks.com/products/simulink/

1.2.2.5 C vers VHDL

Depuis quelques années déjà, la tendance dans le domaine de la synthèse de haut niveau est à la génération d'accélérateurs depuis des descriptions dans le langage C où « à la C » (Handel-C, etc.). Le choix de ce langage est lié aux habitudes des programmeurs et à son côté très intuitif. De nombreux outils, tant académiques que commerciaux, ont ainsi vu le jour ces dernières années. D'un côté commercial, on note l'arrivée des logiciels C-TO-HARDWARE [69] pour Altera, PICO [9] pour Synfora, CODEVELOPER [10] pour Impulse, DK DESIGN SUITE [11] de CELOXICA ou encore CATAPULT C [12] pour MENTOR GRAPHICS. D'un point de vue académique, on cite les outils STREAM-C [50] du laboratoire national de Los Alamos, le projet PARO (Massively Parallel VLSI Architectures) [12] de l'université Friedrich-Alexander, ROCCC [53, 23] de l'université de Riverside, SA-C [103] de l'université du Colorado, SPARK [54] de l'université de San Diego, DEFACTO [20] de l'université de la Californie du sud ou encore GAUT [27] du Lester en France.

Il existe bien d'autres outils encore, notre intention n'est pas de les présenter tous. Nous détaillons SPARK qui reste l'un des outils les plus connus.

SPARK La figure 1.7 représente le flot de conception de l'outil SPARK issu de [122]. En plus de la description d'une application dans le langage ANSI-C, SPARK [54] prend en considération des contraintes de ressources, des contraintes de temps d'exécution et des directives permettant de guider transformations et heuristiques. Une représentation intermédiaire de l'application est alors générée, elle conserve les informations structurelles (de types « Si-Alors-Sinon » ou de boucles), les opérations ainsi que les dépendances de données. Cette représentation intermédiaire, illustrée sur la droite de la figure 1.7, est le format utilisé tout au long de la synthèse d'une application. Ainsi, parmi la majorité des outils de compilation de C vers VHDL utilisant des formats intermédiaires de représentation, le plus utilisé est sans doute SUIF [125].

La première étape de la synthèse (présynthèse) consiste, entre autres, à restructurer la représentation intermédiaire de l'application à l'aide de transformations de boucles (déroulage, fusion, etc.), à supprimer le code mort, à propager les constantes et à remplacer les opérations coûteuses (multiplications et divisions) par des opérations plus simples (décalages, additions et soustractions).

La seconde étape ordonnance et alloue les ressources matérielles nécessaires à l'exécution de l'application. Cette étape est guidée par l'utilisateur qui fournit les ressources allouées en utilisant une bibliothèque. L'ordonnanceur est composé d'heuristiques qui guident le processus d'optimisation. Ces heuristiques peuvent appeler des fonctions de transformation (*Transformation Tolbox* sur la figure) mais en restent toutefois indépendantes. Cela facilite le développement de nouvelles heuristiques et de nouvelles transformations.

La troisième étape place les opérations sur des ressources fonctionnelles, interconnecte ces ressources, place les données dans des registres et génère un contrôleur (une machine à états finis) pour la réalisation matérielle de l'ordonnancement. Dans le cas de SPARK, le placement est réalisé de manière à minimiser les interconnexions entre les ressources fonctionnelles et les registres.

[9] http://www.synfora.com/products/picoexpress.html
[10] http://www.impulsec.com/C_to_fpga.htm
[11] http://www.celoxica.com/products/dk/default.asp
[12] http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/

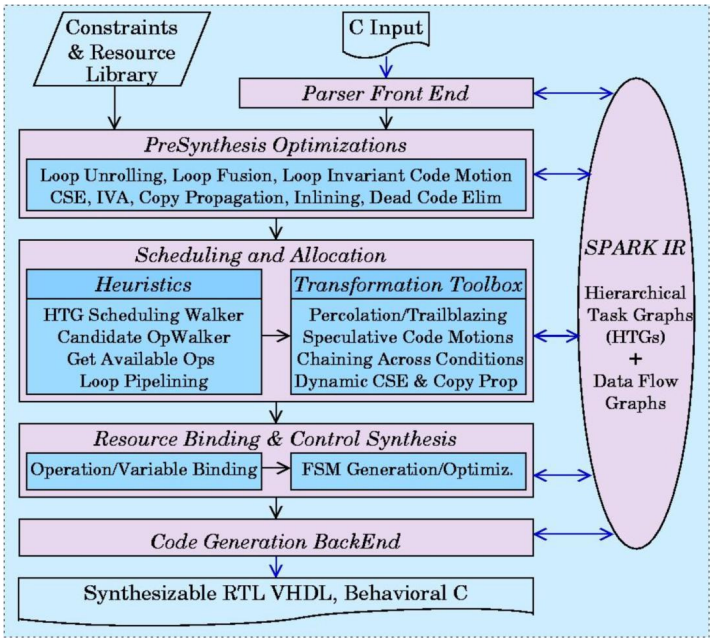


FIG. 1.7: Le flot de conception dans SPARK [122].

En dernier lieu, le code VHDL synthétisable du circuit est généré. Les ressources fonctionnelles sont implémentées sous la forme de composants, ce qui donne une structure au circuit. Il est aussi intéressant de noter la génération d'un code C qui autorise la comparaison entre les descriptions initiale et finale. Il est alors possible de se rendre compte des optimisations réalisées par SPARK.

Bilan de la génération d'accélérateurs depuis le langage C Certains outils cités précédemment ont des inconvénients bien précis qui les rendent non utilisables pour le traitement de signal systématique sur des données multidimensionnelles, comme STREAM-C ou GAUT qui ne gèrent qu'une seule dimension ou ROCCC qui est limité à deux dimensions. Cependant, la caractérisation et la comparaison de tous ces outils restent difficiles car ils évoluent rapidement. C'est donc d'une manière plus générale que nous analysons la génération de code HDL depuis le langage C.

Indéniablement, l'utilisation de C pour la description d'applications vouées à être exécutées par des accélérateurs matériels plaît. Une description en C offre, en théorie, l'opportunité de ne pas travailler au niveau RTL. Cependant, cela ne se fait pas sans quelques inconvénients majeurs :

– une description textuelle, sans doute adaptée aux « petites » applications, rend plus difficile la description d'une application plus complexe comprenant de la hiérarchie, du parallélisme de tâches et du parallélisme de données. La hiérarchie apparait sous la forme de fonctions, le parallélisme de données sous la forme de boucles et le parallélisme de tâches sous la forme d'opérations successives. La construction d'une application « complexe » nécessite de conserver à l'esprit la structure exacte du programme. Par ailleurs, une restructuration d'un programme nécessite la manipulation et le dé-

placement de « bouts de code », qui est à la base un problème majeur pour la construction d'accélérateurs au niveau RTL. Une représentation graphique ne possède pas ces inconvénients, ou les réduit fortement du moins ;
- l'expression des dépendances de données est exprimée par des indices. La manipulation de ces indices est un frein pour la description de dépendances de données sur des tableaux multidimensionnels car le code devient très rapidement peu lisible et donc difficilement réutilisable ;
- la plupart des outils présentés précédemment tentent d'extraire tout le parallélisme potentiel d'une application. Or, le langage C exprime le parallélisme potentiel sous la forme de boucles séquentielles. La reconstruction des dépendances de données à partir du langage C n'est donc pas aisée, alors qu'elle pourrait directement être exprimée par le concepteur (de la même façon que le nœud *iterate* de SYNDEX-IC exprime les dépendances entre les tâches) ;
- le dernier point est plus philosophique et concerne la revendication d'utiliser le langage C car « connu de tous ». En effet, les outils nécessitent souvent qu'un programme soit annoté et/ou que l'utilisateur définisse en marge de ce programme des paramètres de compilation/optimisation, comme c'est le cas pour SPARK [122]. La syntaxe des annotations et des paramètres est dépendante des outils : un nouvel utilisateur n'a donc pas, à priori, les connaissances suffisantes pour exploiter cet outil et est donc contraint, comme c'est toujours dans le cas pour un nouveau langage, d'appréhender ces syntaxes. Par ailleurs, seul un sous-ensemble du langage C est géré par ces outils, c'est le cas pour les pointeurs par exemple. Ainsi, ce n'est pas vraiment du C qui permet la génération de code HDL, mais un « sous-ensemble de C annoté ».

1.2.2.6 Bilan des outils

La description textuelle d'une application repose uniquement sur le code qui contient à la fois l'expression des dépendances de données et des calculs. Leur extraction nécessite par conséquent une analyse de l'intégralité du code. Or, les dépendances de données sont omniprésentes dans les applications de traitement de signal systématique où elles sont généralement représentées sous la forme d'un graphe de tâches. De telles représentations mettent en évidence la hiérarchie, le parallélisme de tâches et le parallélisme de données d'une application. Une description textuelle ne fait donc pas profiter de ces avantages.

Les outils présentés s'accordent sur la nécessité de monter en abstraction pour représenter des applications afin notamment de se focaliser sur l'intention de ces applications plutôt que sur les détails d'implémentation. Cependant, les inconvénients liés aux descriptions textuelles évoqués précédemment nous laissent penser qu'une réelle abstraction des détails d'implémentation n'est possible qu'avec une représentation graphique.

Nous constatons par ailleurs des restrictions quant au nombre de dimensions manipulables dans les tableaux de données. En effet, de nombreux outils se limitent à la gestion de une ou deux dimensions [50, 53, 23, 27], alors même que des applications de traitement de signal intensif « classiques » (traitement vidéo par exemple) nécessitent plus de deux dimensions. Cette contrainte est un obstacle à l'utilisation de ces outils. Cet obstacle se révèle d'autant plus important lorsque les accès aux données dans les tableaux sont réalisés sous la forme d'indices que le programmeur doit calculer. En effet, le calcul de ces indices est un travail à la fois long, fastidieux, délicat et souvent à l'origine de disfonctionnements de programmes. L'expression des dépendances de données dans les applications de traitement

du signal intensif est définitivement un point clé. L'expression de ces dépendances doit par conséquent être « dissociée des calculs » de manière à en faciliter la saisie, l'extraction ou l'analyse.

1.3 ARRAY-OL : spécification d'applications pour le traitement de signal intensif

ARRAY-OL (Array Oriented Langage) est un modèle de calcul qui permet la description d'applications de traitement de signal intensif qui manipulent une grande quantité de données traitées de façon régulière par des tâches. ARRAY-OL a été inventé par Alain Demeure [33] en 1995 chez TUS (THALES Underwater System).

ARRAY-OL est né du constat que la complexité de description des applications provient généralement de la spécification des accès aux données par les tâches de calcul. Ainsi, ARRAY-OL n'exprime pas la fonctionnalité d'une tâche mais se focalise sur la spécification des dépendances de données de l'application, que ce soit au niveau du parallélisme de tâches ou du parallélisme de données. Les données sont représentées sous la forme de tableaux multidimensionnels dont la signification (spatiale, temporelle, etc.) est banalisée. ARRAY-OL est par ailleurs un langage à *assignation unique* : une donnée dans un tableau ne peut être produite qu'une seule fois.

Une application ARRAY-OL se décompose sur deux niveaux que sont le parallélisme de tâches et le parallélisme de données. Le parallélisme de tâches, aussi appelé *modèle global*, définit les dépendances de données qui sont assimilées à des tableaux entre différentes tâches. Le parallélisme de données, aussi appelé *modèle local*, définit les dépendances de données entre une tâche répétée et les tableaux de données auxquels la tâche est connectée.

La hiérarchie d'une application ARRAY-OL permet par ailleurs de spécifier du parallélisme de tâches dans du parallélisme de données et du parallélisme de données dans du parallélisme de tâches.

1.3.1 Expression du parallélisme de tâches

Le parallélisme de tâches est fourni sous la forme d'un graphe dirigé acyclique dans lequel les tâches sont représentées par des nœuds et les tableaux par des arcs, comme illustré à la figure 1.8.

Il n'existe aucune restriction quant au nombre de tableaux et leur dimension dans un modèle global. Une tâche peut donc, par exemple, consommer deux tableaux bidimensionnels et produire un tableau tridimensionnel. Cela permet notamment de créer une dimension qui correspond à la fréquence pour les transformées de Fourier rapides (Fast Fourier Transform, FFT). La dimension temporelle est aussi représentée dans les tableaux mais reste banalisée (il est par exemple possible de représenter un cycle temporel sous la forme d'un tableau fini), l'exception reste la dimension *infinie* qui est associée au temps. Enfin, les tableaux sont toriques, ce qui permet de reconstituer via le modulo la proximité des données telle qu'elle existe dans la réalité (par exemple pour la représentation de données que reçoit l'ensemble des antennes dispersées en cercle autour du sous-marin).

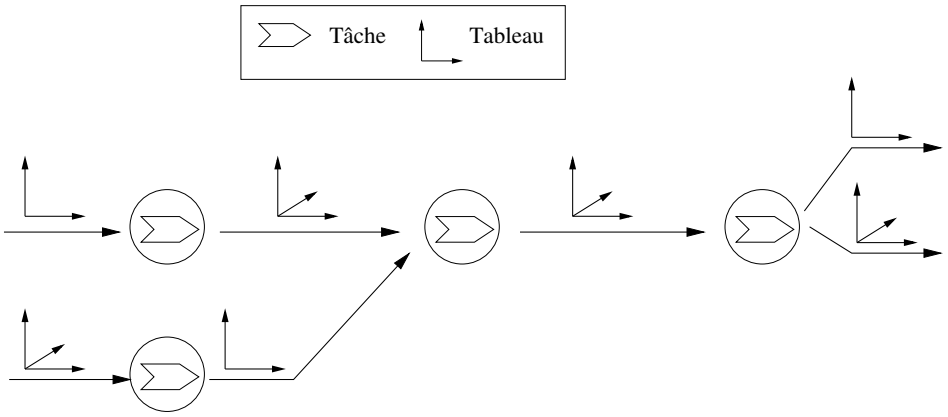


FIG. 1.8: Un exemple de modèle global (parallélisme de tâches) en ARRAY-OL. Les symboles que portent les arcs symbolisent le nombre de dimensions que possède le tableau : la tâche en haut à gauche consomme un tableau bidimensionnel et produit un tableau tridimensionnel.

1.3.2 Expression du parallélisme de données

Le parallélisme de données est exprimé par un modèle local de ARRAY-OL, qui est composé des tableaux d'entrée et de sortie, d'une tâche répétée et de l'expression des dépendances de données. La tâche est répétée selon un *espace de répétition*, chaque répétition consomme (ou produit pour la sortie) un sous-ensemble des tableaux auxquels elle est connectée. Ce sous-ensemble, appelé *motif*, peut être multidimensionnel et est construit à partir d'un *tiler*. La figure 1.9 illustre un exemple de modèle local en ARRAY-OL.

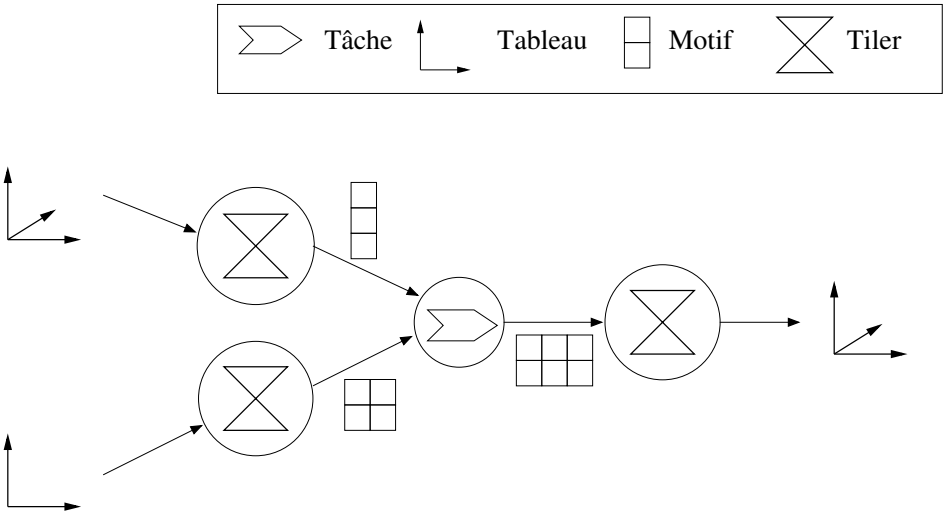


FIG. 1.9: Un exemple de modèle local en ARRAY-OL

Un tiler exprime les dépendances de données entre des motifs et un tableau par le biais de :

- $\vec{m}$: la taille des dimensions du tableau ;
- $\vec{d}$: la taille des dimensions du motif ;

- $\overrightarrow{o}$: l'origine du motif référence dans le tableau ;
- P : la *matrice de pavage* qui décrit la manière dont les motifs pavent le tableau ;
- F : la *matrice d'ajustage* qui décrit la manière dont les motifs sont remplis par les éléments du tableau.

1.3.2.1 Pavage

La construction des motifs débute par le calcul des coordonnées de leur origine dans un tableau à partir de l'origine $\overrightarrow{o}$ du motif référence et de la matrice de pavage P. Ce calcul est un processus répétitif et correspond à la somme des coordonnées de l'origine et d'une combinaison linéaire des vecteurs de pavage, le tout modulo la taille du tableau puisque les tableaux ARRAY-OL sont toriques. L'équation 1.1 correspond à ce calcul : $\overrightarrow{x_q}$ représente le motif d'indice q, $\overrightarrow{Q}$ représente l'espace de répétition, et $\overrightarrow{r_q}$ représente l'origine du motif d'indice q.

$$\forall \overrightarrow{x_q}, \overrightarrow{0} \leq \overrightarrow{x_q} < \overrightarrow{Q}, \overrightarrow{r_q} = (\overrightarrow{o} + P \times \overrightarrow{x_q}) \mod \overrightarrow{m} \tag{1.1}$$

La figure 1.10 représente l'origine de chaque motif dans un tableau bidimensionnel dans le cas d'un tiler défini par une origine $\left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$ et une matrice de pavage $\left(\begin{smallmatrix}2&0\\0&3\end{smallmatrix}\right)$. Les origines de tous les motifs sont construites par itérations successives. Ainsi, l'origine du premier motif correspond à l'origine du motif référence (point d'origine, ici $\left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$). L'origine de chacun des autres motifs est obtenue en multipliant chacun des vecteurs de la matrice de pavage par 0, 1, 2, etc. Le tableau est donc balayé de 2 en 2 à l'horizontale et de 3 en 3 en verticale.

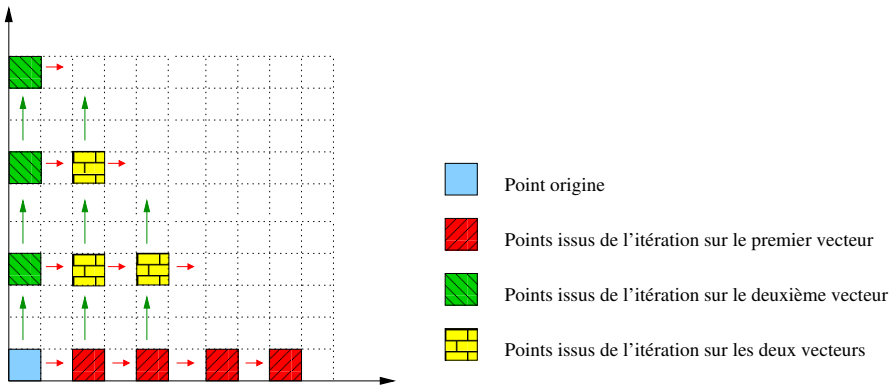


FIG. 1.10: Origine de chaque motif dans un tableau. La distinction de la construction des points issus des itérations sur le premier puis sur le second vecteur est réalisée dans le seul but de faciliter la compréhension du pavage. En pratique, aucune distinction n'est faite.

1.3.2.2 Ajustage

La seconde partie de la construction des motifs correspond à leur remplissage à partir de leur origine dans le tableau. Le remplissage des motifs est défini par la matrice d'ajustage qui est composée d'un ensemble de vecteurs, chacun d'eux étant associé à une dimension du motif. Le calcul pour le remplissage des motifs est similaire au calcul des coordonnées de leur origine : les éléments d'un tableau qui constituent un motif sont calculés par la somme

des coordonnées de l'origine du motif et d'une combinaison linéaire de la matrice d'ajustage, le tout modulo la taille du tableau. Ce calcul est réalisé par l'équation 1.2, où $\overrightarrow{x_d}$ représente l'élément d'indice d du motif.

$$\forall \overrightarrow{x_d}, \overrightarrow{0} \leq \overrightarrow{x_d} < \overrightarrow{D}, (\overrightarrow{r_q} + F \times \overrightarrow{x_d}) \mod \overrightarrow{m} \tag{1.2}$$

La figure 1.11 illustre deux constructions de motifs à partir d'un tableau unidimensionnel et d'un motif bidimensionnel de forme $\binom{3}{2}$. Dans les deux cas, les coordonnées de l'origine du motif dans le tableau sont $\binom{0}{}$. Le haut de la figure représente la construction d'un motif pour la matrice d'ajustage $\binom{1}{3}$. Les 6 éléments du motif sont caractérisés par leur coordonnées, $\binom{0}{0}$, $\binom{1}{0}$, $\binom{2}{0}$, $\binom{0}{1}$, $\binom{1}{1}$, $\binom{2}{1}$. La valeur de l'élément est obtenue en multipliant ses coordonnées avec le vecteur d'ajustage.

- élément $\binom{0}{0} : 0 \times 1 + 0 \times 3 = 0$
- élément $\binom{1}{0} : 1 \times 1 + 0 \times 3 = 1$
- ...
- élément $\binom{2}{1} : 2 \times 1 + 1 \times 3 = 5$

De la même façon, pour le bas de la figure où la matrice d'ajustage est $\binom{2}{6}$, on obtient

- élément $\binom{0}{0} : 0 \times 2 + 0 \times 6 = 0$
- élément $\binom{1}{0} : 1 \times 2 + 0 \times 6 = 2$
- ...
- élément $\binom{2}{1} : 2 \times 2 + 1 \times 6 = 10$

Nous observons que les données consommées par le motif du haut de la figure sont consécutives dans le tableau alors qu'ils sont espacés pour le bas. Il est aussi intéressant de remarquer que les tilers permettent de construire des motifs dont le nombre de dimensions est différent de celui du tableau.

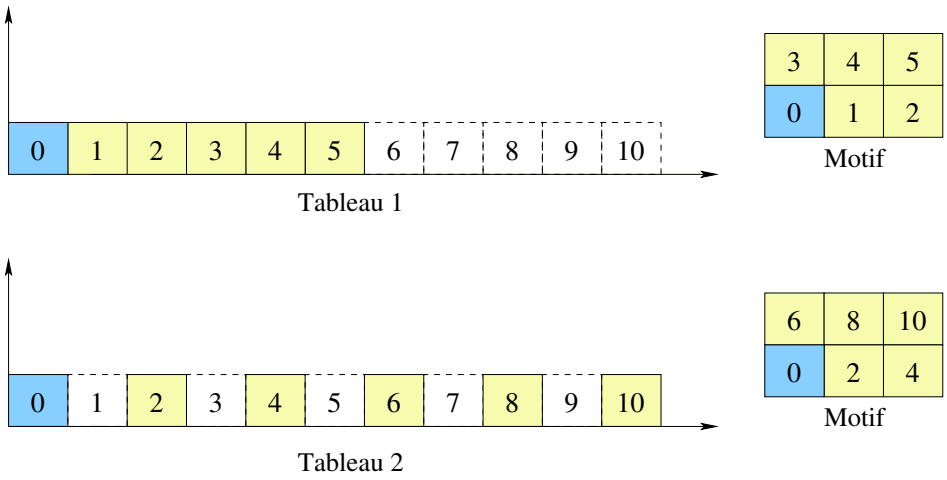


FIG. 1.11: Construction de motifs bidimensionnels à partir d'un tableau monodimensionnel.

1.3.2.3 Combinaison du pavage et de l'ajustage

Nous illustrons la combinaison du calcul du pavage et de l'ajustage par la figure 1.12 et ces cinq cas :

- classique : $\overrightarrow{o} = \binom{0}{0}$, $\overrightarrow{d} = \binom{3}{2}$, $F = \left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$ et $P = \left(\begin{smallmatrix}3 & 0\\ 0 & 2\end{smallmatrix}\right)$
- des motifs non parallèles aux axes : avec $\overrightarrow{o} = \binom{0}{0}$, $\overrightarrow{d} = \binom{2}{3}$, $F = \left(\begin{smallmatrix}1 & 1\\ 0 & 1\end{smallmatrix}\right)$ et $P = \binom{2}{0}$
- des motifs qui s'entrecroisent : $\overrightarrow{o} = \binom{0}{0}$, $\overrightarrow{d} = \binom{2}{2}$, $F = \left(\begin{smallmatrix}1 & 0\\ 0 & 2\end{smallmatrix}\right)$ et $P = \binom{0}{1}$
- deux motifs successifs qui se chevauchent : $\overrightarrow{o} = \binom{0}{0}$, $\overrightarrow{d} = (3)$, $F = (1)$ et $P = \binom{1}{0}$
- un tableau torique : $\overrightarrow{o} = \binom{2}{0}$, $\overrightarrow{d} = (3)$, $F = (1)$ et $P = \left(\begin{smallmatrix}3 & 0\\ 0 & 1\end{smallmatrix}\right)$

Le vecteur d'origine de la première combinaison définit les coordonnées de référence utilisée pour la construction des différents motifs. Par ailleurs, ces coordonnées correspondent aussi aux coordonnées d'origine du premier motif dans l'espace de répétition (en rouge sur la figure) : (0,0). Les coordonnées d'origine du second motif (plus précisément, du motif issu de la première itération sur l'une des dimensions de l'espace de répétition) sont définies par la matrice de pavage qui spécifie un déplacement de 3 sur l'axe horizontal et de 0 sur l'axe vertical. Il en résulte la position (3,0). A partir de ces coordonnées d'origine, la matrice d'ajustage permet de définir les données utilisées pour la construction des motifs. Dans cet exemple, les motifs sont construits parallèlement aux axes et de façon compacte car seuls des déplacements unitaires dans le tableau sont réalisés, ces déplacements étant dépendants d'une dimension de l'espace de répétition bien précise (déplacement horizontal pour la première dimension de l'espace de répétition, déplacement vertical sinon). $\overrightarrow{d}$ définit les bornes l'itération pour la construction des motifs qui est un rectangle dans cet exemple. Quelque soit le motif, la forme dans le tableau est identique.

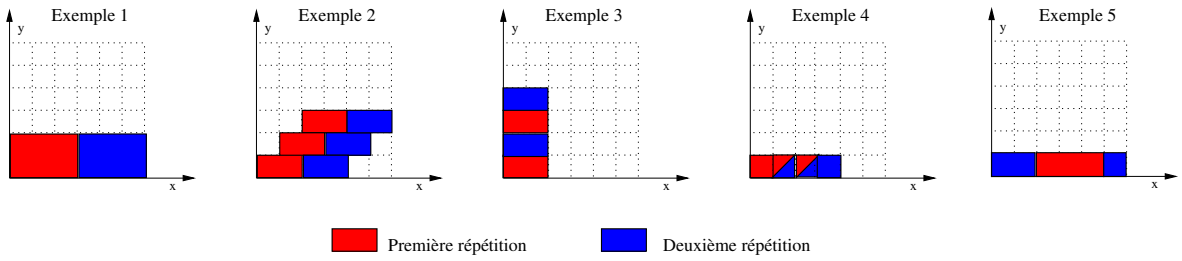


FIG. 1.12: Différentes constructions de motifs.

1.3.3 Exemple de la multiplication de matrices

Nous illustrons le langage ARRAY-OL par la description d'une multiplication de matrices. Soit la matrice A1 de taille 5×3 et la matrice A2 de taille 2×5. La matrice A3 de taille 2×3 est issu du résultat du produit des matrices A1 et 2 : $\mathrm{A1} \times \mathrm{A2} = \mathrm{A3}$. Le produit de matrice peut être décomposé en calcul du produit scalaire de chaque ligne de A1 par chaque colonne de A2 Les calculs nécessaires à la réalisation du produit de matrices sont indépendamment les uns des autres et peuvent donc être exécutés en parallèle.

La description en ARRAY-OL de ce produit de matrice est représentée à la figure 1.13. La tâche élémentaire `ProduitScalaire` correspond au produit scalaire réalisé entre chaque ligne de la matrice A1 et chaque colonne de la matrice A2. La matrice de sortie est de taille 2×3, l'espace de répétition est donc nécessairement défini par le vecteur $[2, 3]$ pour remplir la matrice A3. La tâche élémentaire consomme deux motifs pour produire un point dans la matrice de sortie. Ces motifs sont tous deux de taille 5 et sont construits à partir de A1 et A2 par le biais de tilers. Sur la figure 1.13, nous représentons en pointillés les éléments

consommés et produits lors de la première exécution de la tâche sur son espace de répétition, et illustrons en rouge et en vert les deux premières itérations de la tâche dans son espace de répétition. Les flèches vertes illustrent le déplacement du motif sur les lignes de la matrice A1, les flèches rouges les déplacements de l'autre motif sur les colonnes de la matrice A2. On retrouve ces flèches sur la matrice A3 pour en illustrer la construction.

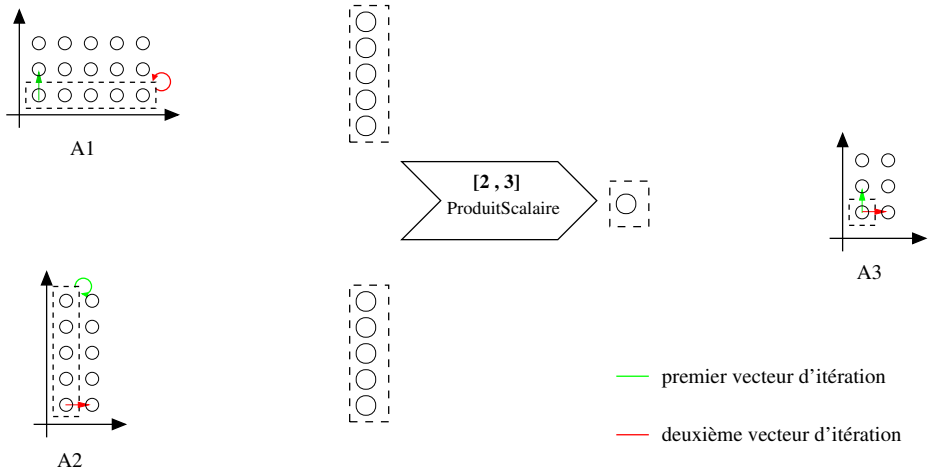


FIG. 1.13: Exemple d'un produit de matrice

1.3.4 fonctions de refactoring ARRAY-OL

Depuis sa création en 1995, des travaux réalisés au sein de notre équipe ont permis de faire évoluer ARRAY-OL. En 2001, Julien Soula propose des fonctions de *refactoring* des applications ARRAY-OL [110], ces fonctions sont enrichies en 2005 par Philippe Dumont [39] qui propose entre autres la *boîte à outils* ARRAY-OL [115]. Encore aujourd'hui, les travaux menés par Calin Glitia visent à étendre l'utilisation du refactoring sur des dépendances de données inter-itération (dépendances qui permettent la représentation d'architectures systoliques par exemple).

Ces fonctions de refactoring sont développées de manière à ce qu'une application ARRAY-OL refactorée reste une application ARRAY-OL, plusieurs fonctions de refactoring successives peuvent donc être réalisées sur une même application. Il n'est donc pas nécessaire de passer par une représentation dans un format intermédiaire et le concepteur de l'application a directement accès au résultat d'une fonction de refactoring. Les fonctions de refactoring permettent de créer des hiérarchies, d'en supprimer, de fusionner les boucles, etc. Chacune d'elles trouve son utilité pour atteindre un objectif dans la description d'une application. Il en découle souvent un impact significatif sur le comportement de l'exécution d'une application sur une cible d'exécution donnée (nous présentons certains de ces modèles d'exécution de ARRAY-OL par la suite). Par exemple, l'utilisation de la fusion permet de réduire la quantité de données manipulées dans un tableau produit par une tâche et consommé par une autre. Lors d'une exécution sur une architecture multiprocesseurs, cela permet d'exploiter les mémoires caches pour le stockage de ce tableau et de limiter ainsi les défauts de cache [19]. Ces fonctions de refactoring sont aussi utilisées pour structurer une application de manière à lui donner une forme bien précise et attendue par un processus d'optimisation

par exemple. C'est dans ce contexte d'optimisation que nous utilisons certaines fonctions de refactoring, nous détaillons cet aspect dans le chapitre 6.

1.3.5 Différentes exécutions de ARRAY-OL

Le modèle de calcul ARRAY-OL est indépendant de toute exécution, il spécifie les dépendances de données entre les tâches d'une application. L'exécution d'une application ARRAY-OL dépend de la cible d'exécution choisie. Dumont [39] identifie les cibles d'exécution séquentielle, SPMD, pipelinée et sur mémoires distribuées :

- l'exécution séquentielle est principalement utilisée pour la validation fonctionnelle de l'application ;
- l'exécution SPMD permet l'exécution du parallélisme de données sur des architectures multiprocesseurs à mémoires partagées. Les tableaux sont placés sur ces mémoires et les motifs sont construits/produits localement par les processeurs qui exécutent les tâches ;
- l'exécution pipelinée traite les tableaux de données entre les tâches sous la forme *d'un flux de tableaux*. Les différents étages d'un pipeline de tâches peuvent alors être exécutés en parallèle, mais aucun étage ne travaille sur la même itération. Dans cette exécution, ce sont les données qui circulent et non les tâches ;
- l'exécution sur une architecture à mémoires distribuées permet des exécutions localisées sur chaque processeur du parallélisme de l'application. L'objectif est de minimiser les communications entre les processeurs en limitant les transferts de données.

Récemment, Wood *et al.* [126] proposent d'exécuter en matériel les applications ARRAY-OL en utilisant des accélérateurs matériels. Wood *et al.* décrit la relation entre un accélérateur et l'application qu'il exécute sous deux aspects. Le premier est l'espace de répétition d'une tâche qui correspond dans l'accélérateur (décrit en VHDL) à un paramètre d'instanciation de composant (ce composant réalise en VHDL la tâche répétée). Le second aspect est l'utilisation de paramètres génériques pour le calcul des dépendances de données exprimées par les tilers. Ces paramètres définissent par exemple l'origine et la dimension des motifs. C'est donc une méthodologie reposant sur les paramètres génériques en VHDL que proposent Wood *et al.*.

1.3.6 Comparaison de ARRAY-OL avec d'autres langages et bilan

La gestion des tableaux multidimensionnels est un avantage de ARRAY-OL vis-à-vis des différents outils et modèles de calcul présentés précédemment dans ce chapitre. En effet, ces outils sont bien souvent restreints à la manipulation de une ou deux dimensions tandis que ARRAY-OL n'impose aucune restriction, ni sur le nombre de dimension ni sur la sémantique des dimensions. Les dépendances de données en ARRAY-OL sont exprimées à l'aide de matrices et de vecteurs (dans le tiler) qui permettent la construction de motifs à partir d'un tableau. Dans les autres outils, la construction de ces motifs est réalisée par des indices sur des variables utilisées généralement lors de la description des calculs. Il est ainsi plus délicat d'extraire les dépendances de données qu'avec ARRAY-OL.

Dumont [39] compare ARRAY-OL à SDF, MDSDF et GMDSDF et conclut sur un avantage de ARRAY-OL sur différents points, comme la manipulation de tableaux toriques et la création de motifs à partir de données disjointes dans un tableau.

Nous concluons de ces différents éléments que ARRAY-OL est adapté pour le traitement du signal intensif. La manipulation possible de tableaux multidimensionnels et l'expression factorisée des dépendances de données sont certainement ses deux atouts majeurs. Par ailleurs et d'un point de vue plus pragmatique, les fonctions de refactoring ARRAY-OL et les modèles d'exécution existants contribuent à renforcer le choix d'utiliser ARRAY-OL pour la description d'applications massivement parallèles.

Cependant, la plupart des modèles d'exécution de ARRAY-OL existant visent des architectures multiprocesseurs. La seule exception provient des travaux menés par Wood *et al.* [126] qui présente l'exécution sur un accélérateur d'une tâche répétée en ARRAY-OL. Bien qu'il ne soit jamais question dans ces travaux de hiérarchie ni de parallélisme de tâches, l'exécution des tâches ARRAY-OL par des accélérateurs matériels semble prometteuse. En effet, l'application est exécutée sur une architecture taillée sur mesure pour ses besoins en ressources de calcul et ses dépendances de données.

1.4 Réalisation physique des accélérateurs

Dans cette section, nous nous intéressons à la réalisation physique des accélérateurs par les technologies les plus fréquemment utilisées que sont les ASICs et les FPGAs. Nous présentons par la suite des travaux qui tendent à représenter les FPGAs et le placement de tâches (que nous associons à un accélérateur ou à un sous-ensemble d'un accélérateur).

1.4.1 Technologies ASIC et FPGA

En terme d'implémentation (ou de réalisation physique), la technologie ASIC consiste à réaliser l'accélérateur directement sur une puce, c'est-à-dire en exploitant les transistors de cette puce. Il en résulte un circuit très performant, mais coûteux, long à produire et dont la conception ad hoc peut engendrer de nombreuses erreurs. En effet, la fabrication d'un ASIC fait intervenir différentes équipes de conception et multiplie ainsi les interventions humaines. Une alternative à la solution ASIC est la technologie FPGA qui offre la possibilité de réaliser physiquement l'accélérateur par une configuration. La notion de configuration peut être assimilée à une couche entre les transistors de la puce et l'accélérateur implémenté, c'est cette couche qui est exploitée pour la réalisation de l'accélérateur.

Le choix entre l'utilisation d'une technologie ASIC ou FPGA est généralement déterminé par deux critères que sont la performance du circuit et la quantité de pièces à produire.

Performance du circuit En terme de performance et pour une même technologie d'implémentation ($180\ nm$ par exemple), les ASICs prennent l'avantage sur les FPGAs. Cela est en partie lié aux connexions qui, dans les FPGAs, sont parasitées par les éléments de la couche reconfigurable et qui sont contraintes d'exploiter des ressources de routage reconfigurables existantes. Dans la pratique, on observe que les FPGAs exploitent des technologies de gravure plus récentes que celles utilisées pour la fabrication de puces dédiées. En effet, le passage à de nouvelles technologies est coûteux et les concepteurs préfèrent utiliser des technologies maîtrisées et connues. Ainsi, un système réalisé de façon dédiée sur puce peut parfois être réalisé de façon aussi performante sur un FPGA qui exploite une technologie de gravure plus récente : la transition vers de nouvelles technologies de gravures compense les pertes introduites par la couche reconfigurable du FPGA.

Quantité de pièces La quantité de pièces à produire est aussi un critère de sélection. En effet, si les ASICs sont coûteux à la conception, le coût de production d'une puce est relativement faible une fois le masque réalisé et le circuit validé. En revanche, le coût d'un prototypage sur FPGA est relativement faible puisqu'il ne fait pas intervenir de mécanisme de gravure sur silicium. Une fois la validation du prototype terminée, le prix de revient d'une pièce inclut nécessairement le prix d'achat d'un FPGA. Par ailleurs, Les ASICs structurés sont une solution intermédiaire, ils sont à mi-chemin entre les FPGAs et les ASICs. Ils sont construits depuis une configuration de FPGA mais sont toutefois gravés sur silicium. Seule une partie des masques est spécifique au circuit gravé, offrant un compromis coût-performance intéressant. Le coût de concept d'un ASIC structuré est inférieur à celui d'un ASIC le prix de revient moins élevé que les FPGAs et les ASIC pour les productions intermédiaires.

Bilan Les performances toujours croissantes des FPGAs et leur grande flexibilité en font un support de réalisation physique adéquat pour les accélérateurs matériels. Nous présentons dans cette section différentes représentations des FPGAs dans des outils de placement.

1.4.2 Différentes représentations des FPGAs et du placement

La synthèse d'un circuit (un accélérateur matériel par exemple) sur FPGA est décomposée en différentes étapes, chacune d'elle nécessite la connaissance de certaines caractéristiques du FPGA. L'une de ces étapes est le placement géographique du circuit sur FPGA, couramment appelé *floorplanning*. Le floorplanning consiste à découper le FPGA en zones sur lesquelles sont placés des tâches matérielles ou des éléments d'un circuit. Cette section s'intéresse aux représentations existantes des FPGAs, nous nous focalisons plus particulièrement sur l'expression des placements dans les différents travaux présentés.

Walder *et al.* [123] distinguent deux grandes catégories de placement de tâches sur FPGA : les placements *hors ligne* et les placements *en ligne*. Un placement hors ligne signifie que le placement est déterminé avant l'exécution (ou l'implémentation) sur le FPGA, ce qui laisse du temps au logiciel de placement pour trouver une solution optimale ou quasi optimale. Un placement en ligne signifie que le placement est déterminé lors de l'exécution des tâches sur FPGA, des contraintes temps réel sont alors prises en compte.

Nos travaux s'intéressent plus particulièrement aux placements hors ligne, nous les présentons dans un premier temps. Nous introduisons cependant des travaux relatifs au placement en ligne dans un second temps.

1.4.2.1 Placement hors ligne

Les outils de placement de tâches sur FPGA considèrent différentes représentations des FPGAs dont le niveau de description varie en fonction de l'objectif de l'outil de placement. Nous présentons dans les paragraphes suivants ces différentes représentations et les outils associés.

FPGAs homogènes et réguliers Les FPGAs homogènes et réguliers correspondent à l'architecture du type ilots de calcul, à laquelle sont souvent associés les FPGAs embarqués, embedded FPGA (eFPGA). L'outil Versatile Place and Route (VPR) [16] considère ce type d'architecture. La description réalisée du FPGA est paramétrable, il est par exemple possible

de modifier la taille de la grille d'ilots, le nombre d'entrées et sorties des blocs logiques reconfigurables, le nombre de broches du FPGA [17]. Cette description très précise du FPGA permet aux outils de placement et routage de générer des configurations de ces FPGAs (bitstreams). VPR demeure une référence dans le domaine du placement sur FPGA, et a récemment été étendu par Chaudhuri *et al.* [26] afin de l'exploiter en tant que modeleur de FPGA en plus de ses fonctionnalités de placement et routage. Pour cela, une description VHDL du FPGA est générée depuis sa description initiale, cela est possible car le niveau de description des FPGAs dans l'outil VPR st très proche du niveau RTL.

Lagadec *et.al* [68] introduisent le concept d'*architecture FPGA virtuelle*, similaire au concept de machine virtuelle. Cette vue virtuelle correspond à la vision qu'ont les applications du FPGA lors de leur implémentation par un OS, similairement à une mémoire virtuelle. En d'autres termes, les ressources physiques du FPGA sont regroupées en ensembles de ressources. Cette virtualisation permet de considérer le FPGA plus simplement, avec un grain correspondant à celui des applications qu'il implémente. L'objectif de la virtualisation est la définition d'une structure stable et portable des FPGAs virtuels pour le placement, le routage et l'édition. La figure 1.14 illustre la virtualisation par le biais de la création d'une couche virtuelle de blocs logiques et configurables à partir des blocs logiques et configurables physiques du FPGA. Les blocs de la couche virtuelle sont plus complexes que les blocs issus de la couche physique. Le principal avantage de cette virtualisation est la portabilité de l'architecture virtualisée et son principal défaut la possibilité que les ressources virtuelles ne soient pas pleinement exploitées lors d'un placement.

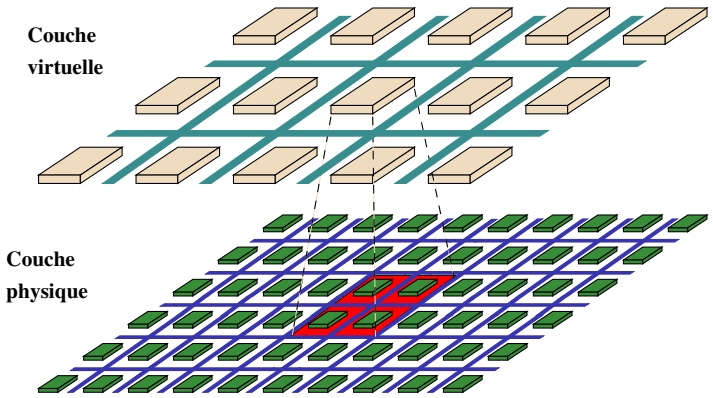


FIG. 1.14: Couches physique et virtuelle proposées par Lagadec *et al.* [68] : un bloc logique reconfigurable virtuel est composé de plusieurs blocs logiques reconfigurables physiques.

La description de la couche physique des FPGAs dans [68] permet de générer directement des bitstreams à partir de JBits [52], API (Application Program Interface) autorisant les applications Java à modifier dynamiquement les bitstreams des FPGA Virtex-II de Xilinx. Il est par ailleurs envisagé de générer le code VHDL du FPGA lui-même à partir de la couche physique [71]. La description du FPGA permet donc de générer à la fois des bitstreams pour des applications placées et le code VHDL du FPGA. Le niveau de description des FPGAs est donc proche du niveau RTL, de la même façon que dans l'outil VPR.

Tessier [116] décompose le placement de tâches sur FPGAs en deux étapes. La première étape décompose un FPGA en groupes de cellules appelés *bins* et assigne chaque tâche appe-

lée *Macros* à un bins. La seconde étape place au sein de chaque bins les *Hard Macro* et les *Soft Macro*, qui correspondent à des régions de placement dont la forme est respectivement rigide et flexible. Dans les deux cas, les Hard et Soft Macro sont des sous-ensembles des Macros. Le placement est ainsi décomposé en deux phases, ce qui introduit deux vues d'un FPGA, que l'on pourrait qualifier de globale ou locale. Cela permet de ne pas gérer directement toutes les ressources du FPGA, mais de les manipuler par groupe lors de la première étape de placement, puis par quantité réduite (les ressources contenues dans chaque groupe) lors de la seconde étape de placement.

Les représentations des FPGAs dans ces différents travaux sont très proches du niveau de description RTL. De l'avis de tous, la complexité d'une description de FPGA au niveau RTL est liée aux ressources de routages, à la fois nombreuses et très variées.

Abstraction des ressources de routages du FPGA Fabiani et Lavenier [45] proposent de placer des tableaux de processeurs linéaires sur FPGA. L'objectif de ces travaux est de réduire les temps de placement comparé aux outils commerciaux. Pour cela, Fabiani et Lavenier tirent profit de la nature régulière du circuit placé et décomposent le placement en deux étapes : un placement au niveau des zones du FPGA et un placement local à ces zones. Lors de la phase de placement dans les zones, les processeurs peuvent être implémentés sous différentes formes. La figure 1.15 représente une décomposition de FPGA en trois zones. La première contient des processeurs de forme A ou B, tandis que les zones 2 et 3 contiennent des processeurs de forme respective A et C. Le tableau linéaire de processeurs est disposé de manière à ce que chaque processeur soit proche des processeurs avec lesquels il communique. La flèche en pointillé illustre le parcours du tableau monodimensionnel de processeurs dans la grille bidimensionnelle du FPGA.

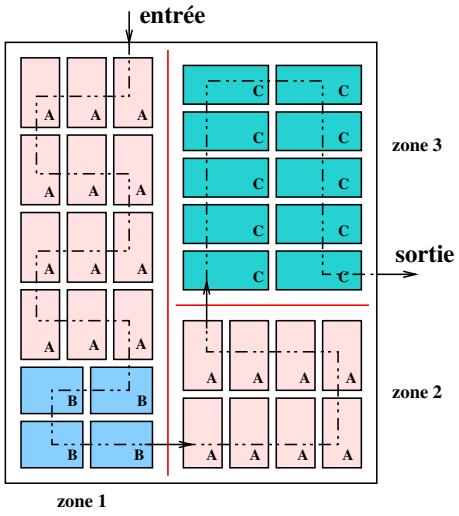


FIG. 1.15: Illustration d'un placement dans [45] : Le FPGA est décomposé en zones et chaque zone contient des processeurs de forme potentiellement différente, ici A, B et C.

Hétérogénéité dans les FPGAs Si un ilot de calcul suffit à représenter des eFPGAs, il n'est pas suffisant pour la représentation de FPGAs commerciaux actuels, c'est-à-dire dont la

structure n'est pas entièrement régulière et qui contiennent plusieurs types de ressources (processeurs, blocs DSP, etc.). Des modules potentiellement hétérogènes sont alors placés, ajoutant ainsi des contraintes de placement pour l'exploitation de ressources du FPGA spécifiques.

Afin de représenter cette hétérogénéité, Singhal et Bozorgzadeh [108] représentent les FPGAs en diverses couches, comme illustré à la figure 1.16. Chaque couche représente un type de ressources dans le FPGA : une couche pour les blocs logiques et configurables (CLBs), une couche pour les mémoires et une couche pour les blocs multiplieurs (DSPs). Cette décomposition en couches permet d'isoler les types de ressources contenues dans un module. Ainsi, lorsqu'un module est attribué à une partie du circuit implémenté, il est possible de déterminer quelles ressources de ce module sont réellement exploitées en fonction de leur type et de leur quantité. Si un module n'exploite pas un type de ressources qu'elle englobe, chacune de ces ressources peut alors être utilisée par d'autres modules. Les modules peuvent ainsi se superposer afin d'exploiter au mieux chaque ressource du FPGA.

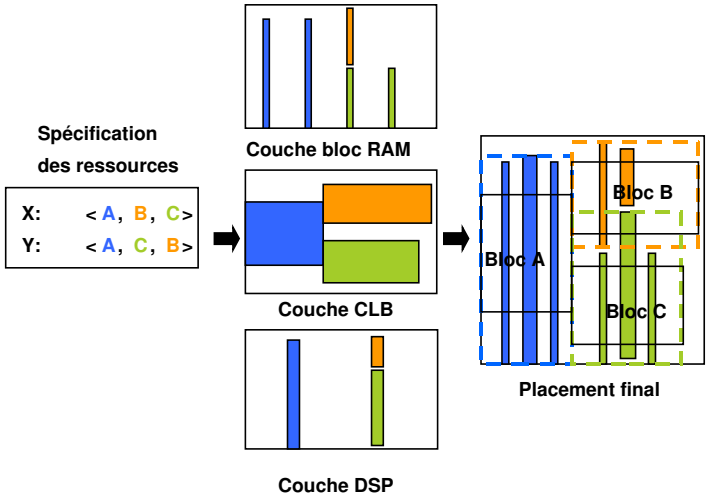


FIG. 1.16: Représentation des FPGAs en plusieurs couches proposée par Singhal et Bozorgzadeh [108]. Trois couches sont représentées sur cette figure et permettent de spécifier les positions de ces ressources : les blocs logiques et reconfigurables (CLBs), les mémoires et les multiplieurs. Les zones de placements peuvent lors contenir différents types de ressources d'implémentation.

Modélisation UML des FPGAs Le standard de modélisation MARTE [101] doit permet la représentation des placements d'architectures fonctionnelles (processeurs, mémoires, etc.) sur des supports physiques, tels que les FPGAs. Il n'est pas question d'heuristique de placement mais de spécification de placement par les concepteurs qui repose sur un *profil UML pour FPGAs*. La figure 1.17 représente le sous-ensemble de MARTE permettant la représentation des FPGAs. Un FPGA est représenté par le stéréotype HwPLD, il peut contenir de la mémoire RAM (HwRAM) et est caractérisé par sa technologie (*SRAM*, *Antifuse*, *Flash*, *Other*). L'organisation de cellules du FPGA est caractérisée par le nombre de lignes et colonnes, mais

aussi par le type de l'architecture : *SymetricalArray*, *RawBased*, *SeaOfGates*, *HierarchicalPLD* ou *Other*.

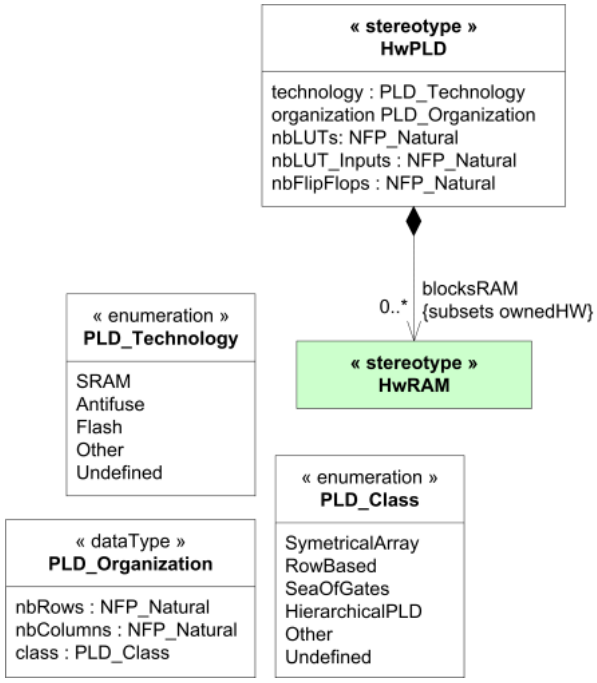


FIG. 1.17: Partie du profil MARTE [101] représentant la modélisation des FPGAs.

Bilan des placements hors ligne Les travaux présentés permettent de réaliser des placements hors ligne sur FPGA, il est donc nécessaire de connaître le comportement d'un circuit et les caractéristiques des tâches implémentées en fonction du FPGA ciblé. Les placements hors ligne ne sont soumis à aucune contrainte temps réel, les solutions sont par conséquent optimisées en quantité de ressources FPGA utilisées, mais sont toutefois statiques. Le placement en ligne pallie à ce défaut en calculant le placement lorsqu'un système est déjà implémenté sur FPGA. Nous présentons brièvement des travaux relatifs à ces placements en ligne dans la section suivante.

1.4.2.2 Placement en ligne

Tabero *et al.* [112] ajoute une troisième dimension à l'architecture de type ilot de calcul des FPGAs afin de prendre en compte leur reconfigurabilité dynamique. Le FPGA correspond alors à un cube dont l'une des dimensions est le temps. Dans cette représentation, chaque cellule est reconfigurable indépendamment des autres cellules et le placement bidimensionnel est enrichi de manière à prendre en considération d'éventuelles reconfigurations partielles et dynamiques. De nouvelles possibilités de placement apparaissent donc, le choix d'un placement est guidé par la caractérisation en temps d'exécution des tâches. L'heuristique de placement est aussi guidée par une projection dans le futur proche de la configuration du FPGA et des tâches qu'il exécute. La figure 1.18 illustre le placement de trois tâches lors d'un

placement statique sur deux dimensions et son placement lorsque le dynamisme du FPGA est pris en compte.

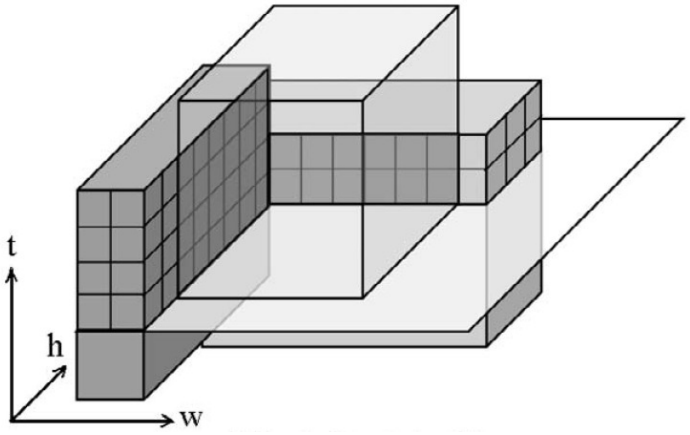


FIG. 1.18: Un placement de tâches sur trois dimensions proposé par [112], la dimension notée t représente le temps. Cette figure illustre l'ordonnancement ainsi que le placement des tâches sur FPGA.

Les placements en ligne sont en général gérés par une ressource extérieure au FPGA (alors considéré comme un esclave) et qui connaît à tout moment l'état de configuration de ce FPGA. Cette ressource peut-être un ordonnanceur matériel dans [112] ou un OS dans [123]

La principale différence entre un placement en ligne et un placement hors ligne est la présence de contraintes temps réel pour le placement en ligne. Les choix de placement se doivent d'être rapides afin de réduire les latences entre les différentes configurations. Par ailleurs, Resano *et.al* [102] décompose le placement en deux étapes : hors ligne puis en ligne. L'étape de placement hors ligne propose plusieurs ordonnancements des tâches et le placement en ligne choisit le placement le plus adéquat.

1.4.2.3 Bilan des représentations des FPGAs et des placements

Niveau de description des FPGAs La littérature fait état d'une grande richesse dans la représentation des FPGAs, la tendance de ces représentations s'oriente vers l'hétérogénéité [108] et se rapproche donc des FPGAs de type Stratix ou Virtex. En fonction de leur précision, les représentations permettent d'exprimer des placements voués ou non à être réalisés sur FPGA. La représentation très précise des FPGAs dans [16, 68] permet la génération directe de bitstream, tandis que [108] réalise un placement sur Virtex 4 par le biais d'un fichier de contraintes utilisateur. Par ailleurs, [123, 11, 112] ne présentent que des modèles théoriques. Ainsi, de l'objectif du placement (génération de bitstream, génération de contraintes de placement ou modèle théorique) dépend la précision avec laquelle les FPGAs sont représentés.

Le standard de modélisation MARTE [101] classifie les FPGAs en fonction du type d'organisation des cellules, la taille de la grille du FPGA, mais aussi par des informations technologiques, comme la technologie de configuration. Cependant, la description de FPGAs hétérogènes n'est pas possible et aucune information ne permet de définir les entrées sorties

du FPGA. De manière plus générale, la classification proposée est intéressante pour la modélisation à très haut niveau d'abstraction (où peu d'informations sont nécessaires pour la description du FPGA) mais n'est pas exploitable telle qu'elle pour la modélisation de placement.

Il ressort de ces travaux qu'une représentation détaillée des FPGAs permet de générer des configurations au prix d'une complexité de représentation élevée. Une représentation plus abstraite (et donc plus simple) est plus difficilement exploitable et il est nécessaire d'utiliser des fichiers de contraintes par exemple.

Caractérisation des implémentations Lorsque la dynamicité du FPGA n'est pas prise en compte, celui-ci est modélisé sous la forme d'une grille homogène de blocs logiques et configurables [16, 68], ou sous la forme de couches pour les grilles hétérogènes [108]. Les tâches placées sur ces FPGAs sont caractérisées par un nombre de ressources du FPGA consommées pour leur implémentation et-ou par la forme de la zone de placement [45, 116], rectangulaire le plus souvent.

En ce qui concerne les travaux permettant de prendre en considération la reconfiguration partielle et dynamique, les FPGAs sont représentés sous la forme d'un cube [112, 11] : deux dimensions pour l'espace et une troisième dimension pour le temps. Afin d'exploiter cette troisième dimension lors du placement, les tâches sont caractérisées par des temps d'exécution.

La caractérisation de l'implémentation d'une tâche sur un FPGA donné est réalisée par le biais d'une quantité de ressources, d'une surface ou d'un temps d'exécution. Ces informations sont capitales pour la définition d'un « bon » placement : un sous-dimensionnement de ces informations peut entraîner un échec de la synthèse, un surdimensionnement débouche sur un placement non optimal, car les ressources allouées peuvent ne pas être pleinement exploitées.

Représentation unifiée Un constat plus général est un manque de généricité et de communication (au sens *diffusion de l'information*) concernant la représentation des FPGAs, des caractéristiques d'implémentation de tâches et des placements. En effet, les mécanismes de représentation mis en œuvre dans ces différents travaux semblent très disparates et il est parfois difficile d'appréhender la forme sous laquelle est manipulé un placement par exemple. L'unification de ces représentations faciliterait à la fois le travail du concepteur de l'outil et celui de ses utilisateurs.

1.5 Systèmes sur puce, SoC

Un système sur puce (System on Chip en anglais, SoC) peut intégrer sur une seule et même puce des processeurs, des mémoires, des réseaux d'interconnexions, des circuits spécifiques (tels que des Digital Signal Processors DSP ou des accélérateurs matériels), etc. Parcequ'ils sont optimisés en conséquence, les SoCs exécutent efficacement des applications souvent gourmandes en puissance de calcul (radiocommunication, analyses des échos radars, etc.). Par ailleurs, leur intégration sur une seule puce permet un gain en place et en consommation d'énergie, ce qui vaut au SoC d'être couramment utilisé dans les systèmes embarqués.

La figure 1.19 représente le SoC utilisé dans le projet européen MORPHEOUS [85]. Ce SoC est composé de trois architectures reconfigurables (PiCoGA, eFPGA, XPP), de mémoires, de bus, de réseaux sur puce, etc. La complexité des SoCs les rend complexes à programmer et nécessite une approche de co-conception que nous détaillons ci-après.

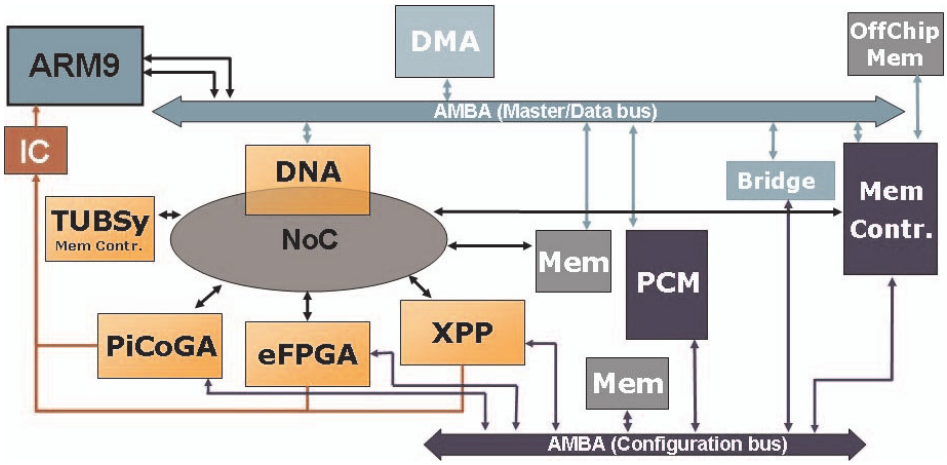


FIG. 1.19: L'architecture du SoC défini dans le projet MORPHEOUS [85].

1.5.1 Programmation des SoCs suivant une approche de co-conception

La programmation des systèmes sur puce est une tâche complexe qui est liée à l'hétérogénéité des systèmes et à la complexité croissante des applications traitées. La conception conjointe de logiciel et de matériel, ou co-conception, réduit l'effort de programmation car elle prend en considération chacun de ces éléments : l'application, l'architecture matérielle et le placement de l'une sur l'autre.

La co-conception est dérivée au fameux schéma en « Y » deGajski et Kuhn [51] présenté à la figure 1.20. Le haut de la figure représente la conception séparée de l'application et de l'architecture. Le centre représente le placement de l'application sur l'architecture et le bas correspond au résultat du placement : un code exécutable par exemple.

1.5.2 Défis de conception rencontrés par les SoCs

Avec l'évolution continuelle de la technologie des semi-conducteurs, une seule puce de silicium peut désormais contenir plusieurs centaines de millions de transistors [60]. Il va de soi que la capacité d'intégration évoluant, les SoCs se complexifient, tant dans la quantité de ressources qu'ils intègrent que dans leur hétérogénéité. Par ailleurs, la puissance de calcul réclamée par les applications ne cesse d'augmenter. Tout cela a un impact conséquent sur la conception des SoCs qui fait face aujourd'hui à de nombreux défis : un temps de mise sur le marché très court, le coût de la conception, la complexité du silicium, la productivité, etc.

1. **Temps de mise sur le marché :** Afin d'être réactifs aux publications de nouveaux standards (décodages audio par exemple) et de nouveaux besoins, les concepteurs de SoC n'ont que peu de temps pour concevoir et programmer leur système. L'objectif est de

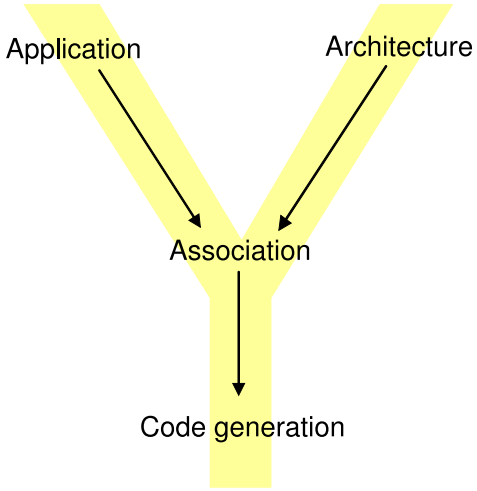


FIG. 1.20: Un schéma en « Y » basé sur le schéma en « Y » [51].

diffuser sur le marché le plus rapidement possible (avant les autres si possible) un produit répondant à une nouvelle norme. Cette contrainte de temps ne laisse que peu de place à une réelle exploitation des transistors disponibles sur une puce. L'exploitation de la plus grande partie des transistors disponibles sur une puce passe par un outillage qui permet d'augmenter la productivité des concepteurs.

2. **Coût de la conception :** La plus grande menace pour l'avenir du semi-conducteur est son coût de conception, qui se chiffre en dizaines de millions d'euros pour un nouveau SoC [21]. Or 80% du coût est lié à la réalisation du logiciel qui permet de concevoir le SoC. Il est donc urgent de proposer de nouvelles techniques et méthodologies pour la conception des SoCs qui permettront de réduire ce coût. Par exemple, l'unification des représentations d'un SoC facilite la collaboration entre les différents spécialistes intervenants dans sa conception.
3. **Complexité de la conception :** Les SoCs sont hétérogènes et intègrent de nombreuses ressources qui sont très diverses et qu'il convient de faire communiquer. Cela nécessite des connaissances approfondies de chacune des ressources par le concepteur du SoC. De plus, à la complexité des communications s'ajoutent les multiples fréquences d'horloge auxquelles ces ressources fonctionnent. Tous ces facteurs influent considérablement sur la complexité de la conception des SoCs.

1.5.3 Éléments de réponses aux problèmes de conception des Socs

La productivité reste l'élément majeur qui permet aux équipes de conception de SoCs de faire face à chacun de ces défis. Selon l'ITRS [60] (International Technology Roadmap for Semiconductors), cette productivité se traduit plus techniquement par les éléments suivants : la réutilisation, la vérification et le test, des plates-formes de conception et d'implémentation sûres et des processus de gestion de la conception. La pérennisation des outils reste cependant la principale préoccupation des développeurs d'environnements de co-conception, elle nécessite :

– la description dans un même langage de l'application et de l'architecture ;

– la réutilisation et la pérennisation des descriptions réalisées avec les outils de simulation ou synthèse actuels et à venir ;
– l'utilisation de formats standards pour l'échange, la sauvegarde et la réutilisabilité ;
– l'intégration dans un seul environnement de toute la chaîne de co-conception.

Afin de répondre à ces attentes, différentes approches ont vu le jour, nous présentons l'une de ces approches dans la section suivante.

1.6 Ingénierie Dirigée par les Modèles (IDM)

La compréhension d'un système complexe passe par une représentation abstraite et simplifiée de ce système : une *modélisation*. Un *modèle* met en évidence certaines caractéristiques du système en faisant abstraction notamment de ses détails d'implémentation. Plusieurs méthodes et langages ont permis par le passé de manipuler les modèles : Merise [114] (années 1970), SSADM (Structured Systems Analysis and Design Methodology) [44] (1980), UML (Unified Modeling Language) [90] (1995), etc. Actuellement, différentes approches permettent la modélisation d'un système, et cela en fonction du domaine d'utilisation. On retrouve parmi elles l'*Ingénierie Dirigée par les Modèles* (IDM ou MDE pour Model Driven Engineering [97][13]) qui est plus particulièrement utilisée pour la conception de modèles *productifs*, c'est-à-dire compréhensibles et interprétables par les machines [47]. De cette façon, l'IDM se démarque d'autres méthodologies de modélisation qui ont vocation à concevoir des modèles *contemplatifs*, c'est-à-dire non interprétables par les machines.

Nous présentons dans cette section l'aspect méthodologique lié à l'IDM et faisons référence tout au long de ce manuscrit à certains aspects technologiques et aux outils. Les trois concepts de base qui permettent de caractériser la méthodologie liée à l'IDM sont les *modèles*, les *métamodèles* et les *transformations*.

1.6.1 Modèles

Un modèle correspond à une abstraction de la réalité et en permet une représentation à l'aide de concepts et de relations entre ces concepts. L'IDM utilise les modèles tout au long du développement logiciel. Il est cependant important de noter que la notion de modèle apportée par l'IDM n'est pas nouvelle : Favre [46] considère que la notion de modèle remonte à plusieurs millénaires avec l'alphabet cunéiforme ougaritique (3400 av. J.-C.). Cette notion était déjà présente en informatique dans les années 1970.

Les modèles ne sont donc pas nouveaux mais reflètent une manière de penser que l'homme utilise ; l'IDM reprend cette manière de penser et l'applique à l'informatique.

1.6.2 Métamodèles

L'IDM recommande l'utilisation des modèles à différents niveaux d'abstraction. Un modèle représente une vue abstraite de la réalité et *est conforme* à un *métamodèle* qui définit précisément les concepts présents à ce niveau d'abstraction ainsi que les relations entre ces concepts. Un métamodèle permet donc de représenter des mécanismes complexes faisant

[13] Il existe des variantes de cette méthodologie telles que *Model-Driven Architecture* (MDA), *Model-Driven Development* (MDD), *Model Integrated Computing* (MIC) ou encore *Model-Driven Software Development* (MDSD). Nous ne présentons pas les différences qu'entraînent ces variantes mais nous intéressons à leurs racines communes.

intervenir plusieurs concepts. En IDM, un métamodèle spécifie en quelque sorte la syntaxe des modèles, de la même façon qu'un langage spécifie sa grammaire.

1.6.2.1 Niveaux M0, M1, M2 et M3

Le niveau M0 correspond à la réalité. Le niveau M1 correspond à une première abstraction de la réalité, c'est un modèle. Le niveau M2 correspond à une abstraction du modèle, c'est un métamodèle. Un métamodèle est aussi un modèle, il est par conséquent conforme à un autre métamodèle (niveau M3) qui est lui même conforme à un autre métamodèle, etc. En pratique, afin d'éviter une infinité de niveaux, le métamodèle du niveau M3 peut s'autodéfinir, à l'image des métamodèles (très similaires) Ecore [41] et MOF [89] (Meta-Object Facility) qui définissent entre autres les concepts d'objet, de classe, d'héritage, d'association, etc.

La figure 1.21 illustre les relations entre réalité (M0), modèles (M1), métamodèles (M2) et méta-métamodèle (M3) :

- M0 représente la réalité (un programme informatique). Sur la figure 1.21, les variables *Numéro* et *Solde* sont affectées par des valeurs ;
- M1 correspond au premier niveau d'abstraction, c'est un modèle et c'est lui que manipule un développeur. Pour cet exemple, le modèle contient la déclaration des variables utilisées dans M0 et la notion de *Compte*. Un modèle de niveau M1 est conforme à un métamodèle de niveau M2 ;
- M2 est le niveau dans lequel sont définis les concepts manipulés dans un modèle de niveau M1 (c'est-à-dire les concepts manipulés par un développeur)[14]. Dans cet exemple, *Compte* est une classe tandis que les déclarations de variables sont des *Attributs* de cette *Classe*. *Classe* et *Attributs* sont tous deux des concepts du métamodèle de niveau M2, qui est conforme à un métamodèle de niveau M3 ;
- M3 correspond au plus niveau d'abstraction, un méta-métamodèle de niveau M3 est conforme à lui même. Dans cet exemple, les concepts *Classe* et *Attributs* sont des *Méta-Classes*, tandis que la notion de contenance est une *Méta-Relation*. Ce méta-métamodèle se décrit lui-même, la *Méta-Classe* et la *Méta-Relation* sont des *Méta-Classes* et les relations *source* et *destination* qui les lient sont des *Méta-Relations*.

1.6.2.2 Exemple de métamodèle

Nous présentons dans cette section un exemple de métamodèle de niveau M2. Nous familiarisons ici le lecteur avec des notations et des termes que nous retrouvons dans l'ensemble de ce document.

La figure 1.22 représente un ensemble de concepts et les relations entre ces concepts. Les concepts manipulés sont des METACLASS et les relations sont des liens de *composition*, de *référence* ou d'*héritage*. Le concept COMPONENT est abstrait (le nom d'un concept abstrait est indiqué en italique) et peut être spécialisé en concept COMPOUND, REPETITIVE et ELEMENTARY car ils *héritent* de COMPONENT (lien d'héritage sur la figure). Un composant *contient* des concepts PORT (lien de composition sur la figure), chaque port *référence* un concept DATATYPE (lien de référence sur la figure).

[14] UML est un métamodèle de niveau M2.

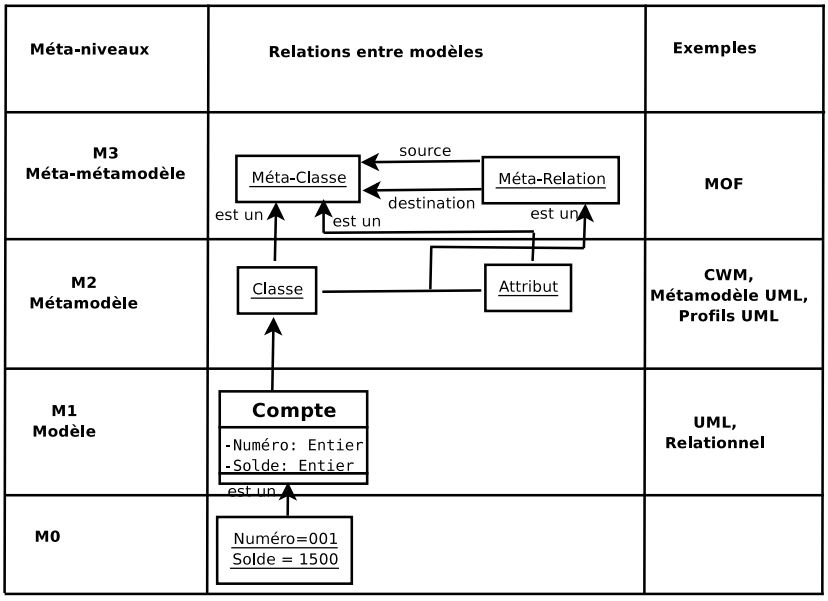


FIG. 1.21: Les différents niveaux de modélisation.

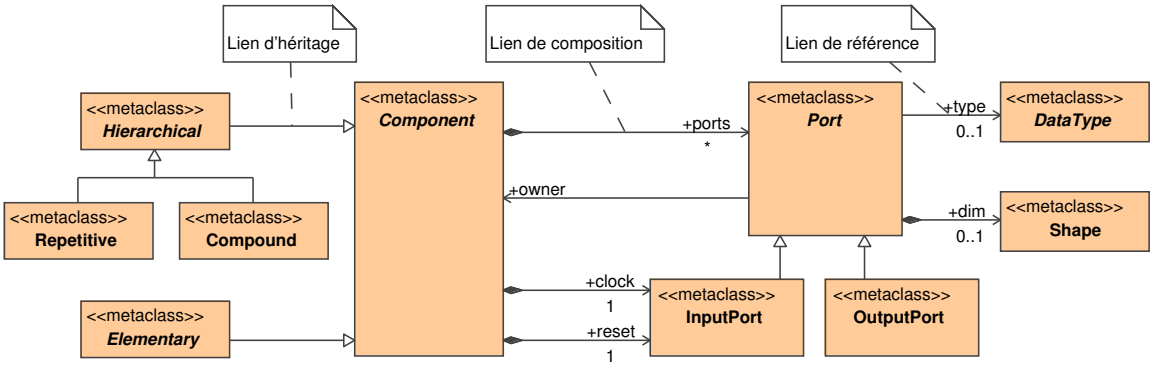


FIG. 1.22: Exemple de métamodèle de niveau M2.

1.6.2.3 Composition de vues

Chaque modèle peut être représenté selon différents *points de vue*, c'est-à-dire selon un angle sous lequel le modèle est observé : on n'en voit qu'un sous-ensemble ; on ne voit alors qu'une partie de la réalité représentée par le modèle. Ce découpage en vues permet de contextualiser la représentation de différents concepts en fonction d'une intention particulière (c'est-à-dire ce que la vue tend à montrer). Cela permet, par exemple, de représenter sur une vue tous les détails techniques associés à un concept et de représenter sur une autre vue l'utilisation de ce concept dans un contexte plus général. La figure 1.23 représente le métamodèle illustré précédemment mais sous la forme de différentes vues. Sur la gauche de la figure, l'élément central est COMPONENT, les autres concepts sont immédiatement perçus

comme des raffinements. En revanche, le concept central sur la droite de la figure est PORT.

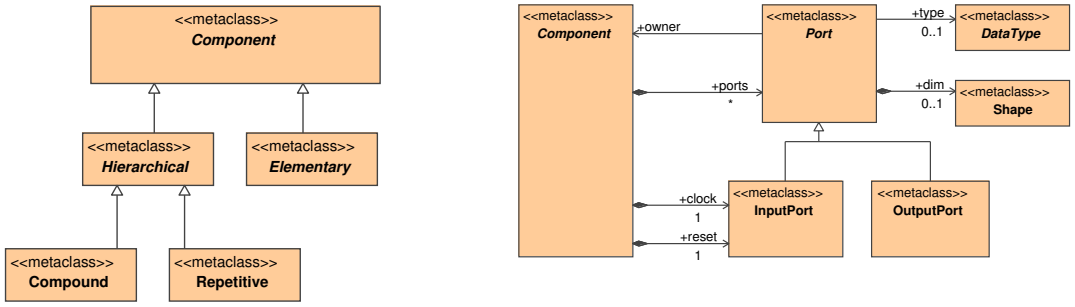


FIG. 1.23: Deux vues d'un même modèle : on ne représente que les concepts nécessaires à la compréhension de ce que la vue tend à montrer.

1.6.2.4 Flot de conception

En règle générale, un flot de conception guidé par l'IDM utilise des métamodèles qui permettent la description d'un système à un très haut niveau d'abstraction, des métamodèles *cibles* qui en permettent une description très fine (proche du niveau de description d'un code logiciel) et des métamodèles *intermédiaires* entre ces différents niveaux d'abstraction[15]. Le métamodèle de plus haut niveau d'abstraction ne contient pas d'information sur les aspects technologiques, qui sont introduits au fur et à mesure par les niveaux intermédiaires et cela jusqu'aux métamodèles cibles. Les métamodèles cibles marquent la sortie de l'environnement IDM lorsqu'ils sont utilisés pour une génération de code. Cette génération de code n'implique toutefois pas que ce métamodèle soit dépendant d'une syntaxe particulière ; simplement, le niveau d'abstraction est suffisamment proche de ce que tend à décrire le code en question pour en générer la syntaxe. Il est donc possible d'observer dans des flots de conception plusieurs générations de code à partir d'un même métamodèle.

Le passage d'un métamodèle à un autre métamodèle est réalisé par une *transformation de modèles*, qui est l'équivalent de la compilation dans le monde logiciel classique.

1.6.3 Transformations de modèles

Un autre point clé de l'IDM est la *transformation de modèles*, elle permet de passer d'un modèle *source* décrit à un certain niveau d'abstraction à un modèle *destination* décrit éventuellement à un autre niveau d'abstraction. Ces modèles source et destination sont conformes à leur métamodèle respectif (métamodèles source et destination) et le passage de l'un à l'autre (*i.e.* la transformation) est décrit par des *règles de transformation*. Ces règles sont exécutées sur les modèles source afin de générer les modèles destination, comme illustré à la figure 1.24.

1.6.3.1 Différents types de transformations

Tom Mens *et al.* [84] distinguent deux catégories de transformations : *exogène* et *endogène* selon la source et la destination d'une transformation de modèles. Une transformation est

[15]Il est nécessaire de préciser ici que tous ces métamodèles sont de niveaux M2.

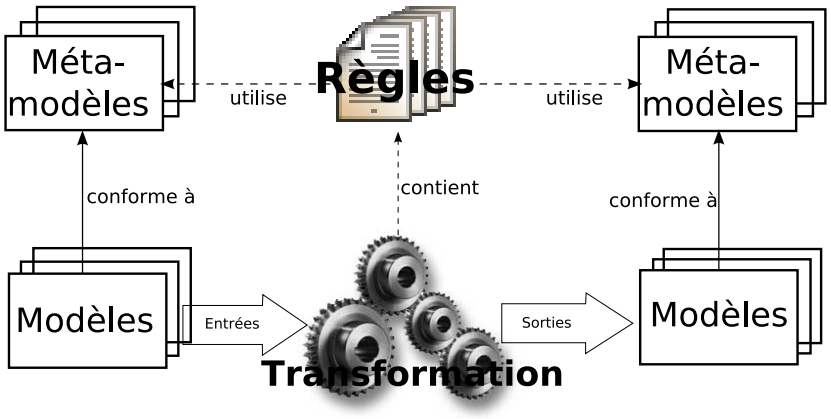


FIG. 1.24: Une transformation de modèles qui permet de passer d'un modèle source en un modèle destination par l'exécution de règles de transformation.

endogène lorsque le métamodèle source est le même que le métamodèle de destination, cela peut être assimilé à une optimisation ou à une restructuration du modèle transformé. Une transformation est exogène dans le cas contraire, c'est-à-dire lorsque les métamodèles source et destination sont différents. Deux sous-catégories sont alors définies et correspondent aux transformations *horizontales* lorsque le niveau d'abstraction des métamodèles source et destination est le même et *verticales* dans le cas d'un processus de raffinement dans un flot de conception.

Nous avons jusqu'à présent utilisé les notions de source et de destination pour orienter les transformations de modèles. On trouve cependant dans la littérature [128, 111] des travaux qui permettent la bidirectionnalité d'une transformation. Les règles qui composent cette transformation sont applicables dans un sens comme dans l'autre, mais sont toutefois sujettes à certaines contraintes afin de garantir la cohésion dans cette bidirectionnalité (une somme de deux nombres ne peut être bidirectionnelle par exemple).

La traçabilité dans une transformation de modèle consiste à retrouver dans le modèle source un concept ayant servi à la création d'un concept dans le modèle de destination. Pour cela, une trace sur l'exécution d'une transformation est conservée [4]. La traçabilité à un avenir certain pour le débogage car elle permet de retrouver en remontant dans un flot de conception le ou les concepts qui entraînent un bogue à un niveau d'abstraction donné. Cependant, la traçabilité n'est que peu supportée par les outils de transformation.

1.6.3.2 Règles de transformation

Une règle de transformation définit la manière dont un ensemble de concepts du métamodèle source est transformé en un ensemble de concepts du métamodèle destination. Pour arriver à cela, il existe les langages impératifs qui décrivent *comment* la règle est exécutée et les langages déclaratifs qui décrivent *ce qui* est créé par la règle. Cette dernière méthode est plus utilisée en IDM que la précédente car elle permet de spécifier l'intention (le but) de la règle indépendamment de son exécution.

Du point de vue du fonctionnement, une règle peut se décomposer en trois étapes que sont la vérification de la condition, l'exécution et la création. La première étape correspond à l'analyse du modèle source de la transformation de manière à détecter la présence d'un en-

semble de concepts qui correspond à ce que la règle « attend » en entrée. Chaque ensemble de concepts qui valide la condition permet l'exécution de la règle par le moteur de transformation. Celui-ci sauvegarde à l'aide de variables des éléments relatifs à cet ensemble de concepts et lance l'exécution d'éventuelles sous-règles. La dernière étape, la création, consiste à générer un ensemble de concepts dans le modèle de sortie dont les champs sont remplis par les variables affectées durant l'exécution. L'agencement et la structuration des règles exécutées dans une transformation permettent de reconstruire l'ensemble du modèle de sortie.

La décomposition en règles de la transformation permet d'isoler les groupements de concepts qui sont utilisés : seuls les concepts utilisés dans une règle sont représentés et/ou manipulés. Cela favorise non seulement la clarté d'une règle, mais aussi celle de l'ensemble de la transformation de modèles. De plus, cette séparation en règle augmente la réutilisation des développements, car certaines règles peuvent être utilisées plusieurs fois dans une même transformation.

1.6.3.3 Moteurs d'exécution des règles de transformation

Différents outils sont développés pour la description et l'exécution des règles de transformation (Kermeta [119, 86], ATL [62], ModelMorf [34], etc.) mais seul QVT [93] (Query, View, Transformations) est standardisé par l'OMG. QVT permet la représentation graphique et la description textuelle des règles de transformation. Ce standard n'est cependant que peu utilisé car les outils qui le supportent permettent uniquement l'exécution de la partie impérative [16] qui ne repose pas sur la notion de règle. Une alternative stable et répandue à ce standard peu utilisé est EMF [41] (Eclipse Modeling Framework) qui est une bibliothèque Java permettant de créer et modifier des modèles. Les règles y sont toutefois décrites à un très bas niveau.

1.6.4 L'IDM en pratique

Le succès de l'IDM repose avant tout sur les concepts fondamentaux que nous venons d'évoquer. Aussi, des langages et des standards pour la modélisation unifiée de ces concepts fondamentaux ont permis une large diffusion de l'IDM auprès de la communauté informatique.

1.6.4.1 UML pour la modélisation

Le métamodèle UML [90] (Unified Modeling Language) a été standardisé en 1997 par l'OMG et est très répandu dans la communauté modèle. Ce métamodèle n'est toutefois pas directement exploitable pour l'IDM (qui tend à utiliser des métamodèles productifs) car la sémantique d'UML n'est pas suffisamment précise, malgré des tentatives de clarification [106]. UML reste toutefois utilisé en IDM sous forme de *profils* qui correspondent à des extensions du métamodèle UML. Un profil est composé de *stéréotypes* qui permettent de spécialiser des classes UML et de *tagged values* qui permettent d'ajouter des attributs à ces classes.

L'intérêt majeur de l'utilisation du métamodèle UML reste donc que les concepts de base à la métamodélisation sont déjà présents ; un profil permet d'étendre le pouvoir de repré-

[16] http://smartqvt.elibel.tm.fr/

sentation d'UML en fonction de ce que l'on souhaite modéliser. Le principal inconvénient reste que les modèles réalisés avec ce profil ne sont pas conformes au métamodèle mais à son implémentation en UML : une transformation de l'un vers l'autre est nécessaire.

Un point clé du succès de l'utilisation de UML en IDM est que de nombreux outils supportent UML et son mécanisme de profil. Ces outils permettent une modélisation simple et immédiate sous forme de représentations graphiques. Bon nombre de standards de modélisation pour l'IDM voient donc le jour sous la forme d'un profil UML.

1.6.4.2 Profils standards

SysML [92] (System Modeling Language) est un standard OMG qui fournit des mécanismes génériques pour la description de systèmes complexes. SysML n'apporte cependant pas une attention particulière aux systèmes embarqués, comme le fait le profil pour la modélisation et l'analyse des systèmes embarqués et temps réel MARTE (Modeling and Analysis of Real-Time and Embedded systems) [101]. MARTE fournit entre autres des concepts précis pour la description des détails d'architectures matérielles dans les SoCs. Dans le cas de similitudes entre des concepts du profil SyML avec des concepts du profil MARTE, MARTE réutilise les stéréotypes définis par SysML ou définit des éléments qui sont conceptuellement et terminologiquement alignés sur SysML. MARTE raffine les concepts de temps proposé dans le profil SPT (Scheduling, Performance and Time) [91] ce qui permet la modélisation de temps logique, discret et continu. Les modèles conformes au profil MARTE peuvent être annotés par des propriétés non fonctionnelles dans le but d'exprimer des contraintes auxquelles le système est soumis (contraintes de temps, de consommation d'énergie, etc.)

Le profil MARTE, en finalisation de standardisation à l'OMG, permet la représentation factorisée de structures répétitives, facilitant la représentation d'architectures régulières. La factorisation dans le profil MARTE est le résultat de la contribution de notre équipe à la proposition de ce standard. MARTE permet par ailleurs la représentation d'applications et de leurs placements sur des architectures matérielles. Pour toutes ces raisons, le profil Gaspard utilisé dans l'environnement Gaspard que nous décrivons dans ce chapitre repose sur le profil MARTE[17].

1.6.4.3 Quid de l'IDM pour les accélérateurs et langages HDL ?

Nous nous intéressons dans cette section aux travaux guidés par l'IDM qui permettent la génération de code HDL. Nous élargissons volontairement notre éventail de recherche, initialement centré autour des accélérateurs matériels, dans le but d'obtenir une idée précise de l'avancée des travaux qui visent à définir un « pont » entre le haut niveau d'abstraction et le niveau RTL.

Damasevicius et Stuikye [30] manipulent des IPs matériels sous la forme de classes UML dont les attributs correspondent aux ports des IPs. Ces IPs sont utilisés par composition, ce qui rend les modèles d'application complètement dépendant des IPs utilisés. Ils proposent par ailleurs un métamodèle VHDL dans lequel sont définis, entre autres, les concepts d'entité, d'architecture, de process du langage VHDL. Ce métamodèle est dépendant de la syntaxe VHDL et ne peut, par conséquent, être réutilisé pour d'autres langages HDL. Un processus de mappage identifie les équivalents entre les éléments des diagrammes de classes en UML et les concepts du langage VHDL, l'implémentation de cette traduction est réalisée au

[17] En pratique, le profil Gaspard est en voie de « Martification » car la standardisation de MARTE est récente.

moyen d'un script exécuté dans le modeleur UML UMLStudio [18]. Le code est donc directement généré depuis le diagramme de classe en UML, ce qui introduit, à nouveau, une forte dépendance avec le langage VHDL. Au final, la description d'un métamodèle dépendant de la syntaxe VHDL et son analyse pour la création de la génération de code VHDL directement depuis depuis un diagramme de classe UML rendent ces travaux difficilement réutilisables pour un autre langage ou un autre métamodèle.

Il n'y a pas non plus de raffinement dans les travaux de Coyle et Thornton [28] qui parsent directement le XML généré depuis un modeleur UML afin d'en générer le code VHDL. Il est ainsi possible de générer le code VHDL de machines à états finis modélisées dans les diagrammes d'états UML. Par ailleurs, l'utilisation du moteur de transformation MODCO garantit une certaine indépendance vis-à-vis du modeleur UML, contrairement à Damasevicius et Stuikye [30] qui écrivent les scripts de génération de code dans une syntaxe dépendante du modeleur UMLStudio.

D'autres travaux permettent la génération de code VHDL depuis des modélisations UML de machine à états finis. Parmi eux, McUmber et Cheng [83] reprennent l'idée de mapper les concepts VHDL sur le métamodèle UML. Akehurst *et al.* proposent « d'implémenter simplement des transformations simples » avec le moteur de transformation SiTra [5]. Cette méthodologie est appliquée pour la transformation d'une machine à états finis en un modèle conforme à un métamodèle VHDL [6], à partir duquel le code est généré par l'intermédiaire de transformations s'appuyant sur des templates. L'utilisation successive d'une transformation de modèles et d'une génération de code décompose la chaîne de transformations en deux « petites » étapes, plutôt qu'en une seule « grande » étape pour les travaux de McUmber et Cheng [83] ou encore de Coyle et Thornton [28]. Cependant, l'utilisation d'un métamodèle VHDL ne permet toujours pas une généricité suffisante pour la génération de code d'un autre langage HDL.

Björklund et Lilius [18] utilisent un langage de description intermédiaire des automates. Un automate décrit à ce niveau est généré depuis les statecharts de UML. Cette description intermédiaire permet d'optimiser l'automate puis de générer le code VHDL avec un outil existant. Nous notons donc l'intérêt de ne pas générer directement le code HDL, mais de repousser cette échéance afin de profiter d'outils existants.

Les travaux concernant la génération de code VHDL à l'aide de l'IDM nous amènent à deux constats :

- le premier concerne la définition et l'utilisation quasi systématique d'un « métamodèle VHDL »[19]. Ce métamodèle dépend de la syntaxe VHDL et ne peut, par conséquent, être utilisé pour la génération de toute autre syntaxe HDL (Verilog par exemple). Cette dépendance immédiate avec une syntaxe dans une chaîne de transformations est contraire au raffinement en IDM, qui préconise un ajout d'informations dans une transformation qui dépendent de la technologie d'implémentation ;
- le second constat concerne la limitation de ces outils aux exemples de machines à états finis. D'une part, les exemples traités sont minimalistes, d'autre part les machines à états finis sont à l'opposé du traitement du signal systématique. En effet, une machine à états finis décrit l'évolution au cours du temps d'un état en fonction d'une succession d'événements, tandis que le traitement du signal systématique est caractérisé par des calculs réguliers appliqués systématiquement et qui ne dépendent pas des valeurs des

[18] http://www.pragsoft.com/

[19] Il n'existe pas de métamodèle VHDL standard.

données.

L'IDM pour la génération d'accélérateurs matériels dédiés au traitement du signal systématique est une voie encore inexplorée.

1.7 Environnement de co-conception pour SoC Gaspard

L'environnement Gaspard (Graphical Array Specification for Parallel and Distributed Computing) [31] reprend le schéma en « Y » de Gajski et Kuhn [51] afin de générer différents codes à partir d'un même placement d'une application sur une architecture.

La génération de langages synchrones déclaratifs [15] (tels que Lustre [25] ou Signal [73]) à partir de l'environnement Gaspard permet de vérifier formellement la modélisation d'une application. La génération de langages procéduraux tels que Fortran/OpenMP rend possible l'exécution concurrente de différents processus sur une architecture multiprocesseur (architectures à mémoire partagée dans l'état actuel de Gaspard). Enfin, la génération de code SystemC permet la simulation du comportement d'un SoC à différents niveaux d'abstraction.

Un mécanisme de déploiement permet de relier les composants atomiques de l'application et de l'architecture avec des IPs en bibliothèques. Il est possible d'utiliser des IPs pour les différents langages cibles de façon indépendante aux représentations de l'application et de l'architecture.

Gaspard offre la possibilité de modéliser sous une forme factorisée des applications et des architectures régulières. L'expression du parallélisme est basée sur ARRAY-OL (que nous avons présenté dans la section 1.3), étendu en conséquence pour la modélisation des architectures [29] et du placement. Ainsi, l'expression factorisée du parallélisme potentiel de l'application et de l'architecture est décrite de la même façon.

L'ensemble de l'environnement Gaspard est construit dans une démarche IDM, de la modélisation en UML jusqu'à la génération de code.

1.7.1 IDM dans Gaspard

La figure 1.25 représente l'implémentation de l'environnement Gaspard en IDM. Chaque boîte représente un métamodèle et les flèches entre ces boîtes des transformations de modèles. Le métamodèle Deployed est celui de plus haut niveau d'abstraction à partir duquel différentes transformations permettent de raffiner un modèle jusqu'à la génération de code. À partir d'un modèle conforme au métamodèle Deployed, l'environnement Gaspard peut générer plusieurs implémentations de ce modèle pour différentes plateformes d'exécution.

1.7.2 Métamodèle Deployed

Le métamodèle Deployed est une composition de plusieurs métamodèles (application, architecture, association et déploiement) comme illustré à la figure 1.26.

1.7.2.1 Métamodèles de spécification de Gaspard

Le métamodèle d'*application* regroupe différents concepts permettant la modélisation des applications Gaspard. Le métamodèle d'*architecture* permet la conception de l'architecture à

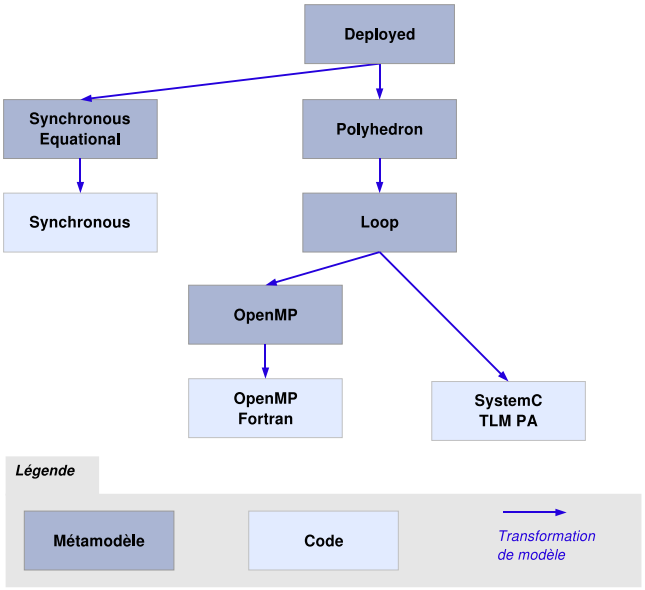


FIG. 1.25: L'environnement Gaspard et ses différents métamodèles.

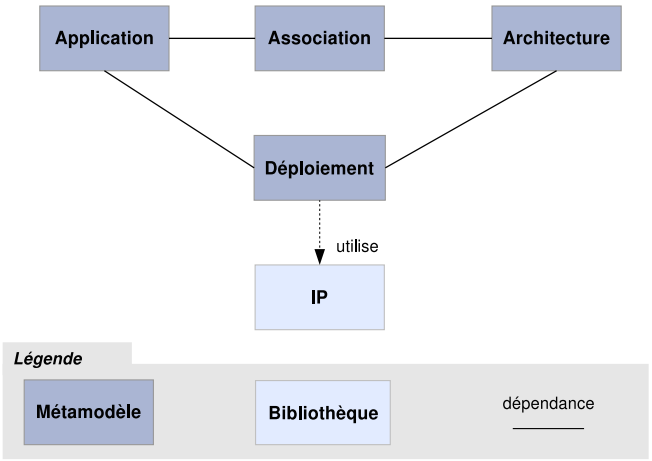


FIG. 1.26: Les différents métamodèles de saisie de l'environnement Gaspard et leurs dépendances.

l'aide de composants matériels, qui sont des abstractions de composants physiques. Le métamodèle d'*association* permet la description du placement des tâches et des données d'une application sur des composants de l'architecture : il permet de spécifier par quel composant matériel est exécutée une partie de l'application, ou dans quelle mémoire est stocké un ensemble de données. Pour cela, l'association importe l'application et l'architecture.

L'expression du parallélisme dans l'application est sans doute la plus proche de la repré-

sentation ARRAY-OL initiale décrite dans la section 1.3. Les besoins de représentation des interconnexions dans l'architecture et des placements de données et de tâches dans l'association ont amené des extensions de ARRAY-OL [22, 19, 113]. L'une d'elles consiste à combiner deux tilers sur un même connecteur de manière à exprimer des topologies de connexions plus complexes que celles exprimables par un seul tiler [19].

Les métamodèles d'application, d'architecture et d'association ne dépendent d'aucune technologie. La technologie est introduite par le métamodèle de *déploiement*, permettant par exemple de spécialiser une exécution sur une grille de processeurs ou encore une simulation SystemC. Le déploiement correspond en partie à la création d'un lien entre un composant (application ou architecture) et un IP disponible en bibliothèque. Le déploiement d'une application et d'une architecture marque en général la fin de la conception d'un système, le reste étant automatisé.

1.7.2.2 Modélisation dans Gaspard via un profil UML standard

Un modèle Deployed (conforme au métamodèle Deployed détaillé précédemment) est modélisé en UML à l'aide du profil Gaspard [14]. Ce profil reprend les concepts du métamodèle Deployed qui n'apparaissent pas dans le métamodèle UML. Une transformation de modèles (que nous ne représentons pas) permet de générer un modèle Deployed à partir d'une telle modélisation UML : UML est donc l'interface de saisie utilisée pour la conception d'un modèle Deployed. L'utilisation d'UML ouvre la voie à la standardisation de la conception de systèmes embarqués car le profil Gaspard correspond à un sous-ensemble du standard OMG MARTE [101].

1.7.3 Exemples de modélisation

Cette section est destinée à familiariser le lecteur avec les modélisations UML utilisées dans l'environnement Gaspard.

1.7.3.1 Parallélisme de tâches

Dans cette section, nous fournissons des modélisations à l'aide du profil Gaspard des exemples d'applications ARRAY-OL présentés dans la section 1.3. Le premier exemple concerne l'expression du parallélisme de tâches que nous illustrons à la figure 1.27. Le composant T est composé des instances de composants $t1$, $t2$, $t3$ et $t4$ (ces instances de composants correspondent à des tâches) et les dépendances de données sont modélisées au moyen de connecteur. Les ports des tâches permettent de spécifier la dimension des tableaux (le nombre de valeurs dans le vecteur) et sa taille (les valeurs dans le vecteur). Ainsi, la tâche $t1$ consomme un tableau bidimensionnel de taille $[(4, 2)]$ et produit un tableau tridimensionnel de taille $[(2, 2, 2)]$. Les ports de l'application sont spécialisés en port d'entrée ou de sortie, le modeleur ULM modifie en conséquence la couleur des ports : bleu pour une entrée et rouge pour une sortie. Un port de couleur neutre représente une entrée-sortie, ce concept est couramment utilisé dans l'architecture mais est interdit dans l'application.

1.7.3.2 Parallélisme de données

L'exemple de la multiplication de matrices présenté précédemment dans ce document nous permet d'illustrer la modélisation du parallélisme de données, figure 1.28. Les ports

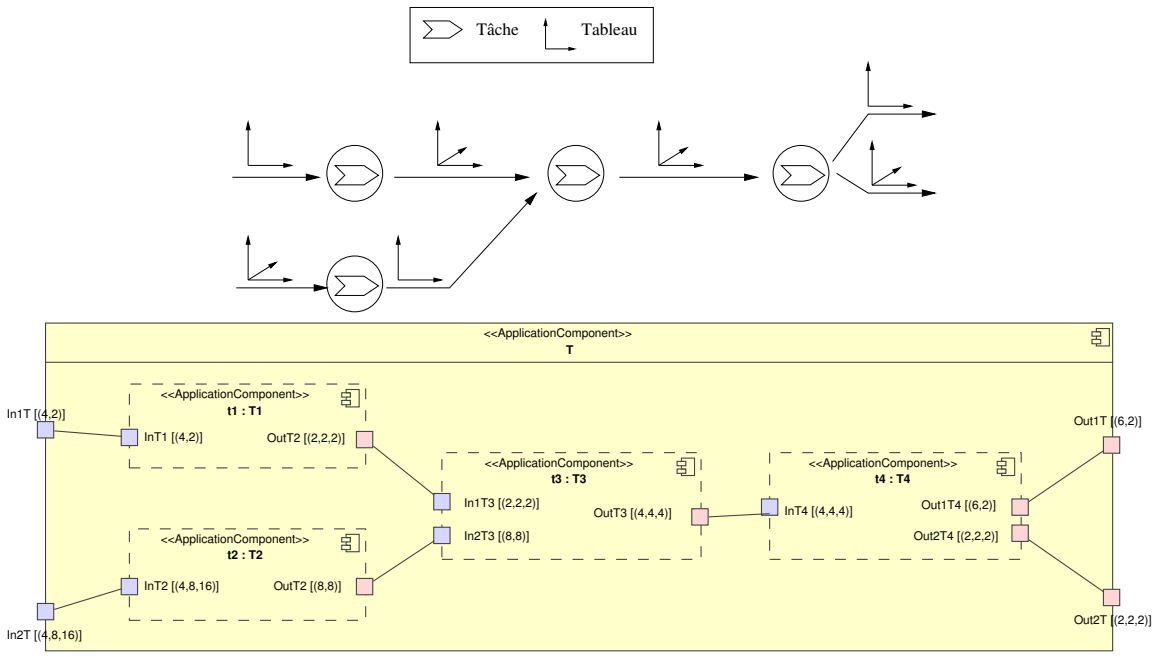


FIG. 1.27: Représentation en ARRAY-OL du parallélisme de tâches dans une application et sa modélisation dans le profil Gaspard.

$A1$, $A2$ et $A3$ du composant $MultiplicationMatrice$ représentent les matrices $A1$, $A2$ et $A3$ de la multiplication de matrices et sont par conséquent bidimensionnels. La taille des matrices est spécifiée par la taille des ports : $[(5,3)]$ pour $A1$, $[(2,5)]$ pour $A2$ et $[(2,3)]$ pour $A3$. L'instance de composant te du composant $ProduitScalaire$ possède un espace de répétition de dimension $[(2,3)]$ et *est assimilée à une tâche de l'application modélisée*. Les ports de ce composant sont les motifs de la tâche ARRAY-OL et sont donc monodimensionnel et de taille $[(5)]$ ou unitaire (lorsqu'un motif ou tableau n'est composé que d'une donnée, il est possible de ne pas indiquer de taille, c'est le cas pour le résultat du produit scalaire dans notre exemple). Les tilers sont représentés sous la forme de connecteurs stéréotypés $\ll$Tiler$\gg$ et possédant les attributs ORIGIN, PAVING et FITTING[20] (il est aussi possible d'autoriser des accès cycliques dans les tableaux par le biais de l'attribut MODULO). Les tilers qui construisent les motifs d'entrées permettent un parcours des lignes pour la matrice $A1$ et un parcours des colonnes pour la matrice $A2$. Le tiler de sortie construit la matrice de sortie point après point. Les matrices représentées sous la forme d'une chaîne de caractères en UML correspondent aux matrices :

- tiler pour $A1$: $o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $P = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$ et $F = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$;
- tiler pour $A2$: $o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $P = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$ et $F = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$;
- tiler pour $A3$: $o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $P = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ et $F = \begin{pmatrix} \\ \end{pmatrix}$.

[20] Nous retrouvons les notions d'origine, de pavage (PAVING) et d'ajustage (FITTING) du modèle de calcul ARRAY-OL.

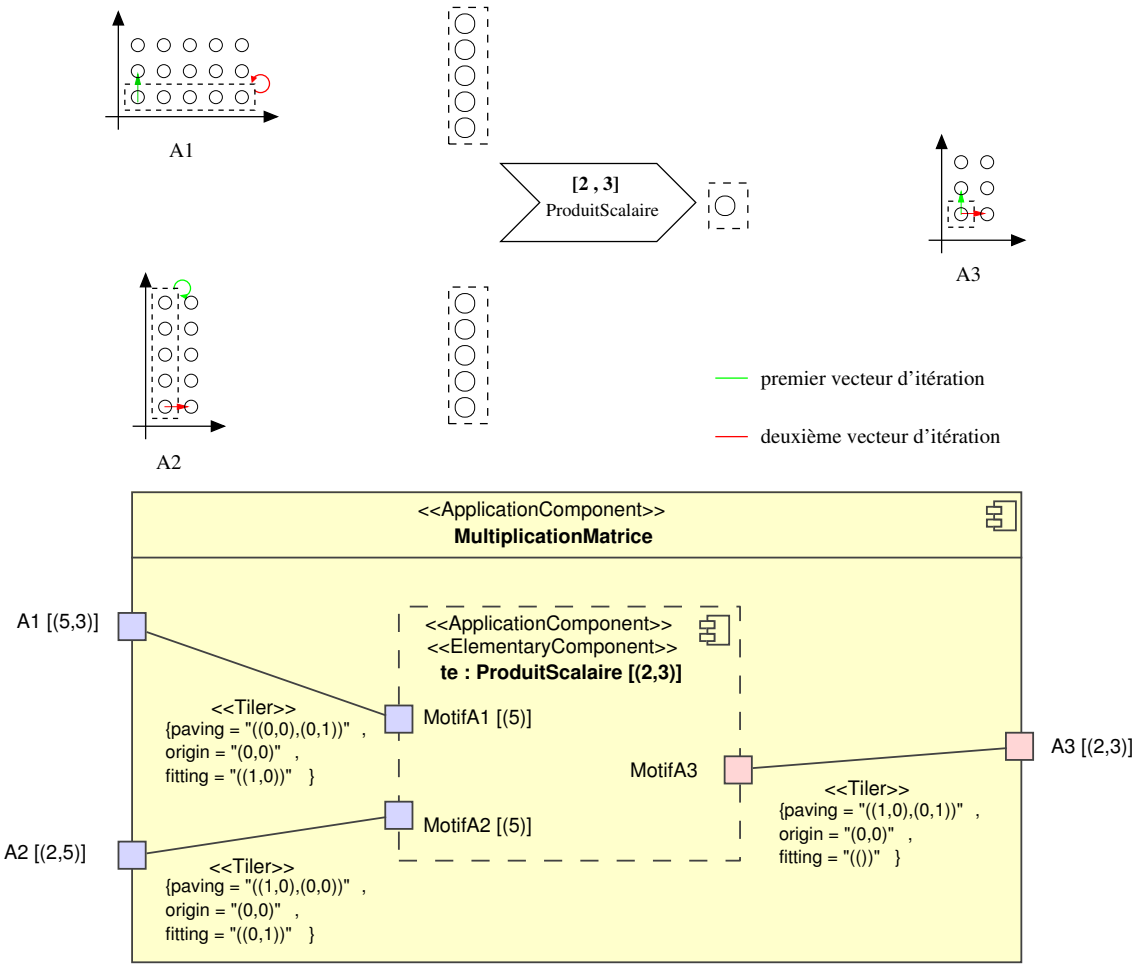


FIG. 1.28: Représentation en ARRAY-OL du parallélisme de données contenu dans une multiplication de matrice et sa modélisation en UML avec le profil Gaspard.

1.7.3.3 Modélisation d'une application, d'une architecture et du placement de l'une sur l'autre

Le bas de la figure 1.29 représente le composant $RepProcessingUnit$ dans une architecture matérielle. Ce composant contient deux ports connectés à quatre instances de composant $ProcessingUnit$ par l'intermédiaire de tilers (l'expression de la factorisation pour l'application est le même que pour l'architecture). Le haut de cette figure représente une application de multiplication de matrices de taille $[(2000, 2000)]$ (ce n'est donc pas la même application que celle modélisée précédemment). Le calcul de la multiplication est décomposé en 4×4 calculs différents, cela correspond à l'espace de répétition $[(4, 4)]$ sur la tâche dpB.

L'association entre cette application et cette architecture est matérialisée au centre de la figure par le connecteur stéréotypé ≪TaskAllocation≫ et ≪Distribution≫[21]. Le rôle de ce

[21] L'association permet par ailleurs un placement des données de l'application sur les mémoires de l'architec-

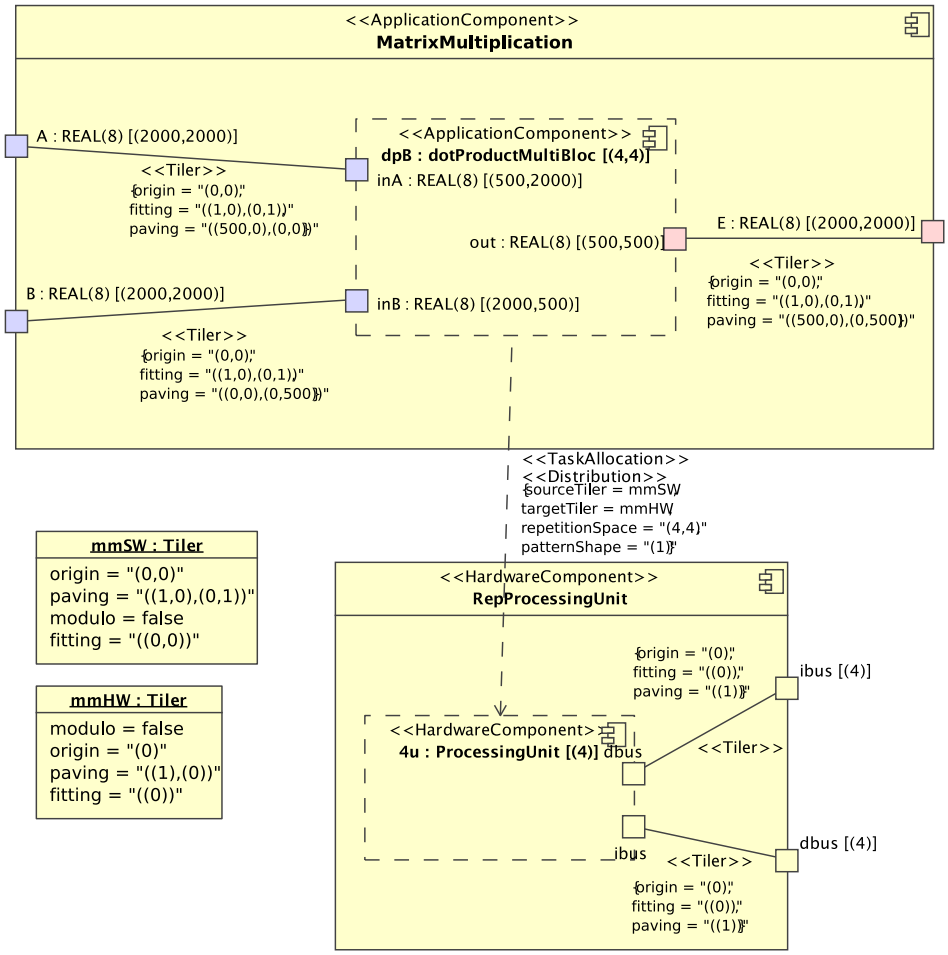


FIG. 1.29: Modélisation UML avec le profil Gaspard de l'expression ARRAY-OL du parallélisme de données dans une multiplication de matrices.

connecteur est de placé les 4×4 tâches de l'application (qui correspondent à l'instance de composant répété 4×4 fois dans le modèle) sur les 4 processeurs de l'architecture. La spécification du placement est définie au niveau de la source du connecteur (les tâches à placer) et au niveau de sa destination (les processeurs dans l'architecture) par des tilers (en bas et à gauche de la figure 1.29).

Le résultat de cette association est représenté à la figure 1.30, les 4×4 tâches de l'application sont sur la gauche, les 4 processeurs sur la droite. Chaque processeur est associé à une couleur, la couleur d'une tâche définit le processeur qui exécute cette tâche : un placement de tâches en colonnes qui est réalisé dans cet exemple.

1.7.3.4 Modélisation du déploiement

L'application, l'architecture et l'association sont indépendantes de toute plateforme d'exécution. La dépendance envers une plateforme est définie dans le déploiement qui

ture, nous ne représentons pas de tels placements ici.

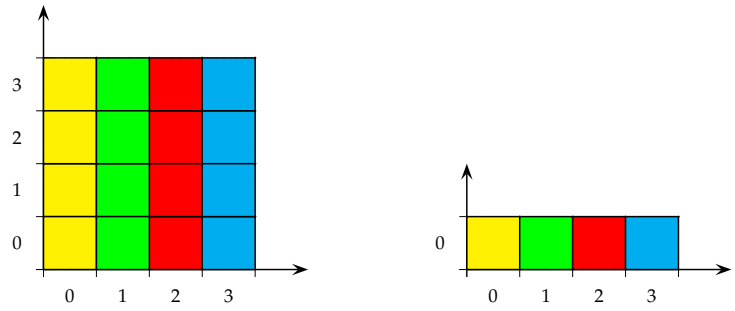


FIG. 1.30: Placement de tâches sur une grille de processeurs.

consiste à lier les éléments atomiques de l'application (les tâches élémentaires) et de l'architecture (processeur, mémoire, accélérateur, bus, etc.) avec un IP dans une bibliothèque. La figure 1.31 représente le déploiement d'une tâche élémentaire (en haut) et de ses ports sur un IP (en bas). Il peut exister plusieurs façons/langages pour implémenter une tâche, le composant stéréotypé ≪AbstractSoftwareImplementation≫ fait donc référence à la fonctionnalité de la tâche (une addition dans l'exemple). Cette vue abstraite d'un IP contient les « vrais » IPs, qui eux sont stéréotype ≪SoftwareImplementation≫. Dans cet exemple, la tâche élémentaire est déployée sur l'IP *additionSequentielleFortran* écrit en Fortran/OpenMP. Le fonctionnement du déploiement d'une architecture est identique, nous ne l'illustrons donc pas.

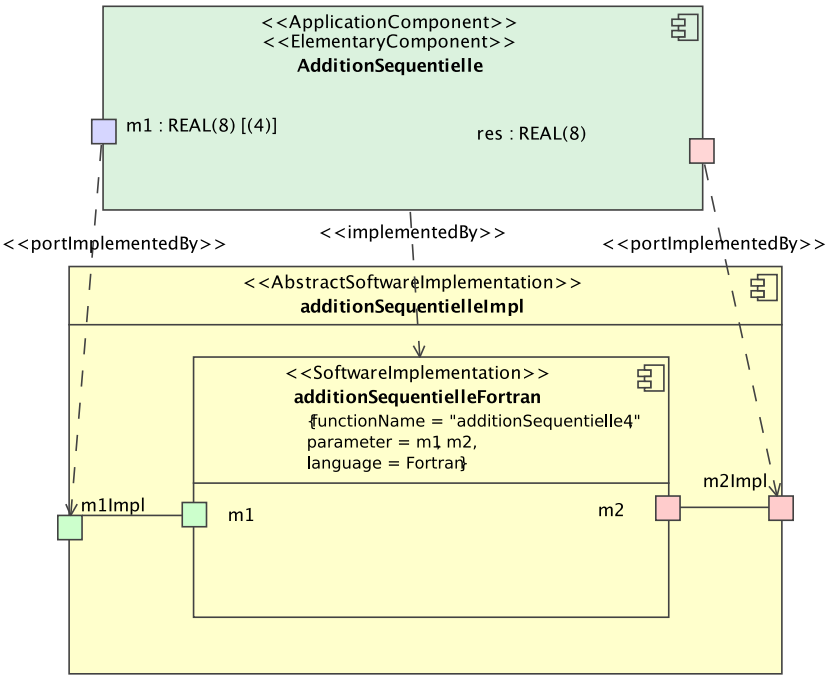


FIG. 1.31: Représentation du déploiement dans Gaspard. Le composant applicatif *AdditionSequentielle* et ses ports sont déployés sur un IP en Fortran/OpenMP.

1.7.4 Métamodèles cibles

Les chaînes qui permettent la génération de code SystemC et Fortran/OpenMP sont en partie communes : l'association d'une application sur une architecture permet de générer un modèle polyèdre. Ce polyèdre alimente CLooG [10] qui produit alors des boucles associées aux ressources de calcul. La séparation des deux chaînes intervient à cet endroit car la différence entre les codes à générer ne permet plus la factorisation des transformations. Les concepts de polyèdres et de boucles sont respectivement introduits dans les chaînes de conception par les métamodèles *Polyhedron* et *Loop*.

La complexité des polyèdres dépend de la forme du placement et leur nombre dépend du nombre de motifs compacts placés. Ce modèle polyédrique est obtenu par compilation de l'association [113, 96], puis est transformé vers un modèle de boucles, après passage dans l'outil CLooG. Le résultat est un code exécuté par chaque processeur de la grille, les tâches exécutées correspondant bien entendu aux tâches placées sur ces processeurs. La figure 1.32 représente sur la gauche le polyèdre issu du placement de la multiplication de matrices sur 4 processeurs (figure 1.30), la droite de la figure représente le code exécuté par chacun des processeurs. Dans ce polyèdre, les x0 et x1 représentent les indices de répétition de la tâche, q0 et q1 sont relatifs au pavage issu du placement, tandis que d0 est relatif à l'ajustage.

$$\begin{cases} p0 \geq 0, 3 - p0 \geq 0 \\ -p0 + q0 = 0 \\ -x0 + q0 = 0 \\ -x1 + q1 = 0 \\ x0 \geq 0, 3 - x0 \geq 0 \\ x1 \geq 0, 3 - x1 \geq 0 \\ q0 \geq 0, 3 - q0 \geq 0 \\ q1 \geq 0, 3 - q1 \geq 0 \\ d0 \geq 0, 0 - d0 \geq 0 \end{cases}$$

```
do  q1=0,3
      x1=q1
      d0=0
      q0=p0
      x0=p0

      dotProductMultiBloc(inA,inB,out)

end  do
```

FIG. 1.32: A gauche le polyèdre, à droite le code des boucles exécuté par chacun des quatre processeurs.

Il est possible, à partir du métamodèle de boucle, de générer du code SystemC ou du code Fortran/OpenMP par l'intermédiaire du métamodèle de langage procédural.

1.7.5 Transformations de modèles

L'IDM est omniprésente dans l'environnement Gaspard : le passage d'un métamodèle à un autre est réalisé par des transformations de modèles. Pour cela, Gaspard utilise des transformations *modèle vers modèle* et des transformations *modèle vers texte* pour la génération de code. Le manque de standards pour la représentation et l'exécution de ces transformations a abouti à l'intégration dans l'environnement Gaspard d'outils répondant à ses besoins propres : TrML [43] (Transformation Modeling Language) pour la représentation graphique des règles de transformations, MoMoTE (Model to Model Transformation Engine) pour leur

exécution et MoCodE (Model to Code Engine) pour la génération de code. Nous détaillerons ces outils et leur utilisation pour nos travaux plus tard dans ce document.

1.7.6 Bilan de l'environnement Gaspard

En terme d'outillage, l'environnement Gaspard peut se résumer à :
- un profil Gaspard qui permet la modélisation en UML d'une application, d'une architecture et d'un placement dans Gaspard. Ce profil correspond en partie à un sous-ensemble du standard MARTE et est étendu par le déploiement présenté en détails dans les thèses d'Eric Piel [96] et de Rabie Ben Atitallah [13] ;
- un ensemble de métamodèles qui permettent la description d'un système à différents niveaux d'abstraction et qui permettent de le spécialiser en fonction de la plateforme d'exécution choisie ;
- un ensemble de transformations de modèles qui assurent le passage d'un métamodèle à un autre ainsi que la génération de code ;
- des outils qui représentent et exécutent les règles de transformation.

En terme de cible, Gaspard permet de générer du code exécutable et simulable sur des architectures multiprocesseurs ou du code permettant de vérifier formellement une application. Avant mon arrivée au sein de l'équipe, Gaspard ne permettait pas de cibler l'exécution d'une application sur un accélérateur matériel.

1.8 Positionnement de nos travaux

Nos travaux de thèse s'inscrivent dans le développement de l'environnement Gaspard. L'objectif de nos travaux est de permettre la génération d'accélérateurs matériels à partir d'une modélisation à un haut niveau d'abstraction (un modèle Deployed). Ces travaux sont innovants sous deux aspects fondamentaux que sont l'exécution matérielle (*i.e.* par un accélérateur matériel) des applications Gaspard et la réalisation d'un flot de conception qui prend en entrée une application modélisée en UML pour générer le code VHDL de l'accélérateur correspondant. Nous introduisons nos travaux et nos contributions plus en détail dans le chapitre suivant.

Chapitre 2

Un flot de conception IDM permettant l'exécution matérielle d'applications Gaspard

Contents

Ce chapitre présente dans les grandes lignes les contributions de cette thèse. Nous nous appuyons sur le contexte de notre étude présenté dans le chapitre précédent afin de proposer notre flot de conception. L'objectif de ce flot de conception est de générer des accélérateurs matériels à partir d'applications modélisées à un haut niveau d'abstraction : le point d'entrée de notre flot est une application modélisée à haut niveau d'abstraction en UML et la sortie est un code HDL synthétisable qui décrit l'accélérateur matériel et qui permet l'exécution sur FPGA de l'application initialement modélisée en UML.

Notre flot de conception a deux particularités principales : l'utilisation de l'IDM qui permet de se concentrer, dans la description et l'utilisation du flot de conception, sur l'intention plutôt que sur les détails d'implémentation ; le domaine d'application qui relève du traitement de signal intensif, où le parallélisme est omniprésent.

Ce chapitre est décomposé en trois parties : la section 2.1 motive la nécessité d'un flot de conception IDM pour la génération d'accélérateurs matériels qui permettent l'exécution d'applications de traitement de signal intensif. La section 2.2 présente les grandes lignes de ce flot de conception et la section 2.3 le positionne dans l'environnement Gaspard.

2.1 Nécessité d'un flot de conception

2.1.1 Chaînes de compilation classiques à l'utilisation de l'IDM

Le chapitre précédent fait état de divers outils et langages permettant la création d'accélérateurs matériels à partir de spécifications qui font plus ou moins abstraction des détails d'implémentation de ces accélérateurs. La grande majorité de ces outils débute par une spécification en C des algorithmes afin de fournir une certaine abstraction des langages HDL nécessaires aux implémentations sur FPGAs. Il ne permet toutefois pas de monter en abstraction dans la description du matériel. La montée en abstraction permet d'éviter de modéliser les détails liés à l'implémentation et de réutiliser un même modèle d'application pour différentes cibles d'exécution.

La montée en abstraction des descriptions des applications nécessite la création de niveaux de descriptions intermédiaires, chacun de ces niveaux permettant de se rapprocher du niveau de description proche de celui de l'implémentation et à partir duquel le code peut être généré. La même application spécifiée à un haut niveau d'abstraction peut donc être exécutée sur plusieurs plateformes en utilisant ces niveaux de descriptions intermédiaires. La spécification de ces niveaux n'est pas un travail aisé car il faut définir avec précision l'objectif de chacun d'entre eux. Par ailleurs, le passage d'un niveau de description à un autre doit être automatisé de manière à raffiner une description en la rapprochant du niveau de description de l'implémentation. Pour finir, la montée en abstraction pour la description des applications passe par une représentation graphique de ces dernières. Le principal avantage de la représentation graphique est qu'elle offre une vision immédiate de la structure des applications, de leurs hiérarchies à leurs dépendances de données. Les outils de compilation actuels supportent mal les représentations graphiques et préfèrent utiliser une description textuelle (en C par exemple).

Pour ces raisons, nous pensons que l'IDM permet de mettre en œuvre ces processus de compilation en utilisant notamment des transformations de modèles [119, 86, 62]. En effet, l'IDM met à disposition des outils pour la réalisation de la compilation en uniformisant dans un même environnement la représentation des différents niveaux de description et en permettant la représentation graphique du processus de compilation [43, 93].

Au-delà de l'outillage, l'IDM porte sur l'intention plutôt que sur l'implémentation : la description d'une application se concentre sur son intention. Elle est indépendante des détails d'implémentation et peut donner lieu à des exécutions dans différents langages par le biais de processus de raffinement. Ces processus de raffinement reposent sur l'existence de niveaux de descriptions intermédiaires. Chaque niveau est spécifié par un métamodèle : plus le niveau de description est raffiné, plus elle est dépendante d'une implémentation et plus le métamodèle est détaillé. Des transformations de modèles permettent d'automatiser le passage entre les différents niveaux d'abstraction. Ces métamodèles et transformations de modèles peuvent être réutilisés pour les besoins d'une nouvelle implémentation de l'application sur une nouvelle plateforme. Ainsi, un métamodèle de description d'architectures matérielles indépendant de toute syntaxe HDL permet de générer un code Verilog ou VHDL par exemple.

2.1.2 Génération de circuits électroniques depuis ARRAY-OL

Nous avons présenté le langage ARRAY-OL qui exprime sous une forme factorisée les dépendances de données sur des tableaux multidimensionnels. Il est intéressant de vérifier dans quelle mesure ce langage est adapté pour la description d'accélérateurs matériels.

Peu de travaux s'intéressent à la génération de circuits électroniques depuis des applications décrites en langage ARRAY-OL. L'idée principale de Wood *et al.* [126] est de réaliser les dépendances de données exprimées en ARRAY-OL à l'aide du mappage des ports en VHDL. Cette méthode permet d'exprimer les dépendances de données sous une forme factorisée (similaire à la forme ARRAY-OL classique) mais nécessite toutefois que les ports potentiellement multidimensionnels des tâches répétées soient déroulés. En dehors de cet inconvénient, seules des applications possédant un seul composant sont illustrées et aucune indication sur la gestion de la hiérarchie et du parallélisme de tâches n'est donnée. De plus, les dépendances de données sur le temps ne sont pas gérées car seules des connexions simples sont réalisées pour les dépendances de données dans l'espace (par un mappage des composants). Pour finir, le code VHDL est directement généré à partir de la description de l'application ARRAY-OL.

Ces différentes constatations nous amènent à reconsidérer l'exécution matérielle des applications dont le parallélisme de données est exprimé avec ARRAY-OL. Pour cela, il est nécessaire de proposer un modèle d'exécution matériel pour ces applications dont on connaît le comportement en présence de hiérarchie et de parallélisme de tâches. Concernant les dépendances de données, il est nécessaire de considérer celles qui interviennent sur le temps car elles sont couramment utilisées en traitement du signal intensif, lors des étapes de filtrage notamment. Enfin, la dépendance directe de la compilation avec un langage HDL doit être évitée dans le but de favoriser la ré-utilisabilité en factorisant les développements pour tous les HDL.

2.1.3 Optimisation des accélérateurs

L'objectif d'un accélérateur est de réaliser des calculs tout en remplissant des contraintes données. Ces contraintes peuvent être des latences, des temps d'exécution, une surface, etc. L'adaptation de l'accélérateur vis-à-vis de ces contraintes est réalisée durant une phase d'optimisation. De manière générale, une optimisation consiste à adapter un accélérateur à son environnement.

L'outil SYNDEX-IC [109, 99] permet de défactoriser les spécifications d'applications de manière à trouver une implémentation du circuit qui remplisse les contraintes de latence tout en étant implémentable sur un FPGA donné. Cependant, la factorisation automatique des applications n'est pas gérée puisqu'elle est spécifiée par l'utilisateur de l'outil. La factorisation consiste à exécuter sur le temps le parallélisme initialement spécifié sur l'espace. L'outil commercial SYNPLIFY DSP [81, 9] utilise ce principe de factorisation (sous le nom de folding) afin de réaliser une prospection architecturale semi-automatique.

L'automatisation des factorisations et défactorisations des applications de traitement de signal intensif permet de guider le processus d'optimisation de manière à ajuster la quantité de ressources nécessaires pour une implémentation en fonction de la quantité de ressources disponibles sur FPGA. Dans le cas précis des applications Gaspard, les fonctions de refactoring ARRAY-OL introduites dans [39, 110] permettent de modifier une application en une autre application en jouant sur la hiérarchie et sur le parallélisme de données. Plusieurs fonctions de refactoring ARRAY-OL peuvent alors être appliquées successivement afin de façonner une application en fonction de sa cible d'exécution. Ces fonctions de refactoring peuvent donc être utilisées pour réaliser les factorisations et défactorisations des applications que nous traitons. Cependant, il est nécessaire d'étudier l'impact de ces fonctions de refactoring ARRAY-OL sur les accélérateurs matériels qui permettent l'exécution des applications.

Par ailleurs, les fonctions de refactoring sont appliquées à un niveau de description des applications indépendant des implémentations. L'accélérateur généré à partir de la nouvelle description de l'application permet de décider d'une éventuelle nouvelle fonction de refactoring à réaliser sur l'application (c'est-à-dire dans le niveau de description indépendant de son implémentation). L'optimisation des accélérateurs utilise donc deux niveaux de raffinement des applications, la représentation de l'application correspond au code généré de l'accélérateur au terme du processus d'optimisation. Par ailleurs, les fonctions de refactoring ARRAY-OL à un niveau de description indépendant des plateformes d'exécution permettent de factoriser leur utilisation car chaque cible d'exécution peut en tirer profit de la manière qui vient d'être présentée.

L'optimisation des accélérateurs au moyen des fonctions de refactoring ARRAY-OL est guidée par des caractéristiques d'implémentation sur FPGA. La quantité de ressources du FPGA consommées pour l'implémentation d'un accélérateur pourra être prise en compte.

2.1.4 Représentation de FPGAs et expressions des placements

Dans la littérature, les FPGAs sont représentés de manière à exprimer des placements de tâches et à fournir des caractéristiques sur les implémentations de ces tâches. La précision et le niveau de détails avec lesquels sont représentés les FPGAs varient en fonction de l'objectif de la représentation : la génération de code VHDL du FPGA lui-même requiert un niveau proche du RTL [71, 26], la génération d'une configuration nécessite des connaissances avancées sur sa structure [68, 16, 17], la génération de fichiers de contraintes de placements repose sur une vision topologique du FPGA [108], la définition d'heuristiques de placement se contente d'une vue en grille [123, 11, 112]. Le standard de modélisation des circuits temps réels et embarqués MARTE [101] permet la représentation des FPGAs mais se contente d'une classification en fonction de l'agencement des cellules reconfigurables.

Les différentes représentations des FPGAs utilisés dans ces travaux manquent de généricité dans la mesure où de nouvelles représentations voient régulièrement le jour, pour les architectures hétérogènes par exemple [108]. Il est donc utile d'unifier la représentation des

FPGAs en fonction de leurs objectifs et du niveau de détails souhaité. La métamodélisation des FPGAs permet d'exprimer chacune de leurs caractéristiques avec des concepts précis et de définir précisément les relations entre ces concepts. Chaque type de ressource dans un FPGA correspond à un concept, son organisation topologique en est un autre, etc. La modélisation des placements de tâches sur FPGA peut, elle aussi, être assimilée à un ou plusieurs concepts : la même représentation d'un FPGA doit permettre à la fois la caractérisation des ressources nécessaires à l'implémentation d'une tâche et à la fois la spécification du placement de cette tâche.

La caractérisation de l'implémentation d'une tâche sur un FPGA doit pouvoir être réalisée par construction et par assemblage afin d'être la plus indépendante possible vis-à-vis des FPGAs. Pour cela, il est nécessaire de proposer des mécanismes génériques pour la représentation de ces caractéristiques d'implémentation. Ces caractéristiques d'implémentation sont alors exploitées pour guider le placement des tâches sur FPGA. Bien que les fabricants de FPGAs s'orientent vers de l'hétérogénéité en y intégrant des processeurs [127], des mémoires ou des circuits à grain moyen [7], la régularité dans l'organisation des cellules reconfigurables demeure. Cette régularité doit être exploitée pour le placement de tâches répétitives et régulières afin d'augmenter les performances des implémentations [61] issues des résultats de synthèse. Techniquement, la synthèse est toujours réalisée par les outils commerciaux mais est guidée par des scripts de placement.

2.1.5 Bilan des motivations et utilisation de l'IDM

Nous avons montré tout au long de cette section que l'utilisation de l'IDM est une solution intéressante pour la génération d'accélérateurs matériels pour les applications de traitement de signal intensif. L'IDM permet d'élever le niveau d'abstraction de la description des applications en favorisant l'intention de l'application plutôt que son implémentation. Cela permet de modéliser une application de façon totalement indépendante de son exécution : le même modèle d'application peut être implémenté et exécuté sur différentes plateformes lors de la compilation.

Dans le cas de la génération d'accélérateurs matériels dont les dépendances de données sont exprimées avec le langage ARRAY-OL, il est intéressant de définir un niveau d'abstraction indépendant de toute syntaxe HDL. La compilation de ces dépendances de données pourra être réutilisée pour la génération de code VHDL ou Verilog par exemple.

Pour finir, l'IDM favorise les développements en mettant à disposition des utilisateurs et des concepteurs des outils de modélisation et de transformation. Par exemple, la standardisation récente du profil MARTE [101] permet d'unifier la représentation de ces systèmes. La distribution du modeleur UML Papyrus [94] intégré dans l'environnement Eclipse permet une modélisation dans un outil libre d'accès.

Ces différentes constatations nous amènent à proposer un flot de conception reposant sur l'IDM et permettant de générer automatiquement le code VHDL d'accélérateurs matériels à partir des applications modélisées à un haut niveau d'abstraction.

2.2 Notre flot de conception

Nous proposons un flot de conception basé sur l'IDM qui permet de générer des accélérateurs matériels à partir d'une modélisation à haut niveau d'abstraction des applications de traitement de signal intensif. Nous appelons le sous-ensemble des applications que nous

traitons *les applications Gaspard* car nos travaux sont intégrés à l'environnement de conception Gaspard. Par ailleurs, nous considérons le déploiement des tâches élémentaires de l'application sur des IPs écrits dans une syntaxe HDL, c'est donc un modèle Deployed de l'environnement Gaspard que nous considérons car celui-ci contient l'application[1]. Notre flot de conception est illustré à la figure 2.1, nous le présentons de manière très générale dans un premier temps et introduisons par la suite les éléments que nous avons développés pour le mettre en œuvre.

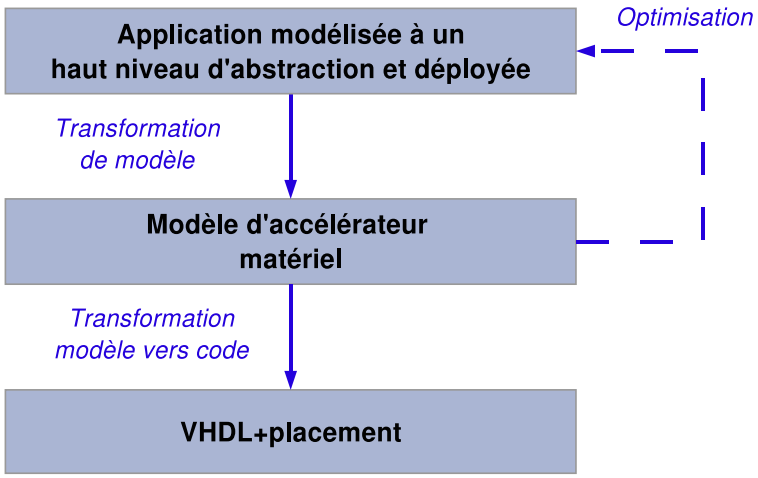


FIG. 2.1: Notre flot de conception guidé par l'IDM.

2.2.1 Vue d'ensemble de notre flot de conception

Le point d'entrée de notre flot de conception est une application Gaspard modélisée à un haut niveau d'abstraction, c'est-à-dire indépendamment de son implémentation. Ces modèles d'application respectent le profil MARTE et sont « compilés » à l'aide d'une transformation de modèles vers un modèle d'accélérateur matériel conforme au métamodèle RTL, pour Register Transfer Level (niveau transfert de registre). Le métamodèle RTL reprend les concepts utilisés lors de la conception d'accélérateurs matériels qui permettent l'exécution des applications Gaspard. En plus de la métamodélisation des accélérateurs, le métamodèle RTL permet la modélisation des FPGAs et du placement des accélérateurs sur ces FPGAs. Les résultats de ces placements permettent d'estimer les ressources du FPGA nécessaires à l'implémentation d'un accélérateur.

Un processus permet d'optimiser un accélérateur en fonction de la quantité de ressources qu'il consomme et de celles disponibles sur le FPGA ciblé. Ce processus d'optimisation est itératif, il prend en entrée un modèle RTL et produit en sortie un modèle RTL optimisé. Il repose sur les fonctions de refactoring ARRAY-OL qui permettent de modifier le modèle d'application Gaspard.

[1]Nous utilisons dans ce document les termes « applications Gaspard » et « modèle Deployed », mais c'est bien une application déployée sur des IPs que nous prenons en compte dans nos travaux.

Chaque itération se décompose en deux phases : premièrement, les caractéristiques d'implémentation de l'accélérateur généré sont évaluées et comparées aux ressources du FPGA ; deuxièmement, une modification de l'application (*i.e.* modèle Deployed) est réalisée si nécessaire, ce qui entraîne la génération d'un nouvel accélérateur (*i.e.* modèle RTL). Le processus d'optimisation s'appuie donc sur une description de l'application dépendante de la plateforme d'exécution matérielle mais modifie le modèle d'application décrit à un haut niveau d'abstraction. De cette façon, le modèle Deployed final correspond à l'accélérateur optimisé et décrit par le modèle RTL.

La description de l'accélérateur matériel dans le métamodèle RTL est suffisamment détaillée pour permettre la génération de code synthétisable. Nous ciblons le langage de description matériel VHDL mais pouvons tout aussi bien générer du code Verilog car le métamodèle RTL modélise les accélérateurs matériels indépendamment de leur syntaxe. Par ailleurs, le placement de l'accélérateur sur le FPGA au niveau du métamodèle RTL permet la génération d'un script de placement pour les outils de synthèse.

Le code VHDL généré est alors synthétisé sur FPGA via les outils commerciaux comme Quartus d'Altera ou ISE de Xilinx. Le résultat d'une synthèse est l'implémentation sur FPGA d'un accélérateur matériel qui permet l'exécution d'une application initialement modélisée en UML et indépendamment de cette exécution sur FPGA.

La suite de cette section présente dans les grandes lignes les travaux nécessaires à la mise en œuvre de notre flot de conception. Nous débutons par la présentation du modèle d'exécution matérielle des applications Gaspard qui a permis notamment de définir un sous-ensemble du métamodèle RTL. Ce modèle d'exécution est un point central dans nos travaux car c'est de lui que dépend la qualité des accélérateurs matériels générés par notre flot de conception.

2.2.2 Aperçu du modèle d'exécution matérielle des applications Gaspard

La puissance de l'expression du parallélisme de données et la gestion des dépendances de données entre des tableaux multidimensionnels rendent ARRAY-OL particulièrement bien adapté à la description d'applications de traitement de signal intensif. Une application décrite en ARRAY-OL ne peut cependant pas être directement exécutée, des *modèles d'exécution* pour des plateformes d'exécution spécifiques doivent être utilisés. Un modèle d'exécution décrit la manière dont une application ARRAY-OL s'exécute sur une cible bien précise.

Le parallélisme de données des applications Gaspard est issu du langage ARRAY-OL, dont plusieurs modèles d'exécution ont été proposés : séquentielle, SPMD, pipelinée [39]. Ces modèles d'exécution sont dédiés aux architectures multiprocesseurs et ne conviennent donc pas aux accélérateurs matériels. Par ailleurs, les travaux proposés par Wood *et al.* [126] concernant le mappage des ports en VHDL pour la réalisation des dépendances de données exprimées en ARRAY-OL ne sont pas suffisamment génériques pour être exploités dans nos travaux car directement dépendant de VHDL. De plus, il n'est jamais question, dans ces travaux, d'exécution de tâches sur plusieurs niveaux de hiérarchie ou d'exécution du parallélisme de tâches. Nous proposons ainsi dans cette thèse un modèle d'exécution matérielle des applications Gaspard.

Nous survolons dans un premier temps les exécutions matérielles du parallélisme de tâches et de données des applications Gaspard que nous proposons dans cette thèse (nous

y reviendrons plus en détail dans la suite du document). Nous présentons par la suite les avantages et inconvénients de ce modèle d'exécution.

2.2.2.1 Parallélisme de tâches

Chaque tâche d'une application Gaspard s'exécute en matériel par une unité de calcul implémentée sous la forme d'un composant. Plusieurs tâches peuvent donc être exécutées en parallèle par plusieurs unités de calcul.

Le parallélisme de tâches des applications Gaspard est similaire à un graphe de tâches : les entrées et sorties des tâches sont des tableaux de données et les connecteurs entre les entrées et sorties de différentes tâches représentent les dépendances de données. Nous retrouvons cette même structure dans les accélérateurs qui permettent l'exécution de ce parallélisme de tâches : les différentes unités de calcul qui exécutent les tâches sont connectées entre elles par des connecteurs. Un pipeline de tâches est donc réalisé lorsqu'un connecteur existe entre la sortie d'une tâche et l'entrée d'une autre tâche.

2.2.2.2 Parallélisme de données

Le parallélisme de données est plus spécifique aux applications de traitement de signal intensif. Dans les applications Gaspard, le parallélisme de données est contenu dans une tâche répétitive et apparaît sous différentes formes : l'espace de répétition autour de la tâche répétée, la dimension des motifs consommés par la tâche répétée, l'expression factorisée des dépendances de données par les tilers et la dimension des tableaux de la tâche répétitive.

Le modèle d'exécution matérielle qui permet l'exécution de ce parallélisme de données reprend ces différentes informations. Nous notons toutefois deux différences majeures que sont l'expression des dépendances de données et l'espace de répétition de la tâche répétée :

- les dépendances de données dans le parallélisme de données sont exprimées dans les applications Gaspard par les tilers. Les tilers expriment sous une forme factorisée des dépendances de données complexes entre l'itération de la tâche exécutée et les tableaux de données. Dans les accélérateurs, les tilers n'existent plus sous leur forme factorisée, ils sont compilés vers des connecteurs qui réalisent les dépendances de données qu'ils expriment. Cette étape est appelée le précalcul des tilers, car les dépendances de données sont calculées avant que l'accélérateur soit placé sur FPGA et donc avant l'exécution de l'application ;
- l'espace de répétition d'une tâche exprime le nombre de fois qu'une tâche est exécutée : il n'implique pas d'ordre d'exécution. Le modèle d'exécution matérielle permet d'exécuter le parallélisme de données contenu dans les applications Gaspard de deux façons différentes : parallèlement ou séquentiellement. L'exécution parallèle génère un accélérateur matériel performant mais coûteux en ressources de FPGA, l'exécution séquentielle génère un accélérateur moins performant mais peu coûteux en ressources. Par ailleurs, nous nous servons des fonctions de refactoring ARRAY-OL pour générer des accélérateurs qui exécutent ce parallélisme de données de façon intermédiaire. Ce point sera détaillé dans le chapitre 5.

Ce modèle d'exécution prévoit par ailleurs le précalcul des tilers afin de détecter la présence de dépendance de données sur le temps et de les gérer dans l'accélérateur généré. Ce point sera détaillé dans le chapitre 5 lors de la description de la compilation des dépen-

dances de données des applications Gaspard vers la réalisation de ces dépendances dans l'accélérateur.

Nous venons de présenter brièvement le modèle d'exécution matérielle des applications Gaspard, dont les avantages et inconvénients sont définis dans la section suivante.

2.2.2.3 Quand et pourquoi utiliser ce modèle d'exécution ?

Les dépendances de données dans le parallélisme de tâches sont réalisées par des connecteurs. Lorsque les unités de calcul de l'accélérateur sont synchronisées sur une horloge, ces tableaux sont générés à chaque cycle d'horloge et sont directement lus par les autres unités de calcul par le biais des connecteurs : nous obtenons un flux de tableaux traité à la volée et qui ne nécessite pas de mémoires intermédiaires (il n'y a donc pas de linéarisation des données nécessaires aux accès mémoires). De façon plus générale, les accélérateurs permettent d'ajuster la dimension d'un connecteur en fonction de l'application qu'il exécute. Dans le domaine du traitement du signal systématique, les données traitées sont généralement issues de capteurs dont les valeurs sont codées sur quelques bits. Cet ajustement de la dimension des connecteurs permet d'optimiser et d'améliorer le circuit, aussi bien en surface qu'en performance.

La quantité de ressources nécessaires à l'implémentation d'un accélérateur varie avec la complexité de l'application. Une application gourmande en ressources de calcul peut donc ne pas être implémentée en matériel car la taille du circuit produit peut ne pas être maîtrisée. Dans ce cas, il est préférable d'implémenter l'application sur des architectures plus standard de type multiprocesseur par exemple.

2.2.2.4 Bilan de l'exécution matérielle des applications Gaspard

Cette introduction au modèle d'exécution matérielle des applications Gaspard nous permet de mettre en évidence deux points : l'exécution du parallélisme de données et le précalcul des tilers. Le parallélisme de données peut s'exécuter de deux différentes façons : en parallèle ou en séquentiel. Selon le choix de l'exécution, la puissance de calcul et la quantité de ressources du FPGA consommée par l'accélérateur varient. Les performances de son exécution dépendent principalement de l'application traitée et du choix d'une exécution parallélisée ou séquentialisée du parallélisme de données. Le second point est le précalcul des tilers qui est une solution alternative au mappage des ports en VHDL proposée par Wood *et al.* [126]. Le précalcul des tilers permet de gérer des dépendances de données sur le temps.

2.2.3 Aperçu de la métamodélisation dans notre flot de conception

La conception de notre flot de conception nécessite un niveau de description intermédiaire entre le niveau de description des applications et le code VHDL généré. Ce niveau intermédiaire correspond au métamodèle RTL que nous avons développé durant cette thèse et que nous présentons succinctement ici.

2.2.3.1 Métamodélisation des accélérateurs

Le métamodèle RTL reprend l'ensemble des concepts nécessaires à la modélisation des accélérateurs matériels. Le métamodèle RTL reprend les concepts du modèle d'exécution matérielle des applications Gaspard que nous venons d'introduire ainsi que les relations

entre ces concepts. Il permet la modélisation de la hiérarchie, du parallélisme de données, etc. Un accélérateur dans le métamodèle RTL est automatiquement généré à partir d'un modèle d'application conforme au métamodèle Deployed de Gaspard par une transformation de modèles appelée DEPLOYED2RTL. Le métamodèle RTL, que nous décrivons en détail dans le chapitre 3, est indépendant de toute syntaxe HDL, mais son niveau de description permet la génération de code écrit dans un langage HDL. Nous avons développé une transformation de modèle du type *modèle vers texte* qui permet la génération de code VHDL des accélérateurs modélisés dans le métamodèle RTL. Nous nommons cette transformation RTL2VHDL et la présentons dans le chapitre 4.

2.2.3.2 Métamodélisation des FPGAs

Le métamodèle RTL permet par ailleurs la modélisation des FPGAs. Pour cela, nous avons extrait les concepts relatifs aux placements de tâches et à la caractérisation des implémentations. Le niveau de description des FPGAs que nous proposons permet de générer des scripts de placement afin de guider les outils de synthèse. De la même façon, nous avons conceptualisé la notion d'implémentation qui permet de définir les caractéristiques d'implémentation d'un composant de l'accélérateur sur un FPGA donné. Cela nous permet d'estimer la quantité de ressources du FPGA nécessaire à l'implémentation d'un accélérateur. Nous présentons la métamodélisation des FPGAs dans le chapitre 3 de ce document.

2.2.3.3 Bilan de la métamodélisation

Cette section présente de façon volontairement succincte la métamodélisation dans notre flot de conception car nous y reviendrons largement par la suite dans le chapitre 3 de ce manuscrit. Nous pouvons toutefois dégager les trois grands axes de développement liés à la métamodélisation : la définition d'un métamodèle RTL, la transformation DEPLOYED2RTL et la transformation RTL2VHDL.

2.2.4 Optimisation des accélérateurs

La transformation de modèle DEPLOYED2RTL génère un accélérateur donné pour une application donnée. Afin de procéder à une optimisation de l'accélérateur en fonction du FPGA sur lequel il est implémenté, nous proposons un processus d'optimisation guidé par l'estimation de la quantité de ressources du FPGA nécessaires à l'implémentation de l'accélérateur. Cette estimation est fournie dans le métamodèle RTL sous la forme du concept d'implémentation.

En fonction de cette estimation, le processus d'optimisation modifie la spécification de l'application Gaspard à l'aide de fonctions de refactoring ARRAY-OL. Ces fonction de refactoring ARRAY-OL sont couplées aux exécutions possibles du parallélisme de données dans notre modèle d'exécution matérielle : séquentiel ou parallèle. En jouant sur ces deux critères, nous sommes en mesure de générer plusieurs accélérateurs qui permettent tous l'exécution de la même application Gaspard. L'accélérateur sélectionné pour l'implémentation définitive (la génération de code) est celui dont la quantité de ressources consommée est la plus proche (dans la limite inférieure) de celle disponible sur FPGA.

2.2.5 Bilan du flot de conception

Nous venons de présenter notre flot de conception qui permet de générer des accélérateurs matériels à partir d'applications Gaspard modélisées à un haut niveau d'abstraction. Ce flot est innovant du point de vue de la génération d'accélérateur puisqu'il adopte une démarche IDM, tant dans la modélisation que dans la compilation.

La mise en œuvre de ce flot de conception à requis que nous développions :

- le modèle d'exécution matérielle pour des applications Gaspard ;
- le métamodèle RTL pour la modélisation des accélérateurs et de leur placement sur FPGA ;
- le processus d'optimisation pour ajuster la surface des accélérateurs avec celle du FPGA sur lequel ils sont implémentés ;
- la génération de code VHDL pour utiliser les outils de synthèse commerciaux classiques.

Ce sont ces travaux que nous mettons en avant tout au long de ce document.

2.3 Notre flot de conception dans l'environnement Gaspard

Notre flot de conception génère des accélérateurs pour des applications de traitement de signal systématique. Nos travaux font partie intégrante de l'environnement de co-conception Gaspard, dont l'articulation avec les différentes étapes de notre flot de conception est illustrée à la figure 2.2.

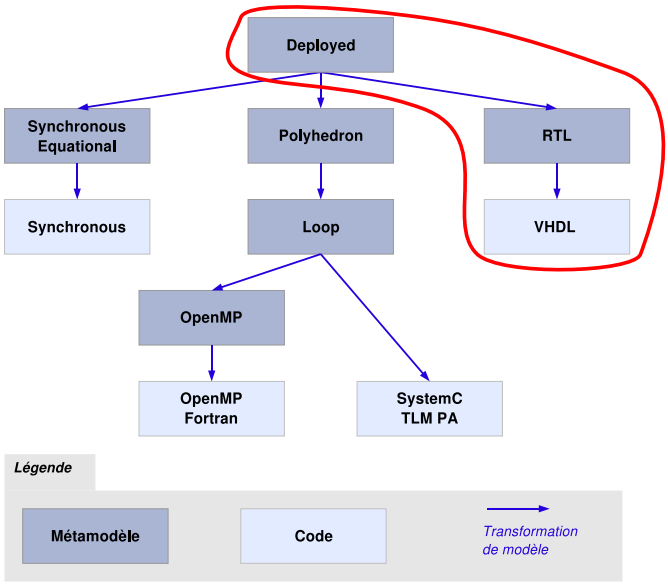


FIG. 2.2: L'environnement Gaspard incluant notre flot de conception dans le cercle rouge.

Nous débutons cette section par le positionnement des accélérateurs matériels depuis le métamodèle Deployed de l'environnement Gaspard.

2.3.1 Modélisation des accélérateurs dans l'environnement Gaspard

Les accélérateurs existent dans le métamodèle Deployed sous la forme de composants dans l'architecture et peuvent être exploité de deux façons différentes : *utilisation* ou *création* d'IP.

Utilisation d'IP Lorsque l'IP correspondant à la fonctionnalité de ce composant est disponible en bibliothèque, le composant est déployé vers cet IP, comme illustré à la figure 2.3. Dans cet exemple, le composant *Accelerator* est déployé sur l'IP *FIR_VHDL* écrit dans le langage VHDL. L'utilisation du déploiement sur un accélérateur est restreinte aux IPs disponibles en bibliothèque.

Création d'IP Lorsque nous souhaitons créer l'IP, le placement d'une application ou d'un sous-ensemble de l'application sur l'accélérateur définit le comportement de cet accélérateur (*i.e.* l'IP), comme illustré à la figure 2.4. Dans cet exemple, l'instance *task* dans le composant applicatif *Main* est placée sur l'instance d'accélérateur *acc* dans le composant d'architecture *SoC*. *task* instancie le composant applicatif *Compute* qui est lui-même composé des instances *t1*, *t2*, *t3* et *t4*, etc. L'objectif de notre flot de conception est de générer automatiquement l'IP qui permet l'exécution de l'application placée sur l'accélérateur. Pour cet exemple, nous générons l'IP qui permet l'exécution du composant *Compute*.

Bilan Notre flot de conception étend considérablement l'utilisation des accélérateurs dans les modélisations Gaspard puisqu'il n'est pas nécessaire de posséder l'IP de l'accélérateur en bibliothèque : nous générons automatiquement ces IPs.

Il n'y a pas de restrictions quant à la structure des composants, que ce soit au niveau de la hiérarchie, du parallélisme de tâches ou du parallélisme de données. La seule « contrainte » (déjà existante dans l'environnement Gaspard) est que les composants élémentaires de l'application soient déployés sur des IPs. C'est le cas du composant *T3* qui est déployé sur l'IP VHDL *Multiplication*.

Par ailleurs, une application Gaspard est modélisée indépendamment de son implémentation : elle est modélisée à un haut niveau d'abstraction. Ce haut niveau d'abstraction permet alors au concepteur de ne pas se soucier des détails de mise en œuvre et lui permet donc de créer un système sans être expert du langage ciblé. Dans notre cas, un concepteur de systèmes sur puce peut générer et utiliser un accélérateur matériel sans être expert de ces circuits.

2.3.2 Chaîne de transformation

Le point de départ de notre chaîne de transformation est un modèle Deployed et son point d'arrivée le code VHDL. Gaspard permet également la génération de code synchrone, SystemC et Fortran/OpenMP. Chaque cible possède sa propre chaîne de transformation que nous positionnons par rapport à notre chaîne de transformation.

2.3.2.1 Chaîne Deployed vers synchrone

La chaîne de transformations composée du métamodèle *Synchronous Equational* permet la génération de code de langages synchrones. Seuls l'application et le déploiement des IPs sont

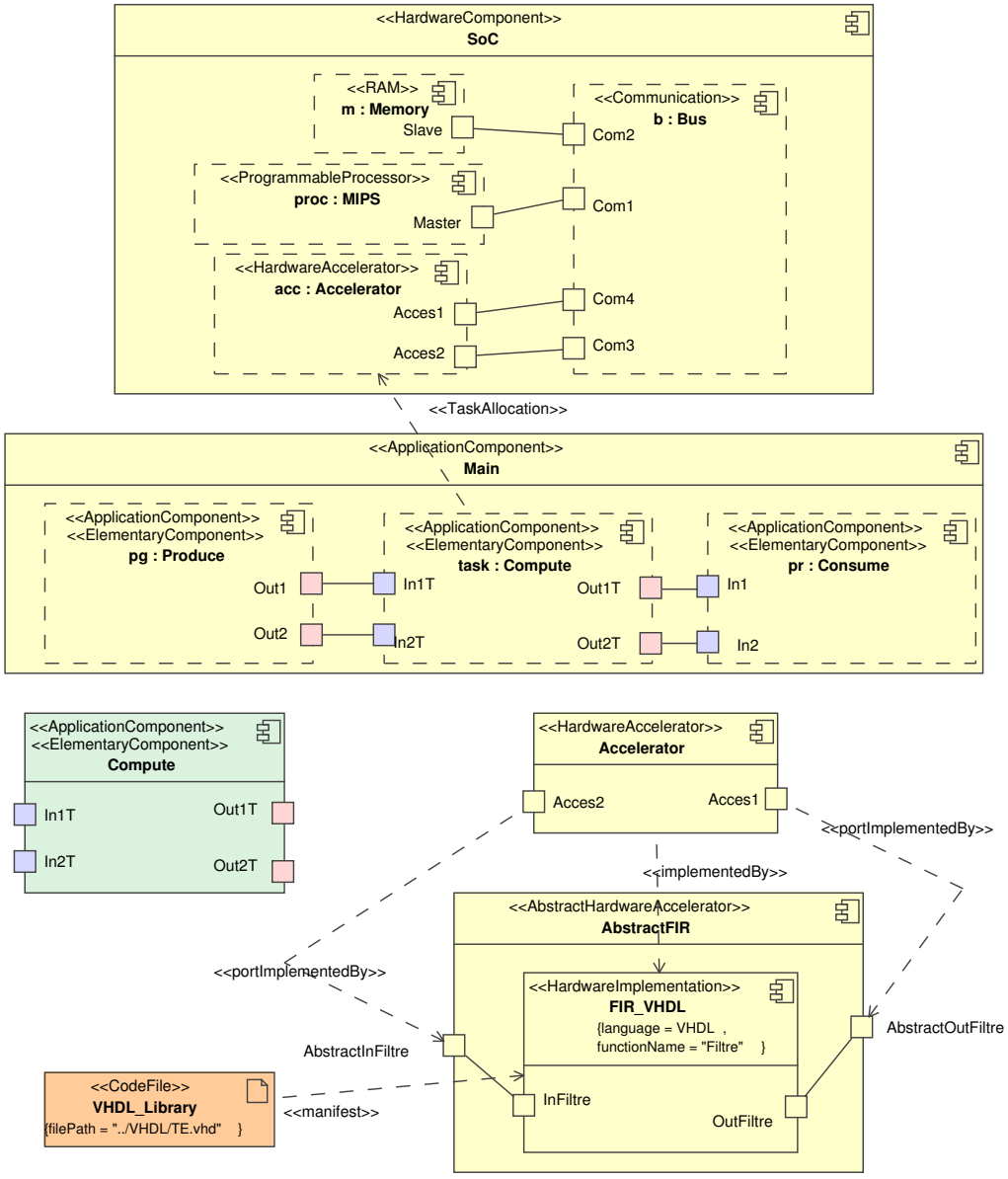


FIG. 2.3: Utilisation des accélérateurs dans l'environnement Gaspard : déploiement d'un accélérateur sur un IP existant.

transformés, le sous-ensemble du métamodèle Deployed utilisé par cette chaîne de transformation correspond donc à celui utilisé par notre chaîne. Cependant, les métamodèles Synchronous Equational et RTL n'ont pas les mêmes objectifs (vérification formelle pour l'un et synthèse pour l'autre), par conséquent, les concepts manipulés ne sont pas les mêmes.

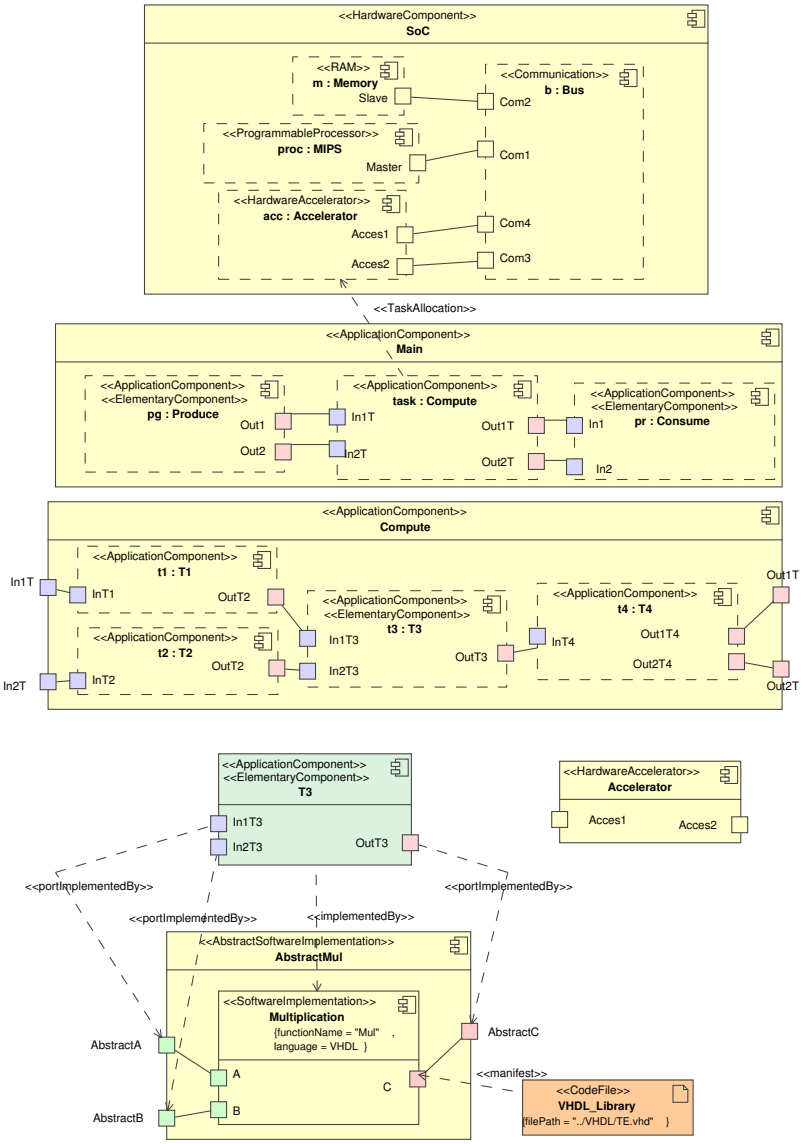


FIG. 2.4: Utilisation des accélérateurs dans l'environnement Gaspard : placement d'une application sur un accélérateur. Pour ce dernier cas, notre flot de conception génère l'accélérateur qui permet l'exécution de l'application placée sur l'accélérateur.

2.3.2.2 Chaînes Deployed vers SystemC et Deployed vers Fortran/OpenMP

Les générations de codes des langages SystemC et Fortran/OpenMP sont réalisées en utilisant une chaîne qui permet la gestion des placements d'applications répétitives sur des architectures répétitives. Un placement est retranscrit sous une forme polyédrique et alimente ClooG [10]. L'utilisation des polyèdres convient parfaitement aux placements de tâches sur des architectures multiprocesseurs car le nombre d'unités de calcul est connu (le nombre de processeurs pour une architecture multiprocesseur par exemple). Cependant, dans une

optique de modification du nombre d'unités de calcul utilisé pour l'implémentation d'un accélérateur, l'utilisation des polyèdres pour la représentation des placements n'est pas souhaitée. Les métamodèles et transformations de ces chaînes ne conviennent pas à la génération d'accélérateurs. De plus, les objectifs des métamodèles sont l'exécution des applications Gaspard sur des architectures multiprocesseurs (Fortran/OpenMP) et la simulation au niveau TLM du comportement d'un SoC (SystemC TLM PA). Les objectifs de ces métamodèles sont différents de ceux du métamodèle RTL, nous ne pouvons donc pas les réutiliser.

Il convient toutefois de reconsidérer l'utilisation des polyèdres pour le placement d'applications sur des grilles d'accélérateurs[2]. En effet, le placement d'applications sur ces grilles de FPGAs peut donner lieu à des distributions de tâches complexes qui sont efficacement gérées par des représentations polyédriques (dont nous savons que CLooG est en mesure de générer le code des boucles associées).

L'utilisation des polyèdres pour la représentation des placements sur des grilles de FPGAs nécessite toutefois soit une mémoire partagée par tous les FPGAs (ce qui n'est pas vraiment le cas dans la littérature puisque les grilles de FPGAs fonctionnement généralement avec des mémoires distribuées) soit une gestion des communications pour les accès aux données qui sont distribuées sur les mémoires locales aux FPGAs. Il est donc intéressant de considérer dans une extension potentielle de nos travaux une distribution des tâches sur la grille d'accélérateur par le biais de représentations polyédriques et d'optimisations locales des accélérateurs dans notre flot de conception.

2.3.3 Bilan de l'articulation de notre flot de conception au sein de l'environnement Gaspard

Nos travaux portent sur la génération d'accélérateurs qui permettent l'exécution d'applications Gaspard. Cela nécessite que nous manipulions le parallélisme des tâches tel que spécifié dans le métamodèle Deployed, c'est-à-dire sous la forme d'un espace de répétition et non sous la forme d'un nid de boucles comme c'est le cas au niveau du métamodèle Loop. Cet espace de répétition nous permet d'optimiser l'accélérateur généré et de le placer de façon régulière sur FPGA.

Du point de vue de la chaîne de transformations, notre choix se concrétise par une première transformation de modèles depuis le métamodèle Deployed vers le métamodèle RTL. Ce métamodèle reprend les concepts utilisés pour la conception des accélérateurs qui permettent l'exécution matérielle des applications Gaspard. Le point de sortie de cette chaîne de transformations est la génération d'un code VHDL synthétisable.

2.4 Conclusion

Nous avons présenté dans ce chapitre les grandes lignes de nos contributions qui, une fois mises bout à bout, permettent la génération d'accélérateurs matériels pour les applications Gaspard. La *branche* de Gaspard que nous avons développé étend l'utilisation des

[2] Il demeure une ambiguïté qu'il convient de lever quant à la réalité physique d'une grille d'accélérateur lors de sa réalisation sur une seule puce et qu'elle correspond à une décomposition structurelle de l'accélérateur matériel. Cette décomposition est déconseillée car elle ne fait que définir une structure à un accélérateur matériel qui n'en a pas. Dans le cas où la grille d'accélérateurs est implémentée sur plusieurs puces (FPGA par exemple), la décomposition en grille et le placement associé ont un sens.

accélérateurs dans Gaspard car nous les générons automatiquement : Gaspard n'est pas restreint à l'utilisation d'accélérateurs disponibles en bibliothèques.

Notre flot de conception fait parti de l'environnement Gaspard, la modélisation d'une application est la même pour la génération des accélérateurs que pour la génération des différentes autres cibles d'exécution. En particulier, le profil UML que nous utilisons offre un cadre de modélisation unifié pour l'environnement Gaspard et le métamodèle Deployed est le point d'entrée de toutes les chaînes de transformations. Cela est possible car notre approche est entièrement guidée par l'IDM qui favorise, entre autres, la réutilisabilité des métamodèles.

Nous détaillons dans la suite de ce document la mise en œuvre de notre flot de conception. Le chapitre 3 présente le modèle d'exécution matérielle des applications Gaspard et le métamodèle RTL qui lui est associé. Nous présentons la génération de code VHDL depuis le modèle RTL dans le chapitre 4. le chapitre 5 la compilation d'un modèle d'application Gaspard vers un accélérateur et le chapitre 6 son optimisation en fonction de sa cible d'implémentation FPGA. Le chapitre 7 est consacré à une étude de cas qui consiste à réaliser, à l'aide de notre flot de conception, un accélérateur matériel permettant la détection d'obstacles dans le cadre du transport routier.

Chapitre 3

Le métamodèle RTL pour la description des accélérateurs matériels

Contents

Le métamodèle RTL est au cœur de notre flot de conception, il correspond à un niveau de description intermédiaire entre la description d'une application à un haut niveau d'abstraction et son exécution matérielle par un accélérateur matériel. De ce fait, il prend en considération des détails d'implémentation liés à cette exécution matérielle. Ce métamodèle rassemble donc l'ensemble des concepts nécessaires à la description d'un accélérateur matériel qui permet l'exécution d'une application modélisée dans l'environnement Gaspard. La description des accélérateurs matériels doit être suffisamment détaillée pour en générer un code synthétisable sur FPGA.

Nous commençons ce chapitre par une description détaillée du modèle d'exécution matérielle d'applications modélisées dans l'environnement Gaspard. Nous présentons dans la section 3.1 deux exécutions possibles d'une même application. À partir de ce modèle d'exécution, nous extrayons les concepts qui servent de base au métamodèle RTL présenté dans la section 3.2. Dans la mesure où nous nous intéressons à l'implémentation de ces accélérateurs sur FPGA, nous proposons dans la section 3.3 une métamodélisation des FPGAs. Par ailleurs, des concepts d'implémentation sont aussi intégrés, et permettent de faire le lien, dans le métamodèle RTL, entre un accélérateur et un FPGA. Il devient alors possible d'exprimer des placements des accélérateurs sur les FPGAs.

Nous illustrons chaque partie de ce chapitre au travers d'un accélérateur qui exécute un filtre d'images, d'un FPGA Stratix2s60, et du placement de l'un sur l'autre. Bien entendu, les concepts sont créés indépendamment de ces éléments, et le travail présenté permet de modéliser toute application, prend en compte tout FPGA et autorise tout placement.

3.1 Modèle d'exécution matérielle des applications Gaspard

Nous montrons dans le chapitre 1 que les applications Gaspard sont modélisées indépendamment de toute cible d'exécution. Dans le cadre de la génération d'accélérateurs matériels, notre flot de conception raffine un modèle d'application Gaspard en un modèle conforme au métamodèle RTL. La conception de ce métamodèle nécessite l'identification des concepts liés à l'exécution matérielle des applications Gaspard. Nous illustrons cette exécution matérielle par le biais d'une application que nous détaillons dans un premier temps. Dans un second temps, nous présentons le modèle d'exécution des applications Gaspard.

3.1.1 Exemple d'application : le filtre d'images

Nous choisissons l'application du filtre d'images qui correspond à la convolution sur deux dimensions d'une fenêtre composée de 3×3 coefficients avec une image composée de M×N pixels. La valeur des coefficients détermine le comportement de l'application et permet ainsi de réaliser une moyenne des pixels, une détection de contour, etc. Le filtre produit alors une image de dimension (M-2)×(N-2) pixels.

La figure 3.1 illustre la modélisation en UML du filtre d'images que doit exécuter l'accélérateur matériel. Selon l'espace de répétition de la tâche et la taille des tableaux (*i.e.* des images), le circuit correspondant à l'implémentation physique de l'accélérateur matériel varie. Afin d'illustrer le fonctionnement de ces accélérateurs, nous travaillons sur des images composées de 4×4 pixels.

La figure 3.2 représente ce filtre d'images sous une forme « déroulée » afin d'illustrer le parallélisme de données exprimées sous une forme factorisée à la figure 3.1. Les images source et produite sont respectivement placées à gauche et à droite de la figure 3.2, chaque

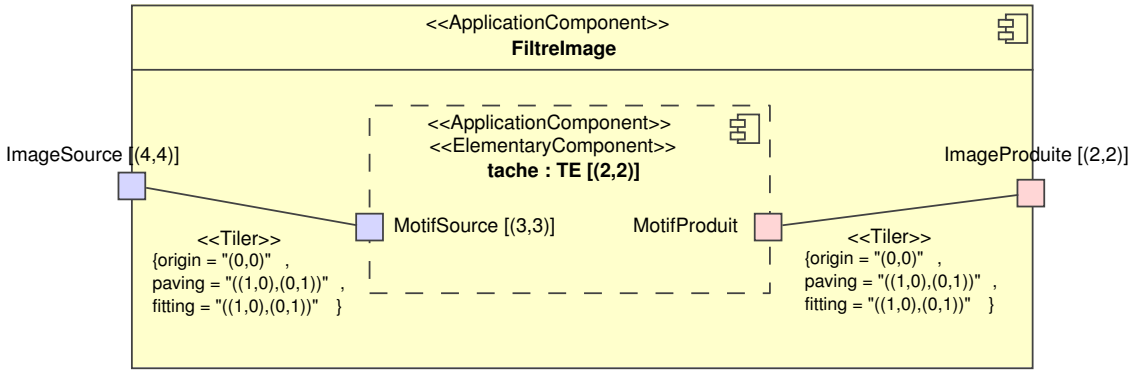


FIG. 3.1: Modélisation UML de l'application du filtre d'images.

case de ces images représente un pixel (une donnée). Les motifs sont construits à partir de leur forme, de l'espace de répétition de la tâche ($[(2,2)]$ dans cet exemple) et d'un tiler contenant les informations d'origine, de pavage et d'ajustage. Afin de mettre en évidence les dépendances de données entre une tâche et les pixels qu'elle consomme et produit, nous associons une couleur à chacune des tâches de l'espace de répétition. Lorsqu'un pixel est associé à l'une de ces couleurs, cela signifie qu'il est consommé ou produit par la tâche de couleur correspondante. Par exemple, les 4 pixels au centre de l'image source sont consommés par toutes les tâches dans l'espace de répétition. Les couleurs n'ont pas d'autre signification que la mise en évidence dans la figure des dépendances de données, la valeur d'un pixel est donc indépendante des couleurs. Nous considérons dans la suite de ce chapitre que ces pixels sont codés en niveaux de gris.

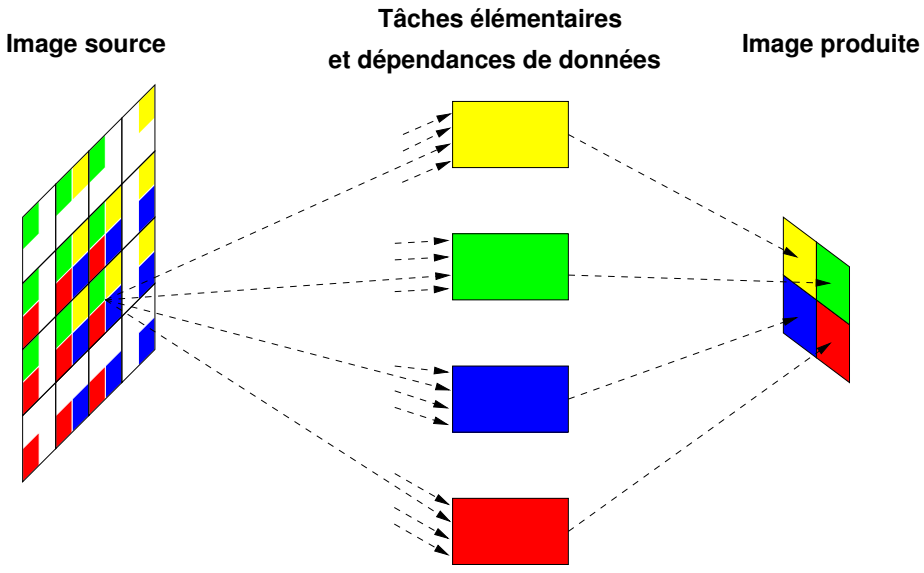


FIG. 3.2: Représentation « déroulée » du filtre d'images.

Deux exécutions matérielles d'une d'application Gaspard sont possibles, elles sont fonction du placement de l'application sur un accélérateur dans l'architecture : un placement entraîne une exécution parallèle tandis qu'un autre placement engendre une exécution séquentielle. Nous verrons dans le chapitre 6 qu'il est possible de réaliser des exécutions alternatives à ces deux exécutions. Les paragraphes suivants présentent les exécutions parallèle et séquentielle.

3.1.2 Exécution parallèle sur un accélérateur

3.1.2.1 Parallélisme de données

L'exécution parallèle du parallélisme de données engendre une mise à plat du circuit, ce qui permet d'exécuter en même temps (dans un même cycle d'horloge) chacune des itérations de la tâche répétée. Ainsi, N unités de calculs sont utilisées pour l'exécution parallèle d'une tâche répétée N fois.

Il existe toutefois une exception concernant la répétition infinie sur le temps, notée $[(\tilde{})]$ dans un modèle d'application Gaspard. Une répétition infinie est exécutée par le biais de l'horloge de l'accélérateur : chaque nouveau cycle de cette horloge engendre une nouvelle itération sur l'espace de répétition temporel de l'application. En d'autres termes, l'horloge permet d'exécuter la répétition temporelle des tâches.

Le même principe est appliqué pour les ports d'une application dont l'une des dimensions est infinie : le port équivalent en matériel ne contient pas cette dimension infinie, le temps génère le flux de données.

Nous nous plaçons dans un système à horloge unique, l'accélérateur entier est donc cadencé à une seule fréquence. Cela implique que les données présentes sur les ports sont consommées au même rythme que l'exécution des tâches. Autrement dit, un pas de 1 dans la consommation des données sur le temps doit correspondre à un pas de 1 dans l'exécution d'une tâche sur la dimension infinie de son espace de répétition. Toutes les dépendances de données exprimables dans Gaspard ne peuvent alors être prises en compte. Afin de garantir le bon fonctionnement des accélérateurs, nous imposons que les tilers aient une matrice de pavage de 1 sur la dimension du vecteur qui correspond à l'espace de répétition sur le temps. Cela ne s'applique qu'aux composants de l'application où une répétition temporelle est clairement exprimée.

3.1.2.2 Parallélisme de tâches

Le modèle d'exécution matérielle autorise l'exécution pipelinée d'une application : un tableau de données produit par une tâche peut être consommé par la tâche qui la succède. L'exécution pipelinée de tâches permet en général d'augmenter la fréquence de fonctionnement d'un accélérateur en décomposant son chemin critique.

Afin que cette exécution soit réellement pipelinée, des registres cadencés par la même horloge sont introduits dans les différents chemins de données. Pour cela, nous utilisons des tâches élémentaires qui possèdent des registres dans leur chemin de données, l'utilisation de ces tâches élémentaires permet la génération d'un flux de tableaux dans les chemins de données du parallélisme de tâches.

Ce flux de tableaux implique que plusieurs tâches sont exécutées en même temps mais que chacune d'elles itère sur un instant différent sur la dimension temporelle de leur espace

de répétition. Par ailleurs, ce pipeline introduit une latence dans la production des tableaux de sortie d'une tâche puisqu'il est nécessaire de remplir le pipeline de tâches.

Il existe toutefois des limitations quant à l'utilisation de ce pipeline pour le parallélisme de tâches. En effet, des de-synchronisations peuvent apparaître lorsque des dépendances de données ne franchissent pas tous les étages du pipeline, comme illustré sur le bas de la figure 3.3. Dans ce cas précis, le calcul réalisé par l'accélérateur est mauvais si le chemin de données de la tâche *t5* est composé de registres. C'est au concepteur de l'application de garantir que l'utilisation d'une tâche élémentaire ne de-synchronise pas les calculs.

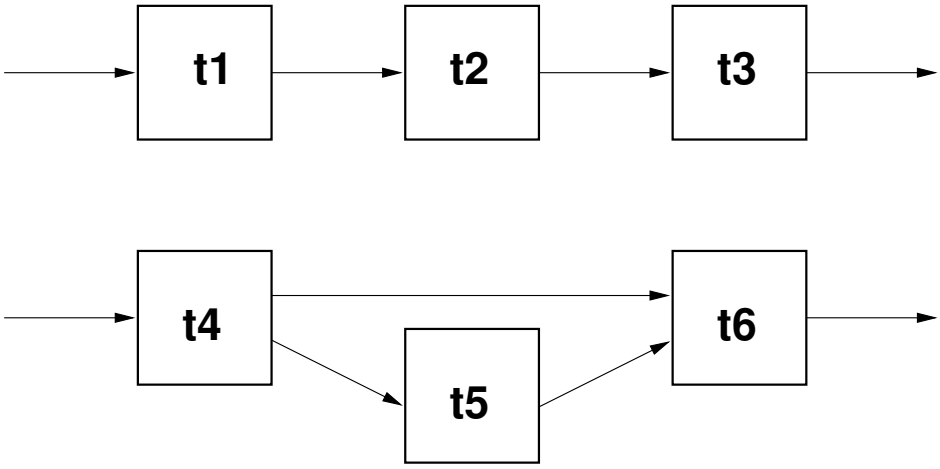


FIG. 3.3: Exécution du parallélisme de tâches : en haut un pipeline simple et en bas un pipeline potentiel couplé d'un risque de de-synchronisation du calcul.

3.1.2.3 Hiérarchie des accélérateurs

Notre modèle d'exécution gère la hiérarchie dans les accélérateurs, elle peut être composée de parallélisme de tâches, de données ou d'une combinaison des deux. Cela est possible car notre modèle d'exécution permet l'exécution de tous les calculs dans un même cycle d'horloge. Les sections suivantes présentent la réalisation des dépendances de données dans le parallélisme de données.

3.1.2.4 Précalcul des tilers

Un tiler exprime de façon factorisée les dépendances de données entre des motifs et un tableau, ces dépendances sont résolues par les calculs d'équations introduites dans le chapitre 1. Le précalcul consiste à résoudre ces dépendances de données avant l'implémentation physique de l'accélérateur (sur FPGA par exemple). Dans les accélérateurs, chaque dépendance de donnée se traduit par l'utilisation de ressources électroniques de type connecteur ou registre à décalage. Il devient alors envisageable de réaliser des dépendances de données à partir de ressources existantes, réduisant ainsi la quantité de ressources mises en œuvre pour la réalisation des dépendances de données. Nous détaillons ce processus « d'optimisation » et le précalcul des tilers dans le chapitre 5.

3.1.2.5 Regroupement du précalcul de tiler dans un composant matériel

Comme nous venons de le présenter, le modèle d'exécution matérielle précalcul les tilers de manière à optimiser l'utilisation des ressources mises en œuvre pour la réalisation des dépendances de données. Cette optimisation peut avoir lieu lors de la construction d'un motif, mais peut aussi être utilisée lors de la construction de tous les motifs. Pour que cela soit possible, il est nécessaire de manipuler une vision globale du calcul du tiler : nous regroupons donc le précalcul de chaque tiler dans un composant matériel.

Par ailleurs, le précalcul des tilers implique une grande quantité de code à générer et est donc nuisible à la lisibilité du code généré. Cet inconvénient est le prix à payer pour l'optimisation des tilers, mais est toutefois minimisé par le regroupement des résultats du précalcul dans un composant matériel : le code généré dans ce composant n'affecte pas la lisibilité des autres composants.

3.1.2.6 Illustration

La figure 3.4 représente la modélisation UML du filtre d'images et son placement sur un accélérateur. Les tâches de l'application sont placées sur l'accélérateur par le biais de distributions de tâches via un connecteur de dépendance stéréotypé ≪TaskAllocation≫. Le placement représenté par cette figure est assimilé à la génération d'un accélérateur matériel par le biais de notre flot de conception. Toutes les tâches de l'application sont exécutées par cet accélérateur de façon parallèle.

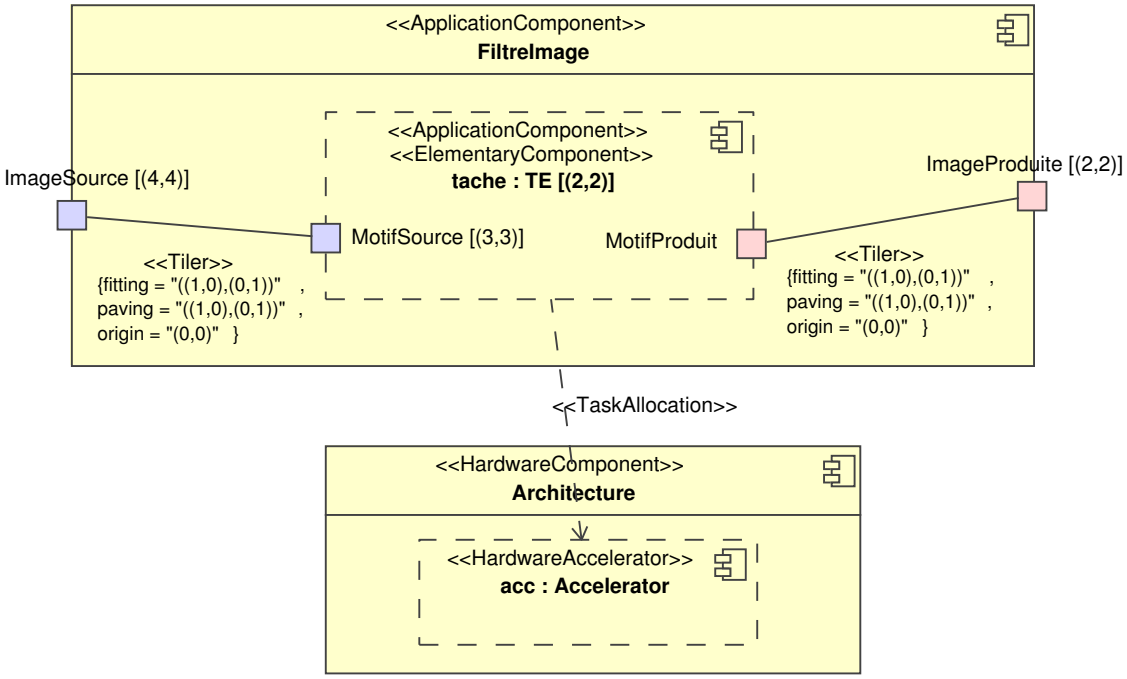


FIG. 3.4: Placement d'une application sur un accélérateur permettant une exécution parallèle.

La figure 3.5 illustre le circuit électronique de la tâche répétitive de l'accélérateur. La partie gauche de la figure représente les ports d'entrée de l'accélérateur correspondant au tableau d'entrée de la tâche, pastille **1**. Par symétrie, les ports de sortie correspondent au tableau produit par la tâche répétitive, pastille **5**. Ces ports sont connectés à des boîtes représentant des instances de composants, chacun exécutant un tiler du modèle d'application initial. Le tiler d'entrée, pastille **2**, est connecté en lecture au tableau d'entrée et produit les motifs consommés par les tâches dans l'espace de répétition. Les tâches sont exécutées en parallèle et sont au nombre de 2×2 pour cet exemple (pastilles **3x**). Chacune d'elle produit un motif de sortie. L'ensemble de ces motifs est connecté au tiler de sortie, pastille **4**, chargé de reconstituer le tableau de sortie.

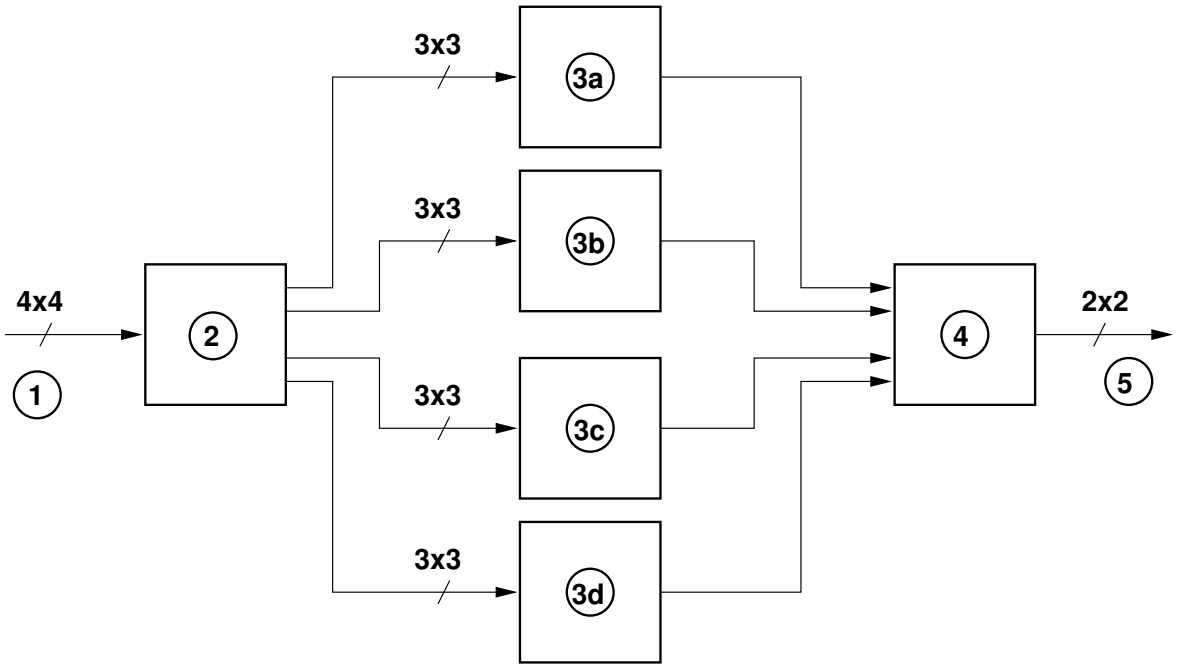


FIG. 3.5: Schéma du circuit électronique qui permet une exécution entièrement en parallèle du filtre d'images.

3.1.2.7 Bilan

L'exécution entièrement parallèle d'une application Gaspard est extrême puisqu'elle permet l'exécution de toutes les tâches dans un même cycle d'horloge et la réalisation de toutes les dépendances de données exprimées par les tilers. Afin de limiter les ressources de calcul nécessaires à la mise en œuvre d'un tel circuit, nous proposons d'exécuter séquentiellement une partie des tâches d'une application Gaspard. Cette exécution séquentielle permet donc une économie de ressources de calcul en contrepartie d'une diminution de la puissance de calcul. La section suivante présente l'exécution séquentielle du parallélisme de données dans ARRAY-OL. D'autres solutions intermédiaires, issues de la combinaison de l'exécution séquentielle et parallèle, sont possibles. Ces exécutions intermédiaires permettent d'ajuster les

ressources requises par l'accélérateur avec celles disponibles sur un FPGA et sont détaillées dans le chapitre suivant.

3.1.3 Exécution séquentielle sur un accélérateur

Nous venons de montrer qu'une exécution parallèle d'une application Gaspard permet l'exécution en un seul cycle d'horloge de tous les calculs exprimés par le parallélisme de données. S'il n'est pas nécessaire de produire un circuit aussi puissant ou si les ressources de calcul requises ne sont pas suffisantes, il est possible d'exécuter séquentiellement le parallélisme de données d'une tâche ARRAY-OL. En contrepartie d'une baisse de la performance, l'exécution séquentielle permet une économie des ressources de calcul, qu'elle réutilise pour l'exécution complète d'une application : une seule ressource de calcul est utilisée pour exécuter l'ensemble des instances parallèles exprimées par le parallélisme de données.

3.1.3.1 Utilisation d'un contrôleur pour la séquentialisation des tâches

La principale différence qu'introduit l'exécution séquentielle provient de l'unique ressource de calcul utilisée, réduisant à la fois la surface nécessaire à l'implémentation du circuit et ses performances. Cette ressource de calcul exécute séquentiellement chacune des tâches de l'espace de répétition. Pour cela, les motifs sont multiplexés vers cette ressource de calcul par le biais d'un contrôleur cyclique. Un cycle complet du contrôleur correspond à un balayage complet de l'espace de répétition de la tâche ARRAY-OL.

Dans notre implémentation, tous les multiplexeurs sont contrôlés par un même contrôleur, réduisant ainsi le coût d'implémentation. L'utilisation du même contrôleur pour toutes les entrées de la ressource de calcul est directe : la ressource de calcul itère sur l'espace de répétition à l'aide des multiplexeurs et consomme les motifs associés à l'itération courante. Tous les motifs d'entrée concernés sont connectés simultanément par les multiplexeurs à la ressource de calcul qui exécute la tâche et produit les motifs de sortie. Une fois tous les motifs de l'espace de répétition reconstitués (c'est-à-dire lorsque le contrôleur termine le parcours de l'espace de répétition), les tilers de sortie sont exécutés, et reconstituent le tableau de sortie (correspondant à une image dans notre exemple). Un délai intervient lors de cette exécution. Il dépend de la composition interne de la ressource de calcul (hiérarchique ou élémentaire selon la modélisation) et de la longueur de pipeline dans les feuilles de la hiérarchie (les tâches élémentaires). Ce délai est appliqué à la commande des multiplexeurs de sortie, garantissant le bon positionnement des données dans le tableau de sortie.

3.1.3.2 Pipeline de tâches exécutées séquentiellement

L'exécution séquentielle d'une tâche introduit nécessairement un délai (en cycle d'horloge) dans la création des tableaux de sortie. Il est donc nécessaire d'attendre la fin de l'exécution séquentielle d'une tâche (et donc la fin de la construction des tableaux) pour lire ces tableaux de données. Dans le cas d'un pipeline de tâches, le flux des tableaux est alors rythmé par les délais introduits par les exécutions séquentielles des différentes tâches : les tâches pipelinées doivent attendre que les tableaux soient entièrement construits avant de les lire. Des barrières de synchronisations apparaissent alors, à l'image des synchronisations existantes dans l'outil SYNDEX-IC [63].

Le pipeline de tâches ARRAY-OL exécutées séquentiellement pourrait donc s'inspirer des travaux de SYNDEX-IC en ce qui concerne la synchronisation des tâches. Cette solution

n'a cependant pas été implémentée et reste donc une extension potentielle de notre modèle d'exécution.

3.1.3.3 Hiérarchie dans le contrôle de l'exécution des tâches

L'exécution séquentielle des tâches que nous venons d'introduire n'est illustrée que pour un seul niveau de hiérarchie, bien que les applications traitées puissent en contenir plusieurs. Dans le chapitre 6, nous proposons des solutions intermédiaires pour les exécutions séquentielles couplées à des exécutions parallèles d'une même application. Nous verrons que ces solutions intermédiaires ne gèrent pas directement l'exécution séquentielle sur plusieurs niveaux de hiérarchies mais qu'elles la modifient de manière à permettre différentes exécutions séquentielles. L'objectif de cette section est de montrer ce qu'implique la gestion de l'exécution séquentielle sur plusieurs niveaux de hiérarchies pour notre modèle d'exécution matérielle.

Une séquentialisation des tâches dans la hiérarchie revient à introduire des délais d'attente dans les flux de tableaux à chaque niveau de hiérarchie où une séquentialisation des tâches est réalisée. L'information locale de la séquentialisation d'une tâche n'est plus suffisante. Jusqu'à présent, nous avions considéré que chaque motif fourni à la ressource de calcul par le biais des multiplexeurs était consommé dans le cycle. Nous savons que chaque motif construit dans un niveau de hiérarchie peut devenir un tableau dans le niveau de hiérarchie inférieur, à partir duquel peuvent être produits des sous-motifs. Lors de l'exécution séquentielle des tâches sur ces deux niveaux de hiérarchie, plusieurs cycles sont nécessaires pour consommer chacun des sous-motifs. Cela implique que le tableau permettant de construire ces sous-motifs est conservé durant plusieurs cycles. Or, ce tableau correspond à un motif dans le niveau de hiérarchie supérieur : notre contrainte initiale imposant que chaque motif soit consommé en un seul cycle n'est donc plus garantie.

Deux solutions permettent de gérer la séquentialisation des tâches sur plusieurs niveaux de hiérarchies. La première solution consiste à prendre en compte les délais introduits dans les niveaux de hiérarchie inférieurs et de les introduire dans les contrôleurs. Ces derniers peuvent alors maintenir un motif durant plusieurs cycles. La seconde solution consiste à introduire des fréquences de fonctionnement différentes à chaque niveau de hiérarchie exécutant les tâches séquentiellement : plus l'on descend dans la hiérarchie, plus la fréquence augmente. Cette solution de-synchronise le circuit en imposant la présence de multiples horloges, dont il est nécessaire de garantir l'ordre de grandeur : quelques centaines de MHz.

S'il est réaliste de vouloir augmenter le nombre de tâches exécutées séquentiellement par le biais de la hiérarchie, les solutions proposées introduisent des extensions de notre modèle d'exécution que sont la prise en compte des délais dans la hiérarchie et la gestion de multiples horloges. Bien que réalistes, nous ne retenons aucune de ces solutions et les considérons comme de potentielles extensions de notre modèle d'exécution. Cependant, cela ne signifie aucunement qu'il est impossible de modifier le nombre de tâches exécutées séquentiellement. En effet, l'utilisation des fonctions de refactoring ARRAY-OL permet de modifier la hiérarchie, regroupant par exemple les deux premiers niveaux de hiérarchie d'une application sur un seul niveau. L'exécution séquentielle de cette tâche sur ce nouveau niveau de hiérarchie correspond alors à l'exécution séquentielle sur les deux précédents niveaux de hiérarchie.

3.1.3.4 Gain en ressources entre une exécution parallèle et séquentielle

L'exécution séquentielle est moins directe que l'exécution parallèle puisqu'elle introduit des concepts liés au contrôle des connexions entre la ressource de calcul et ses motifs. Ces concepts se matérialisent en électronique sous la forme d'un contrôleur, de multiplexeurs et de démultiplexeurs. Chacun d'eux introduit un coût supplémentaire à l'implémentation d'un circuit exécutant séquentiellement les tâches. Le coût d'implémentation de l'exécution séquentielle dépend de différents paramètres de la tâche exécutée. Ces paramètres sont l'espace de répétition, la forme du motif, la taille des données et le nombre de tilers. Nous verrons dans la suite de ce document, à la section 3.3.2.1, comment l'estimation en ressource de ces différents concepts est réalisée.

3.1.3.5 Illustration d'une exécution séquentielle

La modélisation d'une application exécutée séquentiellement sur accélérateur est très similaire à celle présentée à la figure 3.5. L'unique différence provient d'une information supplémentaire sur le connecteur de dépendance ≪TaskAllocation≫ indiquant que l'exécution est réalisée séquentiellement.

Le circuit électronique qui correspond de l'implémentation du filtre d'image exécuté séquentiellement est représenté à la figure 3.6. Comme pour l'exécution parallèle, les ports d'entrées et de sortie (les images source et produite) sont placés aux extrémités de la figure, pastilles **1** et **5** respectivement. Ces ports sont connectés aux instances de composant tilers, qui sont calculés de la même manière que pour l'exécution parallèle : ils sont regroupés dans un composant et précalculés. Dans le cas du filtre d'image, il y a un tiler d'entrée, pastille **2**, et un tiler de sortie, pastille **4**. Les motifs produits et consommés par ces tilers sont tour à tour connectés à la seule ressource de calcul, pastille **3**, qui exécute une itération de la tâche par cycle d'horloge. Plusieurs cycles d'horloge sont donc nécessaires à l'exécution complète du parallélisme de données. Les connexions entre les motifs produits et consommés par les tilers sont réalisées par un multiplexeur, pastille **6**, et d'un démultiplexeur, pastille **8**. Ces multiplexeurs et démultiplexeurs sont commandés par le contrôleur, pastille **7**, qui parcourt itérativement l'espace de répétition de la tâche. Ainsi, à chaque cycle d'horloge, un nouveau motif est connecté à la tâche de calcul. Les calculs se terminent lorsque l'espace de répétition complet a été balayé par le contrôleur ; un nouveau cycle recommence alors : le contrôleur parcourt cycliquement l'espace de répétition de la tâche.

Plusieurs différences notables existent entre les exécutions matérielles parallèle et séquentielle d'ARRAY-OL. Nous constatons l'utilisation d'un contrôleur, de multiplexeur, de demultiplexeur, et d'une seule ressource de calcul. Dans les paragraphes suivants, nous faisons le bilan de ces différences.

3.1.4 Bilan du modèle d'exécution matérielle

Dans cette section, nous avons proposé deux exécutions matérielles du parallélisme de données de ARRAY-OL pour les applications Gaspard : parallèle et séquentielle. Le choix d'une exécution dépend du placement de l'application sur l'accélérateur dans la modélisation Gaspard. L'exécution parallèle d'une application Gaspard permet d'exécuter toutes les tâches de l'application en un cycle d'horloge tandis que l'exécution séquentielle nécessite plusieurs cycles d'horloge pour parcourir l'espace de répétition de la tâche du plus haut niveau de hiérarchie.

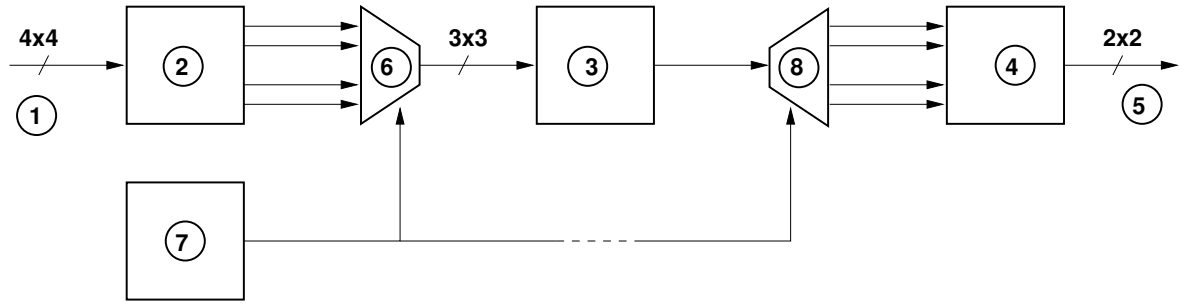


FIG. 3.6: Le schéma électronique qui permet l'exécution séquentielle du parallélisme de données pour le filtre d'images.

Dans la démarche de création de notre flot de conception, nous conceptualisons ce modèle d'exécution matérielle sous la forme du *métamodèle RTL* que nous présentons dans la section suivante.

3.2 Proposition d'un métamodèle RTL

Le métamodèle RTL regroupe l'ensemble des concepts utilisés pour la description d'accélérateurs matériels qui permettent l'exécution d'applications Gaspard. Ces concepts sont issus du modèle d'exécution introduit au début de ce chapitre [1].

Nous ne présentons pas comment *générer* le modèle RTL d'un accélérateur à partir d'une application Gaspard, mais ce que nous pouvons *décrire* avec le métamodèle RTL. La compilation d'une application Gaspard vers un accélérateur est détaillée dans le chapitre 5.

Nous débutons cette section par les motivations qui nous ont poussés à développer le métamodèle RTL, puis nous détaillons les concepts introduits afin de modéliser au niveau RTL des accélérateurs matériels qui permettent d'exécuter les applications Gaspard. Nous présentons en particulier les concepts liés au parallélisme de données.

3.2.1 Motivations et objectifs du métamodèle RTL

Le métamodèle RTL possède les avantages des métamodèles en général, c'est-à-dire qu'il est générique, qu'il permet la ré-utilisation de concepts définis précisément et dont les liens avec les autres concepts sont clairement identifiés.

Par ailleurs, et de façon plus spécifique à son contexte d'utilisation qu'est l'environnement Gaspard, le métamodèle RTL correspond à un niveau de description intermédiaire entre le métamodèle d'application dans Gaspard et le code VHDL des accélérateurs. Ce niveau intermédiaire permet de décomposer en deux étapes le travail de compilation d'un modèle d'application Gaspard en un code VHDL, la première étape étant indépendante du langage HDL ciblé (VHDL ou Verilog). Pour finir, la création du métamodèle RTL permet

[1] L'emploi du terme *modèle* pour la description du comportement à l'exécution des accélérateurs (*i.e.* modèle d'exécution) ne doit en rien semer le trouble chez le lecteur. En effet, il n'existe *a priori* aucune relation entre un *modèle* d'exécution et un *modèle* selon l'IDM. Aussi, nous limitons autant que possible l'utilisation du terme modèle dans le contexte de l'exécution matérielle et l'utilisons sinon sous la forme *modèle d'exécution*.

l'intégration de nos travaux dans l'environnement Gaspard, ce qui lui permet de profiter des outils qu'il met en œuvre.

L'objectif du métamodèle RTL est la description de circuits électroniques qui permettent l'exécution matérielle d'applications Gaspard. Ce modèle d'exécution est celui présenté dans la section précédente et qui permet, entre autres, les exécutions parallèle et séquentielle du parallélisme de données de ARRAY-OL.

3.2.1.1 Similitudes et différences du métamodèle RTL avec les autres métamodèles de l'environnement Gaspard

Au cours de cette section, nous aurons l'opportunité de constater que certaines similitudes existent entre le métamodèle RTL et les autres métamodèles du flot de conception. Ces similitudes peuvent être des concepts très familiers à l'informatique en général, comme la notion composant qui est utilisée dans chaque métamodèle de notre flot de conception. Il existe d'autres similitudes qui sont plus spécifiques au modèle de calcul ARRAY-OL : le composant qui exprime le parallélisme de données est particulier à notre flot de conception mais est aussi présent dans chaque métamodèle de ce flot.

Cependant, l'ensemble des concepts existants dans les différents métamodèles de l'environnement Gaspard ne suffit pas à exprimer un modèle de circuit électronique au niveau RTL : la notion d'horloge est inexistante et il n'est donc pas possible de générer des circuits synchrones, les multiplexeurs nécessaires à l'exécution séquentielle du parallélisme de données sont eux aussi inexistants, etc. De manière plus générale, le niveau de description RTL d'un circuit nécessite des détails d'implémentations qui sont inexistants dans les autres métamodèles de l'environnement Gaspard.

Ainsi, nous présentons dans cette section l'ensemble du métamodèle RTL, mais mettons en avant les concepts nouveaux que nous avons introduits pour la modélisation RTL des accélérateurs matériels qui permettent l'exécution d'applications Gaspard. Par ailleurs, nous verrons que les règles de transformations présentées ultérieurement dans le chapitre 5 permettent de mettre en évidence, par le biais de représentations graphiques, les différences entre le métamodèle RTL avec le métamodèle Deployed.

3.2.1.2 Indépendance aux syntaxes HDL

Le métamodèle RTL est suffisamment précis pour permettre la génération de code synthétisable, tout en restant conceptuel, c'est-à-dire indépendant de toute syntaxe de langage de description matérielle. Ce métamodèle fait donc abstraction de toute syntaxe HDL et prête attention à la sémantique des concepts manipulés au niveau RTL et exploités pour l'exécution matérielle des applications Gaspard.

Nous introduisons l'indépendance à toute syntaxe de langage de description matérielle par la description d'un multiplexeur. La description d'un multiplexeur varie selon le langage de description matérielle (VHDL et Verilog par exemple) et au sein même d'un langage (utilisation de **Case** ou **If** en VHDL par exemple).

La figure 3.7 représente un multiplexeur « 4 vers 1 » indépendamment de toute syntaxe. Les quatre entrées de ce multiplexeur sont E1, E2, E3 et E4 et sont routées vers la sortie S en fonction de l'entrée de contrôle C.

La figure 3.8 présente la métamodélisation d'un multiplexeur : le concept MULTIPLEXER est composé d'une sortie de données, d'une entrée de contrôle et de plusieurs entrées de

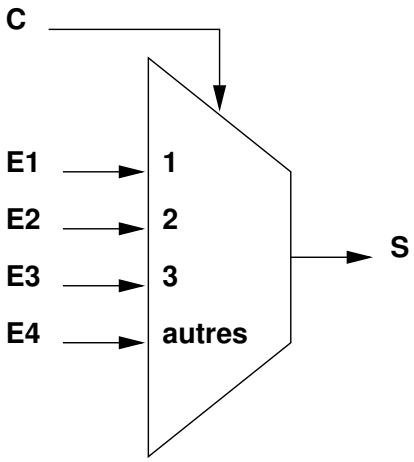


FIG. 3.7: Représentation électronique d'un multiplexeur : l'affectation de la sortie S par l'une des entrées (E1, E2, E3 ou E4) est conditionnée la valeur de C.

données. Ces entrées et sorties sont des éléments connectables, concept CONNECTABLEELEMENT, ce qui signifie qu'ils peuvent être connectés dans le circuit de l'accélérateur matériel. En fonction de l'entrée de contrôle *controlFlow*, la sortie de données *DataFlowOutput* est connectée à une des entrées de données *DataFlowInput*. Le choix de cette entrée de données est lié au concept INPUTSELECT, qui associe chaque entrée de données à un état *ModeCondition* de l'entrée de contrôle. Si un état de l'entrée de contrôle n'est associé à aucune entrée de données, la sortie prendra l'entrée de données par défaut *DefautDataFlowInput*. Ce métamodèle fournit les concepts nécessaires à la modélisation du multiplexeur représenté à la figure 3.7.

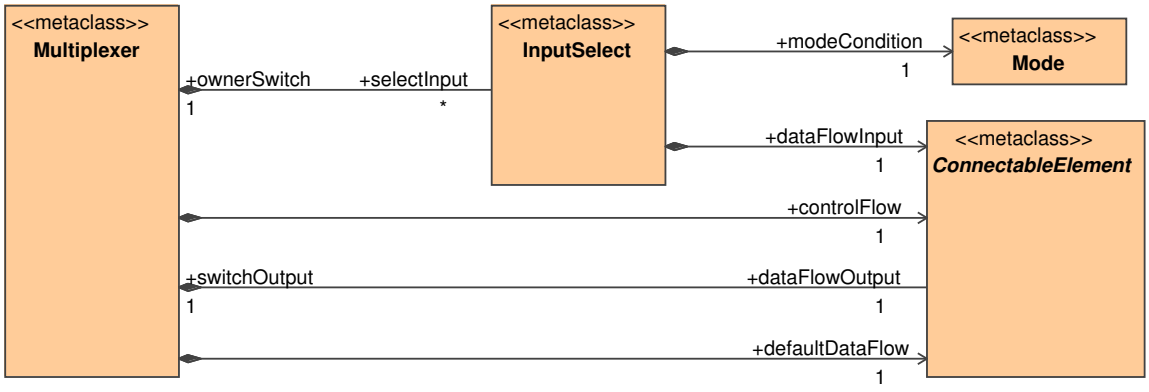


FIG. 3.8: Métamodèlisation d'un multiplexeur : il possède une sortie de données, une entrée de données par défaut, une entrée de contrôle et un ensemble de possibilités de routage (concept INPUTSELECT).

Nous venons d'illustrer l'indépendance du métamodèle RTL envers les syntaxes HDL

par le biais de la métamodélisation du multiplexeur. La suite du document présente différents concepts du métamodèle RTL.

3.2.2 Vue générale du métamodèle RTL

Le modèle d'exécution matérielle des applications Gaspard (décrit au début de ce chapitre) peut être décomposé de manière à en faire ressortir différents aspects. Il est par exemple possible de faire ressortir les « types » d'exécution parallèle ou séquentielle tout en conservant à l'esprit qu'elles reposent en partie sur des concepts communs : port, connecteur, etc.

De la même façon, la composition de vues selon l'IDM (introduite dans le chapitre 1) permet de mettre évidence certaines relations entre des concepts d'un métamodèle. Une vue peut donc ne pas « tout » représenter mais se focaliser uniquement sur un sous-ensemble d'un métamodèle. Nous utilisons ce mécanisme de vue pour la description du métamodèle RTL et mettons de ce fait en évidence certains mécanismes qu'il permet de décrire (à l'image du multiplexeur présenté précédemment).

Par ailleurs, nous ne décrivons pas l'intégralité du métamodèle RTL, mais détaillons les points qui nous semblent importants. Nous présentons dans un premier temps les concepts relatifs aux composants utilisés dans les accélérateurs (composants élémentaires, hiérarchiques, etc.) et détaillons par la suite l'ensemble des concepts nécessaires à l'expression du parallélisme de données.

3.2.3 Concept COMPONENT

Un composant est un élément hiérarchique et permet donc la description structurelle d'un accélérateur matériel. Le comportement d'un composant dépend de sa composition.

La figure 3.9 représente le concept de COMPONENT dans le métamodèle RTL : de par son héritage à NAMEDELEMENT, COMPONENT possède un nom sous la forme d'une chaîne de caractère. Les possibilités de description du comportement d'un composant sont infinies, des contraintes de modélisation sont donc apportées. Nous nous appuyons sur les types de composants manipulés lors des exécutions matérielles de ARRAY-OL pour introduire trois concepts de composants que sont ELEMENTARYTASK, COMPOUND et REPETITIVE[2]. Chacun de ces concepts spécialise un composant en fonction du rôle qui lui est associé dans le modèle d'exécution que nous avons proposé.

ELEMENTARYTASK représente l'élément atomique du modèle d'exécution matérielle de ARRAY-OL et, par conséquent, n'a pas de structure. Son comportement est associé à un IP par la référence *implementation*.

Le concept IMPLEMENTATION, illustré à la figure 3.10 correspond donc un IP dans notre métamodèle . Le terme IP est utilisé car la composition de l'implémentation est inconnue. Cependant, des informations de caractéristiques et de paramètres sont associées à chaque implémentation par le biais du double héritage CHARACTERIZABLE et SPECIALIZABLE du concept IMPLEMENTATION. Les paramètres (PARAMETER) référencés par *characteristics* permettent de fournir des informations sur le comportement d'un IP, comme le nombre d'étages dans son pipeline. Les paramètres référencés par *specialization* permettent la spécialisation du comportement d'un IP lors de son instanciation, par exemple la définition de la taille d'une mémoire.

[2]Nous retrouvons ces concepts dans le modèle ARRAY-OL tel que définit dans la section 1.3.

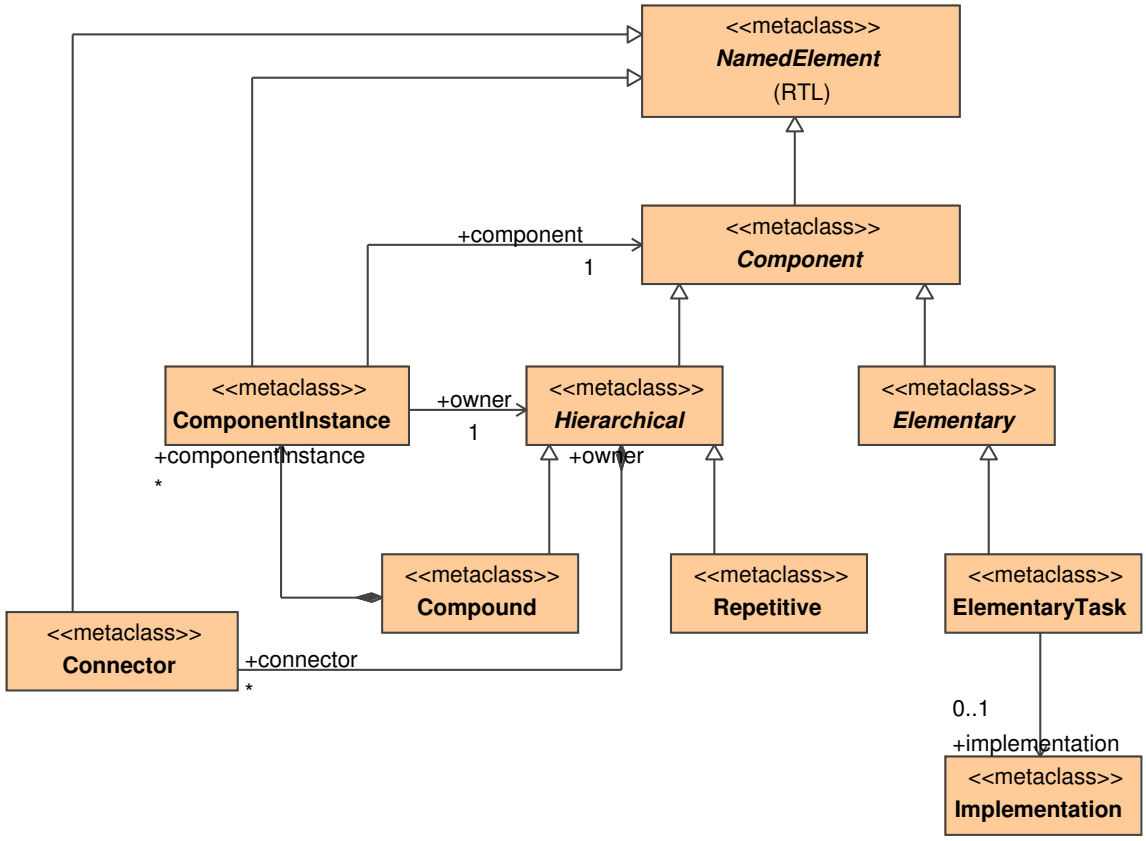


FIG. 3.9: Métamodélisation des composants élémentaire, composé et répétitif.

IMPLEMENTATION et les concepts associés que nous venons de présenter sont repris des travaux décrits dans la thèse d'Eric Piel [96], bien que les IPs manipulés dans notre flot de conception sont différents des IPs logiciels manipulés dans [96, 13] : nous manipulons des IPs à flot de données pouvant recevoir de nouvelles données à chaque cycle d'horloge et pouvant contenir plusieurs étages dans le pipeline. Cela est possible car nous manipulons des IPs qui ne nécessitent pas d'interfaçage complexe et dont le grain est adapté à des exécutions dédiées. En ce qui concerne les IPs matériels utilisés dans la modélisation de l'architecture dans l'environnement Gaspard, il est possible de trouver un point de convergence. En effet, le grain des accélérateurs matériels que nous générons dans notre flot de conception correspond au grain des IPs sur lesquels sont déployés les accélérateurs matériels dans l'architecture de Gaspard. La démarche de notre flot de conception va d'ailleurs dans ce sens puisqu'il est amené à générer des IPs pour des accélérateurs matériels qui ne sont pas disponibles en bibliothèques.

Par ailleurs, il est possible d'enrichir les caractéristiques des IPs afin d'obtenir des informations supplémentaires sur leur implémentation FPGA par exemple. Dans la section 3.3.2.1, nous étendons les caractéristiques de nos IPs dans ce sens.

Les concepts COMPOUND et REPETITIVE introduisent tous deux une hiérarchie dans un modèle, ce qui entraîne des similitudes dans les relations de ces deux concepts, comme l'ins-

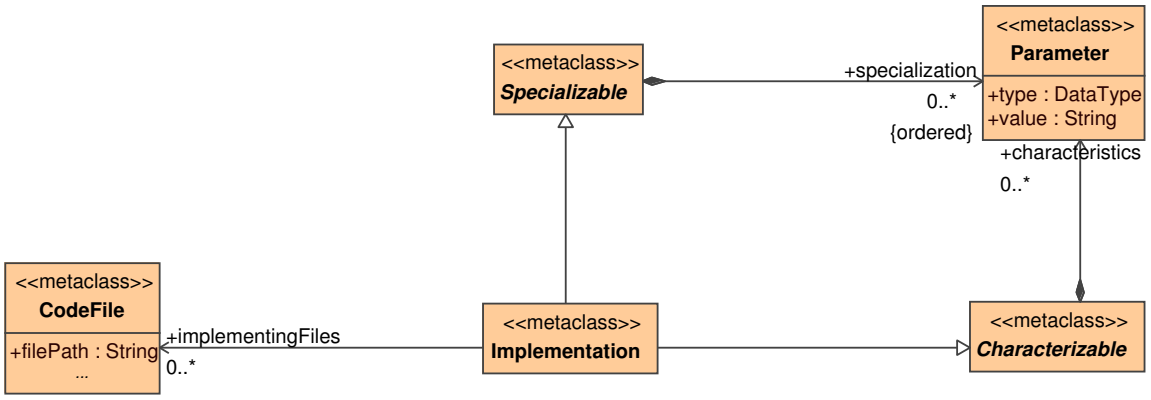


FIG. 3.10: Métamodélisation des implémentations sur IPs de composants élémentaires.

tanciation de composants ou la composition de connecteurs. Ces similitudes sont factorisées sont la forme d'un héritage des concepts COMPOUND et REPETITIVE sur le concept abstrait HIERARCHICAL. Un composant hiérarchique contient des CONNECTOR et des COMPONENTINSTANCE : un connecteur définit une relation entre deux éléments et une instance de composant référence un composant. Selon la spécialisation d'un composant hiérarchique en composant composé ou répétitif, une instance de composant est référencée différemment.

Un composant composé, concept COMPOUND, exprime le parallélisme de tâches de l'exécution matérielle de ARRAY-OL. Pour cela, il contient des instances de composant et des connecteurs : les instances de composant sont référencées par la relation *componentInstance* représentent les tâches du parallélisme de tâches et les connecteurs les dépendances entre ces tâches. L'ensemble représente donc un graphe de tâches. Le métamodèle RTL n'exprime aucune restriction sur la nature des composants référencés par ces instances de composant. Un composant composé peut donc contenir des tâches élémentaires, répétitives ou composées, et permet donc la représentation d'un accélérateur sur plusieurs niveaux de hiérarchie. Cette représentation est conforme à l'exécution matérielle de ARRAY-OL que nous avons présentée.

Un composant répétitif, concept REPETITIVE dans le métamodèle RTL, exprime le parallélisme de données de l'exécution matérielle de ARRAY-OL. Ce concept est détaillé dans la section suivante.

3.2.4 Concept REPETITIVE

REPETITIVE, illustré par la figure 3.11, est le concept de composant répétitif qui permet de décrire l'exécution matérielle du parallélisme de données. Les exécutions parallèles et séquentielles présentées dans la section précédente introduisent deux façons de modéliser le contenu d'un composant REPETITIVE.

Le haut de la figure modélise l'exécution parallèle du parallélisme de données en permettant l'instanciation multiple du composant répété. Chaque COMPONENTINSTANCE est associé au concept SHAPE par le biais d'une composition *dim*, qui exprime donc l'espace de répétition d'une tâche. La tâche correspond au composant que référence le concept COMPO-

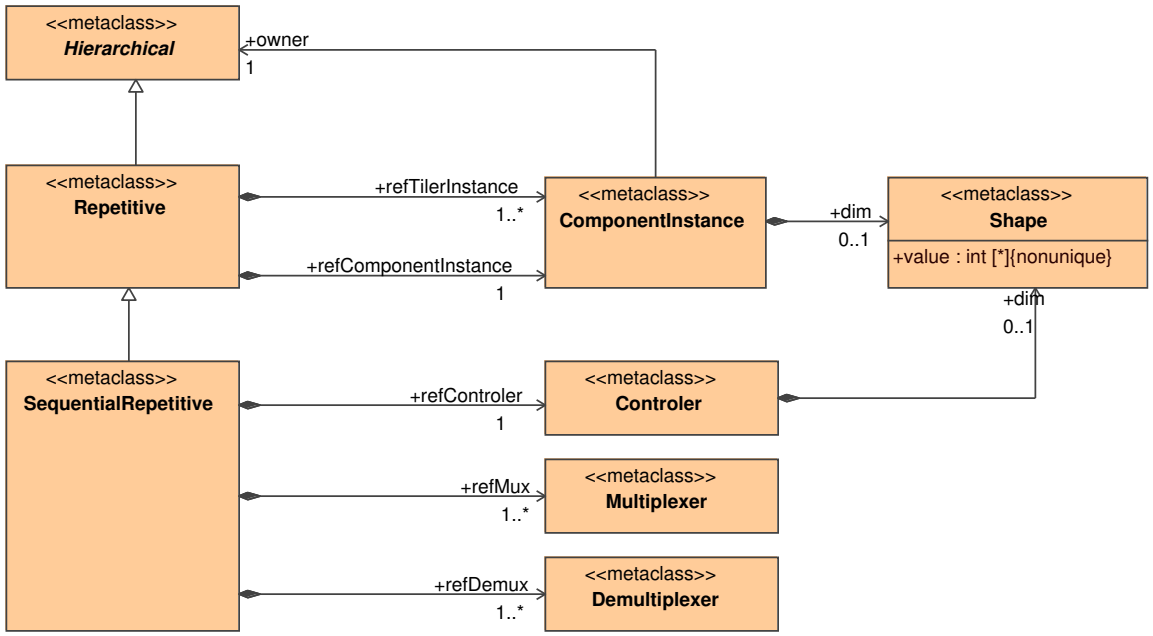


FIG. 3.11: Métamodèlisation d'un composant répétitif permettant une exécution séquentielle ou parallèle.

NENTINSTANCE.

Le bas de la figure représente le concept SEQUENTIALREPETITIVE qui correspond à la métamodélisation de l'exécution séquentielle du parallélisme de données. Ce concept possède les mêmes propriétés que le composant répétitif par le biais de l'héritage. Cependant, SHAPE n'est pas référencé par COMPONENTINSTANCE lors d'une exécution séquentielle des tâches puisque l'unité de calcul n'est instanciée qu'une seule fois selon notre modèle d'exécution. En revanche, une exécution séquentielle, modélisée par le concept SEQUENTIALREPETITIVE, utilise les concepts de contrôleur, multiplexeur et demultiplexeur que nous détaillons ci-dessous :

- CONTROLER référence l'espace de répétition d'une tâche et le balaie afin de contrôler les multiplexeurs et démultiplexeurs. Un espace de répétition peut être multidimensionnel, le contrôleur est donc construit en conséquence et référence une SHAPE.
- MULTIPLEXER alimente en motifs l'instance de composant qui exécute la tâche. Le concept MULTIPLEXER est celui présenté précédemment dans ce document.
- DEMULTIPLEXER reconstruit l'ensemble des motifs pour construire l'espace de répétition. Sa construction est en partie symétrique à celle du concept MULTIPLEXER et n'est donc pas détaillée.

Que ce soit pour une exécution séquentielle ou parallèle, un composant répétitif contient des instances de composant correspondant à des composants tilers, concept TILER. En effet, dans la section 3.1.2, nous choisissons de précalculer les tilers dans des composants afin de les réutiliser. Ce concept d'instance de composant est identique à celui de la tâche répétée et correspond à COMPONENTINSTANCE. Afin de différencier ces instanciations, COMPONENTINSTANCE est référencé par *refTilerInstance* ou *refComponentInstance*. Des contraintes de mo-

délisation imposent que chaque composant répétitif contienne une instance de composant répété et au moins une instance de composant tiler. En effet, un composant répétitif doit posséder au moins un tiler d'entrée ou un tiler de sortie.

Un point commun à chaque composant du métamodèle RTL (ELEMENTARY, COMPOUND, etc.) est l'interface. Une interface est décrite par des concepts PORT et PARAMETER, que nous détaillons dans la prochaine section.

3.2.5 Interface des composants

3.2.5.1 Ports de composant

Il est possible de communiquer avec un composant par le biais de ses ports, dont le concept est illustré par la figure 3.12. PORT se spécialise en une entrée INPUTPORT ou une sortie OUTPUTPORT et contient des informations sur l'organisation des tableaux de données via SHAPE et les types de données véhiculés via DATATYPE. SHAPE définit la dimension et la taille du port, et DATATYPE définit son type. Les types sont présentés dans la section 3.2.6.
.

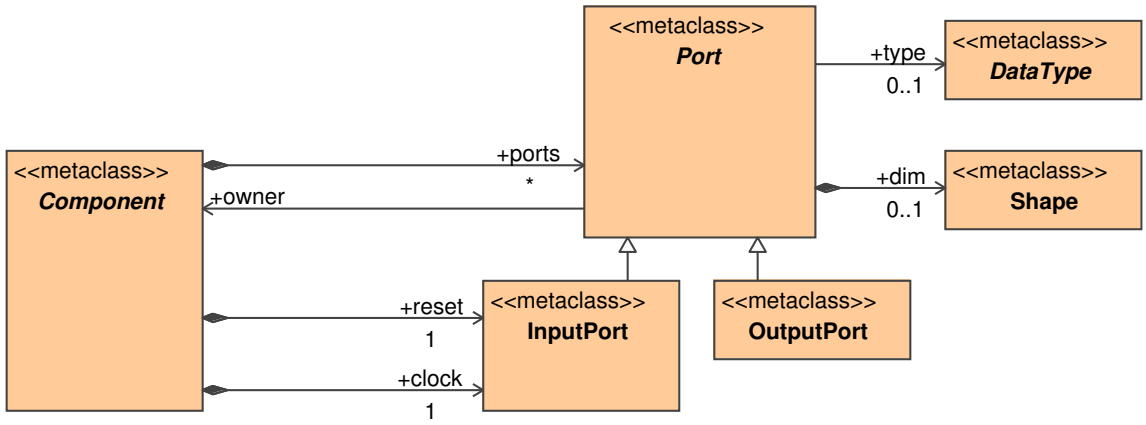


FIG. 3.12: Chaque composant possède plusieurs ports dont un spécifique à l'horloge et un autre spécifique à la remise à zéro.

En dehors de la communication avec un composant, aucune fonctionnalité spécifique n'est attribuée au concept de PORT. Il est toutefois possible de créer des ports à partir de relations différentes de *ports* avec COMPONENT. C'est le cas des relations *reset* et *clock* qui spécialisent chacune un port d'entrée INPUTPORT en un port de remise à zéro et un port d'horloge. Une sémantique particulière est alors attribuée aux ports concernés. Par ailleurs, les références unitaires (les 1 sur les références) sur *clock* et *reset* impliquent que chaque composant d'un modèle RTL possède un seul port d'horloge et un seul port de remise à zéro, ce qui fige en partie la composition des ports d'un composant.

3.2.5.2 Instances de port et de composant

Du point de vue du métamodèle RTL, un port est directement accessible par le composant auquel il appartient par la composition *ports*. Lorsque ce composant est instancié, concept COMPONENTINSTANCE, c'est le concept PORTINSTANCE qui est accessible par la composition *portinstance* à partir d'une instance de composant, comme illustré à la figure 3.13. Le lien entre une instance de port et le port qu'il instancie est réalisé par la référence *ref*. De la même façon, une instance de composant référence un composant par le biais de *component*.

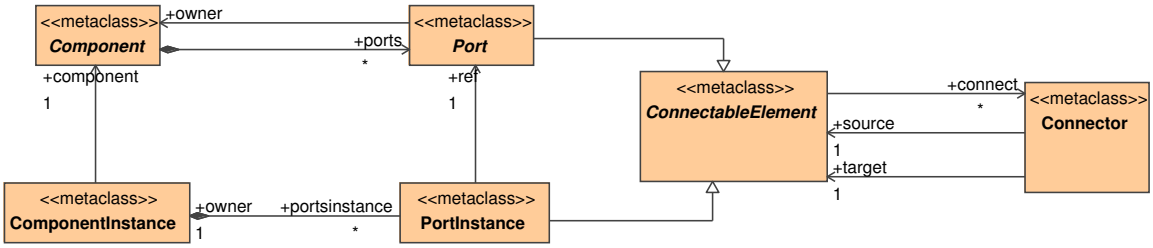


FIG. 3.13: Un composant et une instance de composant contiennent respectivement ses ports et ses instances de ports. Une instance de composant et une instance de port référencent respectivement un composant et un port.

PORT et PORTINSTANCE spécialisent CONNECTABLEELEMENT, qui référence CONNECTOR : chaque connecteur possède un *source* et un *target*. CONNECTOR permet la création de liens entre les éléments d'un modèle RTL, il est utilisé pour véhiculer tous les types de signaux : donnée, contrôle, horloge, etc. Chacun des éléments qu'il connecte peut être connecté à d'autres connecteurs, ce qui permet, par exemple, de connecter le port horloge d'un composant **A** à toutes les instances de port qui référencent les ports d'horloge des composants instanciés par le composant **A**.

3.2.5.3 Bilan de l'interfaçage des composants

Le concept PORT permet de spécifier l'interface d'un composant. Lors de l'instanciation d'un composant, concept COMPONENTINSTANCE, les instances de ports réalisent l'interface avec l'instance du composant.

Nous venons de présenter entre autres le concept de port et savons qu'il référence un concept DATATYPE par le biais de *type*. La section suivante présente les types de données supportés par le métamodèle RTL.

3.2.6 Types de données

En traitement du signal intensif, les types de données peuvent être variés et sortent parfois de l'ordinaire en matière de standard que sont les octets, les entiers ou les flottants. Par exemple, El Hillali [56] préconise l'utilisation d'un encodage sur 4 bits d'un signal numérique issu d'une conversion analogique en numérique d'un signal radar, des flots de données de types complexes sont manipulés dans une application de traitement d'images, etc. Enfin,

les types énumérés sont largement exploités pour l'encodage des automates car à chaque état de cet automate correspond un élément du type énuméré. Pour ces raisons, les différents métamodèles de l'environnement Gaspard exploitent la même base de modélisation en matière de type. Nous reprenons, dans le métamodèle RTL, cette base et l'enrichissons d'informations pour les besoins de la modélisation au niveau RTL.

Au niveau de la métamodélisation, un type de données, concept abstrait DATATYPE est spécialisé en type primitif (PRIMITIVETYPE), complexe (COMPLEXETYPE) ou énuméré (ENUMERATETYPE), comme illustré par la figure 3.14. Nous détaillons ci-dessous chacun de ces trois types :

- PRIMITIVETYPE : un type élémentaire dans le métamodèle RTL. Il permet la représentation des types entier, booléen, etc.
- COMPLEXETYPE : un type complexe composé de différents champs (FIELD) qui référencent chacun un type de données. Toutes les combinaisons sont donc possibles pour la construction d'un type complexe car le métamodèle n'impose aucune contrainte sur la composition du type complexe.
- ENUMERATETYPE : un type énuméré composé d'énumérations (ENUMVALUE) qui font chacune référence à un type de données. Les types énumérés sont couramment utilisés pour la description des automates, ou chaque état de l'automate est associé à une énumération.

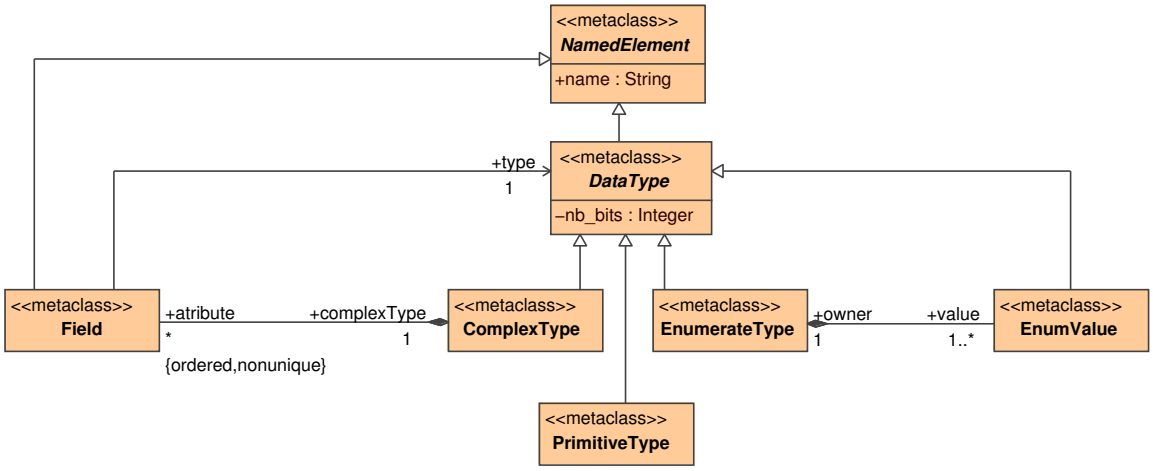


FIG. 3.14: Métamodélisation des types de données utilisée dans les différents métamodèle de l'environnement Gaspard et par conséquent pour les types de données du métamodèle RTL.

En dehors de la nature du type, aucune sémantique n'est attachée à leur comportement. Ce manque de sémantique ne permet pas une modélisation suffisamment riche pour la génération de code des types complexes et primitifs dans les différents langages cibles (La sémantique des types énumérés qui consiste à définir une suite de noms, les ENUMVALUE dans les différents métamodèles de l'environnement Gaspard est suffisante). La solution temporaire de Gaspard est la génération de code des types primitif et complexe directement à partir de leur nom dans un modèle.

L'utilisation du nom des types est suffisante pour la génération d'un code VHDL synthétisable mais n'est pas suffisante pour permettre l'évaluation du circuit généré (évaluation des ressources de calcul consommées sur un FPGA par exemple). En effet, la quantité de ces ressources dépend principalement de la nature du type : plus un type nécessite de bits pour son , plus le circuit dont des éléments utilisent ce type consomme des ressources. C'est le cas, par exemple, des multiplexeurs introduits précédemment. Dans le cadre de l'optimisation des accélérateurs, il est nécessaire d'évaluer au niveau du modèle RTL la quantité de ressources FPGA nécessaires à l'implémentation de ce circuit. Pour cela, nous enrichissons la métamodélisation RTL des types de données par la spécification du nombre de bits nécessaires à leur stockage. Chaque type de données contient un attribut *nb_bits*.

Le métamodèle RTL réutilise les types tels qu'ils sont définis dans les autres métamodèles de l'environnement Gaspard et permet ainsi de modéliser des types primitifs, complexes et énumérés. Nous avons étendu le concept DATATYPE par l'attribut *nb_bits*, ce qui permet de connaître la quantité de registres nécessaires au stockage des types de données.

3.2.7 Concept TILER

Le concept TILER, représenté par la figure 3.15, étend le concept de composant ELEMENTARY car il ne contient pas d'instance de composant et qu'il correspond à une feuille de l'application. Chaque tiler est donc un élément atomique dans un modèle RTL. TILER est spécialisé par INPUTTILER et OUTPUTTILER, qui donnent une direction aux tilers : un tiler d'entrée permet de construire des motifs à partir d'un tableau, un tiler de sortie permet de construire un tableau à partir de motifs. Chaque TILER référence deux concepts SHAPE, l'une correspond à *repetitionSpace* et l'autre à *patternShape*. L'espace de répétition correspond aux motifs que produit ou consomme un tiler et *patternShape* détermine la forme de ces motifs.

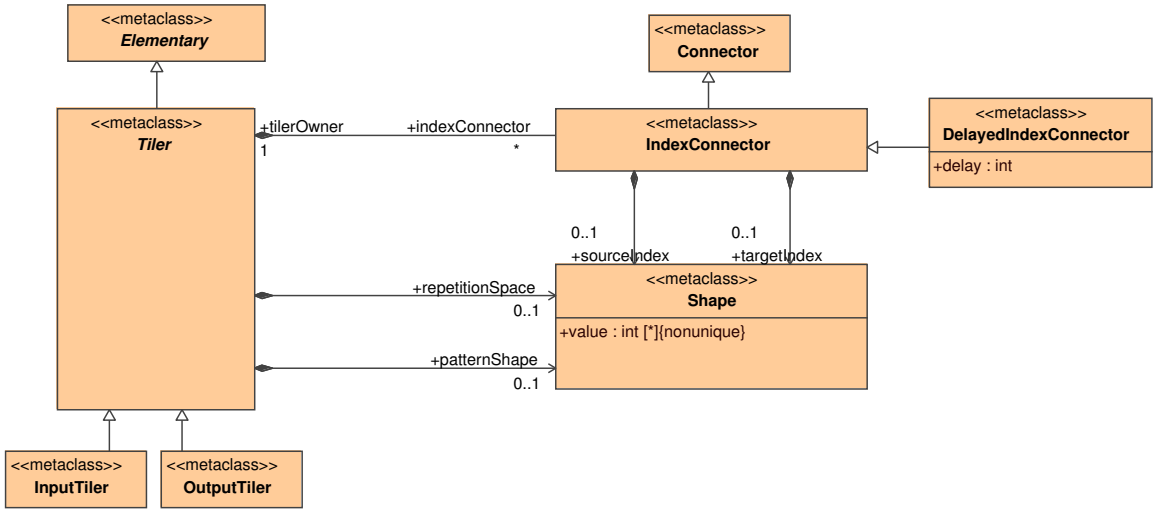


FIG. 3.15: Les tilers sont des composants élémentaires composés de connecteurs qui permettent de définir les dépendances de données entre des données.

Le modèle d'exécution matérielle des applications Gaspard prévoit que les tilers sont

précalculés (section 3.1.2). Ce précalcul implique que les tilers n'existent plus sous une forme ARRAY-OL classique (origine, pavage et ajustage), mais sous la forme de connexions qui lient les données des motifs avec des données du tableau. Les concepts de connecteur indicé INDEXCONNECTOR et de connecteur indicé et retardé permettent d'exprimer ces connexions matérielles.

Le concept INDEXCONNECTOR étend CONNECTOR et permet de ne connecter qu'une partie de deux ports par le biais des indices *sourceIndex* et *targetIndex* : une donnée dans un motif peut directement être connectée à une donnée dans un tableau. Aucune contrainte sur les dimensions des tableaux n'existe, ce qui permet de créer des liens entre des tableaux indépendamment de leur dimension.

Le concept de connecteur indicé avec délai (DELAYEDINDEXCONNECTOR) étend INDEXCONNECTOR et possède un attribut DELAY qui correspond au retard qui existe entre le moment où la donnée est présente dans le tableau et le moment où elle est transférée au motif de sortie. Le concept DELAYEDINDEXCONNECTOR est la modélisation d'un registre à décalage. Un délai est un entier et il représente en matériel un nombre de cycles d'horloge : un délai de 3 signifie que la donnée est retardée de 3 cycles d'horloge.

3.2.8 Bilan

Certains concepts présentés dans cette section sont issus de concepts présents dans différents métamodèles de l'environnement Gaspard, comme le concept de composant. Cependant, de nouveaux concepts ont été définis afin de décrire des accélérateurs matériels qui exécutent des tâches ARRAY-OL. Nous avons par exemple créé les concepts d'horloge, de reset, de registre à décalage, de contrôleur et de multiplexeur.

Le métamodèle RTL nous permet de connaître la composition et la structure des accélérateurs modélisés. Suffisamment de détails existent pour nous permettre de générer le code VHDL de ces accélérateurs, comme présenté plus tard dans ce chapitre. Cependant, le coût d'implémentation de ces accélérateurs reste inconnu car les accélérateurs sont modélisés indépendamment de leur cible d'implémentation.

Afin d'estimer ce coût d'implémentation, nous enrichissons le métamodèle RTL en prenant en compte les caractéristiques des supports d'implémentation et en associant ces caractéristiques au circuit modélisé. La section suivante présente nos résultats.

3.3 Implémentation des accélérateurs sur FPGA

Notre flot de conception prévoit le placement des accélérateurs matériels sur FPGA. Dans la mesure où il n'existe pas de « métamodèle de FPGA », nous proposons notre propre métamodélisation que nous intégrons dans le métamodèle RTL. Cette métamodélisation repose sur des représentations existantes introduites dans le chapitre 1, nous la décrivons dans la section 3.3.1. La métamodélisation des FPGAs permet décrire sous une forme abstraite un FPGA de manière à le manipuler dans notre flot de conception. Elle est par ailleurs associée à des concepts qui expriment la notion de placement des accélérateurs sur FPGA, section 3.3.2.

3.3.1 Métamodélisation des FPGAs

Chaque caractéristique du FPGA que nous souhaitons voir apparaître dans le métamodèle RTL est associée à un concept. Nous découpons ce métamodèle en vues, chacune d'elle

correspond à un niveau de détail auquel le FPGA est considéré[3]. La figure 3.16 illustre notre métamodélisation des FPGAs, les pointillés représentent les quatre différentes vues du FPGA que sont les vues boîte noire, quantitative, topologique et configuration. Dans les sections suivantes, nous présentons ce métamodèle de FPGA en fonction de ces quatre vues.

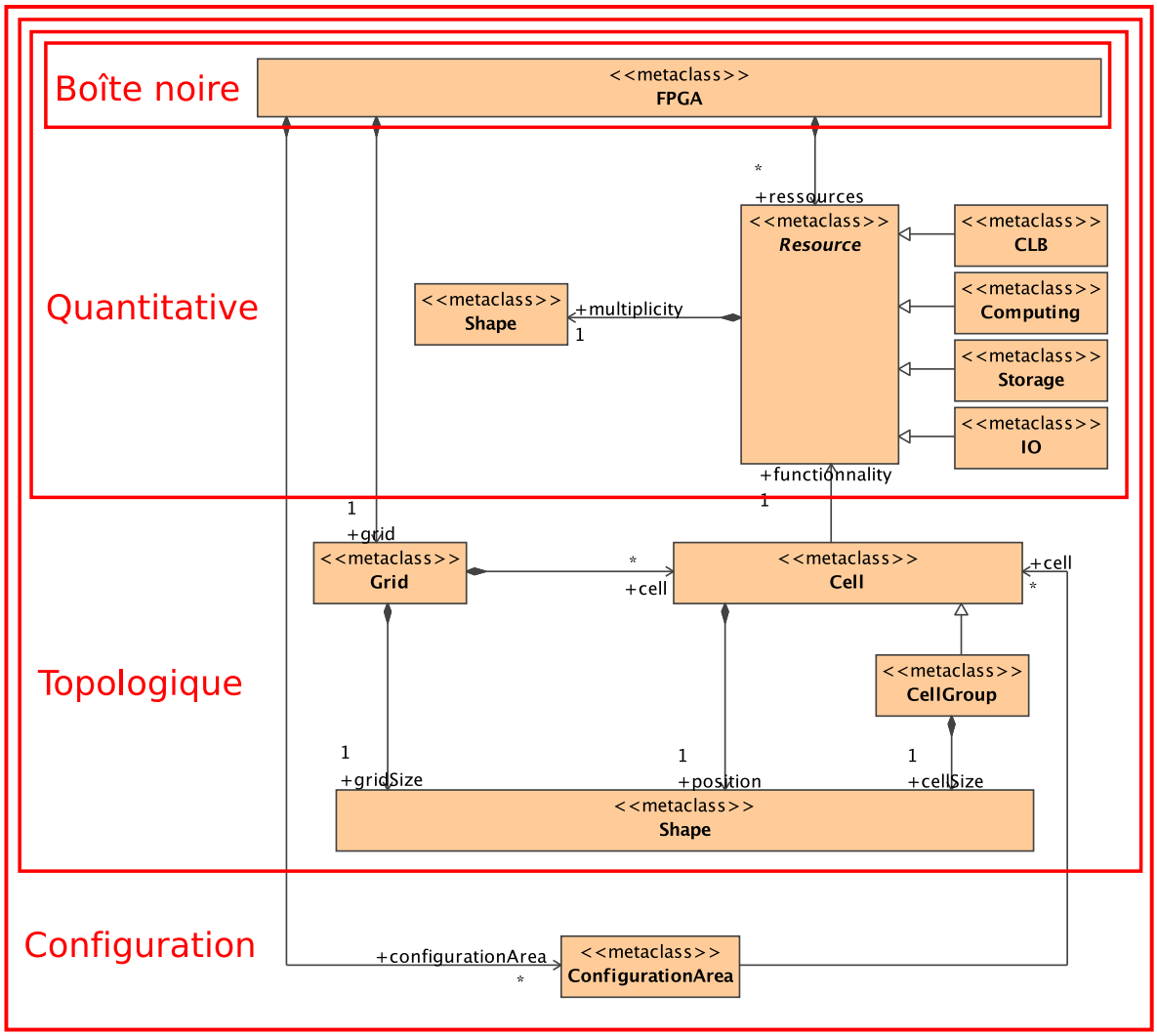


FIG. 3.16: Métamodélisation d'un FPGA et représentations des vues boîte noire, quantitative, topologique et configuration.

3.3.1.1 Vue boîte noire

Cette première vue permet de considérer un FPGA comme une boîte noire, c'est-à-dire sans en préciser les caractéristiques. Ainsi, cette vue permet de représenter tout FPGA, même

[3]La nature « imbriquée » des vues rend leur représentation sur plusieurs figures difficile. Nous avons préféré les représenter de façon groupée sur une même figure tout en les mettant en évidence.

ceux dont les caractéristiques sont inconnues. Cette vue permet, par ailleurs, de désigner le FPGA utilisé.

3.3.1.2 Vue quantitative

La vue quantitative identifie les ressources contenues dans le FPGA. Une ressource est une unité de calcul (COMPUTING), de stockage (STORAGE), configurable (CLB) ou d'entrée et sortie (IO). Nous détaillons ci-dessous chacun de ces concepts :

- Le concept COMPUTING exprime une unité de calcul à grain moyen telle que les multiplieurs (DSP) dans les FPGAs Stratix [7] ou une unité de calcul à gros grain comme les processeurs embarqués dans les FPGAs Virtex 4 de Xilinx [127] ;
- Le concept STORAGE représente une unité de stockage, comme les mémoires RAM de taille 512 bits et 1 k-octets contenues dans les FPGAs Stratix2 :
- Le concept CLB (Configurable Logic Bloc) représente un bloc logique configurable du FPGA. Cette ressource est la base de la modélisation des FPGAs car exprime la reconfigurabilité du FPGA. En fonction d'une configuration, une ressource CLB peut devenir une ressource COMPTING ou STORAGE.
  IO représente une broche du FPGA exploitable par l'utilisateur.

La classification des types de ressources d'un FPGA que nous proposons dans notre métamodèle permet de représenter aussi bien des FPGAs homogènes tels que ceux manipulés par [101, 45] que les FPGAs hétérogènes [108]. Par ailleurs, le concept SHAPE référencé par RESOURCE permet de spécifier sous une forme factorisée le nombre de chaque type de ressources du FPGA. La modélisation d'un FPGA est donc plus compacte et plus simple à réaliser.

En dehors des entrées et sorties, nous ne représentons pas les communications au sein d'un FPGA. En effet, le nombre et la complexité des communications existantes dans les FPGAs rendent difficile le travail de conceptualisation. Par ailleurs, de nombreux travaux cités précédemment ne prennent pas en compte ces ressources de routages [108, 45], assumant le fait que les outils commerciaux savent gérer le routage des signaux entre les ressources. Ainsi, notre métamodèle ne permet pas de modéliser les FPGAs représentés de façon très détaillée dans les outils de placement, comme c'est le cas dans [71, 16, 116, 68]. En effet, chacun de ces travaux prend en compte les ressources de routage des FPGAs.

Nous sommes confrontés au problème de l'abstraction lié à la métamodélisation : il est techniquement possible de décrire chaque type de communication dans notre métamodèle, mais la quantité de concepts à ajouter nuirait à sa clarté. La modélisation des ressources de routage reconfigurables d'un FPGA s'apparente alors à une nouvelle vue du FPGA que nous n'exploitons pas. Cette vue demeure une extension potentielle de notre métamodélisation des FPGAs.

3.3.1.3 Vue topologique

En plus des concepts liés à la vue quantitative, la vue topologique identifie la grille d'un FPGA, concept SHAPE référencé par GRID, dont la dimension correspond à *gridSize*. Cette grille contient des cellules dont la position est indiquée par la composition *position*. Le concept CELLGROUP représente un groupe de cellules indissociables les unes des autres dont la forme est définie par une SHAPE par la composition *cellSize*. Les cellules et les

groupes de cellules référencent une ressource particulière du FPGA (référence *functionnality* sur la figure 3.16).

La référence *functionnality* détermine la fonctionnalité d'une cellule dans la grille du FPGA : la ressource référencée par une cellule peut être un CLB, un COMPUTING, un STORAGE ou un IO. Le principe est identique pour les groupes de cellules, à la différence que c'est le groupe entier de cellules qui référence une seule ressource du FPGA. Les groupes de cellules permettent donc de représenter des blocs de FPGA dédiés à une fonctionnalité précise. Ces blocs peuvent être, par exemple, des colonnes de mémoires ou de multiplieurs pour les Stratix3 [7] ou des rectangles pour les processeurs PowerPC embarqués dans les FPGAs Virtex4 [127].

La combinaison des informations de topologie et des informations de ressources permet de de représenter aussi bien les FPGAs homogènes [116, 45, 68] que ceux hétérogènes, où des colonnes entières du FPGA peuvent être allouées à des blocs mémoires [108].

3.3.1.4 Vue configuration

La vue configuration identifie les zones de configuration d'un FPGA, concept CONFIGURATIONZONE. Chaque zone référence des cellules (ou des groupes de cellules) d'un FPGA qui ne peuvent être reconfigurées indépendamment les unes des autres. Ainsi, plutôt que d'ajouter une troisième dimension à la grille du FPGA pour modéliser la reconfigurabilité, nous créons un concept afin de distinguer clairement la notion de configuration d'une grille.

Le concept de zone de configuration permet la modélisation des FPGAs reconfigurables partiellement et dynamiquement, tel que défini dans [11, 112]. En effet, dans ces travaux, chaque cellule d'un FPGA est reconfigurable indépendamment des autres, ce qui correspond, en terme modélisation, à la création d'autant de concepts CONFIGURATIONZONE que de concepts CELL. Il n'y a pas de contrainte sur la forme des zones de configuration, il est donc possible de modéliser les zones de configuration classiques (c'est-à-dire compactes et de forme rectangulaire) des FPGAs [127], ou bien de modéliser des zones de configuration plus irrégulières, bien que ce soit rarement le cas dans la littérature.

Toutefois, si la gestion des reconfigurations du FPGA doit être gérée de façon entièrement automatisée jusqu'à l'implémentation sur FPGA, il est probable qu'un grand nombre d'informations doivent venir enrichir notre métamodélisation. Nous pouvons pas exemple citer les interfaces de gestion de communication ICAP [8] (Internal configuration access port) des FPGAs de Xilinx, qui permet l'accès à la mémoire de configuration (le stockage du bitstream) et aux registres de configuration du FPGA. De la même façon, les *bus macro* [1] devraient clairement apparaître dans le métamodèle de FPGA car ils permettent de gérer les communications avec des modules placés dynamiquement sur FPGA. La prise en compte de la dynamicité du FPGA devrait donc introduire des extensions à notre modélisation. Des travaux menés actuellement au sein de notre par Imran Quadri tendent à déterminer les besoins exactes de ces extensions [58].

Au final, il s'avère que la dissociation de la grille du FPGA avec sa reconfigurabilité permet de modéliser des architectures plus complexes que celles manipulées par les outils [11, 112]. Cela est possible car le concept de zone de configuration permet d'exprimer des reconfigurations sur des blocs de cellules.

3.3.1.5 Modélisation du FPGA Stratix2s60

Dans le cadre de cette thèse, nous avons modélisé le FPGA Stratix2s60 de Altera. Ce FPGA possède dse CLBs, des multiplieurs, des entrées-sorties, etc. Chacune de ces ressources est intégrée et identifiée dans le modèle à partir des concepts du métamodèle (RESOURCE, CELL, CELLGROUP, etc.). Ce modèle ne décrit pas *exactement* le FPGA, il le décrit à un certain niveau d'abstraction en se focalisant sur différents aspects. Ainsi, certains détails ne sont pas pris en compte, comme les ressources de routages configurables du FPGA ou encore la composition détaillée de ces ressources.

3.3.1.6 Bilan des vues

La modélisation des FPGAs proposée dans cette section permet une représentation des différentes caractéristiques du FPGA. Une décomposition en vue permet de n'utiliser que les informations contenues dans la vue boîte noire si cela est suffisant, ou alors d'utiliser la vue de configuration si nécessaire. Cependant, la description la plus détaillée du FPGA que nous proposons est éloignée des représentations existantes pour les outils de synthèse tels que Quartus d'Altera, ISE de Xilinx ou encore des outils académiques qui permettent d'aller jusqu'à la génération d'un bitsream [68]. En effet, ces outils connaissent la structure et la composition exacte du FPGA : types de LUT utilisés, topologies des connexions dans les cellules reconfigurables, des ressources de communication reconfigurables, etc.

Cependant, ce niveau de description n'est pas celui que nous ciblons car l'objectif de notre flot est d'aider à la conception d'accélérateurs placés sur FPGA afin de pouvoir ensuite utiliser les outils de synthèse et non de les remplacer. En effet, la métamodélisation d'un FPGA permet de conceptualiser ses caractéristiques en ressources, cellules, zones de configuration, etc.

Du point de vue de notre flot de conception, les modèles de FPGAs sont disponibles sous la forme d'une bibliothèque. Cette bibliothèque est extensible par l'utilisateur de notre flot de conception lui-même dans la mesure où il dispose des outils de modélisation adéquats.

Le placement d'un accélérateur sur un FPGA est aussi exprimé sous la forme de modèles. La métamodélisation de ces placements est présentée dans la section suivante.

3.3.2 Implémentations des accélérateurs sur FPGAs

Afin de modéliser le placement des accélérateurs sur ces FPGAs, nous deux concepts d'implémentations : le premier correspond aux ressources du FPGA consommées par un composant de l'accélérateur, le second correspond au placement complet d'un accélérateur sur FPGA.

3.3.2.1 Métamodélisation des ressources d'un FPGA consommées par un composant

Afin de réduire les cycles d'optimisation et de procéder à des optimisations rapides de l'accélérateur, il est nécessaire de connaître, ou du moins de pouvoir éstimer, le coût d'implémentation de chaque composant d'un accélérateur avant même la phase de synthèse[4].

[4]Il est à noter qu'en comparant les ressources référencées par la vue quantitative du FPGA avec les ressources nécessaires à l'implémentation du composant de plus haut niveau de l'accélérateur, il est possible de déterminer si le FPGA est capable d'implémenter cet accélérateur. Cette information est nécessaire mais insuffisante pour l'optimisation des accélérateurs.

Nous réalisons donc cette estimation au niveau du métamodèle RTL et avant même l'étape de génération de code car cela nous permet de rester dans le domaine de l'IDM et de tirer profit de ses avantages en terme d'outillage notamment.

Concept COMPONENTFPGAIMPLEMENTATION Lors de la synthèse d'un composant d'un accélérateur sur FPGA, ce composant consomme les ressources du FPGA nécessaires à son implémentation : chaque COMPONENT est implémenté par plusieurs RESOURCES du FPGA. L'ensemble des ressources nécessaires à l'implémentation de ce composant est appelé COMPONENTFPGAIMPLEMENTATION. Chaque COMPONENTFPGAIMPLEMENTATION est associé à un composant contenu dans l'accélérateur matériel et représente les ressources nécessaires à l'implémentation de ce composant sur le FPGA, de la même façon que [108, 3, 55] disposent des caractéristiques d'implémentation de chaque tâche. Le concept COMPONENTFPGAIMPLEMENTATION est représenté à la figure 3.17, et constitue une nouvelle vue du FPGA dans la mesure où une nouvelle décomposition des ressources du FPGA est réalisée. Cependant, et contrairement aux vues boîte noire, quantitative, topologique et configuration, cette vue dépend d'un composant de l'accélérateur et d'un FPGA.

FIG. 3.17: Un composant est implémenté sur un FPGA par le biais du concept COMPONENTFPGAIMPLEMENTATION, qui référence les ressources consommées par cette implémentation.

Le concept COMPONENT du métamodèle RTL est abstrait et se concrétise en composant élémentaire, composé, répétitif, etc. La construction des concepts COMPONENTFPGAIMPLEMENTATION pour chaque composant d'un accélérateur dépend donc en grande partie de la nature de ce composant.

Construction des COMPONENTFPGAIMPLEMENTATION pour chaque type de composant Le concept de tâche élémentaire référence un IP dans une bibliothèque de composants. En enrichissant cette bibliothèque par les caractéristiques d'implémentation de chaque IP sur différents FPGAs, il devient possible de créer directement les COMPONENTFPGAIMPLEMENTATION associées à ces IPs. Il peut exister différentes façons d'implémenter un même composant sur un même FPGA. Sur un Stratix2 d'Altera, une multiplication est implémentée par défaut sur un multiplieur existant dans le FPGA, mais cette multiplication peut aussi être implémentée par des fonctions combinatoires via des LUTs. Chaque composant possède donc potentiellement plusieurs COMPONENTFPGAIMPLEMENTATION.

En ce qui concerne les autres types de composants du métamodèle RTL, la connaissance de leur structure permet d'estimer les ressources nécessaires à leur implémentation, et donc de créer les COMPONENTFPGAIMPLEMENTATION qui leur sont associées. La structure d'un composant est définie par des instanciations de composants (les ressources nécessaires à l'implémentation d'une instance de composant sont les mêmes que celles nécessaires pour le composant). Nous construisons donc la hiérarchie de l'accélérateur en partant du plus bas niveau de hiérarchie. Ce niveau correspond à l'instanciation de tâches élémentaires qui référencent des IPs, dont le COMPONENTFPGAIMPLEMENTATION est connu. Ainsi, pour les composants composés ou répétitifs, le coût en ressources dépend des instances de composants mais aussi des contrôleurs, des multiplexeurs, etc. qui sont utilisés. L'estimation du coût en ressources de chacun de ces concepts dépend du type de données manipulées et plus particulièrement du nombre de registres nécessaires au stockage de ces données. Afin d'estimer le coût en ressources de chaque concept électronique du métamodèle RTL, nous enrichissons les concepts par des expressions mathématiques simples, tels que définies dans [70]. Chaque expression prend en compte la quantité de données manipulées par le concept auquel elle est rattachée et le nombre de registres nécessaires au stockage de ces données.

Estimation des ressources consommées par les différents concepts du métamodèle RTL L'estimation des ressources utilisées pour l'implémentation sur FPGA d'un concept diffère selon le concept : l'estimation pour un contrôleur est différente de celle d'un multiplexeur. Cette estimation prend en compte la quantité de données véhiculées par ce concept, le type de chacune de ces données et l'implémentation matérielle liée à ce concept.

Nous illustrons le principe de l'estimation par le biais du concept MUTIPLEXER. Un multiplexeur est un composant de routage qui sélectionne une entrée parmi plusieurs pour la router jusqu'à sa sortie, cela en fonction de son entrée de contrôle. Du point de vue de l'implémentation, un multiplexeur est réalisé par de la logique combinatoire sur des blocs logiques et configurables (CLBs) du FPGA. Les LUTs sont la base de la partie combinatoire des CLBs et il en existe différentes sortes. Généralement, les LUTs sont caractérisées par le nombre e de leurs entrées et permettent de d'exprimer par le biais de la logique combinatoire un routage de e entrées vers 1 sortie. Si l'on considère l'implémentation d'un multiplexeur sur ces LUTs, il est utile de décomposer ce multiplexeur en sous-multiplexeurs : chaque sous-multiplexeur est en charge du routage d'un bit de sortie du multiplexeur initialement composé de nb_bits bits. L'implémentation d'un sous-multiplexeur avec n entrées nécessite donc l'utilisation de $sup(\frac{n}{e})$ LUTs. Pour l'implémentation complète du multiplexeur, il est nécessaire d'utiliser $nb\_bits \times sup(\frac{n}{e})$ LUTs.

Une LUT couramment utilisée est la LUT à 4 entrées car elle correspond à un bon compromis entre le coût du routage de données jusqu'à ses entrées et sa puissance de calcul. Cette LUT permet un routage de 4 entrées vers 1 sortie. Pour un multiplexeur qui possède 10 entrées et une sortie encodée sur 32 bits, nous pouvons estimer que $32 \times sup(\frac{10}{4})$ (soit 96) LUTs sont nécessaires pour son implémentation. Ce résultat demeure une estimation car le synthétiseur est parfois en mesure de réaliser des optimisations (telles que la propagation de constantes) qui ont pour effet de réduire le nombre de combinaisons combinatoires et donc le nombre de LUTs nécessaires. Le passage d'une quantité de ressources exprimée en LUTs en une quantité de ressources exprimée en CLBs nécessite la connaissance de la structure des CLBs d'un FPGA. A la base, les CLBs sont composés d'un ensemble de LUTs du même

type, la relation est donc triviale. Cependant, les architectures FPGA évoluent vers l'intégration de CLBs plus complexes contenant par exemple des LUTs adaptatives (Adaptative LUT, ALU) [7] dont le nombre d'entrées est paramétrable par le synthétiseur. Dans ce cas, le passage d'un nombre de LUTs en nombre de CLBs devient plus délicat et il devient plus difficile encore d'évaluer la marge d'erreur de nos estimations.

Du point de vue du métamodèle, les expressions qui estiment le nombre de LUTs nécessaires à l'implémentation d'un concept sont directement référencées par ces concepts. Le passage du nombre de LUTs en nombre de CLBs est déterminé par l'expression référencée par le concept CLB dans la métamodélisation du FPGA.

L'estimation du nombre de CLBs nécessaires à l'implémentation d'un concept du métamodèle RTL permet de créer de manière plus précise les concepts COMPONENTFPGAIMPLEMENTATION. La prise en compte des concepts électroniques mis en œuvre pour l'exécution matérielle de ARRAY-OL permet d'améliorer l'estimation des ressources nécessaires à l'implémentation d'un accélérateur et permet ainsi de gérer plus précisément les placements sur FPGA.

Illustration La figure 3.18 illustre l'utilisation du concept COMPONENTFPGAIMPLEMENTATION : la gauche de la figure représente les composants de l'accélérateur matériel, la droite les ressources du FPGA nécessaires à l'implémentation de ces composants, c'est-à-dire le concept COMPONENTFPGAIMPLEMENTATION. Sur cette figure, la couleur d'une ressource correspond à la couleur du composant qu'elle sert à implémenter. Six ressources sont nécessaires pour l'implémentation du premier composant, une seule pour le deuxième, etc.

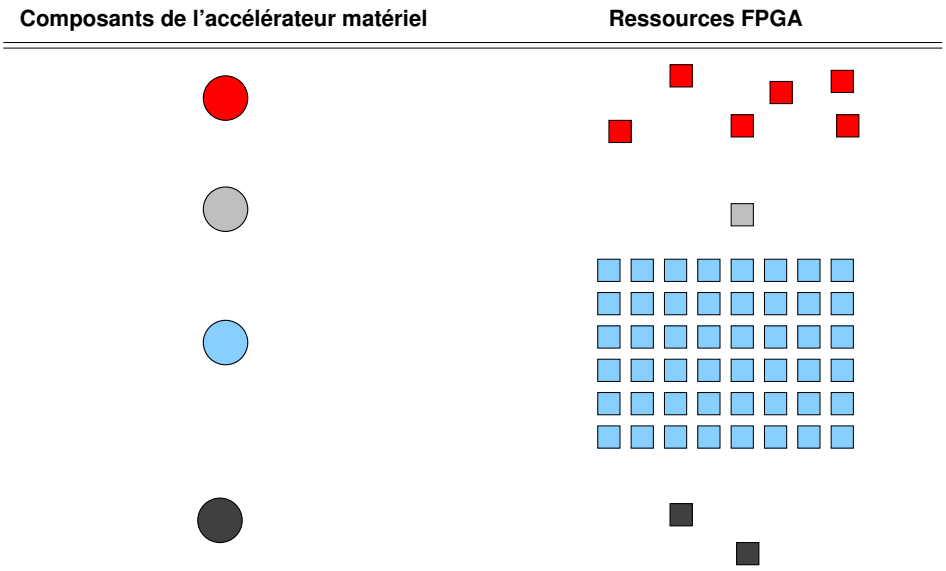


FIG. 3.18: Les composants de l'accélérateur et les ressources nécessaires à leur implémentation.

En allant plus loin dans la description des vues représentant l'implémentation d'un accélérateur sur un FPGA, il est possible d'associer un placement de l'accélérateur sur le FPGA. Cette notion de placement reprend des concepts du FPGA décrits dans la vue topologique,

dont la notion de position dans une grille, et introduit une nouvelle vue du FPGA présentée dans la section suivante.

3.3.2.2 Métamodélisation du placement d'un accélérateur sur un FPGA

Le placement d'un accélérateur sur un FPGA correspond au concept INSTANCEFPGAPLACEMENT, et reprend la notion de vue topologique du FPGA. Contrairement au concept COMPONENTFPGAIMPLEMENTATION introduit dans la section précédente, il ne s'agit pas d'implémenter individuellement chaque composant de l'accélérateur mais de placer l'accélérateur dans sa globalité. Pour cela, il est nécessaire d'utiliser la hiérarchie de cet accélérateur, construite par instanciation de composants à partir du composant du plus haut niveau de hiérarchie.

Concept INSTANCEFPGAPLACEMENT La figure 3.19 représente le concept INSTANCEFPGAPLACEMENT dans le métamodèle RTL. Chaque instance de composant référence un ou plusieurs INSTANCEFPGAPLACEMENT et chaque INSTANCEFPGAPLACEMENT référence des cellules ou des groupes de cellules du FPGA. INSTANCEFPGAPLACEMENT peut aussi contenir une origine et une dimension dans la grille de cellules du FPGA, qui permettent de déterminer une forme rectangulaire qui englobe la totalité des cellules référencées par INSTANCEFPGAPLACEMENT. Cette forme est celle couramment utilisée par les outils de placement sur FPGA, et permet de modéliser les placements de tâches de formes rectangulaires tels que définis dans les outils présentés précédemment [11, 45]. Il est par ailleurs possible de définir plusieurs zones de placement pour chaque instance de composant, cela permet d'exploiter au maximum les les ressources du FPGA, comme c'est le cas [108].

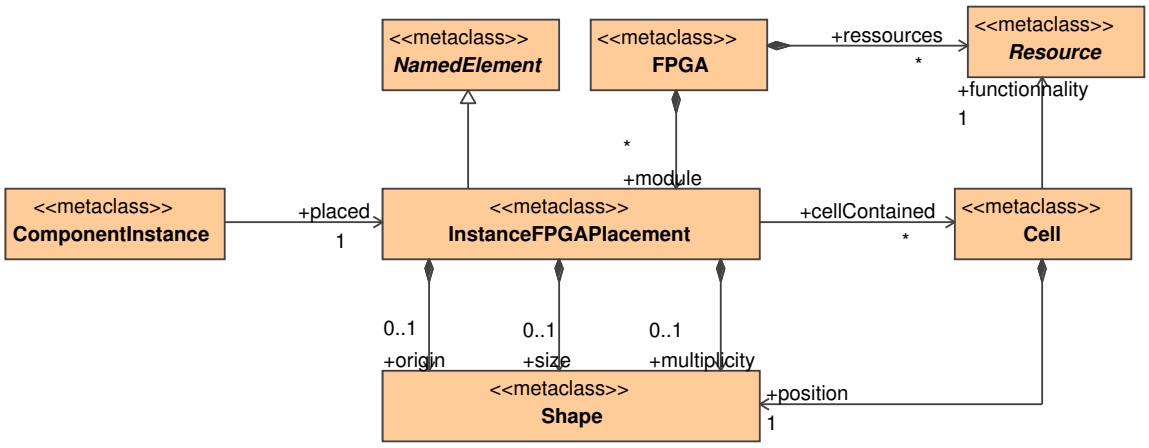


FIG. 3.19: Une instance de composant référence une INSTANCEFPGAPLACEMENT, qui correspond à un groupement de cellules du FPGA.

Par ailleurs, des degrés de liberté peuvent être accordés à ces zones de placements si les liens de composition *origin* et *size* ne sont pas utilisés : il est donc possible de définir l'origine sans pour autant contraindre l'outil de synthèse à une forme de placement spécifique, ou de définir une dimension de cette forme de placement sans en donner les coordonnées.

Le maximum de liberté est accordé à l'outil de placement lorsqu'aucune information n'est accordée. Dans ce cas, l'outil définit la forme et la zone du placement.

Placement régulier de tâches répétitives La factorisation de l'expression du placement permet de conserver l'expression du parallélisme de l'accélérateur lors de son implémentation sur FPGA. L'expression de ce parallélisme dans l'implémentation permet à l'outil de synthèse de réutiliser des résultats de synthèse d'un composant pour chacune des instances répétées de ce composant lors d'une synthèse incrémentale : le même composant optimisé peut être réutilisé à différents endroits. Au-delà de la réduction de l'effort de synthèse par l'outil, la conservation du parallélisme permet dans certains cas d'améliorer ces résultats de synthèse, comme le montre Moeller [61] pour le placement d'architectures systoliques. Dans ces travaux, Moeller améliore les résultats de l'implémentation sur FPGA d'un filtre à l'aide de contraintes de placements qui prennent en compte la régularité de ce filtre. Ces contraintes de placement sont exprimées dans un fichier de contraintes utilisateur. Dans notre métamodèle, la référence *multiplicity* permet le placement régulier et compact d'instances de composants sur un FPGA. Pour cela, chaque concept INSTANCEFPGAPLACEMENT qui contient une SHAPE par le biais de la composition multiplicité verra l'instance de composant COMPONENTINSTANCE qu'il référence être répliqué autant de fois que cette SHAPE. Dans la mesure où il est question de placement sur une grille bidimensionnelle, nous limitons à deux le nombre de dimensions de cette SHAPE. L'instance de composant est répliquée de manière à réaliser un placement compact sur le FPGA, les bords d'une zone de placement étant collés aux bords des autres zones de placement. Cette référence est donc adaptée aux instanciations multiples dans les composants répétitifs. Il est à noter que l'utilisation de ARRAY-OL pour la spécification des placements permet de réaliser des placements réguliers plus complexes tels que ceux utilisés dans Gaspard lors du placement de tâches sur des architectures. Il devient alors possible de créer des espaces entre chaque répétition de l'instance. Les besoins de réaliser de tels placements sur FPGA à l'aide ARRAY-OL restent à être évalués.

Décomposition en arbre de l'accélérateur et illustration Afin d'associer chaque instance de composant à un placement, il est nécessaire de connaître la structure du circuit que nous implémentons. Pour cela, nous représentons un accélérateur sous la forme d'un arbre, dont le sommet est l'instance principale, les feuilles des instances de composant élémentaires et les nœuds des instances de composants hiérarchiques. Des exemples d'arbres sont représentés sur la gauche de la figure 3.20. Chaque instance de composant (représentées sur la figure par des cercles) référence un concept INSTANCEFPGAPLACEMENT, qui référence les cellules du FPGA utilisées pour l'implémentation de l'instance de composant.

La partie a de la figure 3.20 illustre un arbre et le placement de cet arbre sur un FPGA. Sur cette figure, chaque instance de composant est représentée par une couleur différente et son placement sur le FPGA est représenté par un rectangle de la même couleur. Le placement d'une instance de composant à un niveau de hiérarchie est réalisé de manière à englober les cellules utilisées par les instances du niveau hiérarchique inférieur. Ainsi, sur cette figure, la zone de placement de l'instance principale englobe les zones de placement des feuilles. Cet arbre correspond à l'accélérateur qui exécute l'exemple du filtre d'image, tel que représenté à la figure 3.5. Les six feuilles représentent : le tiler d'entrée, le tiler de sorties et les quatre unités de calcul. Ces dernières sont identiques et consomment donc le même nombre de

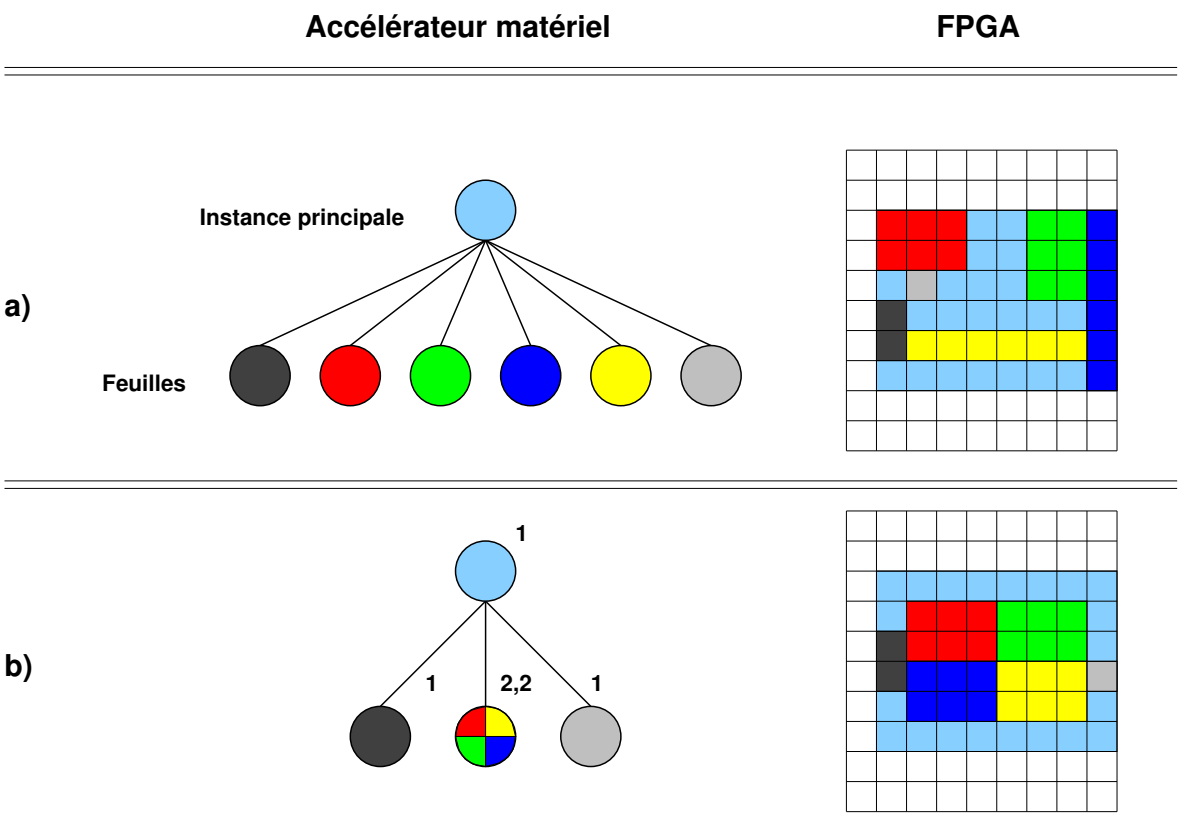


FIG. 3.20: L'accélérateur, sous la forme d'un arbre, est placé sur un FPGA.

ressources du FPGA pour une implémentation. Ces quatre unités de calcul consomment chacun six ressources du FPGA, mais la forme des zones de placement diffère car l'arbre (partie gauche de la figure 3.20) n'exprime pas de régularité.

La partie b de la figure 3.20 représente elle aussi l'arbre correspondant au filtre d'image. Cependant, la répétition est directement exprimée dans cet arbre par l'indice de répétition $[(2, 2)]$. Sur cette figure, nous faisons apparaître les quatre unités de calcul par quatre couleurs dans le rond correspondant à la tâche répétée. La partie droite de la figure représente le placement régulier de cette répétition : chacune des quatre unités de calcul consomme le même nombre de ressources du FPGA et la forme des zones de placement est identique : le placement est régulier. Un placement régulier est exprimé par la référence *multiplicity* du concept INSTANCEFPGAPLACEMENT dans le métamodèle RTL.

Les placements réalisés dans les cas a et b de la figure 3.20 n'influent pas sur le comportement fonctionnel du circuit, mais sur l'efficacité de son implémentation en terme de ressources consommées ou de fréquence de fonctionnement notamment. Par ailleurs, un gain temps de synthèse est attendu lors de la phase de placement [45]. L'objectif de ces placements n'est pas de remplacer les outils de synthèse, qui sont les seuls outils à pouvoir générer les bitstream, mais de les aider dans cette synthèse par des placements qui prennent en compte la structure du circuit à implémenter.

La surface du FPGA, ou plutôt le rectangle de cellules, nécessaire à l'implémentation d'un accélérateur dépend de l'instance principale et du concept INSTANCEFPGAPLACE-

MENT qu'elle référence. Ainsi, il est possible de ne consommer qu'une partie des cellules d'un FPGA lors de l'implémentation d'un accélérateur.

3.3.3 Bilan des vues issues du placement d'un accélérateur sur un FPGA

Dans cette section, nous avons conceptualisé puis illustré chacune des caractéristiques des FPGAs[5] et nous avons associé des vues aux différentes caractéristiques du FPGA.

Nous avons introduit le concept COMPONENTFPGAIMPLEMENTATION qui, associé à un composant de l'accélérateur, définit la quantité de ressources du FPGA nécessaires à l'implémentation de ce composant. Les IPs sur lesquels sont déployées les tâches élémentaires sont enrichis par des COMPONENTFPGAIMPLEMENTATION et servent à la construction d'autres COMPONENTFPGAIMPLEMENTATION (référencées par les composants répétitifs ou composés notamment). Ces dernière concepts COMPONENTFPGAIMPLEMENTATION sont construits à partir d'une estimation des ressources du FPGA nécessaires à leur implémentation et dépendent entre autres des concepts tels que les multiplexeurs, contrôleurs, etc. Pour cela, nous disposons d'expressions mathématiques qui estiment la quantité de ces ressources pour chacun de ces concepts ; ces expressions prennent en compte la quantité de données et le nombre de registres nécessaires à leur stockage.

Le concept INSTANCEFPGAPLACEMENT permet la représentation d'un FPGA sous la forme de modules et de sous-modules dont les dimensions et les structures dépendent de l'accélérateur. Cette représentation correspond à une nouvelle vue du FPGA, construite sur mesure pour l'accélérateur qu'il implémente. INSTANCEFPGAPLACEMENT peut reprendre la description d'un placement régulier sur le FPGA. La description de la régularité d'un placement est réalisée par une multiplicité, ce qui permet de décrire des placements réguliers et compacts (chaque zone issue de ce placement régulier possède au moins une frontière commune avec une autre zone). Cependant, la multiplicité dans le placement ne permet pas de décrire des placements complexes tels que le permet ARRAY-OL. Ces placements sont similaires à ceux effectués lors de l'association entre une application et une architecture dans l'environnement Gaspard (plus spécifiquement lors des distributions de tâches et de données). Il est donc intéressant d'évaluer l'intérêt de réaliser de tels placements sur FPGA et de permettre, si les placements qui en découlent sont intéressants, de les réaliser dans le métamodèle RTL. Par manque de temps, nous n'avons considéré ces placements.

La notion de placement tel qu'elle vient d'être décrite est différente de celle réalisée lors du placement d'une application sur une architecture dans le métamodèle Deployed. La première différence provient de la granularité du placement : le placement d'une application sur une architecture est réalisé sur des unités fonctionnelles telles que les processeurs ou les accélérateurs alors que le placement sur FPGA fait intervenir des placements sur des cellules ou groupes de cellules à grain fin. Les modules INSTANCEFPGAPLACEMENT, issus de la décomposition d'un FPGA lors du placement de l'accélérateur, sont plus fins que les unités fonctionnelles manipulées dans Gaspard.

La seconde différence provient de la nature même des placements : dans Gaspard ils sont réalisés sur des unités fonctionnelles, et dans notre métamodèle ils le sont sur des cellules d'un FPGA. Le placement dans Gaspard ne prend pas en compte ni la topologie ni l'organisation sur une puce des unités fonctionnelles, alors que le placement sur un FPGA considère la topologie des cellules.

[5] A l'exception des zones de reconfiguration : les placements réalisés sur les FPGA sont donc statiques.

Cependant, le récent standard pour la modélisation des systèmes embarqués et temps réel, MARTE, (sur lequel se base la modélisation UML de Gaspard) introduit la notion de placement géographique des éléments de l'architecture. Ce placement est complémentaire à celui que nous proposons dans le métamodèle RTL car il s'applique à un niveau différent de finesse : un placement dans MARTE s'applique à des unités fonctionnelles, tandis qu'un placement dans le métamodèle RTL s'applique sur des ressources reconfigurables élémentaires.

La figure 3.21, issue de [104], représente une architecture fonctionnelle contenant quatre processeurs, une mémoire SDRAM, un DMA, une batterie et un bus qui connecte chacun de ces éléments. Les informations de position sur chaque unité fonctionnelle permettent de placer ces unités sur une grille. Cette représentation est valable lorsqu'une décomposition en grille du support d'implémentation est possible. C'est le cas des FPGAs tel qu'ils sont modélisés dans le métamodèle RTL. Ainsi, dans la mesure où la modélisation dans Gaspard tend à s'accorder avec la modélisation dans MARTE, une extension de Gaspard vers des placements topologiques est envisageable. Cette extension permettrait alors de spécifier des contraintes de placement dans une modélisation UML dans Gaspard et de les prendre en compte dans le métamodèle RTL.

Reprenons l'exemple du filtre d'images et son placement sur FPGA. La boîte englobante, en bleu sur les figures 3.18 et 3.20, représente l'accélérateur, identifié dans un modèle Gaspard comme une unité fonctionnelle. Le placement sur FPGA de cette unité fonctionnelle dans une extension de Gaspard intégrant la notion de placement topologique du profil MARTE permettrait de spécifier la position et la forme de cette zone de placement sur le FPGA.

Le métamodèle RTL permet la représentation d'un accélérateur matériel, d'un FPGA et du placement de l'un sur l'autre. Ces descriptions existent sous la forme de modèles et ne sont pas directement exploitables par des outils de synthèse pour FPGA. L'utilisation de ces outils commerciaux nécessite la génération d'un code qu'ils reconnaissent et que nous présentons dans le chapitre suivant.

3.4 Conclusion

Ce chapitre présente dans les détails le modèle d'exécution matérielle des applications Gaspard qui est soumis à deux restrictions :

- les dépendances de données dans l'exécution du parallélisme de tâches sont sujettes à des restrictions quant à l'utilisation des IPs dans le déploiement des tâches élémentaires. L'utilisateur de notre flot de conception doit prendre en compte les latences qu'introduisent ces IPs (en terme de cycles d'horloge) afin de garantir la validité du circuit généré
- l'exécution séquentielle du parallélisme de données n'est possible qu'au plus haut niveau de hiérarchie.

Le métamodèle RTL reprend les concepts introduits par cette exécution matérielle et permet donc de décrire un accélérateur qui exécute les applications Gaspard sous la forme d'un *modèle conforme au métamodèle RTL*. Ce métamodèle est enrichi des concepts liés aux FPGAs, aux implémentations et aux placements. Un accélérateur peut alors être placé sur un FPGA.

Le métamodèle RTL représente par ailleurs la première étape de la construction de notre flot de conception guidé par l'Ingénierie Dirigée par les Modèles. Nous présentons la gé-

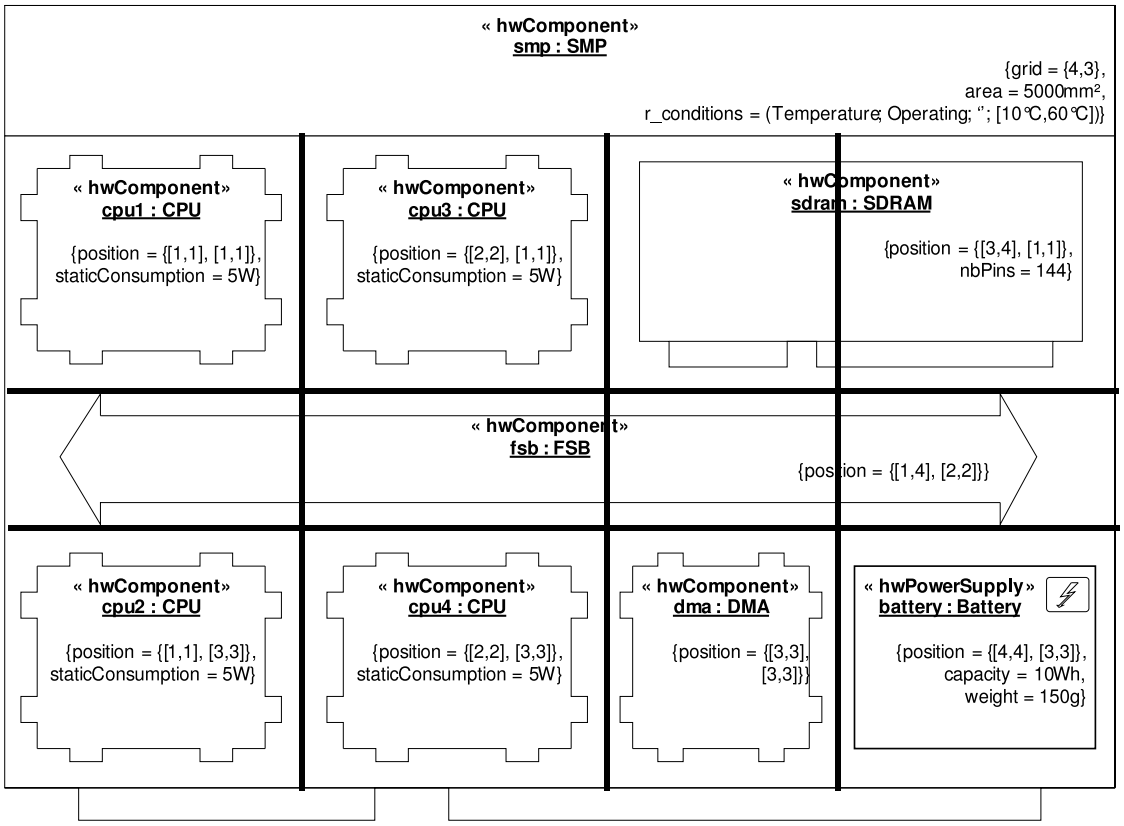


FIG. 3.21: Placement topologique d'une architecture fonctionnelle dans MARTE. Les informations de position permettent de définir les zones où sont placées les unités fonctionnelles sur un SoC.

nération de code VHDL depuis ce métamodèle dans le chapitre suivant et présentons la transformation de modèles « Deployed vers RTL » dans le chapitre 5.

Chapitre 4

Génération de code depuis le métamodèle RTL

Contents

Ce chapitre présente le dernier maillon de notre flot de conception qui consiste générer du code depuis le métamodèle RTL. La génération de code est assimilée à une transformation *modèle vers texte*, appelée *RTL2VHDL* dans notre flot de conception. RTL2VHDL est représentée par l'ellipse sur la figure 4.1, elle génère à la fois le code VHDL d'un accélérateur décrit par un modèle RTL et le code correspondant à son placement sur FPGA. Le code généré marque la fin de l'IDM et le début de l'utilisation d'outils « classiques », nous illustrons donc des résultats de simulation et de synthèse de code générés sur des outils commerciaux.

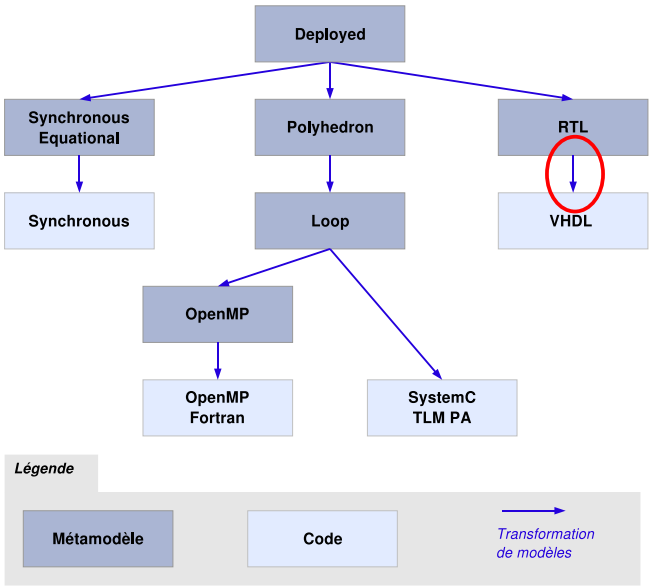


FIG. 4.1: L'environnement Gaspard et la génération de code VHDL depuis un modèle RTL.

Nous détaillons dans la section 4.1 le principe de génération de code en IDM et nous développons plus particulièrement la méthodologie utilisée dans l'environnement Gaspard. La section 4.2 présente la génération de code VHDL et la section 4.3 la génération des fichiers de contraintes à partir d'un modèle conforme au métamodèle RTL.

4.1 Génération de code en IDM

Un modèle RTL n'est pas directement exploitable par les outils classiques de simulation et synthèse, contrairement au VHDL. Il est donc nécessaire de procéder à une génération de code du métamodèle RTL. Nous présentons dans une première partie les différentes techniques existantes et nous présentons la solution choisie dans une seconde partie.

4.1.1 Choix de mise en œuvre de la génération de code VHDL

Deux solutions permettent de générer du code VHDL à partir du métamodèle RTL :

– la première solution consiste à métamodéliser la grammaire du langage VHDL et à réaliser une transformation de modèles depuis le métamodèle RTL vers un « métamodèle VHDL », à partir duquel une génération de code VHDL est possible. Cependant,

le chapitre 1 montre que seules les machines à états finis sont gérées dans les travaux dans ce domaine [6], des extensions sont donc nécessaires pour prendre en considération l'ensemble de la syntaxe VHDL utilisée pour la génération de code des concepts du métamodèle RTL. Ces extensions concernent aussi bien le métamodèle VHDL que les templates de génération de code associées. Au final, il s'avère que l'utilisation d'un métamodèle VHDL repousse la génération de code en contrepartie d'un effort de développement conséquent puisque la quasi-totalité de la chaîne doit être réalisée (nous rappelons qu'il n'existe pas de métamodèle VHDL standard). Il est en effet nécessaire de créer le métamodèle VHDL, la transformation de modèles associée vers ce métamodèle et la génération de code, comme illustré sur la gauche de la figure 4.2 ;

– la seconde solution consiste à générer le code VHDL directement depuis le métamodèle RTL. L'avantage de cette méthode, représentée sur la droite de la figure 4.2, est qu'elle ne fait intervenir aucun autre métamodèle dans le flot de conception. Par ailleurs, le niveau de description proche du matériel du métamodèle RTL permet la génération de code HDL par le biais de templates relativement simples. Pour ces raisons, c'est cette solution que nous adoptons.

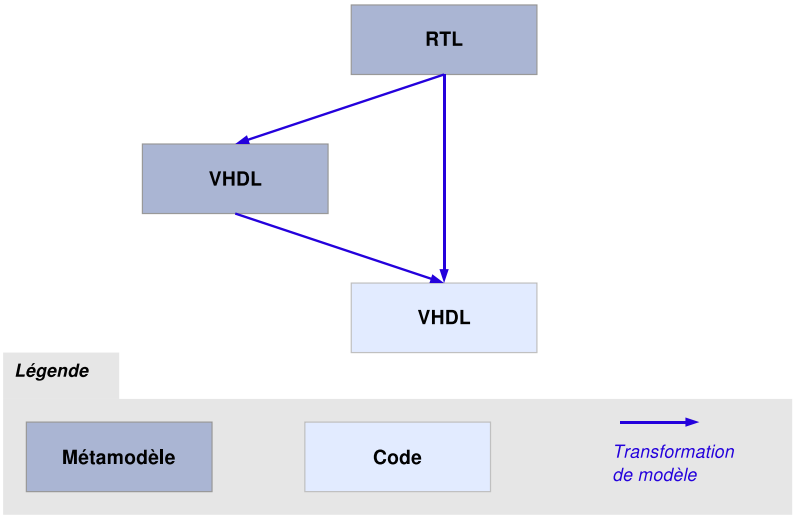


FIG. 4.2: Deux possibilités de générer du code VHDL, nous retenons la solution qui consiste à la générer directement depuis le métamodèle RTL.

Nous choisissons donc de générer le code VHDL directement depuis le métamodèle RTL. Non seulement cette solution n'est pas moins générique que la solution consistant à utiliser un métamodèle intermédiaire VHDL, mais elle limite la quantité de développements à réaliser. Notre choix serait toutefois à reconsidérer dans le cas d'une standardisation d'un métamodèle VHDL et de la génération de code VHDL associée.

La transformation d'un modèle en texte est supportée par plusieurs moteurs de transformations dont l'objectif principal reste cependant la transformation de modèles : Kermeta [119], QVT [93], ATL [62], etc. L'inconvénient est que, en pratique, seule la partie impérative du moteur de transformation est utilisée, un langage classique (Java par exemple) permet de faire la même chose en matière de génération de code [42, 2].

L'environnement Gaspard prend le parti de générer du code à partir de templates JET [42] (Java Emitter Templates), dont nous présentons le principe de fonctionnement dans la section suivante.

4.1.2 Fonctionnement de JET

L'objectif premier de JET (Java Emitter Templates) [42] est la génération de modèles d'implémentation à partir de modèles de définition en Ecore, au sein d'Eclipse. Mais JET peut aussi être utilisé, de façon plus générale, pour la génération de code à partir d'un modèle.

La figure 4.3 illustre le fonctionnement de JET lors d'une transformation de modèle (à gauche de la figure) en texte (à droite de la figure). Le chargement du modèle d'entrée est réalisé par JET GENERATOR, qui se charge d'appliquer les JET GENLETS et de sauvegarder le code généré dans un fichier. JET GENLET est une classe Java avec une méthode GENERATE qui prend en argument le modèle (ou une partie du modèle) et qui retourne une chaîne de caractère. Les JET GENLETS sont générées à partir des JET TEMPLATE, scripts qui définissent les relations entre un concept dans un modèle et le code.

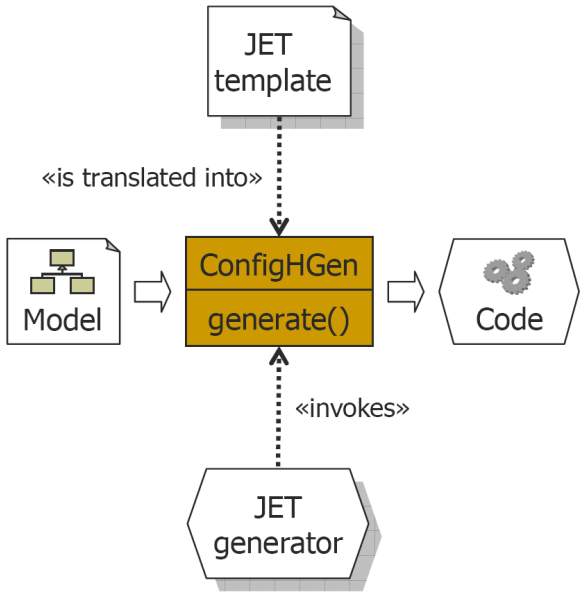


FIG. 4.3: Flot de conception de JET qui permet de générer du code à partir d'un modèle en Ecore.

En tant que concepteurs du flot de conception pour accélérateurs matériels, nous sommes des utilisateurs de JET. Nous écrivons donc les templates JET associés aux concepts du métamodèle RTL. Nous rappelons que ces templates permettent la génération des classes Java, à partir desquelles la transformation d'un modèle vers du texte est réalisée.

Grâce aux templates, JET permet d'écrire du texte paramétré par du code Java. Un template JET est donc composé de deux types d'éléments : du texte écrit dans le code cible souhaité (ici du VHDL) et du code Java. Ce code Java peut être de deux natures différentes :

- <% SCRIPT %> : le script est directement reproduit dans le code Java de la classe générée. Ce script permet par exemple d'évaluer des boucles dont la taille dépend du modèle d'entrée.
- <%= EXPRESSION %> : l'expression dépend en général du modèle d'entrée. Il est par exemple possible d'écrire le nom du concept *a* (attribut name) par le biais de l'expression <%= A.GETNAME()%> ou d'appeler un template d'un autre concept : <%= TS.GENERATE(A.GETB()) %> appelle le template correspondant au concept lié au concept courant (A) par la référence B.

La génération de code à partir de JET est donc réalisée par l'intermédiaire de JET GENERATOR. L'appel des différents templates est géré par MoCodE (Model to Code Engine), une API développée au sein de l'équipe. MoCodE permet d'optimiser les templates en permettant par exemple l'héritage entre concepts. MoCodE permet également de séparer le code généré dans plusieurs fichiers différents ce qui n'est pas possible avec JET seul.

Dans les sections suivantes, nous présentons des templates qui permettent de transformer un modèle RTL en un code. Nous associons des mots clés VHDL aux concepts des accélérateurs matériels, section 4.2, et des mots clés de contraintes de placement aux concepts de placements sur FPGA, section 4.3.

4.2 Génération de code VHDL

Un des objectifs de notre flot de conception est la génération d'un code VHDL lisible et compréhensible par les utilisateurs pour un éventuel débogage du modèle RTL de l'accélérateur ou même du modèle d'application Array-OL initiale. Ainsi, nous nous imposons des contraintes quant à la forme du code VHDL généré. Ces contraintes sont prises en compte lors de l'écriture des templates JET. Voici les quelques règles d'écriture que nous nous imposons :

- La structure de l'accélérateur décrit par le code VHDL correspond à la structure de l'accélérateur modélisé à partir du métamodèle RTL ;
- Chaque composant de l'accélérateur est décrit dans un fichier, le nom de ce fichier est celui du composant ;
- Le parallélisme de l'accélérateur s'exprime sous une forme factorisée à l'aide de la syntaxe VHDL « generate » ;
- Les tableaux multidimensionnels ne sont pas linéarisés lors de la génération de code.

Dans ce document, nous décrivons le mécanisme des templates par des exemples. Nous commençons par la description du template qui génère le code VHDL du multiplexeur décrit précédemment dans la section 3.2.1.2, puis nous abordons la génération de code pour les différents concepts relatifs à l'exécution matérielle du parallélisme de données présenté précédemment dans la section 3.2.4.

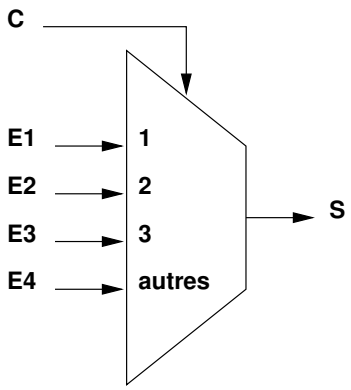
4.2.1 Génération de code des multiplexeurs

Dans la section 3.2.1.2, nous avons présenté la métamodélisation d'un multiplexeur et son schéma électronique. Nous reprenons ici la métamodèlisation de ce multiplexeur et l'associons au template suivant, dont les éléments de la syntaxe VHDL sont en gras :

```
<%=(element.getDataFlowOutput().getName()%> <= <%
for (InputSelect hwis : (List<InputSelect>) element.getSelectInput())
    %><%=(hwis.getDataFlowInput().getName()
    %>  when <%=(element.getControlFlow().getName()%>
  = <%=hwis.getModeCondition().getValue()%> else
  <%=((element.getDefaultDataFlowInput().getName()%>;
```

Ce template permet la génération de code d'un multiplexeur. La première ligne de ce template génère la syntaxe d'affectation de la sortie. La seconde ligne itère sur les concepts INPUTSELECT référencés par le multiplexeur et les lignes 3 et 4 permettent la génération de code de la partie de droite de l'affectation en VHDL conditionnée par la valeur de l'entrée de contrôle. La dernière ligne correspond à l'affectation par défaut de la sortie du multiplexeur.

Nous illustrons le résultat de ce template pour un multiplexeur 4 vers 1 dans la figure 4.4. La gauche de la figure représente le schéma électronique du multiplexeur et la droite le code généré par le template qui nous venons de présenter.




```
S <= E1 when C=1 else
     E2 when C=2 else
     E3 when C=3 else
     E4;
```

FIG. 4.4: La gauche représente un multiplexeur en utilisant les concepts définis dans le métamodèle RTL et la droite le code généré par le template associé au concept MULTIPLEXER.

4.2.2 Génération de code des composants

Les interfaces des composants sont définies par leurs ports d'entrées et sorties. Du point de vue de la génération de code, les templates de génération de code des composants appellent les templates des concepts de ports. Le code suivant présente le template associé au concept INPUTPORT. Ce template génère le nom du port, poursuit par la génération des mots clés VHDL **: IN**, et se termine par l'appel au template permettant de générer le type du port :

```
<%=element.getName()%> : IN <%=ts.generate(element.getType())%>
```

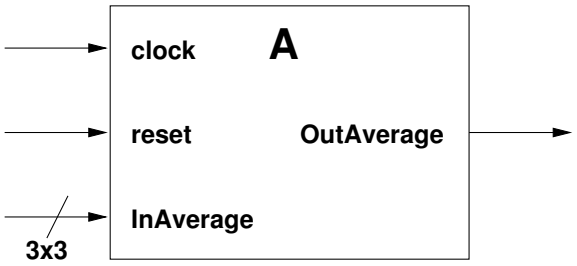
La construction de l'interface d'un composant nécessite l'utilisation des briques de base que sont les templates des ports d'entrée (que nous venons d'illustrer) et de sortie. Quelque soit le type d'un port et la manière dont il est référencé par le composant, le mécanisme de génération de code est similaire et passe par le biais de TS.GENERATE(). TS.GENERATE() appelle le template correspondant au concept de l'élément en paramètre, qui est, dans le cas de la génération de code de l'interface d'un composant, soit un port d'entrée soit un port de

sortie. Le template qui permet de générer le code de l'interface de composant est présenté ci-dessous :

```
ENTITY <%=element.getName()%> IS
PORT
(
<%=ts.generate(element.getClock())%>;
<%=ts.generate(element.getReset())%>
for (Port p : (List<Port>) element.getPorts())
%>;
  <%=ts.generate(p)%><%
%>);
END <%=element.getName()%>;
```

Dans le métamodèle RTL, les ports d'horloge et de remise à zéro sont référencés par *clock* et *reset*, ce qui les rend directement accessibles par un composant. La génération de code de ces ports est donc dissociée de celle des ports référencés dans le métamodèle par *ports*. La boucle for de la ligne 6 permet de retrouver ces ports.

La figure 4.5 illustre la génération de code d'un composant qui possède trois ports d'entrée et un port de sortie. Ce composant est celui utilisé dans l'application du filtre d'images.




```
ENTITY A IS
PORT
(
clock : IN STD_LOGIC;
reset : IN STD_LOGIC;
inAverage : IN TYPE_3_3_Integer_0_to_255;
outAverage : OUT Integer range 0 to 255);
END A;
```

FIG. 4.5: La gauche de la figure représente un modèle de composant dans le métamodèle RTL, la droite de la figure illustre le code généré à l'aide la template associée au concept de composant.

Le template représenté par le code permet de générer l'interface d'un composant dans le métamodèle RTL, appelé entité en VHDL. La génération de code correspondant au comportement du composant dépend du type de ce composant dans le métamodèle RTL. Lorsqu'il s'agit d'une tâche élémentaire, une instanciation de l'IP et un mappage de ses ports sont réalisés. Pour un composant composé et le parallélisme de tâches qu'il exprime, des instances de composants et des connecteurs sont créés, etc. La section suivante présente le template correspondant à la génération de code des boucles dans un composant répétitif.

4.2.3 Génération de codes des boucles dans un composant répétitif

La génération des boucles dans un composant répétitif est associée au concept SHAPE référencé par l'instance de composant dans le composant répétitif. Cette SHAPE correspond à l'espace de répétition de la tâche, elle guide donc l'instanciation du composant répété. Le code suivant représente la partie du template de génération de code du concept REPETITIVE qui permet d'exprimer l'instanciation multiple d'un même composant.

```
<%int indexRepetition = 0;
for (Integer v : (List<Integer>)  ...
     ... element.getRefComponentInstance().getDim().getValue()){
%>genit<%=indexRepetition%> : for ...
            .....it<%=indexRepetition%> in 1 to <%=v%> generate
  <% indexRepetition++;
}%>

<%=ts.generate(element.getRefComponentInstance())%>

<% indexRepetition = 0;
for (Integer v : (List<Integer>) ...
      ... element.getRefComponentInstance().getDim().getValue())
%>end generate;
  <% indexRepetition++;
%>
```

Le code suivant présente le code généré pour la tâche répétée pour l'application du filtre d'images. Les deux dimensions de l'espace de répétitions apparaissent en VHDL sous la forme de deux boucles imbriquées et qui permettent d'instancier les composants de calculs autant de fois que le composant itère sur l'espace. Le mappage des ports, non illustré ici, est dépendant de la ligne 7 du template qui nous venons de présenter.

```
genit0 : for it0 in 1 to 2 generate
genit1 : for it1 in 1 to 2 generate

...

end generate;
end generate;
```

Nous venons d'illustrer différents templates JET associés à des concepts du métamodèle RTL. Ces templates permettent de passer de la forme d'un modèle à une syntaxe VHDL. Nous illustrons dans la section suivante l'utilisation de ces templates pour l'exemple du filtre d'image.

4.2.4 Génération de code VHDL du filtre d'image

Les templates présentés dans la section précédente permettent la génération du code VHDL de tout accélérateur modélisé à l'aide du métamodèle RTL. Nous illustrons ce processus de génération de code par l'exemple académique du filtre d'image étudié dans le chapitre 3. Ce code, présenté en Annexe B est simulable et synthétisable dans tout outil commercial dans la mesure ou nous utilisons la librairie standard [57].

4.2.4.1 Simulation sous ModelSim

Afin de faciliter la vérification fonctionnelle du code VHDL décrivant le filtre d'image, les entrées et sorties de l'accélérateur sont connectées à deux composants dont le rôle et d'alimenter l'accélérateur en images sources est de lire les images produites. De manière générale, un composant connecté à une entrée de l'accélérateur est assimilé à un capteur,

celui connecté à une sortie à un actionneur. Chaque capteur ou actionneur est un composant élémentaire qui référence un IP en bibliothèque, tel que défini dans le métamodèle RTL. Dans le cas de l'exemple du filtre d'image, les images sources qui alimentent l'accélérateur dépendent donc de l'IP référencé par le capteur, les images produites par le filtre sont envoyées à l'IP référencé par l'actionneur.

Pour cette étude de cas, nous définissons une image initiale composée de pixels encodés en niveau de gris et dont la valeur dépend de la position de ce pixel dans l'image : 1 pour le pixel en haut à gauche, 16 pour le pixel en bas à droite, et les valeurs intermédiaires pour les autres pixels. À chaque cycle d'horloge, une nouvelle image dépendante de l'image précédente est créée : chaque pixel est incrémenté de 1, le tout modulo 256 afin de rester conforme à l'encodage en niveau de gris. Voici la succession des images produites par le capteur :

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \Rightarrow \begin{pmatrix} 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 \\ 14 & 15 & 16 & 17 \end{pmatrix} \Rightarrow \begin{pmatrix} 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 \end{pmatrix} \dots$$

Les images générées ne sont pas réalistes car trop petites, mais facilitent néanmoins la compréhension des différents résultats de simulation présentés dans les paragraphes suivants. Les coefficients du filtre d'image sont, quant à eux, directement intégrés dans l'IP réalisant le calcul et sont définis de manière à réaliser une moyenne des pixels. Réaliser la moyenne des pixels permet de valider aisément le résultat produit par l'accélérateur. Toutefois, d'autres types de filtres peuvent être expérimentés, comme la détection de contour. L'image est donc corrélée avec le filtre suivant :

$$\frac{1}{9} \times \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Les capteurs et les actionneurs sont modélisables dans le métamodèle RTL par le biais des tâches élémentaires. Afin d'être utilisés dans l'exemple du filtre d'image, nous créons un composant de plus haut niveau de hiérarchie. Ce composant instancie un composant capteur et un composant acteur en plus de l'accélérateur lui-même. Ce composant de plus haut niveau de hiérarchie possède les entrées d'horloge et de remise à zéro, qu'il diffuse à chaque instance de composant. La remise à zéro positionne l'accélérateur dans un état connu, l'horloge cadence le système. La création de ces stimuli (horloge et remise à zéro) est réalisée par un processus en VHDL qui génère une remise à zéro durant 25ns et qui génère une horloge de période 10ns[1]. Il est par ailleurs possible de générer des scripts qui dépendent des outils de simulation, mais cette solution est moins générique.

La figure 4.6 représente la simulation du filtre d'image en VHDL dans l'environnement ModelSim. Les deux premiers signaux, marqués **1**, sont l'horloge et la remise à zéro. Les signaux marqués **2** correspondent aux sorties du capteur et représentent donc les images sources du filtre d'image. À chaque front montant de l'horloge, une nouvelle image composée de 16 pixels est générée par le capteur. Ces signaux sont transmis à l'accélérateur qui, à l'aide des tilers d'entrée, construit les motifs de chacune des quatre tâches répétées (marque

[1]Il est à noter que ces stimuli ne sont utilisés que pour la simulation du circuit. En effet, ils sont écrits dans le sous-ensemble non synthétisable de VHDL.

3). Chacun des quatre motifs, composé de 9 pixels, alimente une tâche de l'accélérateur et produit un pixel de sortie : ce pixel est obtenu par la somme des produits des pixels du motif de l'image avec les coefficients du filtre. Les quatre pixels produits correspondent aux quatre motifs de sortie de l'accélérateur, marque **4**, qui, à l'aide du tiler de sortie, permettent de reconstituer le tableau de sortie, c'est-à-dire l'image de sortie du filtre d'image, marque **5**.

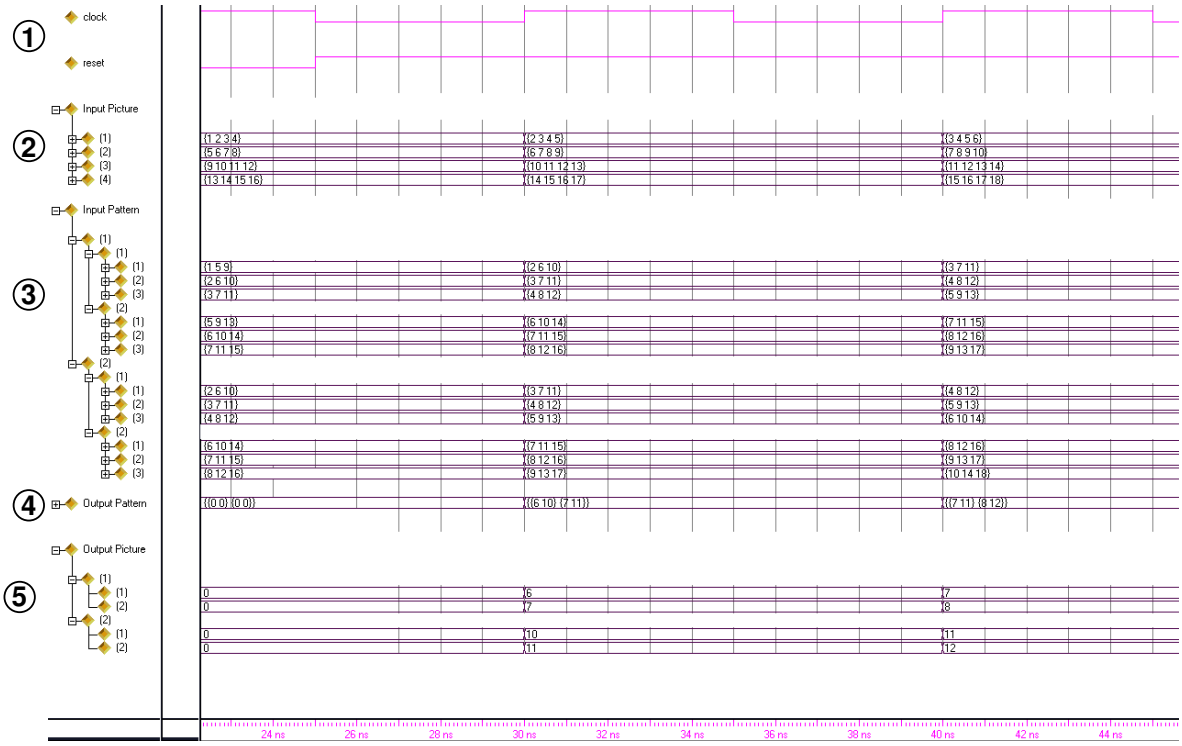


FIG. 4.6: Simulation sous ModelSim du code VHDL décrivant le filtre d'image. Le circuit est initialisé lorsque le signal de remise à zéro raz est actif, et exécute, dans le cas contraire, le filtre d'image au rythme de l'horloge clk.

Le circuit complet est cadencé au rythme de l'horloge, et est initialisé par le signal de remise à zéro. La figure 4.6 illustre le fonctionnement du circuit lors de son initialisation par le signal de remise à zéro et lors de deux fronts montant de l'horloge. Lorsque la remise à zéro est activée, le circuit produit des pixels dont la valeur est égale à zéro, c'est-à-dire une image noire ; lorsque la remise à zéro est désactivée, le circuit réalise la fonction de filtrage des images. Au premier front montant de l'horloge, les quatre tâches élémentaires de l'accélérateur réalisent la moyenne de l'image telle que fournie par le capteur à l'instant qui précède le coup d'horloge, c'est-à-dire l'image initiale générée par le capteur. Au second front montant de l'horloge, le même processus est exécuté, et seule l'image traitée est différente car le capteur l'a modifiée (nous rappelons que dans cette simulation, chaque pixel de l'image initiale est incrémenté par le capteur à chaque front montant de l'horloge). Les valeurs de sorties du filtre permettent de valider que les images produites réalisent bien une moyenne des pixels des images d'entrée ; le circuit est donc fonctionnel. Par ailleurs, une fois le pipeline de tâches rempli (un seul étage de pipeline pour cet exemple), le circuit est en mesure de produire une image par cycle d'horloge.

La simulation du code VHDL permet une vérification fonctionnelle du circuit. Cette vérification dépend principalement des capteurs et des données qu'ils véhiculent dans l'accélérateur. Les capteurs doivent donc être écrits avec un jeu de données suffisant pour permettre une validation de l'accélérateur sur plusieurs cycles d'horloge.

Cependant, cette simulation fonctionnelle ne prend pas en compte l'implémentation physique de l'accélérateur et les délais qu'elle fait intervenir. Une nouvelle simulation basée sur l'implémentation de ce filtre sur FPGA est donc nécessaire ; elle est présentée dans les paragraphes suivants.

4.2.4.2 Synthèse sous Quartus

Dans le cadre de la simulation de l'implémentation de cet accélérateur sur FPGA, le circuit est synthétisé à l'aide d'outils commerciaux, par exemple Quartus pour les FPGAs Altera. La démarche de la synthèse sous Quartus est similaire à celle de la simulation sous ModelSim dans le sens où le composant qui instancie les capteurs, les acteurs et les accélérateurs est défini comme le composant de plus haut niveau de hiérarchie. Nous rappelons que ce composant possède les ports d'horloge et de remise à zéro. Ces ports sont automatiquement connectés par l'outil de synthèse sur les entrées du FPGA dédiées à ces fonctionnalités. Afin d'écarter le risque de suppression partielle ou totale du circuit par le synthétiseur lors des phases d'optimisation, les images produites par le filtre sont connectées sur des pattes du FPGA.

La vue RTL issue de la synthèse sous Quartus du circuit est illustrée par la figure 4.7. Le haut de la figure représente le composant de plus haut de niveau de hiérarchie avec le capteur d'image marqué **1**, le filtre **2**, l'actionneur d'image **3** et la connexion des images produites sur les pattes du FPGA **4** pour faciliter l'accès aux images produites. Le bas de la figure représente l'accélérateur lui-même : il est composé du tiler d'entrée **5** et de sortie **7** et des quatre unités de calcul travaillant chacun sur une partie de l'image, pastilles **6x**. Le bas de la figure correspond au résultat de synthèse sous Quartus du circuit exécutant matériellement une tâche Array-OL ; il est donc très similaire à la figure 3.5 présentée en début de chapitre qui illustre l'exécution parallèle d'une tâche Gaspard. Les instances des tilers sont représentées par les boîtes marquées **5** et **6**. L'espace de répétition 2×2 de la tâche se traduit par l'instanciation de quatre composants (pastilles **6x**).

De plus, l'outil Quartus fournit des informations sur les ressources consommées par le circuit, son chemin critique, sa fréquence de fonctionnement maximale, etc. Pour cette implémentation, la fréquence maximale de fonctionnement du circuit est estimée à 55MHz, que nous ramenons à 50MHz pour la simulation.

De manière plus générale, le succès d'une synthèse de circuit valide le fait que ce circuit est correct du point de vue de sa description et qu'il est implémentable sur FPGA. Une nouvelle simulation de ce circuit implémenté sur FPGA est présentée dans le paragraphe suivant.

4.2.4.3 Simulation sous Quartus

Contrairement à la simulation sous Modelsim, la simulation d'un circuit sous l'environnement Quartus prend en compte l'implémentation de ce circuit sur le FPGA. La réalité physique ainsi prise en compte, les temps de simulation sont largement augmentés et de nouvelles informations sont disponibles, comme les délais de communication.

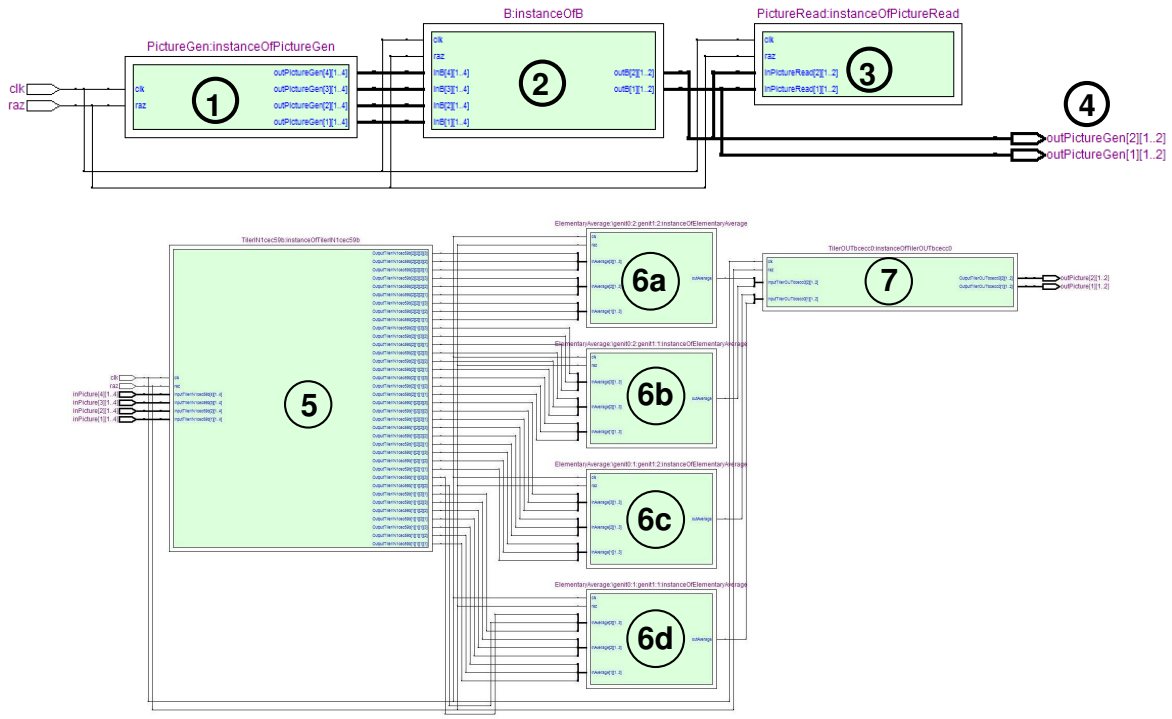


FIG. 4.7: Résultat de synthèse sous Quartus. Le haut de la figure représente le capteur, le filtre lui-même et l'actionneur. La sortie, sur la droite, facilite l'accès aux images produites. Le bas de la figure représente le filtre : il est composé du tiler d'entrée, des quatre instances de composants travaillant chacune sur une partie de l'image, et le tiler de sortie.

La figure 4.8 reprend la même simulation que celle présentée à la figure 4.6, à l'exception du fait que seul le résultat de l'application est présenté. Ce résultat, les images produites, est visible sur les ports de sortie que nous avons ajoutés, comme illustré par la pastille **4** de la figure 4.7. La fréquence d'horloge, initialement de 100MHz pour une simulation sous ModelSim, est ajustée sous Quartus aux résultats de synthèse et est donc fixée à 50MHz. Les images sont produites à chaque cycle d'horloge et chaque front montant de l'horloge correspond à une image valide. Les délais liés à l'implémentation de l'accélérateur sur le FPGA apparaissent, contrairement à la simulation sous Modelsim. Sur la figure 4.8, seuls les délais liés à la création de l'image de sortie sont illustrés, mais des délais internes aux circuits existent par ailleurs. Par exemple, les tâches élémentaires du filtre d'images sont composées d'un étage de pipeline, ce qui permet de découper le chemin critique du circuit. Il est à noter que les simulations sous ModelSim et sous Quartus produisent les mêmes résultats.

La génération de code présentée dans cette section permet de passer d'un accélérateur matériel décrit sous la forme d'un modèle à un code VHDL simulable et synthétisable dans différents outils. Dans cette section, nous avons illustré différentes utilisations possibles du code VHDL généré.

Lors du placement sur FPGA d'un accélérateur décrit en VHDL, section 3.3.2, les informations de placement ne sont pas définies par le code VHDL, mais à l'aide d'un fichier

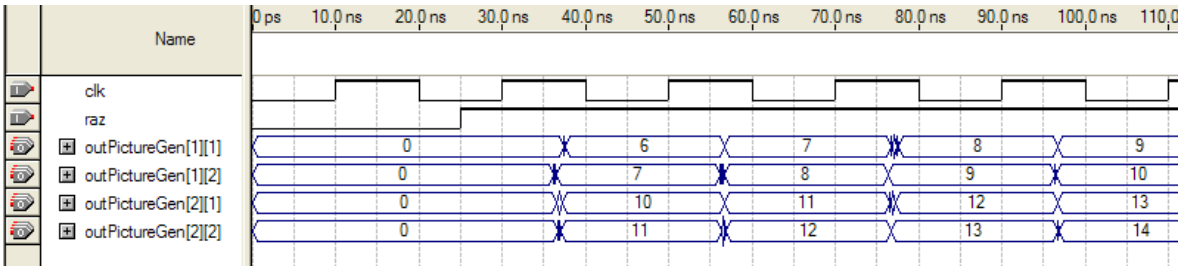


FIG. 4.8: Simulation sous Quartus de l'implémentation du filtre d'image sur le FPGA Stratix2s60.

utilisateur. La section suivante présente la génération de code pour de tels placements.

4.3 Génération de code des fichiers de contraintes de placement

Les outils de synthèse de type Quartus de Altera ou ISE de Xilinx peuvent prendre en compte des informations de placement de circuits sur FPGA. Ces contraintes peuvent être créées par le biais de l'outil en question (cette solution nécessite une première phase de compilation qui permet à l'outil de construire la structure hiérarchique du circuit) ou sous la forme d'un fichier utilisateur.

Dans la section 3.3.2, les accélérateurs sont placés sur des FPGAs au moyen du concept INSTANCEFPGAPLACEMENT. Ce concept est utilisé pour générer un fichier utilisateur dans lequel les placements modélisés dans le métamodèle RTL sont retranscrits. Le placement est alors pris en compte par l'outil lors de la synthèse d'un accélérateur.

La démarche de génération de code pour les fichiers utilisateur est la même que celle utilisée pour la génération de code VHDL présentée dans la section précédente. Cependant, et contrairement à la génération de code VHDL, la génération de code pour le placement est dépendante de l'outil : nous utilisons Quartus et générons donc un fichier de contrainte de placement pour cet outil. Il est toutefois possible de générer des contraintes de placement pour d'autres outils dans la mesure ou la sémantique des concepts relatifs à INSTANCEFPGAPLACEMENT est similaire dans ces différents outils de synthèse.

4.3.1 Génération de code d'un fichier de contraintes de placement

INSTANCEFPGAPLACEMENT est le concept utilisé pour générer le code d'un placement. Ce template est différent de ceux présentés précédemment car il exploite les informations liées au placement d'un accélérateur sur un FPGA.

La génération de code d'un placement nécessite les informations suivantes : le nom de l'instance du composant placé, le chemin de cette instance dans la hiérarchie de l'accélérateur, le nom de la zone, la position de son origine dans la grille du FPGA, sa hauteur et sa largeur. Le point clé de cette génération de code est la création du lien entre l'instance de composant et sa zone de placement.

Chacune de ces informations est individuellement et indépendamment générée ; le squelette de la génération de code est donc relativement simple. Le code suivant correspond à ce template :

```
set_global_assignment -name LL_ORIGIN LAB_X ...
  ...      <%=element.getOrigin().getValue().get(0)%> ...
  ...      _Y<%=element.getOrigin().getValue().get(1)%> ...
  ...      -section_id <%=element.getName()%>
```

Nous illustrons dans le paragraphe suivant l'utilisation de ce template pour un placement du filtre d'images sur le FPGA Stratix2S60.

4.3.2 Génération d'un placement sur un FPGA Stratix2S60

Afin d'illustrer la génération de code pour les contraintes de placement, nous reprenons l'exemple du filtre. Le modèle de cet accélérateur est celui utilisé dans la section 4.2.4. Nous enrichissons ce modèle par l'utilisation d'un FPGA dans une bibliothèque de FPGA et par la création d'un placement entre l'accélérateur et ce FPGA.

Nous implémentons l'accélérateur sur un FPGA Stratix2s60, et reprenons donc la modélisation de ce FPGA introduite précédemment dans le chapitre 3. En fonction des informations issues de la synthèse, les ressources du FPGA consommées par chaque composant de l'accélérateur sont connues et sont la base de la construction des concepts INSTANCEFPGAPLACEMENT. Ces vues modélisent le placement de l'accélérateur sur le FPGA et prennent notamment en compte la régularité de l'accélérateur et du FPGA. Nous reprenons la modélisation du placement illustrée précédemment en bas de la figure 3.20 car l'arbre représenté sur la partie gauche de cette figure correspond au modèle de l'accélérateur du filtre d'image. Ce placement est toutefois modifié afin de prendre en compte les ressources du FPGA nécessaires à l'implémentation de chaque composant de l'accélérateur.

Par exemple, chaque unité de calcul répétée dans le parallélisme de données consomme 14 ressources du FPGA d'après les résultats de synthèse sous Quartus. Par ailleurs, ce placement est enrichi et prend en compte le capteur et l'acteur de l'application. Le capteur nécessite 40 ressources de calcul (toujours d'après les résultats de synthèse sous Quartus), l'acteur une ressource de calcul.

Au final, et du point de vue de la modélisation, nous obtenons un modèle contenant l'accélérateur pour le filtre d'image ave ses capteurs et acteurs, le FPGA Stratix2s60 et le placement de l'application sur ce FPGA. À partir de ce modèle, nous générons le code VHDL de l'accélérateur (il est identique à celui présenté dans la section 4.2.4) et le fichier de contraintes représenté en partie par le code suivant.

```
set_global_assignment -name LL_ORIGIN LAB_X32_Y5 -section_id Acc1
set_global_assignment -name LL_HEIGHT 2 -section_id Acc1
set_global_assignment -name LL_WIDTH 7 -section_id Acc1

set_instance_assignment -name LL_MEMBER_OF Acc1 -to ...
    "B:instanceOfB|A:\\genit0:1:genit1:1:instanceOfA" -section_id Acc1
```

Les trois premières lignes de ce code définissent l'origine, la hauteur et la largeur de la zone de placement nommée *Acc1* : son origine dans la grille du FPGA est placée à la position 32 sur l'axe des x et 5 sur celui des y, sa hauteur est 7 et sa largeur 2. La zone nommée "Acc1" permet le placement d'une unité de calcul dans l'espace de répétition de la tâche. L'attribution de la zone à cette unité de calcul est réalisée à la ligne 5 : le chemin de l'instance de composant dans l'accélérateur détermine l'unité de calcul concernée. Certaines libertés peuvent être données à l'outil. Ces libertés sont une flexibilité dans la dimension des zones de placement et une flexibilité dans le positionnement de cette zone. La flexibilité

dans la zone de placement permet à l'outil d'étendre une zone s'il lui est impossible de placer et router le circuit qui lui est attribué. La flexibilité dans le placement permet à l'outil de réorganiser le placement en fonction notamment des dépendances de données et des ressources requises par la partie de circuit placée sur cette zone. Ces libertés sont prises en compte dans la métamodélisation des placements et l'utilisation ou non d'une origine et d'une dimension pour chaque zone de placement.

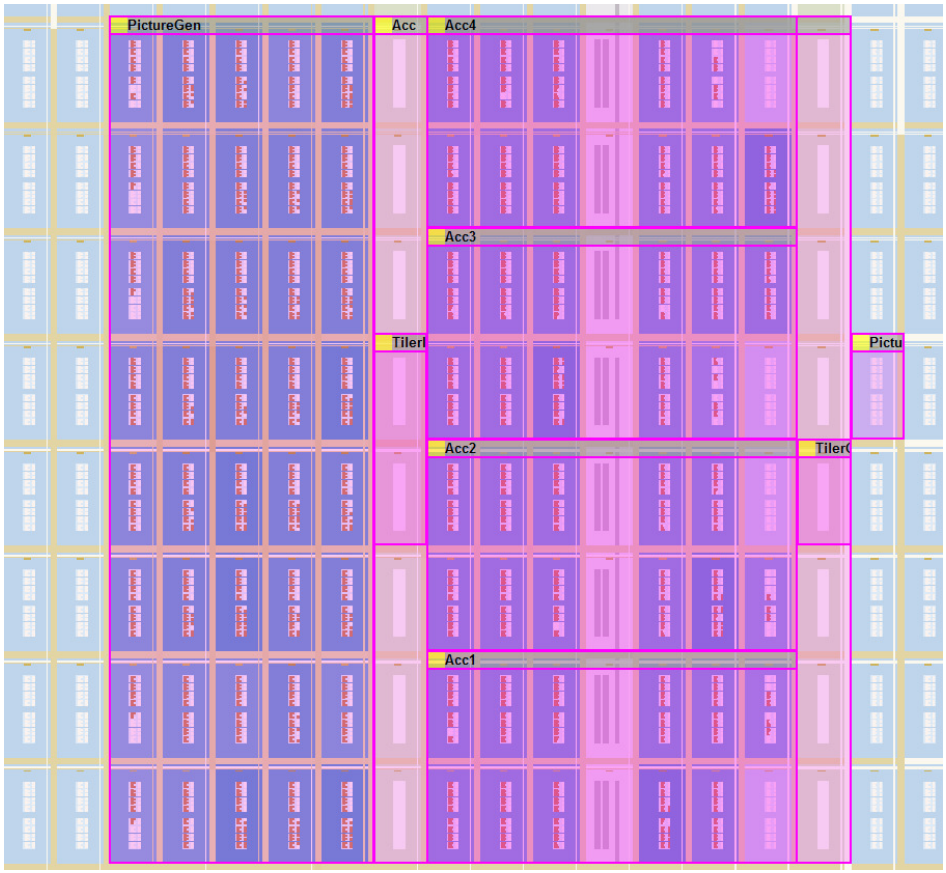


FIG. 4.9: Résulat de synthèse sous Quartus du filtre d'image placé sur un Stratix2s60.

Comme précisé précédemment, ces contraintes sont insérées dans un fichier du projet Quartus qui contient certains paramètres relatifs à la configuration de l'outil. Ces contraintes sont donc prises en compte lors de la synthèse de l'accélérateur. Un résultat de cette synthèse est illustré à la figure 4.9, elle représente une vue du FPGA et du circuit qu'il implémente dans l'outil Quartus. Cette vue est focalisée sur la partie du FPGA sur laquelle nous avons placé le circuit. La cellule en bas à gauche à pour coordonnées 24,5, celle en haut à droite 41,12. Les zones de configuration apparaissent sous forme de rectangles, leur nom est indiqué dans la partie grisée en haut de chaque zone. La zone *Acc1*, dont le code généré a été présenté dans cette section, est située en bas au milieu de la figure. Elle recouvre une zone de 7 cellules du FPGA sur 2 en hauteur. De la même façon, les zones *Acc2*, *Acc3* et *Acc4* sont placées régulièrement et de bas en haut. Le parallélisme de l'accélérateur, c'est-à-dire c'est quatre itérations de tâche, est placé de manière régulière et profite donc du parallélisme du

FPGA. Ces quatre zones sont alimentées en données par le tiler d'entrée, qui est collé à la gauche de l'accélérateur. Ce dernier alimente le tiler de sortie, situé à droite des zones dédiées aux unités de calcul. Chacun de ces éléments fait parti d'une zone, peu visible sur la figure, qui représente le placement de l'accélérateur. Cet accélérateur est connecté au capteur et à l'acteur. Le capteur est placé sur la zone nommée *PictureGen*, l'acteur sur la zone *PictureRead*. De nouveau, les éléments de l'application du filtre d'image sont disposés de manière à faciliter le routage de signaux entre les éléments de l'application. Chaque cellule du FPGA est composée de plusieurs ressources similaires à des LUTs. Lorsqu'elles sont utilisées, ces LUTs sont « foncées ». Il est ainsi possible de distinguer, au sein de chaque zone, la quantité de ressources nécessaires à l'implémentation du circuit.

4.4 Conclusion

Ce chapitre présente la transformation RTL2VHDL qui permet la génération d'un code à partir d'un modèle conforme au métamodèle RTL. Nous avons montré que l'expression de l'espace de répétition d'une tâche en VHDL est conservée sous une forme compacte et que la structure de l'accélérateur est conservée. Cela permet à l'utilisateur de notre flot de conception de mieux appréhender le circuit généré lors de sa visualisation dans l'outil de synthèse notamment.

Concernant la description des accélérateurs matériels, la génération de code n'introduit aucune limitation dans le flot de conception : tout ce qui est modélisable dans le métamodèle RTL peut être retranscrit en VHDL. Cela est aussi valable pour les tableaux de données multidimensionnels, tout modèle conforme au métamodèle RTL peut donc être retranscrit en VHDL afin d'être vérifié par le biais de simulations ou de synthèses.

Nous avons par ailleurs démontré le bon fonctionnement de la génération de code pour les scripts de placements. Cette génération de code n'est toutefois valable que pour l'outil Quartus car il n'existe pas de standard pour leur représentation. L'utilisation de ces placements lors de la synthèse montre que la description du placement dans un modèle RTL est conforme à l'interprétation qu'en fait l'outil.

Chapitre 5

Transformation de modèles vers le métamodèle RTL

Contents

Ce chapitre présente la compilation d'un modèle Deployed en un modèle RTL qui décrit un accélérateur permettant l'exécution matérielle d'une application. Cette compilation est réalisée par la transformation de modèles *Deployed2RTL* identifiée dans l'environnement Gaspard par le cercle sur la figure 5.1.

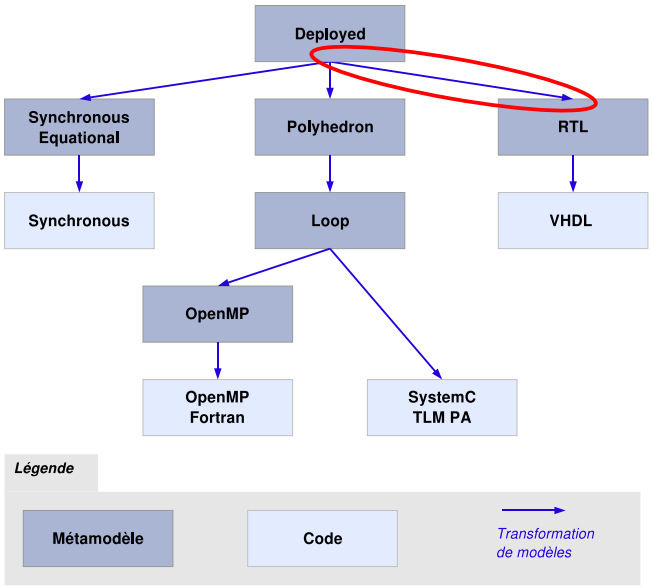


FIG. 5.1: L'environnement Gaspard et la compilation d'un modèle Deployed en un modèle RTL.

Un modèle conforme au métamodèle Deployed est indépendant de toute implémentation : il est modélisé à un haut niveau d'abstraction. La transformation de modèles Deployed2RTL ajoute ces détails d'implémentation qui permettent l'exécution matérielle de l'application décrite par le modèle Deployed. Deployed2RTL est décomposée en règles de transformations que nous présentons sous une forme graphique et que nous implémentons sous une forme textuelle pour les exécuter. L'exécution de ces règles permet de générer un accélérateur matériel à partir d'un modèle Deployed qui est implémentable sur FPGA grâce à la génération de code VHDL présentée dans le chapitre précédent.

La section 5.1 de ce chapitre présente brièvement le métamodèle Deployed et les concepts nécessaires à la compréhension de nos travaux. La section 5.2 détaille les règles de transformation qui permettent de compiler le parallélisme de données du métamodèle Deployed en parallélisme de données dans le métamodèle RTL.

5.1 Métamodèle source : Deployed

Dans cette section, nous décrivons le métamodèle source de la transformation permettant de générer un modèle RTL. Ce métamodèle correspond au métamodèle de saisie de l'environnement Gaspard qui permet notamment de modéliser les applications Gaspard.

5.1.1 Vue d'ensemble du métamodèle Deployed

Le métamodèle Deployed a servi de base à la définition d'une partie du métamodèle MARTE. Cependant, la version du standard de l'OMG et celle du métamodèle Deployed ne sont pas identiques même si elles sont très proches. Le métamodèle Deployed a pour vocation à devenir, dans son intégralité, un sous-ensemble de MARTE et évolue dans ce sens depuis que le standard a été proposé. On notera néanmoins que les travaux réalisés dans cette thèse ont été effectués avec une version du métamodèle Deployed plus ancienne et donc plus éloignée du métamodèle MARTE.

Dans le cadre de la transformation Deployed2RTL, nous nous intéressons à la compilation des applications Gaspard déployées, nous ne décrivons donc que brièvement le sous-ensemble concerné.

5.1.2 Composants applicatifs

Le métamodèle Deployed permet la modélisation de différents composants applicatifs en fonction de leur structure :

- ELEMENTARY : un composant possédant une structure élémentaire correspond à un composant atomique dans l'environnement Gaspard et dont la composition est inconnue. Les composants élémentaires sont déployés sur des IPs qui réalisent des fonctionnalités spécifiques pour l'application ou qui instancient des composants matériels pour l'architecture. Dans le cadre de la transformation Deployed2RTL, le déploiement est pris en compte afin de récupérer les IPs VHDL auxquels font référence les tâches élémentaires. La partie du métamodèle Deployed spécifique au déploiement n'est toutefois pas détaillée dans ce document, le lecteur intéressé peut trouver des informations complémentaires dans la thèse dÉric Piel [96] ;
- COMPOUND : un composant hiérarchique dans lequel est décrit le parallélisme de tâches. Un composant composé peut instancier d'autres composants composés, il n'y a donc pas de restriction sur la hiérarchie des applications ;
- REPETITIVE : ce composant permet la description du parallélisme de données des applications Gaspard.

La figure 5.2 illustre les différents composants dans le métamodèle Deployed. On y retrouve le concept abstrait COMPONENT qui peut être spécialisé en composant applicatif, concept APPLICATION. Un composant contient une structure interne, INTERNALSTRUCTURE, qui spécifie que ce composant est élémentaire, composé ou répétitif. Une application répétée est donc modélisée par un composant APPLICATION avec une structure interne REPETITIVE.

Cette figure représente par ailleurs les concepts de ports, d'instances de composant, d'instance de ports, et de connecteur. Nous détaillons ces différents points ci-dessous.

Les ports représentent le moyen de communiquer avec un composant, ils contiennent des informations sur les données qu'ils véhiculent, comme leur type et leur organisation. Un port est abstrait et se concrétise en port d'entrée, INPUTPORT, ou en port de sortie, OUTPUTPORT.

Les composants composés ou répétitifs autorisent l'instanciation de composants, concept COMPONENTINSTANCE. Lorsque cette instanciation est réalisée dans un composant répétitif, une SHAPE est associée à l'instance de composant. Cela correspond à l'espace de répétition de la tâche représentée par l'instance de composant. Il est possible de communiquer avec les instances de composant par l'intermédiaire des instances de port, concept PORTINSTANCES.

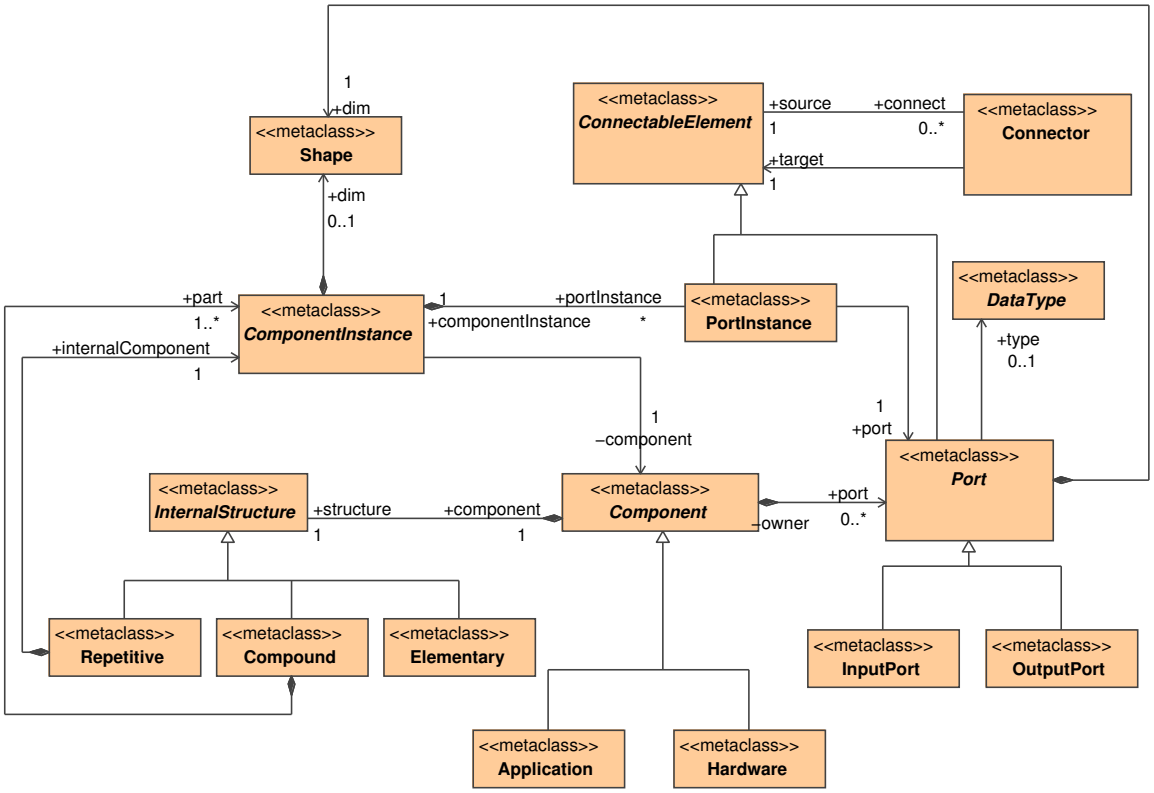


FIG. 5.2: Les composants applicatifs dans le métamodèle Deployed.

Le concept CONNECTOR permet d'établir des connexions entre des éléments connectables, qui sont soit des ports, soit des instances de ports.

5.1.3 Expression des dépendances de données par les tilers

Dans les applications Gaspard, les dépendances de données du parallélisme de données sont spécifiées par des tilers. Le concept TILER étend CONNECTOR est contient TILINGDESCRIPTION. Cette dernière permet la description classique des origines, pavage et ajustage en Array-OL à l'aide des conceptsINTVECTOR et MATRIX, comme illustré à la figure 5.3. Le pavage et l'ajustage sont exprimés par une matrice qui contient potentiellement plusieurs vecteurs, l'origine est définie par un vecteur. Les dépendances de données sont donc spécifiées sous une forme factorisée.

5.1.4 Bilan du métamodèle Deployed

Le métamodèle permet la modélisation des applications Gaspard au travers de composants applicatifs dont la structure les spécialise en composant élémentaire, composé ou répétitif. Les dépendances de données du parallélisme de données sont exprimées avec ARRAY-OL, c'est-à-dire sous une forme « origine, pavage et ajustage ».

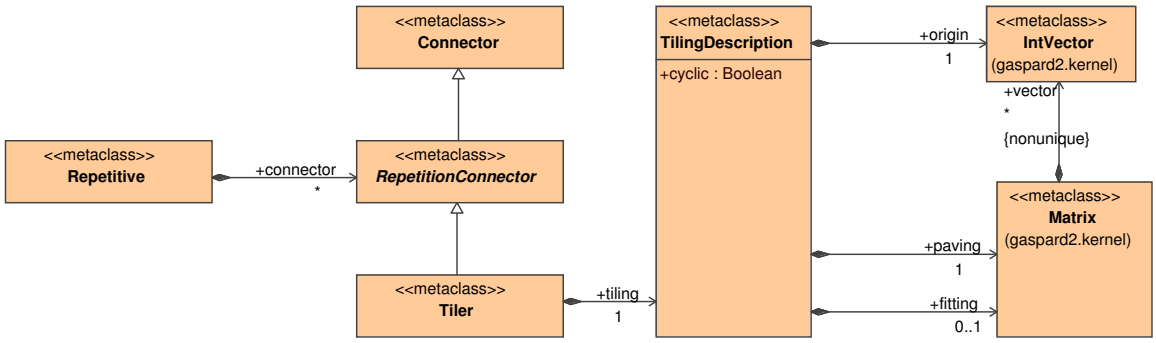


FIG. 5.3: Le concept tiler dans le métamodèle Deployed : un tiler étend le concept de connecteur et est nécessairement contenu dans un composant répétitif.

Les modèles d'application du métamodèle Deployed sont indépendants des différentes cibles d'exécutions de l'environnement Gaspard. C'est la compilation vers ces cibles d'exécutions qui spécialise ces modélisations. Nous présentons dans la section suivante la compilation des modèles d'applications Gaspard vers un modèle RTL d'accélérateur via une transformation de modèles.

5.2 Transformation d'un modèle Deployed vers un modèle RTL

Comme nous l'avons vu dans le chapitre introductif, une transformation de modèles se décompose en règles de transformation. Une règle correspond à la transformation d'un ensemble de concepts du métamodèle source en un ensemble de concepts du métamodèle cible. Il est en général possible d'exprimer une même transformation à l'aide de différentes compositions de règles, qui est un point clé du développement d'une transformation de modèles en général. Plus les règles sont compliquées et font intervenir un grand nombre de concepts, plus elles sont difficiles à documenter et à mettre au point. En revanche, la lisibilité et la compréhension d'une transformation de modèles dans sa globalité reposent sur le nombre de règles qu'elle contient. Ce paradoxe lié au nombre de règles et à leur complexité nous amène à représenter graphiquement nos règles de transformation à l'aide de TrML (Transformation Modeling Language) qui est un outil de représentation de règles développé au sein de notre équipe [43]. TrML facilite la structuration des transformations de modèles car il permet une visualisation directe de la complexité de chaque règle utilisée dans la transformation de modèles. Par ailleurs, les dépendances entre les différents concepts manipulés dans une règle apparaissent clairement sur sa représentation graphique TrML.

Le métamodèle Deployed est le métamodèle source de notre transformation de modèle, tel que défini dans notre flot de conception. Les concepts de ce métamodèle utiles à l'exécution matérielle des applications Gaspard sont transformés en concepts du métamodèle RTL. Dans ce chapitre, nous nous intéressons plus particulièrement à la transformation du parallélisme de données. Nous savons que le métamodèle Deployed est indépendant de toute exécution car il permet la modélisation des applications Gaspard qui peuvent ensuite être exécutées sur des processeurs ou sur des accélérateurs. L'un des objectifs de notre transformation de modèle est donc de spécialiser cette modélisation du parallélisme de données

indépendante de son exécution en modélisation dépendante du modèle d'exécution matérielle que nous avons défini dans le chapitre précédent.

Nous introduisons dans un premier temps TrML, section 5.2.1, puis nous représentons les règles de transformations permettant de compiler le parallélisme de données du métamodèle Deployed vers le métamodèle RTL, sections 5.2.2 à 5.2.7.

5.2.1 TrML, outil de représentation graphique des règles de transformation

Depuis plusieurs années, l'activité de modélisation est principalement réalisée avec des modeleurs UML car la notation graphique des modèles est devenue familière. Une représentation graphique des règles de transformation faisant abstraction de ses détails d'implémentation semble correspondre au meilleur niveau de description d'une règle. Les règles décrites sont alors indépendantes du moteur de transformation. TrML [43] propose d'unifier et de standardiser la représentation graphique de règles en s'appuyant sur les recommandations de l'IDM qui préconise la représentation graphique des règles indépendamment de leur langage d'implémentation. Par ailleurs, les travaux menés durant cette thèse ont contribué à l'évolution de TrML. En particulier, l'implémentation en TrML de la transformation de modèles Deployed vers RTL a permis de préciser et valider certains besoins pour la représentation graphique des transformations.

TrML se décline sous la forme d'un profil UML et d'un métamodèle. Le profil répond aux attentes des utilisateurs car il respecte leurs habitudes de modélisation. Ce profil permet l'utilisation de TrML avec tout modeleur UML (MagicDraw, Objecteering, Papyrus [1], etc.). Chaque utilisateur de TrML peut donc représenter ses règles et les communiquer sans avoir besoin d'apprendre un nouveau langage. Le métamodèle permet la portabilité de TrML sur des moteurs de transformation existants, sous réserve qu'ils soient eux aussi décrits sous la forme d'un métamodèle. La transformation d'un modèle de règle décrit en TrML vers un modèle de règle dans un moteur de transformation existant permet l'exécution de la règle.

Dans cette thèse, nous utilisons TrML pour documenter les règles nécessaires à la transformation d'un modèle Deployed en un modèle RTL : le modèle source de la transformation est un modèle Deployed, le modèle de destination un modèle d'accélérateur qui permet l'exécution matérielle de l'application. Cette dépendance entre les modèles source et destination est exprimée dans une première règle appelée DEPLOYED2RTL. Cette règle appelle d'autres règles qui transforment les différents éléments ou ensembles d'éléments du modèle d'application en leurs équivalents dans le modèle RTL. Il existe donc des règles pour transformer les composants, les ports, les tilers, etc. Nous décrivons dans ce document certaines de ces règles.

Nous expliquons le fonctionnement de TrML à l'aide de la règle COMPONENT2COMPONENT qui transforme un composant dans le métamodèle Deployed en un composant dans le métamodèle RTL. Pour le lecteur intéressé, Etien *et.al.* [43] fournissent une description plus formelle de TrML. Nous détaillons cette règle COMPONENT2COMPONENT dans la section suivante.

5.2.2 Règle COMPONENT2COMPONENT

Nous débutons l'illustration du fonctionnement de TrML avec la description de la règle de transformation COMPONENT2COMPONENT. Nous décrivons dans un premier temps l'ob-

[1] http://www.papyrusuml.org/

jectif de la règle et détaillons par la suite son implémentation en TrML.

5.2.2.1 Objectif de la règle

COMPONENT2COMPONENT transforme un composant abstrait du métamodèle Gaspard en un composant abstrait du métamodèle RTL. Les ports d'horloge CLOCK et de remise à zéro RESET sont créés et ajoutés au composant du modèle RTL. Cette règle est abstraite, c'est-à-dire qu'elle doit être spécialisée par une autre règle de transformation afin d'être exécutée sur un modèle : COMPONENT2COMPONENT permet de factoriser la création du port d'horloge et du port de remise à zéro des composants concrets du métamodèle RTL.

5.2.2.2 Détails de la règle

La représentation TrML de la règle COMPONENT2COMPONENT est illustrée à la figure 5.4. Une règle TrML se décompose en trois parties que sont le patron d'entrée (qui conditionne l'exécution d'une règle), la règle elle-même et le patron de sortie (qui correspond au résultat de l'exécution de la règle) :

Le patron d'entrée Le patron d'entrée d'une règle est représenté sur la partie gauche des différentes figures présentées dans ce document[2]. Il représente l'ensemble des éléments nécessaires à l'exécution d'une règle ainsi que les relations entre ces éléments. Le patron est un arbre dont l'élément racine est stéréotypé ≪Pattern≫.

L'élément racine du patron d'entrée de la règle COMPONENT2COMPONENT est *g_c*, qui correspond à un concept de composant applicatif dans le métamodèle Gaspard. *g_c* est illustré sur la partie gauche de la figure 5.4. La variable *name_c* permet le stockage du champ qui correspond au nom du composant *g_c*. En TrML, les variables permettent de manipuler des informations relatives à des éléments du patron d'entrée.

La règle La règle est matérialisée par une classe à laquelle sont associés les patrons d'entrée et de sortie. Une règle ne définit pas comment transformer les concepts, mais spécifie les propriétés qui doivent être transformées et où stocker les résultats.

Il existe différents types de règles en TrML. La règle de base est **Rule**, elle peut être appelée depuis une autre règle et peut en appeler d'autres. Cette règle peut être spécialisée en **TopRule**. Les règles **TopRules** sont automatiquement exécutées par le moteur de transformation lorsque la condition d'exécution de cette règle est remplie. Pour cela, le moteur de transformation parcourt le modèle d'entrée à la recherche des patrons correspondants. Trois autres spécialisations de **Rule** permettent d'exprimer différents appels multiples de règles :

- **SelectRule** : les sous-règles sont ordonnées et mutuellement exclusives. Seule la première sous-règle satisfaite est exécutée.
- **SetRule** : toutes les sous-règles sont exécutées ;
- **ListRule** : toutes les sous-règles sont exécutées dans l'ordre spécifié par la définition de **ListRule**.

[2]TrML permet la description de règles réversibles. Aucune information concernant l'orientation d'une transformation n'est précisée. Cependant, dans le cas de notre flot de conception, nous nous intéressons uniquement à la transformation d'un modèle Deployed en un modèle RTL et non l'inverse. La représentation à gauche des concepts du métamodèle Deployed est donc arbitraire mais permet de conserver le sens de lecture habituel pour chacune des règles présentées.

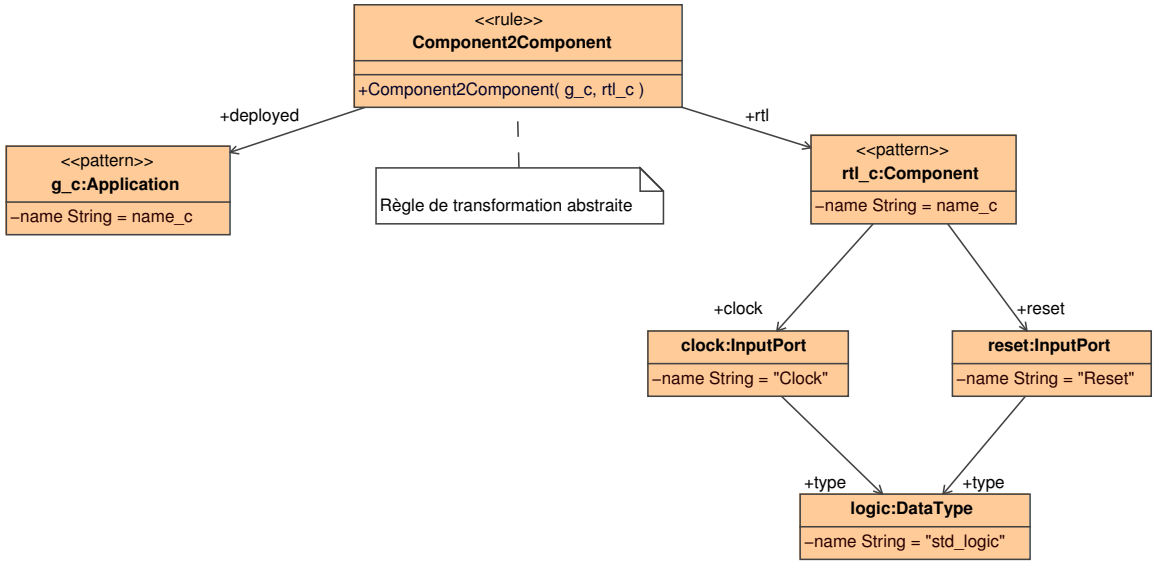


FIG. 5.4: Représentation TrML de la règle COMPONENT2COMPONENT. En TrML, une règle se lie de la gauche vers la droite : le patron d'entrée, la règle et le patron de sortie.

A chacun de ces 5 types de règles est associé un stéréotype : ≪Rule≫, ≪TopRules≫, ≪SelectRule≫,≪SetRule≫ et ≪ListRule≫.

Lorsque la règle COMPONENT2COMPONENT est exécutée, elle sauvegarde les variables utilisées dans le patron d'entrée de la règle (seule la variable *name_c* est utilisée dans cette règle) et les réutilise lors de la création du patron de sortie.

Le patron de sortie En TrML, le patron de sortie d'une règle de transformation est représenté sur la partie droite des figures. Un patron de sortie correspond au résultat d'exécution d'une règle et inclut les concepts créés et les relations entre ces concepts. La paramétrisation des concepts et de leur relation est assurée par la règle elle-même : les valeurs des variables issues du patron d'entrée de la règle peuvent être utilisées lors de la création du patron de sortie.

La racine du patron de sortie de la règle COMPONENT2COMPONENT correspond à *rtl_c* qui est un concept abstrait de composant (le *c* de *rtl_c*) dans le métamodèle RTL (le *rtl* de *rtl_c*), c'est-à-dire COMPONENT. Le nom de *rtl_c* est donné par la variable *name_c* et correspond donc au nom de *g_s* dans le patron d'entrée de la règle. Par ailleurs, COMPONENT2COMPONENT crée les ports d'horloge et de remise à zéro (tout deux des concepts INPUTPORT du métamodèle RTL) qu'elle associe à *rtl_c* par le biais des références CLOCK et RESET. *clock* et *reset* sont typés de manière à représenter un signal numérique sur des en matériel, ce qui correspond à la syntaxe « STD_LOGIC » en VHDL ou « WIRE » en Verilog.

Dans les sections suivantes, nous représentons graphiquement trois règles de transformation qui constituent un point clé de notre compilation vers le métamodèle RTL : la compilation du parallélisme de données. La première règle transforme un composant de boucle dans le métamodèle Deployed en un composant répétitif dans le métamodèle RTL. Cette règle correspond à la première étape de la compilation du parallélisme de données. Les

autres étapes sont réalisées via des compilations de tilers. Dans le métamodèle Deployed, les tilers existent sous la forme de connecteurs, alors que ce sont des composants dans le métamodèle RTL. Deux règles existent pour la compilation de ces tilers : la première crée le composant et la seconde instancie ce composant.

5.2.3 Règle REPETITIVE2REPETITIVE

La règleREPETITIVE2REPETITIVE transforme le concept de composant qui exprime le parallélisme de données dans le métamodèle Deployed en concept équivalent dans le métamodèle RTL.

5.2.3.1 Objectif de la règle

Dans le métamodèle Deployed, la nature d'un composant est définie par sa structure, qui peut être composée, répétitive ou élémentaire. Dans le métamodèle RTL, des composants sont directement associés aux différents composants du modèle d'exécution et sans avoir à utiliser de structure dans les composants. Cela est lié au fait que nous ne nous intéressons pas à la transformation de l'architecture dans le métamodèle Deployed, par conséquent, moins de concepts composants sont manipulés dans le métamodèle RTL par rapport au métamodèle Deployed. L'objectif de la règle REPETITIVE2REPETITIVE est de créer un composant REPETITIVE dans le métamodèle RTL à partir d'un composant COMPONENT dans le métamodèle Deployed dont la composition indique qu'il exprime du parallélisme de données.

Le modèle d'exécution matérielle des applications Gaspard prévoit deux exécutions différentes du parallélisme de données : séquentielle et parallèle. La règle REPETITIVE2REPETITIVE s'intéresse à la génération d'un modèle qui permet l'exécution parallèle de ce parallélisme de données.

5.2.3.2 Détails de la règle

Le parallélisme de données d'une application Gaspard est exprimé dans le métamodèle Deployed avec les composants COMPONENT dont la structure interne référence un concept REPETITIVE. La création des composants REPETITIVE dans le métamodèle RTL dépend donc de la présence d'une structure qui référence un REPETITIVE dans un composant du métamodèle Deployed. Cette condition d'exécution correspond au patron d'entrée de la règle REPETITIVE2REPETITIVE illustrée sur la partie gauche de la figure 5.5. Cette règle spécialise la règle COMPONENT2COMPONENT définie dans la section précédente. En TrML, cette spécialisation est représentée par un héritage d'une règle sur une autre règle, comme illustré par le haut de la figure. Cette héritage signifie que *rtl_rep* possède des ports d'horloge et de remise à zéro et que son nom correspond au nom du composant *g_c*.

Lorsque les conditions d'exécution de la règle sont remplies, la racine du patron de sortie est créée. Cette racine correspond au composant REPETITIVE nommé *rtl_rep*. Ce composant référence une instance de composant COMPONENTINSTANCE nommée *rtl_ci* et dont l'espace de répétition *rtl_rep_space* correspond à la transformation dans le métamodèle RTL de l'espace de répétition *rep_space* du patron d'entrée. Cette information est stockée dans la variable *S*.

La règle REPETITIVE2REPETITIVE appelle des règles qui transforment les ports, l'instance de composant et les tilers. Les règles INPUTPORT2INPUTPORT et OUTPUTPORT2OUTPUTPORT permettent de transformer respectivement les ports d'entrée et sortie.

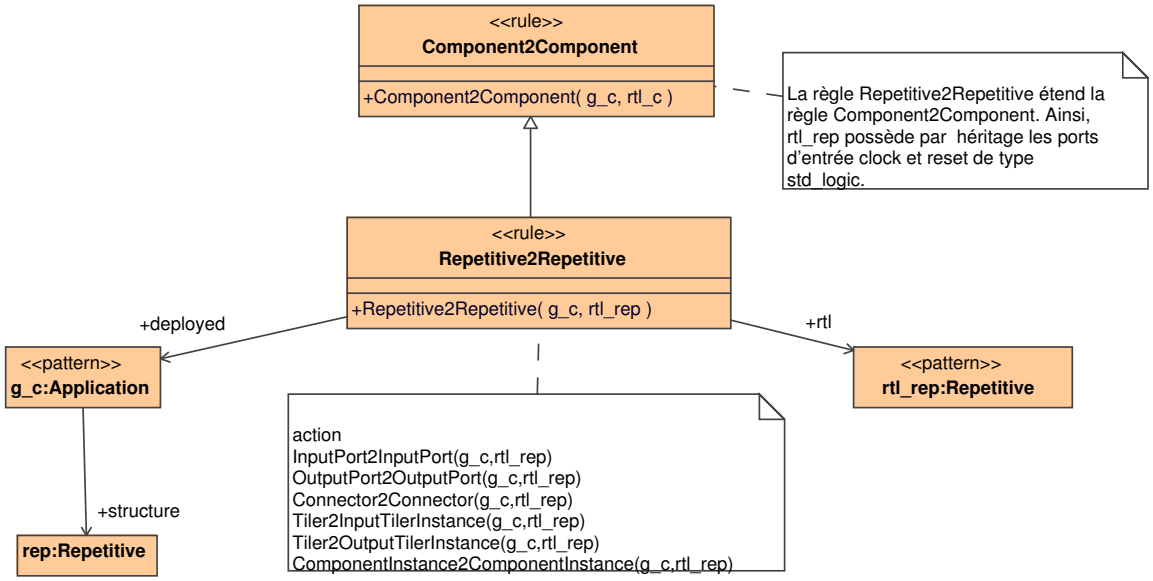


FIG. 5.5: Représentation TrML de la règle REPETITIVE2REPETITIVE. L'héritage sur la règle COMPONENT2COMPONENT permet de factoriser la génération des ports d'horloge et de remise à zéro lors de la transformation d'un composant dans un modèle Gaspard en composant dans un modèle RTL.

La règle COMPONENTINSTANCE2COMPONENTINSTANCE génère une instance de composant dans le modèle RTL et connecte les instances des ports d'horloge et de remise à zéro avec les ports d'horloge et de remise à zéro du composant qui instancie le composant. Les règles de création des ports et des instances de composant ne sont pas présentées dans ce document. La section suivante détaille la règle qui transforme un tiler du modèle Deployed en tiler dans un modèle RTL.

5.2.4 Implémentation des règles de transformation

A l'aide de représentations graphiques, nous avons formalisé et documenté chacune des règles intervenant dans la transformation d'un modèle Deployed en un modèle RTL. La représentation de certaines de ces règles en TrML a nécessité des extensions de TrML, nous avons donc contribué à l'évolution de TrML au sein de notre équipe.

La représentation graphique de TrML facilite le découpage d'une transformation de modèle en règles. Au sein d'une règle, les concepts nécessaires à l'exécution de cette règle et les relations entre ces concepts sont clairement identifiés. La projection des règles représentées en TrML vers des règles exécutables par un moteur de transformation n'est à ce jour pas terminée [43], il est donc nécessaire de développer les règles exécutables en profitant de leur représentation TrML.

Nous introduisons le moteur de transformation MoMoTE (Model to Model Transformation Engine) développé au sein de notre équipe et pour lequel nous donnons le code de la règle REPETITIVE2REPETITIVE dont nous venons d'illustrer la représentation TrML.

5.2.4.1 MoMoTE : Model to Model Transformation Engine

Les constats issus de l'utilisation des moteurs de transformations existants[3] ont amené notre équipe à proposer MoMoTE, et cela afin de satisfaire aux exigences des différents concepteurs de l'environnement Gaspard. L'élément le plus décisif pour la création de MoMoTE reste l'impossibilité d'appeler des boîtes noires dans les autres moteurs de transformations[4].

Nous présentons dans un premier temps les caractéristiques techniques de MoMoTE, puis décrivons le moteur d'exécution des règles et terminons sa présentation par la description des règles.

API MoMoTE MoMoTE est une API Java intégrée dans l'environnement Eclipse sous la forme d'un plugin. MoMoTE est codé en Java 5.0 et permet la manipulation des modèles par le biais de EMF [41]. Les requêtes dans les modèles sont réalisées avec EMFT Query.

MoMoTE permet la transformation de N modèles d'entrée en M modèles de sortie. Les règles s'appliquent aussi bien sur un modèle complet que sur un sous-ensemble de ce modèle et sont capables de créer en sortie la racine d'un patron. Les règles s'exécutent en deux temps : la première passe crée les éléments dans le modèle de sortie et la seconde résout les références entre les différents éléments.

Nous l'avons vu précédemment, l'atout majeur de MoMoTE vis-à-vis des autres moteurs de transformations est la possibilité d'appeler des boîtes noires. Sans cela, nous ne pourrions exécuter la règle de précalcul des tilers par exemple. Par ailleurs, MoMoTE utilise le mécanisme d'héritage de règles, ce qui permet la factorisation des points communs de différentes règles.

D'un point de vue plus technique et pour l'utilisateur de MoMoTE que nous sommes (un concepteur de règles), les règles s'écrivent sous la forme d'une classe Java et de cinq méthodes exécutées dans l'ordre suivant :

- `initRule` : contient l'appel aux sous-règles (fonction addRule qui prend deux paramètres, le premier correspond à la référence dans le métamodèle destination qui va contenir les éléments créés par les sous-règles correspondant au second paramètre) ;
- `getCondition` : une condition d'exécution de la règle, décrite sous la forme d'une requête dans le modèle source. Cette requête est équivalente à la description des patrons d'entrée en TrML ;
- `create` : une création de la racine du patron de sortie de la règle. Cela est équivalent, en TrML, à l'élément du patron référencé par règle et stéréotypé ≪Pattern≫ ;
- `process` : une création des éléments dans le patron de sortie de la règle ;
- `processReferences` : permet la résolution des références entre les éléments du patron de sortie créés lors de l'exécution de la méthode `process`.

Nous illustrons dans la section suivante ces différentes méthodes au travers de la description d'une règle de transformation en MoMoTE.

[3] Dans le chapitre 1, nous avons vu qu'aucun standard n'est supporté pour l'exécution des règles de transformations.

[4] Dans le cadre plus précis de la transformation de modèles Deployed2RTL, le précalcul des tilers que nous détaillerons par la suite est implémenté sous la forme d'une fonction. Cette fonction est considérée comme une boîte noire du point de vue de la transformation de modèles, appelée depuis une règle de transformation par MoMoTE.

5.2.4.2 Règle REPETITIVE2REPETITIVE en MoMoTE

Nous choisissons d'illustrer la règle de transformation REPETITIVE2REPETITIVE représentée précédemment dans ce chapitre en TrML. Voici le code de la règle :

```
public class Repetitive2Repetitive extends Component2Component {

  public initRule()
  {
    addRule(ports,new InputPort2InputPort());
    addRule(ports,new OutputPort2OutputPort());
    addRule(connector,new Connector2Connector());
    addRule(refTilerInstance,new Tiler2InputTilerInstance());
    addRule(refTilerInstance,new Tiler2OutputTilerInstance());
    addRule(refComponentInstance,new ComponentInstance2ComponentInstance());
  }

  @Override
  protected EObjectCondition getCondition(EObject srcElementContext)
  {
    return new EObjectReferenceValueCondition(
        new EObjectTypeRelationCondition(
            getApplicationComponent()),
            getComponent_Structure(),
            new EObjectTypeRelationCondition(getRepetitive()));
  }

  @Override
  protected EObject create(EObject srcElement)
  {
    return createRepetitive();
  }

  @Override
  protected void process(EObject srcElement, EObject tgtElement)
  {
    //ne rien faire
  }

  @Override
  protected void processReferences(EObject srcElement, EObject tgtElement)
  {
    //ne rien faire
  }
}
```

La règle REPETITIVE2REPETITIVE est relativement simple puisque son patron d'entrée est composé de deux éléments, son patron de sortie d'un seul élément, comme décrit par la représentation TrML de cette règle précédemment. Cette règle appelle d'autres règles par l'intermédiaire des méthodes ADDRULE. On y retrouve les règles appelées depuis la représentation de cette règle en TrML. La condition d'exécution de la règle REPETITIVE2REPETITICE permet de reconstruire son patron d'entrée : un composant applicatif dont la structure est répétitive. La méthode de création permet de construire la racine du patron de sortie, qui correspond à un composant répétitif du métamodèle RTL dans ce cas. Aucun autre élément n'est créé par cette règle, ce qui explique que les deux dernières méthodes PROCESS et PROCESSREFERENCES soient vides.

Durant nos travaux, nous avons codé en MoMoTE chaque règle nécessaire à la transformation d'un modèle Deployed en un modèle RTL.

5.2.5 Règle TILER2INPUTTILER

Cette section présente la règle qui permet de transformer un tiler dans un modèle Deployed en tiler d'entrée dans un modèle RTL.

5.2.5.1 Objectif de la règle

Dans le métamodèle Deployed, les tilers existent sous la forme de connecteurs répétitifs qui expriment une dépendance entre un port d'un composant COMPONENT et l'instance répétée de ce composant. Il n'existe pas de notion de tiler d'entrée ou de sortie dans le métamodèle Deployed, contrairement au métamodèle RTL. Le premier objectif de la règle TILER2INPUTTILER est donc de définir l'orientation du connecteur tiler afin de lui attribuer la bonne fonctionnalité : un tiler d'entrée ou un tiler de sortie.

Le modèle d'exécution matérielle des applications Gaspard prévoit que les implémentations issues du précalcul des tilers soient regroupées dans des composants matériels dédiés à cet effet. L'objectif de la règle TILER2INPUTTILER est de créer ce type de composants et de les instancier dans des composants répétitifs du métamodèle RTL. Dans le métamodèle Deployed, les tilers sont exprimés sous une forme origine, pavage et ajustage. Le précalcul des tilers tel que défini dans le modèle d'exécution matérielle des applications Gaspard prévoit que les tilers prennent la forme de connecteurs ou de registres à décalage dans le métamodèle RTL. Ce précalcul des tilers est indépendant de la règle TILER2INPUTTILER et est introduit sous la forme d'une boîte noire dans la section 5.2.7.

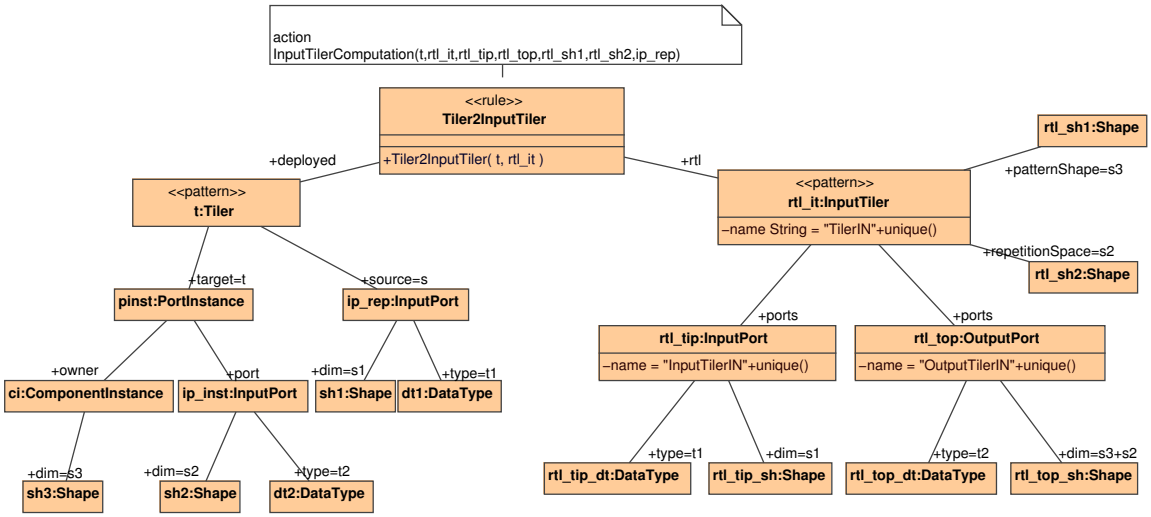


FIG. 5.6: Représentation TrML de la règle TILER2INPUTTILER.

5.2.5.2 Détails de la règle

Le patron d'entrée de la règle TILER2INPUTTILER, représenté à la figure 5.6, est composé du concept TILER *t* dont la source est un port d'entrée et la cible une instance de port. Cette direction correspond à la lecture d'un tableau sur un composant et à l'écriture des motifs sur les instances de port du composant répété. L'inverse est appliqué pour ne récupérer que les tilers de sortie : la source connectée à une instance de port et la cible connectée à un port de sortie. Dans ce chapitre, nous ne présentons que la création d'un tiler d'entrée (la création d'un tiler de sortie est très similaire.

La source et la destination de ce tiler sont donc extraites car elles permettent d'identifier le sens du tiler. Des informations relatives à cette source et à cette destination sont par ailleurs transformées en concept du métamodèle RTL. Ces informations sont le type et la dimension des tableaux de données véhiculés par le tiler : la source *ip_rep* permet un accès direct à ces informations, tandis que la destination *pinst* (concept PORTINSTANCE) nécessite de récupérer le port *ip_inst* qu'il référence.

Le calcul des tilers nécessite l'espace de répétition de la tâche. Dans le métamodèle Deployed cet espace de répétition est toujours présent ; il est attaché à l'instance de composant sur lequel porte la répétition. Cet espace de répétition apparaît sous la forme de SHAPE noté *sh3*.

La racine du patron de sortie est le tiler d'entrée INPUTTILER dans le métamodèle RTL appelé *hw_it*. Le nom de ce composant est composé d'une partie générique, « TilerIN » et d'une chaîne de caractères unique. En effet, les tilers dans le métamodèle Deployed sont des spécialisations de connecteurs et ne sont pas nommés. Cependant, pour les besoins de la compilation, un nom doit être affecté aux composants tiler dans le métamodèle RTL.

hw_it dépend de deux concepts SHAPE du métamodèle RTL, définis par les références *patternShape* et *repetitionShape* qui correspondent respectivement au domaine de répétition de la tâche et à la forme des motifs produits par le tiler.

- Le port d'entrée *rtl_tip* possède les mêmes caractéristiques que la source du connecteur tiler *t* dans le patron d'entrée. Le type de *rtl_tip* correspond au résultat de la transformation dans le métamodèle RTL du type associé à la relation *t1* dans le métamodèle Deployed. Il en va de même pour la forme des données véhiculées sur le port *rtl_tip* avec la relation *s1*.
- Le port de sortie *rtl_top* possède le type *rtl_top_sh*, résultat de la transformation du type *dt2* dans le métamodèle RTL. La forme des données véhiculées par *rtl_top* est particulière puisqu'elle correspond à la SHAPE *rtl_top_sh*. Cette SHAPE est issue du produit des transformations vers le métamodèle RTL des concepts SHAPE *sh2* et *sh3*. Cela signifie que le port de sortie *rtl_top* est construit à partir de la forme du motif et de l'espace de répétition de la tâche.

La règle TILER2INPUTTILER appelle une boîte noire du point de vue de la transformation de modèle qui calcule le contenu d'un tiler (nous détaillerons la fonction associée par la suite). Cet appel est paramétré par *TilingDescription*, qui correspond aux descriptions des dépendances de données exprimées en Array-OL. Cet appel est aussi paramétré par les ports de *rtl_it*, l'espace de répétition et la forme des motifs. Cette boîte noire précalcule le tiler et le transforme en circuit tel que défini dans la section 3.1.2. Cette fonction fait intervenir un algorithme complexe qui correspond à du code sous la forme impérative et ne peut donc pas être représenté en TrML.

Le résultat de la règle TILER2INPUTTILER est la création d'un composant tiler à par-

tir d'un connecteur tiler. Pour que ce composant puisse être utilisé, il est instancié dans le contexte d'un composant répétitif. La règle suivante présente cette instanciation.

5.2.6 Règle TILER2INPUTTILERINSTANCE

La règle TILER2INPUTTILER que nous venons de décrire transforme un tiler dans le métamodèle Deployed en un composant tiler d'entrée dans le métamodèle RTL. Pour que ce composant tiler puisse être utilisé, il est instancié dans un composant répétitif du métamodèle RTL et les instances de ses ports connectées.

5.2.6.1 Objectif de la règle

La règle TILER2INPUTTILERINSTANCE illustre l'instanciation d'un composant INPUTTILER et son utilisation dans un composant répétitif du métamodèle RTL. De la même que pour la création des composants tiler, la génération des instances de tilers de sorties est symétrique à celle des instances de tilers d'entrée. Dans ce document, nous ne présentons que la règle qui permet d'instancier un tiler d'entrée.

5.2.6.2 Détails de la règle

Le patron d'entrée de la règle TILER2INPUTTILERINSTANCE, qui est représentée à la figure 5.7, est composé du tiler *t* ainsi que de ses source *pinst* et destination *ip_rep* qui permettent de détecter si le tiler est un tiler d'entrée.

La règle TILER2INPUTTILERINSTANCE crée l'instance de composant *rtl_ci* qui référence le tiler d'entrée *rtl_it*. L'instance de port *rtl_pi1* référence le port d'entrée *rtl_tip* du tiler d'entrée et est connectée au port d'entrée du composant répétitif *rtl_ip_rep* par le biais du connecteur *rtl_sc*. Les données qui circulent sur ce connecteur possèdent le type et la forme des données véhiculées sur le port *rtl_tip*, par le biais des références *t1* et *s1*. Symétriquement, nous connectons l'instance de port *rtl_pi2* avec l'instance de port *rtl_pinst*, qui correspond à l'instanciation d'un port d'entrée du composant dans le composant répétitif.

Par ailleurs, cette règle appelle la règle COMPONENTINSTANCE2COMPONENTINSTANCE qui permet de connecter les instances de port de l'horloge et de la remise à zéro avec ceux du composant qui instancie le tiler.

5.2.7 Fonction de précalcul des tilers d'entrée

L'exécution matérielle du parallélisme de données nécessite que les dépendances de données exprimées par des tilers Array-OL soient analysées afin de connecter chaque donnée d'un motif avec une donnée dans un tableau. Cette étape est appelée précalcul des tilers car les dépendances de données sont calculées avant que le circuit ne soit exécuté sur le FPGA. Le précalcul des tilers de sortie est symétrique à celui des tilers d'entrée, nous ne décrivons donc que la manière dont les tilers d'entrée sont précalculés.

Le calcul des tilers lors d'une transformation de modèles vers le métamodèle RTL permet de conserver les dépendances de données sous la forme d'un modèle et d'isoler les concepts de toute syntaxe HDL. Le résultat du précalcul d'un tiler est un réseau d'interconnexion reliant les données des motifs du tiler à des données dans un tableau : les connexions elles-mêmes sont des concepts du métamodèle RTL et ont été introduites dans le chapitre précédent.

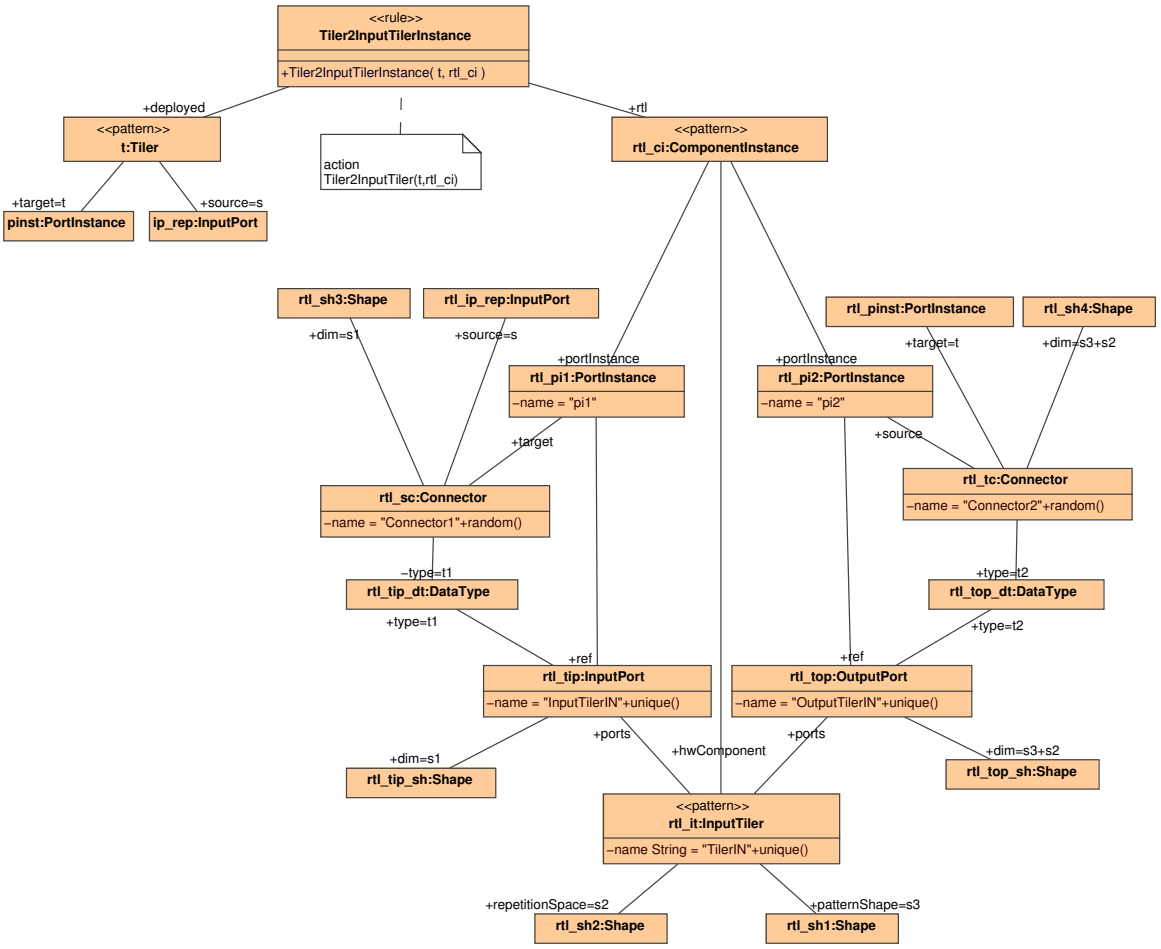


FIG. 5.7: Représentation TrML de la règle TILER2INPUTTILERINSTANCE.

La fonction de précalcul de tiler que nous présentons dans cette section est utilisable pour la génération des codes VHDL ou Verilog.

5.2.7.1 Fonction de précalcul des tilers dans la transformation de modèles Deployed2RTL

La combinaison des règles de transformations dont dépendent les tilers avec la fonction de précalcul des tilers du modèle Deployed permet de générer de façon entièrement automatisée un modèle RTL qui permet l'exécution matérielle de l'application Gaspard. La partie en haut à gauche de la figure 5.8 représente une application répétée sur le temps. Le tableau d'entrée contient deux dimensions, l'une est de taille 2 et représente l'espace, l'autre est infinie et représente le temps. La tâche, dont les répétitions sont représentées par des couleurs sur cette figure, consomme deux données. Les dépendances de données (issues du tiler) montrent que l'itération d'une tâche à l'instant t consomme, dans le tableau, des données présentes à ce même instant t. Cela signifie qu'un connecteur suffit à gérer cette dépendance de données. L'algorithme que nous présentons dans cette section suffit à construire ce

connecteur. Ainsi, lors de la transformation de cette application vers le modèle RTL, les ports de dimension $[(2, \tilde{})]$ sont ramenés à la dimension $[(2,)1]$, la tâche répétée une infinité de fois est transformée en une seule unité de calcul et le tiler est compilé en connecteur simple.

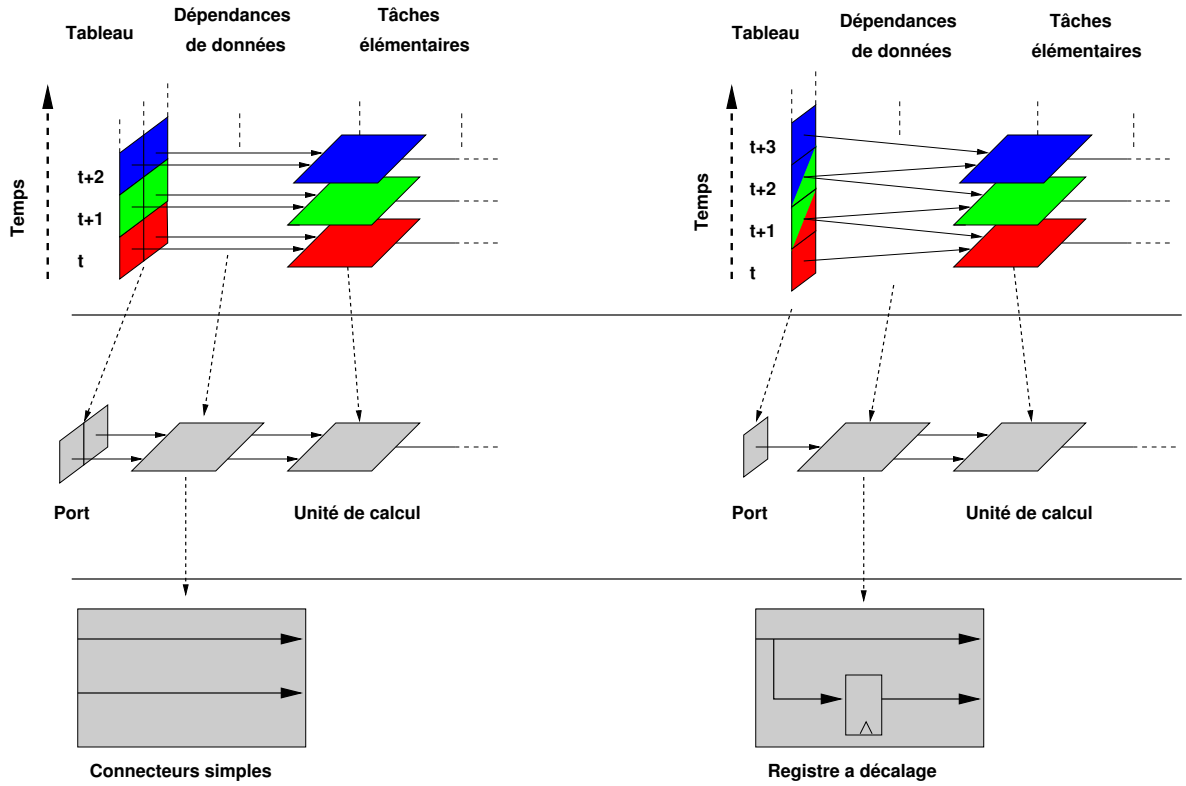


FIG. 5.8: Représentation des transformations de deux applications décrites en Array-OL (en haut) vers des modèles RTL (au milieu), puis représentation du résultat de la fonction de calcul du tiler (en bas). Le passage des deux applications en Array-OL produit des structures équivalentes dans les modèles RTL, la seule différence provient de la dimension du port (deux pour la gauche de la figure, un pour la droite de la figure). Le modèle RTL est généré à partir de règles de transformation. Le passage du milieu au bas de la figure illustre l'utilisation de la fonction de calcul des tilers. La gauche de la figure représente une application dont le tiler exprime des dépendances de données qui ne dépendent pas du temps. Ce tiler est transformé en connecteurs simples, c'est à dire qui ne gèrent pas les dépendances sur le temps. La partie en haut à droite de la figure représente une application dont le tiler exprime des dépendances de données sur le temps. Ce tiler est transformé en registre à décalage, qui gère le recouvrement des données sur le temps.

5.2.7.2 Calcul des dépendances de données exprimées par les tilers

Rappel des équations de calcul des tilers Le précalcul des tilers d'entrée est réalisé par la fonction COMPUTEINPUTTILER appelée depuis la règle TILER2INPUTTILER. Cette fonction est une boîte noire du point de vue des transformations de modèle car le précalcul des tilers ne peut être formalisé sous la forme de patron. Les tilers identifient les dépendances de don-

nées selon les équations 5.1 et 5.2 : l'équation 5.1 renvoie les coordonnées (dans un tableau de données) de l'origine d'un motif tandis que l'équation 5.2 renvoie les coordonnées de chacun des éléments de ce motif.

$$\forall \overrightarrow{x_q}, \overrightarrow{0} \leq \overrightarrow{x_q} < \overrightarrow{Q}, \overrightarrow{r_q} = (\overrightarrow{o} + P \times \overrightarrow{x_q}) \mod \overrightarrow{m} \tag{5.1}$$

$$\forall \overrightarrow{x_d}, \overrightarrow{0} \leq \overrightarrow{x_d} < \overrightarrow{D}, (\overrightarrow{r_q} + F \times \overrightarrow{x_d}) \mod \overrightarrow{m} \tag{5.2}$$

Le calcul d'un tiler est décomposé en deux étapes : la première étape exploite l'équation 5.1 et calcul l'origine de chaque motif dans le tableau. La seconde étape du calcul exploite l'équation 5.2 et renvoie la position d'un élément de motif dans le tableau. Cette position est relative à la position de l'origine dans le tableau, il est donc nécessaire de faire la somme sur chacune des dimensions du tableau entre l'origine du motif et la position relative à ce motif.

Algorithme de précalcul des tilers et gestion des dépendances sur l'espace Les cas les plus simples à gérer sont les dépendances de données sur l'espace : ce type de dépendances de données est réalisable par de simples connexions (fil). Nous présentons ci-après l'algorithme `InputTilerComputation` qui génère des connecteurs de façon à réaliser les dépendances de données identifiées lors du précalcul des tilers. Cet algorithme correspond à la boîte noire appelée depuis la règle TILER2INPUTTILER et est paramétré par le tiler du modèle Deployed, le tiler d'entrée du modèle RTL généré dans la règle de transformation TILER2INPUTTILER, ses ports d'entrée et de sortie, l'espace de répétition de la tâche, la forme des motifs et le port d'entrée du modèle Deployed qui correspond au tableau d'entrée de la tâche. Cet algorithme créé des connecteurs avec indices qui sont attachés au composant tiler généré en sortie de la règle. Un connecteur avec indices possède des champs source et destination pour son orientation et des champs d'indice qui lui permettent de définir les coordonnées dans la source et la destination des données qu'il connecte. Ces sont ces champs que remplie cet algorithme :

```
InputTilerComputation(Tiler t, InputTiler rtl_it, InputPort rtl_tip,
  OutputPort rtl_top, Shape rtl_sh1, Shape rtl_sh2, Gaspard.InputPort ip_rep)
{

Pour chaque motif m de l'espace de repetition rtl_sh1 Faire
   Pour chaque element e du motif m de forme rtl_sh2 Faire

      c= new IndexConnector
      c.tilerOwner(rtl_it)
      rtl_it.getIndexConnector().add(c)

      c.destination = rtl_top
      c.source = rtl_tip

      c.destination.indice = m.e
      c.source.indice = calcul_Array-OL(t,m,e)

      ShiftRegister(c, ip_rep)
   FinPour
FinPour
}
```

Cet algorithme itère sur l'espace de répétition multidimensionnel d'une tâche et chaque donnée du motif multidimensionnel par deux boucles imbriquées. Pour chaque donnée de chaque motif, un connecteur avec indices est créé. Le reste de l'algorithme permet de remplir les différents champs du connecteur : sa source correspond au tableau d'entrée et sa destination au motif créé. L'indice de la destination du connecteur définit la position de la donnée dans les motifs de sortie et dépend directement des positions dans l'espace de répétition et dans le motif. L'indice de la source est issu du résultat du calcul des équations 5.1 et 5.2. Le résultat d'une itération de l'algorithme est la connexion d'une donnée dans un motif avec une donnée dans le tableau. L'ensemble des itérations sur l'espace de répétition et sur les données de chaque motif génère un ensemble de connecteurs. Par ailleurs, cet algorithme appelle l'algorithme `ShiftRegister` qui permet de gérer certaines dépendances de données sur le temps. Cet algorithme est détaillé par la suite dans ce manuscrit.

Exemples de précalcul de tilers La figure 5.9 représente des circuits générés pour de simples dépendances de données. La gauche de chaque tiler représente un tableau et la droite les motifs. Les pastilles rectangulaires représentent des éléments atomiques dans ces tableaux et motifs et les liens les connecteurs avec indices qui réalisent les dépendances de données dans le circuit.

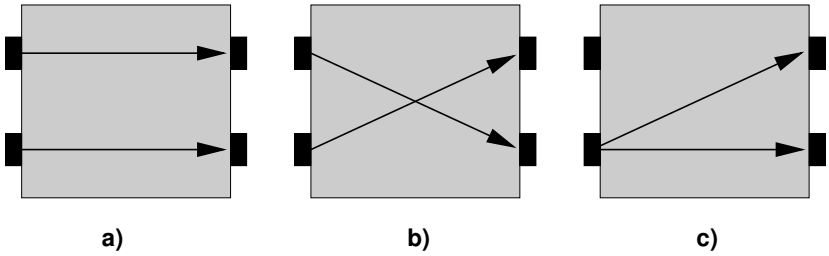


FIG. 5.9: Différents circuits issus du précalcul des tilers et qui réalisent des dépendances de données dans l'espace.

5.2.7.3 Gestion des glissements sur le temps

Les tilers permettent de traiter des flots de données dont certaines dépendances de données, décrites par des tilers, expriment des glissements sur le temps. Ces glissements sont couramment utilisés dans le domaine du traitement de signal car ils permettent de filtrer des signaux sur le temps. Ainsi, dans le cas d'une dépendance de données telle que décrite ici, le précalcul des tilers par l'algorithme 5.2.7.2 renvoie la position d'un élément dans le flot infini de données, comme illustrée par la partie en haut à droite de la figure 5.8. La fonction crée une « connexion » entre un élément de motif à un instant t et un élément dans le tableau à l'instant passé t-n. La transformation de modèles génère un composant matériel tiler (à droite et au milieu de la figure) avec un port d'entrée de dimension $[(1)]$ qui est issue du flot infini de données et un port de sortie de dimension $[(2)]$ qui correspond à la dimension du motif. Ce tiler matériel lie la donnée présente sur le port d'entrée à l'instant t, retarde cette donnée durant n cycles et l'envoie à l'un des ports de sortie. La partie en bas à droite de la figure 5.8 illustre la structure de ce tiler matériel. Les registres à décalage remplissent parfai-

tement cette fonction de retard et de stockage : la réalisation matérielle de la dépendance de donnée est donc réalisée au travers de registres à décalage.

5.2.7.4 Optimisation de la création des registres à décalage

La partie en haut à gauche de la figure 5.10 représente la création de registres à décalage lors d'un glissement sur le temps des motifs d'une tâche répétée. Lorsque le calcul d'une dépendance de données renvoie une position antérieure à l'instant de l'exécution de la tâche, un registre à décalage est créé, comme illustré par la partie en bas à gauche de la figure 5.10. Cette figure illustre la création de quatre registres à décalage, chacun d'eux décale la même donnée mais avec une profondeur dans le décalage qui varie. Ainsi, il est intéressant de limiter le surcoût lié à l'introduction de registres à décalage en réutilisant ou en modifiant ceux existants.

Le concept de registre à décalage est donc modélisé de manière à permettre les connexions dans les différents étages de ce circuit : un registre à décalage de longueur n est accessible aux étages n, $n\text{-}1 \ldots 1$ et 0 (l'étage 0 correspond à la donnée présente à l'instant t sur le port et qui entre dans le registre à décalage). Il devient alors possible d'optimiser l'utilisation de ces registres à décalage en permettant de multiples connexions. Il est possible d'augmenter la longueur du décalage afin de permettre la lecture d'un élément plus ancien ou de réaliser une connexion dans un registre à décalage existant. La partie droite de la figure 5.10 illustre l'utilisation d'un même registre à décalage pour la création de quatre dépendances de données. Une économie en ressources est donc réalisée.

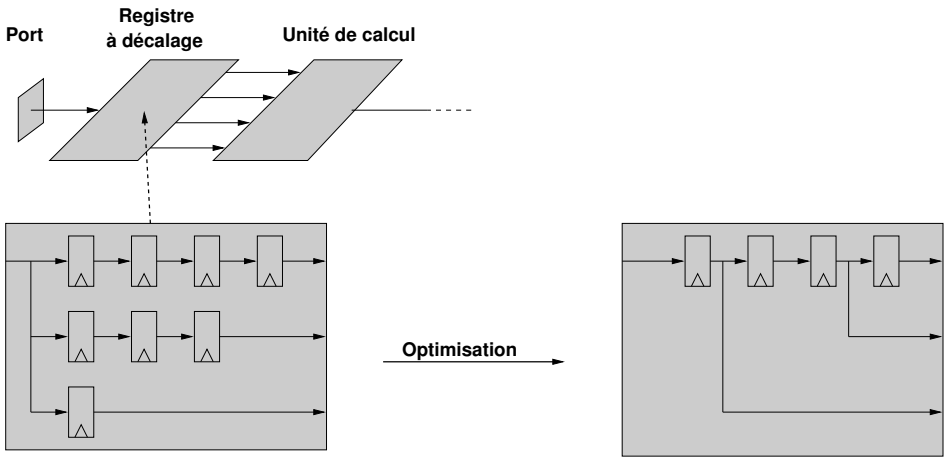


FIG. 5.10: Les dépendances de données, issues du calcul d'un tiler, génèrent des registres à décalage. Ils retardent les données et permettent de gérer les glissements de motifs sur le temps. L'utilisation de ces registres est alors optimisée de manière à réduire leur coût d'implémentation.

5.2.7.5 Gestion des registres à décalage dans l'algorithme de précalcul des tilers

Afin de prendre en compte la création des registres à décalage et leur ré-utilisation dans un tiler, nous proposons un second algorithme qui est appelé depuis l'algorithme de précal-

cul présenté précédemment. Cet algorithme prend en paramètre le connecteur créé dans le précédent algorithme, ainsi que le port d'entrée du composant applicatif Deployed :

```
ShiftRegister(IndexConnector c, Gaspard.InputPort ip_rep)
{

Pour chaque dimension dim du tableau d'entree ip_rep Faire
    Si ip_rep.dim.value = infini
      Si c.source.indice(dim) != 0 Alors
         delai = c.source.indice(dim) //sauvegarde du delai
         c.source.indice(dim) = 0
         Si bd.get(c.source.indice) = vrai Alors
            r = DelayedIndexConnector identifie dans bd
            Si delai > r.delay Alors
               r.delay = delai
            FinSi
            c.source = r
            c.source.indice = delai

         Sinon
            r = new DelayedIndexConnector
            r.source = c.source
            r.source.indice = c.source.indice
            r.delay = delai
            c.source = r
            c.source.indice = delai
            bd.add(r)
         FinSi
      FinSi
  FinSi
FinPour
}
```

Description de l'algorithme La première partie de l'algorithme détecte si une dimension infinie (le temps) existe dans le tableau d'entrée. Si effectivement une dimension temporelle existe, nous regardons la valeur de la coordonnée sur la dimension temporelle. Si elle est différente de 0, cela signifie que le connecteur lit une donnée qui n'est plus présente, il est donc nécessaire de réaliser la connexion via un registre à décalage. Pour cela, la valeur de la coordonnée sur la dimension temporelle est sauvegardée dans la variable *delai* et le champ source du connecteur est supprimé.

Afin d'éviter de créer inutilement des registres à décalage, nous regardons dans une base de données si la donnée du tableau est déjà retardée par un registre à décalage pour les besoins de réalisation d'une autre dépendance de données. Si c'est le cas, nous comparons notre délai avec le délai du registre à décalage existant. S'il le délai actuel est supérieur, nous augmentons la taille du registre à décalage, sinon, il n'est pas modifié. La source du connecteur est alors connectée à la position *delai* dans le registre à décalage.

Si la donnée que nous souhaitons décaler n'est pas présente dans la base de données, cela signifie qu'il n'y a pas de registre à décalage associé à cette donnée, nous créons donc un registre à décalage dont la source est connectée à la donnée que nous souhaitons retarder. La longueur de ce registre est définie par *delai*. La source de notre connecteur est alors connectée

à la position *delai* du registre à décalage.

Bilan Cet algorithme permet la gestion glissements de données sur le temps en réutilisant si besoin des registres à décalage existants. Les registres à décalage sont créés indépendamment les uns des autres, un composant tiler peut donc en contenir plusieurs. L'unique restriction de cet algorithme est qu'il ne fonctionne que lorsque les tilers expriment un pavage de un sur la dimension temporelle, ce qui signifie que les données sur les ports sont cadencées au même rythme que les calculs dans les composants répétitifs. Cette contrainte existe déjà dans le modèle d'exécution matérielle des applications Gaspard décrit dans le chapitre précédent : la gestion des glissements sur le temps n'a donc pas d'impact négatif sur les autres applications gérées par notre modèle d'exécution.

5.2.7.6 Bilan de la fonction de calcul des tilers

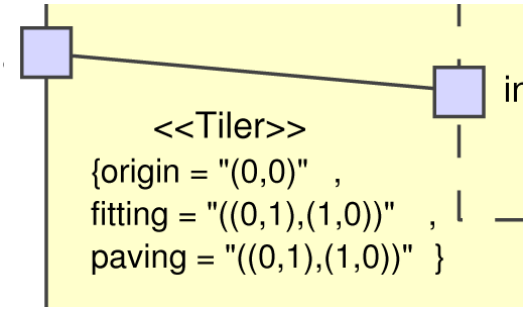
Dans cette section, nous venons de présenter la fonction de précalcul les tilers et génèrent des connecteurs, réalisant ainsi les dépendances de données. Un premier algorithme gère les dépendances de données sur l'espace et appel un second algorithme dédié à la gestion des dépendances de données sur le temps : nous gérons les glissements sur le temps, très courants en traitement du signal et compilés dans aucun autre travail concernant les exécutions de Array-OL. Le précalcul des tilers est donc pleinement justifié. Le précalcul des tilers ne souffrent d'aucune restriction sur le nombre de dimensions que possèdent les motifs, les tableaux, etc., nous gérons donc les dépendances de données multidimensionnelles.

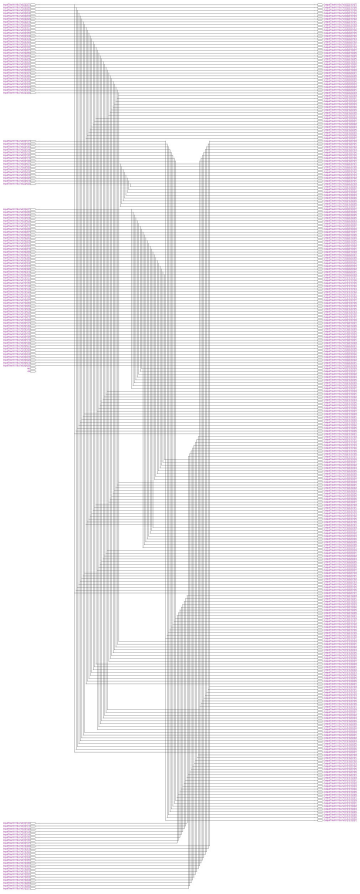
Pour finir, les connecteurs sont créés sous la forme de concepts du métamodèle RTL et sont donc indépendants des syntaxes HDL : la même fonction peut être utilisée pour la génération de code VHDL ou Verilog des tilers. La fonction de précalcul des tilers est présentée en Annexe C.

5.3 Transformation depuis un modèle Deployed vers du code VHDL

Afin d'illustrer le fonctionnement complet de notre chaîne de transformations, nous reprenons l'exemple du filtre d'images. Nous avions alors illustré la génération de code VHDL depuis le modèle RTL, sans préciser que ce modèle est généré depuis un modèle Deployed. Nous reprenons ici une partie des résultats de synthèse afin d'illustrer la compilation des tilers pour cette application. La modélisation en UML de ce tiler est représentée en haut à gauche de la figure 5.11 (nous rappelons que le port d'entrée de cette application est un tableau de dimension de $[(4,4)]$, que le motif est de dimension $[(3,3)]$ et que la tâche est répétée 2×2 fois), le code VHDL généré automatiquement pour ce tiler est représenté partiellement en bas à gauche. Ce code est généré depuis un modèle de tiler conforme au métamodèle RTL, ce modèle est issu de la transformation de modèle depuis le modèle Deployed représenté par la modélisation en UML.

Le composant tiler (TILERIN173BB318 dans le code VHDL pour lui garantir un nom unique) est composé des ports d'horloge, de remise à zéro (ces ports ne sont pas utilisés pour ce tiler) d'un port de données entrantes (l'image du filtre d'image) et d'un port de données sortantes (les motifs consommés par la tâche répétée). Le port d'entrée est bidimensionnel et





```
...
ENTITY TilerIN173b318 IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
InputTilerIN173b318 : IN TABLE_TYPE_4_4_INTEGER_RANGE_0_to_25
OutputTilerIN173b318 : OUT TABLE_TYPE_2_2_3_3_INTEGER_RANGE_
END TilerIN173b318;

ARCHITECTURE archiTilerIN173b318 OF TilerIN173b318 IS
BEGIN

...
OutputTilerIN173b318(1)(2)(2)(1) <= InputTilerIN173b318(2)(2);
OutputTilerIN173b318(1)(2)(2)(2) <= InputTilerIN173b318(3)(2);
OutputTilerIN173b318(1)(2)(2)(3) <= InputTilerIN173b318(4)(2);
OutputTilerIN173b318(1)(2)(3)(1) <= InputTilerIN173b318(2)(3);
OutputTilerIN173b318(1)(2)(3)(2) <= InputTilerIN173b318(3)(3);
OutputTilerIN173b318(1)(2)(3)(3) <= InputTilerIN173b318(4)(3);
OutputTilerIN173b318(2)(1)(1)(1) <= InputTilerIN173b318(1)(2);
OutputTilerIN173b318(2)(1)(1)(2) <= InputTilerIN173b318(2)(2);
OutputTilerIN173b318(2)(1)(1)(3) <= InputTilerIN173b318(3)(2);
..
END archiTilerIN173b318;
```

FIG. 5.11: La partie en haut à gauche représente l'expression des dépendances de données exprimées en ARRAY-OL, la partie en bas à gauche représente le code VHDL généré du compotant tiler et la droite de la figure le résultat de synthèse sous Quartus de ce composant. Les connecteurs sont bien visibles.

de taille $[(4,4)]$[5], cette taille correspond à l'image traitée par l'application. Le port de sortie contient 4 dimensions et est de taille $[(2,2,3,3)]$[6]. Les connecteurs simples dans le modèle RTL prennent la forme en VHDL d'une affectation des données de sorties (4 dimensions) par des données en entrées (2 dimensions).

Lors de la synthèse sous Quartus de ce code VHDL, on obtient le schéma représenté à droite de la figure 5.11, la gauche de ce schéma représente les ports d'entrées, la droite ceux de sortie. On distingue les connecteurs qui réalisent les dépendances de données multidimensionnelles entre ces ports (bien entendu, la synthèse « met à plat » ces dépendances de données pour le placement sur FPGA), mais il est cependant difficile d'identifier chacune d'elles au vu de leur nombre.

La génération complète de l'application du filtre d'image illustre le bon fonctionnement

[5] le type TABLE_TYPE_4_4_RANGE_0_TO_255 est généré automatiquement dans une librairie, il est défini de manière à préserver la multidimensionnalité du port.

[6] $(2,2)$ correspond à l'espace de répétition de la tâche, $(3,3)$ à la dimension des motifs de la tâche.

de notre flot de conception, incluant la génération du code VHDL présentée dans le chapitre précédent et la transformation de modèles détaillée dans ce chapitre. Nous détaillons l'utilisation de notre flot de conception pour une application plus significative dans le chapitre 7.

5.4 Conclusion

Ce chapitre présente les travaux réalisés pour permettre la génération d'accélérateurs matériels dans le métamodèle RTL à partir d'un modèle Deployed. Pour cela, nous avons développé la transformation de modèles Deployed2RTL décomposée en règles de transformations que nous avons représentées en TrML et implémentées en MoMoTE. Cette transformation de modèle, de la représentation graphique des règles leur implémentation, est conforme à l'approche Ingénierie Dirigée par les Modèles. Au sein de ces règles de transformations, la compilation des tilers est réalisée à l'aide d'une fonction qui génère de simples connecteurs pour les dépendances de données sur l'espace et des registres à décalage pour celles sur le temps. La gestion de ces glissements étend considérablement les applications Gaspard exécutables en matériel car ces glissements sont fréquents en traitement du signal lors des étapes de filtrage. La totalité de cette transformation de modèles est intégrée dans l'environnement Gaspard, la génération d'accélérateurs matériels à partir d'applications Gaspard est donc entièrement automatisée.

Cette transformation de modèle complète la chaîne de transformation de Gaspard qui permet de générer du code VHDL à partir de modélisations en UML.

Chapitre 6

Optimisation de l'implémentation des accélérateurs sur FPGA

Contents

La transformation de modèles Deployed2RTL est en mesure de produire un modèle RTL à partir d'un modèle Deployed, nous proposons un processus d'optimisation qui modifie un modèle Deployed afin de générer différents accélérateurs pour la même application. Notre optimisation consiste à ajuster les performances et la puissance de calcul de d'un accélérateur issu d'une transformation de modèles, cela en contrepartie d'une modification des ressources nécessaires à sa mise en œuvre sur FPGA. Le processus d'optimisation nécessite donc des informations relatives au FPGA sur lequel est implémenté l'accélérateur et des informations relatives aux caractéristiques d'implémentation. En fonction de ces informations, l'optimisation modifie la structure du modèle Deployed et la manière dont s'exécute en matériel son parallélisme de données.

La modification d'un modèle Deployed est réalisée par l'utilisation de fonctions de refactoring ARRAY-OL existantes que nous couplons au choix d'exécution du parallélisme de données (parallèle ou séquentiel). Il en résulte les fonctions de refactoring PARALLÉLISATION et SÉQUENTIALISATION qui permettent respectivement d'augmenter la quantité de ressources FPGA consommée par l'accélérateur et de la diminuer. Ces fonctions de refactoring sont utilisées dans une heuristique d'optimisation qui tend à ajuster la quantité de ressources requise par l'accélérateur avec celle disponible sur FPGA. Une originalité de cette heuristique est qu'elle pilote la transformation de modèle Deployed2RTL afin de générer un nouvel accélérateur à chaque itération.

La section 6.1 introduit les pré-requis nécessaires à la mise en œuvre d'une optimisation des accélérateurs. Nous présentons ce processus d'optimisation dans la section 6.3 et l'illustrons dans la section 6.4 au travers de l'exemple du filtre d'images.

6.1 Principe d'optimisation des accélérateurs

Afin de modifier le modèle RTL dans une optique d'optimisation, il est nécessaire de modifier la spécification initiale de l'application. Pour cela, nous prenons en compte les caractéristiques d'implémentation des accélérateurs sur FPGA.

6.1.1 Caractéristiques d'implémentations sur FPGA des accélérateurs

Dans le flot de conception, le concept de FPGA est exploité lors de la modélisation du placement des accélérateurs et de leurs caractéristiques d'implémentation. Ces placements sont directement modélisés dans le métamodèle RTL et ne sont pas issus d'une transformation de modèles car les concepts de FPGA et de placement géographique ne sont pas supportés dans le métamodèle Deployed, point d'entrée de notre flot de conception. En revanche, nous avons présenté dans le chapitre 3 une évaluation des caractéristiques de l'implémentation d'un composant sur un FPGA : l'évaluation des caractéristiques de chaque composant d'un accélérateur repose sur le principe que les caractéristiques d'implémentation des tâches élémentaires sont connues. A partir de ces tâches élémentaires et en remontant dans la hiérarchie de l'accélérateur, nous sommes en mesure d'évaluer les caractéristiques d'implémentation.

Dans le métamodèle RTL, les caractéristiques d'implémentation d'un composant sont conceptualisées par COMPONENTFPGAIMPLEMENTATION. Ce concept modélise le type et la quantité de ressources d'un FPGA nécessaires à l'implémentation d'un composant. Le concept COMPONENTFPGAIMPLEMENTATION associé au composant de plus haut niveau de hiérarchie de l'accélérateur reflète donc les ressources nécessaires à l'implémentation de

l'accélérateur, en particulier en terme de surface du FPGA nécessaire. Il est ainsi possible d'estimer, avant l'étape de synthèse par un outil commercial, si le FPGA supporte l'implémentation de l'accélérateur et dans quelle mesure il la permet. Ces estimations sont la base de notre processus d'optimisation.

6.1.2 Refactoring du modèle d'application

Des travaux menés dans notre équipe [39, 110, 19] visent à modifier la structure des applications ARRAY-OL afin de l'adapter à sa cible d'exécution. [1]. On retrouve parmi ces fonctions de refactoring le changement de pavage (fonction de refactoring *Change paving*), le *Tiling* et l'aplatissement (*Collapse*).

Ces fonctions de refactoring sont illustrées à la figure 6.1, les cercles représentent des tâches, les indices leur espace de répétition. La hiérarchie dans les tâches est illustrée du haut vers le bas. Nous commentons ces fonctions de refactoring et évaluons leur impacte sur les accélérateurs générés dans notre flot de conception :

- *Tiling* : introduit dans l'application une tâche hiérarchique, comme illustré sur la partie gauche de la figure 6.1. La tâche créée permet l'exécution partielle du parallélisme de données de la tâche initiale. *Tiling* a donc un impact direct sur la structure de l'accélérateur qui prend en compte la hiérarchie introduite. L'impact sur la quantité de ressources nécessaires à l'implémentation de l'accélérateur est plus difficile à quantifier puisqu'il dépend du choisi de l'exécution du parallélisme de données (nous rappelons que l'exécution séquentielle du parallélisme de données n'est autorisée qu'au niveau le plus haut de la hiérarchie). S'il est exécuté en séquentiel, on assiste à une augmentation de la quantité de ressources nécessaire à son implémentation puisque la répétition autour de la sous-tâche créée est exécutée en parallèle. S'il est exécuté en parallèle, le nombre d'unités de calcul instancié est identique et la quantité de ressources FPGA ne varie donc pas. Une exception concerne toutefois l'utilisation de *Tiling* sur une application possédant un espace de répétition infini sur une dimension (que nous considérons être le temps). Dans ce cas précis, l'espace de répétition de la sous-tâche créée exprime une partie du parallélisme initial de l'application sur le temps en parallélisme sur l'espace : la quantité de ressources consommées par l'accélérateur augmente.
- *Change paving* : permet de descendre des répétitions autour d'une tâche dans les sous-tâches de sa hiérarchie : du parallélisme de données est exécuté plus bas dans la hiérarchie. *Change paving* est différent *Tiling* dans la mesure où il ne crée pas de hiérarchie, cependant, son impact sur les ressources consommées par l'accélérateur est identique.
- *Collapse*, ou aplatissement en français, supprime un niveau de hiérarchie en remontant le parallélisme des sous-tâches supprimées. Lorsque le parallélisme de l'application est exécuté en parallèle, *Collapse* ne modifie pas les ressources que consomme l'accélérateur. En revanche, s'il est exécuté en séquentiel, la répétition de la sous-tâche (initialement exécuté en parallèle) est exécutée en séquentiel : la quantité de ressources diminue. Il est à noter que *Collapse* ne permet pas de passer d'une expression de l'espace de répétition sur l'espace en espace de répétition sur le temps.

[1] La section 1.3 présente un ensemble de fonctions de refactoring ARRAY-OL qui permettent de modifier la structure d'une application. Le résultat d'une application ARRAY-OL *refactorée* reste une application ARRAY-OL : il est donc possible de combiner différentes fonctions de refactoring ARRAY-OL.

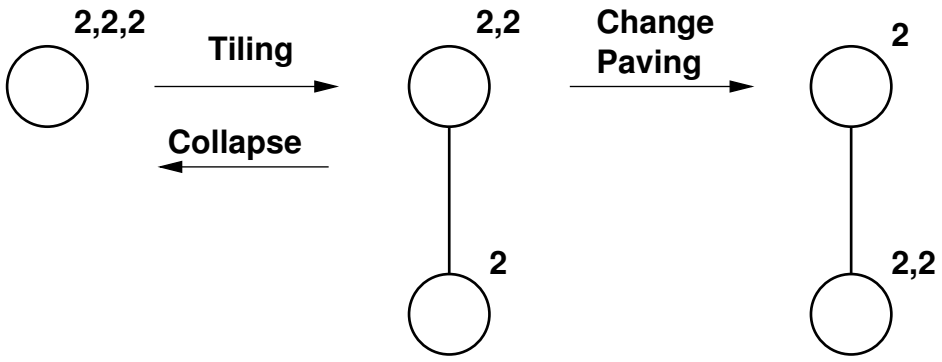


FIG. 6.1: Application des fonctions de refactoring ARRAY-OL *Tiling*, *Change Paving* et *Collapse* sur une application.

6.1.3 Bilan

Les fonctions de refactoring ARRAY-OL permettent de modification des applications Gaspard, nous avons étudié dans cette section l'impacte de ces modifications sur les accélérateurs générés par notre flot de conception. De manière générale, l'utilisation d'une fonction de refactoring modifie la structure de l'application et de l'accélérateur généré depuis cette application. La quantité de ressources consommées par l'accélérateur varie avec la fonction de refactoring utilisée et le choix dans l'exécution du parallélisme de données :

- Lorsque du parallélisme de l'application sur le temps est modifié en parallélisme sur l'espace (utilisation des fonctions de refactoring *Change Paving* et *Tiling*) et qu'une exécution parallèle est spécifiée, la quantité de ressources consommée par l'accélérateur et sa puissance de calcul augmentent ;
- lorsque le parallélisme de plus haut niveau de hiérarchie est modifié avec le parallélisme dans les niveaux de hiérarchie inférieurs et qu'une exécution séquentielle est spécifiée, la quantité de ressources consommée par l'accélérateur varie. La *descente* du parallélisme dans la hiérarchie (utilisation des fonctions de refactoring *Change Paving* et *Tiling*) permet d'augmenter la quantité de ressources et la puissance de calcul de l'accélérateur. La *montée* du parallélisme vers la tâche de plus haut niveau de hiérarchie (fonction de refactoring *Collapse*) permet de diminuer la quantité de ressources et la puissance de calcul de l'accélérateur.

Les possibilités pour modifier la quantité de ressources que consomme un accélérateur sont nombreuses. Il n'est par ailleurs pas possible d'assimiler une fonction de refactoring ARRAY-OL à un résultat sur l'accélérateur car il dépend aussi du choix de l'exécution du parallélisme de données qui est fait. Du point de vue de l'optimisation, cela n'est pas pratique car il faut gérer dans une heuristique les différentes combinaisons entre les fonctions de refactoring ARRAY-OL et ce choix d'exécution du parallélisme. Nous préférons « masquer » ces combinaisons au processus d'optimisation en proposant deux fonctions de refactoring dont l'impact souhaité est clairement défini : augmenter ou diminuer les ressources du FPGA nécessaires à l'implémentation d'un accélérateur. La section suivante présente ces deux fonctions de refactoring.

6.2 Les fonctions de refactoring PARALLÉLISATION et SÉQUENTIALISATION

Nous proposons dans cette section les fonctions de refactoring PARALLÉLISATION et SÉQUENTIALISATION des applications Gaspard. La fonction de refactoring PARALLÉLISATION tend à augmenter la quantité de ressources consommée par l'accélérateur, résultat inverse de la fonction de refactoring SÉQUENTIALISATION.

6.2.1 Parallélisation

Nous avons montré que l'utilisation de *change paving* ou de *Tiling* sur des tâches qui contiennent de la répétition sur le temps permet de générer des tâches hiérarchiques contenant du parallélisme sur l'espace. Du point de vue de l'exécution matérielle des applications Gaspard, nous obtenons une répétition de tâches dans l'espace à partir d'une répétition de tâches dans le temps ; l'accélérateur généré exécute plus de calculs en parallèle. Cette fonction de refactoring est donc similaire à un *unfolding* (fonctionnalité inverse du *folding* [9]) ou à la défactorisation [38]. Cependant, l'utilisation de cette fonction de refactoring ne permet un gain en puissance de calcul que lorsqu'elle est appliquée sur la dimension temporelle (que nous assimilons à la dimension infinie de l'espace de répétition). Dans la suite de nos travaux, nous nommons PARALLÉLISATION l'application du *Change Paving* dans ce cas précis.

En contrepartie d'un gain en puissance de calcul, plus de ressources de calcul sont nécessaires à la réalisation du circuit dont la répétition sur le temps est en partie transformée en parallélisme sur l'espace. Cette fonction de refactoring peut donc être exploitée dans une phase d'optimisation de l'accélérateur lorsqu'une plus grande puissance de calcul est nécessaire ou lorsque de nombreuses ressources de calcul restent disponibles. La PARALLÉLISATION permet d'augmenter la surface d'un accélérateur (la surface fait directement référence aux ressources de calcul exploitées par l'implémentation de l'accélérateur sur un FPGA). Elle est applicable tant que le support d'implémentation (FPGA) possède les ressources nécessaires à la mise en œuvre de l'accélérateur. Dans le cas contraire, c'est-à-dire lorsque l'accélérateur consomme plus de ressources que le FPGA n'en dispose, la PARALLÉLISATION n'est pas intéressante puisque c'est l'effet inverse (la réduction de la quantité de ressources) qui est souhaité. Nous présentons dans le paragraphe suivant un moyen de réduire le nombre de ressources du FPGA nécessaire à l'implémentation de l'accélérateur.

6.2.2 Séquentialisation

Le modèle d'exécution matérielle des applications Gaspard permet une exécution séquentielle du composant de plus haut niveau de hiérarchie ; pour cela, un même composant de calcul est réutilisé pour l'exécution de toutes les tâches de l'espace de répétition. L'exécution séquentielle réduit la quantité de ressources du FPGA nécessaires à l'implémentation d'un accélérateur et permet donc l'implémentation sur un FPGA de certains accélérateurs gourmands en ressources de calcul. Cependant, l'exécution séquentielle du parallélisme de données ne peut générer qu'un seul circuit car notre modèle d'exécution séquentielle ne supporte ni la hiérarchie ni le pipeline de tâches.

L'exécution séquentielle n'est possible que pour le composant répétitif de plus haut niveau de hiérarchie. La fonction de refactoring *Collapse* modifie la hiérarchie des applications

Gaspard en remontant par exemple du parallélisme de données dans le niveau de hiérarchie supérieur. Il en résulte un nouveau composant répétitif de plus haut niveau de hiérarchie. La séquentialisation des deux composants de plus haut niveau de hiérarchie ne produit pas les mêmes circuits puisque l'espace de répétition des tâches est différent. Nous couplons donc l'exécution séquentielle du parallélisme de données et la fonction de refactoring *Collapse* afin de procéder à une réduction de la quantité de ressources nécessaire à l'implémentation de l'accélérateur sur FPGA.

Nous appelons SÉQUENTIALISATION l'utilisation de la fonction de refactoring *Collapse* dans le but d'exécuter sur le temps (via l'exécution séquentielle) des tâches exécutées initialement sur l'espace. La fonction de refactoring SÉQUENTIALISATION est utilisée pour réduire la quantité de ressources du FPGA nécessaires à l'implémentation d'un accélérateur.

L'implémentation d'une exécution séquentielle nécessite la consommation d'un surplus de ressources de calcul pour l'implémentation du contrôleur, des multiplexeurs et des démultiplexeurs. Ce surcout dépend de la taille de l'espace de répétition, du nombre de motifs consommés par les tâches et de leur taille. Nous avons identifié les cas favorables à l'utilisation de la SÉQUENTIALISATION. Dans ces cas, le surcoût en ressource du FPGA lié à l'implémentation de l'exécution séquentielle est inférieur au coût d'implémentation d'une exécution parallèle car les composants de calcul sont plus gourmands en ressources du FPGA que les routages et le contrôleur de l'exécution séquentielle. Autrement dit, plusieurs unités de calcul nécessaires à une exécution parallèle consomment plus de ressources qu'une seule unité de calcul, le contrôleur, les multiplexeurs et les démultiplexeurs nécessaires à l'exécution séquentielle [2].

6.2.3 Bilan de fonctions de refactoring

Afin d'élargir les possibilités d'implémentation sur FPGA d'une même application, nous proposons d'utiliser les fonctions de refactoring ARRAY-OL *Change Paving* et *Collapse*. La fonction de refactoring *change paving*, appliquée sur la répétition temporelle, introduit une modification significative dans l'accélérateur produit et dans l'augmentation de sa puissance de calcul : la fonction de refactoring PARALLÉLISATION consiste à optimiser un accélérateur en vue d'une augmentation de sa puissance de calcul. La fonction de refactoring inverse, c'est-à-dire la réduction de la puissance de calcul, ou encore la réduction de la surface de FPGA nécessaire à l'implémentation de l'accélérateur, est réalisé par la fonction de refactoring SÉQUENTIALISATION qui est une combinaison de l'exécution séquentielle du parallélisme de données et de la fonction de refactoring *Collapse*. La figure 6.2 illustre les optimisations PARALLÉLISATION et SÉQUENTIALISATION. La gauche de cette figure représente quatre tâches exécutées sur l'espace et la droite de cette figure les quatre mêmes tâches exécutées sur le temps. Le passage d'une exécution à l'autre est possible par l'intermédiaire des fonctions de refactoring PARALLÉLISATION et SÉQUENTIALISATION.

L'utilisation des fonctions de refactoring PARALLÉLISATION et SÉQUENTIALISATION permet de modifier les accélérateurs générés dans le modèle RTL. Afin d'être exploitées dans un processus d'optimisation, les caractéristiques d'implémentation sur FPGA des accélérateurs doivent être connues. Ces caractéristiques (ressources de calcul du FPGA notamment) sont disponibles dans le métamodèle RTL au travers des concepts d'implémentation.

[2] Les cas non favorables correspondent aux applications Gaspard qui contiennent des unités de calcul peu consommatrices en ressources du FPGA et un grand espace de répétition.

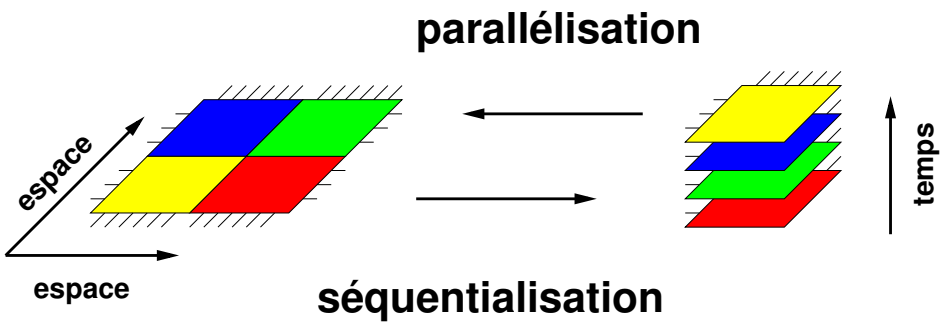


FIG. 6.2: Les fonctions de refactoring pour la génération des accélérateurs : la PARALLÉLISATION permet l'exécution sur l'espace du parallélisme sur le temps, la SÉQUENTIALISATION permet l'exécution sur le temps (via l'exécution séquentielle) du parallélisme sur l'espace.

6.3 Processus d'optimisation

Nous introduisons dans cette section notre processus d'optimisation qui fait appel aux fonctions de refactoring PARALLÉLISATION et SÉQUENTIALISATION. L'intégration de ce processus dans notre flot de conception dirigé par l'IDM est présentée en fin de section.

6.3.1 Heuristique d'optimisation des accélérateurs

Nous définissons l'optimisation de l'implémentation d'un accélérateur sur un FPGA comme l'ajustement de la surface d'implémentation de cet accélérateur avec sa surface allouée sur FPGA. Pour cela, nous utilisons les fonctions PARALLÉLISATION et SÉQUENTIALISATION que nous associons (du point de vue du processus d'optimisation) à l'augmentation de la surface de l'accélérateur et de sa puissance de calcul pour la PARALLÉLISATION et à la réduction de la surface et de la puissance de calcul pour la fonction SÉQUENTIALISATION. Nous considérons donc que ces fonctions sont appliquées sur les composants de plus haut niveau de hiérarchie dans le modèle d'application Gaspard, qu'elles lancent la transformation de modèle Deployed2RTL présentée dans le chapitre 5 et qu'elle retourne le nouveau modèle RTL généré.

Par ailleurs, nous travaillons sur des applications dont le composant de plus haut niveau de hiérarchie exprime du parallélisme de données et non de tâche. Cela garantit qu'une exécution séquentielle introduite par la fonction SÉQUENTIALISATION ne soit appliquée que sur un composant car le modèle d'exécution matérielle ne gère pas le parallélisme de tâches exécutées séquentiellement. Pour cela, nous utilisons la fonction de refactoring *fusion*, clairement décrite par Dumont [39], qui permet de ramener toute application Gaspard à une structure telle que celles présentées à la figure 6.2.

Le cœur de notre heuristique est assimilé à la fonction `Optimisation` qui prend en paramètres le modèle d'accélérateur généré dans l'itération en cours `acc`[3] et le modèle de FPGA sur lequel est implémenté l'accélérateur. Cette fonction retourne un modèle d'accélé-

[3]Cette fonction est appelée après une première passe dans le processus d'optimisation et d'une première utilisation des fonctions SÉQUENTIALISATION ou PARALLÉLISATION. Nous ne détaillons pas cette première passe.

rateur qui est valide ou non selon le résultat de l'heuristique. Le code suivant correspond à notre heuristique d'optimisation :

```
RTL_Modele Optimisation(RTL_Modele acc, RTL_FPGA fpga)
{

RTL_Modele final = acc
RTL_Modele pre-acc = acc
Booleen fin_opt = faux

Faire
{
   Si acc.ressources > fpga.ressources Alors
      Si pre-acc.ressources < fpga.ressources Alors
         final = pre-acc // cas a
         fin_opt = vrai
      Sinon
         Si pre-acc.resources <= acc.ressources Alors
            final = pre-acc // cas b
            fin_opt = vrai
         Sinon
            pre-acc = acc
            acc = sequentialisation(acc)
         FinSi
      FinSi
   Sinon
      Si pre-acc.resources > fpga.ressources Alors
         final = acc // cas c
         fin_opt = vrai
      Sinon
         Si pre-acc.resources >= acc.ressources Alors
            final = pre-acc // cas d
            fin_opt = vrai
         Sinon
            pre-acc = acc
            acc = parallelisation(acc)
         FinSi
      FinSi
   FinSi
} Tant_que fin_opt = faux

Retourne final
}
```

La base de cette heuristique est la comparaison des ressources consommées par l'accélérateur (ACC.RESSOURCES dans le code) avec celles disponibles sur le FPGA (FPGA.RESSOURCES). Nous ne comparons que les ressources reconfigurables, les CLBs, car le processus d'estimation utilise dans un premier temps les ressources dédiées du FPGA (blocs DSP, mémoires, etc.) puis, lorsqu'il n'en reste plus, les ressources configurables qui permettent d'implémenter tous les types de ressources. Il n'est donc pas possible de consommer plus de ressources dédiées que n'en possède le FPGA, la limitation viendra donc des CLBs. Par ailleurs, cette heuristique conserve en mémoire l'accélérateur généré à l'itération

précédente, PRE-ACC, et compare les ressources qu'il nécessite avec celle de l'accélérateur en cours ACC.

Lorsque le processus d'optimisation re-itère, le modèle d'application initial est modifié via les fonctions de refactoring SÉQUENTIALISATION ou PARALLÉLISATION. L'utilisation de ces fonctions de refactoring est conditionnée par l'évaluation des ressources consommées par l'accélérateur avec celles qui lui sont allouées sur FPGA : s'il reste des ressources sur les FPGAs, une PARALLÉLISATION est déclenchée, une SÉQUENTIALISATION est déclenchée dans le cas contraire. Avant de procéder à ces fonctions de refactoring, l'accélérateur est conservé sous la forme PRE-ACC.

Le processus d'optimisation peut être stoppé de quatre façons différentes selon notre heuristique :

- cas a : la quantité de ressources de l'accélérateur dépasse celles des ressources disponibles sur FPGA suite à une PARALLÉLISATION (l'accélérateur est implémentable sur FPGA et l'accélérateur généré à l'itération précédente est validé), ce qui engendre VALIDE(PRE-ACC)
- cas b : deux SÉQUENTIALISATIONS successives produisent le même résultat en terme de ressources consommées (l'accélérateur n'est pas implémentable sur le FPGA), cela engendre ERREUR(IMPOSSIBLE DE RÉDUIRE)
- cas c : la quantité de ressources nécessaires à l'accélérateur devient inférieure à celle des ressources disponibles sur FPGA suite à une SÉQUENTIALISATION (L'accélérateur généré dans l'itération est implémentable), cela correspond dans l'algorithme à VALIDE(ACC).
- cas d : deux PARALLÉLISATIONS successives génèrent des accélérateurs dont les quantités de ressources nécessaires sont identiques ou diminuent (l'accélérateur est implémentable sur FPGA mais des ressources restent disponibles), cela engendre VALIDE(PRE-ACC) et correspond à une incapacité de PARALLÉLISATION pour augmenter la puissance de calcul de l'accélérateur

Notre heuristique est relativement simple puisqu'elle n'est pas multicritère et qu'elle ne cherche des solutions optimales que du point de vue des ressources exploitées sur FPGA. Ainsi, aucun critère sur la puissance de calcul, le débit, les latences ou la fréquence ne sont utilisés. L'utilisation de ces autres caractéristiques reste envisageable, mais nécessite alors une profonde modification de notre heuristique pour la gestion du multicritère. Par ailleurs, il sera nécessaire d'évaluer l'impact des fonctions de refactoring ARRAY-OL sur chacun de ces critères afin de guider le refactoring de l'application.

6.3.2 Processus d'optimisation dans notre flot de conception IDM

Le processus d'optimisation que nous proposons est utilisé dans un flot de conception entièrement guidé par l'IDM. Ce processus d'optimisation a les particularités suivantes :

- le point de départ de l'heuristique est l'évaluation des ressources du FPGA nécessaires à l'implémentation de l'accélérateur. L'évaluation est réalisée au niveau du métamodèle RTL car il permet l'utilisation des concepts d'implémentation sur FPGA et que le niveau de description est plus précis que dans les autres métamodèles dans la chaîne de transformation, ce qui permet d'augmenter la précision de l'évaluation ;
- l'heuristique exploite les fonctions PARALLÉLISATION et SÉQUENTIALISATION qui sont appliquées au niveau du métamodèle Deployed. En effet, les fonctions de refactoring ARRAY-OL nécessitent, entre autres, que les tilers soient exprimés sous leur

forme « origine, pavage et ajustage ». Ainsi, les fonctions de refactoring ARRAY-OL ne peuvent être appliquées sur un modèle RTL car les tilers sont précalculés ;
– chaque itération du processus d'optimisation qui implique une modification du modèle d'application en entrée nécessite une nouvelle compilation vers un modèle RTL. Afin d'estimer les ressources du FPGA nécessaires à l'implémentation de cet accélérateur, les concepts d'implémentation sont créés dans le modèle RTL. L'itération se termine et le résultat de l'estimation introduit une nouvelle optimisation ou termine le processus.

La figure 6.3 illustre l'articulation entre le processus d'optimisation et notre flot de conception guidé par l'IDM. Le haut de la figure représente un modèle d'application Gaspard, transformé en modèle RTL à l'aide de la transformation de modèles présentée dans le chapitre 5. Les ressources consommées par l'accélérateur sont modélisées sous la forme de concepts COMPONENTFPGAIMPLEMENTATION et sont comparées aux ressources de l'accélérateur allouées sur FPGA. Notre heuristique d'optimisation guide alors le processus d'optimisation complet.

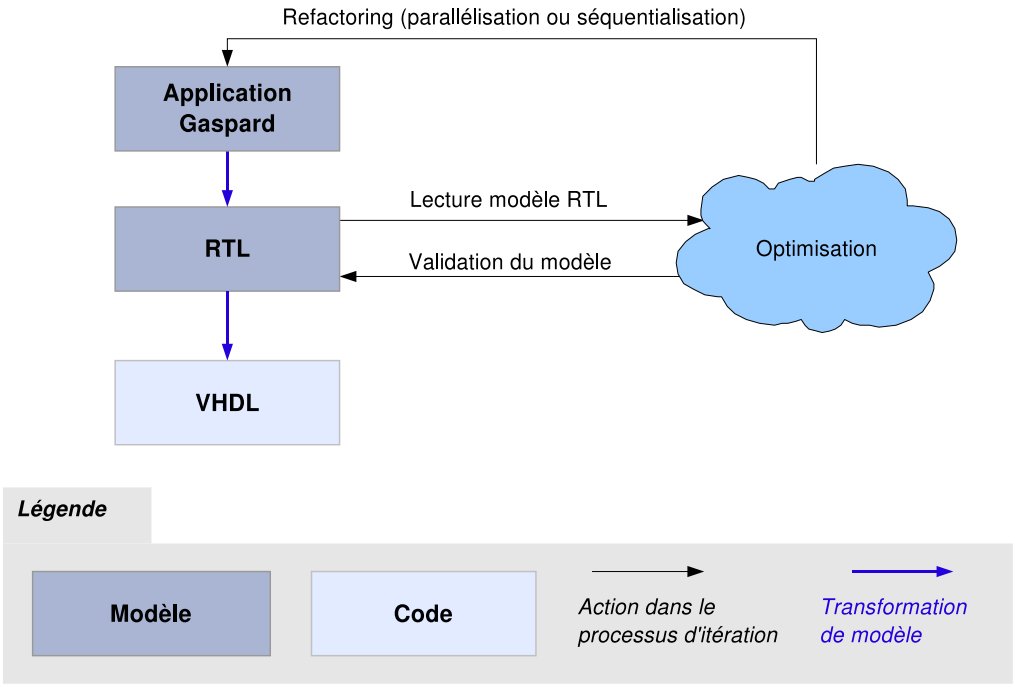


FIG. 6.3: Articulation du processus d'optimisation avec notre flot de conception IDM.

Le processus d'optimisation est confronté à plusieurs difficultés pour sa mise en œuvre et son automatisation :
– la création des concepts d'implémentation (concept INSTANCEFPGAIMPLEMENTATION du métamodèle RTL) pour permettre l'estimation des ressources du FPGA nécessaires à l'implémentation de l'accélérateur ;
– l'utilisation des fonctions de refactoring ARRAY-OL implémentées implémentées sous

la forme d'une boîte à outils [39] et dont il convient de paramétrer correctement les méthodes d'accès ;
– l'utilisation des fonctions de refactoring PARALLÉLISATION et SÉQUENTIALISATION sur des modèles Gaspard alors que les estimations sont réalisées sur des modèles RTL.

Le processus d'optimisation que nous présentons n'est pas entièrement automatisé du fait de ces différents détails techniques. Toutefois, et afin de valider ce processus, nous créons les concepts d'implémentation sur FPGA et l'utilisation des fonctions de refactoring PARALLÉLISATION et SÉQUENTIALISATION manuellement sur les modèles d'application en entrée de notre flot de conception. Nous illustrons dans la section suivante les résultats de l'utilisation de notre flot de conception sur l'application du filtre d'images.

6.3.3 Bilan du processus d'optimisation

Dans cette section, nous avons présenté un processus d'optimisation qui permet de générer différents accélérateurs (modèles RTL) à partir d'une même application (modèle Gaspard). La génération de différents accélérateurs est réalisée au travers de l'utilisation des fonctions de refactoring ARRAY-OL et des exécutions parallèles et séquentielles du parallélisme de données dans le modèle d'exécution matérielle des applications Gaspard. La fonction de refactoring PARALLÉLISATION augmente la puissance de calcul d'un accélérateur et les ressources du FPGA nécessaires tandis que la fonction SÉQUENTIALISATION réduit sa puissance de calcul et réduit son coût d'implémentation. Les fonctions de refactoring PARALLÉLISATION et SÉQUENTIALISATION permettent l'optimisation des accélérateurs en fonction de leur cible d'implémentation FPGA et de ses ressources.

Notre processus d'optimisation génère l'accélérateur le plus puissant possible sur la surface du FPGA alloué à cet effet car nous ne prenons pas en compte les puissances de calcul requises par les applications Gaspard. Une évolution de notre heuristique d'optimisation dans ce sens permettra de guider plus efficacement le processus d'optimisation afin notamment de le stopper lorsque la puissance de calcul est atteinte ou lorsqu'elle ne l'est plus.

6.4 Illustration de l'optimisation d'un accélérateur

Nous illustrons l'utilisation du processus d'optimisation au travers du filtre d'images présenté au chapitre précédent. Afin d'illustrer plus particulièrement l'utilisation des fonctions PARALLÉLISATION et SÉQUENTIALISATION, nous générons des accélérateurs pour le filtrage de deux images de dimensions différentes. Nous reprenons l'exemple des images composées de 4×4 pixels dans la section 6.4.1 puis à des images composées de 34×34 pixels dans la section 6.4.2. Par ailleurs, nous ne considérons plus le traitement d'une seule image, mais le traitement d'un flux d'images. Le traitement de ce flux implique une répétition temporelle de l'application entière sur le temps. Les accélérateurs issus de ces deux exemples sont implémentés sur des FPGA Stratix2s60.

6.4.1 Optimisation à l'aide de la fonction PARALLÉLISATION

Nous reprenons donc l'exemple académique du filtre d'images composées de 4×4 pixels codés en niveau de gris (8 bits par pixel). La modélisation en UML de cette application est illustrée la gauche de la figure 6.4. La transformation du modèle Deployed initiale en modèle

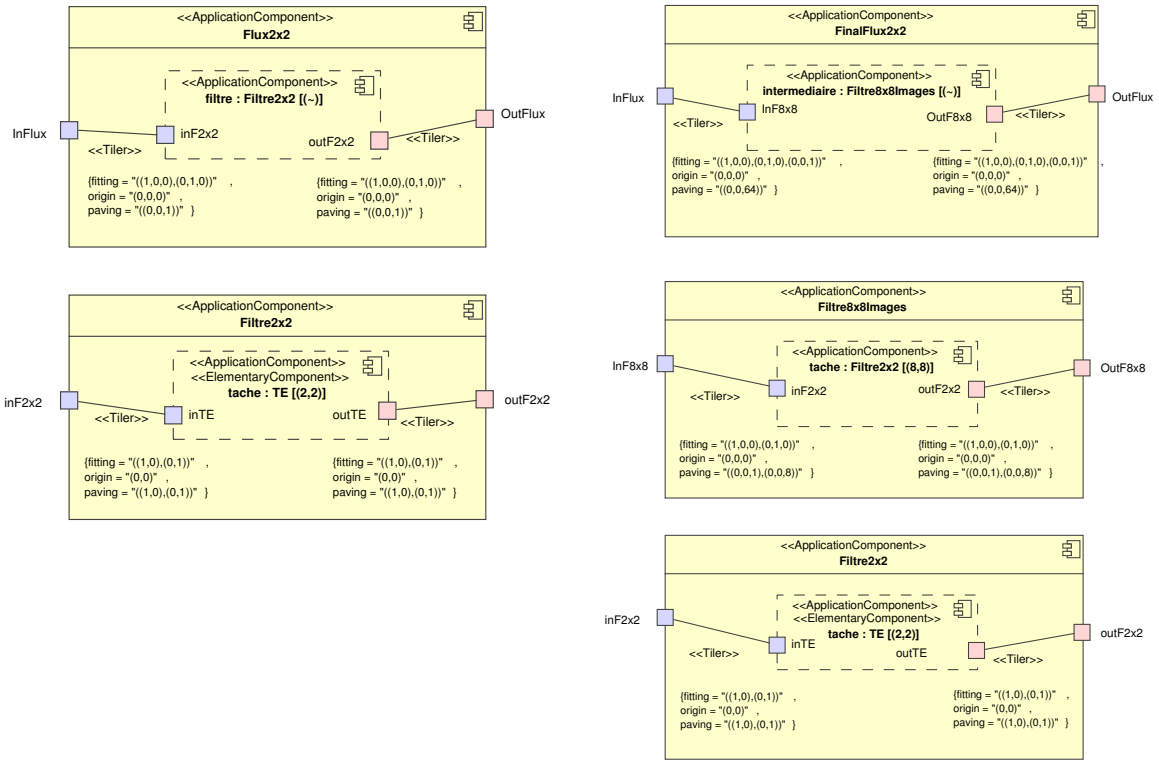


FIG. 6.4: La modélisation initiale de l'application à gauche, la modélisation de la même application à l'issu de l'optimisation à droite.

RTL permet la génération d'un modèle d'accélérateur matériel. L'estimation que nous réalisons au niveau du métamodèle RTL indique que 1.2% des ressources CLB du FPGA sont nécessaires à l'implémentation de l'accélérateur.

Le processus d'optimisation modifie le modèle d'application initiale par le biais de la fonction de refactoring PARALLÉLISATION. La transformation de modèle génère un nouvel accélérateur dont l'estimation indique une augmentation des ressources consommées, sans toutefois dépasser celles du FPGA. Plusieurs PARALLÉLISATIONS de l'application sont alors lancées, jusqu'à l'itération à laquelle les ressources consommées par l'accélérateur dépassent celles du FPGA. Le processus est donc stoppé, et l'accélérateur généré à l'itération précédente est validé. La modélisation UML de l'application qui a permis la génération de l'accélérateur est représentée sur la droite de la figure 6.4.

La dimension des images traitées par cette application a pour conséquence directe que l'accélérateur qui permet l'exécution du filtre ne consomme que très peu de ressources du FPGA. Le processus d'optimisation, par le biais de la fonction de refactoring PARALLÉLISATION permet l'exécution sur l'espace des tâches initialement exécutées sur le temps. Finalement, l'accélérateur généré est en mesure de filtrer 64 images images (8×8) en parallèle sur le FPGA en utilisant environ 80 % de ses CLBs.

6.4.2 Optimisation à l'aide de la fonction SÉQUENTIALISATION

Dans cette section, nous reprenons l'exemple du filtre d'images mais en augmentant la taille des images traitées (34×34 pixels afin de produire 32×32 pixels à la sortie du filtre). La gauche de la figure 6.5 représente la modélisation UML initiale de cette application. L'estimation de la quantité de ressources consommée par l'accélérateur généré depuis ce modèle indique que 300 % des ressources CLB du FPGA sont nécessaires.

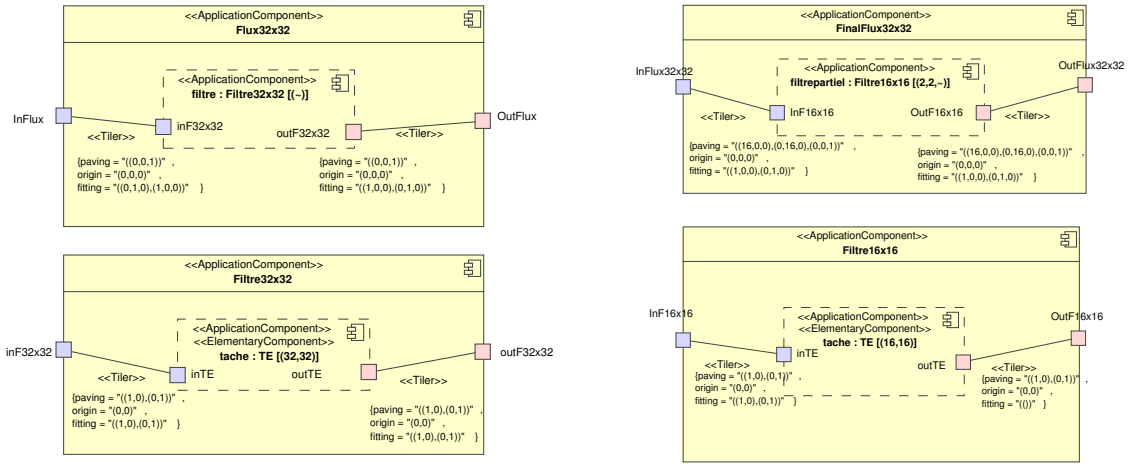


FIG. 6.5: La gauche de cette figure illustre la modélisation initiale de l'application, la droite représente le modèle à l'issue de l'optimisation.

Afin de réduire la quantité de ressources du FPGA nécessaires à l'implémentation de l'accélérateur, le processus d'optimisation lance une succession d'appel à la fonction de refactoring SÉQUENTIALISATION sur le modèle d'application initiale. Il en résulte une diminution des ressources consommées par l'accélérateur. La dernière itération de l'optimisation correspond au moment où les ressources CLBs de l'accélérateur n'exploitent pas toutes celles disponibles sur le FPGA. Le processus d'itération est donc stoppé et l'accélérateur généré à cette itération est validé. La modélisation UML de l'accélérateur généré par cette dernière itération est illustrée à droite de la figure 6.5, elle montre qu'aucun niveau de hiérarchie n'est créé mais que le parallélisme initialement spécifié en bas de la hiérarchie (c'est-à-dire $[(32, 32)]$) est partiellement remonté au même niveau que le la répétition sur le temps.

Le résultat est une répétition de $[(2, 2, \tilde{}]$ autour d'une instance de composant qui est exécutée en séquentiel. Le composant de hiérarchie inférieure exécute 16×16 tâches en parallèle. Le résultat de synthèse ce cet accélérateur illustré à la figure 6.6 représente le composant de plus haut niveau de hiérarchie (*FinaFlfux32x32* dans s le modèle UML) qui exécute séquentiellement les $2 \times 2 \times \tilde{}$ répétitions l'instance de composant *filtrepartiel* (marque **3**) par le biais d'un contrôleur (marque **7**), d'un multiplexeur (marque **6**) et d'un demultiplexeur (marque **8**). Les marques **1**, **2**,**4** et **5** représentent respectivement le port d'entrée des images, le tiler d'entrée, le tiler de sortie et le port de sortie des images.

Le circuit généré à partir de la spécification initiale de l'application n'est pas implémentable car trop gourmand en ressources de calcul. La succession des appels à la fonction de refactoring SÉQUENTIALISATION permet de réduire la quantité de ces ressources, cela en

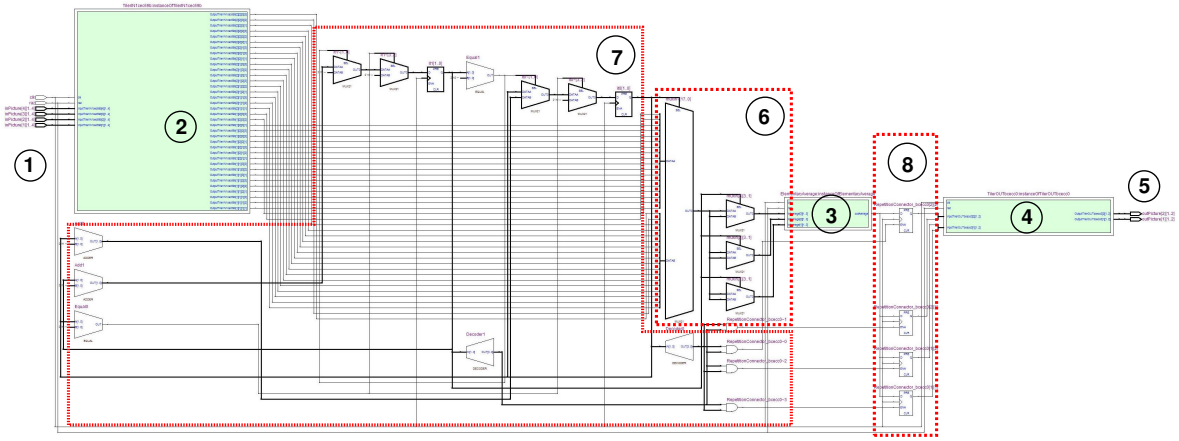


FIG. 6.6: Schéma de synthèse du circuit issu du composant *FinalFlux32x32* dans le modèle UML. Nous retrouvons les concepts liés à l'exécution séquentielle du parallélisme de données dans notre modèle d'exécution.

contrepartie d'une diminution de la puissance de calcul. Au final, 4 cycles d'horloge sont nécessaires pour le filtrage d'une image.

La dimension des images traitées dans cet exemple ne permet pas de passer à l'échelle, il est cependant possible d'appliquer le même procédé pour des images plus conséquentes.

6.4.3 Bilan des études de cas

L'utilisation du processus d'optimisation pour l'application du filtre d'images montre que nous sommes en mesure d'ajuster la quantité de ressources nécessaires à l'implémentation d'un accélérateur en fonction de sa cible FPGA. Cet ajustement se traduit par une augmentation ou une diminution de la puissance de calcul de l'accélérateur qui n'est cependant pas maîtrisée, car non prise en compte dans l'optimisation. Il peut alors en résulter un circuit dimensionné au FPGA sur lequel est implémenté l'accélérateur, mais dont la puissance de calcul est disproportionnée à celle requise par l'application.

6.5 Conclusion

Ce chapitre montre l'utilisation couplée des fonctions de refactoring ARRAY-OL *Change Paving* et *Collapse* avec les exécutions parallèles et séquentielles du parallélisme de données dans le modèle d'exécution matérielle. Il en résulte les fonctions de refactoring SÉQUENTIALISATION et PARALLÉLISATION qui modifient la puissance de calcul et les ressources FPGA consommées des accélérateurs. Ces fonctions de refactoring sont appliquées sur des modèles Deployed au travers d'un processus d'optimisation qui tend à exploiter le maximum de ressources du FPGA alloué pour l'implémentation d'un accélérateur. Des évolutions de notre heuristique vers le multicritère permettraient de stopper le processus d'optimisation plus rapidement, par exemple lorsque la puissance de calcul de l'accélérateur est suffisante. Ces extensions requièrent toutefois des extensions de l'environnement Gaspard afin de prendre en compte notamment les contraintes temps réels.

Chapitre 7

Utilisation de notre flot pour la conception d'un système radar anti-collision

Contents

Le secteur du transport est sujet à de nombreuses innovations technologiques. Dans le cadre plus précis de l'automobile, les conducteurs sont de plus en plus aidés et appuyés par des systèmes électroniques de sécurité dits « intelligents », comme les régulateurs de vitesse avec GPS [67], les radars anticollision ou encore un système couplant les deux [72, 32]. Les algorithmes qui composent les radars anti-collision relèvent du traitement du signal intensif car ils requièrent un grand nombre de calculs réguliers sur un nombre important de données. La puissance de calcul nécessaire à l'exécution de ces algorithmes est donc considérable et il est courant d'implémenter ces algorithmes sous la forme d'accélérateurs matériels [117, 56]. Ces accélérateurs sont souvent créés de façon ad hoc.

Nous pensons que notre flot de conception est en mesure de générer automatiquement de tels circuits à partir de descriptions à haut niveau d'abstraction. Nous nous intéressons plus particulièrement aux algorithmes de détection d'obstacles utilisés dans les radars anti-collision. La section 7.1 présente en détail un système anti-collision, son contexte d'utilisation et les algorithmes de détection d'obstacles utilisés. La section 7.2 présente la modélisation à haut niveau d'abstraction de cette application et la section 7.3 présente les performances du circuit généré lors de son implémentation sur FPGA. La dernière section conclut ce chapitre.

7.1 Système de détection d'obstacles

Le système de détection d'obstacles auquel nous nous intéressons est composé d'une antenne et d'un FPGA. L'antenne émet une onde dont l'écho lui est retourné lorsque l'onde émise rencontre un obstacle. Des informations sur la distance de cet obstacle sont contenues dans l'onde reçue (l'écho), mais ne sont pas directement lisibles car réparties sur le temps et noyées dans le bruit. Afin de retrouver l'information intéressante, l'onde radar est traitée par le FPGA contenu dans le système embarqué. Nous étudions en détail le traitement de cette onde par le biais d'un algorithme de détection, nous présentons toutefois son contexte d'utilisation dans la section suivante.

7.1.1 Système radar-FPGA

L'ensemble du système est représenté à la figure 7.1. L'envoi de l'onde par le radar est réalisé au travers de la génération d'un code pseudo aléatoire composé de 1023 bits. La composition de ce code est différente pour chaque véhicule équipé de ce système, ce qui permet d'éviter les conflits entre les véhicules. Ce code numérique est converti en signal analogique par le biais d'un Convertisseur Numérique Analogique (CNA) et vient ensuite moduler un signal porteur de fréquence 76-77 GHz. Le signal résultant de cette modulation est transformé en onde par l'antenne qui la diffuse dans l'environnement du véhicule.

Dans le cadre du radar anti-collision, cette onde est émise vers l'avant du véhicule afin de détecter de potentiels obstacles. Lorsque l'onde percute un obstacle, elle est reflétée, créant ainsi un écho radar. Lorsque cet écho est reçu par l'antenne, celle-ci le convertit en signal qui est renvoyé au système embarqué. En effet, le circulateur situé en amont de l'antenne permet de router les signaux entrant et sortant de l'antenne, ce qui permet à l'antenne d'émettre et de recevoir des ondes en continu (dans un système plus simple, un commutateur permet d'alterner l'émission et la réception des ondes). Les ondes émises sont celles issues de la modulation du code de référence et les ondes reçues sont celles qui permettent de détecter les obstacles. Les ondes reçues par le radar sont tout d'abord démodulées et le signal analogique qui en découle est numérisé par le biais d'un Convertisseur Analogique Numérique (CAN).

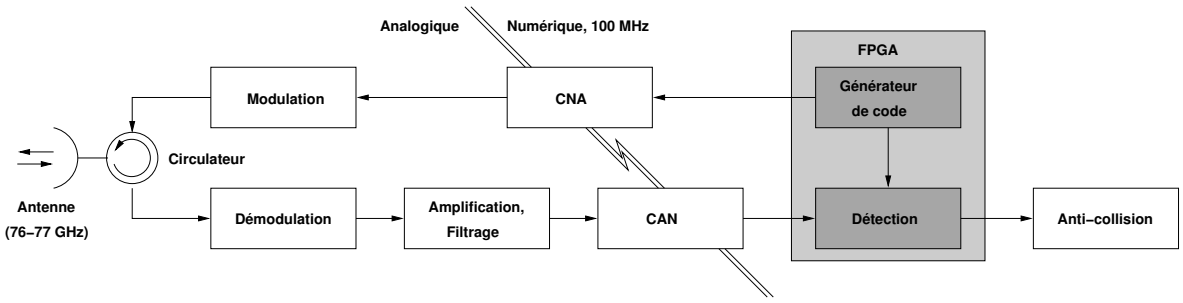


FIG. 7.1: Le système embarqué composé notamment de l'antenne et du FPGA.

Ce signal numérique contient des informations sur la présence de potentiels obstacles mais est aussi bruité. Aussi, il n'est pas nécessaire d'exploiter toute la précision du CAN dans la mesure où les informations contenues dans les bits de poids faibles sont noyées dans le bruit. El Hillali [56] préconise de n'utiliser que les 4 bits de poids fort du CAN pour un tel système car les informations contenues dans le 5^e bit et les suivants sont mineures. Ce signal numérique est alors envoyé au FPGA de façon continue sous la forme d'échantillons codés sur 4 bits et à la fréquence de 100 MHz.

Le FPGA permet l'implémentation d'un algorithme de détection qui isole les informations intéressantes contenues dans le signal reçu. Ces informations correspondent au code de référence utilisé pour l'émission de l'onde et qui est donc particulier à chaque véhicule. Le rôle de l'algorithme de détection est de mettre en évidence les similitudes entre le code de référence et le signal reçu : une forte ressemblance est assimilée à la présence du code de référence dans le signal reçu et donc à la présence d'un obstacle, l'inverse signifie que le signal reçu ne contient pas ou peu d'informations sur le code de référence et que, par conséquent, aucun obstacle n'est détecté.

La présence d'un obstacle génère un *pic de détection* en sortie de l'algorithme car les informations contenues dans le signal reçu correspondent au code de référence. Le délai entre la présence d'un pic et le moment où l'onde a été émise permet de retrouver la distance avec l'obstacle qui est la cause du pic ; cela est réalisé par l'équation $D = d.\frac{c}{2f}$ où D est la distance, d le délai mesuré, c la vitesse de propagation de l'onde radar ($3.10^8\ m/s$) et f la fréquence de l'onde porteuse du système. Les informations de détection d'obstacles et de distances sont alors transmises à la dernière partie du système qui est en mesure de reconstituer l'environnement de la voiture et d'anticiper les collisions. Cette dernière partie nécessite des calculs plus irréguliers et n'est pas adaptée à une exécution matérielle, nous ne la détaillerons donc pas.

Nous venons de présenter le système de détection d'obstacles et constatons que de nombreux éléments sont mis en œuvre pour sa réalisation. Dans ce genre de système, la performance de la détection se traduit, entre autres, par la distance maximale des obstacles que le système est capable de détecter et de mesurer. La qualité de la détection dépend de chaque élément du système et par conséquent de l'algorithme de détection utilisé : plus l'algorithme est puissant et plus il est possible de faire ressortir de l'information intéressante du signal numérique bruité. Il existe différents algorithmes de détection dans la littérature, nous présentons dans la section suivante deux de ces algorithmes.

7.1.2 Algorithmes de détection

Corrélation Le premier algorithme de détection est appelé corrélation en traitement du signal, son expression mathématique est la suivante :

$$C_{cy}(j) = \frac{1}{N} \sum_{i=0}^{N-1} c(i) \cdot y(i+j) \tag{7.1}$$

où $C_{cy}(j)$ correspond au résultat de l'algorithme, $c(i)$ au code de référence, $y(i+j)$ au signal reçu par le système embarqué et N à la longueur du code de référence. Dans la mesure où nous nous intéressons à la corrélation d'un signal avec un code de référence composé de 1023 échantillons et que la normalisation $\frac{1}{1023}$ n'améliore pas la qualité de la détection elle-même, nous considérons l'expression suivante :

$$C_{cy}(j) = \sum_{i=0}^{1022} c(i) \cdot y(i+j) \tag{7.2}$$

Cet algorithme de corrélation est aussi appelé filtre FIR, pour Finite Impulse Response (filtre à Réponse à Impulsionnelle Finie), et a déjà fait l'objet de nombreuses implémentations et optimisations dans la littérature [117]. L'utilisation de ce filtre nous permettra de nous positionner par rapport à ces approches.

Algorithme basé sur les statistiques d'ordre supérieur Le second algorithme que nous considérons est basé sur les statistiques d'ordre supérieur (Higher Order Statitics (HOS) en Anglais), et plus particulièrement sur un algorithme de troisième ordre proposé par Tugnait [120, 121]. Cet algorithme est amélioré par Zaidouni *et al.* [130] afin d'améliorer la portée de la détection d'obstacles pour le système embarqué qui nous intéresse. L'algorithme $J_{toce}(io)$ proposé par Zaidouni correspond aux équations suivantes :

$$J_{toce}(i_0) = \sum_{i=0}^{N-1} C_{ycc}(j) C_{ycy}(j+i_0) \tag{7.3}$$

$$C_{lmn}(j) = \sum_{i=0}^{N-1} l(i) m(i+n) n(i+j) \tag{7.4}$$

Ces équations permettent de constater que cet algorithme est composé de trois corrélations : les deux équations 7.4, consistent à réaliser des corrélations entre le code de référence et le signal reçu. La troisième corrélation, équation 7.3, est réalisée avec les résultats des deux précédentes corrélations.

Comparaison des deux algorithmes de détection L'algorithme $J_{toce}(io)$ nécessite plus de calculs que l'algorithme de la corrélation, mais permet d'améliorer dans certains cas la qualité de la détection pour l'antenne utilisée dans le système qui nous intéresse [130]. Les différences de performance entre les deux algorithmes dépendent principalement du ratio signal à bruit (Signal to Noise Ratio, SNR) du signal reçu. L'utilisation couplée de ces deux algorithmes pour la détection est alors intéressante puisque chacun d'eux est performant pour une plage de SNR. Nous proposons dans la section suivante de coupler ces algorithmes et d'améliorer ainsi les performances de détection de notre système embarqué.

7.1.3 Utilisation couplée de deux algorithmes

L'utilisation des deux algorithmes génère un gain de performance substantiel du point de vue de la détection car les algorithmes sont complémentaires. Nous exploitons par conséquent ces deux algorithmes en parallèle pour la détection d'obstacles. Le traitement de données issues de ces algorithmes est réalisé lors d'une étape visant à prévenir les collisions. Cette dernière étape est plus irrégulière et n'est, par conséquent, pas adaptée à notre environnement de modélisation.

7.1.4 Bilan

Le système embarqué qui réalise un radar anti-collision contient, entre autres, une étape de détection d'obstacles. La détection d'obstacles consiste à mesurer les similitudes entre un code de référence et un signal reçu au travers d'un algorithme de détection. La qualité de la détection dépend en partie de l'algorithme utilisé. Nous avons présenté deux algorithmes bien différents qui réalisent tout deux la fonctionnalité de détection mais avec des performances qui diffèrent selon le niveau de bruit dans le signal reçu. Afin de tirer profit des complémentarités de ces algorithmes, nous les utilisons en parallèle sur un FPGA. Nous présentons la modélisation UML de ce système dans la section suivante.

7.2 Modélisation UML du système de détection

Cette section présente les modélisations UML du système implémenté sur FPGA. Nous présentons tout d'abord en détail le modèle UML de la corrélation, puis nous abordons celle de l'algorithme $J_{toce}(io)$ et celle du contrôleur. Nous terminons la présentation du modèle UML par une vue d'ensemble du système.

7.2.1 Modélisation de l'algorithme de corrélation

Nous décrivons l'algorithme de corrélation du haut de la hiérarchie vers le bas.

7.2.1.1 Vue de haut niveau

Pour produire un résultat, la corrélation nécessite 1023 multiplications entre les 1023 éléments du code de référence et les 1023 derniers échantillons reçus. Les résultats de ces 1023 multiplications sont ensuite additionnés les uns aux autres : la somme de ces 1023 données produit un résultat de la corrélation. Ces deux grandes étapes dans le calcul de la corrélation apparaissent sur la figure 7.2 : l'instance *trm* du composant *TimeRepeatedMultiplication* réalise les multiplications et celle (*trat*) du composant *TimeRepeatedAdditionTree*, la somme.

La corrélation possède deux ports d'entrée, l'un (*ReceivedSignal*) pour le signal reçu et l'autre (*Coeff*) le code de référence. Le signal reçu, le port *ReceivedSignal*, est un flux infini de données, nous notons par conséquent $[(\tilde{})]$ l'unique dimension du port *ReceivedSignal*. Seuls les quatre bits de poids fort du signal d'entrée sont retenus, nous utilisons donc le type INTEGER RANGE -8 TO 7 pour ce signal. Le code de référence est initialement composé de 1023 échantillons dans le système. Afin de pouvoir standardiser ce port d'entrée en puissance de 2, nous ajoutons un 1024^e élément dans le code de référence dont la valeur est neutre et n'affecte pas le résultat du calcul. Le code de référence prend les valeurs -1, 0 et 1 (le 0 permet d'encoder la valeur ajoutée. Le type primitif du port *Coeff* est donc INTEGER RANGE -1 TO

1 et sa dimension est notée $[(1024, \tilde{})]$. Le choix de modéliser le code de référence sous la forme d'un flux sur le temps (par le biais de $\tilde{}$ dans sa dimension) permet de modifier ce code durant l'exécution de l'algorithme. Cette fonctionnalité ne sera cependant pas traitée par la suite.

La sortie de la corrélation, (port *outCorr*), est aussi un flux infini et sa dimension est donc notée $[(\tilde{})]$. Par ailleurs, l'algorithme nous permet de spécifier que la valeur maximale de la sortie (le port *SumResult* de l'instance *trat*) sera comprise entre -8192 et 8191, le type primitif associé est donc INTEGER RANGE -8192 TO 8191. Afin que la sortie du système soit plus standard, nous instancions le composant *ConvInteger* qui convertit le type en sortie de l'algorithme en INTEGER.

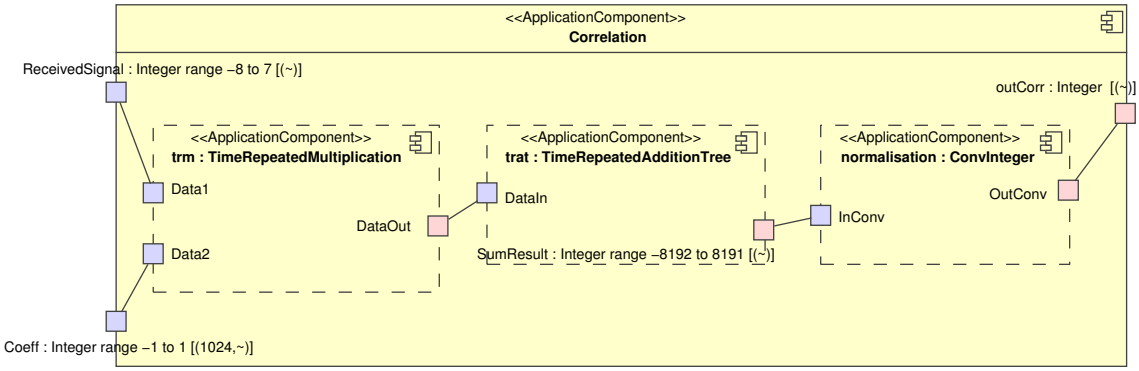


FIG. 7.2: Vue de plus haut niveau de la corrélation : le composant de gauche exprime les multiplications, le composant de droite la somme des résultats des multiplications.

7.2.1.2 Modélisation des multiplications

La modélisation UML des multiplications de l'algorithme est illustrée à la figure 7.3 au travers de deux composants : le composant *TimeRepeatedMultiplication* exprime la répétition sur le temps, *RepeatedMultiplication* exprime la répétition sur l'espace. Au niveau du composant *TimeRepeatedMultiplication*, le motif *inDataM* de l'instance de composant *rm* est composé de 1024 données et est construit par une fenêtre glissante sur le temps de longueur 1023. Ce glissement sur le temps s'exprime en ARRAY-OL par le tiler connecté au port *Data1*. Le tiler qui exprime les dépendances de données entre le code de référence et les calculs expriment une relation « un à un », c'est-à-dire que chaque donnée du tableau est utilisée individuellement pour la construction des motifs et cela de façon uniforme. Le composant *RepeatedMultiplication* réalise 1024 multiplications entre les données du port *InDataM* et *InCoeffM* au travers des instanciations du composant *Multiplication* (ce composant est élémentaire).

7.2.1.3 Modélisation de la somme

Nous détaillons à la figure 7.4 le composant qui réalise la somme sur l'espace de 1024 données. Le port d'entrée *InAdditionTree* de cet arbre est de dimension $[(1024)]$ et le port de sortie, *Result*, est un scalaire.

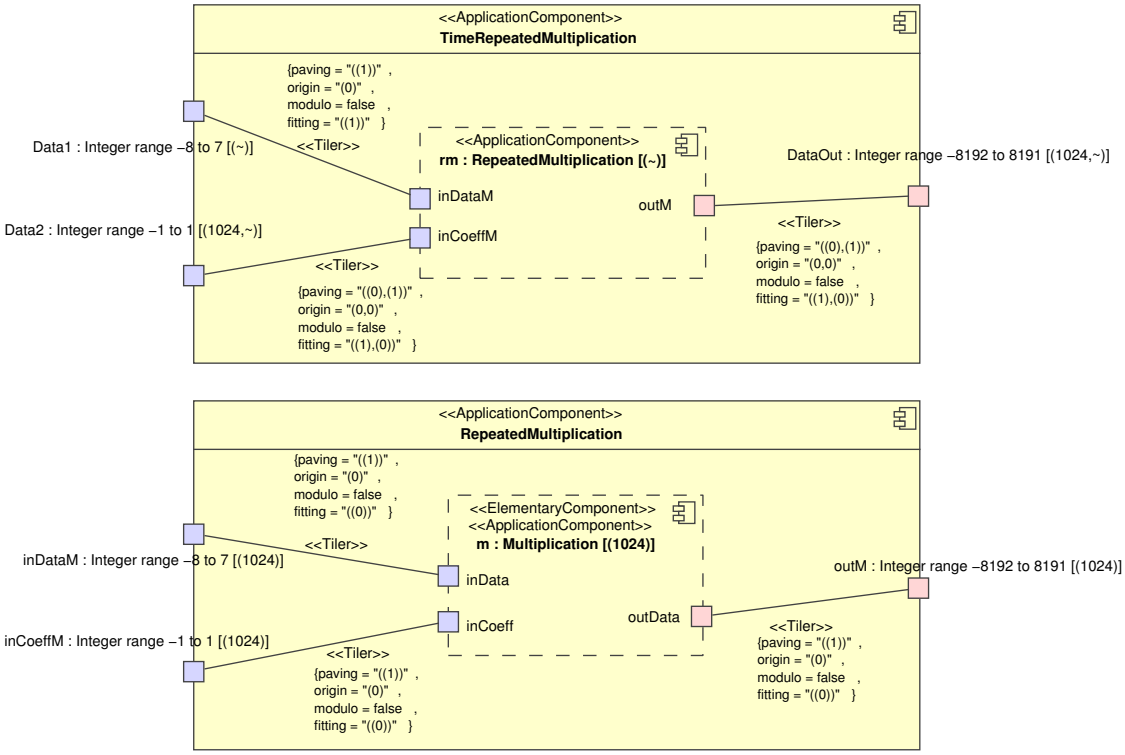


FIG. 7.3: Modélisation des multiplications nécessaires à l'exécution de l'algorithme de la corrélation. Cet algorithme nécessite le glissement d'une fenêtre sur le temps.

Nous décomposons le calcul de cette somme en un arbre, chaque étape de cet arbre réalisant des sommes partielles. Les dimensions des ports entre chaque étage de ce pipeline de tâches diminuent en puissance de deux ($1024 \rightarrow 512 \ldots 2 \rightarrow 1$). Le bas de la figure 7.4 représente le 8^e étage du pipeline de tâche, et qui réalise les sommes partielles des quatre données du port en deux données du port. Afin de réaliser ces deux sommes partielles, la tâche de calcul est répétée 512 fois. Cette tâche est élémentaire et correspond à un additionneur.

Cet additionneur et le multiplieur utilisé pour les besoins de la multiplication sont tous les deux des tâches élémentaires qu'il convient de déployer sur des IPs. Ce déploiement est présenté dans la section suivante.

7.2.1.4 Modélisation du déploiement

L'additionneur et le multiplieur sont des composants élémentaires et correspondent donc à des feuilles de l'application. En dehors de cette sémantique de feuille, aucune information de comportement n'est fournie au travers de cette modélisation. Ce comportement est lié à l'IP que référence chacun de ces composants élémentaires.

La figure 7.5 illustre le déploiement de ces composants. Le haut de la figure représente les deux composants élémentaires, le milieu de la figure les représentations des IPs et le bas de la figure le fichier de bibliothèque qui contient le code des IPs. Les composants sont déployés sur les représentations des IPs par le biais d'une dépendance stéréotypée ≪ImplementedBy≫

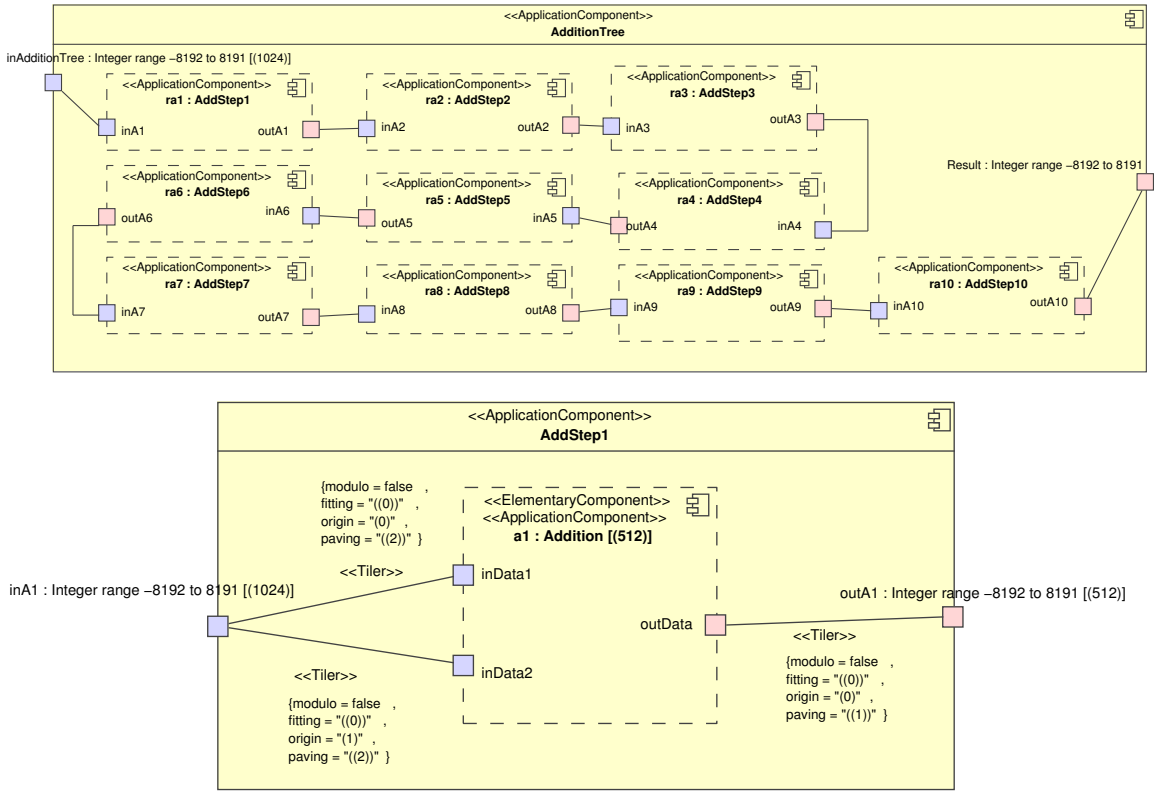


FIG. 7.4: L'arbre qui réalise la somme en parallèle et en pipeline de 1024 données.

et les ports des composants sont liés aux ports des IPs par le biais des dépendances stéréotypées ≪PortImplementedBy≫. Le nom du composant qui représente l'IP et les noms de ses ports sont ceux utilisés dans le code de ces IPs, cela permet notamment de spécifier la manière dont est connecté l'IP au composant élémentaire. Les deux représentations d'IPs pointent le même fichier de bibliothèque dont le chemin dans les répertoires est spécifié par l'attribut *filepath*.

Nous venons de modéliser un algorithme de détection pour un radar dans l'environnement Gaspard. La puissance de calcul requise pour l'exécution de cet algorithme n'est pas négligeable puisqu'environ 200 GOp/s sont nécessaires (le découpage très fin de l'algorithme contribue largement à l'augmentation de cette puissance de calcul, dont le nombre d'opérations par seconde est donc à tempérer puisque les calculs sont réalisés sur quelques bits). Cet algorithme est utilisé dans notre système embarqué de la même façon que l'algorithme $J_{toce}(io)$.

7.2.2 Modélisation de l'algorithme $J_{toce}(io)$

Bien que l'algorithme $J_{toce}(io)$ soit plus complexe que l'algorithme de corrélation, sa modélisation est relativement proche car il est composé de trois corrélations (équations 7.3

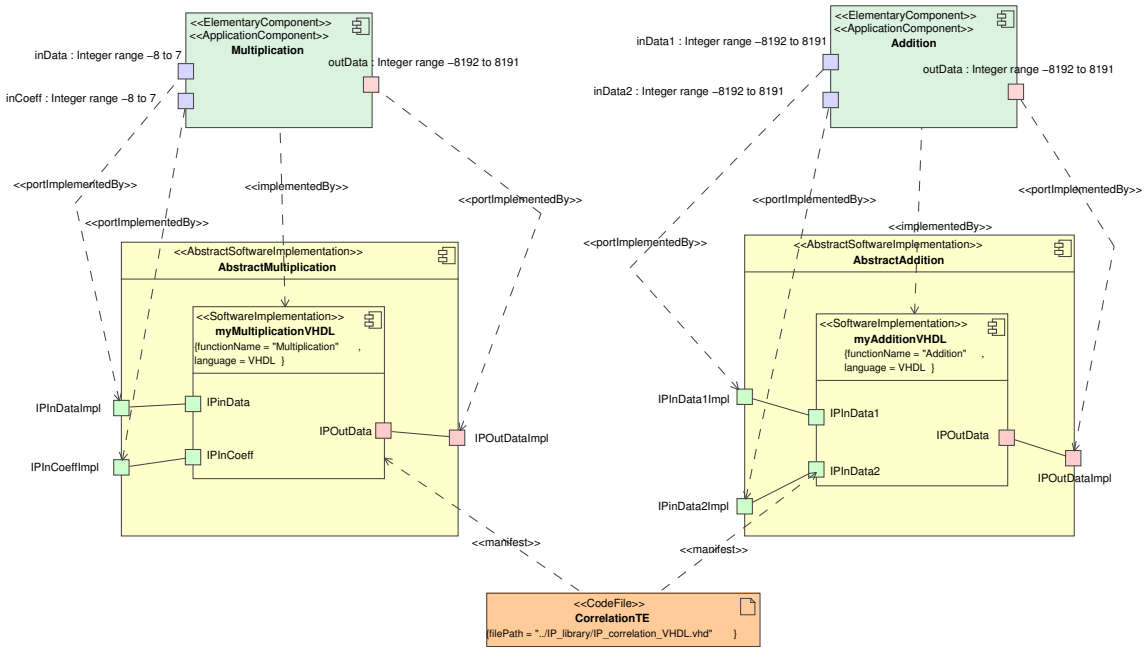


FIG. 7.5: Modélisation UML du déploiement de la multiplication et l'addition. Ces composants élémentaires sont déployés sur des représentations d'IPs, qui référencent elles-mêmes un fichier en bibliothèque contenant le code de ces IPs.

et 7.4) : une corrélation appliquée sur le résultat de deux autres corrélations[1].

7.2.3 Modélisation du système de détection complet

La figure 7.6 représente le système de détection complet. Le signal reçu depuis le radar *SignalRadar* est envoyé aux deux algorithmes de détection (les instances *corr* et *jtoce*) de la même façon que le flux de code de référence *Coeff*. Les résultats des algorithmes sont connectés aux ports de sortie du système (ports *OutCorrelation* et *OutJtoce*).

7.2.4 Bilan

Hormis les types utilisés et le déploiement, la modélisation de l'application réalisée dans cette section est indépendante de toute cible d'exécution. Au travers de modifications mineures, cette modélisation peut être exécutée sur une plateforme multiprocesseurs par le biais de la génération d'un code Fortran/OpenMP ou simulée sur une architecture multiprocesseurs en SystemC. Nous ciblons une exécution matérielle de cet algorithme et le plaçons par conséquent sur un accélérateur matériel dans l'architecture. Nous présentons dans la section suivante l'accélérateur matériel généré par notre flot de conception au travers de résultats de simulation et de synthèse.

[1] L'algorithme $J_{toce}(io)$ est aussi disponible sous la forme d'un IP que nous avons développé en collaboration avec l'Université de Montréal [70].

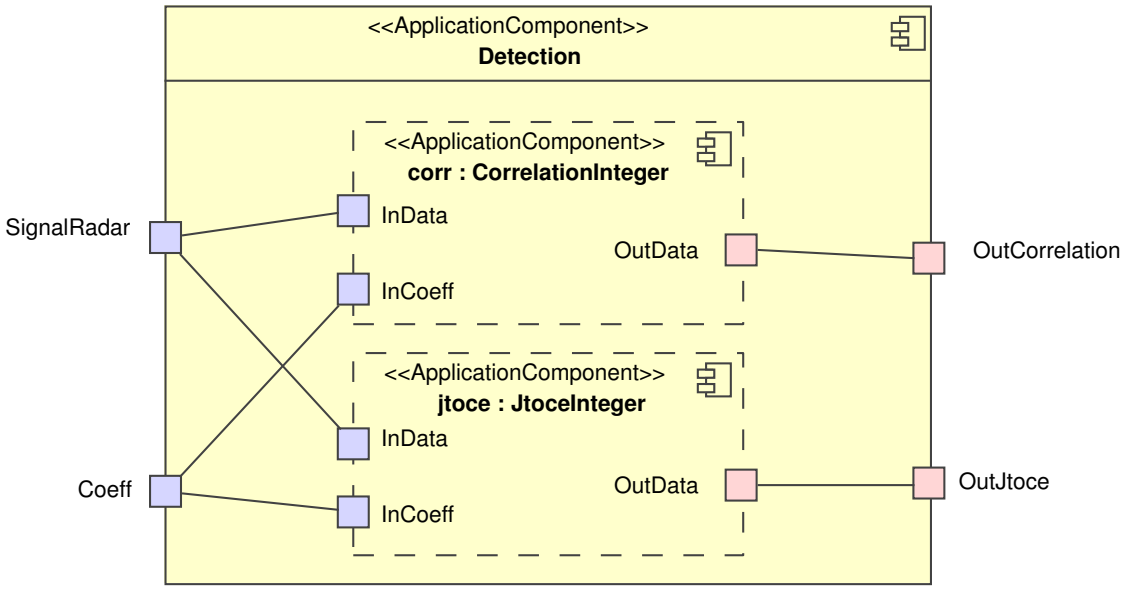


FIG. 7.6: Vue du système complet de détection composé des deux algorithmes.

7.3 Génération de l'accélérateur matériel

Le modèle UML du système de détection est compilé par notre flot de conception dans le but de générer un accélérateur matériel. La figure 7.7 représente l'environnement Gaspard (intégré dans Eclipse sous la forme d'un plugin) et les modèles intervenant dans la compilation. Le modèle UML correspond au modèle de notre système de détection (il est directement généré depuis le modeleur UML) et est transformé en un modèle à l'aide d'une transformation de modèles qui « traduit » un modèle UML stéréotypé en un modèle conforme au métamodèle Deployed. La transformation de modèles présentée dans le chapitre 5 permet alors la création d'un modèle conforme au métamodèle RTL décrit dans le chapitre 3. La dernière étape de notre flot de conception est la génération d'un code VHDL par le biais de la transformation modèle vers texte détaillée dans le chapitre 4. Nous présentons dans le reste de la section les différents résultats de simulation et de synthèse de ce code VHDL.

7.3.1 Simulation sous ModelSim

Afin de procéder à la simulation de l'accélérateur dans l'environnement ModelSim et à la vérification de son comportement, nous utilisons des stimuli qui émulent la présence d'un obstacle dans un environnement plus ou moins bruité. Dans ces stimuli, le bruit est initialement nul, puis croît jusqu'à devenir, en dernier lieu, omniprésent (il n'y a plus l'information d'obstacles). Le paragraphe suivant présente le comportement de l'accélérateur en présence de ces stimuli.

La figure 7.8 illustre les signaux produits par les deux algorithmes de détection en présence de stimuli introduits précédemment, avec en haut la corrélation (le port *OutCorrelation* sur la figure 7.6) et en bas l'algorithme J_{toce} (le port *OutJtoce*). Les pics représentent une forte

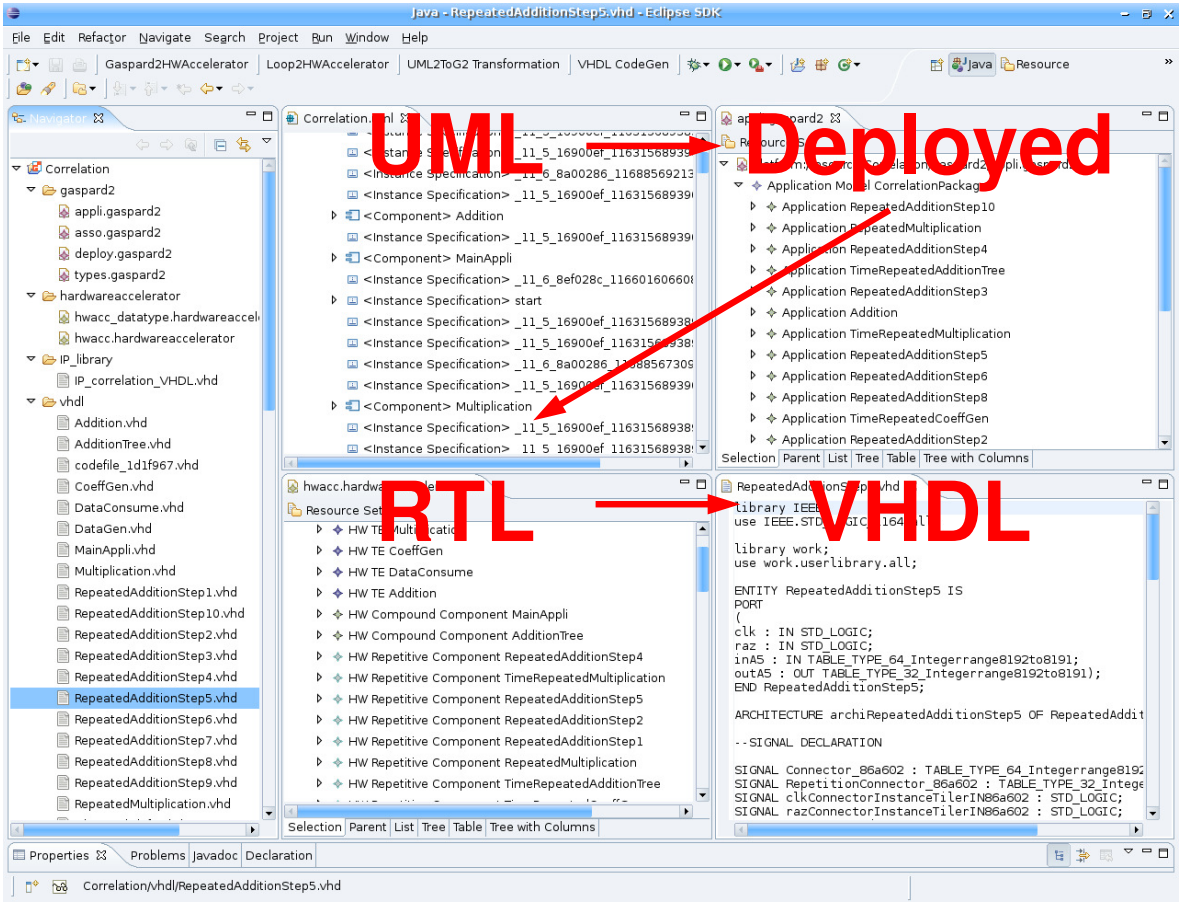


FIG. 7.7: Les modèles UML, Deployed et RTL ainsi que le code VHDL sont produits automatiquement par transformation de modèles dans l'environnement Gaspard.

correspondance entre le code de référence et le signal reçu, ils sont assimilés à la présence d'obstacles. Plus la valeur du pic est élevée, plus la correspondance est forte et plus la présence d'un obstacle est avérée. Avec l'augmentation continuelle de bruit dans le signal reçu, nous constatons une légère diminution des pics pour l'algorithme de corrélation ainsi qu'une légère augmentation du niveau de bruit. L'algorithme J_{toce} se comporte différemment, la valeur des pics augmente avec la montée du niveau de bruit et le niveau de bruit augmente plus fortement comparé à celui de la corrélation. La fin de la simulation correspond dans les stimuli à l'absence d'obstacles et à la réception de bruit. Le niveau de bruit généré dans l'algorithme J_{toce} et plus élevé que dans celui de la corrélation.

La présence de pics très élevés dans les réponses des algorithmes aux stimuli montre clairement que l'obstacle (émulée dans les stimuli) est détecté. Nous montrons par ailleurs des différences dans le comportement de ces algorithmes, ces différences proviennent de la nature même de ces algorithmes. En effet, les résultats obtenus dans ces simulations sont les mêmes que ceux obtenus par Zaidouni [130] lors de simulation sous MatLab. Par ailleurs, l'exactitude des résultats de simulation obtenus par Zaidouni et les nôtres montre que l'accélérateur matériel généré depuis sa modélisation UML est fonctionnel. L'étape suivante

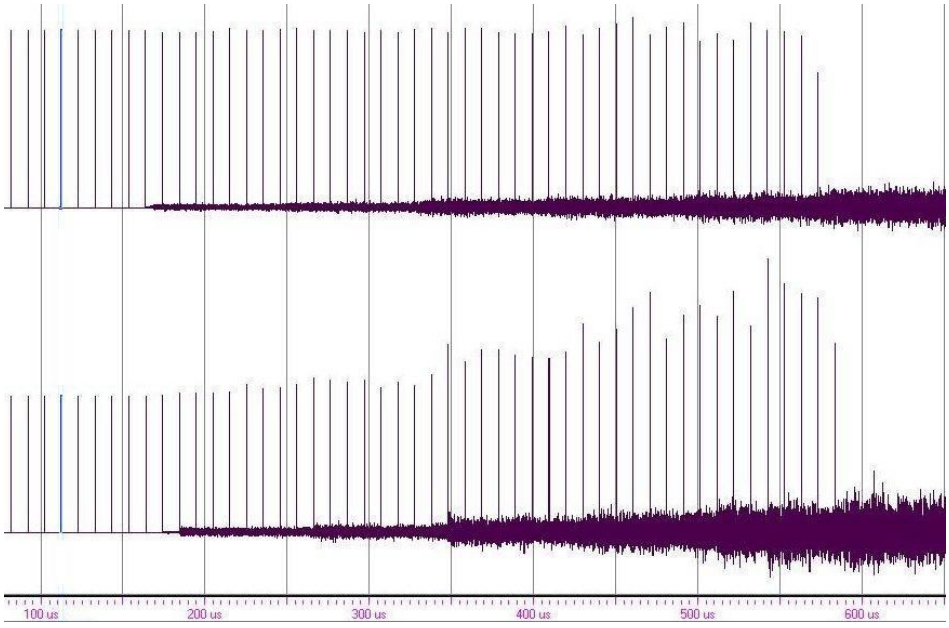


FIG. 7.8: Résultat de simulation des deux algorithmes de détection exécutés sur l'accélérateur matériel généré.

consiste en la synthèse sous Quartus de ce même accélérateur.

7.3.2 Synthèse de l'accélérateur sur un FPGA Stratix2s180

Nous illustrons dans cette section les résultats de synthèse de notre accélérateur sur un FPGA Stratix2s180. Des extraits de code VHDL et différentes vues du circuit synthétisé sont présentés.

7.3.2.1 Vue globale de l'accélérateur

La compilation au travers de notre flot de conception du modèle UML présenté précédemment débouche sur la génération de plusieurs fichiers VHDL. Chaque fichier décrit un composant à l'exception de fichier de librairie **userlibrairy**, lui aussi généré automatiquement, qui définit les interfaces d'instanciation des composants et les types de données (éventuellement multidimensionnels) utilisés dans l'accélérateur.

Le premier fichier présenté correspond au composant de plus haut niveau de hiérarchie de l'accélérateur, il s'agit du composant *Detection* dont le code VHDL est le suivant :

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

library work;
use work.userlibrary.all;

ENTITY Detection IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
SignalRadar : IN TABLE_TYPE_1_Integerrange8to7;
Coeff : IN TABLE_TYPE_1024_1_Integerrange1to1;
OutJtoce : OUT TABLE_TYPE_1_Integer;
OutCorrelation : OUT TABLE_TYPE_1_Integer);
END Detection;

ARCHITECTURE archiDetection OF Detection IS

--SIGNAL DECLARATION

SIGNAL Connector_746115 : TABLE_TYPE_1_Integerrange8to7;
SIGNAL Connector_1f2787b : TABLE_TYPE_1_Integer;
SIGNAL Connector_1c9b01d : TABLE_TYPE_1024_1_Integerrange1to1;
SIGNAL Connector_496eff : TABLE_TYPE_1_Integer;
SIGNAL Connector_11632c7 : TABLE_TYPE_1_Integerrange8to7;
SIGNAL Connector_91af0f : TABLE_TYPE_1024_1_Integerrange1to1;
SIGNAL clkConnectorjtoce : STD_LOGIC;
SIGNAL razConnectorjtoce : STD_LOGIC;
SIGNAL clkConnectorcorr : STD_LOGIC;
SIGNAL razConnectorcorr : STD_LOGIC;


--BEGIN DESCRIPTION

BEGIN

--SIGNAL AFFECTATION

Connector_746115 <= SignalRadar;
Connector_1c9b01d <= Coeff;
OutJtoce <= Connector_1f2787b;
OutCorrelation <= Connector_496eff;
Connector_11632c7 <= SignalRadar;
Connector_91af0f <= Coeff;
clkConnectorjtoce <= clk;
razConnectorjtoce <= raz;
clkConnectorcorr <= clk;
razConnectorcorr <= raz;

--COMPONENT MAPPING

instanceOfJtoceInteger : JtoceInteger
port map(
clk => clkConnectorjtoce,
raz => razConnectorjtoce,
OutData =>Connector_1f2787b,
InData =>Connector_746115,
InCoeff =>Connector_91af0f);

instanceOfCorrelationInteger : CorrelationInteger
port map(
clk => clkConnectorcorr,
raz => razConnectorcorr,
InData =>Connector_11632c7,
InCoeff =>Connector_1c9b01d,
OutData =>Connector_496eff);

END archiDetection;
```

La première partie du code définit les bibliothèques utilisées, nous notons l'utilisation de **userlibrary** qui correspond au fichier de bibliothèque généré. L'entité du fichier (**Entity**) décrit l'interface du composant **Detection**, nous retrouvons les ports **SignalRadar**, **Coeff**, **OutJtoce** et **OutCorrelation** définit dans le modèle UML.

L'architecture **archiDetection** déclare dans un premier temps les signaux utilisés pour la

description de son comportement, puis affecte ces signaux (**Connector_746115 <= SignalRadar ;** par exemple). La dernière étape de la description de **archiDetection** consiste à instancier les composants **JtoceInteger** et **CorrelationInteger**.

La figure 7.9 illustre l'implémentation sur FPGA de l'accélérateur permettant l'exécution matérielle du système de détection complet. Cette vue est issue du résultat de synthèse sous Quartus du code VHDL généré par notre flot de conception.

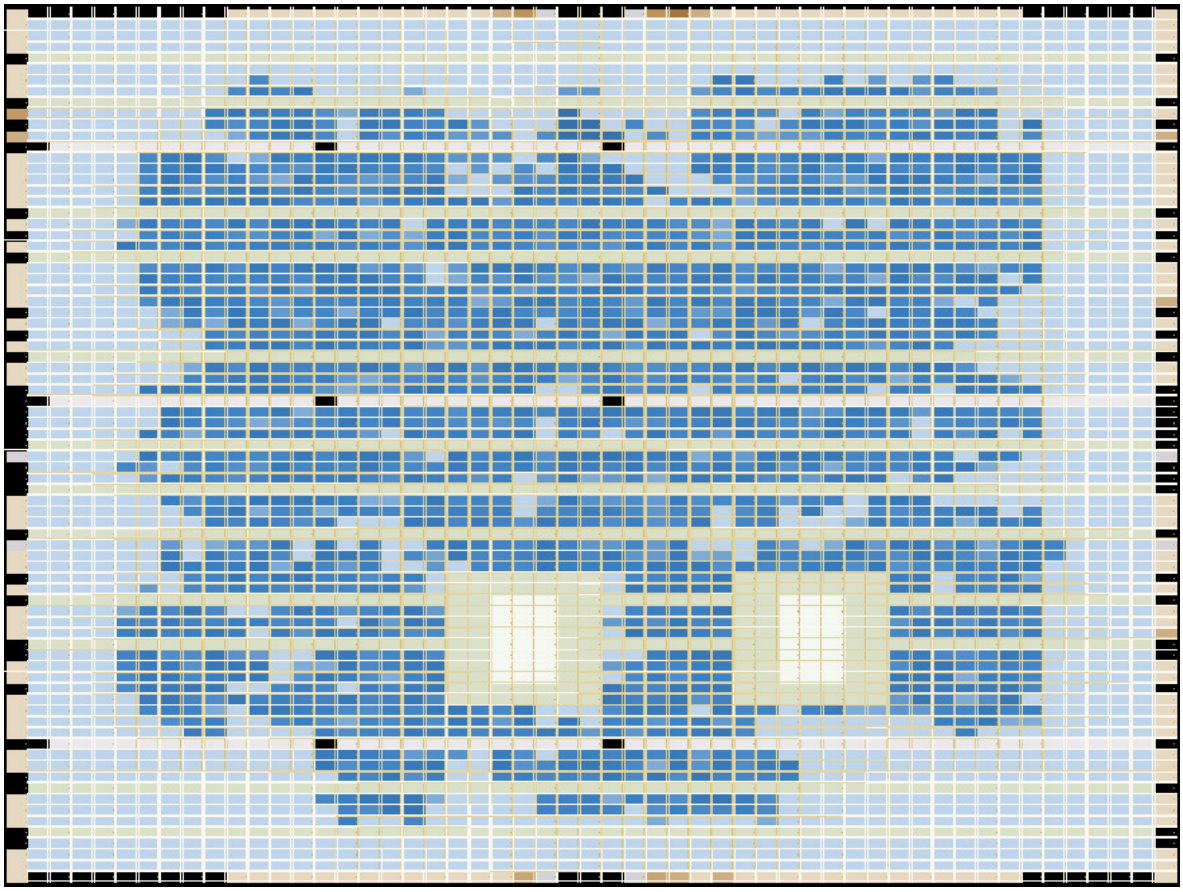


FIG. 7.9: Implémentation sur FPGA de l'accélérateur matériel généré depuis notre flot de conception.

7.3.2.2 Gestion des dépendances de données sur le temps

La description du modèle UML de l'algorithme de corrélation montre qu'un glissement sur le temps est nécessaire lors de la réalisation des multiplications. Ce glissement sur le temps est géré par un tiler dont voici un extrait du code :

```
...
ENTITY TilerIN1db6b07 IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
InputTilerIN1db6b07 : IN TABLE_TYPE_1_Integerrange8to7;
OutputTilerIN1db6b07 : OUT TABLE_TYPE_1_1024_Integerrange8to7);
END TilerIN1db6b07;

ARCHITECTURE archiTilerIN1db6b07 OF TilerIN1db6b07 IS

SIGNAL SignalInputTilerIN1db6b07_1 : TABLE_TYPE_1024_Integerrange8to7;
SIGNAL delayedInputTilerIN1db6b07_1 : TABLE_TYPE_1024_Integerrange8to7;

BEGIN

process (clk,InputTilerIN1db6b07)
begin
    if clk'event and clk='1' then
        delayednputTilerIN1db6b07_1(1024) <= InputTilerIN1db6b07(1);
        for i in 1 to 1023 loop
            delayedInputTilerIN1db6b07_1(i) <= delayedInputTilerIN1db6b07_1(i+1);
        end loop;
    end if;
end process;

SignalInputTilerIN1db6b07_1 <= delayedInputTilerIN1db6b07_1;


OutputTilerIN1db6b07(1)(1) <= SignalInputTilerIN1db6b07_1(1);
OutputTilerIN1db6b07(1)(2) <= SignalInputTilerIN1db6b07_1(2);
OutputTilerIN1db6b07(1)(3) <= SignalInputTilerIN1db6b07_1(3);
OutputTilerIN1db6b07(1)(4) <= SignalInputTilerIN1db6b07_1(4);
...
```

Le signal **delayedInputTilerIN1db6b07_1** permet la création du registre à décalage de longueur 1024 (la 1024^e est directement connectée à la donnée à décaler), c'est-à-dire la taille du signal. La boucle dans le **process** VHDL correspond à la création du registre à décalage dont le point d'entrée est la donnée **InputTilerIN1db6b07(1)**. Les données dans le motif de sortie (**OutputTilerIN1db6b07**) sont connectées sur les différents étages du registre à décalage. La figure 7.10 présente une partie de la synthèse de ce tiler, les boîtes en haut représentent la cascade de registres, des fils permettent de connecter les données retardées dans les registres avec les données du motif sur la droite.

7.3.2.3 Arbre de la corrélation

Cette section s'intéresse au pipeline de tâches de l'arbre d'additions dans l'algorithme de la corrélation. La modélisation en UML de cet arbre démontre que de simples connecteurs permettent de définir des dépendances contenant 1024 données, puis 512, etc. Il est donc intéressant d'observer les résultats de synthèse de cet arbre afin de se rendre compte du pouvoir d'expression d'un « simple » connecteur UML. La figure 7.11 représente la vue partielle du résultat de synthèse (seuls les 6 derniers étages sont représentés) et l'évolution de la quantité de données transmises entre chaque tâche dans l'arbre.

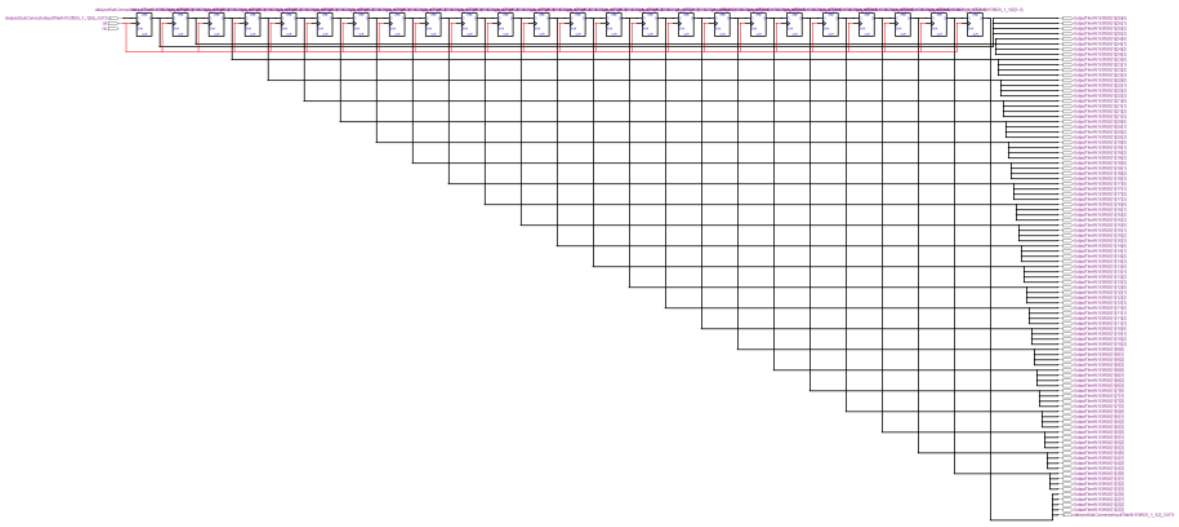


FIG. 7.10: Vue partielle du résultat de synthèse sous Quartus du tiler issue de la gestion de dépendance de données sur le temps.

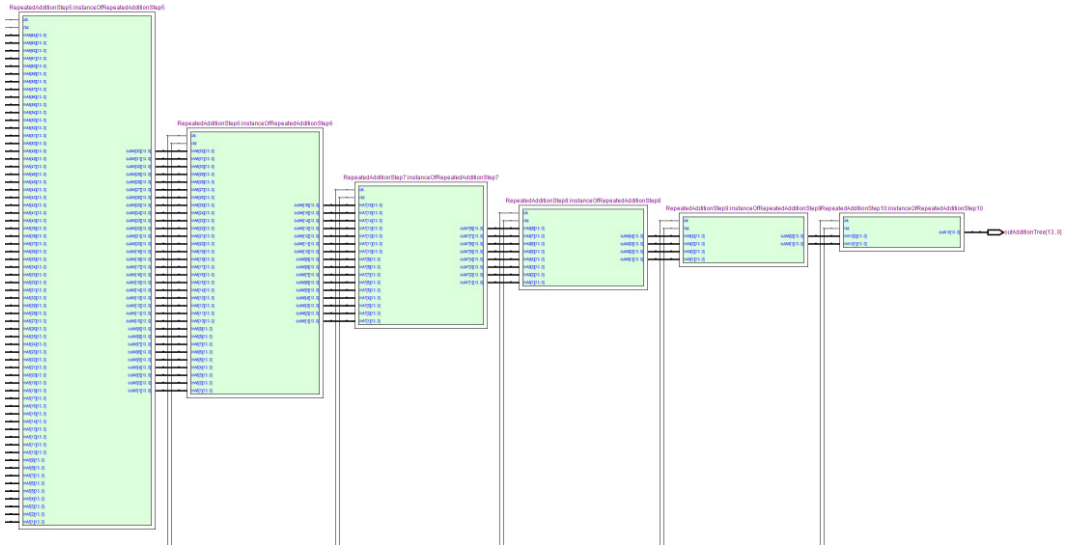


FIG. 7.11: Vue partielle du résultat de synthèse de la somme dans la corrélation.

7.3.3 Test de l'accélérateur matériel

Dans le cadre du projet européen ModEasy[2], nous avons eu l'opportunité de bénéficier du matériel nécessaire à la mise en œuvre du système de détection. Nous avons ainsi participé, en collaboration avec l'INRETS[3] de Lille (Institut National de Recherche sur les

[2] http://www2.lifl.fr/modeasy/
[3] http://www.inrets.fr/infos/centres/ct_vascq.html

Transports et leur Sécurité) et l'IEMN[4] de Valenciennes (Institut d'Electronique de Microélectronique et de Nanotechnologie), à la validation du bon fonctionnement de l'algorithme de corrélation pour la détection d'obstacles lors de tests en conditions réelles sur route.

7.4 Conclusion

Ce chapitre illustre l'utilisation de notre flot de conception dans le cadre de la sécurité et du transport. L'objectif précis de cette étude est la conception d'un radar anti-collision dont l'étape de traitement de signal systématique est exécutée par un accélérateur matériel implémenté sur FPGA. Cette étape consiste à mettre en évidence les similitudes entre le signal reçu par l'antenne (signal préalablement démodulé et numérisé) et un code de référence. Pour cela, nous utilisons deux algorithmes, l'un est modélisé finement en UML, l'autre est utilisé sous la forme d'un IP. Le code VHDL complet de l'accélérateur est généré à l'issue de la compilation du modèle UML dans notre flot de conception. Les résultats de simulation du code VHDL généré valident le bon fonctionnement des deux algorithmes. Trois vidéos sont à disposition [59], la première[5] montre la modélisation UML de l'algorithme de corrélation, la seconde[6] illustre le processus de compilation (la transformation de modèles et la génération de code) et la troisième[7] reprend des résultats de simulation et synthèse du code VHDL généré.

Par ailleurs, les modélisations en UML ont facilité les échanges, les transferts d'informations et la communication avec les personnes impliquées par ce projet. En effet, des spécialistes du traitement du signal, de l'IDM, des architectures processeurs et du parallélisme ont interagi et communiqué autour du modèle UML modélisant l'application.

Au final, nous disposons d'un flot de conception qui, à partir d'une modélisation d'application en UML, génère le code VHDL d'un accélérateur synthétisable. La modélisation d'une application est indépendante de tout détail implémentation, dans la mesure où la saisie d'un modèle est réalisée dans un langage unifié. Notre flot de conception est donc utilisable par des non-spécialistes des architectures processeurs.

[4] http://www.iemn.univ-lille1.fr/
[5] http://www2.lifl.fr/west/DaRTShortPresentations/correlation-magicdraw.avi
[6] http://www2.lifl.fr/west/DaRTShortPresentations/correlation-eclipse.avi
[7] http://www2.lifl.fr/west/DaRTShortPresentations/correlation-vhdl.avi

Conclusion

Bilan

Les travaux présentés dans ce document s'intéressent à la génération d'accélérateurs matériels permettant l'exécution d'applications de traitement de signal systématique modélisées à un haut niveau d'abstraction. Il résulte de ces travaux la proposition d'un flot de conception automatisé et entièrement dirigé par les modèles. Sa mise en œuvre a nécessité des contributions dans différents domaines tels que l'Ingénierie Dirigée par les Modèles (IDM) et la co-conception.

Exécution matérielle des applications Gaspard

Le point de départ pour la construction du flot de conception est indéniablement la définition d'un modèle d'exécution pour les applications Gaspard. Le modèle d'exécution que nous avons proposé supporte la hiérarchie, le parallélisme de tâches et le parallélisme de données. Deux restrictions peuvent être portées à ce modèle d'exécution : les données doivent être cadencées au rythme des calculs et l'exécution séquentielle du parallélisme de données n'est effective qu'au plus haut niveau de hiérarchie de l'accélérateur. Nous avons toutefois montré que nous pouvions passer outre cette dernière restriction au travers de notre processus d'optimisation.

Métamodèle RTL

La conceptualisation des mécanismes mis en œuvre pour les besoins de l'exécution matérielle des applications Gaspard a débouché sur la création du métamodèle RTL. Un des objectifs alors fixé était la métamodélisation des accélérateurs *au niveau RTL* et non *dans un langage HDL* : aucune syntaxe HDL ne devait venir entacher le métamodèle RTL. Par manque de temps, nous n'avons pas défini la transformation modèle vers texte qui permettrait la génération de code Verilog. Cette transformation aurait constitué une preuve rigoureuse de l'indépendance du métamodèle RTL vis-à-vis des langages HDL. Cependant l'absence de syntaxe propre à VHDL dans le métamodèle RTL ainsi que la complexité des templates de génération de code VHDL, qui ne nécessitent pas de parcours complexes d'un modèle RTL, laissent effectivement penser que le métamodèle RTL est bien indépendant des syntaxes HDL.

Nous avons réalisé ce même travail de conceptualisation pour les représentations des FPGAs et des placements de tâches. Si le choix d'utiliser les FPGAs comme cible d'implémentation des accélérateurs était une évidence, leur métamodélisation restait plus délicate. En effet, la nature même des FPGAs prête à confusion car ils représentent, par le biais d'une

configuration, à la fois un support de réalisation physique et une architecture fonctionnelle potentiellement complexe. Notre premier objectif a donc consisté à faire abstraction de la configuration en conceptualisant les caractéristiques intrinsèques aux FPGAs. Le second objectif était la conceptualisation dans le métamodèle RTL des notions d'implémentation de composants et de placement de tâches.

Au final, le métamodèle RTL offre l'opportunité de représenter dans un même modèle un accélérateur et son placement sur FPGA. La génération de code permet de retranscrire sous la forme d'un script interprétable par des outils de synthèse ces informations de placement.

Transformation de modèles

La transformation d'un modèle Deployed en un modèle RTL est cruciale dans notre chaîne de compilation. Elle permet le raffinement d'un modèle d'application Gaspard indépendant de toute cible d'exécution en un modèle RTL qui décrit un accélérateur matériel. Ce raffinement passe, par exemple, par la propagation d'une horloge dans l'accélérateur ou la génération de multiplexeurs pour l'aiguillage des motifs lors d'une exécution séquentielle du parallélisme de données. Selon la devise « Diviser pour mieux régner », nous avons décomposé la transformation de modèles en règles de transformation. Plusieurs règles sont ainsi exécutées pour la compilation du parallélisme de données. En particulier, la règle de transformation des tilers appelle une fonction, dite boîte noire du point de vue de la transformation, qui génère un circuit à partir des dépendances de données exprimées dans un tiler. Cette fonction est par ailleurs en mesure de gérer aussi bien les dépendances de données sur l'espace que celles sur le temps.

L'écriture des règles de transformations a chronologiquement été précédée de représentations graphiques des règles en TrML. Certaines représentations de ces règles ont mis en évidence des ambiguïtés ou des anomalies dans la notation graphique. Dans la mesure où des modifications en conséquence ont été apportées à TrML, nous avons contribué à son évolution.

Optimisation des accélérateurs

À partir des fonctions de refactoring ARRAY-OL existantes, nous avons construit les fonctions de refactoring PARALLÉLISATION et SÉQUENTIALISATION qui augmentent ou réduisent la quantité de ressources consommées par un accélérateur au coût d'une baisse ou d'une hausse de performances. Ces fonctions de refactoring sont pilotées par une heuristique d'optimisation qui tend à ajuster la quantité de ressources consommées par un accélérateur avec celles disponibles sur FPGA. Le principal avantage de cette méthode est que nous profitons des travaux existants concernant les fonctions de refactoring ARRAY-OL. Par ailleurs, l'idée d'utiliser les fonctions de refactoring ARRAY-OL pour l'optimisation des applications n'est pas nouvelle dans notre équipe, mais elles n'avaient pourtant jamais été utilisées dans un processus automatisé. Il résulte de nos travaux que les fonctions de refactoring ARRAY-OL doivent être manipulées dans les processus d'optimisation sous une forme abstraite et dépendante de leurs impacts sur les cibles d'exécution.

Validation expérimentale

Tout au long de cette thèse, nous avons eu l'opportunité de mettre à l'épreuve notre flot de conception dans le domaine du transport et de la sécurité automobile. Nous nous

sommes plus particulièrement intéressés à des algorithmes de détection d'obstacles utilisés dans le cadre d'un système anti-collision pour véhicules. Un algorithme a ainsi été modélisé en UML. Le code VHDL généré depuis notre flot de conception a été simulé puis synthétisé sur FPGA, validant le bon fonctionnement de l'accélérateur produit et démontrant ses performances suffisantes par rapport aux contraintes initiales. Cette étude de cas a permis de valider notre flot de conception, des vidéos qui illustrent sa compilation complète au travers de notre flot de conception sont disponibles [59].

Perspectives

Grille de FPGAs

La gestion des placements de tâches sur une grille d'accélérateurs (une grille de FPGAs) demeure une extension potentielle de nos travaux. Elle nécessite l'utilisation des représentations polyédriques à un niveau hiérarchique donné car ils permettent d'abstraire la complexité des placements de tâches réalisés sur cette grille. Cependant, l'utilisation de l'espace de répétition d'une tâche au niveau local à un accélérateur (un FPGA dans la grille) permet de conserver les avantages liés au placement et à l'optimisation que nous avons mis en œuvre durant cette thèse. La plus grande difficulté sera sans doute la gestion des communications entre les différents FPGAs de la grille.

Interfaçage des accélérateurs

Les accélérateurs que nous générons sont en parti destinés à être utilisés dans des SoCs. Dans ce contexte d'utilisation, les ports de l'accélérateur sont connectés à un bus ou à un réseau d'interconnexion sur puce. Ces connexions sont réalisées au travers d'interfaces et d'un protocole qu'il convient de gérer. Dans leur état actuel, nos travaux ne supportent pas la génération de ces interfaces depuis la modélisation à haut niveau d'abstraction. Nous avons cependant identifié les points clés qui permettront de générer ces interfaces dans des extensions de nos travaux. Nous détaillons ces points clés dans l'annexe A.

La compilation du flot de contrôle dans les applications Gaspard

Des travaux menés au sein de notre équipe (débutés par Ouassila Labbani [66] et poursuivis actuellement par Huafeng Yu [129]) visent à intégrer la notion de flot de contrôle dans le flot de données des applications Gaspard. Dans le cadre de cette thèse, j'ai récemment mené une étude qui a débouché sur une extension de notre modèle d'exécution matérielle et, par conséquent, sur le métamodèle RTL. Ces extensions, couplées à de nouvelles règles de transformation et de nouveaux templates de génération de code VHDL, ont montré que nous savions générer ce flot de contrôle jusque dans les accélérateurs matériels, cela à partir de modélisations en UML.

Il résulte de cette étude préliminaire que le contrôle d'une application Gaspard modélisée indépendamment de la cible d'exécution permet de générer un accélérateur qui implémente le contrôle sous la forme d'un multiplexeur commandé par un automate. La surface du FPGA est donc exploitée pour la réalisation de ce contrôle. Des travaux menés par Imran Quadri [58] sont en cours dans notre équipe pour le développement d'une nouvelle compilation du contrôle qui vise à exploiter la reconfiguration dynamique et partielle qu'offrent cer-

tains FPGA. Le résultat attendu est un contrôleur implémenté statiquement sur une région du FPGA tandis qu'une autre région serait dédiée à l'implémentation des tâches contrôlées au travers de reconfigurations. Des gains conséquents en énergie, surface et flexibilité sont attendus de cette compilation du contrôle, qui nécessite toutefois des extensions de nos travaux concernant la métamodélisation des FPGAs afin de permettre la représentation et la gestion des configurations.

Les FPGAs dans l'environnement Gaspard

L'utilisation des FPGAs dans l'environnement Gaspard va perdurer. En effet, divers projets en cours de réalisation visent à exploiter leur dynamicité (via la compilation du contrôle que nous venons d'introduire) ou à les utiliser en tant que plateformes d'exécution pour diverses architectures modélisées en UML. En particulier, le projet MppSoC [40, 80] consiste à construire une machine SIMD à partir d'une modélisation à un haut niveau d'abstraction en réutilisant des IPs[8]. Un des objectifs de ce projet reste la réalisation de cette machine SIMD sur FPGA depuis sa modélisation en UML. Pour cela, le métamodèle RTL sera enrichi de concepts architecturaux tels que des réseaux d'interconnexions, des processeurs maîtres et esclaves, etc. et de nouvelles règles de transformations seront développées.

[8]Nous présentons dans [80] l'architecture de MppSoC et les premiers résultats d'implémentation sur FPGA. Dans [40], nous introduisons le réseau d'interconnexion utilisé pour la connexion des PEs (Processing Elements) dans la grille de la machine SIMD.

Bibliographie personnelle

[p1] Jean-Luc Dekeyser, Sébastien Le Beux, and Philippe Marquet. Une approche modèle pour la conception conjointe de systèmes embarqués hautes performances dédiés au transport. In *Workshop International : Logistique & Transport (LT' 2007)*, Sousse, Tunisie, novembre 2007.

[p2] 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.

[p3] Simon Duquennoy, Sébastien Le Beux, Philippe Marquet, Samy Meftali, and Jean-Luc Dekeyser. MpNoC design : Modeling and simulation. In *15th IP Based SoC Design Conference (IP-SoC 2006)*, Grenoble, France, décembre 2006.

[p4] Sébastien Le Beux, Vincent Gagne, El Mostapha Aboulhamid, Philippe Marquet, and Jean-Luc Dekeyser. Hardware/software exploration for an anti-collision radar system. In *The 49th IEEE International Midwest Symposium on Circuits and Systems*, San Juan, Puerto Rico, août 2006.

[p5] Sébastien Le Beux, Philippe Marquet, and Jean-Luc Dekeyser. A design flow to map parallel applications onto FPGAs. In *17th IEEE International Conference on Field Programmable Logic and Applications, FPL*, Amsterdam, Netherlands, août 2007.

[p6] Sébastien Le Beux, Philippe Marquet, and Jean-Luc Dekeyser. Multiple abstraction views of FPGA to map parallel application. In *3rd International Workshop on Reconfigurable Communication Centric System-on-Chips, ReCoSoC*, Montpellier, France, juin 2007.

[p7] Sébastien Le Beux, Philippe Marquet, Ouassila Labbani, and Jean-Luc Dekeyser. FPGA implementation of embedded cruise control and anti-collision radar. In *DSD'2006, 9th Euromicro conference on digital system design*, Dubrovnik, Croatia, août 2006.

[p8] Sébastien Le Beux and Loic Lagadec. Madeo, une approche MDA pour la programmation et la synthèse d'architectures reconfigurables. In *Sympa'2005*, pages 1–12, Le Croisic, France, avril 2005.

[p9] Philippe Marquet, Simon Duquennoy, Sébastien Le Beux, Samy Meftali, and Jean-Luc Dekeyser. Massively parallel processing on a chip. In *ACM Int'l Conf. on Computing Frontiers*, Ischia, Italy, mai 2007.

[p10] Sébastien Le Beux, Philippe Marquet, Antoine Honoré, and Jean-Luc Dekeyser. A Model Driven Engineering Design Flow to Generate VHDL. In *International ModEasy'07 Workshop*, Barcelona, Spain, septembre 2007. http://www.lifl.fr/modeasy/workshop.html.

Bibliographie

[1] Two Flows for Partial Reconfiguration : Module Based or Difference Based. In *Xilinx Application Note XAPP290, Version 1.2*, septembre 2004.

[2] Acceleo. Générateur de code Acceleo. http://www.acceleo.org/pages/accueil/fr.

[3] Ali Ahmadinia, Christophe Bobda, Marcus Bednara, and Jurgen Teich. A new approach for on-line placement on reconfigurable devices. 4 :134a, 2004.

[4] 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.

[5] D. H. Akehurst, B. Bordbar, M. Evans, W. G. Howells, and K. D. McDonald-Maier. SiTra : Simple transformations in java. In *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences)*, Genova, Italy, octobre 2006.

[6] David Akehurst, Gareth Howells, Klaus McDonald-Maier, and Behzad Bordbar. An Experiment in Using Model Driven Development : Compiling UML State Diagrams into VHDL. In *Forum on specification and design languages (FDL'07)*, Barcelona, Spain, septembre 2007.

[7] Altera. Stratix III Device Family, 2007.

[8] B. Blodget and S. McMillan and P. Lysaght. A lightweight approach for embedded reconfiguration of FPGAs. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 399–400, 2003.

[9] Shiv Balakrishnan and Chris Eddington. Efficient dsp algorithm development for fpga and asic technologies. White paper, Synplicity, juillet 2007.

[10] 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.

[11] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers*, 17(1) :68–83, 2000.

[12] Marcus Bednara and Jürgen Teich. Automatic Synthesis of FPGA Processor Arrays from Loop Algorithms. *The Journal of Supercomputing*, 26(2) :149–165, septembre 2003.

[13] 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.

[14] 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.

[15] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L . Guernic, and R. de Simone. Synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1) :64–83, janvier 2003.

[16] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Press, 1999.

[17] Vaughn Betz and Jonathan Rose. Vpr : A new packing, placement and routing tool for fpga research. In *FPL '97 : Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 213–222, London, UK, 1997. Springer-Verlag.

[18] Dag Björklund and Johan Lilius. From UML Behavioral Descriptions to Efficient Synthesizable VHDL. In *Proceedings of the 20th IEEE Norchip Conference*, novembre 2002.

[19] Pierre Boulet. Array-OL revisited, multidimensional intensive signal processing specification. Rapport de recherche RR-6113, INRIA, février 2007.

[20] Pierre Boulet, Jean-Luc Dekeyser, Alain Demeure, Florent Devin, and Philippe Marquet. Une approche à la SQL du traitement de données intensif dans Gaspard. In *RenPar'11, Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, Rennes, France, juin 1999.

[21] Pierre Boulet, Cédric Dumoulin, and Antoine Honoré. *From MDD concepts to experiments and illustrations*, chapter Model Driven Engineering for System-on-Chip Design. ISTE, International scientific and technical encyclopedia, Hermes science and Lavoisier, septembre 2006.

[22] 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, septembre 2007. Invited Paper.

[23] Betul Buyukkurt, Zhi Guo, and Walid A. Najjar. Impact of loop unrolling on area, throughput and clock frequency in ROCCC : C to VHDL compiler for FPGAs. In Springer Berlin / Heidelberg, editor, *Reconfigurable Computing : Architectures and Applications*, volume 3985/2006, pages 401–412, 2006.

[24] Jaime S. Cardoso and Luís Corte-Real. Accumulator size minimization for a fast cumulant-based motion estimator. In *IEEE Transactions on Circuits and Systems for Video Technology*, volume 15, pages 1660–1664, décembre 2005.

[25] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice. LUSTRE *: a declarative language for real-time programming*. ACM Press, 1987.

[26] Sumanta Chaudhuri, Jean-Luc Danger, , and Sylvain Guilley. Efficient Modeling and Floorplanning of Embedded-FPGA Fabric. In *17th IEEE International Conference on Field Programmable Logic and Applications, FPL*, Amsterdam, Netherlands, août 2007.

[27] Philippe Coussy, Gwenole Corre, Pierre Bomel, Eric Senn, and Eric Martin. High-level synthesis under I/O timing and memory constraints. In *IEEE International Symposium on Circuits and Systems, ISCAS*, pages 23–26, mai 2005.

[28] Frank P. Coyle and Mitchell A. Thornton. From UML to HDL : a model driven architectural approach to hardware-software co-design. *Information Systems : New Generations Conference (ISNG)*, pages 88–93, avril 2005.

[29] Arnaud Cuccuru, Jean-Luc Dekeyser, Philippe Marquet, and Pierre Boulet. Towards UML 2 extensions for compact modeling of regular complex topologies. In *MoDELS/UML 2005, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, octobre 2005.

[30] Robertas Damasevicius and Vytautas Stuikys. Application of uml for hardware design based on design process model. In *ASP-DAC '04 : Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 244–249, Piscataway, NJ, USA, 2004. IEEE Press.

[31] DaRT Team LIFL/INRIA, Lille, France. Graphical array specification for parallel and distributed computing (GASPARD2). https://gforge.inria.fr/projects/gaspard2/, 2008.

[32] Jean-Luc Dekeyser, Sébastien Le Beux, and Philippe Marquet. Une approche modèle pour la conception conjointe de systèmes embarqués hautes performances dédiés au transport. In *Workshop International : Logistique & Transport (LT' 2007)*, Sousse, Tunisie, novembre 2007.

[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.

[34] Tata Research Development and Design Centre. Modelmorf – a model transformer. http://www.tcs-trddc.com/ModelMorf.

[35] Harald Devos, Kristof Beyls, Mark Christiaens Jan Van Campenhout, Erik H. D'Hollander, and Dirk Stroobandt. Finding and applying loop transformations for generating optimized fpga implementations. *Transactions on High-Performance Embedded Architectures and Compilers I*, 4050/2007 :159–178, juillet 2007.

[36] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, and Dirk Stroobandt. From loop transformation to hardware generation. In *Proceedings of the 17th ProRISC Workshop*, pages 249–255, Veldhoven, novembre 2006.

[37] Catherine Dezan. *Géneration automatique de circuits avec ALPHA du CENTAUR*. PhD thesis, IRISA, février 1993.

[38] A. Dias, C. Lavarenne, M. Akil, and Y. Sorel. Optimized implementation of real-time image processing algorithms on field programmable gate arrays. In *4'th Int. Conf. on Signal Processing, ICSP'98*, Beijing, China, 1998.

[39] 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.

[40] Simon Duquennoy, Sébastien Le Beux, Philippe Marquet, Samy Meftali, and Jean-Luc Dekeyser. MpNoC design : Modeling and simulation. In *15th IP Based SoC Design Conference (IP-SoC 2006)*, Grenoble, France, décembre 2006.

[41] Eclipse Consortium. EMF. http://www.eclipse.org/emf, 2007.

[42] Eclipse Consortium. JET, Java Emitter Templates. http://www.eclipse.org/modeling/m2t/?project=jet, 2007.

[43] Anne Etien, Cedric Dumoulin, and Emmanuel Renaux. Towards a unified notation to represent model transformation. Rapport de recherche RR-6187, INRIA, mai 2007.

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

[45] Erwan Fabiani and Dominique Lavenier. Placement of linear arrays. In *FPL '00 : Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pages 849–852, London, UK, 2000. Springer-Verlag.

[46] 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.

[47] 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.

[48] Terry Filiba, Man-Kit Leung, and Vinayak Nagpal. VHDL code generation in the ptolemy II environment. Technical report, Electrical Engineering and Computer Sciences, UC Berkeley, décembre 2006.

[49] Antoine Fraboulet and Tanguy Risset. Efficient on-chip communications for data-flow ips. In *ASAP '04 : Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference on (ASAP'04)*, pages 293–303, Washington, DC, USA, 2004. IEEE Computer Society.

[50] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the streams-c c-to-fpga compiler : an applications perspective. In *FPGA '01 : Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 134–140, New York, NY, USA, 2001. ACM Press.

[51] D. D. Gajski and R. Kuhn. Guest editor introduction : New VLSI-tools. *IEEE Computer*, 16(12) :11–14, décembre 1983.

[52] S. Guccione, D. Levi, and P. Sundararajan. JBits : A java-based interface for reconfigurable computing. In *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference MAPLD*, 1998.

[53] Zhi Guo, Betul Buyukkurt, Walid Najjar, and Kees Vissers. Optimized generation of data-path from c codes for fpgas. In *DATE '05 : Proceedings of the conference on Design, Automation and Test in Europe*, pages 112–117, Washington, DC, USA, 2005. IEEE Computer Society.

[54] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. SPARK : a high-level synthesis framework for applying parallelizing compiler transformations. In *Intl. Conf. on VLSI Design*, pages 461–466, 2003.

[55] Manish Handa and Ranga Vemuri. A fast algorithm for finding maximal empty rectangles for dynamic FPGA placement. In *Design, Automation and Test in Europe Conference and Exhibition, DATE'04*, pages 744– 745, Vol.1, Paris, France, février 2004.

[56] Y. El Hillali. *Etude et réalisation d'un système de communication et de localisation, basé sur les techniques d'étalement de spectre aux transports guidés*. PhD thesis, University of Valenciennes, 2005.

[57] IEEE, editor. *Std 1076-1993 IEEE Standard VHDL Language, Reference Manual - Description*. Inst of Elect & Electronic, 1994.

[58] Imran Rafiq Quadri and Samy Meftali and Jean-Luc Dekeyser. An MDE Approach for Implementing Partial Dynamic Reconfiguration in FPGAs. In *16th International Conference on IP Based System-on-chip, IP'07*, Grenoble, France, décembre 2007.

[59] INRIA. DaRT short presentations and demos. http://www.lifl.fr/west/DaRTShortPresentations/, 2007.

[60] ITRS, International Technology Roadmap for Semiconductors. Design, 2005 edition. http://www.itrs.net/, 2005.

[61] Tyler J.Moeller. *Field Programmable Gate Arrays for Radar Front-End Digital Signal Processing*. PhD thesis, Massachusetts Institute of Technology, mai 1999.

[62] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of the Model Transformation in Practice Workshop*, Montego Bay, Jamaica, octobre 2005.

[63] L. Kaouane, M. Akil, Y. Sorel, and T. Grandpierre. From algorithm graph specification to automatic synthesis of FPGA circuit : a seamless flow of graph transformations. In *13th international conference on Field-Programmable Logic and Applications, FPL'03*, Lisbon, Portugal, septembre 2003.

[64] Linda Kaouane, Mohamed Akil, Thierry Grandpierre, and Yves Sorel. A methodology to implement real-time applications onto reconfigurable circuits. *J. Supercomput.*, 30(3) :283–301, 2004.

[65] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3) :563–590, juillet 1967.

[66] 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.

[67] Ouassila Labbani, Éric Rutten, and Jean-Luc Dekeyser. Safe design methodology for an intelligent cruise control system with GPS. In *64th IEEE Vehicular Technology Conference (VTC 2006)*, Montréal, Québec, Canada, septembre 2006.

[68] Loïc Lagadec, Dominique Lavenier, Erwan Fabiani, and Bernard Pottier. Placing, routing, and editing virtual FPGAs. In *11th International Conference on Field-Programmable Logic and Applications, FPL'01*, pages 357–366, Belfast, Northern Ireland, UK, août 2001.

[69] David Lau, Orion Pritchard, and Philippe Molson. Automated generation of hardware accelerators with direct memory access from ansi/iso standard c functions. In *FCCM '06 : Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 45–56, Washington, DC, USA, 2006. IEEE Computer Society.

[70] Sébastien Le Beux, Vincent Gagne, El Mostapha Aboulhamid, Philippe Marquet, and Jean-Luc Dekeyser. Hardware/software exploration for an anti-collision radar system. In *The 49th IEEE International Midwest Symposium on Circuits and Systems*, San Juan, Puerto Rico, août 2006.

[71] Sébastien Le Beux and Loic Lagadec. Madeo, une approche MDA pour la programmation et la synthèse d'architectures reconfigurables. In *Sympa'2005*, pages 1–12, Le Croisic, France, avril 2005.

[72] Sébastien Le Beux, Philippe Marquet, Ouassila Labbani, and Jean-Luc Dekeyser. FPGA implementation of embedded cruise control and anti-collision radar. In *DSD'2006, 9th Euromicro conference on digital system design*, Dubrovnik, Croatia, août 2006.

[73] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers, Special Issue on Application Specific Hardware Design*, avril 2003.

[74] Patricia Le Moenner, Laurent Perraudeau, Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset. Generating regular arithmetic circuits with ALPHARD. Technical report, IRISA, mars 1996.

[75] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The alpha language and its use for the design of systolic arrays. *The Journal of VLSI Signal Processing*, 3(3) :173–182, septembre 1991.

[76] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, janvier 1987.

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

[78] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 36(1) :24–35, janvier 1987.

[79] Roberto Manduchi, Guido M.Cortelazzo, and Gian Antonio Mian. Multistage sampling structure conversion of video signals. *IEEE Transactions on circuits and systems for video technology*, 1993.

[80] Philippe Marquet, Simon Duquennoy, Sébastien Le Beux, Samy Meftali, and Jean-Luc Dekeyser. Massively parallel processing on a chip. In *ACM Int'l Conf. on Computing Frontiers*, Ischia, Italy, mai 2007.

[81] MathWorks. Synplify DSP, 2007. http://www.synplicity.com/products/synplifydsp/.

[82] Christophe Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, Université de Rennes I, décembre 1989.

[83] William E. McUmber and Betty H. C. Cheng. Uml-based analysis of embedded systems using a mapping to vhdl. In *HASE '99 : The 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 56–63, Washington, DC, USA, 1999. IEEE Computer Society.

[84] 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.

[85] MORPHEUS project. Multi-purpose dynamically reconfigurable platform for intensive heterogeneous processing. http://www.morpheus-ist.org/.

[86] 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.

[87] P.K. Murthy and Edward A. Lee. A generalization of multidimensional synchronous dataflow to arbitrary sampling lattices. Technical Report UCB/ERL M95/59, EECS Department, University of California, Berkeley, mars 1995.

[88] Praveen K. Murthy and Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8) :2064–2079, août 2002.

[89] Object Management Group, Inc., editor. *MOF 2.0 Core Final Adopted Specification*. http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 2003.

[90] Object Management Group, Inc., editor. *UML 2 Infrastructure (Final Adopted Specifcation)*. http://www.omg.org/cgi-bin/doc?ptc/2003-09-15, septembre 2003.

[91] 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.

[92] Object Management Group, Inc., editor. *Final Adopted OMG SysML Specification*. http://www.omg.org/cgi-bin/doc?ptc/06-0504, mai 2006.

[93] Object Management Group, Inc. MOF Query / Views / Transformations. http://www.omg.org/docs/ptc/07-07-07.pdf, juillet 2007. OMG paper.

[94] Papyrus. Papyrus UML web site, 2007. http://www.papyrusuml.org/.

[95] Abraham Peled and Bede Liu. A new hardware realization of digital filters. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 22, pages 456–462, décembre 1978.

[96] Éric Piel. *Ordonnancement de systèmes parallèles temps-réel, De la modélisation à la mise en œuvre par l'ingénierie dirigée par les modèles*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, décembre 2007.

[97] Planet MDE. Model Driven Engineering, 2007. http://planetmde.org.

[98] Polylib - a library of polyhedral functions. http://icps.u-strasbg.fr/polylib/.

[99] Project INRIA AOSTE. SynDEx-IC home page. http://www.inria.fr/rapportsactivite/RA2005/aoste/uid43.html#uid43.

[100] Project Inria-CNRS COSI. ALPHA home page. http://www.irisa.fr/cosi/ALPHA/.

[101] ProMarte partners. UML Profile for MARTE, Beta 1. http://www.omg.org/cgi-bin/doc?ptc/2007-08-04, août 2007.

[102] Javier Resano, Daniel Mozos, Diederik Verkest, and Francky Catthoor. A reconfiguration manager for dynamically reconfigurable hardware. *IEEE Des. Test*, 22(5) :452–460, 2005.

[103] Robert Rinker, Margaret Carter, Amitkumar Patel, Monica Chawathe, Charlie Ross, Jeffrey Hammes, Walid A. Najjar, and Wim Böhm. An automated process for compiling dataflow graphs into reconfigurable hardware. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1) :130–139, 2001.

[104] L. Rioux, T. Saunier, S. Gerard, A. Radermacher, R. de Simone, T. Gautier, Y. Sorel, J. Forget, J.-L. Dekeyser, A. Cuccuru, C. Dumoulin, and C. Andre. MARTE : A new profile RFP for the modeling and analysis of real-time embedded systems. In *UML-SoC'05, DAC 2005 Workshop UML for SoC Design*, Anaheim, CA, juin 2005.

[105] Tanguy Risset. *Contribution à la compilation de nids de boucles sur silicium*. Habilitation à diriger des recherches, Université de Rennes 1, octobre 2000.

[106] 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.

[107] Simulink, MathWorks. Hdl coder. http://www.mathworks.com/products/slhdlcoder/.

[108] Love Singhal and Elaheh Bozorgzadeh. Novel multi-layer floorplanning for heterogeneous fpgas. In *17th International Conference on Field-Programmable Logic and Applications, FPL'07*, pages 613–616, Amsterdman, Netherlands, août 2007.

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

[110] 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.

[111] 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.

[112] Jesús Tabero, Julio Septién, Hortensia Mecha, and Daniel Mozos. Task placement heuristic based on 3D-adjacency and look-ahead in reconfigurable systems. In *11th Asia and South Pacific Design Automation Conference, ASP-DAC 2006*, pages 384–389, Yokohama, Japan, janvier 2006.

[113] Julien Taillard, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. A Graphical Framework for High Performance Computing using an MDE Approach. In *16th Euromicro International Conference on Parallel, Distributed and network-based Processing*, Toulouse, France, février 2008.

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

[115] West Team. Tools for array-ol. http://www2.lifl.fr/west/aoltools/.

[116] Russell Tessier. Fast placement approaches for FPGAs. *ACM Trans. Des. Autom. Electron. Syst.*, 7(2) :284–305, 2002.

[117] Russell Tessier and Wayne Burleson. Reconfigurable computing for digital signal processing : A survey. *The Journal of VLSI Signal Processing*, 28(1-2) :7–27, décembre 1999.

[118] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, fourth edition, mai 1998.

[119] INRIA Triskell. Kermeta. http://www.kermeta.org/.

[120] J. K. Tugnait. On time delay estimation with unknown spatially correlated gaussian noise using fourth-order cumulants and cross cumulants. In *IEEE transaction on signal processing*, volume 39, pages 1258–1267, juin 1991.

[121] J. K. Tugnait. Time delay estimation with unknown spatially correlated gaussian noise. In *IEEE transaction on signal processing*, volume 42, pages 549–558, février 1993.

[122] San Diego University Of California. The SPARK high level synthesis methodology, 2003. http://mesl.ucsd.edu/spark/methodology.shtml.

[123] H. Walder, C. Steiger, and M. Platzner. Fast online task placement on fpgas : Free space partitioning and 2d-hashing. In *Reconfigurable Architectures Workshop (RAW)*, 2003.

[124] Michael C. Williamson. *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications*. PhD thesis, EECS Department, University of California, Berkeley, 1998.

[125] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF : An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12) :31–37, 1994.

[126] S.K. Wood, D.H. Akehurst, W.G.J. Howells, and K.D. McDonald-Maier. Mapping the design of repetitive structures onto VHDL. In *International ModEasy'07 Workshop*, Barcelona, Spain, septembre 2007. http://www2.lifl.fr/modeasy/workshop.html.

[127] Xilinx. Virtex-4 Family Overview, février 2006. http://direct.xilinx.com/bvdocs/publications/ds112.pdf.

[128] 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.

[129] Huafeng Yu, Abdoulaye Gamatié, Éric Rutten, Pierre Boulet, and Jean-Luc Dekeyser. Vers des transformations d'applications à parallélisme de données en équations synchrones. In *9e édition de SYMPosium en Architectures nouvelles de machines (SympA'2006)*, Perpignan, France, octobre 2006.

[130] J. Zaidouni, A. Rivenq, S. Niar, Y. Elhillali, and L. Sakkila. New Time Delay Estimators for Coded Anti-Collision Radar. In *International ModEasy'07 Workshop*, Barcelona, Spain, septembre 2007. http://www2.lifl.fr/modeasy/workshop.html.

Annexe A

Interfaçage des accélérateurs

Interfaçage des accélérateurs

Les accélérateurs que nous générons peuvent être utilisés dans des environnements dédiés ou dans des SoCs. Selon le contexte choisi, l'interfaçage des accélérateurs est géré différemment :

- lorsqu'un accélérateur est directement connecté à des capteurs ou à des actionneurs, les ports de l'accélérateur sont reliés sans aucun protocole aux ports de ces périphériques [70, 72]. Ce cas est illustré dans le chapitre 3 lors des connexions entre le générateur d'images et le filtre d'images, ainsi que dans le chapitre 7 lors de la connexion de l'accélérateur qui exécute l'application de détection d'obstacles avec un capteur représentant le convertisseur analogique numérique. La connexion directe d'un accélérateur avec un capteur est idéale, car le système complet peut être cadencé par le capteur. Cette première solution d'interfaçage est gratuite en ressource du FPGA puisqu'elle ne nécessite que de simples connexions ;
- dans un contexte de SoC, les ports de l'accélérateur sont connectés un composant de communication de type bus ou réseau d'interconnexion sur puce : ces connexions nécessitent des interfaces et un protocole de communication. Nos travaux ne supportent actuellement pas la génération de ces interfaces, les points clés qui permettront de les prendre en considération dans des extensions sont détaillés par la suite.

Les tableaux de données consommés et produits par les accélérateurs sont souvent composés de plusieurs données et nécessitent donc plusieurs accès mémoire. Ainsi, l'envoi complet des tableaux de données (et réciproquement la réception pour les tableaux produits) est réalisé en plusieurs cycles, il est donc nécessaire de stocker les données au sein de l'accélérateur. Nous préconisons de gérer ce stockage de données dans l'interface de l'accélérateur. Il est à noter que des travaux similaires ont permis de connecter des IPs à flot de données générés depuis l'environnement MMALPHA [100] avec un bus ou un réseau d'interconnexion sur puce [49]. Les résultats de ces travaux sont à prendre en considération pour le développement des interfaces pour nos accélérateurs.

Afin de générer automatiquement de telles interfaces dans notre flot de conception, nous considérons l'association dans le métamodèle Deployed. La figure A.1 représente le placement de l'application filtre d'images sur un SoC composé de l'accélérateur *acc*, de la mémoire de données *m* et d'un processeur *proc*. Ces unités fonctionnelles sont connectées par un bus. Les tableaux de données *pg* et *pr* de l'application sont placés sur la mémoire de données, la tâche *tasks* de l'application est placée sur l'accélérateur.

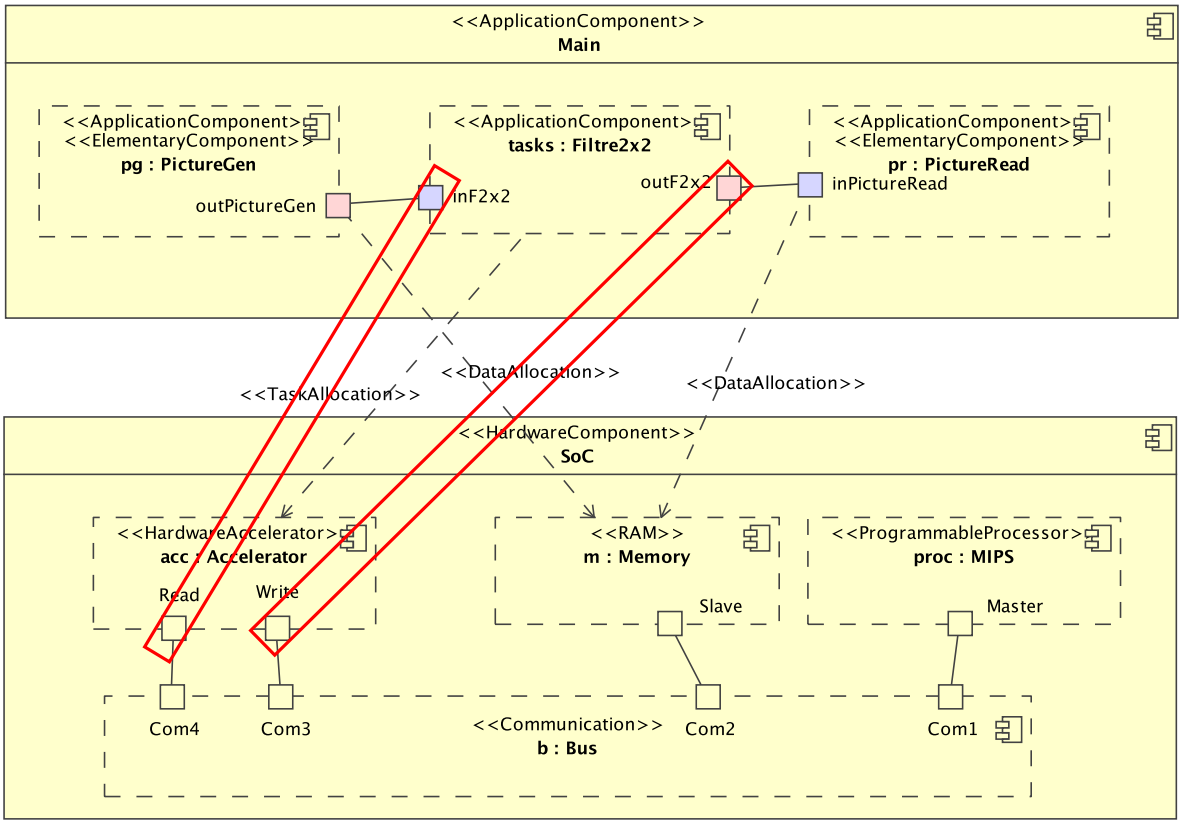


FIG. A.1: Placement d'une tâche sur un accélérateur dans un SoC. La tâche *tasks* est placée sur l'accélérateur *acc* et les tableaux de données sont placés en mémoires. Les rectangles illustrent l'information manquante.

Après compilation de ce modèle vers un modèle RTL, les interfaces du composant généré depuis *acc* doivent gérer d'un côté la communication avec le bus et de l'autre l'envoi de données à l'accélérateur généré depuis la tâche *tasks*. Cependant, il n'existe aucune relation entre les ports de l'accélérateur fonctionnel *acc* dans le SoC et les ports de l'accélérateur matériel que nous générons (accélérateur issu de *tasks*). Il est donc nécessaire d'étendre le mécanisme de placement de données de manière à permettre leur placement sur des ports de l'architecture. Nous illustrons cette extension au travers des rectangles représentés sur la figure A.1. Cette information permettra de réaliser les connexions manquantes et de déterminer les caractéristiques des interfaces en matière de stockage de données dans le métamodèle RTL.

Annexe B

Code VHDL généré pour le filtre d'images

Fichier TilerOUT1feffac.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

library work;
use work.userlibrary.all;

ENTITY TilerOUT1feffac IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
OutputTilerOUT1feffac : OUT TABLE_TYPE_2_2_INTEGERrange0to255;
InputTilerOUT1feffac : IN TABLE_TYPE_2_2_INTEGERrange0to255);
END TilerOUT1feffac;

ARCHITECTURE archiTilerOUT1feffac OF TilerOUT1feffac IS

BEGIN
OutputTilerOUT1feffac(1)(1) <= InputTilerOUT1feffac(1)(1);
OutputTilerOUT1feffac(1)(2) <= InputTilerOUT1feffac(1)(2);
OutputTilerOUT1feffac(2)(1) <= InputTilerOUT1feffac(2)(1);
OutputTilerOUT1feffac(2)(2) <= InputTilerOUT1feffac(2)(2);


END archiTilerOUT1feffac;
```

Fichier userlibrary.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
PACKAGE userlibrary IS

TYPE TABLE_TYPE_4_std_logic IS ARRAY(1 to 4) of std_logic;
TYPE TABLE_TYPE_4_4_std_logic IS ARRAY(1 to 4) of TABLE_TYPE_4_std_logic;
COMPONENT PictureGen IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
outPictureGen : OUT TABLE_TYPE_4_4_std_logic);
END COMPONENT;

TYPE TABLE_TYPE_3_INTEGERrange0to255 IS ARRAY(1 to 3) of INTEGER range 0 to 255;
TYPE TABLE_TYPE_3_3_INTEGERrange0to255 IS ARRAY(1 to 3) of TABLE_TYPE_3_INTEGERrange0to255;
COMPONENT TE IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
MotifSource : IN TABLE_TYPE_3_3_INTEGERrange0to255;
MotifProduit : OUT INTEGER range 0 to 255);
END COMPONENT;

TYPE TABLE_TYPE_2_std_logic IS ARRAY(1 to 2) of std_logic;
TYPE TABLE_TYPE_2_2_std_logic IS ARRAY(1 to 2) of TABLE_TYPE_2_std_logic;
COMPONENT PictureRead IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
inPictureRead : IN TABLE_TYPE_2_2_std_logic);
END COMPONENT;

COMPONENT Main IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
outPictureGen : OUT TABLE_TYPE_2_2_std_logic);
END COMPONENT;

TYPE TABLE_TYPE_4_INTEGERrange0to255 IS ARRAY(1 to 4) of INTEGER range 0 to 255;
TYPE TABLE_TYPE_4_4_INTEGERrange0to255 IS ARRAY(1 to 4) of TABLE_TYPE_4_INTEGERrange0to255;
TYPE TABLE_TYPE_2_INTEGERrange0to255 IS ARRAY(1 to 2) of INTEGER range 0 to 255;
TYPE TABLE_TYPE_2_2_INTEGERrange0to255 IS ARRAY(1 to 2) of TABLE_TYPE_2_INTEGERrange0to255;
COMPONENT FiltreImage IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
ImageSource : IN TABLE_TYPE_4_4_INTEGERrange0to255;
ImageProduite : OUT TABLE_TYPE_2_2_INTEGERrange0to255);
END COMPONENT;

TYPE TABLE_TYPE_2_3_3_INTEGERrange0to255 IS ARRAY(1 to 2) of TABLE_TYPE_3_3_INTEGERrange0to255;
TYPE TABLE_TYPE_2_2_3_3_INTEGERrange0to255 IS ARRAY(1 to 2) of TABLE_TYPE_2_3_3_INTEGERrange0to255;
COMPONENT TilerIN4af41d IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
InputTilerIN4af41d : IN TABLE_TYPE_4_4_INTEGERrange0to255;
OutputTilerIN4af41d : OUT TABLE_TYPE_2_2_3_3_INTEGERrange0to255);
END COMPONENT;

COMPONENT TilerOUT1feffac IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
OutputTilerOUT1feffac : OUT TABLE_TYPE_2_2_INTEGERrange0to255;
InputTilerOUT1feffac : IN TABLE_TYPE_2_2_INTEGERrange0to255);
END COMPONENT;
END userlibrary;
```

Fichier TilerIN4af41d.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

library work;
use work.userlibrary.all;

ENTITY TilerIN4af41d IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
InputTilerIN4af41d : IN TABLE_TYPE_4_4_INTEGERrange0to255;
OutputTilerIN4af41d : OUT TABLE_TYPE_2_2_3_3_INTEGERrange0to255);
END TilerIN4af41d;


ARCHITECTURE archiTilerIN4af41d OF TilerIN4af41d IS


BEGIN

OutputTilerIN4af41d(1)(1)(1)(1) <= InputTilerIN4af41d(1)(1);
OutputTilerIN4af41d(1)(1)(1)(2) <= InputTilerIN4af41d(1)(2);
OutputTilerIN4af41d(1)(1)(1)(3) <= InputTilerIN4af41d(1)(3);
OutputTilerIN4af41d(1)(1)(2)(1) <= InputTilerIN4af41d(2)(1);
OutputTilerIN4af41d(1)(1)(2)(2) <= InputTilerIN4af41d(2)(2);
OutputTilerIN4af41d(1)(1)(2)(3) <= InputTilerIN4af41d(2)(3);
OutputTilerIN4af41d(1)(1)(3)(1) <= InputTilerIN4af41d(3)(1);
OutputTilerIN4af41d(1)(1)(3)(2) <= InputTilerIN4af41d(3)(2);
OutputTilerIN4af41d(1)(1)(3)(3) <= InputTilerIN4af41d(3)(3);
OutputTilerIN4af41d(1)(2)(1)(1) <= InputTilerIN4af41d(1)(2);
OutputTilerIN4af41d(1)(2)(1)(2) <= InputTilerIN4af41d(1)(3);
OutputTilerIN4af41d(1)(2)(1)(3) <= InputTilerIN4af41d(1)(4);
OutputTilerIN4af41d(1)(2)(2)(1) <= InputTilerIN4af41d(2)(2);
OutputTilerIN4af41d(1)(2)(2)(2) <= InputTilerIN4af41d(2)(3);
OutputTilerIN4af41d(1)(2)(2)(3) <= InputTilerIN4af41d(2)(4);
OutputTilerIN4af41d(1)(2)(3)(1) <= InputTilerIN4af41d(3)(2);
OutputTilerIN4af41d(1)(2)(3)(2) <= InputTilerIN4af41d(3)(3);
OutputTilerIN4af41d(1)(2)(3)(3) <= InputTilerIN4af41d(3)(4);
OutputTilerIN4af41d(2)(1)(1)(1) <= InputTilerIN4af41d(2)(1);
OutputTilerIN4af41d(2)(1)(1)(2) <= InputTilerIN4af41d(2)(2);
OutputTilerIN4af41d(2)(1)(1)(3) <= InputTilerIN4af41d(2)(3);
OutputTilerIN4af41d(2)(1)(2)(1) <= InputTilerIN4af41d(3)(1);
OutputTilerIN4af41d(2)(1)(2)(2) <= InputTilerIN4af41d(3)(2);
OutputTilerIN4af41d(2)(1)(2)(3) <= InputTilerIN4af41d(3)(3);
OutputTilerIN4af41d(2)(1)(3)(1) <= InputTilerIN4af41d(4)(1);
OutputTilerIN4af41d(2)(1)(3)(2) <= InputTilerIN4af41d(4)(2);
OutputTilerIN4af41d(2)(1)(3)(3) <= InputTilerIN4af41d(4)(3);
OutputTilerIN4af41d(2)(2)(1)(1) <= InputTilerIN4af41d(2)(2);
OutputTilerIN4af41d(2)(2)(1)(2) <= InputTilerIN4af41d(2)(3);
OutputTilerIN4af41d(2)(2)(1)(3) <= InputTilerIN4af41d(2)(4);
OutputTilerIN4af41d(2)(2)(2)(1) <= InputTilerIN4af41d(3)(2);
OutputTilerIN4af41d(2)(2)(2)(2) <= InputTilerIN4af41d(3)(3);
OutputTilerIN4af41d(2)(2)(2)(3) <= InputTilerIN4af41d(3)(4);
OutputTilerIN4af41d(2)(2)(3)(1) <= InputTilerIN4af41d(4)(2);
OutputTilerIN4af41d(2)(2)(3)(2) <= InputTilerIN4af41d(4)(3);
OutputTilerIN4af41d(2)(2)(3)(3) <= InputTilerIN4af41d(4)(4);


END archiTilerIN4af41d;
```

Fichier TE.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

library work;
use work.userlibrary.all;

ENTITY TE IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
MotifSource : IN TABLE_TYPE_3_3_INTEGERrange0to255;
MotifProduit : OUT INTEGER range 0 to 255);
END TE;

ARCHITECTURE archiTE OF TE IS
 -- this elementary task is not deployed

BEGIN
   -- this elementary task is not deployed


END archiTE;



library IEEE;
use IEEE.STD_LOGIC_1164.all;

library work;
use work.userlibrary.all;

ENTITY PictureRead IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
inPictureRead : IN TABLE_TYPE_2_2_std_logic);
END PictureRead;

ARCHITECTURE archiPictureRead OF PictureRead IS
 -- this elementary task is not deployed

BEGIN
   -- this elementary task is not deployed


END archiPictureRead;
```

Fichier PictureGen.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

library work;
use work.userlibrary.all;

ENTITY PictureGen IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
outPictureGen : OUT TABLE_TYPE_4_4_std_logic);
END PictureGen;

ARCHITECTURE archiPictureGen OF PictureGen IS
 -- this elementary task is not deployed

BEGIN
   -- this elementary task is not deployed


END archiPictureGen;
```

Fichier Main.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

library work;
use work.userlibrary.all;

ENTITY Main IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
outPictureGen : OUT TABLE_TYPE_2_2_std_logic);
END Main;

ARCHITECTURE archiMain OF Main IS

--SIGNAL DECLARATION

SIGNAL Connector_1bc7636 : TABLE_TYPE_2_2_INTEGERrange0to255;
SIGNAL Connector_47193 : TABLE_TYPE_4_4_std_logic;
SIGNAL Connector_36b673 : TABLE_TYPE_2_2_INTEGERrange0to255;
SIGNAL clkConnectortasks : STD_LOGIC;
SIGNAL razConnectortasks : STD_LOGIC;
SIGNAL clkConnectorpr : STD_LOGIC;
SIGNAL razConnectorpr : STD_LOGIC;
SIGNAL clkConnectorpg : STD_LOGIC;
SIGNAL razConnectorpg : STD_LOGIC;


--BEGIN DESCRIPTION

BEGIN

--SIGNAL AFFECTATION

outPictureGen <= Connector_1bc7636;
Connector_36b673 <= Connector_1bc7636;
clkConnectortasks <= clk;
razConnectortasks <= raz;
clkConnectorpr <= clk;
razConnectorpr <= raz;
clkConnectorpg <= clk;
razConnectorpg <= raz;


--COMPONENT MAPPING

instanceOfFiltreImage : FiltreImage
port map(
clk => clkConnectortasks,
raz => razConnectortasks,
ImageProduite =>Connector_1bc7636,
ImageSource =>Connector_47193);

instanceOfPictureRead : PictureRead
port map(
clk => clkConnectorpr,
raz => razConnectorpr,
inPictureRead =>Connector_36b673);

instanceOfPictureGen : PictureGen
port map(
clk => clkConnectorpg,
raz => razConnectorpg,
outPictureGen =>Connector_47193);

END archiMain;
```

Fichier FiltreImage.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

library work;
use work.userlibrary.all;

ENTITY FiltreImage IS
PORT
(
clk : IN STD_LOGIC;
raz : IN STD_LOGIC;
ImageSource : IN TABLE_TYPE_4_4_INTEGERrange0to255;
ImageProduite : OUT TABLE_TYPE_2_2_INTEGERrange0to255);
END FiltreImage;

ARCHITECTURE archiFiltreImage OF FiltreImage IS

--SIGNAL DECLARATION

SIGNAL Connector_4af41d : TABLE_TYPE_4_4_INTEGERrange0to255;
SIGNAL RepetitionConnector_4af41d : TABLE_TYPE_2_2_3_3_INTEGERrange0to255;
SIGNAL clkConnectorInstanceTilerIN4af41d : STD_LOGIC;
SIGNAL razConnectorInstanceTilerIN4af41d : STD_LOGIC;
SIGNAL Connector_1feffac : TABLE_TYPE_2_2_INTEGERrange0to255;
SIGNAL RepetitionConnector_1feffac : TABLE_TYPE_2_2_INTEGERrange0to255;
SIGNAL clkConnectorInstanceTilerOUT1feffac : STD_LOGIC;
SIGNAL razConnectorInstanceTilerOUT1feffac : STD_LOGIC;
SIGNAL clkConnectortache : STD_LOGIC;
SIGNAL razConnectortache : STD_LOGIC;

--BEGIN DESCRIPTION
BEGIN

--SIGNAL AFFECTATION
Connector_4af41d <= ImageSource;
-- the signal RepetitionConnector_4af41d is connected within component instance
clkConnectorInstanceTilerIN4af41d <= clk;
razConnectorInstanceTilerIN4af41d <= raz;
ImageProduite <= Connector_1feffac;
-- the signal RepetitionConnector_1feffac is connected within component instance
clkConnectorInstanceTilerOUT1feffac <= clk;
razConnectorInstanceTilerOUT1feffac <= raz;
clkConnectortache <= clk;
razConnectortache <= raz;

--COMPONENT MAPPING

instanceOfTilerIN4af41d : TilerIN4af41d
port map(
clk => clkConnectorInstanceTilerIN4af41d,
raz => razConnectorInstanceTilerIN4af41d,
InputTilerIN4af41d =>Connector_4af41d,
OutputTilerIN4af41d =>RepetitionConnector_4af41d);

instanceOfTilerOUT1feffac : TilerOUT1feffac
port map(
clk => clkConnectorInstanceTilerOUT1feffac,
raz => razConnectorInstanceTilerOUT1feffac,
OutputTilerOUT1feffac =>Connector_1feffac,
InputTilerOUT1feffac =>RepetitionConnector_1feffac);

genit0 : for it0 in 1 to 2 generate
genit1 : for it1 in 1 to 2 generate

instanceOfTE : TE
port map(
clk => clkConnectortache,
raz => razConnectortache,
MotifSource =>RepetitionConnector_4af41d(it0)(it1),
MotifProduit =>RepetitionConnector_1feffac(it0)(it1));

end generate;
end generate;

END archiFiltreImage;
```

Annexe C

Fonction de précalcul des tilers d'entrée

```
package blackbox;

import java.util.List;

public class HWInputTilerCreation {

  public static void resolveTilingDescription(HW_InputTiler hwit, HW_InputPort hwip, HW_OutputPort hwop, Tiler tiler)
  {
    List<HW_SubConnector> malist = null;

    TilingDescription td= tiler.getTiling();
    IntVector origin= td.getOrigin();
    Matrix paving=td.getPaving();
    Matrix fitting=td.getFitting();

    HW_Shape hwRepetitionSpace = hwit.getRepetitionSpace();
    HW_Shape hwPatternShape = hwit.getPatternShape();

    int inputArraySize = hwip.getDim().getValue().size();

    int RepetitionSpaceSize = hwRepetitionSpace.getValue().size();
    int patternSize = hwPatternShape.getValue().size();

    int[] iteratorRepetitionSpace=new int[RepetitionSpaceSize];
    int[] iteratorPatternShape=new int[patternSize];

    do{

      HW_SubConnector hwscBACK = HardwareacceleratorFactory.eINSTANCE.createHW_SubConnector();
      HW_Shape hwDelayBACK = HardwareacceleratorFactory.eINSTANCE.createHW_Shape();
      hwDelayBACK.getValue().add((Integer)0);
      HW_Shape hwDelayTypeBACK = HardwareacceleratorFactory.eINSTANCE.createHW_Shape();
      hwDelayTypeBACK.getValue().add((Integer)1);
      HW_Shape hwsSourceBACK = HardwareacceleratorFactory.eINSTANCE.createHW_Shape();

      boolean testShiftRegisterNecessaryWARNING = false;

      do{

        HW_SubConnector hwsc = HardwareacceleratorFactory.eINSTANCE.createHW_SubConnector();
        hwsc.setTilerOwner(hwit);
        hwit.getSubConnector().add(hwsc);

        HW_Shape hwsTarget = HardwareacceleratorFactory.eINSTANCE.createHW_Shape();

        for (int i=0; i< RepetitionSpaceSize; i++)
          hwsTarget.getValue().add(iteratorRepetitionSpace[i]);

        for (int i=0; i< patternSize; i++)
          hwsTarget.getValue().add(iteratorPatternShape[i]);

        hwsc.setTargetIndex(hwsTarget);
        hwsc.setTargetPort(hwop);
```

```
    hwop.getConnect().add(hwsc);

    //index of the source port
HW_Shape hwsSourceTemp = HardwareacceleratorFactory.eINSTANCE.createHW_Shape();
HW_Shape hwsSource = HardwareacceleratorFactory.eINSTANCE.createHW_Shape();

//compute coordinate of the element in the input array
computeArrayCoord(
    inputArraySize,
    iteratorRepetitionSpace,
    iteratorPatternShape,
    hwsSourceTemp,
    RepetitionSpaceSize,
    patternSize,
    fitting,
    origin,
    paving);

hwsSource.getValue().addAll(hwsSourceTemp.getValue());
boolean testShiftRegisterNecessary = false;

//the delay detected
HW_Shape hwDelay = HardwareacceleratorFactory.eINSTANCE.createHW_Shape();
HW_Shape hwDelayType = HardwareacceleratorFactory.eINSTANCE.createHW_Shape();

for (int i=0; i<inputArraySize;i++)
{
  //detect if their is a dimension -1 (infinite) in the input array
  if (((Integer) hwip.getDim().getValue().get(i)) ==-1)
  {
    //detect if for this input array their is a data consume on this dimension
    if (((Integer) hwsSourceTemp.getValue().get(i))!=0)
    {
      //a shift register has been detected
      //flag are created and solved later
      hwsSource.getValue().set(i, 0);
      testShiftRegisterNecessary = true;
      testShiftRegisterNecessaryWARNING = true;
      hwDelay.getValue().add((Integer)(hwsSourceTemp.getValue().get(i)));
      hwDelayType.getValue().add((Integer)(hwsSourceTemp.getValue().get(i))+1);
    }
    else
    {
      //information kept in case it's the first element of the shift register to create
      hwsSourceBACK = hwsSource;
      hwsSourceBACK.getValue().set(i, 0);
      hwscBACK = hwsc;
    }
  }

}

//manage the creation of the shift register, delay, signal, etc.
if (testShiftRegisterNecessary == false)
{
  //the classical situation, where no data recovery has been detected
  hwsc.setSourceIndex(hwsSource);
  hwsc.setSourcePort(hwip);
  hwip.getConnect().add(hwsc);
}
else
{
  //creation of the shift register
  ShiftRegisterCreation(
      hwit,
      hwip,
      hwsSource,
      hwsc,
      hwDelay,
      hwop,
      hwDelayType);

  //try to delete the initial conector when a shift register has been detected
  //means that a port has been linked to "by mistake"
  if(hwscBACK.getSourcePort() != null){

  //deleting the subConnector of the context (was a mistake like)
  hwscBACK.setSourcePort(null);
  hwscBACK.setSourceIndex(null);
```

```
      hwip.getConnect().remove(hwscBACK);

      //writing the good context for the first element of the shift register
      ShiftRegisterCreation(
          hwit,
          hwip,
          hwsSourceBACK,
          hwscBACK,
          hwDelayBACK,
          hwop,
          hwDelayTypeBACK);
      }
    }

  }while (HWInputTilerCreation.nextIteration(iteratorPatternShape,(List) hwPatternShape.getValue()));

  if (testShiftRegisterNecessaryWARNING == true)
  {
    System.err.println("WARNING: a shift register has been created in the input tiler "+hwit.getName());
    }

  }while (HWInputTilerCreation.nextIteration(iteratorRepetitionSpace,(List) hwRepetitionSpace.getValue()));
}

private static void computeArrayCoord(
    int inputArraySize,
    int[] repetitionSpace,
    int[] patternShape,
    HW_Shape dataInput,
    int RepetitionSpaceSize,
    int PatternSize,
    Matrix fittingMatrix,
    IntVector origin,
    Matrix pavingMatrix
    )
{
for (int i=0; i < inputArraySize; i++)
  {
  dataInput.getValue().add(i, origin.getValue().get(i));

  for (int j=0; j < RepetitionSpaceSize; j++)
  {
    Integer val = repetitionSpace[j] * ((List<Integer>) ((IntVector)pavingMatrix.getVector().get(j)).getValue()).get(i);
    dataInput.getValue().set(i, (((Integer) dataInput.getValue().get(i))+val));
  }
  for (int j=0; j < PatternSize; j++)
  {
    Integer val = patternShape[j] * ((List<Integer>) ((IntVector)fittingMatrix.getVector().get(j)).getValue()).get(i);
    dataInput.getValue().set(i, (((Integer) dataInput.getValue().get(i))+val));
  }
  }
}


public static boolean nextIteration(int[] iterator,List value)
{
  boolean add=false;
  int index=value.size()-1;
  while ((! add) && (index!=-1))
  {
    Integer maxVal=(Integer) value.get(index);
    if (iterator[index]+1<maxVal)
    {
      iterator[index]+=1;
      add=true;
    }
    else{
      iterator[index]=0;
    }
    index--;
  }
  if (!add)
  {
    return false;
  }
  else{
    return true;
  }
}
```

```
public static void ShiftRegisterCreation(
    HW_InputTiler hwit,
    HW_InputPort hwip,
    HW_Shape hwsSource,
    HW_SubConnector hwsc,
    HW_Shape hwDelay,
    HW_OutputPort hwop,
    HW_Shape hwDelayType
    )
{

  boolean testSignalName = false;

  String sDelayedConnector = "delayedSubConnector"+hwip.getName();
for (Integer i : (List<Integer>) hwsSource.getValue())
  {
  sDelayedConnector+="_";
  sDelayedConnector+= i+1;
  }

String sSignal = "Signal"+hwip.getName();
for (Integer i : (List<Integer>) hwsSource.getValue())
  {
  sSignal+="_";
  sSignal+= i+1;
  }
if (!(hwit.getSignal().isEmpty()))
{
  for (HW_Signal hws : (List<HW_Signal>) (hwit.getSignal()))
  {
    //s contains the name of the declared Signal in the tiler
    //the name is associate to the input port the signal
    //is connected to, within delays.

    if (hws.getName().equals(sSignal))
    {
      hws.getConnect().add(hwsc);
      if (((Integer) hwDelay.getValue().get(0)) > ((Integer) hws.getDim().getValue().get(0)))
        {
          hws.setDim(hwDelayType);
        }
      hwsc.setSourceIndex((HW_Shape) EcoreUtil.copy(hwDelay));
      hwsc.setSourcePort(hws);
      testSignalName = true;
    }
  }

  if (testSignalName ==false)
  {
    //the name does not correspond, we create the HW_Signal
    HW_Signal hwSignal = HardwareacceleratorFactory.eINSTANCE.createHW_Signal();
    hwSignal.setName(sSignal);
    // TODO if necessary : hwSignal.setTilerSignalOwner(hwit);
    hwSignal.setOwner(hwit);
    hwit.getSignal().add(hwSignal);
    hwSignal.getConnect().add(hwsc);
    hwSignal.setType(hwop.getType());
    hwSignal.setDim(hwDelayType);
    hwsc.setSourceIndex((HW_Shape) EcoreUtil.copy(hwDelay));
    hwsc.setSourcePort(hwSignal);

    /////// the delayed connector
    HW_delayedSubConnector hwDelaySubConnector = HardwareacceleratorFactory.eINSTANCE.createHW_delayedSubConnector();
    hwDelaySubConnector.setName(sDelayedConnector);
    hwDelaySubConnector.setTilerOwner(hwit);
    hwit.getSubConnector().add(hwDelaySubConnector);
    hwip.getConnect().add(hwDelaySubConnector);
    hwSignal.getConnect().add(hwDelaySubConnector);
    hwDelaySubConnector.setType(hwop.getType());
    hwDelaySubConnector.setDim((HW_Shape) EcoreUtil.copy(hwDelayType));
    hwDelaySubConnector.setSourceIndex((HW_Shape) EcoreUtil.copy(hwsSource));
    hwDelaySubConnector.setSourcePort(hwip);
    hwDelaySubConnector.setTargetPort(hwSignal);
    hwDelaySubConnector.setDelay((Integer) hwDelay.getValue().get(0));
  }
  testSignalName = false;
  for (HW_SubConnector hwdsc : (List<HW_SubConnector>) (hwit.getSubConnector()))
  {
```

```
      if (hwdsc instanceof HW_delayedSubConnector) {

        if (hwdsc.getName().equals(sDelayedConnector))
        {
          if (((Integer) hwDelay.getValue().get(0)) > ((Integer) hwdsc.getDim().getValue().get(0)))
            {
            hwdsc.setDim((HW_Shape) EcoreUtil.copy(hwDelayType));
            ((HW_delayedSubConnector) hwdsc).setDelay((Integer) hwDelay.getValue().get(0));
            }
          testSignalName = true;
        }
      }
    }
  }

  else
  {
    // this list was, this empty means that we have to create element
    //the Signal
    HW_Signal hwSignal = HardwareacceleratorFactory.eINSTANCE.createHW_Signal();
    hwSignal.setName(sSignal);
    // TODO if necessary : hwSignal.setTilerSignalOwner(hwit);
    hwSignal.setOwner(hwit);

    hwit.getSignal().add(hwSignal);
    hwSignal.getConnect().add(hwsc);
    hwSignal.setType(hwop.getType());
    hwSignal.setDim(hwDelayType);
    hwsc.setSourceIndex((HW_Shape) EcoreUtil.copy(hwDelay));
    hwsc.setSourcePort(hwSignal);

    /////// the delayed connector
      HW_delayedSubConnector hwDelaySubConnector = HardwareacceleratorFactory.eINSTANCE.createHW_delayedSubConnector();
      hwDelaySubConnector.setName(sDelayedConnector);
      hwDelaySubConnector.setTilerOwner(hwit);
      hwit.getSubConnector().add(hwDelaySubConnector);
      hwip.getConnect().add(hwDelaySubConnector);
      hwSignal.getConnect().add(hwDelaySubConnector);
      hwDelaySubConnector.setType(hwop.getType());
      hwDelaySubConnector.setDim((HW_Shape) EcoreUtil.copy(hwDelayType));
      hwDelaySubConnector.setSourceIndex((HW_Shape) EcoreUtil.copy(hwsSource));
      hwDelaySubConnector.setSourcePort(hwip);
      hwDelaySubConnector.setTargetPort(hwSignal);
      hwDelaySubConnector.setDelay((Integer) hwDelay.getValue().get(0));
    }
  }
}
```

Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles

Résumé : Dans cette thèse, nous proposons un flot de conception pour le développement d'applications de traitement du signal systématique implémentées sur FPGA. Nous utilisons une approche Ingénierie Dirigée par les Modèles (IDM) pour la mise en œuvre de ce flot de conception, dont la spécification des applications est décrite en UML.

La première contribution de cette thèse réside dans la création d'un métamodèle isolant les concepts utilisés au niveau RTL. Ces concepts sont extraits d'implémentations matérielles dédiées de tâches à fort parallélisme de données. Par ailleurs, ce métamodèle considère la technologie d'implémentation FPGA et propose différents niveaux d'abstractions d'un même FPGA. Ces multiples niveaux d'abstractions permettent un raffinement des implémentations matérielles.

La seconde contribution est le développement d'un flot de compilation permettant la transformation d'une application modélisée à haut niveau d'abstraction (UML) vers un modèle RTL. En fonction des contraintes de surfaces disponibles (technologie FPGA), le flot de conception optimise le déroulement des boucles et le placement des tâches. Le code VHDL produit est directement simulable et synthétisable sur FPGA. À partir d'applications modélisées en UML, nous produisons automatiquement un code VHDL.

Le flot de conception proposé a été utilisé avec succès dans le cadre de sécurité automobile ; un algorithme de détection d'obstacles a été automatiquement généré depuis sa spécification UML.

Mots clés : Parallélisme, accélérateur matériel, FPGA, systèmes-sur-puce, traitement intensif de données, ingénierie dirigée par les modèles, compilation, flot de conception.

A Model Driven Engineering based design flow for systematic signal processing applications implemented on FPGA

Abstract : This PhD thesis presents a design flow for intensive signal processing applications, which are implemented on FPGA. This flow is based on Model Driven Engineering (MDE) and the specifications of applications is done in UML.

The first contribution of this thesis is a metamodel, which identify the concepts manipulated at the RTL level. These concepts are extracted from customized hardware implementations of massively parallel applications. The metamodel also take intp account the FPGA charactersitics et identify different abstraction levels of the FPGAs. These abstraction levels alow to refine the hardware implementations.

The second contribution is a design flow, which transform applications modeled at high abstraction level model (in UML) into RTL models. According to the FPGA ressources allocated, a process optimizes the hardware implementation thanks to loop transformations.

From a UML model, a VHDL code is automatically generated. The code is simulable and synthetizable on FPGA. The design flow has been successfully used in safety transport for the development an anti-collision radar system.

Keywords : Parallelism, hardware accelerator, FPGA, systems-on-chip, intensive signal processing, model-driven engineering, compilation, design flow.