No. D'ORDRE : 6201

L'UNIVERSITÉ DE PARIS–SUD
UFR SCIENTIFIQUE D'ORSAY

THÈSE

présentée pour obtenir

Le GRADE de DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS XI ORSAY

Spécialité : INFORMATIQUE
Option : ARCHITECTURES PARALLÈLES

Équipe d'accueil : Groupe ESIEE–Laboratoire A2SI
Co-encadrement : INRIA–Rocquencourt, Projet SOSSO

par

Ailton Fernando Dias

SUJET :

Contribution à l'implantation optimisée d'algorithmes bas niveau de traitement du signal et des images sur des architectures mono-FPGA à l'aide d'une méthodologie d'Adéquation Algorithme Architecture

Soutenue le 12 Juillet 2000 devant la Commission d'examen :

MM.	Daniel ETIEMBLE	Président
	Eric MARTIN	Rapporteurs
	Michel PAINDAVOINE	
	Michel ISRAËL	Examinateurs
	Mohamed AKIL	
	Christophe LAVARENNE	
	Yves SOREL	

No. D'ORDRE : 6201

L'UNIVERSITÉ DE PARIS–SUD
UFR SCIENTIFIQUE D'ORSAY

THÈSE

présentée pour obtenir

Le GRADE de DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS XI ORSAY

Spécialité : INFORMATIQUE
Option : ARCHITECTURES PARALLÈLES

Équipe d'accueil : Groupe ESIEE–Laboratoire A2SI
Co-encadrement : INRIA–Rocquencourt, Projet SOSSO

par

Ailton Fernando Dias

SUJET :

Contribution à l'implantation optimisée d'algorithmes bas niveau de traitement du signal et des images sur des architectures mono-FPGA à l'aide d'une méthodologie d'Adéquation Algorithme Architecture

Soutenue le 12 Juillet 2000 devant la Commission d'examen :

MM.	Daniel ETIEMBLE	Président
	Eric MARTIN	Rapporteurs
	Michel PAINDAVOINE	
	Michel ISRAËL	Examinateurs
	Mohamed AKIL	
	Christophe LAVARENNE	
	Yves SOREL	

À mes amours, Júnia et Pedro

Remerciements

Je remercie Daniel Etiemble, Professeur à l'Université de Paris XI Orsay, qui m'a fait l'honneur de présider le jury de cette thèse. Je remercie Éric Martin, Professeur à l'Université de Bretagne-Sud, et Michel Paindavoine, Professeur à l'Université de Bourgogne, d'avoir bien accepté la lourde charge de rapporteurs. Je remercie Michel Israël, Professeur à l'Université d'Évry, d'avoir participé du jury.

Je remercie Mohamed Akil, Professeur à l'ESIEE, et Yves Sorel, Directeur de recherche à l'INRIA–Rocquencourt, d'avoir partagé la tâche d'encadrer ce travail.

Je remercie Christophe Lavarenne, Ingénieur de Recherche à l'INRIA–Rocquencourt, pour son temps, pour ses idées, pour les discussions quelques fois dures mais toujours fructueuses.

Je tiens à témogner mon affection sincère et profonde ainsi que ma gratitude à Júnia et à Pedro pour leurs sacrifices, leur patience, leur soutien et leur compréhension.

Que mes parents, qui m'ont beaucoup appris, et qui, surtout, ont cru en moi trouvent ici l'expression de ma reconnaissance. Que Iracema et Benito trouvent ici l'expression de mes remerciements pour leur support inconditionnel.

Puisque l'occasion m'en est donnée, je veux aussi exprimer toute ma gratitude envers mes plus proches collègues Shahram Zahirazami, Christophe Lohuc, Francisco N. Bezerra, Eric Llorens, Dominique Garry, Stéphane Gailhard et envers tous les autres membres du laboratoire A2SI pour l'aide qu'ils m'ont apporté au cours de ces années.

Mes remerciements spéciaux à José Israel Vargas pour son aide dans les moments les plus critiques. Et qu'ils ne soient pas oubliés pour leur contribution, moins négligéable qu'il n'y paraît, Antônio Helano L. Ferreira, Fernando Soares Lameiras, Gustavo José Pereira, Silvestre Paiano Sobrinho et Vinício de Faria Barreto.

Ce travail a été réalisé au sein du Laboratoire A2SI du Groupe ESIEE, à Noisy-le-Grand, en collaboration avec l'équipe AAA du Projet SOSSO de l'INRIA–Rocquencourt, et il a été soutenu par la *Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior*–CAPES, Brésil (dossier 0100/95-13) et par la *Comissão Nacional de Energia Nuclear*–CNEN, Brésil (dossier 01030.001317/95).

Résumé

Ce travail décrit une méthode d'implantation optimisée d'algorithmes bas niveau de traitement du signal et des images (TSI) sur des architectures mono-FPGA à l'aide d'une méthodologie d'Adéquation Algorithme Architecture, intégrant la synthèse des chemins de données et de contrôle dans un modèle unifié.

Les algorithmes bas niveau de TSI sont caractérisés par une grande régularité et par la répétition d'un motif. Pour spécifier ces algorithmes, nous avons choisi un modèle de graphes factorisés de dépendances de données (GFDD), puisque sa sémantique est très appropriée à leur description comportementale.

Une spécification algorithmique factorisée peut avoir différentes implantations matérielles plus ou moins défactorisées. Pourtant, nous nous sommes intéressés à une implantation matérielle qui respecte les contraintes temporelles tout en minimisant l'augmentation des ressources matérielles due à la défactorisation. Nous sommes face à un problème d'optimisation sous contraintes, qui est un problème NP-complet. Pour le résoudre dans un temps acceptable, nous faisons donc appel à une heuristique de défactorisation. Pour guider cette heuristique de défactorisation, nous avons développée une méthode de caractérisation matérielle des sommets du graphe algorithmique et une méthode d'estimation de surface et de latence. Cela évite de devoir effectuer un cycle complet de conception (spécification algorithmique, optimisation par défactorisation, implantation matérielle, codage, synthèse physique, simulation et estimation) pour chaque implantation possible.

L'implantation matérielle est obtenue par traduction directe de la spécification algorithmique, en remplaçant les sommets du GFDD par les opérateurs qui les implantent. Les mécanismes de synchronisation des opérateurs synchrones sont obtenus de façon triviale à partir de l'analyse des relations entre les sommets de factorisation du graphe algorithmique. Un code VHDL structurel synthétisable peut être produit à partir du schéma logique représenté par le graphe matériel. Ce code VHDL sera fourni à des outils de CAO qui effectueront la génération des *netlists* nécessaires à la configuration des FPGA.

L'originalité de notre approche vis à vis des outils existants réside dans le fait qu'il n'y a pas de rupture entre la spécification algorithmique sous la forme d'un GFDD et l'implantation au niveau RTL.

Abstract

This work describes a methodology for optimized implementation of low level image and signal processing algorithms on single-FPGA architectures. Our methodology unifies data and control path synthesis.

Low level image and signal processing algorithms are featured by regularity and pattern repetition. We chose a data dependency factorized graph for the behavioral specification of these algorithms.

A factorized algorithmic specification can have many hardware implementations at different levels of factorization. We are interested to the implementation that satisfies real-time constraints and minimizes hardware resources. This is a NP-complete problem that requires heuristics to be solved.

We developed hardware characterization and area/time estimation models to guide the optimization heuristics. We obtain an optimized implementation without performing an entire design cycle (algorithmic specification, defactorization, implementation, coding, synthesis, simulation and estimation).

Hardware implementation is produced by direct translation of the algorithmic specification. We replace algorithmic nodes by equivalent hardware operators. The synchronization of these operators is generated in a simple way from analysis of the algorithmic nodes relations. A structural synthesizeble VHDL program can be generated from a hardware graph. A VHDL program will be used by CAD tools to netlist generation.

The main advantages of our methodology are to provide no rupture between behavioral description and RTL implementation and to unify data and control path synthesis.

Acronymes

A2SI : Algorithmes et Architectures pour Systèmes d'Information
AAA : Adéquation Algorithme-Architecture
ANSI : *American National Standards Institute*
AsGa : Arséniure de gallium
ASIC : *Application Specific Integrated Circuit*
ASIP : *Application-Specific Instruction-Set Processor*
BCS : *Binary Constraint Search*
BiCMOS : *Bipolar CMOS*
BLIF : *Berkeley Logic Interchange Format*
BSS : *Braunschweig Synthesis System*
CAN : *Controller Area Network*,
Convertisseur Analogique-Numérique
CAO : Conception Assistée par Ordinateur
CDFG : *Control Data Flow Graph*
CFG : *Control Flow Graph*
CFSM : *Co-design Finite State Machine*
CLB : *Configurable Logic Block*
CMOS : *Complementary Metal-Oxide Semiconductor*
CNA : Convertisseur Numérique-Analogique
CPU : *Central Processor Unit*
CSP : *Communicating Sequential Processes*
DAG : *Direct Acyclic Graph*
DDF : *Dynamic Data Flow*
DE : *Discret Event*
DFF : *D-type Flip Flop*
DFG : *Data Flow Graph*
DFL : *Data Flow Language*
DMA : *Direct Memory Access*
DSP : *Digital Signal Processor*

ECL : *Emitter Coupled Logic*

EDIF : *Electronic Design Interchange Format*

EPLD : *Electrically Programmable Logic Device*

ESIEE : École Supérieure d'Ingénieurs en Électrotechnique et Électronique

ETH : *Eidgenössische Technische Hochschule*

FFT : *Fast Fourier Transform*

F/G : Générateurs de fonction F et/ou G

FIFO : *First In First Out*

FIR : *Finite Impulse Response*

FPGA : *Field Programmable Gate Array*

FSM : *Finite State Machine*

FSMD : *Finite State Machine with Datapath*

GFD : Graphe Flot de Données

GFDD : Graphe Factorisé de Dépendances de Données

GRFF : Graphe de Relations entre Frontières de Factorisation

HDL : *Hardware Description Language*

HLS : *High-Level Synthesis*

INRIA : Institut National de Recherche en Informatique et en Automatique

IC : *Integrated Circuit*

IDE : *Interactive Design and Engineering*

IOB : *Input Output Block*

IP : *Intellectual Property*

ISA : *Instruction Set Architecture*

LCA : *Logic Cell Array*

MASI : Méthodologie et Architecture des Systèmes Informatiques

MCM : *Multi-Chips Module*

MEMS : *MicroElectroMechanical Systems*

MROBDD : *Multi-Reduced and Ordered Binary Decision Diagrams*

NRZ : Non-Rétour à Zéro

PAL : *Programmed Array Logic*

PCB : *Printed Circuit Board*

PLA : *Programmable Logic-Array*

PMV : Produit Matrice-Vecteur

PS : Produit Scalaire

PSM : *Program-State Machine*

RAM : *Random Access Memory*

RISC : *Reduced Instruction Set Code*

RTL : *Register Transfer Level*

SDF : *Synchronous Data Flow*

SHDL : *Structured Hardware Description Language*

SPW : *Signal Processing Worksystem*

SR : *Synchrone/Réactif*

SRAM : *Static Random Access Memory*

SYNDEX : *Synchronous Distributed Executif*

TBNI : Traitement Bas Niveau des Images

TDG : *Typed Decision Graph*

TPN : *Time Petri Net*

TSI : Traitement du Signal et des Images

VHDL : VHSIC *Hardware Description Language*

VHSIC : *Very High Speed Integrated Circuit*

VLSI : *Very Large Scale Integration*

VPM : *Virtual Processor Model*

UART : *Universal Asynchronous Receiver/Transmitter*

ULA : Unité Logique-Arithmétique

XNF : *Xilinx Netlist Format*

Table des matières

Table des figures

Liste des tableaux

LISTE DES TABLEAUX

Liste des algorithmes

LISTE DES ALGORITHMES

Introduction

> *One of the fastest expanding areas of computer exploitation is that involving applications whose prime function is* **not** *that of information processing, but which nevertheless require information processing in order to carry out their prime function. (...) These types of computer applications are generically called* **real-time** *or* **embedded** [BW97].

La complexité croissante des applications dans les domaines du contrôle-commande et du traitement du signal et des images et le coût élevé de leurs implantations imposent le développement d'outils de conception, qui intégrent l'ensemble des étapes, depuis la spécification haut niveau jusqu'à l'implantation sur une architecture-cible. Ces applications nécessitent souvent la mise en œuvre d'une implantation qui respecte les contraintes temporelles (p.ex., le temps réel) et qui minimise les ressources matérielles utilisées.

Nous nous sommes intéressés au développement d'une méthodologie d'implantation optimisée d'algorithmes bas niveau de traitement du signal et des images (TSI) sur des architectures à base de circuits reconfigurables du type FPGA (*Field Programmable Gate Array*), tout en intégrant la partie chemin de données (*data path*) et la partie chemin de contrôle (*control path*) dans un modèle unifié. Les algorithmes bas niveau de TSI (FIR – *Finite Impulse Response*, FFT – *Fast Fourier Transform*, Sobel, Deriche, etc.) présentent généralement une grande régularité et sont caractérisés par la répétition d'un motif. Cela nous a encouragé à utiliser un modèle de graphe factorisé de dépendances de données (GFDD) pour les spécifier. Ces algorithmes sont implantés de façon plus efficace sur des circuits spécifiques (ASIC – *Application Specific Integrated Circuit*) ou plus particulièrement sur des FPGA (*Field Programmable Gate Array*), puisqu'ils exigent d'une part une grande puissance de calcul et, d'autre part, la reconfigurabilité pour faire du prototypage rapide et suivre l'évolution rapide de certains produits grand public (p.ex. téléphone portable).

Dans la plupart des cas, l'amélioration des performances temporelles (augmentation de la cadence des données ou réduction de la latence) d'une implantation matérielle exige en contrepartie l'utilisation d'une plus grande quantité de ressources matérielles, de façon à exploiter le parallélisme potentiel de l'application. Ce compromis entre les contraintes temporelles et la consommation de ressources matérielles d'une implantation définit un espace de solutions avec les différentes im-

plantations possibles d'une même application (figure 1). L'axe correspondant aux *performances temporelles* (latence ou temps de réponse) a une borne supérieure représentant la contrainte temps réel qu'on ne peut pas dépasser. L'axe correspondant aux *ressources matérielles* peut représenter le nombre de composants, le nombre de blocs logiques, la surface en silicium, etc., en fonction du type d'architecture choisie pour l'implantation. Pour trouver l'*implantation optimisée* (celle qui respecte la contrainte temporelle et minimise les ressources matérielles utilisées) à partir d'une spécification initiale, il faut explorer l'espace de solutions. Cela exige l'utilisation d'une méthode formelle de conception de systèmes capable de le faire de façon efficace et systématique.

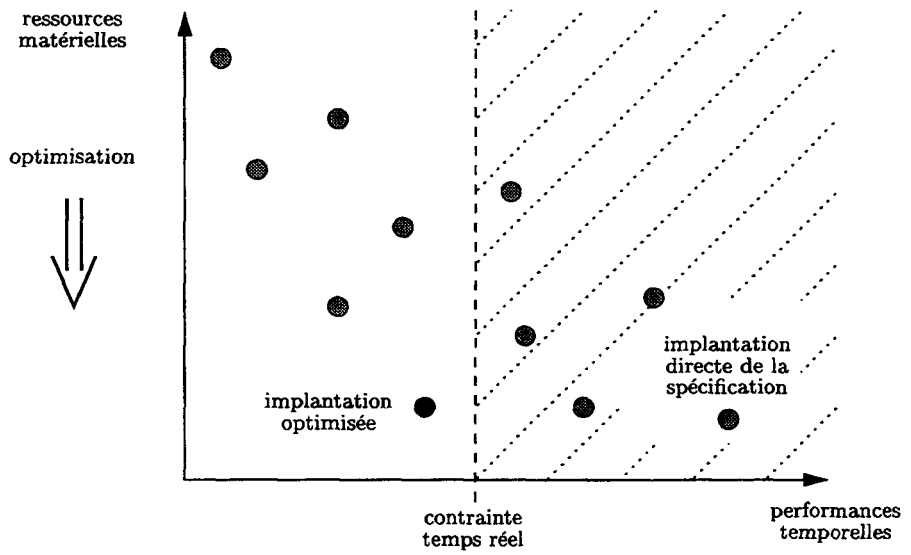


FIG. 1: Implantations d'une spécification algorithmique

À l'INRIA–Rocquencourt (Institut National de Recherche en Informatique et en Automatique), dans le cadre du projet SOSSO, il a été développée une méthodologie appelée "Adéquation Algorithme Architecture" (AAA) pour la conception, le prototypage rapide et l'optimisation d'application distribuée temps réel embarquée et un logiciel nommé *SynDEx* (acronyme de *Synchronised Distributed Executif* – exécutif distribué synchronisé) [GMP*90, LSSS91a, LSSS91b, ELS92, LS92, LMPRS93, BLLMS94, LS94, Sor94], qui la supporte pour des architectures *multi-composants*, c'est-à-dire construite avec des processeurs et des circuits intégrés spécifiques (ASIC, FPGA).

Le but de nos recherches est d'étendre la méthodologie AAA aux circuits reconfigurables de type FPGA, de définir une méthodologie de conception appropriée à ces circuits et d'établir les bases de son intégration dans le logiciel *SynDEx*. L'intégration de cette extension dans *SynDEx* placera ce logiciel au rang des outils de conception conjointe matériel-logiciel, puisqu'il sera capable de générer des implantations optimisées pour les architectures multi-composants, à base de pro-

cesseurs et de circuits intégés spécifiques, ces derniers ayant eux-même été conçus et réalisés avec l'extension de *SynDEx* aux FPGA. Nous nous restreindrons, dans le cadre de ce travail, aux architectures mono-FPGA puisque l'aspect multi-FPGA sera traité au niveau de *SynDEx* multi-composants.

La technologie FPGA allie la flexibilité de programmation et la puissance des architectures spécialisées. De plus, la capacité des circuits reconfigurables augmente constamment, permettant la production rapide de circuits de plus en plus complexes, ce qui demande la mise en œuvre d'une méthodologie de conception de haut niveau.

Dans l'extension de la méthodologie AAA aux circuits reconfigurables, on obtient, par transformations de graphes, plusieurs implantations matérielles, à partir d'une description de l'algorithme (*spécification algorithmique*) sous la forme d'un GFDD. Chacune de ces implantations est décrite sous la forme d'un graphe d'opérateurs interconnectés (*implantation matérielle*). La logique économique veut qu'on cherche à minimiser le coût de l'implantation, choisissant celle qui minimise les ressources matérielles. Mais les contraintes de l'application, principalement le temps réel, imposent une borne à cette minimisation.

Ainsi, nous sommes face à un problème d'*optimisation* : la recherche d'un minimum sous contraintes (voir figure 1). Comme le problème d'allocation de ressources est un problème NP-complet [LS97], cette optimisation n'est véritablement possible qu'à travers l'utilisation d'heuristiques. Ces heuristiques doivent fournir de bons résultats en un temps acceptable, éventuellement avec l'aide d'informations supplémentaires fournies par le concepteur.

L'extension de la méthodologie AAA aux circuits reconfigurables a été validée à travers quelques études de cas. Nous avons choisi d'en présenter une où les algorithmes du produit scalaire (PS) et du produit matrice-vecteur (PMV) mettent en évidence quelques cas intéressants de parallélisme de données et d'opérations, tels que le parallélisme des multiplications dans le PS et le parallélisme des PS dans le PMV.

Par l'intermédiaire de l'analyse des études de cas, notamment le PS et le PMV, nous avons proposé et validé un ensemble de transformations de graphes qui permettent de réduire la latence de l'implantation d'une spécification algorithmique factorisée, au détriment de la consommation des ressources matérielles nécessaires pour sa réalisation. Ces transformations peuvent être nommées *transformations spatio-temporelles* puisque, en partant d'un graphe algorithmique factorisé (multiplexé temporellement), elles nous permettent de défactoriser ce graphe, mettant en évidence le parallélisme potentiel de l'algorithme (dépliage spatial).

Les graphes algorithmiques sont traduits sous la forme de graphes d'opérateurs, nommés graphes matériels, en remplaçant les sommets des GFDD par les opérateurs qui les implantent et ses arcs, par des interconnexions entre opérateurs. De plus, les graphes matériels contiennent des éléments qui assurent la synchronisation des transferts de données entre les registres des opérateurs. Ces éléments de

synchronisation et leurs interconnexions sont générés à partir de l'analyse des relations de consommation et de production entre les sommets du graphe algorithmique.

Comme nous sommes intéressés à optimiser l'implantation sans passer par des cycles complets de développement (comprenant le codage, la simulation, la synthèse et l'estimation), nous avons proposé une méthode pour estimer le coût en termes de surface et de latence de l'implantation matérielle. Les opérateurs du graphe matériel doivent donc être caractérisés en termes de surface et de latence, en fonction de l'architecture-cible. Les performances temporelles et la consommation de ressources matérielles de l'implantation du graphe matériel sont alors estimées et, ensuite, comparées aux contraintes de l'application, afin d'obtenir les informations nécessaires à la réalisation des transformations spatio-temporelles.

Ces transformations effectuent l'optimisation par défactorisation de la spécification, dans le but de satisfaire les contraintes temporelles et de minimiser le nombre de ressources matérielles nécessaires à l'implantation. L'heuristique d'optimisation génère donc une implantation optimisée, composée d'un chemin de données (*datapath*) et d'un chemin de contrôle ou contrôleur (*control path*), décrite sous la forme d'un graphe matériel détaillé. À partir de ce graphe, nous produisons le code VHDL (*Very high speed integrated circuits Hardware Description Language*) correspondant, qui nous permettra de synthétiser une architecture basée sur des circuits reconfigurables du type FPGA, en utilisant des outils de CAO (Conception Assistée par Ordinateur), tels que *Leonardo*, *Cadence*, etc.

Un des grands avantages de la méthodologie proposée, est la possibilité de synthétiser simultanément le chemin de données et le chemin de contrôle. L'architecture-cible, qui a été choisie pour l'implantation matérielle (synthèse) des algorithmes étudiés, est à base de circuits reconfigurables du type FPGA, de la série XC4000XL de *Xilinx*. Nous avons utilisé des outils de CAO développés et/ou commercialisés par *Mentor Graphics* et *Xilinx* pour la compilation, la simulation, la synthèse, la génération des *netlists* et les estimations de performance et de consommation de ressources des implantations matérielles.

Dans le chapitre 1, nous présentons d'abord l'état de l'art sur la conception conjointe matériel-logiciel, décrivant de façon succincte les principaux outils de conception conjointe existants. Nous décrivons également la méthodologie AAA développée à l'INRIA–Rocquencourt pour multi-composants et nous introduisons notre méthode d'extension de cette méthodologie aux circuits reconfigurables. Cette extension nous permettra, par la suite, d'utiliser le logiciel *SynDex* pour la conception conjointe.

Dans le chapitre 2, nous présentons l'état de l'art sur la spécification, la modélisation, les langages de spécification et la validation de systèmes réactifs embarqués temps réel. Nous y décrivons les principes de la spécification algorithmique d'une application et notre modèle unifié de graphes factorisés, en définissant les sommets de base et de factorisation de motifs de graphes répétitifs finis et infinis utilisés pour cette spécification.

Dans le chapitre 3, nous présentons l'état de l'art sur la synthèse de circuits, s'intéressant plutôt à la synthèse aux niveaux *comportemental* et *transfert de registres* qui sont les niveaux concernés par ce travail. Nous y abordons les principes de l'implantation matérielle, en décrivant de façon détaillée les opérateurs de base et de factorisation de motifs de graphes répétitifs finis et infinis. Dans ce chapitre, nous décrivons également le contrôle des opérateurs de factorisation et nous présentons notre méthode de génération automatique de ce contrôle.

Dans le chapitre 4, nous présentons un état de l'art sur les méthodes d'optimisation utilisées par quelques outils de synthèse matérielle. Nous y exposons notre flot de conception, orienté vers la production de code VHDL structurel et synthétisable, fourni à des outils de CAO pour obtenir les *netlists* de configuration d'architectures mono-FPGA. Nous présentons nos méthodes de caractérisation matérielle, d'estimation de consommation de ressources matérielles, d'estimation de performances temporelles et d'optimisation par défactorisation de la spécification algorithmique à l'aide d'une heuristique.

Le Glossaire contient les définitions des principaux termes et expressions techniques utilisés dans ce travail. L'Annexe A présente l'application des principes décrits précédemment au PMV. Nous y montrons quelques défactorisations possibles de la spécification factorisée initiale, les résultats des différentes implantations matérielles possibles, les résultats produits par l'estimateur de surface et de latence et quelques résultats produits par l'heuristique de défactorisation proposée. L'Annexe B contient la bibliothèque d'opérateurs VHDL utilisée pour l'implantation matérielle des opérateurs de base, des opérateurs de factorisation de motifs de graphes répétitifs et des unités de contrôle et le code VHDL correspondant à l'implantation du PMV totalement factorisé.

INTRODUCTION

Chapitre 1

Adéquation Algorithme–Architecture

The goal of the AAA methodology is to find the best matching between an algorithm and an architecture, while satisfying constraints [Syn99].

1.1 Introduction

Les systèmes embarqués temps réel sont de plus en plus présents dans les applications complexes des domaines de l'aérospatiale, de l'automobile, de la robotique, du traitement du signal et des images et du contrôle-commande. Ces systèmes sont souvent intégrés dans des composants hétérogènes (matériel-logiciel, analogique-numérique et électronique-mécanique). Cela exige le développement de méthodologies de conception permettant la spécification, la validation, l'optimisation et l'implantation de systèmes hétérogènes, en utilisant, de préférence, un modèle unifié pendant tout le cycle de conception afin de permettre au concepteur de tels systèmes d'accomplir sa tâche dans un délai plus court, tout en assurant le respect aux contraintes de conception (taille, vitesse, consommation, etc.).

Le plus souvent, l'implantation de ces applications complexes fait appel à des architectures hétérogènes avec des composants programmables (processeurs, microcontrôleurs ou DSP – *Digital Signal Processor*) et/ou de circuits spécialisés (ASIC – *Application Specific Integrated Circuit*, FPGA – *Field Programmable Gate Array*, IC – *Integrated Circuit* ou PAL – *Programmed Array Logic*) communiquant par l'intermédiaire d'un réseau de communication. Les composants programmables offrent plus de flexibilité (le même composant peut implanter différentes applications en exécutant des codes différents), mais les circuits spécialisés plus performants (les circuits sont dédiés à l'implantation de l'application-cible). La reconfigurabilité des circuits du type FPGA permet donc d'allier la performance du matériel à la flexibilité du logiciel. Pour résoudre les problèmes de prototypage rapide optimisé et de

conception matériel/logiciel posés par l'implantation de ces applications, il est essentiel de développer une méthode formelle de conception. Nous avons choisi l'approche "Adéquation Algorithme Architecture" (AAA), développée à INRIA–Rocquencourt et le logiciel de CAO (Conception Assistée par Ordinateur) qui la supporte (*SynDEx*). Elle met en correspondance de façon efficace l'algorithme de l'application envisagée et l'architecture-cible, tout en respectant les contraintes temps réel et minimisant les ressources utilisées pour réaliser une implantation optimisée. C'est un problème d'optimisation sous contraintes.

Dans ce chapitre, nous définissons les systèmes réactifs temps réel, nous discutons le choix entre les circuits reconfigurables et les ASIC pour l'implantation de la partie matérielle d'un système hétérogène et nous présentons l'état de l'art sur les techniques de conception conjointe matériel/logiciel (*hardware/software codesign*), à savoir : les définitions, le concept de modèle unifié, les critères d'évaluation des outils de conception conjointe, la description et la comparaison de quelques outils de conception conjointe universitaires et commerciaux. Nous y présentons aussi la méthode AAA développée à l'INRIA–Rocquencourt et notre proposition d'extension de cette méthode aux circuits reconfigurables.

1.1.1 Systèmes réactifs temps réel

Un système réactif temps réel (figure 1.1) est un système qui intéragit de façon continue avec son environnement, en produisant des réactions à des stimuli venant de l'extérieur dans un temps borné. Le système consomme les informations reçues et les traite dans un délai approprié au contrôle de l'application. Un système fonctionne en temps réel quand il est soumis à des contraintes temporelles (latence et cadence) qui doivent être respectées [Sta92]. Les systèmes embarqués sont fréquemment des systèmes réactifs temps réel utilisés pour le traitement du signal et des images, les télécommunications (commutateurs, téléphones portables) et le contrôle-commande (robotique, automobile, aérospatiale). Ils sont typiquement réalisés à l'aide de diffrentes technologies, tels que des microprocesseurs, des microcontrôleurs, des DSP, des circuits reconfigurables, des circuits analogiques et de microondes, et même des systèmes microélectromécaniques (MEMS). Généralement, le logiciel est utilisé là où il faut avoir plus de flexibilité et le matériel où il faut plus de performance.

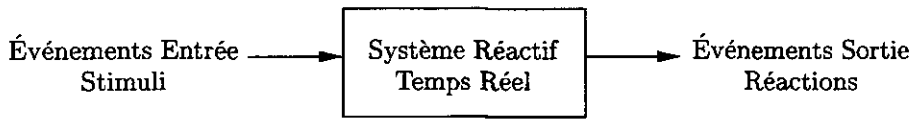


FIG. 1.1: Système réactif temps-réel

Outre les contraintes temporelles, un système réactif est soumis à des contraintes technologiques d'embarquabilité (réduction du volume, du poids, de la surface, de la consommation) et à des contraintes de coût de conception et de fabrication qui encouragent la minimisation des ressources matérielles nécessaires à sa

réalisation. Selon Burns et Wellings [BW97], les caractéristiques les plus importantes d'un système embarqué temps réel sont les suivantes :

- couverture et complexité,
- manipulation de nombres réels,
- fiabilité et sécurité extrêmes,
- contrôle concurrent des composants du système,
- contrôle en temps-réel,
- intéraction avec des interfaces matérielles,
- implantation efficace.

Afin de satisfaire à ces exigences, il est impératif de disposer d'outils performants pour la conception de tels systèmes. Pourtant, les outils de conception de systèmes embarqués existants sont encore très limités et exigent, par exemple, que les spécifications matérielle et logicielle soient réalisées séparément [Pol98]. Les problèmes de ces outils de conception sont les suivants :

- le manque d'une représentation unifiée pour le logiciel et le matériel rendant plus difficile la vérification du système complet et produisant des incompatibilités au niveau des frontières entre le logiciel et le matériel ;
- la définition *a priori* des partitions produisant des systèmes sous-optimaux ;
- le manque d'un flot de conception sans rupture rendant difficile la révision de la spécification et, conséquemment, augmentant le temps de dévelopement.

Il y a différentes approches universitaires et commerciales pour résoudre le problème de la conception de systèmes embarqués. Pourtant, aucune n'offre pas au concepteur, de façon satisfaisante, des fonctionnalités telles que la spécification indépendante de l'implantation et à la synthèse automatique de systèmes réactifs temps réel. Notre extension de la méthodologie AAA aux circuits reconfigurables visera à combler les manques décrits ci-dessus.

1.1.2 FPGA × ASIC

Pourquoi choisir les FPGA et non les ASIC pour cette extension de la méthodologie AAA ? Les ASIC et les FPGA sont également appropriés à l'implantation d'algorithmes bas niveau de TSI qui présentent une grande régularité et qui ont besoin de hautes performances de calcul. Pourtant, il est admis généralement que les FPGA présentent tous les avantages de la fonctionnalité sur mesure offertes par les ASIC, tout en évitant les coûts de développements très élévés de ces derniers et leurs incapacités d'être modifiés après leur production [Gos95]. Ceci rend évident notre choix.

Les principaux avantages des architectures à base de FPGA [Pra96] sont donc les suivants :

- le cycle de conception est beaucoup plus court (quelques jours, contre quelques semaines pour les ASIC) ;
- le risque pendant la conception de prototypes en FPGA est faible à cause de son coût négligeable par rapport au coût élévé de la fabrication d'un ASIC ;
- la propriété de reprogrammabilité des FPGA basés sur des SRAM (*Static Random Access Memory* – mémoires à accès aléatoire statiques) face au caractère non-reconfigurable des ASIC ;
- la régularité de la structure des FPGA rend plus facile le développement des outils de synthèse automatique.

Par contre, les inconvénients des architectures à base de FPGA [Pra96], sont les suivants :

- la surface d'un FPGA est environ dix fois supérieure à celle d'un ASIC équivalent ;
- la vitesse d'un FPGA est inférieure et plus difficilement contrôlable ;
- le coût des FPGA pour la production à grande échelle est plus élévé.

Pour éviter ces inconvénients des FPGA, nous envisageons de privilégier leur utilisation pendant la phase de prototypage du système, où sa configuration finale n'a pas encore été retenue. Une fois que le système a été validé et l'application n'est plus soumise à des modifications à court ou à moyen terme, nous pouvons donc l'implanter sur des ASIC à partir du même code VHDL utilisé pour la synthèse des FPGA. Ceci montre encore un autre avantage de notre choix des FPGA au détriment des ASIC.

Le FPGA est une matrice de blocs logiques (cellules) placés dans une infrastructure d'interconnexions. Il peut être configuré à trois niveaux : (1) la fonction des blocs logiques, (2) les interconnexions des blocs logiques, et (3) les entrées et sorties. Cette configuration est réalisée par l'intermédiaire d'une chaîne de bits chargée à partir d'une source externe. Un FPGA est *programmable* au niveau matériel, offrant les avantages des processeurs génériques et des circuits spécialisés. En fonction de la façon dont ils sont configurés, les FPGA peuvent être classés comme *configurables* (ils ne peuvent être configurés qu'une seule fois) ou *reconfigurables* (ils peuvent être configurés plusieurs fois). Les FPGA reconfigurables peuvent être *statiques* (la configuration est réalisée avant l'opération du circuit) ou *dynamiques* (le circuit peut être partiel ou totalement configuré pendant son opération). Les blocs logiques peuvent être des circuits simples, tels que les portes logiques OU, ET, et NON (grain fin), ou des circuits plus complexes, tels que les multiplexeurs, les tables de correspondance et les unités logiques et arithmétiques (gros grain) [Mar99].

Nous utilisons, dans ce travail, les FPGA *Xilinx* de la série XC4000, qui sont des FPGA reconfigurables statiquement à gros grain, comme décrit en [Xil94, Xil96, Xil99].

1.1.3 Conception conjointe matériel/logiciel

La conception de systèmes matériels spécifiques est en pleine expansion. La tendance est l'élaboration de méthodes qui, profitant de la rigueur formelle des outils de conception existants, assurent un haut degré de fiabilité. La recherche s'oriente vers la définition des outils permettant un niveau d'abstraction de plus en plus élevé et des choix de mise en œuvre plus variés (circuits hétérogènes, architectures mixtes matérielle/logicielle, etc.)[1] [Bel94].

Mead et Conway [MC80] affirmaient déjà en 1980 la nécessité de se servir de méthodologies structurées de conception pour maîtriser la complexité de la conception et produire de bons circuits. Au début des années 90, la recherche autour de méthodes de conception d'architectures était déjà très active [MPC90, Cam90, GDWL92]. Les méthodes de conception sont traditionnellement orientées vers les implantations soit matérielles, soit logicielles des applications. Les implantations matérielles offrent normalement des performances plus élévées puisqu'elles sont dédiées à la solution du problème à résoudre. Elles sont plus rapides à cause de l'exécution spatiale, avec un fort degré de parallélisme. Les implantations logicielles offrent plus de flexibilité, car elles utilisent des processeurs programmables qui sont pourtant plus lents à cause de l'exécution temporelle/séquentielle. Elles sont moins efficaces parce que les opérateurs peuvent éventuellement être inadaptés aux tâches exécutées [DW99]. Ainsi, on s'oriente de plus en plus vers des systèmes mixtes matériel/logiciel, capables d'offrir simultanément performance et flexibilité.

La conception conjointe de systèmes matériel/logiciel (*hardware/software codesign*) désigne la conception de systèmes comportant une partie matérielle, constituée d'un ou de plusieurs circuits spécifiques (ASIC, composants reconfigurables), et une partie logicielle qui est exécutée sur une architecture à base de processeurs standards et/ou spécifiques (microcontrôleurs, DSP). La conception conjointe peut être interpretée comme la spécification, la validation et l'exploration des différentes alternatives de conception d'un système mixte matériel/logiciel dans le but d'optimiser des critères de coût et/ou de performances [Bel94]. L'exploration de l'espace de solution est une tâche-clef des systèmes de conception conjointe, parce que, étant associée à une phase d'analyse de performances, elle permet de définir des caractéristiques fondamentales telles que le coût, les performances temporelles et la puissance consommée [R*98].

La conception conjointe est donc une tentative d'intégrer les techniques de conception matérielle et logicielle dans une méthodologie unique, structurée et, de préférence, automatique [AT96]. Les avantages de cette intégration de techniques sont l'accélération du processus de conception et la possibilité d'évaluer dynamiquement les différents compromis matériel/logiciel possibles.

Dans le cas des DSP et des circuits spécifiques, le processus de conception

[1] Les techniques de spécification et de modélisation d'un système matériel sont décrites dans le chapitre 2. Les différents niveaux d'abstraction (système, algorithme, transfert de registres, portes, transistors) utilisés pour décrire un système matériel sont présentés en détail dans le chapitre 3.

doit éliminer le fossé existant entre la spécification fonctionnelle hétérogène et son implantation hétérogène. Au niveau système, les DSP, p. ex., ont besoin d'une combinaison entre les modèles flot de données et les modèles réactifs orientés flot de contrôle pour leur spécification complète [RVBM96].

La modélisation des systèmes comprenant du matériel et du logiciel à différents niveaux d'abstraction est nommée *co-spécification*. La validation des conceptions soit par simulation, soit par vérification formelle du matériel et du logiciel à différents niveaux d'abstraction est appelée *co-simulation* ou *vérification formelle*. Le partitionnement de l'application en deux parties, matérielle et logicielle, l'exploration et l'évaluation des différentes alternatives, la génération des circuits correspondant au logiciel, ainsi que la génération de l'interface entre les deux sont appelés *co-synthèse* [TAS93, GD93, EHB93, GVNG94] ou *synthèse système* [GD92].

Les méthodologies de conception de systèmes matériel/logiciel basées sur les niveaux d'abstraction plus élévés (système ou comportemental) doivent être capables de résoudre les problèmes suivants [Cod99] :

- Génération automatique d'interfaces entre les parties matérielles et logicielles ;
- Génération de différentes solutions matérielles/logicielles : les tâches parallèles et temporellement critiques étant implantées sur matériel (IC, ASIC, FPGA), et les tâches séquentielles, irrégulières et complexes étant implantées sur logiciel (microcontrôleurs, DSP ou ASIP–*Application-Specific Instruction-Set Processor*) ;
- Spécification des systèmes matériel/logiciel : les langages de spécification matérielle (VHDL–VHSIC *Hardware Description Language*, Verilog) et les langages de spécification logicielle (C, C++, assembleur) ne sont pas appropriés pour la spécification de systèmes mixtes ;
- Partitionnement : de nouveaux outils sont nécessaires pour effectuer la mise en correspondance (*mapping*) entre la spécification et les composants matériels et logiciels coopérants.

Le flot de conception dans la conception conjointe est schématisé dans la figure 1.2 [Bel94]. Dans ce travail, nous nous intéressons à la synthèse matérielle des sous-ensembles d'une spécification algorithmique implantés sur des circuits reconfigurables du type FPGA.

Les outils et méthodologies existants pour la conception de systèmes embarqués mixtes génèrent en sortie différents langages pour décrire les composants matériels et logiciels (p. ex., C pour spécifier les composants implantés sur logiciel, VHDL ou Verilog pour spécifier le matériel et les interfaces) [Cod99].

1.1.4 Critères d'évaluation des outils de conception conjointe

L'évaluation ou l'apport réel d'une méthode ou d'un outil de conception conjointe peut se faire en fonction de [ACS*92, Fai93, AT96] :

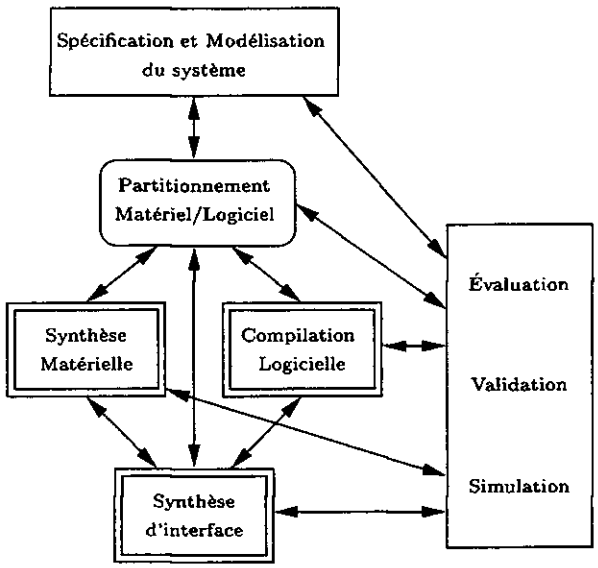


FIG. 1.2: Flot de conception conjointe matériel/logiciel [Bel94]

- **Spécification et modélisation des systèmes** : il faut disposer d'un langage pour la spécification fonctionnelle à différents niveaux d'abstraction ainsi que pour la spécification des contraintes (temps d'exécution, encombrement, type d'architecture cible, etc.). Le modèle doit être interpreté indifféremment comme logiciel ou matériel. Il faut avoir la possibilité d'effectuer des analyses de performance à travers des méthodes habituelles (chaîne de Markov, rétro-annotation, etc.). Il doit offrir des propriétés permettant d'envisager le partitionnement de manière efficace (granularité, modélisations de débits différents, etc.) ;
- **Partitionnement** : il faut offrir une certaine flexibilité (partitionnement manuel, guidé ou automatique) et l'exploration de différentes alternatives de conception basée sur des critères tels que la performance, le coût, la concurrence, les communications et la nature des calculs. Le modèle doit disposer d'une granularité suffisante pour les algorithmes de partitionnement ;
- **Liens avec des outils de synthèse matérielle et de compilation** : il faut offrir des liens avec les principaux outils de CAO par l'intermédiaire de langages de description de matériel (VHDL, Verilog, Silage) qui permettent d'utiliser des outils spécialisés pour quelques applications et des compilateurs efficaces pour des machines mono ou multi-processeurs ;
- **Simulation et validation** : il doit offrir la possibilité d'effectuer des simulations mixtes (logiciel/matériel) à différents niveaux d'abstraction. La simulation au niveau système doit apporter un gain significatif en temps. Il doit offrir aussi la possibilité d'utiliser des techniques d'analyse statique et de preuve formelle ;
- **Intégration d'outils** : l'environnement pour la conception conjointe doit intégrer les fonctionnalités décrites ci-dessus et la facilité d'utilisation.

1.1.5 Modèle unifié de conception

Les outils de conception conjointe existants cherchent à décrire la fonctionnalité de l'application de façon indépendante de l'architecture qui l'implante. Quelques spécialistes pensent que le matériel et le logiciel doivent être modélisés séparément et que l'allocation spatiale (distribution) des fonctions entre les composants matériels et les composants logiciels doit être réalisée de façon optimale [R*98]. Pourtant, d'autres, comme nous, pensent que, sans le développement d'une métodologie de conception unifiée, les outils de CAO pour le matériel et les outils d'IDE (*Interactive Design and Engineering*) pour le logiciel continueront à parler des langues différentes, au détriment de l'efficacité et de la fiabilité du processus de conception. Ainsi, un seul style de description du système doit être utilisé [Pag98], à partir duquel le matériel et le logiciel peuvent être synthétisés. Il faut ajouter que les styles actuellement utilisés pour décrire le matériel sont totalement inadaptés à la description du logiciel, parce qu'ils n'offrent pas un niveau élévé d'abstraction et leur sémantique est très proche du matériel.

La raison principale de l'implantation d'une partie de l'application sur du matériel au détriment du logiciel est dû aux gains en termes de vitesse d'exécution offerts par le premier. Le matériel est plus rapide du fait de son parallélisme inhérent [Pag98]. Ainsi, il faut que les descriptions matérielles soient capables d'exprimer ce parallélisme.

Zhu et Gajski [ZG99] ont développé un modèle architectural formel pour traiter de façon unifiée deux composants importants des systèmes *on chip* : l'architecture de processeurs à ensemble d'instructions (ISA–*Instruction Set Architecture*) et la machine à états finie avec une architecture à chemin de données (FSMD–*Finite State Machine with Datapath*).

1.1.6 Environnements de conception conjointe

Depuis le début des années 90, plusieurs méthodologies, outils et environnements de conception conjointe matériel/logiciel ont été développés par différents groupes de recherche universitaires et industriels. Nous présentons ci-après quelques environnements universitaires (*Chinook*, *CodeSign*, *COSYMA*, *CoWare*, *GRAPE II*, *Polis* et *Ptolemy*) et commerciaux (*Cierto VCC* et *DSP Station*) parmi les plus cités dans la bibliographie concernant la conception conjointe.

Chinook/ipChinook

Chinook [COB95, COH*99] est un outil de CAO pour la co-synthèse matérielle/logicielle de systèmes embarqués, développé à Université de Washington. Il a été conçu pour les systèmes réactifs orientés-contrôle, sous contraintes temporelles, basés sur des architectures distribuées [Chi99]. *Chinook* utilise une spécification commune pour

les composants matériels et logiciels. Le modèle de spécification utilisé mélange les meilleures propriétés des méthodes basées sur des processus (p.ex. CSP–*Communicating Sequential Process*) et des méthodes basées sur les FSM (*Finite State Machine*) hiérarchiques (p.ex. les langages *StateCharts* et *Esterel*) [CB98a]. Ce modèle, appelé *modal process*, permet de spécifier la partie contrôle indépendamment de la sémantique de communication. Il encapsule également la fonctionnalité et définit la granularité du partitionnement.

L'utilisateur de *Chinook* part d'une description comportementale de l'application et peut l'implanter sur plusieurs architectures-cible de façon interactive. La distribution et le partitionnement sont effectués par l'utilisateur et *Chinnok* effectue la mise en correspondance (*mapping*) sur les architectures. La possibilité de changer l'architecture-cible (*retargetability*) offerte par *Chinook* permet au concepteur d'utiliser plus facilement de nouvelles technologies.

ipChinook est un outil de conception de systèmes embarqués distribués, basé sur une approche de construction de systèmes orientée-composants. Cette approche, qui encapsule des IP (*Intellectual Property*), augmente la capacité de *ipChinook* de réutiliser des modules déjà existants [COH*99]. L'outil reçoit en entrée une description comportementale de l'application, une description de l'architecture-cible et une fonction d'allocation qui met les deux en correspondance (*mapping*). La description comportementale correspond à la fonctionnalité du système décrite sous la forme de modules concurrents. La description architecturale décrit les processeurs disponibles, les dispositifs d'entrée/sortie, les bus de communication et la topologie du matériel. Le *mapping* détermine quel composant matériel va exécuter quel module. Un composant matériel peut être un microprocesseur qui exécute un logiciel ou un bloc logique programmable, comme un FPGA, p.ex. Cette structure permet de changer la fonctionnalité, le matériel et la distribution de la fonctionnalité de façon indépendante les uns des autres. *ipChinook* permet d'explorer l'espace de solutions à travers la mise en correspondance (*mapping*) d'une description de haut niveau sur l'architecture-cible choisie par le concepteur. Actuellement, le partitionnement est effectué manuellement par le concepteur ou par un outil de partitionnement et la vérification formelle n'est pas possible. Des outils d'estimation sont en train d'être développés afin de fournir des informations temporelles aux outils d'ordonnancement et de synthèse de communication.

CodeSign/Moses

Le projet *CodeSign* [Ess96], développé au ETH (*Eidgenössische Technische Hochschule*) Zurich, étudie la conception, la modélisation, la simulation, la vérification et l'analyse de systèmes complexes embarqués temps réel. *CodeSign* s'intéresse à l'intégration de multiples formalismes dans une plateforme générique commune, à la simulation efficace de différents modèles de calcul, à la vérification de propriétés temporelles et à la génération rapide de prototypes matériels et logiciels à partir des spécifications d'entrée [Cod99].

L'outil *CodeSign* est un environnement graphique de modélisation de systèmes qui utilise comme formalisme de base les réseaux de Petri temporels (TPN–*Time Petri Net*), orientés-objets de haut niveau. Les composants des TPN sont hiérarchiques et orientés-objets. D'autres formalismes peuvent être utilisés avec le modèle TPN. Ce modèle offre une représentation graphique intuitive et une sémantique exécutable autorisant la simulation pendant les phases de spécification et modélisation. Les différents modèles de calcul ont aussi une sémantique formellement définie qui permet leur simulation [Ess96, Cod99]. L'environnement *CodeSign* permet la modélisation de systèmes matériels et logiciels où les détails de l'implantation sont ajoutés au fur et à mesure que l'on affine la conception. *CodeSign* effectue l'exploration de l'espace de conception à travers la modification des caractéristiques fonctionnelles et structurelles de ces modèles, à partir de la réalimentation des résultats de simulation, de l'analyse formelle et de la génération d'implantation. Ainsi, le concepteur peut explorer les différentes implantations dérivées d'une spécification initiale jusqu'à trouver une solution optimale.

La génération efficace de code est une fonction importante de *CodeSign*. L'outil permet la génération des codes C et VHDL pour l'implantation sur des composants logiciels et matériels. Le VHDL généré décrit de façon algorithmique la fonctionnalité souhaitée. Pour transformer ce VHDL comportemental en VHDL structurel (RTL–*Register Transfer Level*) synthétisable, il faut faire appel à des outils de synthèse de haut niveau, tels que *CoWare*, *Polis* ou *Cierto VCC*.

Le projet *Moses* [Mos99], également développé au ETH Zurich à partir des résultats du projet *CodeSign*, étudie la modélisation, la simulation, l'implantation et la vérification de systèmes hétérogènes. La plateforme de modélisation et de simulation de *Moses* permet l'utilisation simultanée de différents formalismes de modélisation. Le projet *Moses* cherche à offrir une grande flexibilité de modélisation et une grande efficacité de simulation pour des applications industrielles qui exigent une simulation rapide, une configurabilité optimisée et des langages de modélisation plus expressifs.

COSYMA

COSYMA (*COSYnthesis for eMbedded Architectures*) [EHB93] est un environnement de conception conjointe pour les microcontrôleurs et pour les systèmes embarqués, développé à l'Université de Braunschweig, en Allemagne, sous la direction du Prof. Rolf Ernst. L'architecture-cible pour COSYMA est composée d'un processeur RISC (*Reduced Instruction Set Code*) standard, d'une mémoire RAM (*Random Access Memory*) rapide avec temps d'accès d'un seule cycle d'horloge pour les programmes et les données, et d'un co-processeur spécifique généré automatiquement [Cos98].

La description du système se fait par l'intermédiaire de processus communicants (CSP), en utilisant des fonctions en langage C^x. Ce langage correspond à une extension minimale du langage C afin de permettre le traitement de processus parallèles. Cette description en C^x est strictement séparée des directives de contraintes

et d'implantation, pour minimiser les extensions au langage C et permettre plus de portabilité. La description d'entrée est traduite dans un graphe de syntaxe en utilisant une version adaptée du compilateur *Stanford SUIF* [Wil94].

COSYMA utilise un partitionnement matériel/logiciel automatique à grain fin, en s'appuyant sur des estimations du coût de l'implantation du logiciel et du matériel, ainsi que des communications. Le partitionnement part d'une solution totalement logicielle et il essaye d'extraire des composants matériels de façon itérative jusqu'à respecter les contraintes temporelles. Ainsi, le partitionnement cherche à satisfaire les contraintes temps réel, minimiser les coûts matériels et minimiser le temps de réponse du système de CAO, pour permettre à l'utilisateur d'explorer l'espace de solutions [Cos98]. L'estimation des coûts de communication et l'optimisation du code sont effectuées à partir d'une technique basée sur les algorithmes d'Aho, Sethi et Ullman [ASU89] pour l'analyse de flots de données. Le résultat de l'analyse des flots de données peut être utilisé pour guider des optimisations telles que la propagation de constantes, l'optimisation arithmétique, la réduction de la hauteur des arbres et l'élimination de sous-expressions communes.

COSYMA produit du code C pour la partie logicielle et du code HardwareC [KD90] pour la partie matérielle, qui peut être synthétisée par *Olympus* [DKMT90, Oly95], un outil de conception de circuits numériques développée à l'Université de Stanford. Pour la synthèse de haut niveau (HLS–*High-Level Synthesis*), *COSYMA* utilise le *Braunschweig Synthesis System* (BSS), qui est un système de synthèse de haut niveau développé spécifiquement pour la conception de co-processeurs rapides. Le modèle interne au BSS est un CDFG (*Control Data Flow Graph*) hiérarchique. Un outil de synthèse au niveau RTL, le *Synopsis Design Compiler*, génère la *netlist* finale. Pour la synthèse logicielle, un compilateur C standard est utilisé. Les inconvénients de *COSYMA* sont le manque de précision dans l'estimation des coûts ainsi que les temps élevés de compilation.

CoWare

L'environnement *CoWare*, développé à l'IMEC (Belgique), est un environnement pour la conception de systèmes hétérogènes matériel/logiciel [Ime96]. Il permet de modéliser les systèmes aux niveaux *système* et *architectural*. Il permet aussi de représenter le processus de conception de la spécification fonctionnelle à l'implantation matérielle détaillée [RVBM96].

La méthode de spécification utilise un modèle de données basé sur des processus communicants hétérogènes. Ce modèle permet de séparer les descriptions des comportements fonctionnels et de communication. La même méthode de spécification est utilisée pour modéliser l'architecture basée sur des processeurs programmables. Ainsi, *CoWare* intègre une méthodologie de conception matériel/logiciel indépendante du processeur-cible. L'application est spécifiée par l'intermédiaire de processus communicants dont les comportements peuvent être décrits par des langages comme C, C++, DFL (*Data Flow Language*) ou VHDL.

À partir d'une spécification fonctionnelle, des distributions et des partitionnements matériel/logiciel différents peuvent être explorés par l'utilisateur de façon interactive. La co-implantation de l'application est effectuée automatiquement par l'outil de synthèse *Symphony*. Ensuite, les composants matériels et logiciels générés par *Symphony* [Cow99] peuvent être implantés par d'autres outils de CAO, comme le compilateur C *ARM*, le compilateur VHDL *Synopsis* et l'environnement *Cathedral*.

Pendant le processus de conception, *CoWare* permet de réaliser la co-simulation de spécifications à différents niveaux d'abstraction. Pour cela, les simulateurs *Synopsis* pour le code VHDL et *ARM* pour le code C ont été intégrés dans son environnement.

GRAPE II

GRAPE II (*Graphical RApid Prototyping Environment*), développé à l'Université Catholique de Leuven, en Belgique, est un environnement de développement au niveau système pour la spécification, l'estimation de ressources, le *retiming*, le partitionnement, la distribution, l'ordonnancement, le routage, la génération de code, la compilation, le débogage, la simulation et l'émulation des applications basées sur les DSP. GRAPE II permet l'émulation temps-réel d'applications synchrones multi-vitesse (*multi-rate*) sur des architectures-cible hétérogènes, intégrées par de multiples processeurs TMS320C40 et des FPGA *Xilinx* [LEAP94a, LEAP94b]. Il permet également la simulation et le débogage des applications asynchrones ou synchrones *multi-rate* sur une station Unix ou d'autres machines, telles que *Meiko Computing Surface* ou *Parsytec Xplorer* [LEAP94a].

L'application est spécifiée d'une façon totalement indépendante de l'architecture-cible qui va l'implanter : le concepteur peut utiliser son éditeur graphique hiérarchique pour dessiner le graphe de dépendances de données, ou il peut se servir de l'interface pour l'environnement *DSP Station* [DSP99] de *Frontier Design*, qui lui permet de spécifier l'application dans le langage flot de données DFL (dérivé du langage *Silage* [Hil85]). GRAPE II génère automatiquement du code VHDL pour les outils de synthèse matérielle et du code C ou assembleur pour les compilateurs DSP. L'architecture-cible est spécifiée de façon similaire tout en utilisant l'éditeur graphique. On peut spécifier la fréquence d'horloge, le taux et le protocole de communication, la quantité de mémoire disponible, etc.

GRAPE II estime la quantité de ressources nécessaires à l'exécution de chaque tâche. Pour des cibles microprocesseurs, il estime la latence, la taille du code et de la mémoire, et des ressources comme les convertisseurs analogique-numérique. Pour des cibles FPGA, il estime les ressources en termes du nombre de blocs logiques configurables (CLB–*Configurable Logic Block*), entrées/sorties, etc. GRAPE II est un environnement ouvert. Ainsi, le concepteur peut y ajouter ses propres outils. GRAPE II est commercialisé sous le nom *Virtuoso Synchro* par *Intelligent Systems International*.

Polis

Polis, développé à l'Université de Californie, à Berkeley, est un environnement unifié qui implante une méethodologie de spécification, de synthèse automatique et de validation de systèmes réactifs temps réel à contrôle intensif [Pol98], composés d'un logiciel exécuté sur un micro-contrôleur et d'un matériel spécialisé [Bel*98]. Il utilise une représentation unifiée pour le matériel et le logiciel. Ce modèle est conservé tout au long du processus de conception, afin de préserver les propriétés formelles de la spécification. *Polis* effectue le partitionnement de l'application entre modules coopérants logiciels et matériels et synthétise l'interface entre eux.

Le système *Polis* est basé sur une représentation de type machine à état finie (FSM) : la machine à état finie de *co-design* (CFSM). La différence entre les deux modèles est que, dans le CFSM, le modèle de communication synchrone des FSM concurrentes classiques est remplacé par un temps de réaction fini, différent de zéro et illimité. Ce modèle de calcul est globalement asynchrone et localement synchrone. Chaque élément d'un réseau CFSM décrit un composant du système à modéliser. La spécification CFSM est indépendante de l'implantation matérielle ou logicielle. CFSM est un modèle synthétisable et vérifiable à cause du grand nombre de théories et outils développés pour le modèle FSM qui peuvent être facilement adaptés au CFSM. Le flot de conception de *Polis* [Pol98] est décrit ci-dessous :

- *Traduction d'un langage de haut niveau* : l'utilisateur spécifie l'application en utilisant un langage de haut niveau (Esterel, FSM graphique, Verilog ou VHDL) qui est traduit directement vers le modèle CFSM ;
- *Vérification formelle* : la méthodologie de spécification formelle et synthèse utilisée par *Polis* permet d'employer des algorithmes de vérification formelle basés sur des FSM. *Polis* inclut un traducteur du formalisme CFSM vers le FSM pour permettre l'utilisation des systèmes de vérification formelle (p.ex., VIS) ;
- *Co-simulation du système* : la co-simulation logicielle/matérielle au niveau du système permet de guider les choix du concepteur (partitionnement matériel/-logiciel, sélection de la CPU–*Central Processor Unit* et ordonnancement). *Ptolemy* est utilisé pour effectuer la simulation. Le système génère du code VHDL contenant toute l'information nécessaire à la co-simulation. Ainsi, n'importe quel simulateur commercial pourrait être utilisé ;
- *Partitionnement* : les décisions de conception au niveau du système (partitionnement matériel/logiciel, sélection de la CPU et ordonnancement) sont basées sur l'expérience de concepteur, donc très difficiles à automatiser. *Polis* offre encore des mécanismes de rétro-alimentation, de vérification formelle et de co-simulation du système ;
- *Synthèse matérielle* : un sous-réseau CFSM implanté sur du matériel est synthétisé et optimisé en utilisant des techniques de synthèse logique. Chaque CFSM, interpretée comme une spécification RTL, peut être traduite (*mapped*) vers le format BLIF (*Berkeley Logic Interchange Format*), XNF (*Xilinx Netlist Format*), VHDL ou Verilog ;

- *Synthèse logicielle* : un sous-réseau CFSM implanté sur du logiciel est traduit (*mapped*) vers une structure logicielle qui inclut une procédure pour chaque CFSM et un système d'exploitation temps réel. Le comportement réactif est synthétisé à travers un processus de deux étapes : (1) le comportement souhaité est implanté et optimisé utilisant une représentation de haut niveau, indépendante du processeur, similaire à un CDFG et, (2) le CDFG est traduit vers du code C portable et utilise n'importe quel compilateur pour l'implanter et l'optimiser sur un micro-contrôleur. Un prédicteur temporel analyse le logiciel et estime la taille du code et les temps d'exécution (les micro-contrôleurs MIPS R3000, Motorola 68HC11 et 68332 ont été caractérisés). La précision du prédicteur est de l'ordre de 20 % ;
- *Interface avec d'autres domaines d'implantation* : les interfaces entre différents domaines d'implantation (matériel/logiciel) sont automatiquement synthétisées par *Polis*. Ces interfaces se présentent sous la forme de circuits périphériques et de procédures logicielles présentes dans l'implantation synthétisée.

L'estimation de performance de l'implantation peut être réalisée par simulation du comportement de l'architecture sélectionnée, utilisant un modèle temporel abstrait du processeur dans l'environnement *Ptolemy*.

Ptolemy/Ptolemy II

Le projet *Ptolemy* étudie la modélisation, la simulation et la conception de systèmes embarqués temps réel concurrents. Il a été développé à l'Université de Californie, à Berkeley, sous la direction du Prof. Edward Lee. *Ptolemy* s'intéresse particulièrement aux systèmes embarqués qui mélangent différentes technologies, comme l'électronique analogique et numérique, le matériel et le logiciel, et les composants électroniques et mécaniques (y compris les MEMS–*MicroElectroMechanical Systems*). Il s'oriente vers l'assemblage de composants concurrents à partir de l'utilisation de modèles de traitement bien définis, qui contrôlent l'interaction entre ces composants. La classe de systèmes ciblés par *Ptolemy* est celle des systèmes réactifs [Pto00].

Ptolemy Classic est un environnement hétérogène de conception et simulation qui offre plusieurs modèles de calcul. Il est écrit en C++ et dispose d'une interface graphique qui permet de construire des modèles sous la forme de diagrammes de blocs. Il est un outil de description et de simulation de systèmes orienté aux applications de traitement du signal [LM93]. Les modèles de calcul offerts par *Ptolemy* sont les suivants :

- CSP (*Communicating Sequential Processes*) : les modèles CSP sont bien adaptés aux applications où le partage des ressources est un élément-clef, comme dans les modèles de base de données client-serveur, les systèmes multi-tâches et le multiplexage de ressources matérielles ;

- CT (*Continuous Time*) : le domaine CT est utile pour la modélisation de systèmes physiques avec descriptions sous la forme d'équations algébriques/différentielles linéaires ou non-linéaires, comme pour les circuits analogiques et plusieurs systèmes mécaniques. Les systèmes embarqués contiennent souvent des composants qui sont mieux modélisés avec des équations différentielles, comme pour les MEMS et d'autres systèmes mécaniques, les circuits analogiques et les circuits de micro-ondes ;
- DE (*Discret-Event*) : ce modèle de calcul est très utilisé pour la spécification de matériels numériques et pour la simulation de systèmes de télécommunications. Il est présent dans plusieurs environnements de simulation, langages de simulation et langages de description matérielle (y compris VHDL et Verilog) ;
- FSM (*Finite State Machines*) : les modèles FSM sont excellents pour spécifier la logique de contrôle de systèmes embarqués, particulièrement s'il s'agit de systèmes où la sécurité est critique. Ce modèle présente deux inconvénients : d'une part, il n'est pas approprié pour décrire des fonctions récursives partielles et, d'autre part, le nombre d'états peut être assez grand, même pour des systèmes peu complexes ;
- PN (*Process Networks*) : les modèles PN sont faciles à paralléliser ou à distribuer. Ils peuvent être implantés de façon efficace, soit en logiciel, soit en matériel. Le grand inconvénient de ce modèle est qu'il est peu pratique pour spécifier la logique de contrôle. Cet inconvénient peut être surmonté si l'on combine le modèle PN avec le modèle FSM ;
- SDF (*Synchronous Data Flow*) : ce modèle est approprié pour modéliser des applications de traitement du signal synchrones. *Ptolemy* dispose d'une large bibliothèque d'opérateurs de base. Une analyse statique permet de déterminer la cohérence du système [LM87]. Le modèle SDF est un formalisme de spécification extrêmement utile pour les logiciels embarqués temps réel et pour le matériel ;
- DDF (*Dynamic Data Flow*) : ce modèle est utilisé pour la description des aspects dynamiques liés au contrôle des algorithmes flot de données, comme ceux qui réalisent le traitement du signal asynchrone ;
- SR (*Synchronous/Reactive*) : dans ce modèle, les arcs représentent les valeurs des données qui sont synchronisées avec le cadencement de l'horloge globale. Le modèle SR est excellent pour spécifier les applications avec une logique de contrôle complexe et concurrente. Il est aussi approprié aux applications temps réel, où la sécurité est critique, à cause de la synchronisation stricte du modèle. Les langages synchrones tels que Esterel, Signal, Lustre et Argos utilisent le modèle SR ;
- Thor : modèle de simulation d'architectures au niveau RTL (*Register Transfer Level*) ;
- VHDL.

La conception conjointe dans *Ptolemy* est basée sur le modèle SDF [KL93]. Chaque opération dans SDF peut être implantée en matériel ou en logiciel. Le partitionnement est manuel. L'utilisateur dispose de plusieurs types d'interface matériel/-

logiciel dans *Ptolemy*. Il peut générer une implantation sous la forme de codes C et assembleur pour deux DSP différents, à partir d'une même description flot de données du système [Pto00]. *Ptolemy* génère du code C pour la partie logicielle, ainsi que du code *Silage* pour la partie matérielle.

Parmi les résultats obtenus par le projet *Ptolemy*, on peut citer :

- la formalisation des modèles de traitement pour le flot de données ;
- la gestion de la régularité dans les graphes flot de données à partir de fonctions d'ordre supérieur ;
- la synthèse de logiciel embarqué à partir de graphes flot de données ;
- les techniques d'ordonnancement parallèle ;
- l'optimisation de la communication interprocesseurs dans les implantations parallèles ;
- l'intégration d'un visualisateur pour la conception hétérogène.

L'environnement *Ptolemy* est utilisé dans plusieurs applications, comme le traitement d'images, les télécommunications, le parallélisme, les communications sans fil, la conception de réseaux, la gestion d'investissements, la modélisation de systèmes optiques de communication, les systèmes temps réel et la conception matériel/logiciel [Pto00]. Le *Ptolemy Classic* possède des mécanismes peu efficaces pour migrer les simulations idéalisées en point flottant vers des simulations en point fixe sur des logiciels embarqués, FPGA et d'autres matériels spécialisés [Pto99].

Le projet *Ptolemy* a développé un nouvel environnement basé sur Java, qui s'appelle *Ptolemy II*, orienté vers la conception et la modélisation concurrente hétérogène. Il offre une infrastructure unifiée pour l'implantation de plusieurs modèles de calcul.

Cierto VCC

Le *Cierto Virtual Component Co-design* [Cie99], développé par *Cadence Design Systems*, est un environnement de conception au niveau système. Il permet aux concepteurs de systèmes mixtes matériel/logiciel d'intégrer des composants virtuels représentant le matériel et le logiciel, d'explorer les compromis entre logiciel et matériel et d'analyser la performance de l'implantation tout au début du cycle de dévelopement. *Cierto VCC* offre une grande intégration entre les langages et les technologies de création des IP (p. ex., C, C++, SDL, MatLab, HDL comportemental) et le système de traitement de signal *Cadence Cierto* (SPW–*Signal Processing Worksystem*). Il permet au concepteur de spécifier la fonctionnalité de l'application indépendamment de l'implantation. *Cierto VCC* génère les données exigées par les outils de co-vérification afin de simuler le système complet au niveau logique.

La description comportementale du système sous la forme de blocs fonctionnels peut être importée sous les formats C, C++, SDL et HDL comportemental.

Le partitionnement matériel/logiciel est réalisé par l'utilisateur par l'intermédiaire d'un éditeur de mise en correspondance (*mapping*). *Cierto VCC* permet à l'utilisateur d'effectuer une exploration guidée de l'espace de solutions.

Cierto VCC utilise des modèles VPM (*Virtual Processor Models*) pour estimer la performance de logiciels embarqués. Cela permet d'évaluer les effets des décisions prises sur l'implantation dès les phases initiales de conception. L'interface entre *Cierto VCC* et l'outil de vérification matérielle/logicielle *Affirma* permet d'effectuer la co-vérification des composants matériels et logiciels à un très bas niveau d'abstraction. Après avoir effectué la simulation et la vérification, *Cierto VCC* génère du HDL pour la partie matérielle et du code C pour la partie logicielle.

DSP Station

DSP Station est un environnement pour la conception d'applications de traitement du signal développé par *Mentor Graphics* [MG93]. Il dispose d'une interface graphique avec des différents outils de synthèse architectural, d'exploration de l'espace de solutions, de visualisation, de simulation, de création de filtres, etc. Il s'intègre bien dans la chaîne CAO de *Mentor Graphics*. L'outil *DSP Design Lab* permet au concepteur de spécifier l'algorithme de façon indépendante de son implantation. Cet outil permet également de simuler et d'optimiser la spécification avant l'implantation. L'outil de synthèse comportementale *Mistral* offre au concepteur la possibilité d'explorer l'espace de solutions pour évaluer l'implantation de l'algorithme sur différentes architectures, dans le but de trouver l'implantation la plus efficace [DSP99].

La spécification comportementale peut être effectuée à partir d'un diagramme de blocs, d'un programme C ou d'un langage flot de donnée. *DSP Station* génère du code C pour la partie implantée en logiciel et du VHDL comportemental synthétisable pour la partie implantée en matériel. Il offre un flot de conception unique de la spécification à l'implantation : il effectue la synthèse interactive du chemin de données, l'ordonnancement automatique, l'exploration de l'espace de solutions et intègre des DSP, ASIC, IC et FPGA.

DSP Station présente quelques limitations pour la conception de systèmes : le partitionnement matériel/logiciel non fourni et l'intégration difficile des circuits ou logiciels implantés avec d'autres outils.

1.1.7 Comparaison des environnements de conception conjointe

Nous utilisons ci-après les critères d'évaluation des outils de conception conjointe présentés dans la section 1.1.4 pour effectuer une comparaison entre les environnements décrits ci-dessus, comme le montre le tableau 1.1.

TAB. 1.1: Environnements de conception conjointe

Critère d'évaluation	*Chinook*	*CodeSign*	*COSYMA*	*CoWare*	*GRAPE II*
Spécification et modélisation	Description comportementale commune des composants logiciels et matériels : *modal process*	Réseau de Petri temporel (TPN) orienté-objet	Processus communicants (CSP) décrits en langage C^x	Processus communicants hétérogènes décrits en C, C++, DFL ou VHDL	Graphe de dépendances de données ou langage flot de données DFL
Partitionnement	Manuel	Manuel	Automatique, basé sur un algorithme d'optimisation stochastique (*Simulated Annealing*)	Automatique	Automatique, basé sur une heuristique du type *branch-and-bound* [BELP93]
Synthèse matérielle et compilation	Génère la *netlist* pour le matériel et du code C pour le logiciel	Génère du VHDL comportemental pour le matériel et du C pour le logiciel	Génère du HardwareC synthétisé par *Olympus* et du code C compilé par des compilateurs standards	Génère du VHDL pour le matériel et du code C pour le logiciel	Génère du code VHDL pour le matériel et du code C ou de l'assembleur pour le logiciel
Simulation et validation	Utilise l'outil de simulation *Pia* [HB97]	Les modèles peuvent être validés par simulation ou par analyse formelle	Simulation logicielle et analyse formelle après ordonnancement	Simulateurs *ARM* pour le code C et *Synopsis* pour le VHDL	Simulation logicielle
Intégration d'outils	Non	Outils de synthèse RTL	*BBS* pour HLS et *Synopsis Design Compiler* pour la synthèse RTL	Compilateurs C *ARM* et VHDL *Synopsis* et ; environnement *Cathedral*	*DSP Station* pour la spécification et *Synopsis* pour la synthèse

Critère d'évaluation	*Polis*	*Ptolemy*	*Cierto VCC*	*DSP Station*
Spécification et modélisation	Langage de haut niveau (Esterel, FSM graphique, Verilog ou VHDL)	Modèles de calcul CSP, CT, DE, FSM, PN, SDF, DDF, SR, Thor et VHDL	Description comportementale sous la forme de blocs fonctionnels (C, C++, SDL, HDL)	Diagramme de blocs, code C ou langage flot de données
Partitionnement	Manuel	Manuel	Manuel	Manuel
Synthèse matérielle et compilation	Génère du VHDL pour le matériel et du code C pour le logiciel	Génère du code *Silage* pour le matériel et du code C pour le logiciel	Génère du HDL pour le matériel et du code C pour le logiciel	Génère du VHDL comportemental synthétisable pour le matériel et du code C pour le logiciel
Simulation et validation	Possible à travers des outils développés pour les FSM : *Ptolemy*	Simulation logicielle et matérielle	Co-vérification par l'outil *Affirma*	Simulation et vérification possibles
Intégration d'outils	Compilateurs C, outils de synthèse RTL et *Ptolemy*	*Cadence SPW*, *HP DSP Designer* et autres	Bonne intégration avec des outils de création d'IP, comme C, C++, MatLab, SDL, *Cadence Cierto SPW*	Bonne intégration dans la chaîne CAO de *Mentor Graphics* (*DSP Design Lab*, *Mistral*). Intégration difficile des circuits ou logiciels implantés avec d'autres outils

Ptolemy intègre le plus grand nombre de modèles de calcul, ce qui offre au concepteur différents niveaux d'abstraction pour la spécification fonctionnelle de l'application. Pourtant, les modèles basés sur des descriptions comportementales sous la forme de diagrammes de blocs ou flots de données (*CoWare*, *GRAPE II*, *Ptolemy*, *Cierto VCC* et *DSP Station*) sont bien appropriés pour la description de systèmes mixtes matériel/logiciel.

Le partitionnement guidé ou automatique (*COSYMA*, *CoWare*, *GRAPE II* et *Ptolemy*) permet au concepteur d'explorer plus facilement l'espace de solutions, à partir des résultats obtenus par prédiction de performances, par co-vérification et/ou co-simulation. Le concepteur peut toujours imposer l'implantation d'un sous-ensemble de l'application sur des composants matériels ou logiciels en fonction de ses besoins.

Nous pouvons supposer que le choix de générer du VHDL pour la synthèse de la partie matérielle et du code C pour la compilation de la partie logicielle semble le plus approprié, puisqu'il est présent dans la plupart des outils (*CodeSign*, *CoWare*, *GRAPE II*, *Polis*, *Cierto VCC* et *DSP Station*). Pourtant, il faut s'assurer que le VHDL généré soit synthétisable et que le code C soit compatible avec la norme ANSI (*American National Standards Institute*).

CodeSign et *DSP Station* permettent au concepteur d'effectuer la simulation logicielle et matérielle de l'implantation et offrent aussi la possibilité de réaliser la vérification ou l'analyse formelle. Pourtant, seul *Cierto VCC*, par l'intermédiaire de l'outil *Affirma*, permet d'effectuer la co-vérification.

Tous les environnements présentés intègrent plus ou moins des fonctionnalités souhaitables pour un système de conception conjointe, mais aucun d'entre eux n'offre la gamme complète de ces fonctionnalités. *Chinook* nous semble le système le moins performant, tandis que *CoWare* est celui qui satisfait le plus aux critères d'évaluation. *Ptolemy II* est l'environnement qui doit évoluer le plus dans les années à venir, intégrant un modèle unifié de conception, utilisable de la co-spécification à la co-synthèse.

Comme aucun des environnements ci-mentionnées est capable de satisfaire simultanément à tous les critères d'évaluation des outils de conception conjointe pour l'implantation optimisée des algorithmes de TSI bas, moyen et haut niveau, nous nous sommes convaincus de la nécessité d'essayer de combler ce manque. Nous avons choisi donc de travailler sur une extension de la méthodologie AAA développée à l'INRIA–Rocquencourt, laquelle produit des implantations optimisées pour des architectures basées sur des circuits programmables, qui sont appropriés aux algorithmes de TSI moyen et haut niveau. Il fallait ainsi développer une méthodologie AAA capable d'implanter des algorithmes bas niveau de TSI sur des circuits reconfigurables ou des ASIC. Comme nous verrons par la suite, l'intégration de cette extension de la méthodologie AAA au logiciel *SynDEx* qui la supporte nous permettra d'avoir un outil de conception conjointe très performant par rapport aux environnements existants.

1.2 Méthodologie AAA développée à l'INRIA–Rocquencourt

La méthodologie "Adéquation Algorithme-Architecture" (AAA) cherche à produire de façon automatique des exécutifs optimisés distribués temps réel pour les composants processeurs des architectures mono ou multi-processeurs [GMP*90, LSSS91a, LSSS91b, ELS92, LS92, LMPRS93, BLLMS94, LS94, Sor94]. Cette méthodologie, développée à l'INRIA-Rocquencourt (Projet SOSSO), part d'une spécification algorithmique de l'application, fournie par l'utilisateur, sous la forme d'un graphe flot de données. Pour générer l'exécutif optimisé, la méthodologie prend en compte un modèle d'architecture composée d'un ou plusieurs processeurs, sur lequel on doit effectuer l'implantation matérielle de la spécification algorithmique.

Cette méthodologie AAA de prototypage rapide optimisé est fondée sur une approche globale, formalisant l'algorithme, l'architecture et l'implantation, à l'aide d'un modèle unifié de *graphes factorisés* (voir figure 1.3). L'intérêt principal de ce modèle réside dans sa capacité à exprimer tout le parallélisme, non seulement dans le cas de l'algorithme (parallélisme potentiel, exprimé à travers un graphe logiciel : graphe flot de données) et de l'architecture (parallélisme disponible, exprimé à travers un graphe matériel : interconnexion de composants), mais aussi dans le cas de l'implantation de l'algorithme sur l'architecture. Ainsi, on manipule le même modèle pour l'algorithme, l'architecture et l'implantation.

L'implantation d'un algorithme sur une architecture est un problème d'allocation de ressources. Il existe deux stratégies différentes pour cette allocation de ressources : dynamique ou statique. L'allocation statique est la plus appropriée pour l'implantation d'algorithmes de traitement du signal et des images et de contrôle réactif temps réel sur des systèmes embarqués. Dans ce cas, l'algorithme et son environnement sont bien connus, les ressources matérielles sont normalement limitées et les contraintes temporelles très sevères. En tenant compte de ces caractéristiques, AAA utilise l'ordonnancement statique (même les communications sont routées et ordonnancées statiquement) [GLS99].

L'adéquation cherche à mettre en correspondance l'algorithme et l'architecture, en réduisant le parallélisme potentiel de l'algorithme au parallélisme disponible de l'architecture et en transformant le graphe logiciel correspondant à l'algorithme en un graphe matériel correspondant à l'architecture qui implante l'algorithme, comme le montre la figure 1.3.

1.2.1 Modèle de l'algorithme

Un algorithme, selon Turing et Post [Tur36], est une séquence (ordre total) finie d'opérations directement exécutable par une machine à états finie (*finite state machine*). Cette définition doit être étendue afin de permettre d'une part la prise en compte du parallélisme disponible dans les architectures distribuées, composées de

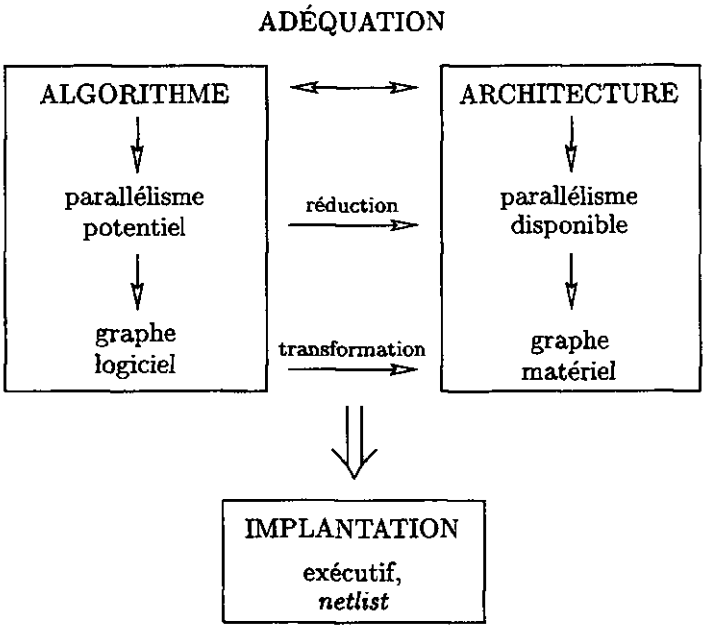


FIG. 1.3: Méthodologie AAA

plusieurs machines à états finies interconnectées, et d'autre part la prise en compte de l'interaction infiniment répétitive de l'application avec son environnement. Ainsi, les algorithmes doivent être spécifiés avec, au minimum, le même degré de parallélisme que l'architecture. Comme on cherche à comparer l'implantation d'un algorithme sur différentes architectures, celui-ci doit être spécifié de façon indépendante de l'architecture.

Pour cela, le modèle d'algorithme utilisé est un *graphe de dépendances factorisé conditionné* : c'est un hypergraphe orienté acyclique, dont les sommets sont des *opérations* partiellement ordonnées (exprimant le parallélisme potentiel de l'algorithme) par leurs dépendances de données. Nous avons besoin d'un modèle d'hypergraphe puisque chaque dépendance de données peut avoir plusieurs extrémités mais une seule origine (diffusion). Ce graphe de dépendance, appélé graphe acyclique orienté (DAG–*Direct Acyclic Graph*), exprime le parallélisme potentiel de l'algorithme : deux opérations qui ne sont pas en relation de dépendance de données peuvent être exécutées dans n'importe quel ordre par le même opérateur ou simultanément par des opérateurs différents [GLS99].

Nous nous intéressons à des systèmes réactifs qui intéragissent constamment avec l'environnement qu'ils contrôlent. Ainsi, les opérations nécessaires au calcul des événements de sortie pour les actionneurs, à partir des événements d'entrée acquis par les capteurs, sont exécutées à chaque interaction avec l'environnement. L'algorithme est donc modélisé par un graphe de dépendances, infiniment large mais périodique, réduit par factorisation à son motif répétitif, appélé *graphe flot de données* (graphe algorithmique).

Une opération peut être, soit un calcul, soit une mémoire d'état (retard : $) ou encore une entrée-sortie. Les sommets qui ne possèdent pas de prédécesseur sont les interfaces d'entrée (capteurs recevant des stimuli de l'environnement) et ceux qui ne possèdent pas de successeur représentent les interfaces de sortie (actionneurs produisant les réactions vers l'environnement). Dans le cas d'une opération de calcul, la consommation des entrées précède la production des sorties. Dans le cas d'un conditionnement, la sortie n'est produite que si l'entrée booléenne est vraie. Par contre, la sortie d'un retard précède son entrée.

Nous nous intéressons ici principalement aux algorithmes bas niveau de TSI. Ils impliquent généralement des volumes de calculs suffisamment importants pour faire apparaître des aspects réguliers. Cette régularité est représentée à travers la répétition d'un sous-graphe (motif répétitif). La factorisation de ce motif permet de transformer la répétition en itération, outre la réduction du volume de la spécification. Dans le cas d'une implantation multi-processeurs, la distribution des motifs est faite pour que chaque processeur exécute une partie de l'itération. La figure 1.4 montre un exemple de graphe algorithmique (identification d'un filtre transversal avec un égaliseur adaptatif). Le signal d'*Entrée* (un signal pseudo-aléatoire) est envoyé d'une part sur le filtre à identifier (*Filt*) et d'autre part sur le filtre adaptatif (*FiltA*). On calcule l'erreur en faisant la différence (*Sub*) entre la sortie estimée par le filtre adaptatif et le signal sortant du filtre à identifier. Cette erreur sert à calculer, à l'aide d'un algorithme de gradients stochastiques (*Adap*), les coefficients du filtre adaptatif. On visualise l'erreur en temps réel (*Sortie*), afin de vérifier la convergence. $ représente un retard [LSSS91b].

Les graphes algorithmiques peuvent être spécifiés directement par l'utilisateur ou être obtenus par l'intermédiaire de l'analyse des dépendances d'une spécification purement séquentielle ou CSP. Sinon, afin de détecter les erreurs de conception dans le cycle de développement le plus tôt possible, les graphes algorithmiques peuvent être produits par des compilateurs de langages de spécification de haut niveau, comme les langages synchrones [Hal93] (Esterel, Lustre, Signal, en utilisant le format DC), qui réalisent des vérifications formelles pour éviter, p. ex., les *deadlocks* (interblocages) [GLS99].

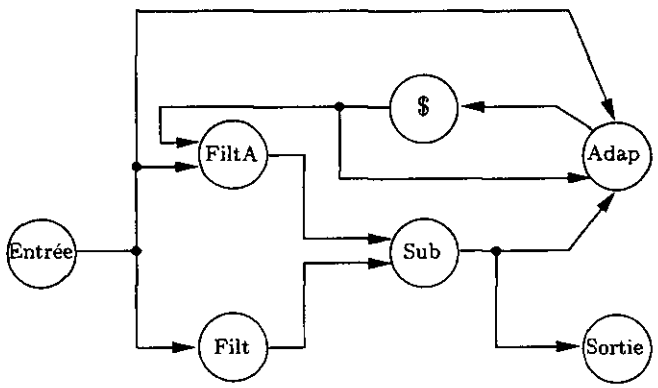


FIG. 1.4: Exemple de graphe algorithmique (identification d'un filtre)

1.2.2 Modèle d'architecture

Les modèles les plus souvent utilisés pour la spécification d'architectures distribuées ou parallèles sont le PRAM (*Parallel Random Acces Machine*) et le DRAM (*Distributed Random Acces Machine*). Le PRAM correspond à un ensemble de processeurs communiquant à travers une mémoire partagée. Le DRAM correspond à un ensemble de processeurs communiquant par passage de message à travers une mémoire distribuée. Ces modèles suffisent pour décrire la distribution et l'ordonnancement des calculs sur des machines homogènes, mais ils ne sont pas appropriés pour la description de machines hétérogènes. Ils ne sont pas assez précis non plus pour décrire la distribution et l'ordonnancement d'opérations de communication entre processeurs.

Le modèle d'*architecture multicomposant hétérogène* utilisé est un hypergraphe non-orienté (graphe matériel), dont les sommets sont des machines à états finies (séquenceurs d'opérations de calcul ou séquenceurs d'opérations de communication), que nous appelons *opérateurs*, et dont les hyperarcs sont des ressources partagées entre séquenceurs (conducteurs, mémoires RAM ou FIFO–*First In First Out* et arbitres), que nous appelons *médias*. Chaque média de communication exécute des opérations de communication de façon séquentielle. Un média n'inclut pas seulement les conducteurs électriques nécessaires pour transporter les données entre les opérateurs, mais aussi les unités de transformation (DMA–*Direct Memory Access* ou UART–*Universal Asynchronous Receiver/Transmitter*) qui séquencent les accès mémoires de chaque côté des conducteurs.

Il faut dire qu'il y a deux types de composants : (i) ceux avec séquenceur d'instructions, c'est-à-dire des composants programmables (processeurs), auxquels on peut allouer un programme (une séquence d'opérations) et (ii) ceux qui n'ont pas de séquenceur d'instructions (ASIC et FPGA), auxquels on ne peut affecter qu'une opération. L'hétérogénéité de l'architecture ne signifie pas seulement que les opérateurs et les médias peuvent avoir chacun des caractéristiques différentes (durée d'exécution des opérations de calcul et de communication), mais aussi que certaines opérations ne peuvent être exécutées que par certains opérateurs, ce qui permet de décrire aussi bien les composants programmables (processeurs), que les composants spécialisés (ASIC figés ou FPGA reconfigurables). Le graphe matériel décrit le parallélisme disponible de l'architecture.

Ce modèle d'architecture est une extension du modèle RTL classique. On l'appelle *Macro-RTL*, parce que chaque opération du graphe algorithmique est une *macro-instruction* (une séquence d'instructions ou un circuit combinatoire) et chaque dépendance de données est un *macro-registre* (un banc de cellules mémoire contigües) [Syn99]. Ce modèle encapsule les détails concernant l'ensemble d'instructions, le microcode, le pipeline, la cache, etc., permettant d'éviter de prendre en compte ces caractéristiques pendant l'optimisation. Ce modèle est bien approprié aux techniques d'optimisation utilisées pour le prototypage rapide, qui doit être précis et efficace.

Dans le cas d'une architecture multi-processeurs, chaque sommet du graphe matériel représente un processeur et chaque arc représente une liaison physique

bidirectionnelle de communication, permettant de transférer des données entre les mémoires des processeurs. Un processeur comprend une unité de calcul (CAL), une unité d'entrée/sortie (E/S) pour chaque interface avec l'environnement, une unité de communication (COM) pour chaque arc adjacent et une unité mémoire partagée (MEM) avec les autres unités. Chaque liaison de communication comprend un médium physique de communication et les unités de communication connectées au médium. La figure 1.5 montre un exemple de graphe matériel : une architecture multiprocesseurs (P_1, P_2, P_3), où chaque processeur est connecté à un bus commun et les processeurs P_1 et P_2 peuvent s'échanger des données à travers une connexion directe entre eux sans utiliser le bus [LS92].

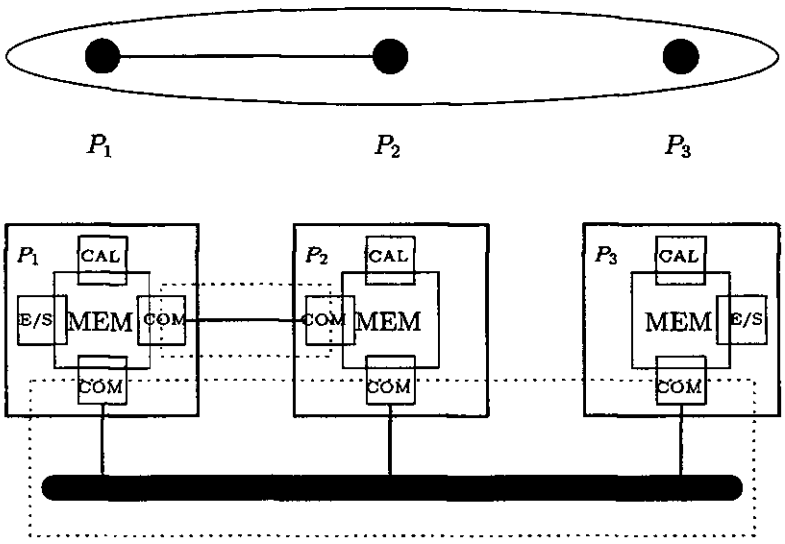


FIG. 1.5: Exemple de graphe matériel (architecture multiprocesseur)

1.2.3 Modèle d'implantation

Le modèle d'implantation part du principe que chaque opération du graphe algorithmique n'exige qu'un seul opérateur pour son exécution, et qu'il y a, au minimum, un opérateur du graphe matériel capable d'exécuter une opération du graphe algorithmique. Un de ces opérateurs doit être choisi pour l'exécuter. Quand deux opérations en relation de dépendance de données sont exécutées par le même opérateur, l'opération qui produit les données doit être exécutée avant l'opération qui consomme les données. Les données produites par l'opérateur lors de l'exécution de la première opération de la séquence doivent être stockées dans des registres, avant d'être consommées lors de l'exécution de la deuxième opération. Quand deux opérations en dépendance de données sont exécutées par des opérateurs différents, il faut s'assurer que l'opérateur producteur des données a fini son exécution, et que les données sont stables en sa sortie avant d'exécuter l'opérateur consommateur. Ces dépendances de données sont appélées dépendances de données inter-opérateurs.

Dans le cas des architectures multiprocesseurs, l'*implantation* d'un algorithme sur une architecture est une *distribution* et un *ordonnancement* des opérations de l'algorithme sur les opérateurs de calcul de l'architecture. Il va de même pour les opérations de communication, qui découlent de la distribution, sur les opérateurs de communication. La distribution et l'ordonnancement des opérations, ainsi que leur optimisation dans le cas des architectures homogènes, ont été formalisées en termes de transformations de graphes [Sor96].

La distribution (figure 1.6) consiste d'une part à affecter chaque opération de l'algorithme à un opérateur de calcul capable de l'exécuter, et d'autre part, à choisir une *route* entre les deux opérateurs (chemin dans le graphe de l'architecture) pour chaque dépendance de données inter-opérateur. Elle consiste également à créer et à insérer entre deux opérations de l'algorithme autant d'opérations de communication qu'il y a d'opérateurs de communication sur la route et à affecter chacune de ces opérations de communication à l'opérateur de communication correspondant. L'ordonnancement consiste à linéariser (rendre total), pour chaque opérateur, l'ordre partiel entre les opérations qui lui ont été affectées ; c'est-à-dire, à ajouter des dépendances d'ordonnancement au graphe de l'algorithme.

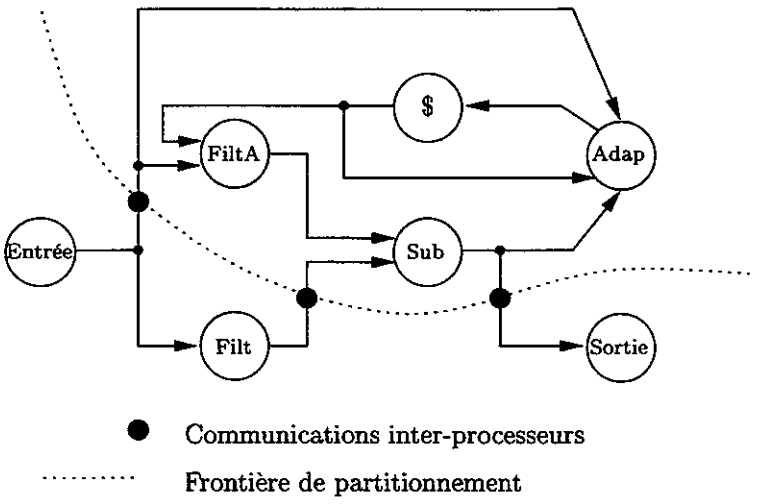


FIG. 1.6: Exemple de distribution

Une implantation est donc le graphe résultat d'une transformation du graphe de l'algorithme (ajout des opérations de communication et des dépendances d'ordonnancement), influencée par le graphe de l'architecture. Elle est formalisée comme une composition de trois relations : le routage, la distribution et l'ordonnancement ; chacune d'entre elles s'appliquant à un couple de graphes (algorithme, architecture). Étant donnés un graphe d'algorithme et un graphe d'architecture, il existe un nombre fini, voire très grand, d'implantations possibles, avec des performances (latences, cadences) différentes. Ces performances sont obtenues par calculs des chemins critiques (latences) et de boucles critiques (cadences) sur le graphe de l'implantation. Ce graphe est étiqueté par les durées d'exécution caractéristiques des opérateurs de l'architecture.

1.2.4 Optimisation de l'implantation

L'adéquation entre un algorithme et une architecture, tenant compte des contraintes temps réel et d'embarquabilité, est un problème complexe d'optimisation, où l'algorithme peut être profondément transformé (modification de la granularité des opérations, spécification de nouveaux parallélismes potentiels), aussi que l'architecture peut aussi être transformée (modification du nombre et/ou des caractéristiques des composants).

Le problème d'optimisation de cette méthode AAA consiste à minimiser les ressources matérielles, tout en respectant des contraintes temporelles et technologiques. L'algorithme, l'architecture et les contraintes sont définis a priori. Ainsi, le problème d'optimisation se réduit à une adéquation entre un algorithme et une architecture, dont les granularités et la topologie ont été pré-définies. Le problème reste NP-complet, même s'il a été réduit. Cela signifie que le nombre d'implantations possibles d'un algorithme sur une architecture dans le cas d'une application réelle est tellement important qu'il est impossible de trouver la solution optimale par recherche exhaustive dans tout l'espace de solutions. Il faut donc utiliser des heuristiques capables de nous donner des solutions approximatives. Pour le prototypage rapide, les heuristiques "gourmandes" semblent bien appropriées puisqu'elles sont très rapides et donnent des bons résultats.

L'heuristique utilisée est du type *ordonnancement de liste*, améliorée pour manipuler les communications inter-opérateurs (la distribution et l'ordonnancement prennent en compte les routes parallèles et les conflits provoqués par les ressources partagées), l'hétérogénéité des opérateurs et des médias, et le conditionnement des opérations. La première solution doit être affinée par le *back-tracking* (retour en arrière) itératif des solutions équivalentes choisies. La solution ainsi obtenue doit être considérée comme une solution initiale pour des heuristiques plus précises, mais plus lentes, de type *tabu* ou *simulated annealing* [Syn99].

1.2.5 Génération d'exécutifs

La génération d'exécutifs pour les processeurs correspond au codage d'une implantation suivant le modèle macro-RTL de l'architecture. La séquence d'opérations sur chaque opérateur est codée par une séquence de macro-instructions. Les dépendances de contrôle sont traduites par l'ordre du codage. Toute dépendance de données est codée par un macro-registre qui est un argument pour une macro-instruction productrice (origine de la dépendance de données) et pour une macro-instruction consommatrice (destination de la dépendance de données). Ces macro-instructions sont implantées de façon triviale par la séquence de macro-opérations dépendantes quand celles-ci appartiennent au même processeur, mais elles exigent des macro-instructions de synchronisation quand les macro-opérations appartiennent à des processeurs différents.

La génération d'exécutifs produit plusieurs fichiers-source, un pour chaque mémoire de programme, pouvant être partagée par plusieurs opérateurs. Chaque fichier-source est un code intermédiaire composé d'une liste de macros, qui sont traduites par un macro-processeur en code-source approprié pour l'opérateur. Il y a deux ensembles de macro-instructions. Le premier est un ensemble extensible, spécifique à l'algorithme de l'application. Il correspond à l'implantation des opérations de calcul. Le deuxième (*generic executive kernel*) est un ensemble de macro-instructions qui implante le chargement initial de la mémoire de programme, la gestion de mémoire, le séquencement, les transferts de données inter-opérations, la synchronisation des macro-registres et les mesures de performances temporelles.

1.2.6 Formalisation de la méthode AAA

L'AAA peut être formalisée [LS93] de la manière suivante :

soit $\mathcal{A}$ l'ensemble des opérations, $\mathcal{D}$ l'ensemble des dépendances inter-opérations, $\mathcal{P}$ l'ensemble des processeurs et $\mathcal{L}$ l'ensemble des liaisons physiques de communications inter-processeurs. Le graphe matériel est donc une paire $(\mathcal{P}, \mathcal{L})$ et le graphe algorithmique est une paire $(\mathcal{A}, \mathcal{D})$. La *distribution* consiste à partitionner l'ensemble $\mathcal{A}$ des opérations du graphe algorithmique en n partitions notées $\mathcal{A}_p$ (où n correspond au nombre de processeurs dans l'ensemble $\mathcal{P}$, et p correspond au p-ème processeur, $p \in 1, \ldots, n$) et à ordonnancer les opérations qui lui ont été affectées sur l'unité de contrôle de chaque processeur. Ce partitionnement entraîne des communications inter-processeurs. Ces communications sont aussi distribuées et ordonnancées sur les liaisons de communications. Même si le graphe matériel est forcément connexe, chaque processeur n'est pas toujours directement connecté à tous les autres. Pour supporter toutes les communications inter-processeurs, on construit l'ensemble $\mathcal{R}$, qui dénote tous les chemins (ou routes) du graphe matériel $(\mathcal{P}, \mathcal{L})$. Les routes, comme les liaisons de communication, sont bidirectionnelles. La transformation du graphe matériel initial en un graphe matériel routé est appélée *routage* :

$$(\mathcal{P}, \mathcal{L}) \xrightarrow{routage} (\mathcal{P}, \mathcal{R}) \tag{1.1}$$

et l'on distribue les communications inter-processeurs sur les routes :

$$\Big((\mathcal{A}, \mathcal{D}), (\mathcal{P}, \mathcal{R})\Big) \xrightarrow{distrib_{\mathcal{R}}} \left(\bigcup_{p \in \mathcal{P}} (\mathcal{A}_p, \mathcal{D}_p), \bigcup_{r \in \mathcal{R}} \mathcal{D}_r\right) \tag{1.2}$$

$\mathcal{A} \supset \mathcal{A}_p$, où $\mathcal{A}_p$ correspond aux opérations exécutées par le processeur p,
$\mathcal{D} \supset \mathcal{D}_p$, où $\mathcal{D}_p$ correspond aux dépendances entre les opérations exécutées par p,
$\mathcal{D} \supset \mathcal{D}_r$, où $\mathcal{D}_r$ correspond aux dépendances inter-processeurs routées sur $r \in \mathcal{R}$.

Dans le cas d'un ordonnancement dynamique, c'est l'exécutif qui décide de l'ordonancement à l'exécution. Dans le cas de l'ordonnancement statique implanté par *SynDEx*, la fonctionnalité de l'exécutif est supportée par le séquenceur du processeur. Jusqu'à maintenant, nous avons considéré que seul le nombre de processeurs était limité. Pour considérer la limitation du nombre de liaisons de communication inter-processeurs, chaque arc inter-partitions $d_r \in \mathcal{D}_r$ doit être transformé en une chaîne (graphe linéaire) comportant un sommet pour chaque liaison de la route suivant la transformation formulée ci-dessous :

$$d_r \xrightarrow{com} (d_p, c_l, d_{p'}, \ldots, d_{p''}, c_{l''}), \quad \forall r \in \mathcal{R}, \ \forall d_r \in \mathcal{D}_r \tag{1.3}$$

Chaque nouveau sommet c_l est une opération de communication qui correspond à un transfert de données entre les mémoires de deux processeurs directement connectés par une liaison physique de la route ($l \in \mathcal{L}$). Ces opérations de communication sont donc implantées par un transfert de mémoire à mémoire et une synchronisation. Toutes les unités de communication connectées à l coopèrent pour exécuter le transfert. Nous pouvons donc considérer que chaque opération de communication est distribuée sur les unités de communication partageant la liaison physique de communication. Les nouveaux arcs d_p sont des transferts intra-processeurs inter-unités (calcul-communication, communication-communication, communication-calcul) appartenant à $\mathcal{D}_p$.

En regroupant les c_l d'un même $l \in \mathcal{L}$ et les d_p d'un même $p \in \mathcal{P}$, on obtient les ensembles $\mathcal{C}_l$ et $\mathcal{D}_p$ et la transformation $distrib_{\mathcal{R}}$ est modifiée. Cette transformation est appelée *distrib*.

$$\Big((\mathcal{A}, \mathcal{D}), (\mathcal{P}, \mathcal{L})\Big) \xrightarrow{distrib} \left(\bigcup_{p \in \mathcal{P}} (\mathcal{A}_p, \mathcal{D}_p), \bigcup_{l \in \mathcal{L}} \mathcal{C}_l\right) \tag{1.4}$$

La distribution et l'ajout des opérations de communication c_l ne modifient pas l'ordre partiel $\prec$ du graphe algorithmique. Sur chaque processeur, un ordonnancement des opérations est un ordre total $<_p \subset \prec$, qui inclut l'ordre partiel restreint à $\mathcal{A}_p$. Sur chaque liaison de communication, un ordonnancement des opérations de communication est un ordre total $<_l \subset \prec$, qui inclut l'ordre partiel restreint à $\mathcal{A}_l$. Après ces ordonnancements, la transformation *distrib* devient la transformation *dist/ord*, qui distribue et ordonnance le graphe algorithmique sur le graphe matériel :

$$\Big((\mathcal{A}, \mathcal{D}), (\mathcal{P}, \mathcal{L})\Big) \xrightarrow{dist/ord} \left(\bigcup_{p \in \mathcal{P}} (\mathcal{A}_p, \bar{\mathcal{D}}_p, <_p), \bigcup_{l \in \mathcal{L}} (\mathcal{C}_l, <_l)\right) \tag{1.5}$$

1.2.7 Logiciel SynDEx

SynDEx (Exécutif Distribué Synchrone) [GMP*90, Syn99] est un logiciel graphique interactif pour le prototypage rapide et l'optimisation de l'implantation d'applications embarquées temps réel sur des architectures multicomposants, supportant la méthodologie "Adéquation Algorithme Architecture" (AAA) [LSSS91a]. Il permet la distribution de programmes synchrones sur des architectures multi-processeurs. Il effectue également la mise en œuvre distribuée d'un algorithme décrit par un graphe logiciel. Il prend en entrée une spécification de l'algorithme de l'application sous la forme d'un graphe flot de données synchrone (ou un langage synchrone), ainsi que la description de l'architecture-cible sous la forme d'un graphe d'opérateurs.

On saisit, à l'aide de la souris, le graphe de l'algorithme et le graphe de l'architecture. *SynDEx* exécute ensuite l'heuristique d'optimisation, réalisant l'adéquation entre l'algorithme et l'architecture, optimisant le temps de réponse. Il s'agit de transformer le graphe logiciel jusqu'à ce qu'il corresponde au graphe matériel. Les transformations pour le faire sont la distribution (allocation spatiale) et l'ordonnancement (allocation temporelle) des calculs et des communications. *SynDEx* effectue cette distribution et cet ordonnancement de façon statique. Il assure que la spécification de l'algorithme et la description de l'architecture conduiront à une implantation correcte sur une machine multi-processeurs.

À partir de cette distribution, un exécutif temps réel fiable est généré automatiquement, permettant l'exécution de l'algorithme sur l'architecture et libérant l'utilisateur des tâches lourdes de programmation bas niveau. Le générateur d'exécutifs de *SynDEx* transforme le graphe flot de données qui spécifie l'application en un graphe flot de contrôle codé par l'exécutif. Cet exécutif a pour rôle d'allouer les ressources de l'architecture (unités de calcul, de mémoire et de communication) au programme d'application. *SynDEx* cible actuellement les architectures à base de transputers (T80X), de processeurs de traitement du signal (TMS320C40 et SHARC-ADSP21060), de micro-contrôleurs (i8051, i8096, MC68332), de processeurs génériques (i80386) et des stations de travail UNIX/C/TCP/IP (Sun, DEC, SGI, HP, PC Linux) [GLS99]. Ainsi, *SynDEx* nous permet de :

- spécifier l'algorithme d'une application sous la forme d'un graphe flot de données conditionné (ou par l'intermédiaire d'une interface avec le compilateur d'un langage synchrone utilisant le format DC) ;
- spécifier l'architecture multi-composants sous la forme d'un graphe d'opérateurs ;
- distribuer et ordonnancer l'algorithme sur l'architecture par l'intermédiaire d'une heuristique qui effectue l'optimisation du temps de réponse ;
- visualiser les performances temporelles estimées de l'implantation de l'algorithme sur l'architecture-cible ;
- générer des exécutifs pour l'exécution temps réel sur l'architecture multi-composants avec des mesures de performances temporelles.

Une méthodologie de conception conjointe, se basant sur *SynDEx* comme langage de spécification, permet de profiter des outils de spécification, d'analyse et de validation au niveau système offerts par les environnements des langages synchrones [Bel94].

SynDEx a été utilisé pour développer plusieurs applications hétérogènes temps réel, parmi lesquelles [KS98] le prototype d'un véhicule urbain controllé par un système distribué embarqué (basé sur un bus CAN–*Controller Area Network* et cinq micro-contrôleurs MC68332, un micro-processeur i80c196, et un micro-processeur i80486), qui assure la conduite autonome et le choix du trajet.

Nous avons décrit en détails les caractéristiques de la méthodologie AAA et du logiciel *SynDEx* qui la supporte. Cela nous permet donc de situer notre extension de la méthodologie AAA aux circuits reconfigurables par rapport à l'AAA orientée multicomposants.

1.3 Extension de l'AAA aux circuits reconfigurables

Dans le cas de l'AAA orientée multicomposants, l'implantation consiste à distribuer et à ordonnancer les opérations sur un nombre d'opérateurs équivalent au nombre de composants programmables. Le parallélisme de l'implantation est donc plus réduit que dans l'AAA orientée circuits reconfigurables où l'implantation consiste à défactoriser la spécification factorisée, à distribuer et à ordonnancer les opérations sur un grand nombre de cellules. Dans l'AAA multicomposant, il faut réaliser l'adéquation entre un modèle d'algorithme et un modéle d'architecture, tout en respectant un modèle d'implantation qui préconise une distribution et un ordonnancement des opérations de l'algorithme sur les opérateurs de calcul de l'architecture. Dans l'AAA circuits reconfigurables (multi-FPGA), la technique consisterait aussi à effectuer une adéquation entre un modèle d'algorithme et un modèle d'architecture, mais en respectant un modèle d'implantation tout à fait différent, qui préconiserait un partitionnement du graphe algorithmique en fonction du graphe matériel et une distribution des opérations de l'algorithme sur les cellules logiques des FPGA. Pourtant, nous nous restreindrons ici aux architectures mono-FPGA. Dans ce cas, la technique d'implantation consiste tout simplement à distribuer les opérations de l'algorithme sur l'ensemble de cellules logiques des FPGA, puisque notre modèle d'architecture se résume à un seul circuit. Pour l'AAA multicomposant, étant donnée une architecture, la stratégie d'optimisation consiste à minimiser le temps de réponse. Pour l'AAA FPGA, la défactorisation permet d'obtenir une implantation qui exploite le parallélisme massif de l'architecture. La stratégie d'optimisation consiste donc à minimiser l'augmentation de la consommation de ressources matérielles dûes à la défactorisation, étant donné un temps de réponse borné (voir tableau 1.2).

L'utilisation d'architectures basées sur des circuits reconfigurables introduit de nouveaux défis et de nouvelles possibilités pour les systèmes de conception conjointe matériel/logiciel en ce qui concerne l'ordonnancement, le partitionnement,

TAB. 1.2: Comparaison entre l'AAA multicomposant et l'AAA FPGA

Caractéristique	*AAA multi-composant*	*AAA FPGA*
Algorithme	- relation d'ordre partiel ; - parallélisme d'opérations	- relation d'ordre partiel ; - parallélisme d'opérations
Architecture	- un séquenceur d'opération par processeur, soit de calcul (exécutées par l'Unité Logique-Arithmétique du processeur), soit de communication (exécutées par l'unité DMA de transfert intermédia) ; - parallélisme réduit ; - multi-composant	- grand nombre de cellules logiques et d'interconnexion intra et inter-FPGA configurées statiquement ; - parallélisme massif ; - multi-FPGA
Implantation - distribution	- opérations distribuées sur un petit nombre d'opérateurs ; - il suffit d'allouer un seul opérateur pendant un certain temps pour exécuter une opération	- opérations distribuées sur un grand nombre de cellules ; - il faut allouer plusieurs cellules pendant un certain temps pour exécuter une opération
- défactorisation	- bénéfices limités par le parallélisme réduit de l'architecture ; - contraintes architecturales limitant l'espace de solutions	- permet de mieux utiliser le parallélisme massif de l'architecture ; - espace de solutions plus large
- ordonnancement	- le même opérateur exécute des opérations différentes à des instants différents ; - il faut séquencer les opérations sur chaque opérateur ; - les dépendances de données inter-séquences impliquent la synchronisation inter-séquences ; - beaucoup d'ordonnancemment ; - peu de synchronisation à gérer ; - allocation séquentielle des opérateurs aux opérations ; - allocation mémoire aux dépendances de données inter-opérateurs	- les mêmes cellules exécutent le même type d'opération sur des données différentes à des instants différents ; - il faut multiplexer les données ; - les dépendances de données inter-registres impliquent la synchronisation des transferts de registres ; - peu d'ordonnancement ; - beaucoup de synchronisations à gérer ; - allocation plutôt parallèle des cellules aux opérations ; - allocation de ressources d'interconnexion aux dépendances de données
Génération de code	- programmes séquentiels communicants (configuration de la mémoire programme)	- *netlist* (configuration des cellules et de leurs interconnexions)
Stratégie	- architecture donnée : minimiser le temps de réponse	- temps de réponse donné : minimiser l'architecture

l'exploration et l'extraction de régularité, les techniques d'optimisation et l'automatisation du flot de conception [DW99]. Les FPGA représentent une alternative face à l'implantation matérielle sur VLSI (*Very Large Scale Integration*), ASIC et PAL, et/ou logicielle sur DSP, micro-contrôleurs et micro-processeurs. Les architectures basées sur des FPGA peuvent atteindre des performances 100 fois supérieures à celles basées sur des processeurs et offrir des performances entre 10 et 100 fois supérieures par unité de surface de silicium [V*96, DeH98].

Les avantages offerts par l'utilisation des architectures reconfigurables sont [DW99] :

- une plus grande densité de calcul que les architectures à base de processeurs ;
- une plus grande puissance sémantique[2] et un contrôle de grain plus fin sur les opérateurs de bits pour les machines à mots étroits (*narrow word*) ;
- la réutilisation de la surface de silicium pour les échelles temporelles de gros grain ;
- la capacité d'adapter l'implantation aux exigences de calculs instantanés, minimisant la quantité de ressources nécessaires pour effectuer un calcul.

Contrairement aux architectures basées sur les processeurs qui effectuent des calculs temporels, les architectures reconfigurables effectuent des calculs spatiaux. Dans une implantation spatiale, chaque opérateur est implanté dans un point différent de l'espace, permettant l'exploration du parallélisme afin d'obtenir des cadences plus élevées et des latences de calculs plus réduites. Dans une implantation temporelle, un nombre plus petit de ressources de calcul plus génériques sont réutilisées dans le temps, permettant l'implantation compacte du calcul [DW99].

L'extension de la méthodologie AAA que nous proposons dans ce travail consiste à son application aux circuits reconfigurables de type FPGA. Dans un premier temps, nous ne nous intéressons qu'aux implantations matérielles mono-FPGA. Ainsi, nous partons d'un modèle algorithmique de l'application pour obtenir par transformation un modèle d'implantation qui respecte les contraintes temps réel de l'application sur un modèle architectural de type mono-FPGA.

L'extension de la méthodologie AAA aux circuits reconfigurables et son intégration à l'environnement *SynDEx* permettra son utilisation pour la conception conjointe matériel/logiciel de systèmes embarqués hétérogènes temps réel. Le concepteur spécifie l'algorithme de l'application sous la forme d'un graphe flot de données ou par l'intermédiaire d'un des langages synchrones, de façon totalement indépendante de l'architecture multi-composants qui doit l'implanter. L'architecture, à son tour, est spécifiée sous la forme d'un graphe d'opérateurs interconnectés. Chaque opérateur y représente soit un composant programmable (partie logicielle), soit un composant configurable (partie matérielle). À travers des heuristiques guidées

[2]La configuration de l'architecture est effectuée par l'intermédiaire de langages orientés au matériel, tels que le VHDL, qui sont beaucoup plus efficaces que les langages de programmation des processeurs, comme l'assembleur et le C.

par l'utilisateur et par les contraintes matérielles (caractérisation des composants logiciels et matériels) et temporelles (latence et/ou cadence), *SynDEx* effectue le partitionnement (distribution et ordonnancement) des opérations du graphe algorithmique sur le graphe matériel. L'exécutif optimisé généré pour les composants programmables et le code VHDL synthétisable généré pour les composants configurables sont respectivement compilé et synthétisé par les outils correspondants. Le tableau 1.3 résume l'évaluation de l'environnement *SynDEx* comme outil de conception conjointe selon les critères présentés dans la section 1.1.4. Le grand avantage de cet environnement est l'utilisation d'un même modèle pendant toutes les étapes de la conception.

TAB. 1.3: Conception conjointe avec *SynDEx*

Critère d'évaluation	*SynDEx*
Spécification et modélisation	Graphe flot de données pour l'algorithme (langages synchrones) ; graphe d'opérateurs pour l'architecture
Partitionnement	Manuel, automatique ou guidé
Synthèse matérielle et compilation	Génération du VHDL structurel synthétisable pour les composants reconfigurables et de l'exécutif optimisé, sous la forme d'un code-source (C, assembleurs) pour les composants programmables
Simulation et validation	Intégration des outils de prédiction de performance temporelle ; analyse et vérification formelle au niveau système avec les environnements des langages synchrones
Intégration d'outils	Compilateurs spécifiques pour l'exécutif des composants processeurs (logiciel) et outils de CAO pour la génération des *netlists* de configuration des circuits à partir du VHDL (matériel)

1.4 Conclusion

Quelques outils de conception de systèmes permettent actuellement la conception conjointe matériel/logiciel et son implantation hétérogène (processeurs programmables et circuits spécialisés). La plupart de ces systèmes sont basés sur le paradigme d'une spécification simple associée à un seul langage de spécification [Chi99, Cos98]. Le paradigme de spécification reflète l'application-cible, laquelle est souvent orientée-contrôle. Cette approche, qui présente un important potentiel analytique, permet d'automatiser le partitionnement matériel/logiciel, l'analyse formelle des propriétés temporelles ou l'optimisation en fonction des contraintes temporelles [RVBM96]. D'autres outils de conception [Pto99, Pto00, RVBM96] partent d'une spécification hétérogène qui mélange différents paradigmes et langages, et qui n'est pas orientée-application. *CoWare*, p. ex., est orienté-implantation.

Dans ce chapitre, nous avons présenté un état de l'art sur la conception conjointe matériel/logiciel (*hardware/software codesign*), quelques critères utilisés pour évaluer les outils de conception conjointe et les principaux (ou les plus cités dans la bibliographie) environnements de conception conjointe existants . Nous avons aussi exposé les principes généraux de la méthodologie AAA orientée-multicomposants développée à l'INRIA-Rocquencourt et de l'environnement *SynDEx* qui l'implante. Nous proposons une extension de cette méthodologie aux circuits reconfigurables du type FPGA (nous nous restreignons ici aux architectures mono-FPGA), ce qui permettra à l'environnement *SynDEx* de réaliser la conception conjointe matériel/logiciel de systèmes multi-composants. Cette méthodologie utilise un modèle unifié pour décrire l'algorithme, l'architecture et l'implantation. Ce modèle est basé sur la théorie des graphes.

Dans le chapitre suivant, nous présentons le modèle unifié de graphes factorisés et son utilisation pour spécifier des algorithmes bas niveau de traitement du signal et des images.

Chapitre 2

Spécification algorithmique et modèle unifié de graphes factorisés

There is a not unique universal specification language to support the whole life cycle (specification, design, implementation) for all kinds of applications. A plethora of specification exists. Each claims superiority but excels only with a restricted application domain [CHL*99].

2.1 Introduction

Le processus de conception est une séquence d'actions qui transforment un ensemble de spécifications décrites de façon non-formelle, en une spécification détaillée qui peut être implantée. Les actions intermédiaires sont normalement caractérisées par une transformation d'une description plus abstraite en une description plus détaillée [ELLS97]. Ce processus de conception doit être basé sur de représentations mathématiquement précises. Ainsi, la vérification et la mise en correspondance de la description initiale peuvent être réalisées à partir d'outils dont la performance est garantie.

Un système embarqué temps réel peut être décrit à différents niveaux d'abstraction. La spécification aux niveaux plus élévés (*système* et *comportemental*) permet au concepteur de décrire toute la fonctionnalité du système, sans s'occuper des détails concernant l'implantation matérielle des composants utilisés pour la description. Face à la complexité croissante de ce type de système, il est souhaitable de le décrire em termes d'un langage de spécification exécutable, qui soit capable d'exprimer la fonctionnalité du système, avec une sémantique permettant la simulation de son fonctionnement et l'automatisation du processus de conception.

D'un point de vue spéculatif, si l'on disposait d'un jeu infiniment riche d'opérations (au sens mathématique), capable de couvrir tous les besoins imaginables, et d'un jeu correspondant d'opérateurs (au sens matériel), capable d'exécuter ce jeu d'opérations, n'importe quel algorithme pourrait être spécifié en utilisant une seule de ces opérations et un seul opérateur correspondant suffirait pour effectuer la transformation d'un ensemble de données en un ensemble de résultats correspondant. En réalité, l'ensemble des opérations implantables, c'est-à-dire les opérations pour lesquelles il existe des opérateurs correspondants est limité. Pour spécifier l'application, il faut donc décomposer son algorithme en termes de cet ensemble d'opérations implantables [LS97].

La décomposition d'un problème en problèmes plus petits permet des gains sur le processus de conception. En découpant le problème, on facilite la réutilisation des parties d'un circuit déjà décrites. C'est le principe "diviser pour régner". L'exploitation de la régularité d'un algorithme nous permet de profiter des spécificités des architectures régulières afin de faciliter ou d'améliorer leur processus de conception [LeM97].

La spécification et la modélisation doivent permettre de décrire aussi bien les fonctionnalités d'un système, que ses contraintes de performance. Ainsi, on associe à un langage de spécification, un modèle qui permet d'analyser et de vérifier les fonctions et les contraintes du système.

Dans notre méthodologie, la spécification algorithmique est le point de départ du processus d'implantation matérielle d'une application sur une architecture. Cette spécification consiste à modéliser l'application sous la forme d'un algorithme, dont les opérations n'ont pour relations d'ordre que celles impliquées par leurs dépendances de données. Celles-ci établissent un ordre partiel qui met en évidence un parallélisme potentiel que ne mettrait pas en évidence un algorithme séquentiel. Pour représenter cette spécification, nous avons choisi un modèle de graphe factorisé de dépendances de données (GFDD) entre opérations (*graphe algorithmique*), où chaque sommet représente, soit une opération de base (qui combine des données en entrée pour produire un(des) résultat(s) en sortie), soit une frontière de factorisation de motifs de sous-graphes répétitifs, et dont chaque arc représente une dépendance de données entre deux sommets. L'étiquetage des sommets et des arcs permet de les caractériser. Chaque sommet représentant une opération ou une frontière de factorisation peut recevoir une étiquette correspondant à sa fonctionnalité algorithmique. Chaque arc représentant une dépendance de données peut recevoir une étiquette correspondant au type des données. En plus, ce modèle permet une représentation graphique des algorithmes, en mettant en évidence le parallélisme potentiel et les dépendances de données. La modélisation basée sur la théorie des graphes nous permet de démontrer formellement les propriétés de la spécification, telles que l'absence de cycles de dépendance.

La spécification de l'algorithme d'une application sous la forme d'un GFDD, décrite en [LS97], comprend des parties régulières (motifs répétitifs périodiques) et non régulières. Cette spécification doit être indépendante de toutes contraintes

liées à l'implantation matérielle et nécessite la mise en œuvre d'un processus de décomposition de l'algorithme en opérations implantables. Cependant, ce processus génère souvent des répétitions de motifs d'opérations (opérations identiques mais opérant sur des données différentes). Pour réduire la taille de la spécification et mettre en évidence ses parties régulières, nous appliquons un processus de factorisation, lequel fait apparaître des sommets spéciaux (sommets frontières de factorisation), à savoir : F (partition d'un tableau en autant d'éléments que de répétitions du motif), J (composition d'un tableau à partir des résultats de chaque répétition du motif), D (diffusion d'une donnée à toutes les répétitions du motif) et I (dépendance de donnée inter-itération du motif), qui spécifient chacun l'une des différentes manières de factoriser les données et les opérations, en traversant une frontière de factorisation.

La factorisation consiste à remplacer, dans un graphe, un motif d'opérations répétitif par un seul exemplaire du motif, délimité par les sommets spéciaux "frontières" mentionnés ci-dessus. L'ensemble de sommets frontières utilisés pour remplacer le motif d'opérations répétitif appartient à une même frontière de factorisation, une abstraction qui nous permet de grouper les sommets frontières. L'intérêt de cette abstraction sera discuté dans le chapitre 3. Une opération, située du côté d'une frontière, qui présente un groupe d'opérations identiques, opérant sur un groupe factorisé de données différentes, sera réalisée au moyen d'un seul opérateur (ou de plusieurs opérateurs en parallèle) utilisé(s) itérativement autant de fois qu'il existe d'opérations dans le groupe factorisé. Ainsi, le but de la factorisation ne se restreint pas uniquement à réduire la taille de la spécification algorithmique, tout en mettant en évidence ses parties répétitives, sans en modifier la sémantique opératoire, mais elle permet également de décrire en intention plusieurs implantations plus ou moins séquentielles ou parallèles, chacune avec des caractéristiques (surface, temps de réponse) différentes.

La figure 2.1 montre le graphe algorithmique factorisé correspondant au produit scalaire (PS) entre deux vecteurs A et B, dont le résultat est exprimé par le scalaire c. Les sommets *mul* et *add* représentent respectivement les opérations de multiplication et addition. Les sommets F_1, F_2 et I délimitent une frontière de factorisation. Les dépendances de données entre les sommets sont représentées par les arcs a_i, b_i, $a_i \cdot b_i$, I_{i-1} et I_i.

Les applications embarquées temps réel interagissent avec leur environnement par une répétition infinie de la séquence acquisition-calculs-commande, qui correspond à un graphe de dépendances infiniment répétitif, dont la factorisation est équivalente à un graphe "flots de données".

L'ensemble des opérations d'un algorithme peut être spécifié sous différentes formes plus ou moins factorisées. La défactorisation, totale ou partielle, est une transformation triviale, qui permet de produire automatiquement toutes les formes plus ou moins défactorisées d'une spécification.

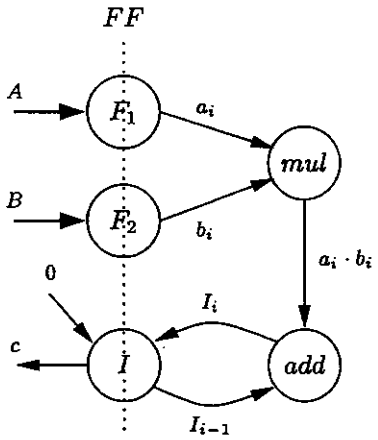


FIG. 2.1: Produit scalaire sous la forme dún GFDD

Dans ce chapitre, nous présentons l'état de l'art sur la spécification, la modélisation et la validation de systèmes embarqués temps réel. Nous introduisons les principales directives concernant la spécification de ces systèmes, les différents modèles de calcul utilisés pour représenter une application embarquée, les techniques de validation par simulation et par vérification formelle. Nous décrivons succinctement quelques langages de spécification de systèmes réactifs, telles que *Esterel* [BD91], *Lustre* [HCRP91, HLR92], *Statecharts* [Har87, Bea*98, Bea99], *Argos* [Mar89, Mar91], *Signal* [LLGL91, LG91], *G* [AK98] et *ECL* [LS99]. Nous présentons comment ces langages peuvent être utilisés pour décrire, sous la forme d'un programme, une spécification algorithmique sous la forme d'un GFDD. Nous y détaillons également notre modèle basé sur des graphes factorisés de dépendances de données (GFDD), les sommets utilisés pour les représenter, la décomposition et la factorisation d'un graphe de dépendances, les frontières de factorisation et les relations entre ces frontières. Nous présentons en dernier deux exemples de spécification algorithmique à partir d'un produit matrice-vecteur (PMV) et d'un filtrage des lignes d'une matrice.

2.1.1 Spécification

Pour spécifier la fonctionnalité d'un système, il faut décomposer cette fonctionnalité par l'intermédiaire d'un modèle du système. Un langage de spécification est utilisé pour décrire ce modèle, lequel est validé par simulation ou vérification formelle. Pour assurer l'indépendance entre la description comportementale et la réalisation matérielle de l'application, la spécification résultante ne doit contenir aucune information concernant l'implantation du système [GV95].

Le choix d'un modèle ou d'un langage approprié pour la spécification de systèmes mixtes est essentiel si l'on s'intéresse à la conception de systèmes embarqués. On peut s'orienter vers des approches hétérogènes qui mélangent plusieurs modèles avec des sémantiques différentes (p.ex., *Ptolemy* [Pto99, Pto00]) ou vers des

approches unifiées, qui décrivent les différentes parties en utilisant un seul modèle (p. ex., réseaux de Petri [Ess96], machines à états concurrentes [Pol98]).

En fonction des exigences de conception de systèmes mixtes embarqués, il faut choisir entre des modèles de spécification orientés langages, orientés état, réseaux de Petri ou hétérogènes. Le choix sera guidé par de caractéristiques, telles que l'abstraction, la concurrence, la hiérarchie, le traitement d'exceptions, les contraintes temporelles et la capacité d'implanter différents mécanismes de communication [Cod99].

La spécification peut se faire par l'intermédiaire des langages procéduraux pour la description ou la spécification des systèmes mixtes (VDHL associé à un traducteur C, C^x [GVNG94], *HardwareC* [KD90]) ; des langages non-procéduraux (flot de données synchrone) ; des langages synchrones ; ou des outils comme *Ptolemy* [KL93], pour la spécification de systèmes. Les langages synchrones constituent une approche rigoureuse de conception de systèmes réactifs. Ils permettent la spécification, la validation et la mise en œuvre d'applications sur des machines mono ou multi-processeurs et sur des circuits spécifiques (ASIC et/ou composnts programmables) [Bel94].

L'utilisation d'un langage de spécification formelle permet de créer une description précise qui peut être simulée. Cela aide le concepteur à détecter et à corriger des éventuelles erreurs fonctionnelles dès les premières étapes de conception [GV95]. Dans le cadre de ce travail, une application peut être spécifiée directement sous la forme d'un graphe flot de données, par l'intermédiaire de l'interface graphique de *SynDEx* ou par l'intermédiaire d'un des langages synchrones en utilisant le format commun DC [Hal93].

2.1.2 Modélisation

La modélisation est l'acte de représenter formellement un système ou un sous-système. Un modèle est une formalisation du processus de décomposition de la fonctionnalité d'un système et du rapport entre les composants qui l'intègrent. Un modèle peut être *mathématique* (un ensemble de déclarations sur les propriétés du système, comme sa fonctionnalité ou ses dimensions physiques), ou *constructif* (définition d'une procédure informatique qui imite un ensemble de propriétés du système). Les modèles constructifs sont fréquemment utilisés pour décrire le comportement d'un système en réponse à un stimulus externe. Ces modèles sont appelés *modèles exécutables.* Les modèles exécutables sont aussi appelés *simulations* et sont construits à partir d'un modèle de calcul [Pto99].

Un modèle formel devrait consister des composants suivants, selon Edwards et al. [ELLS97] :

- une spécification fonctionnelle, décrite sous la forme d'un ensemble de relations implicites ou explicites, contenant des informations sur les entrées, les sorties et les états internes de l'application ;
- un ensemble de propriétés à être respectées, décrites sous la forme d'un ensemble de relations entre les entrées, les sorties et les états internes de l'application ;
- un ensemble d'indices de performance, décrits sous la forme d'un ensemble d'équations, qui évalueront la qualité de la conception en termes de coût, de fiabilité, de vitesse, de taille, de consommation, etc. ;
- un ensemble de contraintes des indices de performances, décrites sous la forme d'un ensemble d'inégalités.

Un modèle de calcul est un ensemble de "lois de physique" qui gouvernent l'interaction des composants dans le modèle. Il y a plusieurs modèles capables de décrire la fonctionnalité d'un système embarqué [GV95, ELLS97] :

- DE (*Discret Event* – événement discret) : ce modèle est basé sur le temps. À chaque événement est associée une étiquette temporelle totalement ordonnée, qui indique l'instant du déclenchement de l'événement. Le matériel numérique est normalement simulé par l'intermédiaire de cette approche. La modélisation par DE peut coûter cher à cause de la phase d'ordonnancement des timbres temporels, qui peut être longue. DE est approprié à la modélisation de systèmes distribués, mais la modélisation de la concurrence reste difficile. Le modèle DE peut être spécifié à partir des langages Verilog [TM91] et VHDL. L'environnement *Ptolemy* intègre un simulateur DE ;
- FSM (*Finite State Machine* – machine à états finie) : ce modèle consiste d'un ensemble de symboles d'entrée (le produit cartésian des ensembles de valeurs des signaux d'entrée), d'un ensemble de signaux de sortie (le produit cartésian des ensembles de valeurs de sortie), d'un ensemble fini d'états avec un état initial, d'une fonction de sortie qui met en correspondance les entrées et les états internes avec les sorties, et d'une fonction *prochain-état* qui met en correspondance les entrées et les états internes avec le prochain état. Les FSM représentent graphiquement un système, sous la forme d'un ensemble d'états et d'arcs, qui indiquent la transition du système d'un état à l'autre, en fonction des événements externes. Les FSM sont appropriées pour modéliser des comportements séquentiels, mais elles ne sont pas indiquées pour modéliser les comportements concurrents ou les mémoires, dû au problème de l'explosion d'états [ELLS97]. L'environnement *Ptolemy* utilise des FSM hiérarchiques, mélangées avec les modèles flot de données, DE ou synchrone/réactif ;
- SR (synchrone/réactif) : ce modèle est basé sur la synchronisation de tous les événements par une horloge globale. Ces événements simultanés peuvent être totalement, partiellement ou non ordonnés. Le modèle SR est celui utilisé par les langages synchrones, telles que *Esterel*, *Lustre*, *Signal* et *Argos*. Ces langages décrivent les systèmes sous la forme d'un ensemble de modules synchronisés, d'exécution concurrente ;

- GFD (graphe flot de données) : ce modèle utilise des graphes orientés, où les sommets correspondent aux calculs et les arcs correspondent à des séquences d'événements totalement ordonnés, pour spécifier une application. Le modèle GFD décompose la fonctionnalité en activités qui transforment les données et en flots de données entre ces activités. Les GFD sont fréquemment représentés de façon graphique et ils sont plutôt hiérarchiques. Cela veut dire qu'un sommet du GFD peut représenter un autre graphe orienté. *Khoros* [LJ95] est un exemple d'environnement graphique de programmation flot de données, orienté au traitement du signal et des images (TSI). L'environnement graphique *Ptolemy* peut également être utilisé comme un outil de programmation flot de données orienté, entre autres, au TSI.
- CSP (*Communicating Sequential Process* – processus séquentiels communicants) : ce modèle décompose le système en un ensemble de processus concurrents, chacun exécutant une séquence d'instructions ;
- PSM (*Program State Machine* – machine à état de programme) : ce modèle mélange les modèles FSM et CSP, autorisant chaque état d'une FSM hiérarchique et concurrente à contenir des actions décrites par des instructions.

D'autres modèles sont les réseaux de Petri, les diagrammes de Jackson, les diagrammes de rapport d'entités, les CDFG (graphes flots de données et de contrôle), les modèles orientés-objets et les modèles de files.

L'ensemble de règles qui gouvernent l'interaction des composants est appelé *sémantique du modèle de calcul.* Aucun modèle n'est idéal pour tous les types de systèmes. Le choix du modèle de calcul dépend fortement du type de système à implanter. Ce choix est guidé par des caractéristiques, telles que la facilité de modélisation, l'efficacité d'analyse (simulation ou vérification formelle), la synthétisabilité automatique et l'optimisabilité. Pour un système purement informatique, qui transforme un ensemble fini de données dans un autre ensemble fini de données, la sémantique impérative, comme celle des langages de programmation C, C++, *Java* [Jav00] et *Matlab* [Mat00], semble la plus apropriée. Par contre, pour modéliser un système mécanique, par exemple la sémantique doit être capable de manipuler la concurrence et le temps continu, comme celle trouvée en *Simulink* [Sim00], *Saber* [Sab98], *HP Advanced Design System* [Hpa99] et VHDL-AMS [Vhd99].

Le modèle GFD est approprié à un système qui répète périodiquement les mêmes transformations/opérations sur des paquets de données, comme un système de traitement du signal et des images. Le modèle FSM est approprié à un système qui n'effectue pas de calculs complexes, mais qui doit répondre à des séquences complexes d'événements externes, comme un système de contrôle-commande. Le modèle CSP est approprié à des systèmes qui effectuent des complexes transformations de données, plusieurs fois en parallèle, comme les bases de données client-serveur ou les systèmes multi-tâches. Le modèle PSM est approprié aux systèmes qui sont à la fois orientés-données et orientés-contrôle, comme un système embarqué distribué [GV95].

La modélisation doit permettre d'analyser ou de vérifier les propriétés logiques du système et d'évaluer ses performances en utilisant des modèles prédictifs, des estimations probabilistes, de la rétro-annotation, etc. [Bel94]. Il existe deux types de modèles [GD92] :

- Modèles basés sur les processus : la modélisation des propriétés logiques et temporelles des processus, ainsi que de leurs interactions, définissent le comportement d'un système ;
- Modèles basés sur les graphes : la théorie des graphes est utilisée pour la modélisation du comportement du système. Les dépendances entre les processus et les opérations qui les constituent sont explicites.

Après avoir choisi le modèle approprié, il faut spécifier la fonctionnalité du système par l'intermédiaire d'un langage de spécification. VHDL et *Verilog* [TM91] sont des standards très utilisés, permettant de décrire les modèles CSP et FSM. *Esterel* est également utilisé pour décrire les modèles CSP et FSM. *StateCharts* permet la description de FSM hiérarchiques et concurrentes. *SpecCharts* décrit les modèles CSP, FSM hiérarchiques et concurrentes, et PSM. SDL [Da*96] permet de décrire des GFD hiérarchiques contenant des FSM. *Silage* [Hil85] décrit des GFD.

De toute façon, le meilleur modèle est celui qui traduit le mieux les caractéristiques du système qu'il modélise. Comme on s'intéresse au traitement bas niveau du signal et des images, dont les algorithmes des respectives applications sont caractérisés par des répétitions périodiques d'un ensemble d'opérations, nous les modélisons par l'intermédiaire d'un modèle basé sur les GFD : les graphes factorisés de dépendances de données (GFDD).

2.1.3 Langages de spécification

Un langage est un ensemble de symboles, de règles pour combiner ces symboles (sa syntaxe) et de règles pour interpréter les combinaisons des symboles (sa sémantique). La plupart des langages sont basés sur un paradigme unique, c'est-à-dire ils sont orientés à un domaine spécifique d'application. Quelques langages sont plus adaptés aux spécifications orientées à l'état (*SDL* et *Statecharts*). D'autres sont plus appropriés aux calculs continu et flot de données (*Lustre* et *Matlab*). Il y a encore les langages orientés à la description algorithmique (C et C++).

Les langages classiques de spécification (C et C++) sont difficilement utilisables pour spécifier des applications temps réel car, d'une part, ils sont mal adaptés à l'exploitation du parallélisme potentiel de l'application et, d'autre part, leur sémantique ne possède pas la notion de temps [ALSV96]. D'autres langages, comme les langages synchrones, permettent l'expression du parallélisme présent dans un algorithme et la modélisation du temps. L'hypothèse synchrone signifie qu'on ne prend pas en considération les caractéristiques matérielles des opérateurs : les durées des opérations sont négligées, donc les propriétés d'un programme ne sont ni liées aux vitesses d'exécution, ni aux débits de communication.

Comme nous avons affirmé dans la section 2.1.1, les langages synchrones qui utilisent le format commun DC peuvent être utilisés par la méthodologie AAA pour spécifier et vérifier des algorithmes d'application hors contraintes matérielles [ALSV96]. Le modèle flot de données de quelques uns de ces langages (*Lustre*, *Signal*) permet de décrire facilement le parallélisme potentiel de l'application. Le choix de la grosseur des grains (granularité des opérations) est réalisé par le concepteur. Nous décrivons ci-dessous, de façon succincte, quelques langages de spécification de systèmes réactifs.

Esterel

Esterel [BD91] est un langage impératif textuel et synchrone orienté-contrôle. Il est approprié à la description de systèmes réactifs et, particulièrement, de contrôleurs, lesquels exigent des réactions continues avec l'environnement. Le modèle de calcul utilisé par *Esterel* est le EFSM (*Extended Finite State Machine*), une extension du modèle FSM, qui permet l'implantation matérielle et/ou logicielle de ses descriptions. *Esterel* dispose de constructions structurées de flot de contrôle analogues à celles des programmes séquentiels traditionnels.

Le compilateur *Esterel v.5* peut générer des implantations matérielles ou logicielles à partir d'un programme réactif. Il peut générer du code C à être incorporé à un programme qui synthétise l'interface et les manipulations de données. Il peut aussi générer du matériel, sous la forme d'une *netlist* de portes logiques, à être incorporé à un circuit.

Lustre

Lustre [HCRP91] est un langage déclaratif textuel, flot de données et synchrone pour la description des algorithmes de type traitement du signal. La nature synchrone de *Lustre* assure que toutes les variables et les expressions d'un programme prennent simultanément la n-ème valeur de leurs respectives séquences de valeurs. Un programme *Lustre*, exprimé sous la forme d'un ensemble d'équations, a un comportement acyclique, où un cycle d'exécution calcule la n-ème valeur de chaque variable ou expression [HLR92]. Un programme *Lustre* spécifie une relation entre les variables d'entrée et les variables de sortie. Toutes les variables ou expressions sont exprimées en fonction du temps, où le temps correspond à un ensemble de nombres naturels et les variables sont définies par des équations. En plus de ses caractéristiques flot de données, *Lustre* offre également un mécanisme de traitement des horloges multiples.

Le système LESAR est utilisé pour la vérification de programmes décrits en *Lustre* [Rat92, Ber92]. Bryant [Bry86] a réalisé un vérificateur de programmes *Lustre* à partir des diagrammes de décision binaires (BDD). Berkone [Ber92] a testé l'adéquation de ce système à la vérification de circuits.

Statecharts

Statecharts [Har87, Bea*98] est un langage impérative graphique et synchrone basé sur des FSM hiérarchiques. Il implante une version modifiée des diagrammes d'*état-transition*, qui inclut la hiérarchie, le parallélisme, les connecteurs de jonction et d'historique. Comme *Statecharts* est basé sur les états, afin d'éviter l'explosion du nombre d'états et de transitions certains mécanismes ont été ajouté au formalisme de base. *Statecharts* consiste donc essentiellement en un ensemble d'états et de transitions. Ainsi, pour modéliser la hiérarchie, un état peut être décomposé et contenir des sous-états et des transitions internes. Aux feuilles de la hiérarchie, les états ne sont pas décomposables. L'introduction de la hiérarchie dans les FSM permet de classer les transitions et peut résoudre certains non-déterminismes [Bea99]. *Statecharts* est utilisé pour la conception de systèmes temps réel, ainsi les instants où les différentes actions sont effectués doivent être précisément définis. L'unité de temps d'un programme *Statecharts* s'appelle *step* et correspond à l'intervalle de temps dans lequel sont regroupées plusieurs actions.

Argos

Argos [Mar89, Mar91] est un langage impérative synchrone, textuel et graphique, dérivé du formalisme *Higraphs*, proposé par Harel [Har87] et utilisé dans le langage *Statecharts*. La définition formelle d'*Argos* est basée sur l'algèbre de processus, dont les termes de base décrivent les automates et dont les opérateurs représentent un système par l'intermédiaire de la composition hiérarchique ou parallèle de ses différents éléments. *Argos* est donc utilisé pour la spécification de systèmes réactifs, décrits graphiquement sous la forme d'automates (FSM hiérarchiques), représentant les différents sous-systèmes qui composent l'application. Il est intégré à l'environnement de vérification *Argonaute* [Mar89], lequel permet, de façon graphique et interactive, l'édition, la visualisation et la simulation des processus décrits en *Argos*.

Signal

Signal [LLGL91, LG91] est un langage flot de données rélationnel (équationnel), textuel et synchrone, construit autour d'un noyau minimum de constructeurs de base. Le langage est muni d'une interface graphique de type *diagrammes de blocs*. Un programme *Signal* décrit des relations entre signaux. Un signal est une suite non bornée de valeurs typées. On associe à chaque signal son horloge, lequel détermine les instants où les valeurs sont disponibles. Il est possible d'exprimer des contraintes de synchronisation entre les différents signaux. Le compilateur se charge de déterminer si les contraintes ont été respectées ou non. *Signal* utilise l'hypothèse du synchronisme fort : toute action (calcul ou communication) est instantanée. On suppose que les sorties produites par une dépendance opératoire sont simultanées aux entrées. Deux signaux sont synchrones si leurs événements sont simultanés.

Le compilateur *Signal* effectue une vérification temporelle du programme, indépendamment de l'architecture sur laquelle il sera implanté. La cohérence de l'ordre logique des événements des signaux est vérifiée grâce aux horloges des signaux. À chaque programme est associé un système d'équations d'horloge, qui doit admettre une solution unique pour être correct [ALSV96]. Si certains signaux ont une horloge indéterminée, il faut les synchroniser avec des signaux d'horloge connus. Dans une approche pour la conception conjointe de systèmes mixtes matériel/logiciel, nous utilisons *Signal* comme langage de spécification et VHDL comme moyen d'accès aux outils de CAO existants. Sans une méthodologie de conception descendante, où le système est décrit en *Signal*, l'implantation peut être mise en œuvre en logiciel (sur mono ou multi-processeurs), en matériel (sur un un plusieurs circuits) ou sur une cible mixte matériel/logiciel [Bel94].

G

G [AK98] est un langage flot de données dynamique, multidimensionnel et homogène, qui permet de représenter un logiciel embarqué sous la forme d'un flot de données. Le compilateur *G* est intégré à l'environnement graphique de développement d'applications *LabVIEW* [Lab00] (*Laboratory Virtual Instrument Engineering Workbench*), développé par *National Instruments Corporation*. Cet environnement s'oriente aux domaines de l'acquisition de données, des testes et mesures, et de l'automatisation industrielle.

ECL

ECL (*Esterel/C Language*) [LS99], comme suggère son nom, est un langage de spécification de systèmes hétérogènes, qui combine les langages *Esterel* et *C*. *ECL* est orienté aux processus. Ces derniers sont les entités communicantes qui décrivent la fonctionnalité du système. Ce langage cible les processus à contrôle intensif. *ECL* mélange les spécifications de données et de contrôle. La portion contrôle, équivalente à une EFSM, utilise une sémantique totalement synchrone, ce qui nous autorise à se servir des techniques d'optimisation, d'analyse et de synthèse développées pour les FSM. La portion données utilise une sémantique comme celle du langage C. Le concepteur doit effectuer le partitionnement des modules synchrones qui se communiquent de façon asynchrone.

2.1.4 Validation

La conception de systèmes et de circuits embarqués temps réel exige l'utilisation de plusieurs outils de conception (compilateurs, outils de synthèse) et de validation. Les outils de validation cherchent à garantir que l'implantation soit correcte. Cette validation, qui doit être effectuée, de préférence, aux niveaux plus élévés d'abstraction, peut se faire par :

- **Simulation** : cette méthode consiste à exécuter la spécification et à comparer la séquence de données produite avec la séquence de données attendue. C'est une méthode très répandue, mais il faut qu'elle soit exhaustive pour assurer que l'implantation résultante sera correcte ;
- **Vérification formelle** : cette méthode cherche à prouver, indépendamment des stimulus d'entrée, la conformité de la spécification et de la description (implantation) résultante, étant donnée une certaine relation (implication, équivalence, etc.). Elle vérifie mathématiquement si le comportement du système, décrit par l'intermédiaire d'un modèle formel, satisfait une certaine propriété, aussi décrite par un modèle formel.

La simulation de systèmes embarqués est normalement complexe à cause de leur hétérogenéité. La plupart de ces systèmes contient des composants matériels et logiciels, qui doivent être simulés simultanément. On est face à un problème de co-simulation, qui doit concilier deux exigences conflictuelles : d'abord, assurer l'exécution rapide du logiciel, même sur une architecture très différente de la CPU-cible et, ensuite, conserver la synchronisation entre le matériel et le logiciel [ELLS97].

La vérification formelle se présente comme la méthode la plus raisonnable pour la validation. Pourtant, elle présente quelques problèmes : l'inexistance d'une méthode générale qui soit applicable à tous les niveaux d'abstraction et aux différents types de circuits-cibles ; la restriction de la complexité des circuits-cibles qui peuvent être formellement vérifiés ; et la non-intégration entre les outils de vérification et les outils de conception d'architectures.

Les principales techniques utilisées pour la vérification formelle sont les suivantes [CP88, Gup91, CBE*92, ELLS97] :

- **Model checking** : la spécification est un ensemble de propriétés exprimées avec des logiques (temporelles, du premier ordre, etc.) et la description est une implantation du circuit. Cette technique cherche à établir si l'implantation satisfait les propriétés décrites par la spécification [BCDM86, BCM*90, McM92] ;
- **Preuve par comparaison** : cette technique vise à établir si la spécification et l'implantation ont un comportement équivalent. Dans ce cas, la spécification et l'implantation sont des descriptions du comportement du circuit [CBM89, Cou91, Ber92] ;
- **Theorem provers** : ces démonstrateurs de théorèmes, manuels ou automatiques, ont été appliqués avec succès à la vérification de circuits combinatoires et séquentiels. Ils offrent la possibilité d'effectuer des preuves hiérarchiques [Gup91] ou des preuves paramétrées utiles pour la preuve des circuits de chemin de données [Pie93] ;
- **Ordre partiel** : cette technique est basée sur la notion d'ordonnancement partiel entre les calculs au moment de l'exécution d'un réseau de processus. Elle est très utile pour la vérification de systèmes hétérogènes avec de multiples agents concurrents [God90].

La technique *model checking* n'est pas appropriée à la vérification d'algorithmes récursifs. Elle est plutôt orientée aux applications de contrôle intensif, décrites en termes de FSM, telles que les protocoles de cohérence de cache, les contrôleurs de bus, les commutateurs téléphoniques, les circuits d'arbitrage et les protocoles de communication [Kur97]. Les outils de vérification formelle devront évoluer de façon à utiliser des modèles comportementaux plus abstraits et à élargir le nombre d'applications qui puissent être vérifiées automatiquement [Kur97].

Les principaux outils commerciaux de vérification formelle sont [Kur97] :

- *CheckOff* [Tuc95, Goe97a, Goe97b], développé par *Siemens* et commercialisé par *Abstract Hardware Ltd.*. *CheckOff* utilise les diagrammes de décision binaires (BDD – *Binary Decision Diagrams*) et il peut vérifier les circuits synchrones et asynchrones. Il existe trois versions de *CheckOff* : *CheckOff-E*, *CheckOff-S* et *CheckOff-M*. *CheckOff-E* est un outil de vérification d'équivalence. Il effectue une vérification exhautive et rapide de la fonctionnalité de deux *netlists* (RTL–RTL, RTL–portes, portes–portes). Il effectue également la vérification d'équivalence fonctionnelle entre une description VHDL et une description *Verilog*. *CheckOff-S* est un outil de vérification d'équivalence basé sur des équations. *CheckOff-M* est un outil de vérification de modèle (*model checking*) ;
- *Designer Verifyer* [Chr99], de *Chrysalis*, est un outil de vérification d'équivalence qui inclut une extension de logique symbolique, afin de permettre l'utilisation du sous-ensemble synthétisable de *Verilog* ;
- *VFormal* [CB96], développé par *Bull* et commercialisé par *Compass*. Il est un outil de vérification d'équivalence et de vérification de modèle qui admet les multiples horloges et le partitionnement automatique. *VFormal* accepte les descriptions RTL en *Verilog* et en VHDL. Il utilise une représentation plus compacte des BDD, nommée graphe de décision typé (TDG – *Typed Decision Graph*), qui profite de la rédundance présente dans les BDD ;
- *RuleBase*, développé par CMU et commercialisé par *IBM*. Il est outil de vérification de modèle qui s'interface à plusieurs environnements de conception, tels que les outils de synthèse *Synopsis* et *Compass* et les langages VHDL et *Verilog* ;
- *FormalCheck* [Tuc99], de *AT&T Design Automation*, est un outil de vérification d'équivalence et de vérification de modèle. Il utilise l'analyse automatique ou la vérification de modèle symbolique pour examiner si les propriétés du système spécifié on été respectées.

Dans notre cas, le concepteur peut se servir des mécanismes de vérification formelle associés aux langages synchrones [LLGL91, Sor92, BLLMS94], pour effectuer la vérification temporelle de la spécification. Cette vérification est basée sur l'ordre partiel des événements. *Signal* est donc son propre système de preuve. Les propriétés de l'application peuvent être exprimées sous la forme de programmes *Signal* et traitées par le compilateur comme des équations additionnelles. Le mécanisme de preuve consiste à vérifier s'il existe de contradictions dans le programme résultant.

Une application embarquée temps réel peut être décrite sous la forme d'un ensemble d'expressions sur les signaux. Chacune de ces expressions définit un signal, son horloge ou les contraintes temporelles. Les langages synchrones (*Esterel*, *Lustre* ou *Signal*) génèrent un format commun DC sous la forme d'un fichier *.sdx*, qui est fourni à *SynDEx* pour décrire l'algorithme sous la forme d'un graphe flot de données conditionné (*SynDEx* multi-composant) ou sous la forme d'un graphe factorisé de dépendances de données (pour l'extension aux circuits reconfigurables).

2.2 Graphes de dépendances

Au sens mathématique, une opération, dans un contexte numérique (opposé à analogique), combine des données fournies en entrée afin de fournir des résultats en sortie. Chaque opération est caractérisée par une liste des types (codage numérique de ses données et résultats), et par une combinatoire qui calcule les valeurs des sorties à partir de celles des entrées. Ce sont des dépendances "opératoires" intra-opération. La composition d'opérations consiste à fournir, à une entrée d'une opération, la valeur d'une sortie d'une autre opération : la valeur de l'entrée dépend de celle de la sortie. Ce sont des dépendances "de données" inter-opérations. À travers une diffusion, la valeur d'une sortie peut être donnée à plusieurs entrées. La composition d'opérations doit être déterministe, c'est-à-dire elle ne doit pas produire de cycle de dépendances [LS97].

La décomposition d'un algorithme en opérations implantables consiste à construire, par composition d'opérations, un graphe de dépendances de données (hyperarcs) entre opérations (sommets), ou dualement un graphe de dépendances opératoires (hyperarcs) entre valeurs (sommets), comme le montre la figure 2.2 pour la décomposition d'un algorithme en quatre opérations E, F, G, H (l'hyperarc H a pour origine c, d et pour extrémités e, f), calculant les valeurs e, f à partir de la valeur a, par l'intermédiaire des valeurs b, c, d. La forme représentée par la figure 2.2(a) est préférable pour un éditeur graphique interactif de graphes, car il est plus commode de représenter chaque opération par une boîte avec des ports d'entrée et de sortie à ses bords.

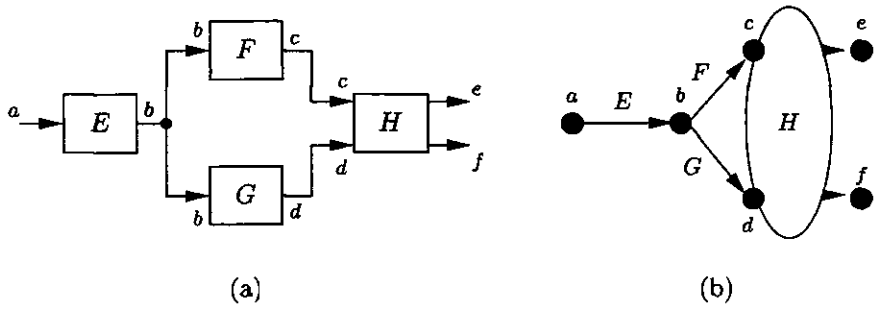


FIG. 2.2: Graphes duaux de dépendances

2.2.1 Représentation et notation

La factorisation des opérations amène à grouper, de manière ordonnée, des données du même type. Les types représentent le codage binaire des données. Nous proposons la notation ci-dessous pour décrire les types de groupes ordonnés de données en fonction du type des données du groupe. Cette définition est récursive, chaque donnée du groupe pouvant être elle-même un groupe de données de type plus primitif. Donc, de manière générale, une donnée de type T est représentée par un vecteur de d_1 éléments de type T'. Chaque élément de type T', à son tour, peut être soit un vecteur de d_2 éléments de type T'', soit une donnée d'un type de base (*logical*, *integer*, *real* ou *dpreal*). Une donnée de type T sera notée de la façon suivante :

$$T \equiv [1..d_1]T' \equiv [1..d_1][1..d_2]T'' \ldots \tag{2.1}$$

Tout au long de ce travail, les opérations de base et/ou les sommets frontières de factorisation d'un GFDD sont représentés par des cercles. Les dépendances de données entre les opérations de base et/ou les sommets frontières de factorisation sont représentées par des arcs entre deux sommets. Une frontière de factorisation est mise en évidence par des pointillés, et délimitée par des sommets spéciaux dits "sommets frontières". Ces sommets entourent le motif répétitif pour le séparer du reste du graphe.

2.2.2 Décomposition d'un graphe de dépendances : description des sommets de base

Les sommets de base décrivent les opérations combinatoires de l'algorithme. Nous y distinguons cinq types : *OPÉRANDE*, *RESULTAT*, *CALCUL*, *IMPLODE* et *EXPLODE*.

Sommet *OPÉRANDE*

Le sommet *OPÉRANDE*, identifié par N, correspond à la déclaration d'une donnée d'entrée. Ce sommet n'a pas de prédécesseur et il marque une source du graphe algorithmique, comme le montre la figure 2.3.

$$N \Longrightarrow T \tag{2.2}$$

FIG. 2.3: Sommet *OPÉRANDE*

Sommet *RÉSULTAT*

Le sommet *RÉSULTAT*, identifié par R, correspond à la représentation d'un résultat dans un graphe fini. Ce sommet n'a pas de successeur et il marque un puits du graphe algorithmique, comme le montre la figure 2.4.

$$T \Longrightarrow R \tag{2.3}$$

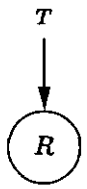


FIG. 2.4: Sommet *RÉSULTAT*

Sommet *CALCUL*

L'ensemble de sommets *CALCUL*, identifiés par *Cal*, est constitué par des opérations combinatoires logiques (and, or, not, exclusive-or, etc.), arithmétiques (addition, multiplication, soustraction, division, etc.) et des opérations de calcul plus complexes (filtres, transformée de Fourier rapide, etc.). Les valeurs des résultats (sorties) de ces opérations ne dépendent que des valeurs des arguments (entrées) ; il n'y a pas de dépendances des entrées précédentes, ni d'effet de bord sur les opérations précédentes. La figure 2.5 montre deux exemples d'opérations *CALCUL* : (a) l'addition de deux variables e_1 et e_2 et (b) le calcul de la valeur moyenne (m) des n pixels d'une image I. Dans un graphe représentant un algorithme, qui possède différents sommets *CALCUL*, le mnémonique *Cal* est remplacé par le mnémonique de l'opération correspondante. Le tableau 2.1 montre les mnémoniques particuliers que nous proposons pour quelques opérations *CALCUL*, aussi bien que les équations décrivant chaque opération.

$$S = Cal(E) \tag{2.4}$$

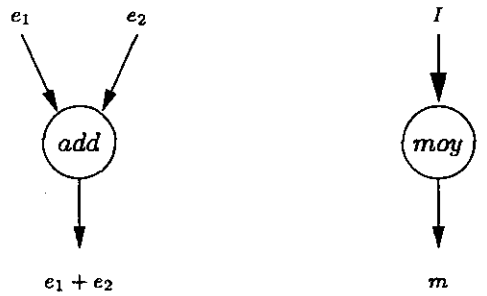


FIG. 2.5: Exemples de sommets *CALCUL*

TAB. 2.1: Mnémoniques pour des sommets CALCUL

OPÉRATION	*MNÉMONIQUE*	*ÉQUATION*
addition	*add*	$s = e_1 + e_2$
multiplication	*mul*	$s = e_1 \times e_2$
soustraction	*sub*	$s = e_1 - e_2$
division	*div*	$s = e_1/e_2$
and logique	*and*	$s = e_1 \wedge e_2$
or logique	*or*	$s = e_1 \vee e_2$
not logique	*not*	$s = \bar{e}$
exclusive-or logique	*xor*	$s = e_1 \oplus e_2$
filtres	*filt*	$S = filt(E)$
transformée de Fourier	*fft*	$S = fft(E)$

Sommet *IMPLODE*

Le sommet *IMPLODE*, identifié par M, effectue le regroupement ordonné de d données d'entrée de même type, T'_1 à T'_d, en un vecteur de sortie de dimension d, $[1..d]T'$, comme le montre la figure 2.6.

$$M\left(T'_1, \ldots, T'_d\right) \Longrightarrow [1..d]T' \tag{2.5}$$

Soit les n éléments C'''_{ijk} d'un cube C, où $n = d_1 \cdot d_2 \cdot d_3$ et $d_1 = d_2 = d_3 = 3$:

$$\begin{array}{lcl} C'''_{111} \ldots C'''_{113} & \cdots & C'''_{131} \ldots C'''_{133} \\ \vdots & & \vdots \\ C'''_{311} \ldots C'''_{313} & \cdots & C'''_{331} \ldots C'''_{333} \end{array}$$

- l'application de 9 opérations *IMPLODE* les rassemble en 9 vecteurs C''_{ij} :

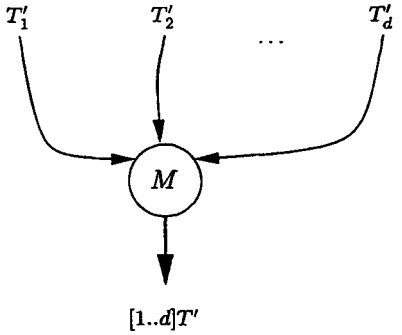


FIG. 2.6: Sommet *IMPLODE*

$$\begin{matrix} C''_{11} & \cdots & C''_{13} \\ \vdots & & \vdots \\ C''_{31} & \cdots & C''_{33} \end{matrix}$$

- l'application de 3 opérations *IMPLODE* les rassemble en 3 matrices C'_i :

$$\begin{matrix} C'_1 \\ \vdots \\ C'_3 \end{matrix}$$

- l'application d'une opération *IMPLODE* les rassemble en un cube C :

$C \equiv [1..d_1]C' \equiv [1..d_1][1..d_2]C'' \equiv [1..d_1][1..d_2][1..d_3]C'''$.

Les trois applications successives de l'opération *IMPLODE* sur les n éléments du cube C ont produit le cube C (tableau à trois dimensions : d_1, d_2, d_3), comme le montre la figure 2.7.

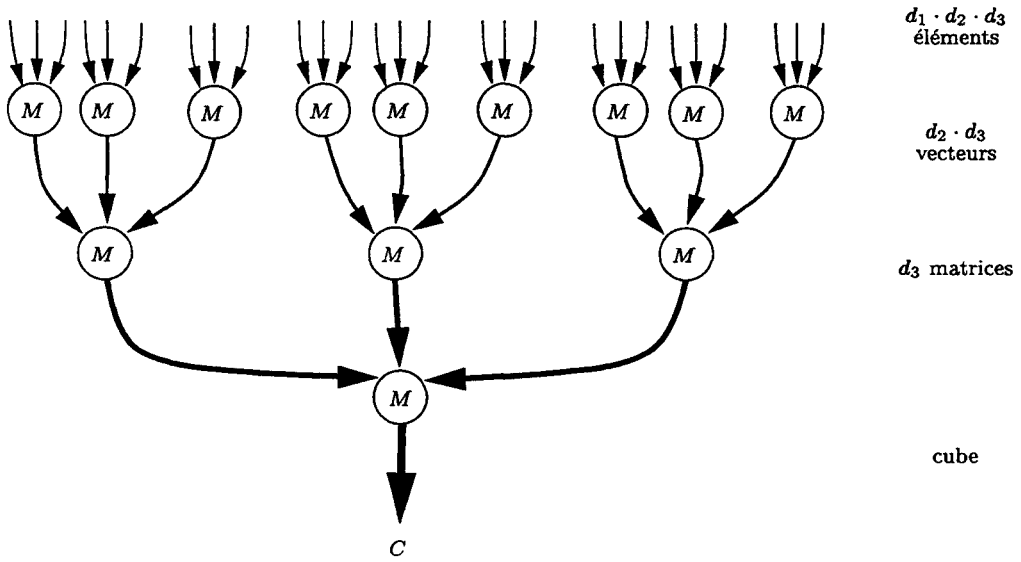


FIG. 2.7: Exemple d'utilisation d'*IMPLODE*

Sommet *EXPLODE*

Le sommet *EXPLODE*, identifié par X, effectue la décomposition d'un vecteur $[1..d]T'$ en ses éléments T'_1 à T'_d, comme le montre la figure 2.8. C'est l'opération inverse de l'*IMPLODE* (cf. 2.2.2).

$$X\Big([1..d]T'\Big) \Longrightarrow T'_1, \ldots, T'_d \tag{2.6}$$

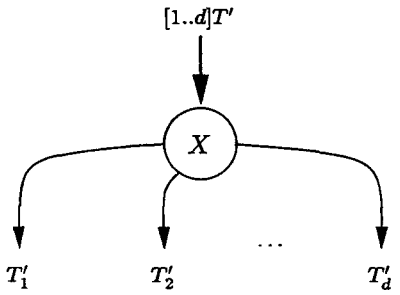


FIG. 2.8: Sommet *EXPLODE*

Soit un cube C de $d_1 \cdot d_2 \cdot d_3$ éléments :

$$C \equiv [1..d_1]C' \equiv [1..d_1][1..d_2]C'' \equiv [1..d_1][1..d_2][1..d_3]C''' \tag{2.7}$$

où C' correspond à une matrice, C'' à un vecteur et C''' à un élément.

Si $d_1 = d_2 = d_3 = 3$:

- l'opération *EXPLODE*, $X\Big(C\Big)$, le sépare en 3 matrices C'_i :

  C'_1
  C'_2
  C'_3

- l'opération *EXPLODE* appliquée sur chacune de ces matrices C'_i les sépare en 9 vecteurs C''_{ij} :

  C''_{11} C''_{12} C''_{13}
  C''_{21} C''_{22} C''_{23}
  C''_{31} C''_{32} C''_{33}

- l'opération *EXPLODE* appliquée sur chacun de ces vecteurs C''_{ij} les sépare en 27 éléments C'''_{ijk} :

  C'''_{111} C'''_{112} C'''_{113} C'''_{121} C'''_{122} C'''_{123} C'''_{131} C'''_{132} C'''_{133}
  C'''_{211} C'''_{212} C'''_{213} C'''_{221} C'''_{222} C'''_{223} C'''_{231} C'''_{232} C'''_{233}
  C'''_{311} C'''_{312} C'''_{313} C'''_{321} C'''_{322} C'''_{323} C'''_{331} C'''_{332} C'''_{333}

Les trois applications successives de l'opération *EXPLODE* sur le cube C (tableau à trois dimensions : d_1, d_2, d_3) ont produit n éléments, où $n = d_1 \cdot d_2 \cdot d_3$, comme le montre la figure 2.9.

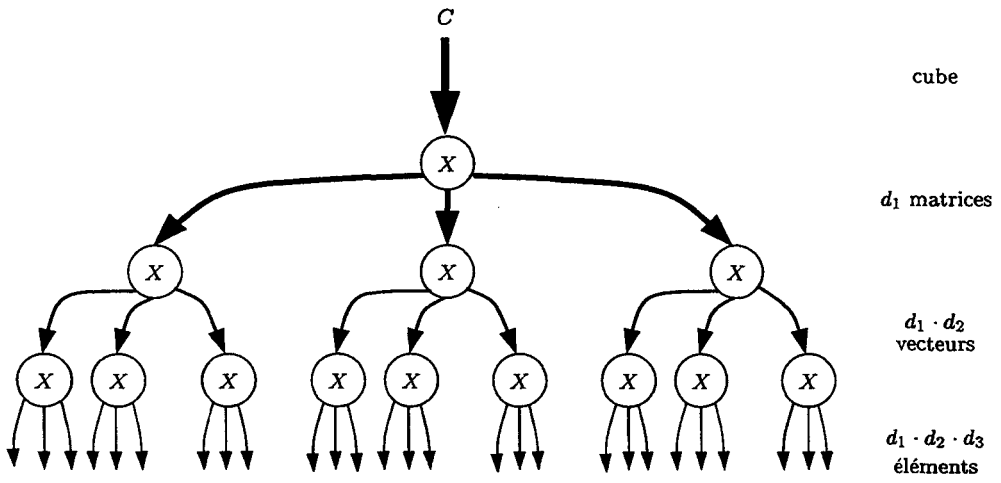


FIG. 2.9: Exemple d'utilisation d'*EXPLODE*

2.2.3 Exemple : décomposition d'un produit matrice-vecteur

Le produit $C = A \cdot B$ d'un vecteur $B \in R^n$ (aussi noté $B = \{b_1, \ldots, b_n\} = (b_j)_{1 \leq j \leq n}$) pour une matrice $A \in R^m R^n$ (composée de m vecteurs : $A = (A_i)_{1 \leq j \leq m}$ avec $b_i = (b_{ij})_{1 \leq j \leq n}$) de résultat $C \in R^m$ peut être décomposé em m produits scalaires $C = (A_i \cdot B)_{1 \leq j \leq m}$. Ces derniers peuvent être décomposés chacun en une somme de n produits $A_i \cdot B = \sum_{1 \leq j \leq n}(a_{ij} \cdot b_j)$.

La figure 2.10 représente le graphe de dépendances d'un produit matrice-vecteur (à gauche) et sa décomposition en produits scalaires (à droite, pour $m = n = 3$) [LS97]. La figure 2.11 présente le graphe de dépendances du produit scalaire et sa décomposition en somme de produits.

2.3 Factorisation d'un graphe de dépendances

La décomposition d'un algorithme en opérations implantables génère souvent des répétitions périodiques de motifs d'opérations (des opérations identiques qui opèrent sur des données différentes), qu'un utilisateur, se lassant des énumérations, va préférer spécifier sous forme factorisée. Ainsi, dans le cas du PMV, le motif répétitif (donc sa factorisation) est le PS, et dans le PS le motif est constitué du couple d'opérations (mul, add). La factorisation d'un graphe de dépendances ne change en rien sa sémantique opératoire, réduisant tout simplement la taille de la spécification et mettant en évidence ses parties régulières, appélées motifs répétitifs [LS97]. L'intérêt de la factorisation ne se restreint pas seulement à réduire la taille des spécifications :

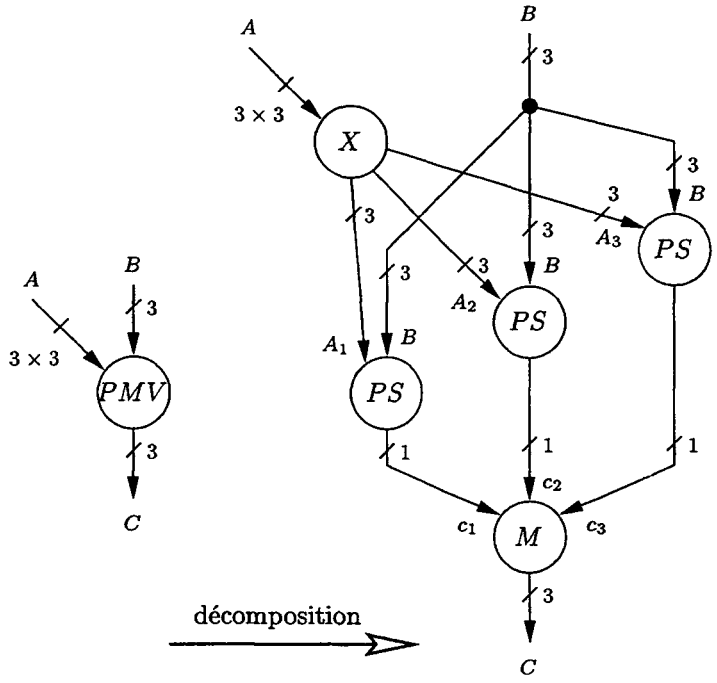


FIG. 2.10: Décomposition d'un produit matrice-vecteur

elle permet aussi de réduire la taille des architectures. Une opération située du côté "rapide" d'une frontière de factorisation, qui représente un groupe d'opérations identiques opérant sur un groupe factorisé de données différentes, se réalise avec un seul opérateur utilisé itérativement, autant de fois qu'il existe d'opérations dans le groupe factorisé.

2.3.1 Factorisation de motifs de graphes répétitifs finis : description des sommets de factorisation

La spécification d'un algorithme en termes des sommets de base précédents peut générer des répétitions périodiques de motifs d'opérations. Nous traitons ici de motifs d'opérations identiques, opérant sur des données différentes.

La factorisation ou la spécification de ce même algorithme en termes de sommets de factorisation de motifs de graphes répétitifs permet d'obtenir une spécification plus compacte. Les motifs répétitifs sont remplacés par un seul motif équivalent et les données d'entrée et de sortie sont énumérées/collectées de façon à produire les mêmes résultats que la spécification non-factorisée. Le sous-graphe du motif de la répétition est délimité, séparé du reste du graphe, par des sommets frontières.

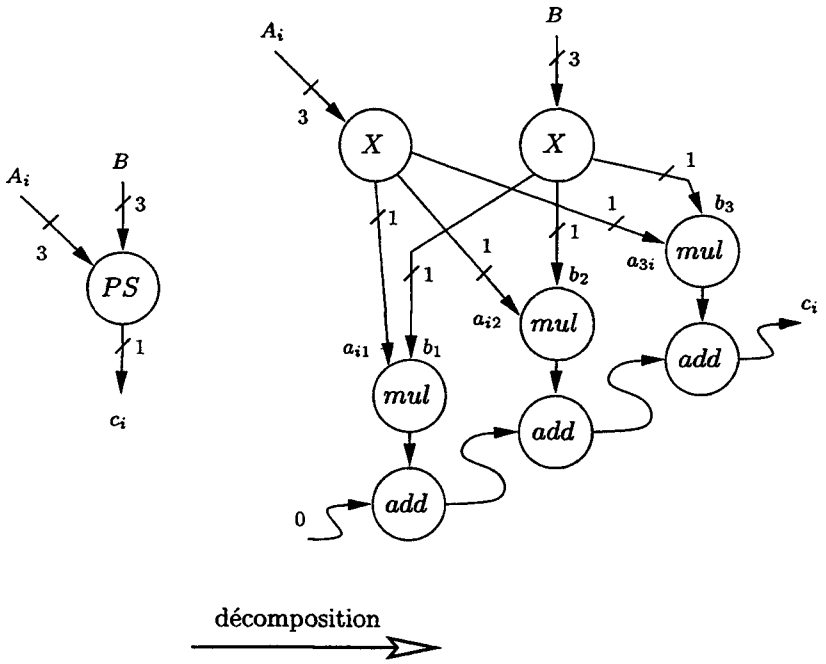


FIG. 2.11: Décomposition d'un produit scalaire

Dans un GFDD, les sommets frontières sont les délimiteurs d'une "syntaxe graphique", de façon similaire aux parenthèses qui délimitent la portée d'un foncteur dans la syntaxe algébrique. Chacun de ces sommets spécifie une forme différente de factorisation des arcs coupés par la frontière du motif [LS97].

Sommet *FORK*

Le sommet *FORK*, identifié par F, effectue la factorisation d'un flot de données sous la forme d'un vecteur $[1..d]T'$ en son entrée, en énumérant des éléments T'_i, où $i \in \{1, \ldots, d\}$, en sortie, comme le montre la figure 2.12. Un changement de la valeur des données en entrée du *FORK* correspond à d changements de la valeur des données en sortie. Le sommet *FORK* permet de traverser une frontière de factorisation, tout en augmentant la cadence des données : en aval du *FORK* la cadence des données est d fois supérieure à la cadence des données en amont de *FORK*. Ainsi, une frontière de factorisation FF définie par un sommet *FORK* sépare le côté "lent" (cadence moins élévée) du côté "rapide" (cadence plus élévée) de ce sommet. Les d sous-graphes (identiques, motif répétitif) en aval de l'*EXPLODE* (cf. Section 2.2.2) sont également factorisés en un seul sous-graphe en aval du *FORK*. C'est le correspondant factorisé de l'*EXPLODE*.

$$F\Big([1..d]T'\Big) \Longrightarrow T'_i \tag{2.8}$$

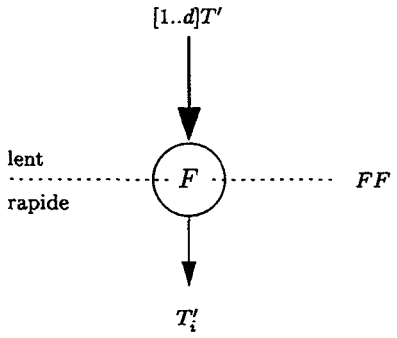


FIG. 2.12: Sommet *FORK*

Correspondance entre *FORK* et *EXPLODE*

La figure 2.13 montre la correspondance entre les sommets *EXPLODE* et *FORK*, pour des motifs répétitifs en aval des sommets *EXPLODE* et *FORK*, où SG_i correspond aux sous-graphes répétitifs (motifs) et $SG_1 = SG_2 = \cdots = SG_d = SG_i$.

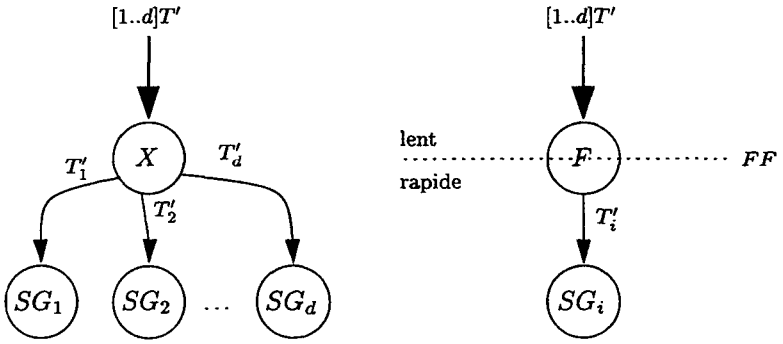


FIG. 2.13: Correspondance entre *FORK* et *EXPLODE*

Exemple

Soit un cube C de $d_1 \cdot d_2 \cdot d_3$ éléments :

$$C \equiv [1..d_1]C' \equiv [1..d_1][1..d_2]C'' \equiv [1..d_1][1..d_2][1..d_3]C'''$$

où C' correspond à une matrice, C'' à un vecteur et C''' à un élément.

Si $d_1 = d_2 = d_3 = 3$:

- l'opération *FORK* enumère 3 matrices C'_i :

  C'_1

  $\vdots$

  C'_3

- l'opération *FORK* appliquée sur chacune de ces matrices C'_i enumère 9 vecteurs C''_{ij} :

$$\begin{array}{lll} C''_{11} & \cdots & C''_{13} \\ \vdots & & \vdots \\ C''_{31} & \cdots & C''_{33} \end{array}$$

- l'opération *FORK* appliquée sur chacun de ces vecteurs C''_{ij} enumère 27 éléments C'''_{ijk} :

$$\begin{array}{lll} C'''_{111} \cdots C'''_{113} & \cdots & C'''_{131} \cdots C'''_{133} \\ \vdots & & \vdots \\ C'''_{311} \cdots C'''_{313} & \cdots & C'''_{331} \cdots C'''_{333} \end{array}$$

Les trois applications successives de l'opération *FORK* sur le cube C (tableau à trois dimensions : d_1, d_2, d_3) permettent d'énumérer n éléments, où $n = d_1 \cdot d_2 \cdot d_3$, comme le montre la figure 2.14.

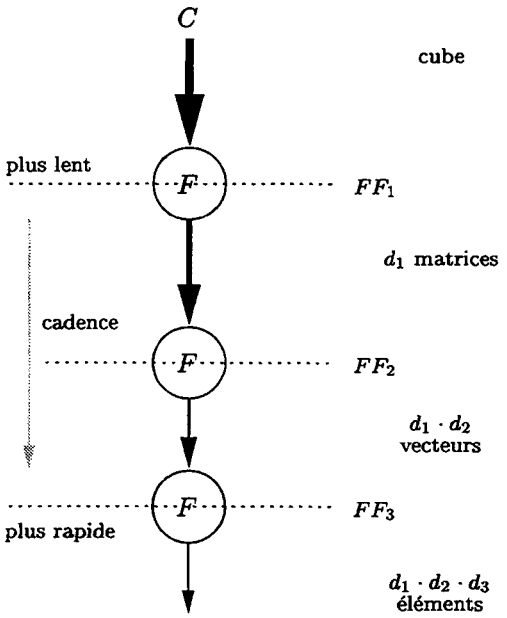


FIG. 2.14: Exemple d'utilisation de *FORK*

Sommet *JOIN*

Le sommet *JOIN*, identifié par J, effectue la factorisation des d flots de données T'_i, où $i \in \{1, \ldots, d\}$, en son entrée, en les collectant sous la forme d'un vecteur $[1..d]T'$, comme le montre la figure 2.15. d changements de la valeur des données en entrée du *JOIN* correspondent à un changement de la valeur des données en sa sortie. Le sommet *JOIN* permet de traverser une frontière de factorisation, tout en réduisant la cadence des données : en amont du *JOIN* la cadence des données est d fois supérieure à la cadence des données en aval de *JOIN*. Ainsi, une frontière de

factorisation FF, définie par un sommet *JOIN*, sépare le côté "lent" (cadence moins élévée) du côté "rapide" (cadence plus élévée) de ce sommet. Les d sous-graphes (identiques, motif répétitif) en amont de l'*IMPLODE* (cf. 2.2.2) sont également factorisés en un seul sous-graphe en amont du *JOIN*. C'est l'opération inverse du FORK et le correspondant factorisé de l'*IMPLODE*.

$$J\left(T'_i\right) \Longrightarrow [1..d]T' \tag{2.9}$$

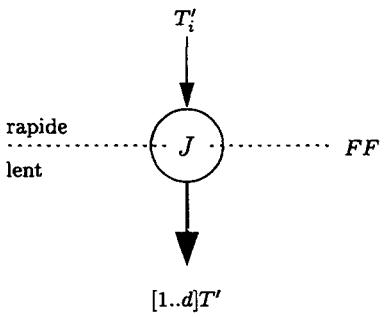


FIG. 2.15: Sommet *JOIN*

Correspondance entre *JOIN* et *IMPLODE*

La figure 2.16 montre la correspondance entre les sommets *IMPLODE* et *JOIN*, pour des motifs répétitifs en amont des sommets *IMPLODE* et *JOIN*, où SG_i correspond aux sous-graphes répétitifs (motifs) et $SG_1 = SG_2 = \cdots = SG_d = SG_i$.

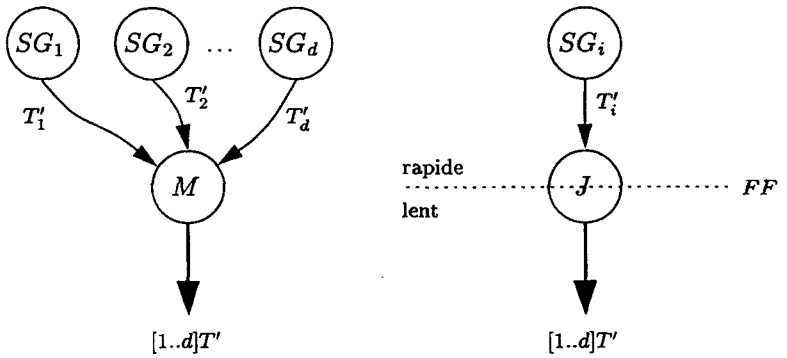


FIG. 2.16: Correspondance entre *IMPLODE* et *JOIN*

Exemple

Soit les n éléments C'''_{ijk} d'un cube C, où $n = d_1 \cdot d_2 \cdot d_3$ et $d_1 = d_2 = d_3 = 3$:

$$\begin{array}{ccc} C'''_{111} \cdots C'''_{113} & \cdots & C'''_{131} \cdots C'''_{133} \\ \vdots & & \vdots \\ C'''_{311} \cdots C'''_{313} & \cdots & C'''_{331} \cdots C'''_{333} \end{array}$$

- l'opération *JOIN* collecte les n éléments de C en 9 vecteurs C''_{ij} :

$$\begin{array}{ccc} C''_{11} & \cdots & C''_{13} \\ \vdots & & \vdots \\ C''_{31} & \cdots & C''_{33} \end{array}$$

- l'opération *JOIN* collecte les 9 vecteurs C''_{ij} en 3 matrices C'_i :

$$\begin{array}{c} C'_1 \\ \vdots \\ C'_3 \end{array}$$

- l'opération *JOIN* collecte les 3 matrices C'_i en un cube C :
  $C \equiv [1..d_1]C' \equiv [1..d_1][1..d_2]C'' \equiv [1..d_1][1..d_2][1..d_3]C'''$.

Les trois applications successives de l'opération *JOIN* sur les n éléments du cube C ont produit le cube C (tableau à trois dimensions : d_1, d_2, d_3), comme le montre la figure 2.17.

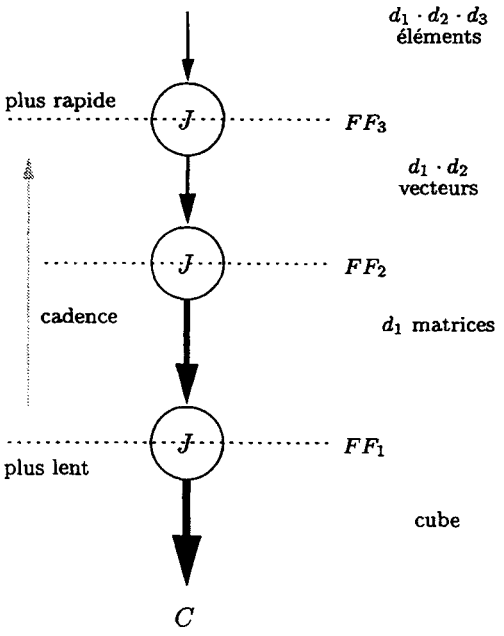


FIG. 2.17: Exemple d'utilisation de *JOIN*

Sommet *ITERATE*

Le sommet *ITERATE*, identifié par I, effectue la factorisation des dépendances de données inter-motifs. Lorsque d sous-graphes SG (identiques, motif répétitif) sont factorisés, chaque dépendance de données inter-motif (entre deux sous-graphes adjacents) apparaît en sortie et en entrée du sous-graphe factorisé. Le sommet *ITERATE* permet de marquer chaque dépendance de données inter-motif, afin que le cycle apparent soit marqué comme n'étant pas un cycle de dépendance. L'entrée e_i reçoit les données qui arrivent du motif et la sortie s_i fournit les données d'entrée du motif. Le sommet *ITERATE* permet de spécifier une connexion inter-motifs, qui apparaît dans le graphe factorisé du motif comme un cycle à travers un sommet *ITERATE* :

- côté "rapide" de la frontière de factorisation :
  sortie du *ième* SG = entrée du $(i+1)$*ème* SG,
  sortie du *ième* $SG \rightarrow$ entrée e_i d'*ITERATE*, sortie s_i d'*ITERATE*, $\rightarrow$ entrée *ième* SG ;
- côté "lent" de la frontière de factorisation :
  entrée du premier SG ("initial") et sortie du dernier SG ("final") = entrée *init* et sortie fin d'*ITERATE*, respectivement.

La figure 2.18 montre le sommet *ITERATE*.

$$I(init, e_i) \Longrightarrow (s_i, fin) \tag{2.10}$$

Soit d le nombre de sous-graphes à être factorisés en un seul graphe et i une itération quelconque entre 1 et d :

- pour $i = 1 \Longrightarrow s_i = init$;
- pour $1 < i \leq d \Longrightarrow s_i = e_{i-1}$;
- pour $i = d \Longrightarrow fin = e_i$.

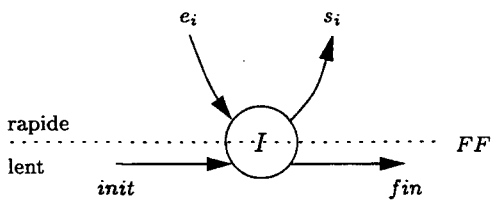


FIG. 2.18: Sommet *ITERATE*

La cadence des flots de données intermédiaires (e_i et s_i) est d fois supérieure à la cadence des flots de données "initial" ($init$) et "final" (fin). Ainsi, les flots de données "initial" ($init$) et "final" (fin) sont du côté "lent" de la frontière de factorisation FF. Du côté "rapide" de FF, on trouve les flots intermédiaires d'entrée (e_i) et de sortie (s_i), comme le montre la figure 2.18.

Utilisation du sommet *ITERATE*

La figure 2.19 montre l'utilisation de l'opération *ITERATE* pour factoriser la dépendance entre les sous-graphes SG_1, SG_2, ..., SG_d, où SG correspond au sous-graphe répétitif (motif) et $SG_1 = SG_2 = \cdots = SG_d = SG_i$.

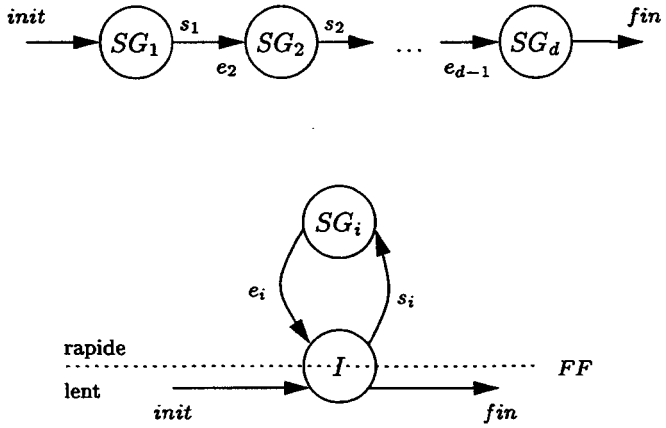


FIG. 2.19: Utilisation du sommet *ITERATE*

Exemple

Soit un vecteur A de 4 éléments a_1, a_2, a_3 et a_4, nous souhaitons calculer l'addition de ses éléments du vecteur A :

$$s = \sum_{i=1}^{4} a_i \tag{2.11}$$

Quand on défactorise le graphe correspondant à l'éq. 2.11 (figure 2.20a), on remplace le sommet *FORK* (F) par le sommet *EXPLODE* (X), le sommet d'addition (*add*) et le sommet *ITERATE* (I) par 4 sommets d'addition (add_1, add_2, add_3 et add_4), comme le montre la figure 2.20b.

Sommet *DIFFUSION*

Le sommet *DIFFUSION*, identifié par D, effectue la factorisation d'un flot de données T traversant une frontière de factorisation FF. Il ne modifie pas ce flot de données, le diffusant tel qu'il se présente du côté "lent" de la frontière vers le côté "rapide", comme le montre la figure 2.21. Le sommet *DIFFUSION* est donc utilisé pour marquer la frontière de factorisation au niveau d'un arc qui la traverse.

$$D\Big(T\Big) \Longrightarrow T \tag{2.12}$$

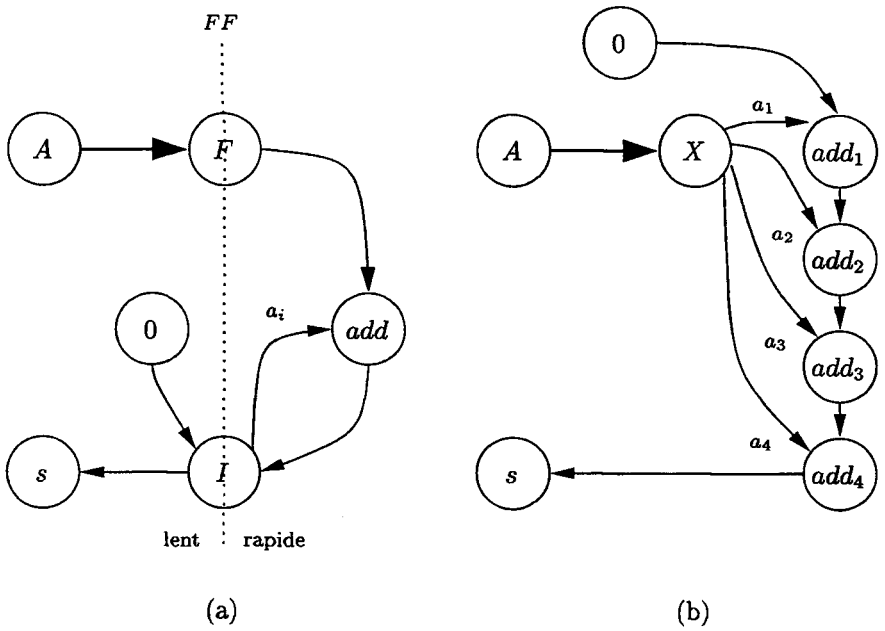


FIG. 2.20: Exemple d'utilisation d'*ITERATE*

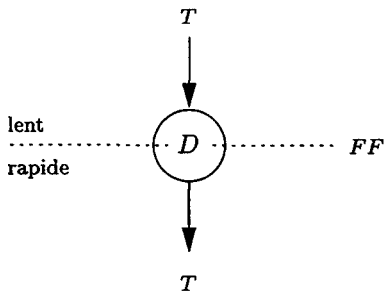


FIG. 2.21: Sommet *DIFFUSION*

Utilisation du sommet *DIFFUSION*

La figure 2.22 montre l'utilisation du sommet *DIFFUSION* pour factoriser la diffusion des données T vers les d sous-graphes SG_1, SG_2, ..., SG_d, où SG correspond au sous-graphe répétitif (motif) et $SG_1 = SG_2 = \cdots = SG_d = SG_i$.

2.3.2 Exemple : factorisation d'un produit matrice-vecteur

En reprenant l'exemple du produit matrice-vecteur utilisé dans la section 2.2.3, nous considèrons que, en notation algébrique, une énumération est, soit intuitivement abrégée par des points de suspension $(A_1, \cdots, A_n)$, soit plus formellement factorisée par des foncteurs (opérations sur les opérations, comme $\sum_{1 \leq j \leq n}$ et $(\,)_{1 \leq j \leq m}$, spécifiant un nombre de répétitions et un indice pour indexer les symboles de valeurs

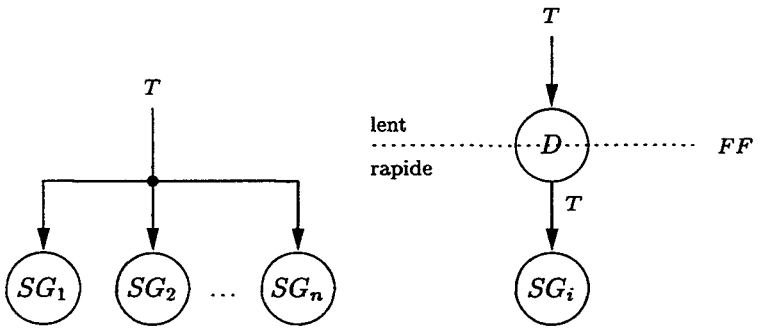


FIG. 2.22: Utilisation du sommet *DIFFUSION*

regroupées par la factorisation [LS97]). La figure 2.23 présente la décomposition du produit matrice-vecteur en produits scalaires (à gauche, pour $m = n = 3$) et sa factorisation (à droite), laquelle fait appel à trois sommets frontières de factorisation de motifs de graphes répétitifs finis (D, F, et J). Ces sommets délimitent la frontière, mise en évidence par des pointillés, entourant le motif de la factorisation pour le séparer du reste du graphe.

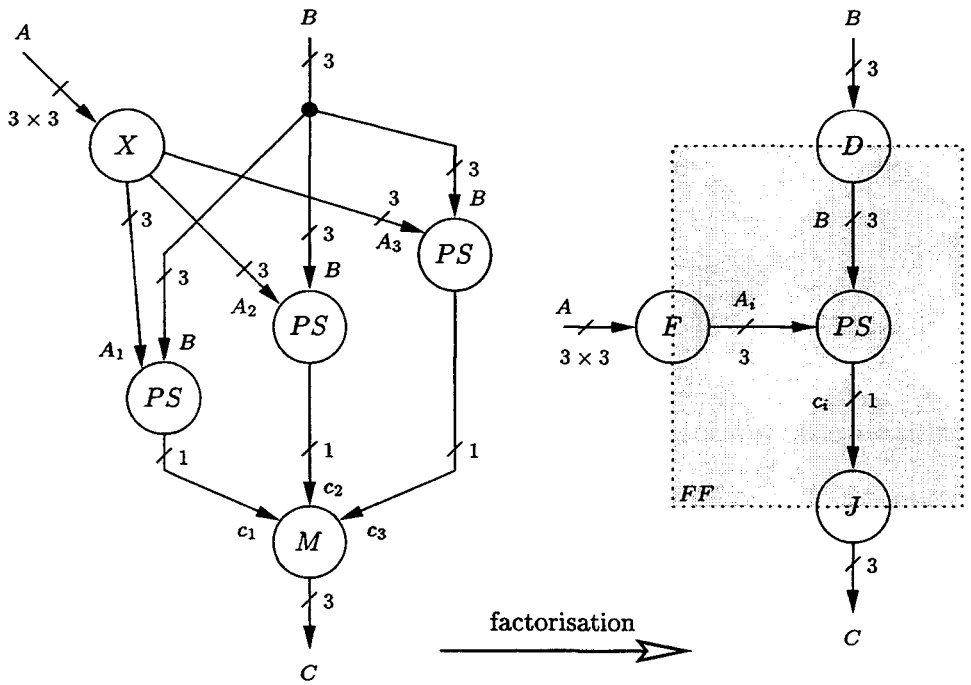


FIG. 2.23: Factorisation d'un produit matrice-vecteur

La figure 2.24 présente la décomposition du produit scalaire en termes d'une somme de produits et sa factorisation, qui fait apparaître le sommet frontière I (*ITERATE*). Chaque sommet frontière spécifie l'une des différentes manières de factoriser les arcs, en traversant la frontière de factorisation.

2.3.3 Factorisation de motifs de graphes répétitifs infinis : description des sommets de factorisation

Un système réel intéragit avec son environnement d'une manière discrète, sous la forme d'une répétition infinie de la séquence *acquisition-calculs-commande*. Alors, pour représenter les graphes correspondants, les sommets spéciaux frontières de factorisation *FORK*, *JOIN*, *ITERATE* et *DIFFUSION* doivent assurer l'interface avec l'environnement. Un graphe répétitif infini factorisé est nommé dans la littérature, un *graphe flot de données* [LS97].

On y retrouve les mêmes types de sommets que pour les graphes finis : $FORK^{\infty}$, $JOIN^{\infty}$, $ITERATE^{\infty}$ et $DIFFUSION^{\infty}$ sont équivalents respectivement aux entrées, sorties, retards et constantes des graphes flots de données. Nous verrons que les versions finie et infinie de ces sommets ont des implantations matérielles différentes (sauf les sommets *DIFFUSION*).

Une frontière de factorisation infinie, FF^{∞}, ne peut pas être défactorisée sans que l'on modifie la spécification des entrées et/ou des sorties du système.

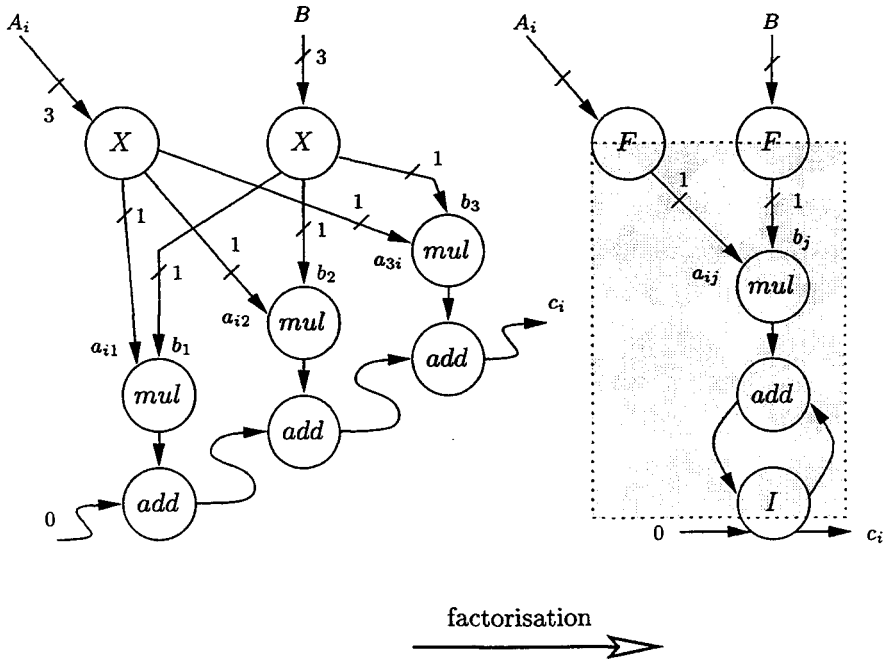


FIG. 2.24: Factorisation d'un produit scalaire

2.4 Frontières de factorisation

2.4.1 Définition

Une frontière de factorisation est une abstraction délimitée par des sommets frontières de factorisation, lequels factorisent des groupes de données utilisées ou produites par une même opération située du côté "rapide" de cette frontière. Une frontière de factorisation peut être délimitée par un ou plusieurs sommets frontières de factorisation.

2.4.2 Relations entre frontières de factorisation

Une frontière de factorisation sépare deux zones, l'une "rapide" et une "lente". Pendant que le côté "lent" est exécuté une fois, le côté "rapide" est exécuté n fois. D'un point de vue comportemental ou opératoire, une frontière peut être consommatrice (située en aval) ou/et productrice (située en amont) par rapport à une autre frontière, en fonction des dépendances de données entre elles. Deux frontières sont voisines s'il existe entre elles au moins une relation de dépendance directe, qui ne passe pas par l'intermédiaire d'une troisième frontière.

Les relations de voisinage entre les frontières de factorisation d'un graphe algorithmique peuvent être représentées sous la forme d'un graphe de voisinage. Les sommets de ce graphe représentent les frontières de factorisation et ils sont sous-divisés en quatre régions, comme le montre la figure 2.25 :

- lent-amont : côté "lent" de FF, consommatrice ;
- rapide-aval : côté "rapide" de FF, productrice ;
- rapide-amont : côté "rapide" de FF, consommatrice ;
- lent-aval : côté "lent" de FF, productrice.

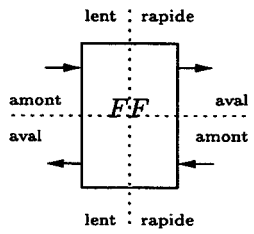


FIG. 2.25: Sommet d'un graphe de relations entre frontières

Les arcs orientés entrant et sortant des sommets représentent les transferts de données et les opérations entre les frontières. L'orientation de l'arc indique les relations de production-consommation : l'arc part d'un producteur vers un consommateur. Ce graphe de relations de voisinage nous permettra d'établir les relations de contrôle entre les frontières, comme nous verrons dans le chapitre 3, section 3.8.

2.4.3 Construction du graphe de relations entre frontières

Le graphe de relations entre frontières de factorisation (GRFF) d'un graphe algorithmique est construit à partir des relations de voisinage entre les frontières, comme décrit ci-dessous par l'Algorithme 1 :

Algorithme 1 Construction du graphe de relations entre frontières

begin

Identifier les sommets-source du graphe algorithmique appartenant à une même frontière de factorisation ;

Créer un sommet du graphe de relations de voisinage pour chaque frontière de factorisation existante dans le graphe algorithmique ;

Parcourir le graphe algorithmique et identifier les relations de voisinage entre les frontières ;

Créer un arc reliant les sommets du graphe de relations de voisinage, pour chaque relation différente de voisinage identifiée dans le graphe algorithmique.

end

2.5 Exemples de spécification algorithmique

Pour illustrer les concepts présentés dans ce chapitre, nous montrons ci-dessous deux exemples de spécification algorithmique, en utilisant le produit matrice-vecteur (PMV) et le filtrage d'une ligne d'image de d_1 pixels qui produit une ligne de d_2 pixels (la cadence de données en entrée et en sortie est au niveau des pixels).

2.5.1 Produit matrice-vecteur

Formalisation du PMV

Le produit d'une matrice $A \in R^m \times R^n$ par un vecteur $B \in R^n$ donne un vecteur $C \in R^m$, et peut s'écrire, sous forme factorisée :

$$C = \left[\sum_{j=1}^{n} a_{ij} b_j \right]_{i=1}^{m} \tag{2.13}$$

où

m : nombre de lignes de la matrice A,
n : nombre de colonnes de la matrice A, dimension du vecteur B,

a_{ij} : ijème élément de la matrice A,
b_j : $j$ème élément du vecteur B.

Dans l'éq. 2.13, on distingue deux composants :

1. le foncteur $\sum_{j=1}^{n}$, qui effectue la somme des produits entre les éléments a_{ij} de la $i$ème ligne de la matrice A et les éléments b_j du vecteur B ;
2. le crochet $\left[\ \right]_{i=1}^{m}$, qui représente la sélection des lignes de la matrice A de 1 à m.

Graphe algorithmique du PMV

On part de l'éq. 2.13, qui décrit le PMV sous la forme factorisée, afin d'obtenir le graphe algorithmique correspondant à la spécification algorithmique factorisée du PMV. Ce graphe algorithmique doit représenter l'interface avec l'environnement et les deux composants de l'éq. 2.13, en tenant en compte qu'on est dans le cadre d'un système réel de commande-contrôle.

L'interface avec l'environnement est composée de deux entrées (matrice A et vecteur B) et d'une sortie (vecteur C), correspondant à une frontière de factorisation de motifs de graphes répétitifs infinis (FF_1). Le crochet $\left[\ \right]_{i=1}^{m}$ correspond à une deuxième frontière de factorisation, délimitée par des sommets de factorisation de motifs de graphes répétitifs finis, qui effectuent la sélection des m lignes de la matrice A, la diffusion du vecteur B et la collecte du vecteur-résultat C. Le foncteur $\sum_{j=1}^{n}$ correspond à une troisième frontière de factorisation (FF_3), délimitée aussi par des sommets de factorisation de motifs répétitifs finis, qui effectuent la sélection des éléments a_{ij} de la $i$ème ligne de la matrice A, la sélection des éléments b_j du vecteur B et la factorisation des dépendances de données inter-motifs, fournissant le résultat de la somme des produits entre a_{ij} et b_j pour chaque ligne de la matrice A, comme le montre la figure 2.26.

La frontière FF_1 est délimitée par deux sommets $FORK\infty$ (F_A^∞ et F_B^∞) et un sommet $JOIN\infty$ (J_C^∞). Les sommets F_A^∞ et F_B^∞ correspondent à des capteurs, fournissant respectivement la matrice A et le vecteur B. Le sommet J_C^∞ correspond à un actionneur, recevant le vecteur-résultat C.

La frontière FF_2 est délimitée par un sommet $FORK$ (F_{21}), un sommet $DIFFUSION$ (D_{21}) et un sommet $JOIN$ (J_{21}). Le sommet F_{21} reçoit la matrice A en entrée et énumère ses m lignes en sortie. Le sommet D_{21} reçoit le vecteur B en entrée et le diffuse en sortie. Le sommet J_{21} reçoit les résultats des produits scalaires (PS) entre les lignes A_i et le vecteur B (c_i) et les collecte sous la forme d'un vecteur C en sortie.

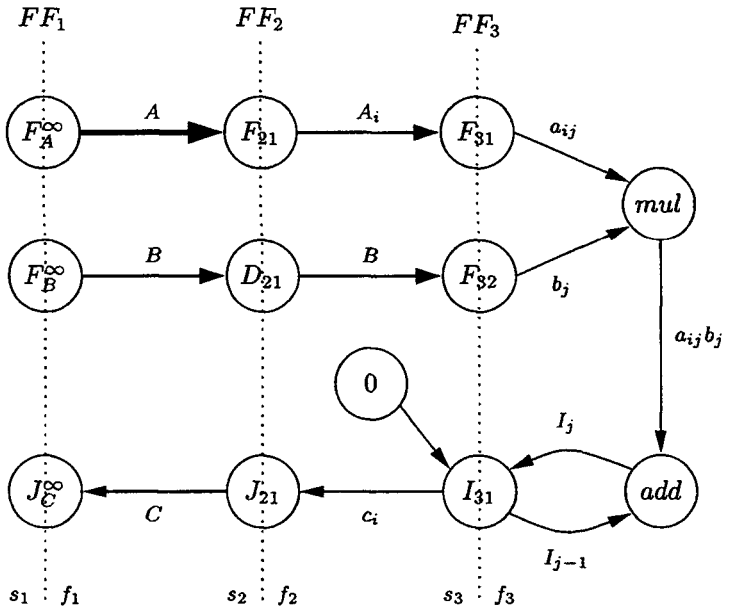


FIG. 2.26: Graphe algorithmique factorisé du PMV (spécification)

La frontière FF_3 est délimitée par deux sommets *FORK* (F_{31} et F_{32}) et un sommet *ITERATE* (I_{31}). Le sommet F_{31} reçoit en entrée la ligne A_i de la matrice A et énumère ses n éléments a_{ij} en sortie. Le sommet F_{32} reçoit en entrée le vecteur B et énumère ses n éléments b_j en sortie. Le sommet I_{31} effectue la factorisation des dépendances de données inter-motifs, étant le motif du graphe répétitif représenté par la figure 2.27. I_{31} est initialisé avec la valeur 0 et fournit en sortie le résultat du PS entre A_i et B.

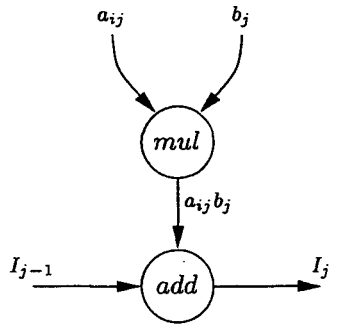


FIG. 2.27: Motif du PS

Le sommet 0 correspond à un sommet *DONNÉE*, qui fournit la valeur d'initialisation pour le sommet I_{31}. Les sommets *mul* et *add* sont des sommets *CALCUL*, qui représentent respectivement les opérations de multiplication et d'addition. Les côtés "lent" et "rapide" de chaque frontière sont étiquetés respectivement "s" (*slow*) et "f" (*fast*).

Le graphe de relations de voisinage entre frontières obtenu à partir du graphe algorithmique du PMV factorisé est montré par la figure 2.28. La frontière FF_1 est une frontière infinie. Ainsi, elle n'a pas de voisines de son côté "lent" (car celui-ci correspond à l'environnement). FF_1 est à la fois productrice (arcs A et B) et consommatrice (arc C) par rapport à FF_2. FF_2 est à son tour aussi productrice (arcs A_i et B) et consommatrice (arc c_i) par rapport à FF_3. FF_3 est productrice et consommatrice par rapport à elle-même, à travers les opérations de calcul *mul* et *add*.

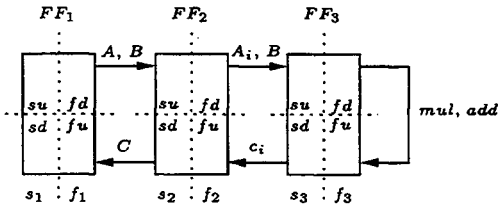


FIG. 2.28: Relations de voisinage entre frontières du PMV factorisé

2.5.2 Filtrage des lignes d'une image

On part de l'éq. 2.14, qui décrit le filtrage des lignes d'une image sous la forme factorisée, afin d'obtenir le graphe algorithmique correspondant à la spécification algorithmique factorisée.

$$pxlout(i) = \sum_{i=1}^{m} \frac{pxlin(i) + pxlin(i+1) + pxlin(i+2)}{3} \tag{2.14}$$

où

m : nombre de pixels de la ligne filtrée, $m = n - 2$,
n : nombre de pixels de la ligne à filtrer,
pxlin : ligne à filtrer,
pxlout : ligne filtrée.

L'interface avec l'environnement est assurée par les sommets F_{11}^{∞} (capteur qui produit des pixels) et J_{41}^{∞} (actionneur qui consomme des pixels). F_{11}^{∞} est le seul sommet de la frontière "infinie" FF_1 et J_{41}^{∞} est le seul sommet de la frontière "infinie" FF_4. Les pixels produits par F_{11}^{∞} sont consommés par le sommet J_{21} (qui les collecte pour former une ligne de n pixels de l'image). J_{21} est le seul sommet de la frontière FF_2. La ligne de n pixels produite par J_{21} est soumise à une opération de filtrage qui produit en sortie une ligne de m pixels en sortie. Cette ligne est consommée par le sommet F_{31}, qui sépare les pixels de la ligne d'entrée. F_{31} est le seul sommet de la frontière FF_3. Le graphe algorithmique de ce filtrage de lignes d'une image est montré par la figure 2.29.

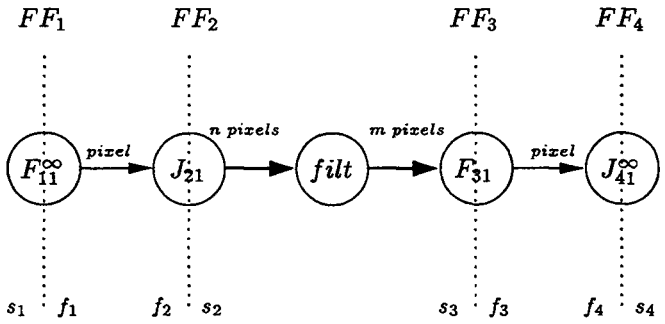


FIG. 2.29: Graphe algorithmique du filtrage des lignes d'une image

Le graphe de relations de voisinage entre frontières obtenu à partir du graphe algorithmique du filtrage factorisé est montré par la figure 2.30. Les frontières FF_1 et FF_4 sont des frontières infinies. Ainsi, elles n'ont pas de voisines de leur côtés "lent", puisque celui-ci correspond à l'environnement. FF_1 est productrice par rapport à FF_2. FF_2 est productrice par rapport à FF_3 à travers l'opération *filt*. FF_3 est productrice par rapport à FF_4.

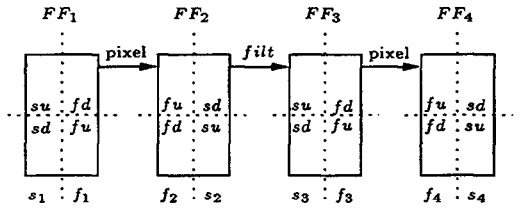


FIG. 2.30: Relations de voisinage entre frontières du filtrage factorisé

2.6 Conclusion

Dans ce chapitre, nous avons présenté l'état de l'art sur la spécification, la modélisation et la validation de systèmes embarqués temps réel, dans le but de situer notre stratégie de spécification par rapport aux techniques existantes. Nous avons défini notre modèle de graphes factorisés et nous avons présenté les sommets de base et de factorisation de motifs de graphes répétitifs finis et infinis utilisés pour les décrire. Ces sommets sont utilisés pour représenter une spécification algorithmique décrite sous la forme d'un graphe factorisé de dépendances de données. Cette forme de description nous permet d'établir des relations de voisinage entre les frontières de factorisation d'une spécification algorithmique. L'intérêt de ces relations est de permettre d'en déduire les relations de contrôle intra et inter-frontières de factorisation.

Le concepteur peut spécifier l'application directement sous la forme d'un graphe flot de données par l'intermédiaire de l'interface graphique de *SynDEx*, ou il peut la spécifier par l'intermédiaire d'un langage synchrone qui utilise le format commun DC (*Esterel*, *Lustre*, *Signal*). Comme nous nous intéressons à la conception de

systèmes réactifs embarqués temps réel, lequels sont caractérisés par des répétitions périodiques d'un ensemble d'opérations, nous les modélisons par l'intermédiaire d'un modèle basé sur les GFD : les graphes factorisés de dépendances de données (GFDD). La spécification à partir des langages synchrones offre au concepteur la possibilité de se servir des mécanismes de validation associés à ces langages.

Dans le chapitre suivant, nous montrons comment le modèle GFDD est utilisé pour représenter une implantation matérielle, laquelle comprend généralement deux parties : le calcul (le chemin de données) et le contrôle (le chemin de contrôle). Le contrôle assure le séquencement temporel des opérations. Nous montrons également comment, à partir d'une spécification algorithmique, décrite par l'intermédiaire de ce modèle, nous pouvons extraire l'implantation matérielle, laquelle décrit, sous la forme d'un graphe d'opérateurs factorisé, aussi bien la partie calcul (les opérateurs) que la partie contrôle (séquencement des calculs sur les opérateurs).

Chapitre 3

Implantation matérielle

The system functionnality can best be understood during the earliers design steps, before a lot of implementation details have been added, and this is why these early phases of the process have become so crucial in system design [GVNG94].

3.1 Introduction

L'implantation matérielle est le résultat de la réalisation matérielle de la spécification algorithmique obtenue après l'application de transformations spatio-temporelles sur la spécification algorithmique initiale. Nous effectuons l'implantation matérielle d'un algorithme à partir de la traduction du graphe algorithmique dans un graphe matériel. Pour une même spécification algorithmique, plusieurs implantations matérielles sont possibles, mais nous nous intéressons à une implantation bien particulière : celle qui respecte les contraintes d'exécution en temps réel et qui, simultanément, minimise la consommation de ressources matérielles. Ainsi, nous sommes dans le cadre d'un problème d'optimisation sous contraintes.

L'implantation consiste à faire correspondre, par traduction directe, à chaque sommet du graphe factorisé de dépendances de données (GFDD) d'une spécification, un opérateur (composant d'une bibliothèque VHDL : combinatoire pour un sommet opération, multiplexeur et/ou registre pour un sommet frontière) et, à chaque arc, une connexion entre opérateurs. La synchronisation des transferts de données entre les opérateurs synchrones (qui contiennent des registres) est générée trivialement à partir de l'analyse des relations entre les frontières de factorisation.

L'implantation matérielle qui réalise la spécification algorithmique peut être obtenue par compilation d'une *netlist* générée par traduction directe du graphe de dépendances de données entre sommets, tout en remplaçant chaque sommet par l'opérateur qui l'implante, et chaque dépendance de données par un média de communication interconnectant les ports des opérateurs. Le parallélisme potentiel, lié à l'ordre partiel établi par les dépendances de données entre opérations, devient ainsi un parallélisme effectif entre opérateurs [LS97].

Dans ce chapitre, nous donnons des règles simples qui permettent de synthétiser les chemins de données et de contrôle du circuit correspondant à l'algorithme spécifié, à l'aide d'un modèle de graphe de dépendances factorisé. Jusqu'au début des années 90, l'on croyait que la synthèse du chemin de contrôle était la plus difficile à réaliser [BCP90]. Pourtant, plusieurs travaux ont récemment démystifié cette difficulté [ET93, RK94, HW94, Ben*98, PA99], comme nous verrons dans la section 3.1.4. Nous montrons ici qu'il est possible de synthétiser le chemin de contrôle, de manière simple et systématique, à l'aide d'une technique de synchronisation des transferts de données entre registres. Cette approche devrait permettre par la suite de réaliser trivialement un générateur automatique de VHDL structurel synthétisable, réalisant ainsi la synthèse automatique de circuits. Nous y présentons également un état de l'art sur la synthèse de circuits et, plus particulièrement, sur la synthèse aux niveaux d'abstraction *comportemental* et *transfert de registres*, qui sont les niveaux ciblés par ce travail. Afin de permettre de situer notre approche par rapport aux outils et méthodes existantes, nous décrivons succinctement quelques systèmes de synthèse.

3.1.1 Synthèse de circuits

La synthèse peut être considérée comme l'action qui permet un changement de domaine de représentation et/ou de niveau d'abstraction. En fonction du niveau d'abstraction des descriptions d'entrée et de sortie, on choisit parmi plusieurs outils de synthèse [Ben93, OBr93], dont les modèles de conception et les points de synchronisation sont différents pour chaque niveau d'abstraction.

Gajski [GK83] a introduit un diagramme en Y décrivant les trois vues d'une architecture, comme le montre la figure 3.1. On y distingue trois domaines pour exprimer une architecture : *comportemental* (il présente son comportement en se basant sur la fonction qu'elle réalise), *structurel* (il décrit l'agencement de ses composants électroniques) et *physique* (il se concentre sur le plan définissant la combinaison des matériaux qui réalisent physiquement le circuit). À l'intérieur de chacun de ces domaines, on distingue de différents niveaux de description, plus ou moins abstraits, par rapport aux détails du circuit. Ces niveaux d'abstraction nous permettent de faire face à la complexité des systèmes à implanter (figure 3.2) :

- Niveau système : il est généralement exprimé en termes de processus communicants. Dans le domaine structurel, il correspond à des interconnexions d'éléments, tels que les processeurs, les mémoires, les unités périphériques ;

- Niveau algorithme : il s'intéresse au comportement de chaque processeur individuellement, c'est-à-dire à la manière dont celui-ci fait correspondre une séquence de sortie à une séquence d'entrée. Structurellement, il peut se composer d'éléments architecturaux, tels que la mémoire, le processeur, les portes ;
- Niveau transfert de registres : il décrit les transferts et les opérations entre registres à chaque état. Dans le domaine structurel, il est représenté par les registres, les opérateurs arithméthiques et logiques ;
- Niveau portes : il se fait à travers d'expressions booléennes. Structurellement, il est représenté par des blocs logiques (combinatoires et bascules) ;
- Niveau transistor : il décrit des fonctions de transfert, des équations de réseaux, etc. Dans le domaine structurel, il se présente comme l'interconnexion de composants électroniques, tels que les transistors, les diodes.

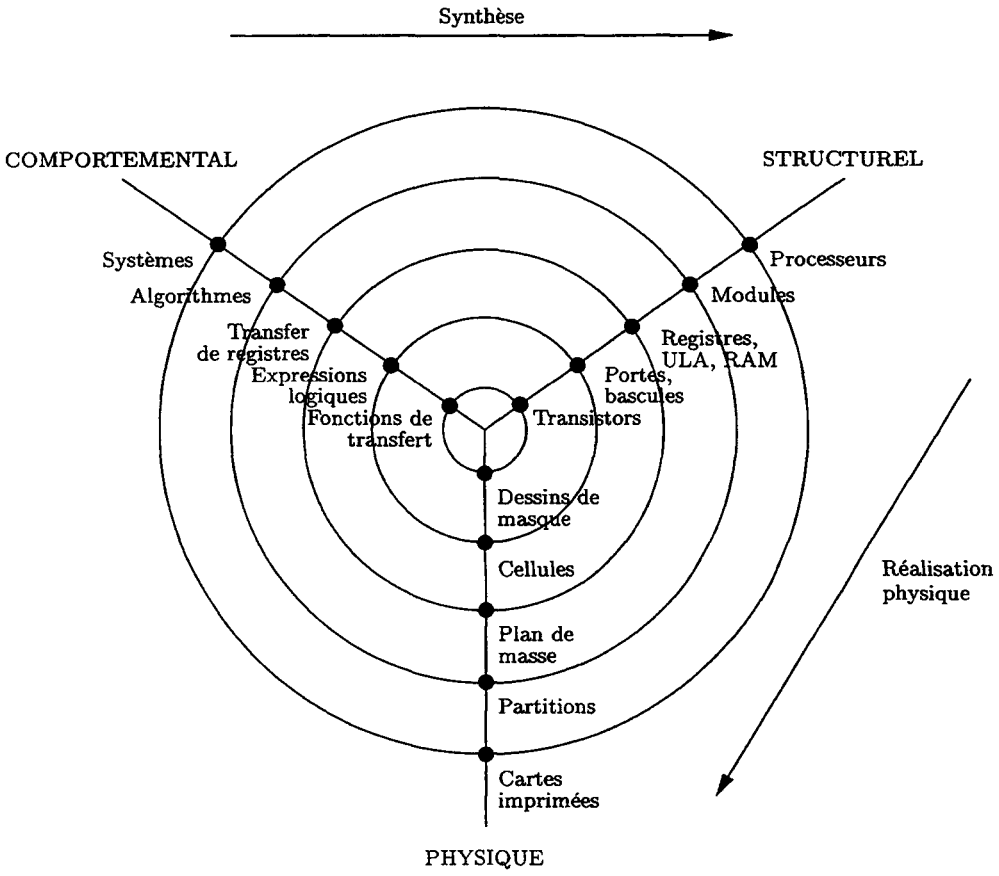


FIG. 3.1: Les trois vues d'une architecture

La synthèse est très sensible au style de la spécification d'entrée. Deux spécifications fonctionnellement équivalentes, écrites en styles différents, peuvent produire des résultats de qualités très différentes en termes de surface et de temps de réponse [Cam96].

La synthèse de haut niveau (HLS–*High-Level Synthesis*) est le processus qui met en correspondance une spécification comportementale en langage de description matérielle avec un réseau RTL (*Register Transfer Level*). La synthèse de haut niveau utilise des modèles internes basés sur des graphes flot de données/contrôle (CDFG–*Control Data Flow Graph*) et produit un modèle RTL de l'implantation matérielle pour un ordonnancement donné. La sortie d'un synthétiseur de haut niveau consiste généralement de deux parties [Lin97] : une structure chemin de données au niveau RTL et une spécification de la FSM (*Finite State Machine*), qui contrôle ce chemin de données. Pour que la HLS soit efficace, on doit être capable de prédire l'effet d'une certaine décision, prise au niveau système ou comportemental (par exemple l'ordonnancement et la distribution), sur l'implantation matérielle au niveau RTL. Jusqu'à présent, cet effet ne peut pas être mesuré avec précision, puisque le modèle CDFG est très différent des modèles RTL/niveau de portes utilisés par les outils de synthèse logique [Ber99], d'où l'intérêt de disposer d'un modèle unifié pour représenter l'algorithme, l'architecture et l'implantation de l'algorithme sur l'architecture, comme celui proposé par notre méthodologie AAA.

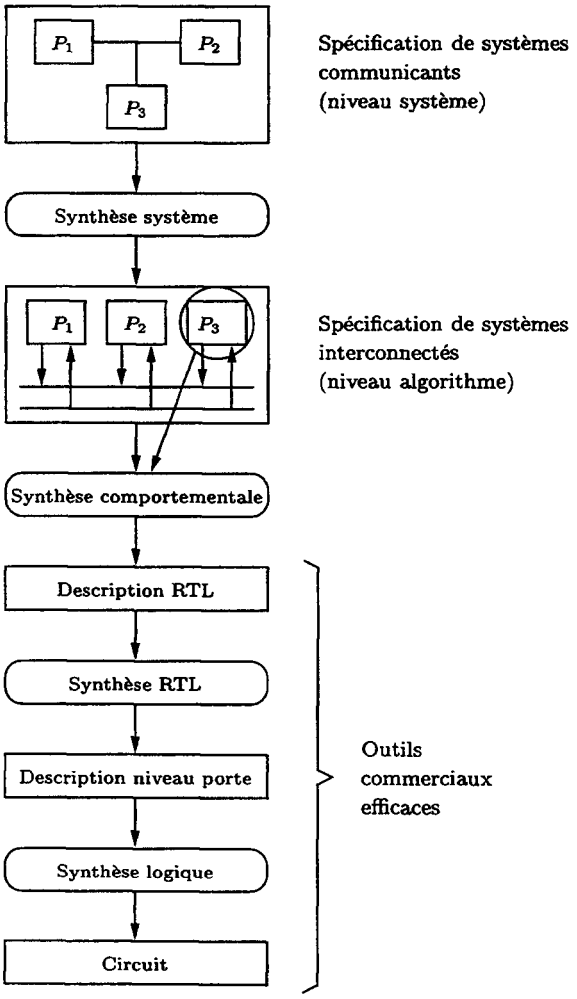


FIG. 3.2: Les niveaux de synthèse

Dans le cas des systèmes embarqués, la synthèse est une combinaison de processus manuels et automatiques. Elle est fréquemment réalisée en trois étapes [ELLS97] : (1) la mise en correspondance (*mapping*) d'une architecture, où le concepteur choisit la structure générale de l'implantation; (2) le partitionnement, où les éléments de la spécification sont associés, de manière manuelle ou automatique, aux unités architecturales et ; (3) la synthèse logicielle et matérielle, où les unités architecturales sont décrites en détails. Dans le cas des systèmes mixtes matériel/logiciel, les outils de co-synthèse matérielle-logicielle doivent explorer les rapports et les interdépendances entre les deux domaines (matériel et logiciel), tenant toujours compte de l'impact d'une décision concernant un domaine sur l'autre domaine. Cela exige une bonne compréhension des effets des décisions sur le coût et la performance du système complet.

Dans ce travail, nous nous restreindrons aux architectures mono-FPGA. Ainsi, le choix de l'architecture est effectué à priori, restant au concepteur le choix du modèle de FPGA (le nombre de CLB–*Configurable Logic Block*, le nombre d'entrées-sorties, la consommation, la vitesse, etc.), en fonction des contraintes de l'application. Le partitionnement est trivial, puisque nous ciblons une architecture monocomposant. Ainsi, la mise en correspondance entre les éléments de la spécification (les sommets et les arcs du graphe algorithmique) peut être réalisée de façon automatique, par l'intermédiaire de la traduction directe du graphe algorithmique vers le graphe matériel. La partie contrôle est également générée automatiquement, selon les règles présentées dans la section 3.7. Comme notre architecture-cible est mono-FPGA, nous ne réalisons que la synthèse matérielle. Le graphe matériel détaillé, composé d'une partie chemin de données et d'une partie chemin de contrôle, est traduit en code VHDL structurel synthétisable. Ce code est fourni à des outils de CAO, afin de générer la *netlist* de configuration du FPGA-cible.

Une fois intégrée à *SynDEx*, notre extension de la méthodologie AAA aux circuits reconfigurables permettra la synthèse de systèmes mixtes matériel/logiciel. Le concepteur pourra donc choisir comme cible une architecture multicomposants, contenant DSP, FPGA, processeurs, microcontrôleurs et/ou stations de travail. Le partitionnement deviendra beaucoup plus complexe, à cause de l'hétérogénéité de l'architecture. Il devra être semi-automatique, guidé par le concepteur. *SynDEx* effectue déjà la synthèse logicielle, générant un exécutif optimisé pour les composants programmables. Après la mise en œuvre de notre extension, *SynDEx* sera capable d'effectuer aussi la synthèse matérielle, générant un code VHDL synthétisable pour les composants configurables.

Nous décrivons ci-dessous les différents niveaux de synthèse et les principaux outils de synthèse correspondant à chacun de ces niveaux. Le tableau 3.1 présente les modèles de conception plus fréquemment utilisés par les outils de synthèse et les points de synchronisation entre les blocs de chaque modèle (processus, sommets d'un graphe, portes, etc.), en fonction des différents niveaux d'abstraction [CSSJ99]. Comme nous verrons par la suite, notre méthodologie se situe entre les niveaux comportemental et RTL, puisque la synthèse du chemin de données est effectuée à partir d'un modèle basé sur des graphes synchronisés par les événements

d'entrée/sortie (nous rappellons que nos applications-cible sont les systèmes embarqués temps réel, qui sont des systèmes réactifs, donc pilotés par des stimuli venus de l'environnement). La synthèse du chemin de contrôle, à son tour, part d'un modèle de graphes afin de générer les équations booléennes qui décrivent le contrôleur. Les sommets du graphe matériel sont synchronisés par une horloge globale.

Synthèse niveau système

Dans la synthèse *niveau système*, celui-ci est spécifié par son fonctionnement et par un ensemble de contraintes (temps, surface, consommation, etc.). Le système à synthétiser est partitionné en sous-systèmes, où chaque sous-système est considéré comme un processus communicant (CSP), synchronisé par l'intermédiaire des échanges de messages entre eux. Après le partitionnement, chaque processus peut être représenté au niveau comportementale par un CDFG synchronisé par les événements d'entrée/sortie [CSSJ99]. Plusieurs environnements de conception au niveau système ont été développés récemment. L'environnement *Tosca* [ABFS94, BFS96] cible les systèmes orientés-contrôle, en partant d'une spécification sous la forme de FSM hiérarchiques. Il génère en sortie un ensemble d'instructions virtuelles, qui assure la synthèse de façon indépendante du processeur-cible. L'environnement *SpecSyn* [GVNG98a, GVNG98b] part d'une spécification fonctionnelle, basée sur le modèle de calcul PSM (*Program-State Machine* – machine à états de programme) et produit une description de haut niveau, partitionnée en sous-systèmes logiciels et matériels, qui peut être simulée, éditée ou synthétisée par des outils des niveaux comportemental et RTL. D'autres outils qui effectuent la synthèse *niveau système* sont *Vulcan II* [GD93] et COSYMA [EHB93, Cos98]. Pourtant, il y a encore peu d'outils commerciaux de synthèse *niveau système* disponibles, puisqu'elles sont encore au stade de la recherche.

Synthèse comportementale

Dans la synthèse *comportementale* (ou *architecturale*), on part d'une spécification algorithmique en entrée, décrite par un langage procédural (VHDL [IEEE87], Verilog [TM91]) ou applicatif (*Silage* [Hil85], ELLA [MPT85]), ou par l'intermédiaire des modèles CFG (*Control Flow Graph*), DFG (*Data Flow Graph*), CDFG ou FSMD (*FSM with Datapath*). Cette spécification est traduite de façon automatique ou semi-automatique (traduction structurelle) dans une description matérielle (modèle RTL), représentée fréquemment sous la forme d'une architecture contrôleur/chemin de données. Par exemple, dans l'outil *Transe* [DKL92], qui utilise *Lustre* comme langage de spécification, on peut effectuer une synthèse *comportementale* par transformation de programme. Plusieurs travaux de recherche universitaires concernant les outils de synthèse comportementale ont été objets de transferts de technologie à l'industrie, comme *Amical* [KCBJ93, KDJ94] et *FPGA Express* [Vie00] de *Viewlogic*, mais il reste encore beaucoup de progrès à faire. Nous présentons, dans la section 3.1.2, les principaux outils de synthèse *comportementale* existants.

Synthèse transfert de registres

La synthèse *transfert de registres* traduit une description de niveau RTL, sous la forme de FSM, de BDD (*Binary Decision Diagram*) ou d'équations booléennes, contenant une partie contrôle et une partie chemin de données, dans une description au niveau porte. Le chemin de données est composé de trois types de composants : les unités fonctionnelles (ULA–Unité Logique-Arithmétique, multiplieurs et registres à décalage), les unités de stockage (registres et mémoires) et les unités d'interconnexion (bus et multiplexeurs). Le contrôleur spécifie l'ensemble de micro-opérations appliquées sur le chemin de données pendant chaque étape de contrôle [Lin97]. Au niveau RTL, les transferts de données sont synchronisés par les fronts du signal d'horloge. La synthèse RTL met en correspondance les composants du contrôleur et du chemin de données avec une bibliothèque de cellules, afin de produire une *netlist* de portes. À partir du langage *Esterel* [Ber91], on peut effectuer la synthèse RTL en traduisant des programmes *Esterel* en circuits. Plusieurs outils de synthèse RTL sont disponibles, tels que *Synopsis* [Syn92], *Compass* [Com92], *Viewlogic* [Vie00], etc.

Synthèse logique

Les outils de synthèse *logique* partent d'une description sous la forme de blocs logiques (combinatoires et registres) interconnectés en entrée et synchronisés par le signal d'horloge. Leur but est d'optimiser ces blocs logiques en termes de surface du circuit généré [BHS93]. Les outils de synthèse logique font appel à des bibliothèques de composants technologiques qui seront utilisés pour l'implantation. À partir de *Lustre*, la synthèse logique est effectuée par l'intermédiaire d'une traduction structurelle [Bel94] qui correspond à une interprétation matérielle des opérateurs *Lustre*, et d'un prototype pour compiler les descriptions *Lustre* vers des circuits programmables du type FPGA [Roc92].

Synthèse physique

Dans la synthèse *physique*, l'on associe à une description dans le domaine structurel, sous la forme d'une *netlist* de portes ou de modèles de *layout*, une représentation dans le domaine physique, afin de produire le *layout* final de l'implantation. À ce niveau-là, la synchronisation est effectuée par les changements des valeurs des conducteurs. Plusieurs solutions sont possibles pour réaliser une architecture : la carte imprimée (PCB – *Printed Circuit Board*) qui contient plusieurs composants interconnectés ; les MCM (*Multi-Chips Modules*) qui contiennent plusieurs puces de silicium interconnectées ; les circuits programmables (PLA–*Programmable Logic-Array*, PAL–*Programmed Array Logic*, LCA–*Logic Cell Array* et FPGA) [Tex92] et les circuits intégrés de technologie CMOS (*Complementary Metal-Oxide Semiconductor*), ECL (*Emitter Coupled Logic*), BiCMOS (*Bipolar CMOS*), AsGa (Arséniure de gallium), etc. [CH90].

Notre méthodologie utilise les techniques de synthèse aux niveaux comportemental et RTL, en profitant de l'intersection existante entre ces niveaux d'abstraction, de façon à permettre la synthèse automatique du chemin de données et du chemin de contrôle. Le chemin de données est synthétisé en partant d'une spécification au niveau comportementale, qui sera utilisée pour générer une description au niveau RTL du chemin de contrôle. Il n'y a donc pas de rupture entre la synthèse comportementale et la synthèse RTL. Cette dernière étant obtenue par traduction de la spécification algorithmique.

Nous partons donc d'une spécification algorithmique sous la forme d'un GFD ou, plus particulièrement, d'un GFDD synchronisé par les événements d'entrée/sortie. Avant de générer l'implantation matérielle optimisée, nous explorons l'espace de solutions par l'intermédiaire de transformations spatio-temporelles appliquées à la spécification initiale (comme nous verrons dans le chapitre 4). Cela nous permettra de trouver la spécification correspondante à une implantation qui respecte les contraintes temporelles et qui minimise l'augmentation du nombre de ressources matérielles utilisées. À la fin de cette étape de synthèse comportementale, nous avons un graphe matériel, obtenu par traduction directe du graphe algorithmique transformé, et un ensemble d'équations, obtenu par analyse des relations de voisinage entre les frontières de factorisation du graphe algorithmique. Cet ensemble d'équations nous permettra de synthétiser la partie contrôle de l'implantation. La synchronisation entre les sommets du graphe matériel est assurée par un signal d'horloge. Après l'étape de synthèse RTL, nous disposons d'un code VHDL synthétisable, représentant les opérateurs interconnectés et synchronisés par un signal d'horloge. Ce code VHDL sera fourni en entrée des outils de CAO qui, à leur tour, effectueront la synthèse logique de l'architecture-cible. Dans notre cas, cela correspondra à la génération des *netlists* de configuration des FPGA.

TAB. 3.1: Points de synchronisation et modèles de conception aux différents niveaux d'abstraction

Niveau d'abstraction	*Points de synchronisation*	*Modèle de conception en entrée*
Système	Messages entre processus	CSP
Comportementale	Événements d'entrée/sortie	CFG, DFG, CDFG, FSMD
RTL	Horloge	FSM, BDD, équations booléennes
Logique	Horloge	Expressions logiques
Physique	Changements des valeurs des conducteurs	*Netlist* des portes et modèles de disposition de composants (*layout*)

3.1.2 Synthèse comportementale

Les méthodes de synthèse RTL actuelles sont fastidieux à utiliser, ce qui fait que le temps de conception soit très important. Ainsi, afin de faciliter la tâche du concepteur et de réduire le temps de conception, il faut partir d'un niveau d'abstraction plus élévé (comportementale ou système) et utiliser les techniques de HLS, pour automatiser la génération de descriptions aux niveaux moins abstraits, comme RTL ou portes. Les avantages de cette approche sont l'amélioration de la qualité de conception et la réduction du cycle de conception [M*99]. La conception de systèmes complexes à partir d'un modèle comportemental offre encore d'autres avantages, tels qu'une vérification fonctionnelle plus rapide (en termes de temps de simulation et de l'abstraction de la fonctionnalité), moins d'erreurs aux niveaux de conception moins abstraits et une maintenance de code plus facile (à cause de sa taille réduite par rapport au modèle RTL).

Moussa et al. [M*99] ont conclu que la méthodologie HLS est très efficace en termes de gestion de la complexité de l'application et en termes de temps de conception, par rapport à la méthodologie de synthèse RTL. En ce qui concerne la surface de l'implantation, les deux méthodologies sont presque équivalentes. Quant à la durée du cycle de conception, la synthèse comportementale est trois fois plus rapide que la synthèse RTL. En plus, le modèle RTL, généré automatiquement à partir de la synthèse comportementale, est 10 % plus compact et presque si lisible que le modèle RTL manuel.

Selon Camposano [Cam96], la migration de la synthèse RTL vers la synthèse comportementale permet d'augmenter la productivité du processus de conception jusqu'à cinq fois, sans dégrader la qualité des résultats concernant le temps d'exécution ou la surface occupée par l'implantation. La synthèse comportementale produit des implantations plus rapides que la synthèse RTL, parce qu'elle effectue plus d'exploration architecturale. Cela offre un meilleur point de départ pour la synthèse. La synthèse comportementale n'est pas d'usage général comme la synthèse RTL ou la synthèse logique. Les applications les plus adaptées à la synthèse comportementale sont celles caractérisées par un fort contenu algorithmique, comme les applications graphiques, les contrôleurs de disques et d'imprimantes, le traitement du signal et des images et les calculs arithmétiques complexes. Les applications non adaptées à ce niveau de synthèse sont les calculateurs à haute performance, les contrôleurs totalement décrits par des FSM et la logique combinatoire.

La principale différence entre les modèles comportemental et RTL réside dans la gestion du temps. Au niveau transferts de registres, le comportement de l'application est décrit en termes de cycles d'horloge. Un modèle comportemental est généralement spécifié en termes d'étapes de calcul. La tâche principale de la synthèse comportementale consiste à séparer ces étapes de calcul en cycles d'horloge, afin de produire un modèle RTL.

Groupes d'outils de synthèse comportementale

Il existe deux groupes d'outils de synthèse comportementale, à savoir : (i) des outils généraux, qui ne visent pas un domaine d'application particulier, comme *Mimola* [Zim80, Mar86], *Hal* [PKG86], *Maha* [PPM86], VSS [Uni92], *Olympus* [DKMT90], *Easy* [SV88], OSYS [Ben93], *Yorktown Silicon Compiler* [B*88], HIS [C*91], *Callas* [L*93], etc. Ces outils ciblent une architecture générique de type contrôleur et chemin de données ; et (ii) des outils spécifiques à des domaines d'application particuliers, comme *First* [DR85], *Larger* [S*91], *Spaid* [HE89], *Cathedral* [DCG*90, L*90, VVB*93], *Piramid* [W*90], *Alpha* [LMQ91], *Hyper* [CPTR89, RP90], *Phideo* [L*91], *Gaut* [MSDP93], etc. L'architecture ciblée par ces outils reflète les propriétés typiques des domaines d'application (la régularité, la complexité du chemin de données, etc.). La majorité des outils ci-mentionnés traite les applications du type traitement du signal ou les applications contenant de traitements intensifs. Les architectures ciblées, à leur tour, peuvent être de type bit-série, bit-parallèle, régulières ou systoliques.

Les outils de synthèse comportementale pour les systèmes de traitement du signal utilisent des caractéristiques particulières de ce domaine pour obtenir des architectures plus performantes [VVB*93] :

- le débit de données constant, qui permet d'évaluer statiquement la mémoire nécessaire ;
- le nombre limité de répétitions, qui permet de déterminer le chemin critique et d'effectuer des optimisations ;
- le concept de flot, qui permet de considérer les retards comme des objets prédéfinis.

Notre méthodologie cible les systèmes réactifs temps réel, orientés au contrôle-commande et au traitement du signal et des images. Ainsi, nous profitons des caractéristiques énumérées ci-dessus pour obtenir des implantations plus performantes : (i) la mémoire nécessaire peut être estimée à partir de l'analyse du graphe algorithmique obtenu après l'optimisation ; (ii) le modèle GFDD, utilisé pour représenter la répétition de motifs dans le graphe algorithmique, nous permet d'estimer le temps de réponse de l'implantation, à partir de la latence des opérateurs composant le graphe matériel et du nombre de cycles nécessaires à l'exécution de l'application ; (3) le concept de flot nous permettra d'utiliser les techniques de pipeline pour réduire le chemin critique et, par conséquent, le temps de réponse par l'intermédiaire de l'insertion de registres supplémentaires, qui permettra de découper la longueur du chemin critique. Nous aborderons ce sujet de manière plus détaillée dans le chapitre 4.

Stratégies de synthèse comportementale

La synthèse comportementale peut se faire en suivant deux stratégies différentes [PR94, LQ92, Thi92, LB91] : (i) la synthèse par transformations successives, où le concepteur choisit des transformations qui préservent la correction de la description initiale. Il existe des outils généralistes comme *Transe* [DKL92, LD96] et des outils de mise en œuvre des algorithmes sur des architectures régulières comme *Alpha de Centaur* [LMQ91], *Presage* [VP90], etc. ; et (ii) la synthèse automatique ou synthèse par allocation-ordonnancement, qui effectue des transformations de haut niveau, telles que l'allocation et l'ordonnancement.

La synthèse par transformations sucessives consiste à transformer progressivement les spécifications comportementales, par réécritures successives et prouvées, jusqu'à obtenir une implantation matérielle qui respecte les contraintes de l'application. Le but de cette technique est d'améliorer l'efficacité de la synthèse, tout en préservant la sémantique de la spécification initiale. Le grand avantage d'un outil comme *Transe* est qu'il permet d'obtenir, par dérivations et à partir d'une spécification en langage *Lustre*, la description de son implantation matérielle en même temps que la preuve formelle de sa conformité [LD96].

Les transformations cherchent à optimiser la spécification initiale et à rendre le comportement du circuit le plus proche possible de l'optimum [Cam90]. Ces transformations peuvent s'appliquer sur le flot de contrôle ou sur le flot de données [DKMT90]. Quelques transformations ont été inspirées sur celles réalisées par les compilateurs (l'élimination de code inutile, la propagation de constantes, l'élimination de sous-expressions communes, le déroulement de boucles, l'expansion de corps de procédures, etc.) [ASU89, Thi92]. D'autres ne concernent que la synthèse comportementale, comme la transformation d'une multiplication par puissance de deux dans un décalage, la réduction du nombre de niveaux dans les graphes de contrôle et de données, l'accroissement du parallélisme, etc.

Dans notre méthodologie, nous effectuons la synthèse comportementale par l'intermédiaire des transformations spatio-temporelles (défactorisation) appliquées à la spécification algorithmique initiale. Ces transformations, réalisées par une heuristique d'optimisation, préservent la sémantique de la spécification initiale, puisqu'elles sont basées sur l'équivalence entre les opérateurs de factorisation de motifs répétitifs (*FORK*, *JOIN* et *ITERATE*) et ses correspondants défactorisés (*EXPLODE* et *IMPLODE*), comme nous avons décrit dans le chapitre 2. Nous effectuons donc une synthèse automatique par transformations successives, en mélangeant les deux stratégies présentées ci-dessus.

3.1.3 Systèmes de synthèse d'architectures régulières

Les architectures régulières autorisent des méthodes de synthèse automatisées, en exploitant la régularité de l'architecture et de l'algorithme, à la fois. Les méthodes de synthèse d'architectures systoliques reposent sur l'expression des calculs par expressions récurrentes et l'application de projections (*mapping*) sur le graphe de dépendances associé au système à résoudre [LeM97].

Les techniques de synthèse sont bien maîtrisées aux niveaux transfert de registres, logique et physique. Cependant, en fonction de la complexité croissante des applications à traiter et de l'augmentation des possibilités d'intégration, il est souhaitable que la spécification initiale de ces applications soit réalisée au niveau algorithmique ou système. Il y a quelques caractéristiques à poursuivre pendant la conception d'un système de synthèse d'architectures régulières : la synthèse par raffinement de descriptions, les réalisations physiques régulières, la conception sans défault, l'exploration de différentes implantations et l'intégration dans un environnement de CAO élargi [LeM97].

Synthèse par raffinement de description

À partir d'un programme initial représentant un algorithme, la synthèse par transformations de programmes doit permettre de déduire, après plusieurs étapes, un programme représentant une architecture fonctionnellement équivalente à la spécification initiale.

Réalisations physiques régulières

L'exploitation de la régularité existante dans les descriptions que l'on envisage d'implanter permet de simplifier la conception, de diminuer les coûts et d'améliorer la performance. Ainsi, il faut produire, à partir du langage utilisé, une réalisation physique régulière.

Conception sans défaut

Le modèle de représentation utilisé par le système, ainsi que les transformations de description, doivent être formellement bien définis, afin de permettre d'obtenir une architecture fonctionnellement équivalente à la spécification initiale. L'architecture obtenue est correcte par construction, sous l'hypothèse que la spécification initiale soit correcte. Pour s'assurer de cela, on peut effectuer des simulations ou des vérifications formelles.

Exploration de différentes implantations

Un outil de synthèse semi-automatique doit permettre au concepteur d'explorer différentes solutions d'implantation. Naturellement, les choix effectués à haut niveau peuvent impliquer des différences très importantes pour l'implantation finale. Il est indispensable d'évaluer le coût des implantations, de prendre en compte certaines contraintes et de mettre en place des stratégies de synthèse en fonction des différents critères à optimiser (temps de exécution, surface, consommation).

Intégration dans un environnement de CAO élargi

Le but d'un système de synthèse est de permettre la simulation, la synthèse structurelle et la réalisation physique des architectures dérivées. Ainsi, afin de cibler le maximum d'outils de CAO existants, il est préférable de générer des descriptions dans un langage standardisé comme VHDL, Verilog ou EDIF.

Les architectures-cible de notre extension de la méthodologie AAA, c'est-à-dire les architectures basées sur des circuits reconfigurables du type FPGA, sont caractérisées par une grande régularité. En plus, les applications-cible sont les applications embarquées temps réel pour le contrôle-commande et le traitement du signal et des images, elles aussi caractérisées par une régularité importante. Ces deux facteurs nous ont motivé à poursuivre les cinq caractéristiques de la synthèse d'architectures régulières, présentées ci-dessus.

L'implantation par défactorisation de la spécification algorithmique initiale permet d'effectuer une *synthèse par raffinement de description*, où le graphe matériel résultant est strictement équivalent au graphe algorithmique initial (ou au programme en langage synchrone, comme nous avons dit dans les sections 2.1.3) et 2.1.4 décrivant l'application. La spécification à partir du modèle GFDD, qui représente la factorisation des parties répétitives de l'application, permet d'obtenir des *réalisations physiques régulières* après la défactorisation. Les transformations appliquées au graphe algorithmique sont triviales et leur correction est assurée par l'équivalence fonctionnelle entre les opérateurs de factorisation et ses correspondants défactorisés. La synthèse du contrôleur est réalisée par l'intermédiaire de l'utilisation de règles simples, comme nous le verrons dans la section 3.9. Cela nous assure une *conception sans défaut*. L'heuristique d'optimisation, en se basant sur un prédicteur de performances, effectue la transformation par défactorisation de la spécification initiale, permettant l'*exploration de différentes implantations*. Enfin, nous générons le code VHDL structurel synthétisable à partir du graphe matériel optimisé, ce qui nous assure l'*intégration dans un environnement de CAO élargi*.

3.1.4 Synthèse du contrôle

Normalement, la spécification de l'unité de contrôle est générée après que la synthèse du chemin de données soit complète. Pourtant, il faut signaler qu'il existe un compromis étroit entre la synthèse du contrôleur et celle du chemin de données. La longueur du chemin critique, dans un circuit RTL composé d'un chemin de données (*datapath*) et d'un chemin de contrôle (*control path*), correspond au chemin du flot de données entre deux registres, qui présente la latence la plus élévée. On y trouve trois valeurs de latence : celle associée au contrôleur, celle associée au cablage de contrôle et celle associée au chemin de données. Comme la longueur (latence) du chemin critique détermine la période de l'horloge, la réduction du chemin critique augmente la performance du circuit. Dans les systèmes numériques à hautes fréquences, le retard introduit par le cablage d'interconnexion a une grande influence sur la longueur du chemin critique, puisque dans un circuit RTL les connexions entre le contrôleur et le chemin de données se font à travers des longs conducteurs. Le chemin de contrôle commence dans les registres d'état du contrôleur, traverse sa logique de contrôle et le cablage de connexion, pour finir dans les points de contrôle du chemin de données [PA99].

Comme les contrôleurs sont fréquemment modélés sous la forme de FSM, plusieurs travaux ont été réalisés sur la décomposition de FSM, dans le but de réduire le chemin critique [ET93, DN89, ADN91]. Eppling [ET93] a réussi à réduire la longueur du chemin critique à travers la décomposition d'un contrôleur centralisé en plusieurs contrôleurs locaux. Papachristou [PA99] a proposé la génération de plusieurs contrôleurs locaux, chacun contrôlant une partition ou un bloc du chemin de données. Ainsi, au moment de la synthèse, chaque contrôleur local est placé proche de son respectif bloc fonctionnel, réduisant la longueur des connexions et, conséquemment, les retards.

Rao et Kurdahi [RK94] ont proposé une approche hiérarchique, où le surcoût de la logique de contrôle est pris en compte à chaque niveau de la hiérarchie, avant que le chemin de données soit complètement synthétisé. Cette approche s'est avérée la plus appropriée pour la synthèse d'algorithmes réguliers. Huang et Wolf [HW94] ont analysé les effets de l'allocation du chemin de données sur les retards associés au contrôleur. Ils ont proposé une approche d'allocation qui considère les effets du contrôleur sur la longueur du chemin critique. Cette approche a permis de minimiser la période de l'horloge pendant la phase d'allocation. Benmohammed et al. [Ben*98] ont proposé une méthodologie pour la génération automatique de contrôleurs reprogrammables dans l'environnement de HLS *Amical*.

Les travaux de Eppling [ET93], Rao et Kurdahi [RK94] et Papachristou [PA99] nous ont montré que la décomposition du chemin de contrôle en plusieurs contrôleurs locaux hiérarchiques permet de réduire de façon importante la longueur du chemin critique et la surface occupée par les contrôleurs. Dans notre méthodologie, l'association d'une unité de contrôle à chaque frontière de factorisation nous assure la décentralisation du chemin de contrôle et, par conséquent, sa décomposition en contrôleurs locaux. Ces contrôleurs (unités de contrôle) sont

hiérarchisés en fonction des relations de voisinage entre les frontières de factorisation (voir section 3.9).

3.2 Traduction matérielle

Pour représenter une implantation matérielle, nous utilisons des graphes d'opérateurs interconnectés (graphe matériel), dont chaque sommet représente un opérateur et chaque arc une connexion inter-opérateurs. L'étiquetage des sommets et des arcs permet de les caractériser. Chaque sommet représentant un opérateur reçoit une étiquette correspondante : nom, surface et latence de l'opérateur, et chaque arc est étiqueté avec la durée de communication entre les opérateurs. En plus de la partie chemin de données, composée des opérateurs et de ses interconnexions, déjà existante dans le graphe algorithmique, le graphe matériel contient une partie contrôle, composée d'une unité de contrôle et des signaux de contrôle, comme nous verrons par la suite (figures 3.37 et 3.40).

L'implantation matérielle sous la forme d'un schéma logique (graphe matériel G_m) est obtenue par traduction directe de la spécification algorithmique sous la forme d'un GFDD (graphe algorithmique G_a), cf. éq. 3.1. Les sommets de ce dernier sont remplacés par les opérateurs qui les implantent et ses arcs (dépendances de données) sont remplacés par une connexion physique entre les opérateurs. La synchronisation des opérateurs synchrones et le contrôle des opérateurs frontière de factorisation sont générés de façon triviale à partir de l'analyse des relations de dépendance entre les frontières de factorisation du graphe algorithmique.

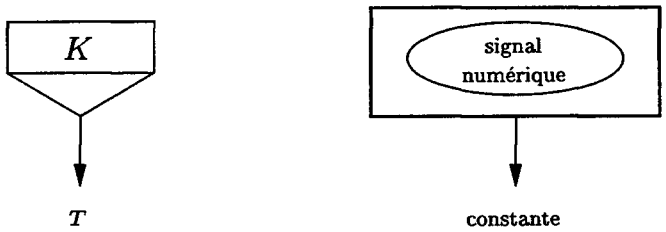
$$G_a \xrightarrow{implant.} G_m \tag{3.1}$$

3.3 Opérateurs de base

Les opérateurs de base implantent les parties combinatoires de la spécification algorithmique. L'implantation d'un graphe algorithmique totalement défactorisé ne peut contenir que des opérateurs de base. Ces opérateurs sont les correspondants matériels des sommets de base présentés dans la section 2.2.2.

Opérateur *CONSTANTE*

L'opérateur *CONSTANTE* (cf. figure 3.3), identifié par K, correspond à l'implantation matérielle du sommet *DONNÉE*. Il est utilisé, par exemple, pour fournir la valeur initiale d'un opérateur *ITERATE* (cf. 2.2.2). Cet opérateur n'a pas de prédécesseur, il marque une source du graphe matériel.

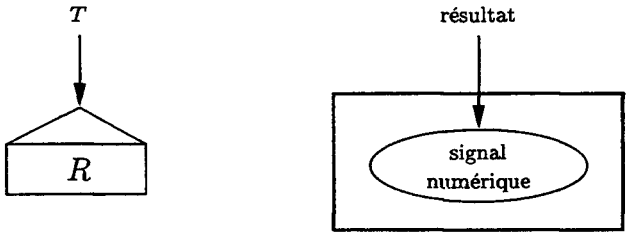


(a) symbole (b) vue interne

FIG. 3.3: Opérateur *CONSTANTE*

Opérateur *RÉSULTAT*

L'opérateur *RÉSULTAT* (cf. figure 3.4), identifié par R, correspond à l'implantation matérielle du sommet *RÉSULTAT*. Cet opérateur n'a pas de successeur, il marque un puits du graphe matériel.



(a) symbole (b) vue interne

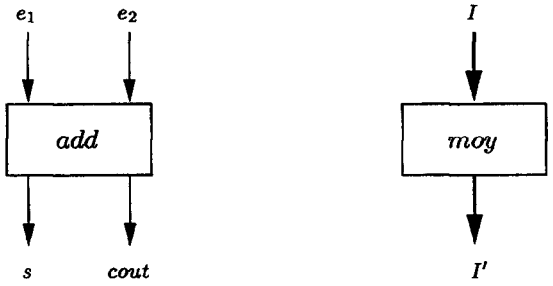
FIG. 3.4: Opérateur *RÉSULTAT*

Les graphes infinis factorisés, qui spécifient les applications que nous intéressent, ne peuvent avoir qu'une infinité de résultats, factorisés par des opérateurs $JOIN^{\infty}$; donc, ils ne peuvent pas avoir de sommets *RÉSULTAT*.

Opérateur *CALCUL*

L'ensemble d'opérateurs *CALCUL*, identifiés par *Cal*, correspond aux implantations matérielles des sommets *CALCUL*. Cet ensemble d'opérateurs logiques (and, or, not, exclusive-or, etc.), arithmétiques simples (additionneur, multiplieur, soustracteur, diviseur, etc.) et de calcul plus complexes (filtres, transformée de Fourier rapide) peut, soit être disponible dans une bibliothèque d'opérateurs pré-définis, soit être défini par l'utilisateur en fonction de ses besoins. Les valeurs des résultats de sortie de ces opérateurs ne dépendent que des valeurs des données en entrée. La figure 3.5 montre deux exemples d'utilisation de l'opérateur *CALCUL* : (a) le demi-additionneur de deux signaux e_1 et e_2 ; et (b) le filtre moyenneur appliqué sur une image I en produisant l'image filtrée I' en sortie. Dans un graphe représentant une architecture multi-composants qui possède différents opérateurs *CALCUL*, le

mnémonique *Cal* est remplacé par le mnémonique correspondant à l'opérateur de calcul utilisé (voir tableau 2.1).

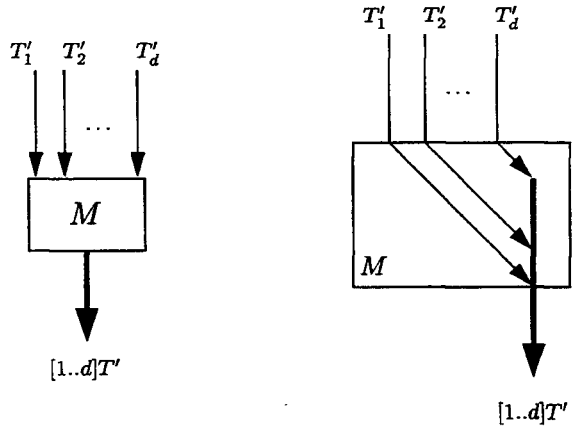


(a) demi-additionneur (b) filtre moyenneur

FIG. 3.5: Exemples d'opérateurs *CALCUL*

Opérateur *IMPLODE*

L'opérateur *IMPLODE*, identifié par M, correspond à l'implantation matérielle du sommet *IMPLODE*. Il réalise un regroupement ordonné de d bus unidirectionnels (le nombre de conducteurs reste le même), comme le montre la figure 3.6. Il implante des connexions directes entre des opérateurs en amont et l'opérateur en aval. L'opérateur *IMPLODE* ne réalise qu'une transformation de typage.



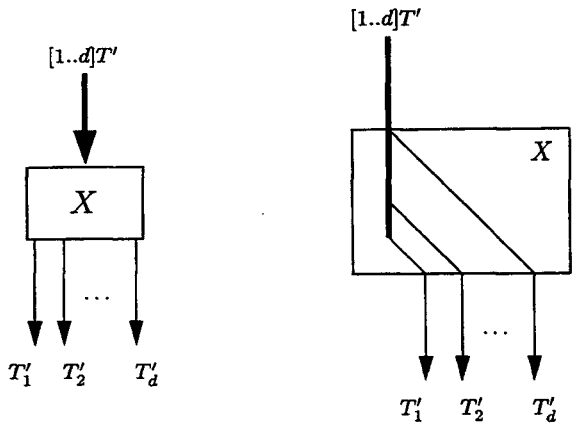
(a) symbole (b) vue interne

FIG. 3.6: Opérateur *IMPLODE*

Opérateur *EXPLODE*

L'opérateur *EXPLODE*, identifié par X, correspond à l'implantation matérielle du sommet *EXPLODE*. Il réalise une décomposition d'un bus unidirectionnel en d sous-bus (le nombre de conducteurs reste le même), comme le montre la figure 3.7. Il

implante des connexions directes entre l'opérateur en amont et des opérateurs en aval. L'opérateur *EXPLODE* ne réalise qu'une transformation de typage. Il implante l'opération inverse de l'*IMPLODE*.



(a) symbole (b) vue interne

FIG. 3.7: Opérateur *EXPLODE*

3.4 Opérateurs de factorisation de motifs de graphes répétitifs finis

La factorisation n'a pas pour seul intérêt que la réduction de la taille des spécifications : elle permet aussi de réduire, dans les mêmes proportions, la taille des architectures. Une opération à l'intérieur d'une frontière de factorisation, FF, qui représente un groupe factorisé d'opérations identiques opérant sur un groupe factorisé de données différentes, se réalise directement avec un seul opérateur utilisé itérativement, autant de fois qu'il y a d'opérations dans le groupe factorisé. Chaque groupe factorisé de données doit être donc multiplexé/demultiplexé à la frontière : alors qu'ils ne jouent qu'un rôle "synthatique" au niveau du graphe de dépendances, les opérateurs correspondant aux sommets frontières doivent réaliser le multiplexage/demultiplexage, sauf l'opérateur *DIFFUSION*, qui se contente de fournir la même valeur à chaque itération [LS97].

Tous les opérateurs frontières de factorisation de motifs répétitifs d'une même frontière partagent une même propriété : la factorisation de d répétitions d'un motif répétitif. Cette propriété est concrétisée au niveau de l'implantation matérielle par l'association d'un compteur à chaque frontière de factorisation.

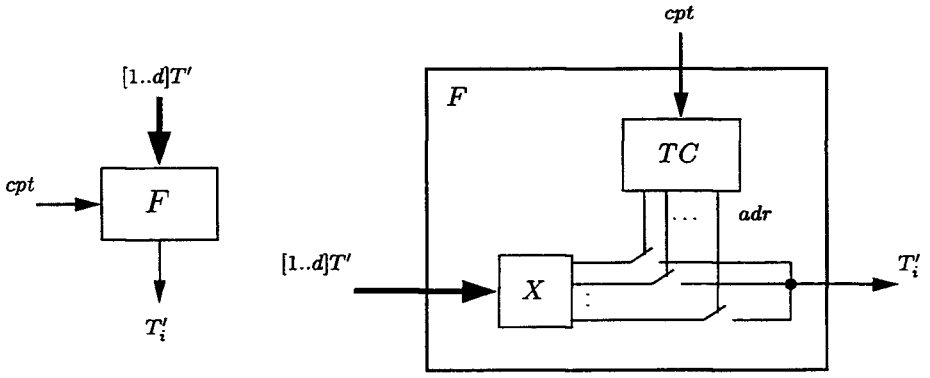
Comme tout multiplexage/demultiplexage implique un séquencement temporel, les opérateurs F, J et I font appel à des compteurs périodiques pour séquencer leurs données d'entrée et/ou de sortie. Nous utilisons un seul compteur par frontière de factorisation, dans le cas où le motif factorisé est implanté sur un seul composant et un compteur par frontière par composant, dans le cas où le motif factorisé est

distribué sur plusieurs composants, car la communication (entre composants) du signal d'horloge consommera moins de ressources d'interconnexion (un bit) que celle de l'état du compteur (d ou $log_2 d$ bits, en fonction du type de compteur implanté, où d correspond au nombre de répétitions du motif factorisé).

Ci-dessous, nous présentons les définitions et les descriptions des structures internes des opérateurs de factorisation des motifs répétitifs finis (*FORK*, *JOIN*, *ITERATE* et *DIFFUSION*).

Opérateur *FORK*

L'opérateur *FORK*, identifié par F, implante la factorisation d'un flot de données sous la forme d'un vecteur $[1..d]T'$ en son entrée, en énumérant les éléments T'_i, où $i \in \{1,\ldots,d\}$, en sortie, comme le montre la figure 3.8. Les d sous-graphes (identiques, motif répétitif) en aval de chacun des d sorties de l'*EXPLODE* sont également factorisés en un seul sous-graphe en aval de l'unique sortie du *FORK*. C'est le correspondant factorisé de l'*EXPLODE*. L'opérateur *FORK* (voir figure 3.8) est composé de :



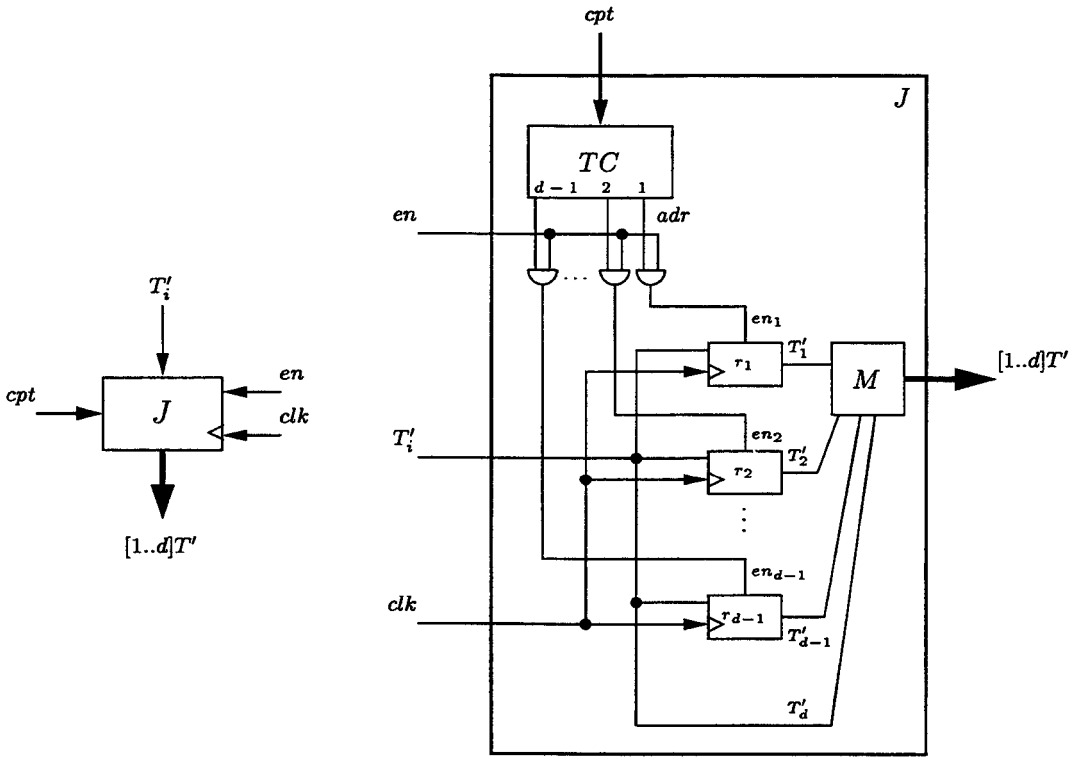
(a) symbole (b) vue interne

FIG. 3.8: Opérateur *FORK*

- un transcodeur, identifié par TC, générant, à partir de la séquence de comptage (*cpt*), une séquence d'adresses (*adr*) pour le multiplexeur. Le transcodeur est neutre, si et seulement si l'ordre initial de l'ensemble de d signaux est préservé en sortie de *FORK* ;
- un *EXPLODE*, identifié par X. Il reçoit en entrée le signal $[1..d]T'$, vecteur composé des d signaux T'_i et il sépare ces signaux en sortie ;
- un multiplexeur, formé par d interrrupteurs sélectionnés par l'adresse (*add*) générée par le transcodeur. Il fournit en sortie le signal T'_i sélectionné.

Opérateur *JOIN*

L'opérateur *JOIN*, identifié par J, implante la factorisation des d flots de données T'_i, où $i \in \{1, \ldots, d\}$, en son entrée, en les collectant sous la forme d'un vecteur $[1..d]T'$, comme le montre la figure 3.9. Les d sous-graphes (identiques, motif répétitif) en amont de chacune des d entrées de l'*IMPLODE* sont également factorisés en un seul sous-graphe en amont de l'unique entrée du *JOIN*. Il implante l'opération inverse de l'opérateur *FORK*. C'est le correspondant factorisé de l'*IMPLODE*. L'opérateur *JOIN* (voir figure 3.9) est composé de :



(a) symbole (b) vue interne

FIG. 3.9: Opérateur *JOIN*

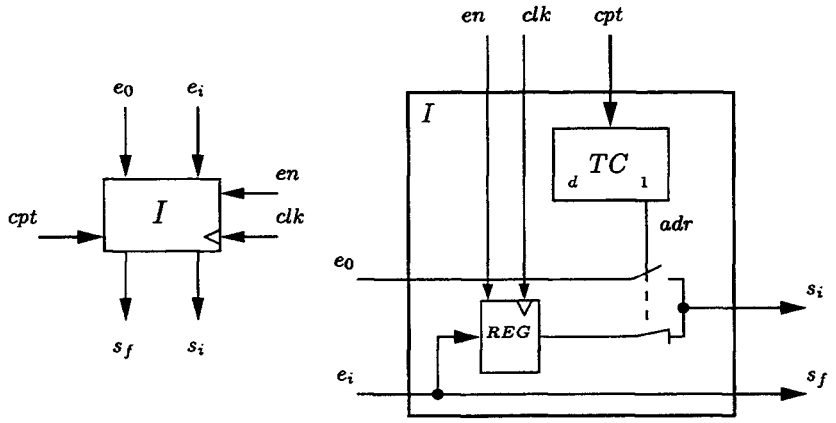
- un transcodeur, identifiée par TC, comme décrit précédemment ;
- un banc de $(d-1)$ registres, identifiés par r_1, r_2, ..., r_{d-1}. Ces registres fournissent en sortie les signaux $[1..(d-1)]T'$, qui représentent un vecteur à $(d-1)$ éléments. Seulement un registre est validé (en_i) à chaque fois par le signal d'adresse (adr) généré par le transcodeur et par le signal de validation en, qui détermine l'instant de transition des registres appartenant à une même frontière de factorisation ;
- un démultiplexeur : un signal T'_i est aiguillé sur l'un des $(d-1)$ registres, sélectionné par le signal adr, pour i variant entre 1 et $(d-1)$. Tous les registres possèdent une entrée horloge (clk) et une entrée de validation (en). Le signal

clk est simultané pour toutes les bascules. La sortie correspondant à T'_d est connectée directement au bus de sortie pour composer le signal $[1..d]T'$, qui représente un vecteur à d éléments ;

- un *IMPLODE*, identifié par M, qui regroupe les d signaux T'_i sous la forme d'un vecteur $[1..d]T'$ en sortie.

Opérateur *ITERATE*

L'opérateur *ITERATE*, identifié par I, implante la factorisation des dépendances de données inter-motifs. Lorsque d sous-graphes SG sont factorisés, chaque dépendance de données inter-motif (entre deux sous-graphes adjacents) apparaît en sortie et en entrée du sous-graphe factorisé. L'opérateur *ITERATE* permet d'implanter une connexion inter-motifs, qui apparaît dans le graphe factorisé du motif comme un cycle à travers un sommet *ITERATE*. La figure 3.10 représente l'opérateur *ITERATE* :



(a) symbole (b) vue interne

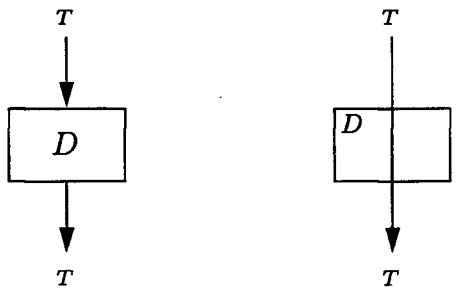
FIG. 3.10: Opérateur *ITERATE*

À son l'état initial, le compteur C sélectionne la valeur initiale e_0. Dans l'état "final" $(d-1)$, la valeur de sortie s_f est celle désirée. L'opérateur *ITERATE* (voir figure 3.10) est composé de :

- un registre, identifié par *REG*. Il mémorise la valeur de l'itération précédente, quand le signal de validation *en* est actif ;
- un transcodeur, identifié par TC, comme décrit précédemment ;
- un multiplexeur formé par deux interrupteurs contrôlés par le bit le moins significatif de l'adresse *adr* (un interrupteur normalement ouvert laisse passer la valeur initial e_0 en sortie, si adr_0 est actif ; sinon, un interrupteur normalement fermé laisse passer la valeur stockée dans *REG* en sortie). Ainsi, il aiguille la valeur d'initialisation (e_0), si le compteur est dans son état "initial", sinon, il aiguille la sortie du registre *REG*.

Opérateur *DIFFUSION*

L'opérateur *DIFFUSION*, identifié par D, implante la factorisation des entrées d'un motif répétitif, provenant toutes d'une même opération externe au motif. Il ne sert qu'à marquer la frontière de factorisation. Il implante des connexions directes entre son entrée et sa sortie, comme le montre la figure 2.12.



(a) symbole (b) vue interne

FIG. 3.11: Opérateur *DIFFUSION*

On remarque que l'opérateur *DIFFUSION* ne reçoit pas le signal valeur de comptage (*cpt*), parce qu'il fournit en sortie la valeur de son entrée pendant un cycle complet du compteur associé à sa frontière de factorisation. Il n'y a pas de séquencement temporel.

3.5 Opérateurs de factorisation de motifs de graphes répétitifs infinis

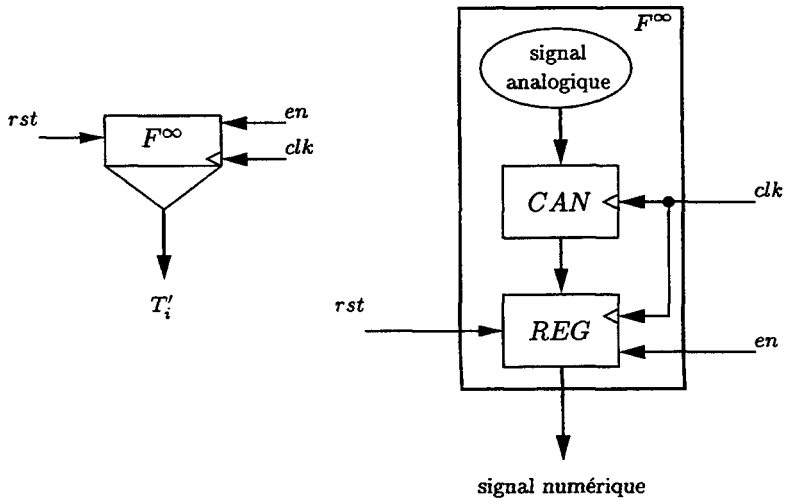
Les opérateurs de factorisation de motifs de GFDD représentent les capteurs ($FORK^{\infty}$) et les actionneurs ($JOIN^{\infty}$) qui assurent l'interface avec l'environnement, ainsi que les retards ($ITERATE^{\infty}$) et/ou les constantes des graphes "flots de données" ($DIFFUSION^{\infty}$).

Les ressources matérielles étant, naturellement, bornées, on est obligé d'avoir des implantations finies pour ces opérateurs infinis, en utilisant le multiplexage/démultiplexage temporel. C'est-à-dire qu'on va rester dans le cas factorisé et qu'on ne va pas appliquer des transformations spatio-temporelles, puisqu'il n'y a pas de parallélisme à mettre en évidence.

Opérateur $FORK^{\infty}$: capteur

L'opérateur $FORK^{\infty}$, identifié par F^{∞}, implante l'énumération d'un flot de données infini $[1..\infty]T'$ en entrée. Il correspond à un capteur qui convertit un signal physique en un signal numérique. Nous considérons que ce capteur intègre tous les

circuits nécessaires pour la conversion et l'échantillonnage des signaux (le convertisseur analogique-numérique, CAN, et le registre, REG), comme le montre la figure 3.12. Le contrôle de l'opérateur $FORK^{\infty}$ est assuré par les signaux d'horloge (clk), de validation (en) et de rémise à zéro (rst). Le signal en est généré par les frontières en aval à celle de l'opérateur $FORK^{\infty}$.



(a) symbole (b) vue interne

FIG. 3.12: Opérateur $FORK^{\infty}$

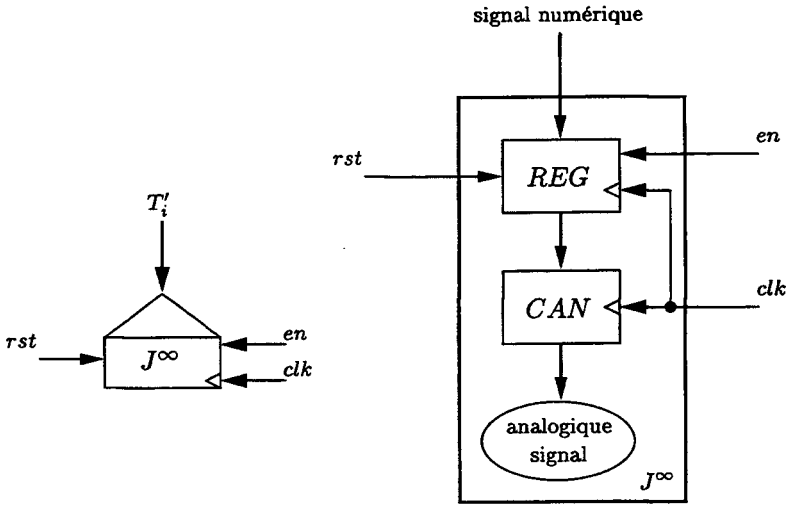
Opérateur $JOIN^{\infty}$: actionneur

L'opérateur $JOIN^{\infty}$, identifié par J^{∞}, implante la collecte des infinis flots de données finis T_i', où $i \in \{1, \ldots, \infty\}$, en son entrée. Il correspond à un actionneur qui convertit un signal numérique en analogique. Nous considérons que cet actionneur intègre tous les circuits nécessaires pour la mémorisation et la conversion des signaux (le registre REG, et le convertisseur numérique-analogique, CNA), comme le montre la figure 3.13. Le contrôle de l'opérateur $JOIN^{\infty}$ est assuré par les signaux d'horloge (clk), de validation (en) et de rémise à zéro (rst). Le signal en est généré par les frontières en amont à celle de l'opérateur $JOIN^{\infty}$.

Opérateur $ITERATE^{\infty}$: retard

L'opérateur $ITERATE^{\infty}$ correspond à un registre REG, comme le montre la figure 3.14. REG maintient en sortie la valeur qu'il avait en entrée à la fin du cycle d'horloge précédent, pendant toute la durée du cycle d'horloge. Pendant que le signal d'initialisation (rst) reste actif, le registre est forcé à la valeur e_0. On peut observer que, par rapport à l'implantation de l'opérateur $ITERATE$ fini (cf. 3.4), il n'y a pas de sortie s_f, puisque dans le cas d'un graphe infini, il n'y a pas d'itération finale. Le contrôle de l'opérateur $ITERATE^{\infty}$ est assuré par les signaux d'horloge (clk) et de

validation (*en*). Le signal *en* est généré par les frontières en amont et en aval à celle de l'opérateur $ITERATE^{\infty}$.



(a) symbole (b) vue interne

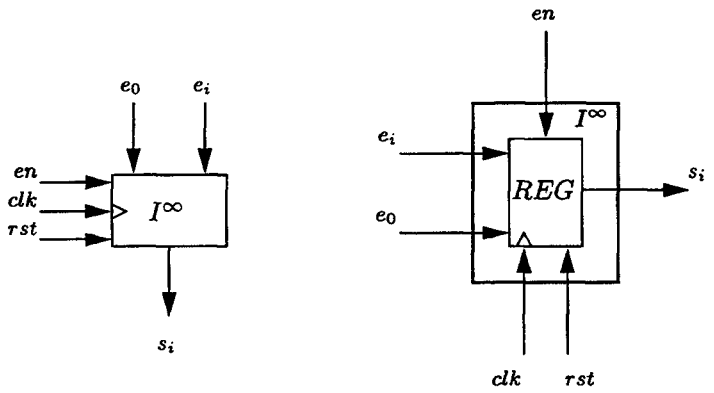
FIG. 3.13: Opérateur $JOIN^{\infty}$

Opérateur $DIFFUSION^{\infty}$: constante

L'implantation de l'opérateur $DIFFUSION^{\infty}$ est identique à celle de l'opérateur *DIFFUSION* dans le cas fini (cf. section 3.4). Il représente l'utilisation répétitive de constantes dans les graphes flots de données.

3.6 Séquencement temporel des opérateurs de factorisation

Les compteurs doivent assurer la synchronisation des opérateurs appartenant à une même frontière de factorisation. Ainsi, à chaque frontière de factorisation de motifs de graphes répétitifs finis, nous associons un compteur. Chaque compteur génère une valeur de comptage (*cpt*), qui sera diffusée aux transcodeurs des opérateurs frontières de factorisation F, J et I de sa frontière respective. Le compteur génère aussi un signal de fin de comptage (fin), en fonction du *modulo* du compteur (d) et de l'état du compteur (*cpt*). Ce signal fin est transmis aux frontières dépendantes en aval de la frontière du compteur, afin de leur signaler que les données concernant "sa frontière" sont disponibles pour être consommées. En plus du signal d'horloge (*clk*), commun à tous les registres et compteurs du circuit, chaque compteur est contrôlé par les signaux d'initialisation (*init*) et de validation (*en*). Ces signaux sont générés par les frontières de factorisation anti-dépendantes en amont et en aval de la frontière du compteur.



(a) symbole (b) vue interne

FIG. 3.14: Opérateur $ITERATE^\infty$

Comme les circuits synchrones basés sur des FPGA *Xilinx* ne doivent surtout pas avoir d'horloges pilotées (*gated clocks*) ou des horloges multiplexées [Xil94], il faut fournir une horloge commune (*clk*) à toutes les bascules et utiliser les signaux de fin de comptage générés par les compteurs, ainsi que les dépendances de données entre les frontières, pour déterminer les équations des entrées d'initialisation et de validation des compteurs (*init* et *en*).

La figure 3.15 représente le composant C, un compteur *modulo* d, où d correspond au nombre de répétitions du motif, cadencé par une horloge commune *clk* et générant un signal de fin de comptage fin, dans un rapport à l'horloge d'entrée *clk* égal à la période du compteur (d). L'entrée *en* permet d'incrémenter le compteur. L'entrée *init* permet d'initialiser le compteur.

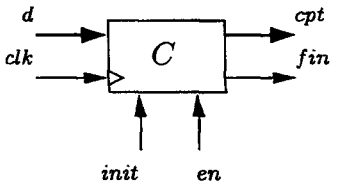


FIG. 3.15: Composant *COMPTEUR*

Signaux d'entrée-sortie

Horloge maître ou *master clock* (*clk*) : la sortie de comptage (*cpt*), c'est-à-dire la valeur des registres internes, change d'état sur le front montant de l'entrée *clk*. La fonction de contrôle *initialisation* (*init*) permet d'initialiser le compteur et la fonction *validation* (*en*) permet d'incrémenter le compteur. Ces signaux sont actifs sur le front montant de l'horloge ;

Initialisation (*init*) : cette entrée permet d'initialiser le compteur sur un front montant de *clk*, si elle est à l'état logique "1". Le signal *init* est prioritaire sur le signal *en* ;

Validation (*en*) : l'état du compteur est incrémenté sur un front montant du signal d'horloge (*clk*), si cette entrée est à l'état logique "1" ;

Fin de comptage (fin) : tant que le compteur est à l'état $(d-1)$, où d correspond au *modulo* du compteur, la sortie fin prend la valeur logique "1" ;

Valeur de comptage (*cpt*) : cette sortie fournit l'état interne du compteur, une valeur entre 0 et $(d-1)$;

Modulo (d) : cette entrée correspond au *modulo* du compteur.

Fonctionnement du compteur

Le diagramme des états d'un compteur *modulo d* est montré par la figure 3.16. La transition d'un état à l'autre est déterminée par le signal de contrôle *en*, sauf pour la transition de n'importe quel état à état 0, qui est déterminée par le signal *init*.

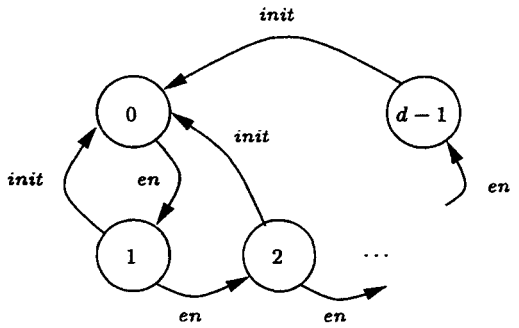


FIG. 3.16: Diagramme des états du compteur

Le tableau 3.2 représente le fonctionnement du compteur en fonction des signaux d'horloge (*clk*), d'initialisation (*init*) et de validation (*en*).

TAB. 3.2: Tableau fonctionnel du compteur

clk	*init*	*en*	*fin(t)*	*cpt(t)*
↑	1	X	0	0
↑	0	0	$fin(t-1)$	$cpt(t-1)$
↑	0	1	1, si $cpt(t) = d-1$ sinon 0	$cpt(t-1)+1$

La figure 3.17 montre le diagramme temporel d'un compteur *modulo* 6 ($d = 6$). Nous y pouvons constater que le signal d'initialisation *init* est prioritaire au signal de validation *en* et que le signal fin de comptage *fin* est actif pendant toute la durée de l'état $(d-1)$ du compteur.

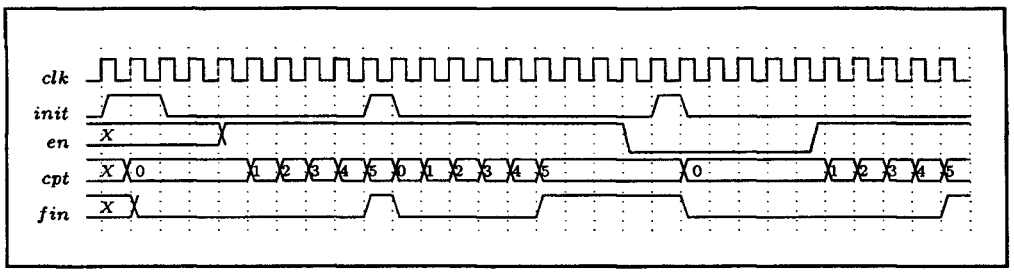


FIG. 3.17: Diagramme temporel d'un compteur *modulo* 6

Le compteur représenté par la figure 3.15 peut avoir plusieurs implantations différentes. Ci-dessous, nous discutons deux de ces implantations : le compteur *one-hot encoding* et le compteur binaire.

Compteur *one-hot encoding*

L'implantation d'un compteur *one-hot encoding*, *modulo d*, exige d registres de 1 bit en cascade. Le premier registre est initialisé dans l'état logique "1" et, à chaque front montant du signal de l'horloge *clk*, cette valeur est transmise au registre suivant, si l'entrée de validation *en* est active, si le signal d'initialisation (*init*) n'est pas actif et si le signal fin de comptage (fin) n'est pas actif. La valeur de comptage *cpt*, codée sur d bits, est fournie par les d sorties des registres. Ce compteur compte de 1 à d. La valeur fin de comptage fin correspond à l'état du d-ème registre. La valeur du *modulo d* n'est pas utilisée parce qu'elle est définie de façon implicite, correspondant au nombre de registres utilisés, comme le montre la figure 3.18. Le choix d'un compteur *one-hot encoding* élimine les transcodeurs des opérateurs F, J et I, puisqu'ils implantent la fonction identité.

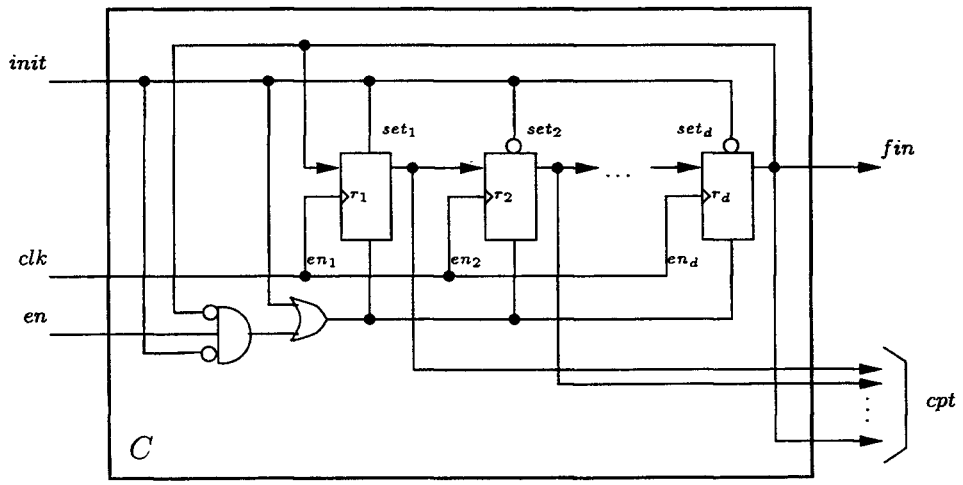


FIG. 3.18: Compteur *one-hot encoding*

Compteur binaire

L'implantation d'un compteur binaire, *modulo d*, exige $p = \lceil \log_2 d \rceil$ registres. Les registres sont initialisés dans l'état logique "0" et, à chaque front montant du signal de l'horloge *clk*, la valeur de comptage *cpt* est incrémentée si les conditions suivantes sont respectées : le signal de validation (*en*) est actif et les signaux d'initialisation (*init*) et de fin de comptage (*fin*) ne sont pas actifs. La valeur de comptage *cpt*, codée en binaire sur p bits, est fournie par les p sorties des registres. Ce compteur compte de 0 à $(d-1)$. La valeur de fin de comptage est produite par une comparaison entre la valeur $(d-1)$, codée sur p bits, et la valeur de comptage *cpt*, comme le montre la figure 3.19. Le choix d'un compteur binaire implique l'utilisation d'un transcodeur, qui convertit la valeur de comptage (*cpt*) codée sur p bits en une valeur d'adresse, codée sur d bits.

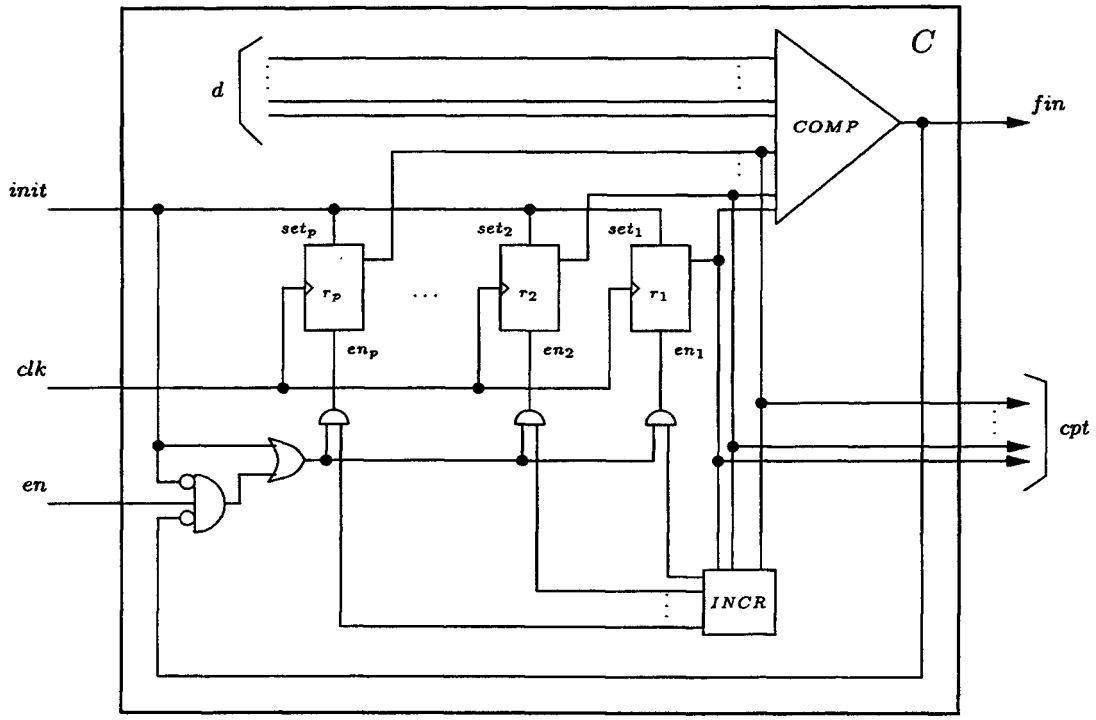


FIG. 3.19: Compteur binaire

Le tableau 3.3 montre une comparaison entre les compteurs *one-hot encoding* et binaire en ce qui concerne le nombre de cellules (registres) et le nombre de lignes de connexion nécessaires à son implantation, et la complexité de l'implantation des fonctions de comparaison, d'incrémentation et de transcodage. D'après ce tableau, on constate qu'un compteur *one-hot encoding* est le plus recommandé si la valeur du *modulo d* n'est pas très importante, parce que la surface occupée par les fonctions de comparaison, d'incrémentation et de transcodage est presque nulle, tandis que ces surfaces sont très importantes dans le cas du compteur binaire.

TAB. 3.3: Comparaison entre les compteurs

Caractéristique	*One-hot encoding*	*Binaire*
Nombre de cellules	d	$\lceil \log_2 d \rceil$
Incrémentation	trivial	complexe
Comparaison	trivial	complexe
Transcodage	trivial	complexe
Nombre de lignes	d	$\lceil \log_2 d \rceil$

3.7 Synthèse automatique de l'implantation

La synthèse automatique de circuits consiste à construire automatiquement une structure physique à partir d'une représentation comportementale de plus haut niveau [GV95] [AKJ96]. Dans le cadre de ce travail, cette représentation comportementale correspond au graphe de dépendances factorisé. La synthèse automatique réduit considérablement le cycle de développement d'un circuit et permet d'explorer les différentes implantations matérielles, afin d'obtenir un compromis surface/performance temporelle idéal pour l'application.

3.7.1 Règles de synthèse du chemin de données

L'implantation matérielle des opérations consiste à faire correspondre un opérateur à chaque sommet du graphe factorisé d'opérations (il s'agit du graphe algorithmique transformé, obtenu après l'optimisation par défactorisation, comme nous verrons dans le chapitre 4). Cet opérateur est combinatoire pour un sommet opération, ou il est composé d'un multiplexeur et/ou de registres pour un sommet frontière. L'implantation matérielle des dépendances de données entre opérations consiste à faire correspondre, à chaque arc du graphe, une connexion physique entre les opérateurs correspondant aux opérations. Les opérateurs et leurs interconnexions constituent le chemin de données du circuit.

3.7.2 Règles de synthèse du chemin de contrôle

Le chemin de contrôle correspond à ce qu'il faut ajouter au chemin de données pour contrôler les multiplexeurs et les transitions des registres des opérateurs frontières. Il est obtenu en synchronisant les transferts entre registres. Dans le cas d'un circuit composé d'un registre d'entrée Reg_E, d'un opérateur combinatoire pur X et d'un registre de sortie Reg_S, comme le montre la figure 3.20, les registres Reg_E et Reg_S sont pilotés par un signal d'horloge clk et ils transitent à chaque front montant du signal clk. Au premier front montant clk, les données T_E seront stockées par Reg_E. Ces données vont se propager à travers l'opérateur combinatoire et le résultat sera stocké par Reg_S lors du prochain front montant de clk. Ainsi, la période de l'horloge

doit être supérieure à la latence de l'opérateur combinatoire X, pour que le circuit produise des résultats corrects.

Si le circuit X comporte de parties qui se décomposent en sous-parties identiques, répétées régulièrement, ces dernières peuvent être factorisées en une seule sous-partie, entourée d'opérateurs frontière composés de multiplexeurs et de registres, contrôlant l'utilisation itérative de la sous-partie. Ces nouveaux registres doivent donc transiter plusieurs fois à chaque transition des registres Reg_E et Reg_S, lesquels ne devront donc plus transiter à chaque cycle d'horloge, mais seulement à tous les d cycles d'horloge, si la sous-partie est itérée d fois. Il faut rappeler que deux conditions sont nécessaires pour qu'un registre puisse transiter : les nouvelles données en son amont doivent être stables, et ses consommateurs en aval doivent avoir fini de consommer les données précédentes [MC80].

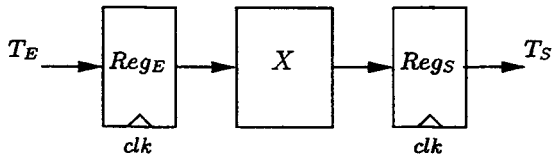


FIG. 3.20: Transferts de données entre registres

Si les données en amont d'un circuit viennent de différentes sources avec des durées de propagation différentes, il est nécessaire d'avoir un circuit synchronisé. La synchronisation d'un circuit X est possible à travers l'utilisation d'un protocole de communication de type requête/acquittement [MC80], comme le montre la figure 3.21. Les producteurs des données en amont de X (Td_1 et Td_2 sur la figure 3.21, le suffixe d signifie aval - *downstream*) indiquent la disponibilité de ces données, en provoquant une transition des signaux de requête (rd_1 et rd_2). Le circuit X utilise ces données seulement quand son entrée de requête (ru) est active. Le circuit X indique à ses producteurs en amont qu'il a fini de consommer leurs données, en activant sa sortie d'acquittement (au, le suffixe u signifie amont - *upstream*). Il en va de même symétriquement du côté aval de X.

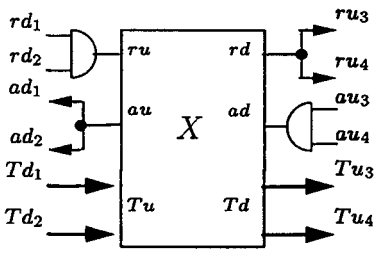


FIG. 3.21: Circuit synchronisé

Dans le cas où X est un circuit acyclique purement combinatoire, comme c'est le cas pour toutes les opérations de l'algorithme (autres que les sommets de factorisation), les entrées de requête ru et d'acquittement ad sont respectivement connectées aux sorties de requête rd et d'acquittement au. Afin de réduire le nombre

de signaux de contrôle à gérer, nous pouvons regrouper plusieurs circuits combinatoires sous la forme d'un seul circuit combinatoire équivalent, puisque la composition, par interconnexion acyclique, de plusieurs opérateurs combinatoires synchronisés est aussi un circuit combinatoire synchronisé [DD97], comme le montre la figure 3.22. On y voit que les trois additionneurs de deux entrées, connectés en cascade, peuvent être composés sous la forme d'un seul additionneur de quatre entrées, sans aucun effet sur les signaux de synchronisation.

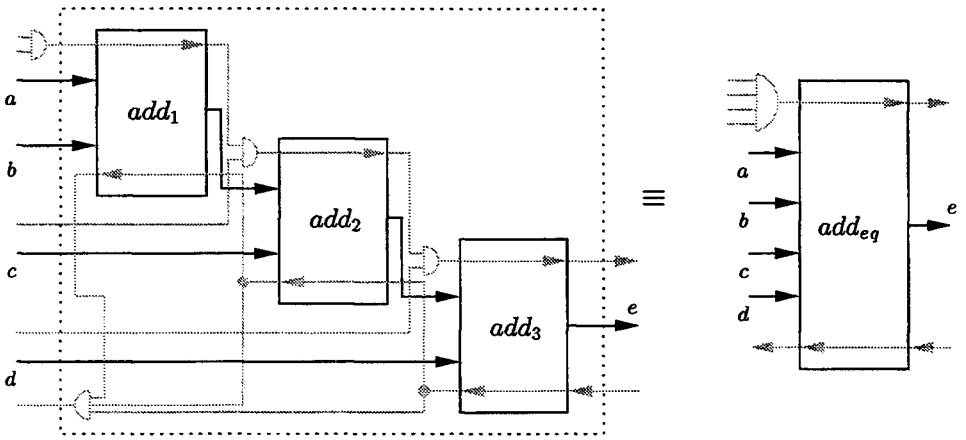


FIG. 3.22: Composition acyclique de circuits combinatoires synchronisés

3.8 Contrôle des opérateurs frontières de factorisation

Dans le chapitre 2, nous avons affirmé que les sommets frontières de factorisation de motifs de graphes répétitifs finis *FORK*, *JOIN*, *ITERATE* et *DIFFUSION*, et infinis $FORK^{\infty}$, $JOIN^{\infty}$, $ITERATE^{\infty}$ et $DIFFUSION^{\infty}$ sont, au niveau algorithmique, les délimiteurs d'une "syntaxe graphique". Ces sommets sont traduits, au niveau matériel, par des opérateurs composés de registres, multiplexeurs et/ou demultiplexeurs, comme le montrent les figures 3.8 à 3.14. La nécessité de multiplexer, démultiplexer et/ou mémoriser les signaux correspond au séquencement temporel. Ce séquencement demande l'utilisation d'une horloge pour piloter les registres associés aux compteurs et aux opérateurs de factorisation J, I, F^{∞}, J^{∞} et I^{∞} et d'une stratégie de contrôle pour coordonner les transitions d'état de ces registres.

Les dépendances de données entre les sommets appartenant à différentes frontières de factorisation d'un graphe algorithmique impliquent, au niveau matériel, des dépendances de contrôle entre les registres des opérateurs frontière de factorisation correspondants. Comme on associe, à chaque frontière, un compteur différent, les dépendances de données inter-frontières impliquent des dépendances de contrôle inter-compteurs. Le rôle du contrôle est donc d'expliciter, d'une part, les rapports entre le compteur et les transcodeurs des opérateurs de factorisation d'une même frontière de factorisation et, d'autre part, les relations entre les compteurs associés aux différentes frontières de factorisation. La partie contrôle d'une implantation

matérielle correspond aux compteurs et à la logique ajoutée pour piloter les compteurs, afin de coordonner leur opération.

Certains FPGA peuvent avoir plusieurs horloges, mais on n'en utilisera qu'une seule. Il faut donc générer les différents signaux de validation et d'initialisation (*en* et *init*) de chaque compteur et de chaque registre en fonction des dépendances de données mentionnées ci-dessus [Xil94, Har95].

3.8.1 Protocole de communication et gestion des transferts de données

Pour respecter les règles données ci-dessus, la bonne gestion des transferts de données implique un protocole de communication [Fur96, Moh96] entre producteurs et consommateurs, comme le montre la figure 3.23 :

- un signal de requête (*req*), généré par le producteur, indique quand ses résultats (T) sont disponibles (stables) en sortie,
- un signal d'acquittement (*acq*), généré par le consommateur, indique quand les données (T), qu'il reçoit en entrée, ont été utilisées et peuvent donc être modifiées par le producteur.

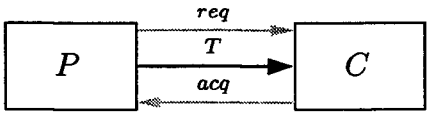


FIG. 3.23: Communication entre un producteur et un consommateur

Il existe deux types de protocoles de communication : l'un à deux phases (figure 3.24) et l'autre à quatre phases (figure 3.25). Dans le protocole à deux phases ou Non-Retour à Zéro (NRZ), les données valides (T) sont placées sur le bus par le producteur, qui indique leur disponibilité en provoquant une transition du signal de requête (*req*). Le consommateur reçoit les données T et provoque une transition du signal d'acquittement (*acq*) pour indiquer que le transfert a été accompli. Il n'y a que l'ordre de ces événements qui est important ; les délais entre eux sont arbitraires. Les fronts montant et descendant ont tous les deux la même signification. Remarquons que, comme nous sommes dans le cadre synchrone, les transitions des signaux *req*, *acq* et des données T sont pilotées par le front montant du signal d'horloge (*clk*). Le protocole à quatre phases présente un deuxième ensemble de transitions *req-acq*, afin de retourner ces signaux à leur état initial (retour à zéro). Ce protocole demande deux fois plus de temps, parce qu'il y a deux fois plus de transitions. Il reste cependant compétitif, puis que dans la plupart des cas, le temps de calcul est plus grand que le temps de communication.

Le diagramme temporel du protocole de communication à deux phases ou NRZ, dont les deux phases sont les suivantes, est montré par la figure 3.24 :

1. le signal de requête *req* fait une transition quand les données T sont stables en sortie du producteur ;
2. le consommateur génère le signal d'acquittement *acq* quand les données T ont été consommées. À ce moment-là, le producteur peut produire des nouvelles données.

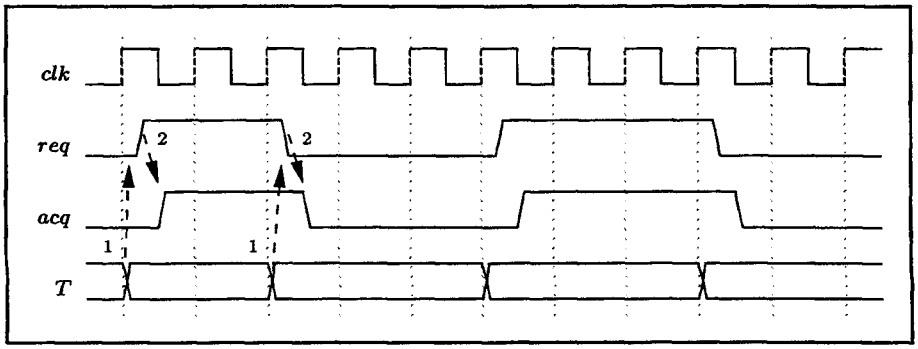


FIG. 3.24: Protocole de communication à deux phases (utilisation d'une transition)

La figure 3.25 montre le diagramme temporel d'un protocole de communication à quatre phases entre un producteur et un consommateur. Les quatre phases sont les suivantes :

1. le signal de requête *req* est produit quand les données T sont stables en sortie du producteur ;
2. le consommateur génère le signal d'acquittement *acq* quand les données T ont été consommées ;
3. quand le producteur reçoit le signal *acq*, le signal *req* est désactivé ;
4. la désactivation du signal *req* provoque celle du signal *acq*. À ce moment-là, le producteur peut produire des nouvelles données.

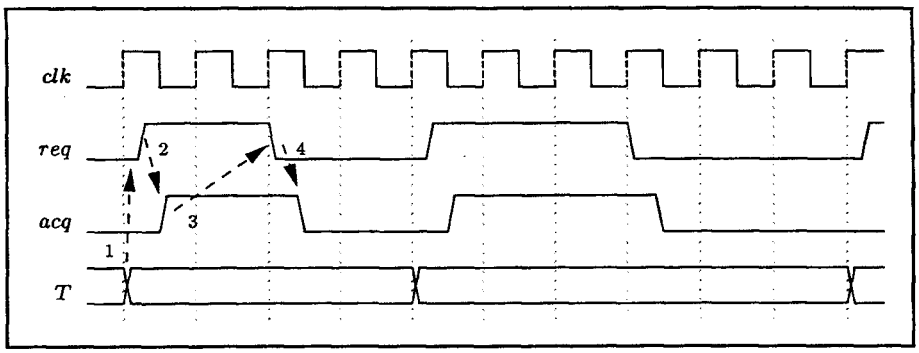


FIG. 3.25: Protocole de communication à quatre phases (utilisation d'un état)

Le protocole de communication à quatre phases est caractérisé par l'utilisation d'un état des signaux *req* et *acq* pour permettre la production de nouvelles données, au moment d'un front montant de l'horloge.

Dans notre cas, le protocole à quatre phases a été choisi pour être implanté au détriment de celui à deux phases, parce que la seconde moitié du protocole à quatre phases peut être effectuée en même temps que les calculs, ce qui permet d'améliorer sa performance. En plus, ce protocole a une implantation moins complexe, puisque seulement le front montant du signal *req* est capable d'initialiser une communication.

3.8.2 Relation de contrôle intra-frontières de factorisation

Les opérateurs de factorisation appartenant à une même frontière de factorisation opèrent de façon synchrone entre eux. Le compteur C diffuse la valeur de comptage (*cpt*) aux transcodeurs des opérateurs *FORK*, *JOIN* et *ITERATE*, afin de générer les adresses de sélection des multiplexeurs et des démultiplexeurs internes de ces opérateurs. Le signal de validation (*en*) autorise les changements d'état du compteur et les transitions des registres internes des opérateurs *JOIN* et *ITERATE* sur un front montant du signal d'horloge (*clk*), comme le montre la figure 3.26. L'opérateur D, n'ayant pas de multiplexeur, démultiplexeur ou registre internes, n'est pas en relation de contrôle avec le compteur C.

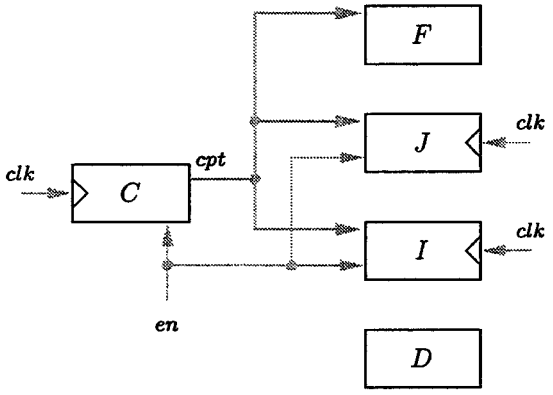


FIG. 3.26: Relation entre les opérateurs frontières "finis" et le compteur

Lorsque $cpt = (d-1)$, d correspondant au *modulo* du compteur, le signal fin de comptage (fin) est activé pendant toute la durée de l'état $(d-1)$. La génération du signal fin signifie, pour un *ITERATE*, que sa valeur finale est disponible pour être utilisée en aval et, pour un *JOIN*, que tous ses éléments sont disponibles pour être utilisés en aval. Le rôle du compteur est donc générer le signal fin de comptage (fin) pour les opérateurs J et I. Dans le cas des frontières "infinies", il n'y a évidemment pas de signal fin de comptage à générer. Ainsi, il est inutile d'associer des compteurs à ces frontières. Le contrôle des registres des opérateurs appartenant à une même frontière "infinie" (sauf pour le D^{∞}, qui n'a pas de registre interne) est alors assuré par un signal de validation (*en*), généré en fonction des rapports entre la frontière "infinie" et ses frontières adjacentes.

L'application d'un protocole de communication du type accusé-réception (*handshake*) présenté dans la section 3.8.1 à une frontière de factorisation FF, qui consomme des données T_1 produites en son amont et qui produit des données T_2 à être consommées en son aval, est montrée dans la figure 3.27.

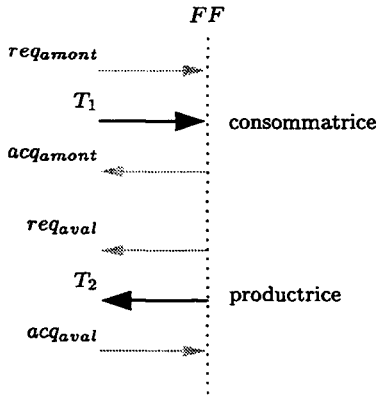


FIG. 3.27: Communication entre une frontière FF et ses frontières en amont et en aval

On remarque que FF est consommatrice par rapport aux frontières en son amont et productrice par rapport aux frontières en son aval. Ainsi, FF reçoit un signal de requête en amont de ses producteurs (req_{amont}), diffuse un signal de requête en aval à ses consommateurs (req_{aval}), reçoit un signal d'acquittement en aval de ses consommateurs (acq_{aval}) et diffuse un signal d'acquittement en amont à ses producteurs (acq_{amont}).

Le signal acq_{amont} spécifie que la frontière FF est prête à consommer des nouvelles données et que les données précédemment émises par les frontières en amont de FF ont bien été consommées. Le signal req_{amont} indique que des nouvelles données fournies par les frontières en amont de FF sont disponibles. Ainsi, tout transfert de données entre les frontières productrices en amont de FF et la frontière consommatrice FF doit être acquitté (acq_{amont}), afin que les frontières productrices puissent produire à nouveau (req_{amont}). Il en est du même pour la frontière FF, qui serait productrice vis à vis de ses frontières consommatrices en aval.

3.8.3 Relation de contrôle inter frontières de factorisation

Nous avons vu dans la section 2.4 que, en fonction des dépendances de données entre les frontières de factorisation, une frontière peut être *dépendante* (consommatrice) ou/et *anti-dépendante* (productrice) par rapport à une autre frontière. Une donnée ne peut être consommée par un opérateur d'une frontière dépendante, qu'après avoir été produite par un opérateur d'une frontière anti-dépendante. Nous avons vu aussi que les frontières ont un côté "lent" et un côté "rapide", en ce qui concerne la cadence des données qui traversent ces frontières.

Par transitivité, même les opérateurs appartenant à des frontières en relation d'anti-dépendance de données présentent des dépendances de contrôle entre elles, dû à la nécessité de synchroniser les transferts de données en amont et en aval de ces frontières de factorisation.

Les relations entre les opérateurs frontières appartenant à des frontières de factorisation différentes correspondent à des relations entre les registres de ces opérateurs. On est donc dans le cadre du problème de transfert entre registres. Le contrôle est généré à partir des dépendances de données entre registres, à travers des multiplexeurs, des démultiplexeurs et des opérateurs combinatoires.

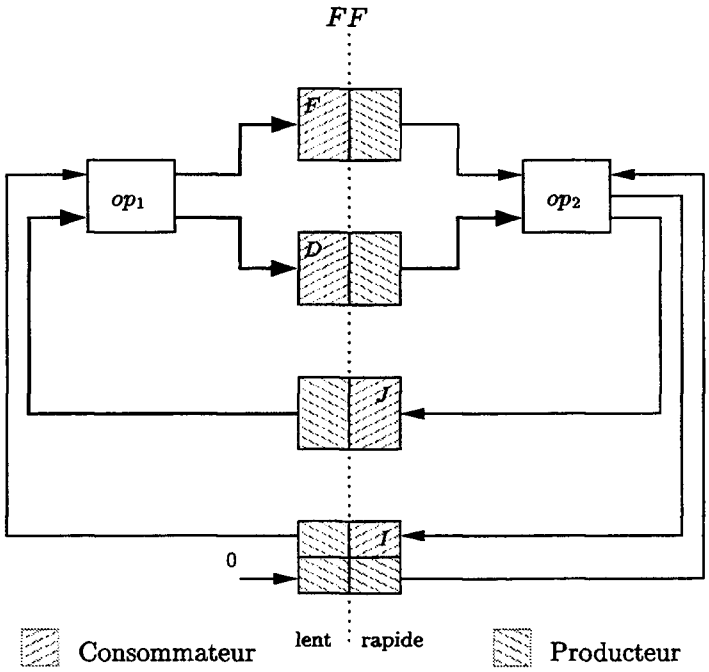


FIG. 3.28: Production/consommation d'une frontière de factorisation

Les opérateurs de factorisation *FORK* et *DIFFUSION* sont toujours consommateurs par rapport au côté "lent" et producteurs par rapport au côté "rapide" de la frontière qu'ils délimitent. L'opérateur *JOIN* est toujours consommateur par rapport au côté "rapide" et producteur par rapport au côté "lent" de la frontière qu'il délimite. L'opérateur *ITERATE*, au début du cycle de son compteur, est consommateur par rapport au côté "lent" de sa frontière et producteur par rapport au côté "rapide". À la fin du cycle, *ITERATE* est consommateur par rapport au côté "rapide" de sa frontière et producteur par rapport au côté "lent". Pour les cycles intermédiaires, *ITERATE* est, à la fois, consommateur et producteur par rapport au côté "rapide" de sa frontière de factorisation. Ce comportement de l'opérateur *ITERATE* s'explique, car il correspond à un registre, qui est producteur et consommateur à la fois. La figure 3.28 montre les relations de production et consommation des opérateurs de factorisation F, J, I et D appartenant a une frontière de factorisation FF.

Si nous tenons compte qu'il peut avoir des frontières tantôt du côté "lent", tantôt du côté "rapide" d'une frontière de factorisation FF, la figure 3.27, représentant la communication entre une frontière FF et ses frontières en amont et en aval, doit être étendue, comme le montre la figure 3.29. FF consomme la donnée T_1, produite par une frontière située de son côté "lent" ; FF produit la donnée T_2, consommée par une frontière située de son côté "rapide" ; FF consomme la donnée T_3, produite par une frontière située de son côté "rapide" ; et FF produit la donnée T_4, consommée par une frontière située de son côté "lent".

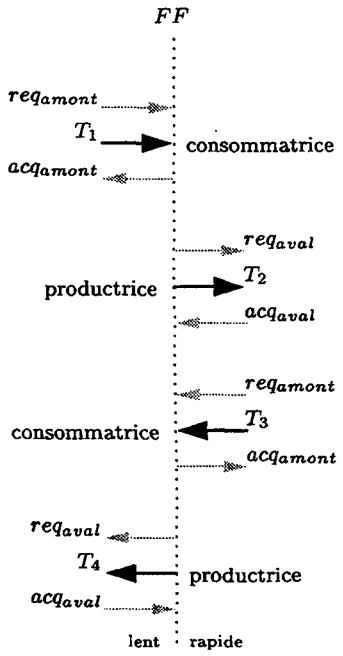


FIG. 3.29: Protocole de communication req/acq appliqué à une frontière FF

3.8.4 Graphe de relation de contrôle inter frontières de factorisation

Une frontière de factorisation FF_0 peut avoir plusieurs frontières productrices et/ou consommatrices de son côté "lent" et plusieurs frontières productrices et/ou consommatrices de son côté "rapide", comme le montre la figure 3.30. Il faut signaler que nous ne nous intéressons localement qu'aux relations entre les frontières en dépendance directe de données, puisque le graphe de voisinage entre les frontières sera construit à partir de ces relations.

À partir du graphe de dépendances de données entre la frontière FF_0 et ses frontières adjacentes (figure 3.30), on obtient le graphe de voisinage entre les frontières, montré par la figure 3.31. Par rapport à FF, nous avons :

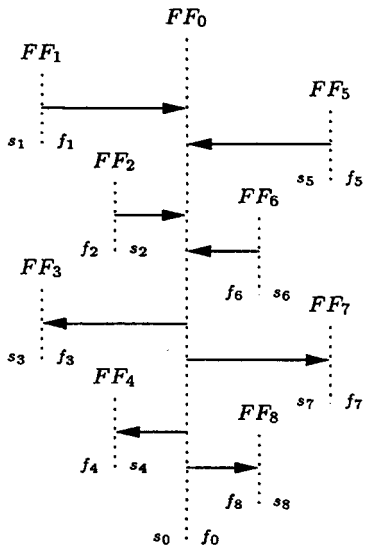


FIG. 3.30: Dépendances de données entre une frontière FF_0 et ses frontières adjacentes

- frontières productrices côté "lent" : FF_1, FF_2
- frontières consommatrices côté "lent" : FF_3, FF_4
- frontières consommatrices côté "rapide" : FF_7, FF_8
- frontières productrices côté "rapide" : FF_5, FF_6.

Les sommets du graphe de voisinage entre frontières de factorisation représentent les opérateurs frontières de factorisations qui délimitent une frontière. Les arcs orientés de ce graphe représentent les transferts de données entre les opérateurs frontières à travers des opérateurs combinatoires de calcul et/ou de communication. L'orientation de l'arc indique la relation de production/consommation des données : "producteur" $\rightarrow$ "consommateur". Comme nous verrons plus loin (section 3.9.3), ce graphe de voisinage nous permettra d'effectuer la synthèse du contrôle de l'implantation, à partir d'un graphe algorithmique.

3.9 Génération du contrôle

La génération du contrôle d'une implantation matérielle, à partir de la spécification algorithmique de l'application, est déduite des relations de production et de consommation et des relations de voisinage entre les frontières de factorisation. Il faut considérer les deux types de frontières de factorisation possibles : celles formées par des opérateurs de factorisation de motifs de graphes répétitifs finis (frontières "finies") et celles formées par des opérateurs de factorisation de motifs de graphes répétitifs infinis (frontières "infinies").

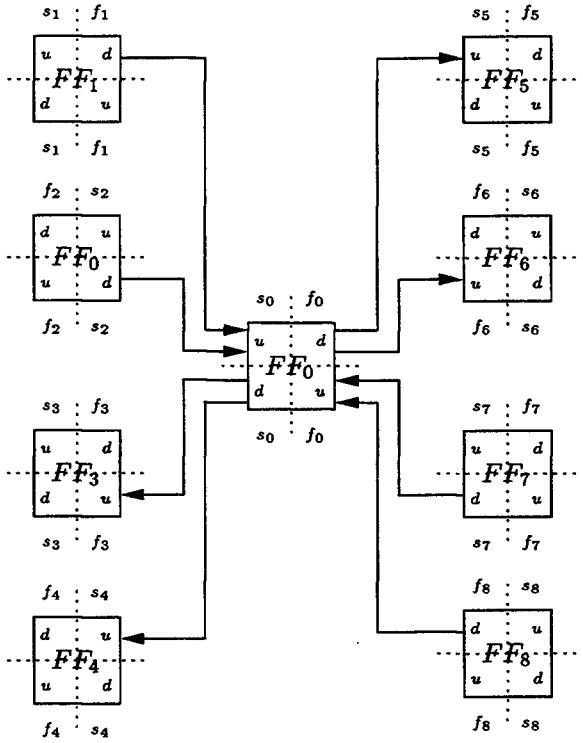


FIG. 3.31: Voisinage entre une frontière FF_0 et ses frontières adjacentes

La synthèse du contrôle d'une implantation matérielle consiste à générer, à partir des dépendances de données entre les opérateurs frontières de factorisation, les signaux de validation (*en*) et d'initialisation (*init*), qui commandent les signaux de validation des registres des compteurs. Ceux-ci commandent, par l'intermédiaire des transcodeurs, les signaux de validation des registres associés aux opérateurs frontières "finies" de factorisation (J et I), ainsi que les signaux de validation associés aux opérateurs de factorisation de motifs répétitifs infinis (F^∞, J^∞ et I^∞).

Ainsi, comme nous verrons par la suite, la génération de la partie contrôle de l'implantation matérielle correspondante à la spécification algorithmique de l'application, exprimée sous la forme d'un GFDD, est effectuée en suivant l'Algorithme 2, décrit ci-dessous. Cet algorithme de génération du contrôleur est trivial et pourra être facilement intégré à *SynDEx*, comme nous montrent les deux exemples de la section 3.10.

3.9.1 Notation

Pour simplifier la description des relations entre les signaux de gestion des frontières, nous proposons la notation suivante :

$?/!$: entrée/sortie,

s/f : *slow/fast* (lent/rapide),

u/d : *upstream/downstream* (amont/aval),

r/a : requête/acquittement,

p/c : productrice/consommatrice,

$\&/|/\neg$: AND/OR/NOT.

Algorithme 2 Génération du contrôleur

Require : Graphe algorithmique ($G_A = (S_A, A_a)$)
Assure : Contrôleur
begin
Construire le graphe de relations de voisinage entre frontières ($G_V = (S_V, A_V)$), cf. décrit par l'Algorithme 1 ;

Établir les relations de voisinage, de production et de consommation entre les frontières de factorisation, à partir du graphe de relations de voisinage (G_V) ;

Déduire les équations des signaux de contrôle, à partir du graphe de relation de voisinage (G_V), cf. décrit ci-dessous (éq. 3.2 à 3.11) ;

Interconnecter les unités de contrôle associées à chaque frontière de factorisation, en fonction des équations des signaux de contrôle.
end

3.9.2 Unité de contrôle

Dans le cas où X est un circuit séquentiel (figure 3.20), c'est-à-dire comprenant des registres, comme c'est le cas pour un circuit comprenant des parties factorisées et donc des opérateurs de factorisation, chaque frontière de factorisation a des relations amont et aval des deux côtés, lent et rapide. Les relations entre les signaux de requête et d'aquittement amont et aval des deux côtés, constituent "l'unité de contrôle" de la frontière de factorisation (figure 3.32). Cette unité de contrôle est composée d'un compteur C à d états et d'une logique supplémentaire, afin de générer le protocole de communication entre frontières (signaux de requête et d'acquittement lent et rapide, en amont et en aval), la valeur du compteur (*cpt*) et le signal de validation (*en*) qui commandent les opérateurs frontières. Le signal *init* remet le compteur à l'état 0. Le signal fin indique que le compteur est à son dernier état ($d-1$). La valeur du compteur *cpt* commande les multiplexeurs des opérateurs F, J et I de la frontière. Le signal de validation *en* détermine les cycles d'horloge où les registres des opérateurs de la frontière (F^∞, J^∞, J et I) devront transiter. Si le signal *en*, qui transite comme les autres signaux pendant un cycle d'horloge, est actif au moment du front

montant de l'horloge, alors le registre transite aussi ; sinon la sortie du registre n'est pas affectée par son entrée. Au niveau de l'interface avec l'environnement (capteurs et actionneurs), c'est l'activation du signal de validation associé aux registres internes des opérateurs F^{∞} et J^{∞}, qui permettra de lire les nouvelles données en entrée et d'écrire les résultats en sortie.

Ces signaux de gestion correspondent aux signaux de requête et d'acquittement générés par les frontières en amont ou diffusés aux frontières en aval et ils sont séparés en deux groupes : ceux qui concernent les frontières situées du côté "lent" et ceux qui concernent les frontières situées du côté "rapide", correspondant aux quatre régions de l'unité de contrôle : lent-amont, lent-aval, rapide-amont, rapide-aval.

À chaque frontière de factorisation FF, nous associons une unité de contrôle UC, dont l'interface est représentée par la figure 3.32.

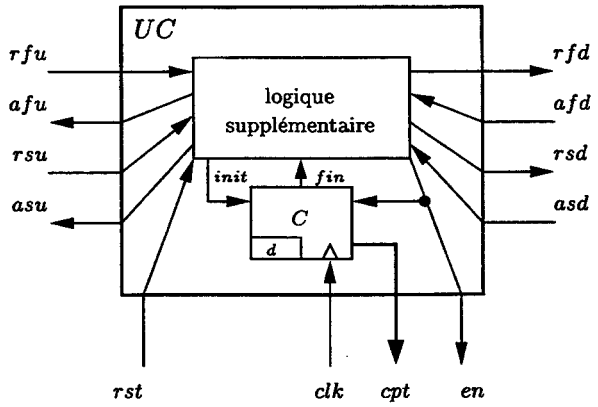


FIG. 3.32: Interface de l'unité de contrôle d'une frontière de factorisation

Les signaux d'entrée et de sortie de l'unité de contrôle sont les suivants :

- rfu : requête-rapide-en-amont (entrée),
  générée par les frontières productrices du côté "rapide" de FF,
- rfd : requête-rapide-en-aval (sortie),
  diffusée aux frontières consommatrices du côté "rapide" de FF,
- afd : acquittement-rapide-en-aval (entrée),
  généré par les frontières consommatrices du côté "rapide" de FF,
- afu : acquittement-rapide-en-amont (sortie),
  diffusé aux frontières productrices du côté "rapide" de FF,
- rsu : requête-lent-en-amont (entrée),
  générée par les frontières productrices du côté "lent" de FF,
- rsd : requête-lent-en-aval (sortie),
  diffusée aux frontières consommatrices du côté "lent" de FF,
- asd : acquittement-lent-en-aval (entrée),
  généré par les frontières consommatrices du côté "lent" de FF,

- asu : acquittement-lent-en-amont (sortie),
  diffusé aux frontières productrices du côté "lent" de FF,
- d : *modulo* du compteur (paramètre interne au compteur),
- cpt : valeur de comptage (sortie),
  diffusée aux transcodeurs des opérateurs F, J et I appartenant à FF,
- en : validation du compteur (sortie),
  diffusée aux registres des opérateurs J et I appartenant à FF,
- clk : horloge (entrée),
- rst : *reset* global (entrée).

L'unité de contrôle associée à la frontière FF (UC) est composée d'un compteur (C) et d'une logique supplémentaire pour générer les signaux de gestion de la frontière FF, comme le montre la figure 3.32. Ces signaux de gestion correspondent aux signaux de requête et d'acquittement générés par les frontières en amont de FF ou diffusés aux frontières en aval de FF :

- rsu : signal de *requête-lent-en-amont* reçu des frontières productrices situées du côté "lent" de FF, quand tous les résultats sont stables en leurs sorties. rsu est la conjonction de tous les signaux de requête en aval des frontières productrices situées du côté "lent" de FF :

$$?\,rsu = !\,rsd_{pi}\ \&\ldots\&\ !\,rfd_{pj}\ \&\ldots \tag{3.2}$$

  où, pi et pj correspondent aux indices des frontières productrices situées du côté "lent" de FF.
- rfu : signal de *requête-rapide-en amont* reçu des frontières productrices situées du côté "rapide" de FF, quand tous les résultats sont stables en leurs sorties. rfu est la conjonction de tous les signaux de requête en aval des frontières productrices situées du côté "rapide" de FF :

$$?\,rfu = !\,rsd_{pi}\ \&\ldots\&\ !\,rfd_{pj}\ \&\ldots \tag{3.3}$$

  où, pi et pj correspondent aux indices des frontières productrices situées du côté "rapide" de FF.
- asd : signal d'*acquittement-lent-en-aval* reçu des frontières consommatrices situées du côté "lent" de FF, quand ces frontières ont fini d'utiliser les données en leurs entrées. asd est la conjonction de tous les signaux d'acquittement en amont des frontières consommatrices situées du côté "lent" de FF :

$$?\,asd = !\,asu_{ci}\ \&\ldots\&\ !\,afu_{cj}\ \&\ldots \tag{3.4}$$

  où, ci et cj correspondent aux indices des frontières consommatrices situées du côté "lent" de FF.

- afd : signal d'*acquittement-rapide-en-aval* reçu des frontières consommatrices situées du côté "rapide" de FF, quand ces frontières ont fini d'utiliser les données en leurs entrées. afd est la conjonction de tous les signaux d'acquittement en amont des frontières consommatrices situées du côté "rapide" de FF :

$$?\,afd = !\,asu_{ci}\ \&\ \ldots\ \&\ !\,afu_{cj}\ \&\ \ldots \tag{3.5}$$

où, ci et cj correspondent aux indices des frontières consommatrices situées du côté "rapide" de FF.

- rfd : signal de *requête-rapide-en-aval* diffusé aux frontières consommatrices situées du côté "rapide" de FF, quand les données sont stables dans les entrées du côté "lent" de FF :

$$!\,rfd = ?\,rsu \tag{3.6}$$

- rsd : signal de *requête-lent-en-aval* diffusé aux frontières consommatrices situées du côté "lent" de FF, quand les résultats sont stables en sortie des frontières productrices situées du côté "rapide" de FF et la frontière FF a fini de consommer les données en ses entrées :

$$!\,rsd = ?\,rfu\ \&\ fin \tag{3.7}$$

- afu : signal d'*acquittement-rapide-en-amont* diffusé aux frontières productrices situées du côté "rapide" de FF, quand les frontières consommatrices du côté "lent" de FF ont fini d'utiliser les données en leurs entrées ou; quand les données sont stables en sortie des frontières productrices situées du côté "lent" de FF, les frontières consommatrices situées du côté "rapide" de FF ont fini d'utiliser les données en leurs entrées et la frontière FF n'a pas fini de consommer les données en ses entrées :

$$!\,afu = ?\,asd \mid (?\,rsu\ \&\ ?\,afd\ \&\ \neg fin) \tag{3.8}$$

- asu : signal d'*acquittement-lent-en-amont* diffusé aux frontières productrices situées du côté "lent" de FF, quand les frontières consommatrices situées du côté "rapide" de FF ont fini d'utiliser les données en leurs entrées et la frontière FF a fini de consommer les données en ses entrées :

$$!\,asu = ?\,afd\ \&\ fin \tag{3.9}$$

- en : signal de *validation* du compteur utilisé pour autoriser les changements d'état des registres des opérateurs J et I appartenant à la frontière FF. Il est actif quand les frontières consommatrices situées du côté "rapide" de FF ont

fini d'utiliser les données en leurs entrées, les résultats sont stables en sortie des frontières productrices situées du côté "lent" de FF et, soit les frontières consommatrices situées du côté "lent" de FF ont fini d'utiliser les données en leurs entrées, soit la frontière FF n'a pas fini de consommer les données en ses entrées :

$$!\,en = ?\,afd \;\&\; ?\,rsu \;\&\; (\neg fin \mid ?\,asd) \tag{3.10}$$

- *init* : signal d'*initialisation* du compteur ; il est actif, soit quand le signal de rémise à zéro global (rst) est actif, soit quand les frontières consommatrices situées du côté "rapide" de FF, les frontières consommatrices situées du côté "lent" de FF et la frontière FF ont fini de consommer les données en leurs entrées :

$$init = ?\,rst \mid (?\,afd \;\&\; ?\,asd \;\&\; fin). \tag{3.11}$$

La figure 3.33 présente, de façon détaillée, l'unité de contrôle d'une frontière "finie" de factorisation (UC), conforme aux éq. 3.3 à 3.11.

Dans le cas des frontières dites "infinies" (FFI), représentant l'interface avec l'environnement, l'unité de contrôle devient un sous-ensemble de celle montrée par la figure 3.33. Cela est dû à l'inexistence de compteurs associés à ces frontières. La supression du compteur implique la suppression des signaux d'initialisation ($init$), d'horloge (clk) et de remise à zéro (rst). La sortie valeur de comptage (cpt) a été également suprimée. Cette unité de contrôle simplifiée est représentée par la figure 3.34. Le signal de validation est utilisé pour autoriser l'écriture et/ou la lecture des registres des opérateurs frontières de factorisation de motifs de graphes répétitifs infinis. Il n'y a pas d'autres frontières du côté "lent" d'une frontière "infinie", car ce côté "lent" représente le monde réel (l'environnement). Les entrées rsu et asd sont toujours actives ($rsu = 1$ et $asd = 1$), parce que les données sont produites et consommées en continu par l'environnement. Les sorties asu et rsd ne sont jamais connectées, puisqu'il n'y a pas de frontières du côté "lent" d'une frontière"infini".

Le signal de *requête-rapide-en-amont* (rfu) est généré selon l'éq. 3.3. Le signal de *requête-rapide-en-aval* (rfd), diffusé aux frontières consommatrices situées du côté "rapide" de FFI est toujours actif, parce qu'il correspond au signal rsu qui, à son tour, est toujours actif. Le signal d'*acquittement-rapide-en-aval* (afd) est généré selon l'éq. 3.5. Le signal d'*acquittement-rapide-en-amont* (afu), diffusé aux frontières productrices situées du côté "rapide" de FF, correspond au signal d'*acquittement-lent-en-aval* (asd). Le signal de validation (en) correspond au signal d'*acquittement-rapide-en-aval* (afd).

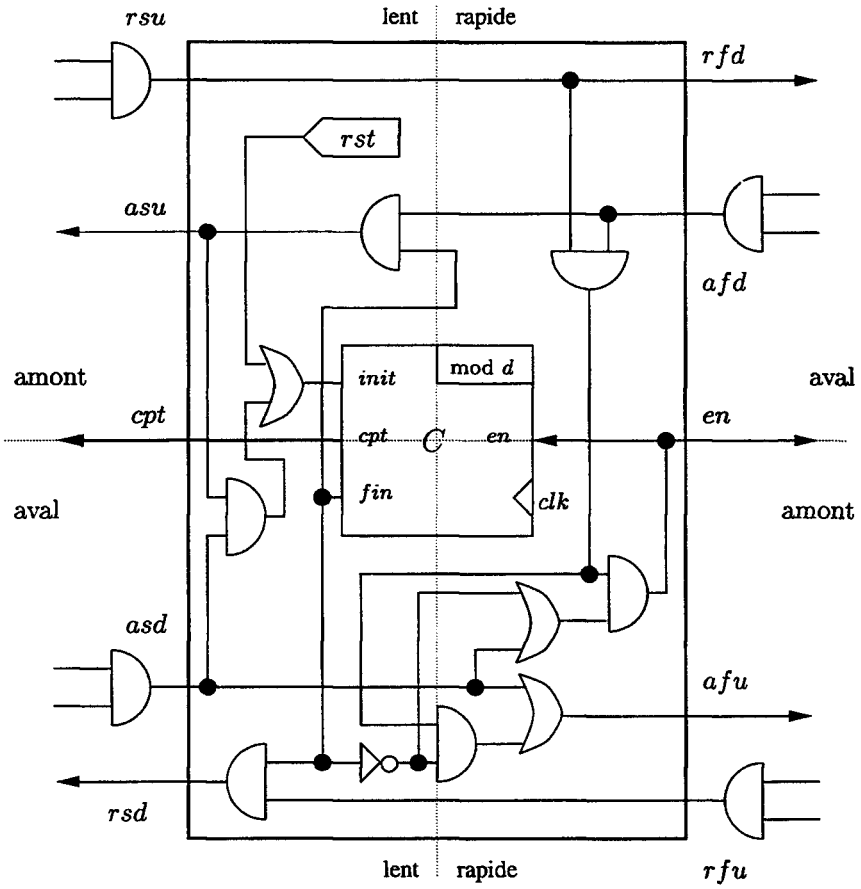


FIG. 3.33: Unité de contrôle d'une frontière "finie" de factorisation

3.9.3 Interconnexion des unités de contrôle

Les unités de contrôle peuvent être interconnectées de façon automatique, à partir du graphe de relations de voisinage entre les frontières correspondant à l'application. Dans ce graphe, les sommets correspondent aux unités de contrôle et les arcs correspondent aux signaux de requête transmis entre les unités de contrôle. Les signaux d'acquittement associés aux signaux de requête sont transmis entre les mêmes unités de contrôle, en sens inverse des signaux de requête. Quand plusieurs signaux arrivent à une même entrée d'une unité de contrôle, on en prend la conjonction par une porte logique *ET*. Dans la section 3.10, nous verrons deux exemples de synthèse des chemins de données et de contrôle.

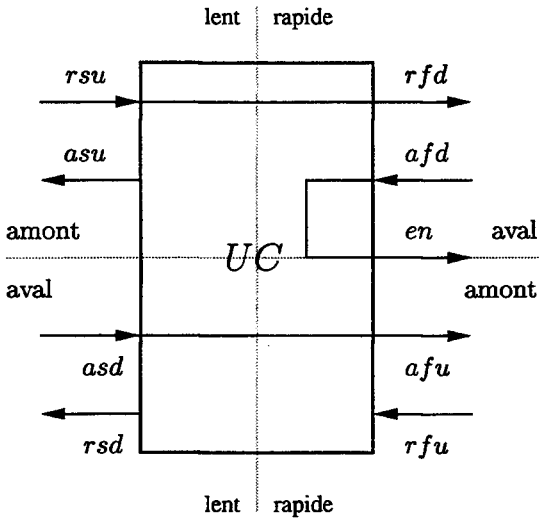


FIG. 3.34: Unité de contrôle d'une frontière "infinie" de factorisation

Exemple

Si nous appliquons les éq. 3.3 à 3.11 pour générer le contrôle d'une frontière "finie" FF_0, dont les dépendances de données avec ses frontières adjacentes sont montrées par la figure 3.30, nous obtenons les connexions de l'unité de contrôle UC_0, comme le montre la figure 3.35.

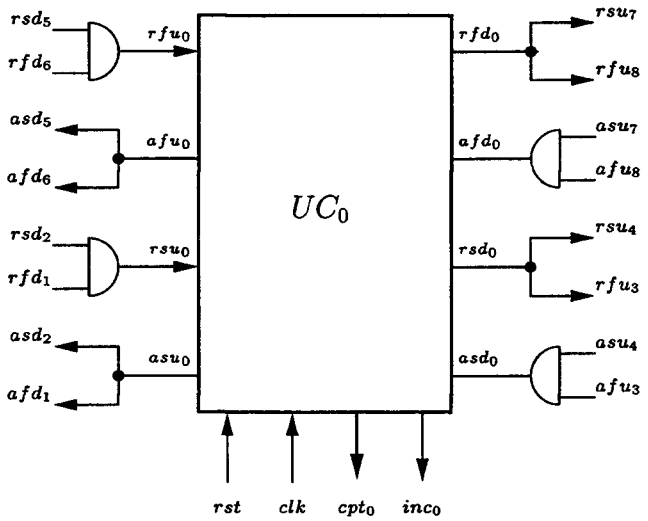


FIG. 3.35: Unité de contrôle d'une frontière "finie" FF_0

3.10 Exemples d'implantation matérielle

Pour illustrer les concepts présentés dans ce chapitre, nous reprendrons les deux exemples de spécification algorithmique montrés dans le chapitre 2 (PMV et filtrage des lignes d'une image), afin de générer les graphes matériels correspondants.

3.10.1 Produit matrice-vecteur

À partir du graphe algorithmique factorisé du PMV montré par la figure 2.26, on doit générer le graphe matériel correspondant. Ce graphe matériel contient les opérateurs équivalents aux sommets du graphe algorithmique et la partie contrôle nécessaire à la synchronisation des transferts entre les registres.

L'interconnexion des signaux de requête et d'acquittement est effectuée par l'intermédiaire de l'analyse des relations de voisinage entre les frontières de factorisation FF_1, FF_2 et FF_3 :

1. le point de départ est le graphe de relations de voisinage (figure 2.28) ;
   - FF_1 est productrice (côté "lent") par rapport à FF_2,
   - FF_1 est consommatrice (côté "lent") par rapport à FF_2,
   - FF_2 est consommatrice (côté "rapide") par rapport à FF_1,
   - FF_2 est productrice (côté "rapide") par rapport à FF_1,
   - FF_2 est productrice (côté "lent") par rapport à FF_3,
   - FF_2 est consommatrice (côté "lent") par rapport à FF_3,
   - FF_3 est productrice (côté "rapide") par rapport à FF_2,
   - FF_3 est consommatrice (côté "rapide") par rapport à FF_2.
2. la frontière FF_1 est une frontière "infinie", étant à la fois, la frontière la plus en amont et la plus en aval du graphe algorithmique. Pour cette frontière "infinie", il faut générer les signaux d'entrée rfu_1, afd_1, rsu_1 et asd_1 :
   - productrice côté "rapide" : FF_2

$$?\,rfu_1 =!\,rsd_2 \tag{3.12}$$

   - consommatrice côté "rapide" : FF_2

$$?\,afd_1 =!\,asu_2 \tag{3.13}$$

   - producteur côté "lent" : l'environnement

$$?\,rsu_1 = 1 \tag{3.14}$$

- consommateur côté "lent" : l'environnement

$$?\,asd_1 = 1 \tag{3.15}$$

3. la frontière FF_2 est une frontière "finie". Pour cette frontière, il faut générer les signaux d'entrée rfu_2, afd_2, rsu_2 et asd_2 :
   - productrice côté "rapide" : FF_3

$$?\,rfu_2 =!\,rsd_3 \tag{3.16}$$

   - consommatrice côté "rapide" : FF_3

$$?\,afd_2 =!\,asu_3 \tag{3.17}$$

   - productrice côté "lent" : FF_1

$$?\,rsu_2 =!\,rfd_1 \tag{3.18}$$

   - consommatrice côté "lent" : FF_1

$$?\,asd_2 =!\,afu_1 \tag{3.19}$$

4. la frontière FF_3 est une frontière "finie". Il faut générer les signaux d'entrée rfu_3, afd_3, rsu_3 et asd_3 :
   - productrice côté "rapide" : FF_3

$$?\,rfu_3 = rfd_3 \tag{3.20}$$

   - consommatrice côté "rapide" : FF_3

$$?\,afd_3 = afu_3 \tag{3.21}$$

   - productrice côté "lent" : FF_2

$$?\,rsu_3 =!\,rfd_2 \tag{3.22}$$

   - consommatrice côté "lent" : FF_2

$$?\,asd_3 =!\,afu_2 \tag{3.23}$$

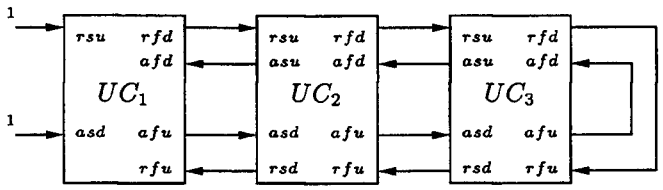


FIG. 3.36: Interconnexion des unités de contrôle pour le PMV factorisé

Ainsi, à partir des éqs. 3.12 à 3.23, on peut interconnecter les unités de contrôle UC_1, UC_2 et UC_3, comme le montre la figure 3.36.

La figure 3.37 représente l'implantation matérielle du PMV factorisé correspondant à la spécification algorithmique représentée par la figure 2.26. Le chemin de données est constitué des opérateurs frontière de factorisation (F, D, J et I) et des opérateurs combinatoires délimités par les boîtes en pointillés : la boîte à droite de FF_3 s'est composée des opérateurs combinatoires *mul* et *add*, les autres boîtes se sont composées trivialement des interconnexions entre les opérateurs frontière de factorisation. Le chemin de contrôle est constitué par les unités de contrôle UC_1, UC_2 et UC_3, et par leurs signaux de contrôle r, a, cpt, en.

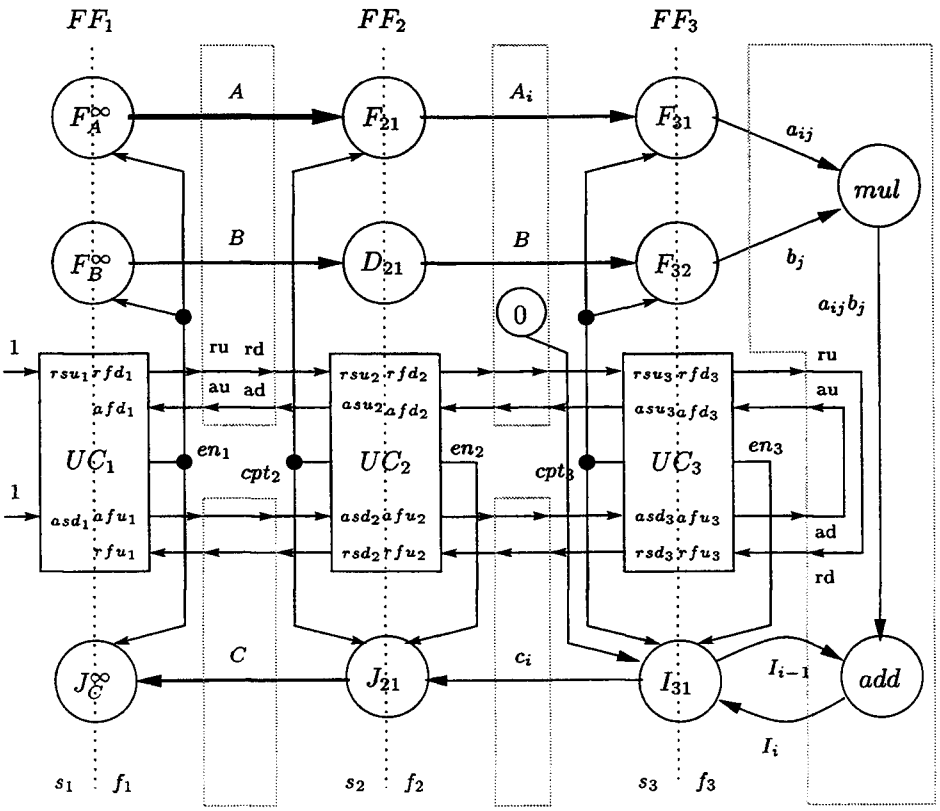


FIG. 3.37: Graphe matériel factorisé du PMV (implantation)

Le tableau 3.4 présente les résultats de l'implantation matérielle factorisée sur un FPGA *Xilinx* XC4052XL du PMV entre une matrice 6×6 et un vecteur de 6 éléments, codés sur 3 bits, en utilisant les outils de CAO ci-dessous :

- compilateur VHDL : *QuickVHDL qvcom v8.4-4.3e HP-UX A.09.05*, développé par *Mentor Graphics Co.* ;
- simulateur VHDL : *QuickVHDL qvsim v8.4-4.3e HP-UX A.09.05*, développé par *Mentor Graphics Co.* ;
- synthétiseur VHDL : *Leonardo Spectrum Level 2, v. 1998.2e*, développé par *Exemplar Logic, Inc.*

TAB. 3.4: Implantation matérielle du PMV totalement factorisé

Implantation	*Nb. cycl.*	*Delay (ns)*	*Latence (ns)*	*Freq. (MHz)*	*Nb. FGs*	*Nb. CLBs*	*Nb. DFFs*
Factorisée	36	81	2916	12,4	155	76	217

La figure 3.38 montre le diagramme temporel obtenu par simulation du code VHDL correpondant à l'implantation matérielle de la spécification factorisée d'un PMV entre une matrice de (6×6) éléments et un vecteur de 6 éléments, selon l'éq. 3.24. Dans ce diagramme, les signaux *entrea*, *entreb* et *sortie* correspondent respectivement aux entrées des capteurs F_A^∞ et F_B^∞, et à la sortie de l'actionneur J_C^∞. Les signaux *matrice* et *vecteur* correspondent respectivement aux sorties des capteurs F_A^∞ et F_B^∞. Les signaux $dif21$, $j21$, $frk21$, $frk31$ et $frk32$ correspondent respectivement aux sorties des opérateurs frontières D_{21}, J_{21}, F_{21}, F_{31} et F_{32}. Les signaux restants correspondent aux signaux de contrôle.

$$\begin{vmatrix} 2 & 3 & 5 & 1 & 4 & 7 \\ 3 & 4 & 6 & 2 & 5 & 1 \\ 1 & 2 & 4 & 3 & 5 & 7 \\ 4 & 1 & 3 & 5 & 2 & 6 \\ 7 & 5 & 1 & 4 & 3 & 2 \\ 5 & 7 & 2 & 6 & 1 & 4 \end{vmatrix} \cdot \begin{vmatrix} 7 \\ 4 \\ 3 \\ 2 \\ 6 \\ 1 \end{vmatrix} = \begin{vmatrix} 74 \\ 90 \\ 70 \\ 69 \\ 100 \\ 91 \end{vmatrix} \tag{3.24}$$

3.10.2 Filtrage des lignes d'une matrice

À partir du graphe algorithmique factorisé du filtrage des lignes d'une image, montré par la figure 2.29, nous devons générer le graphe matériel correspondant.

L'interconnexion des signaux de requête et d'acquittement est effectuée par l'intermédiaire de l'analyse des relations de voisinage entre les frontières de factorisation FF_1, FF_2, FF_3 et FF_4 :

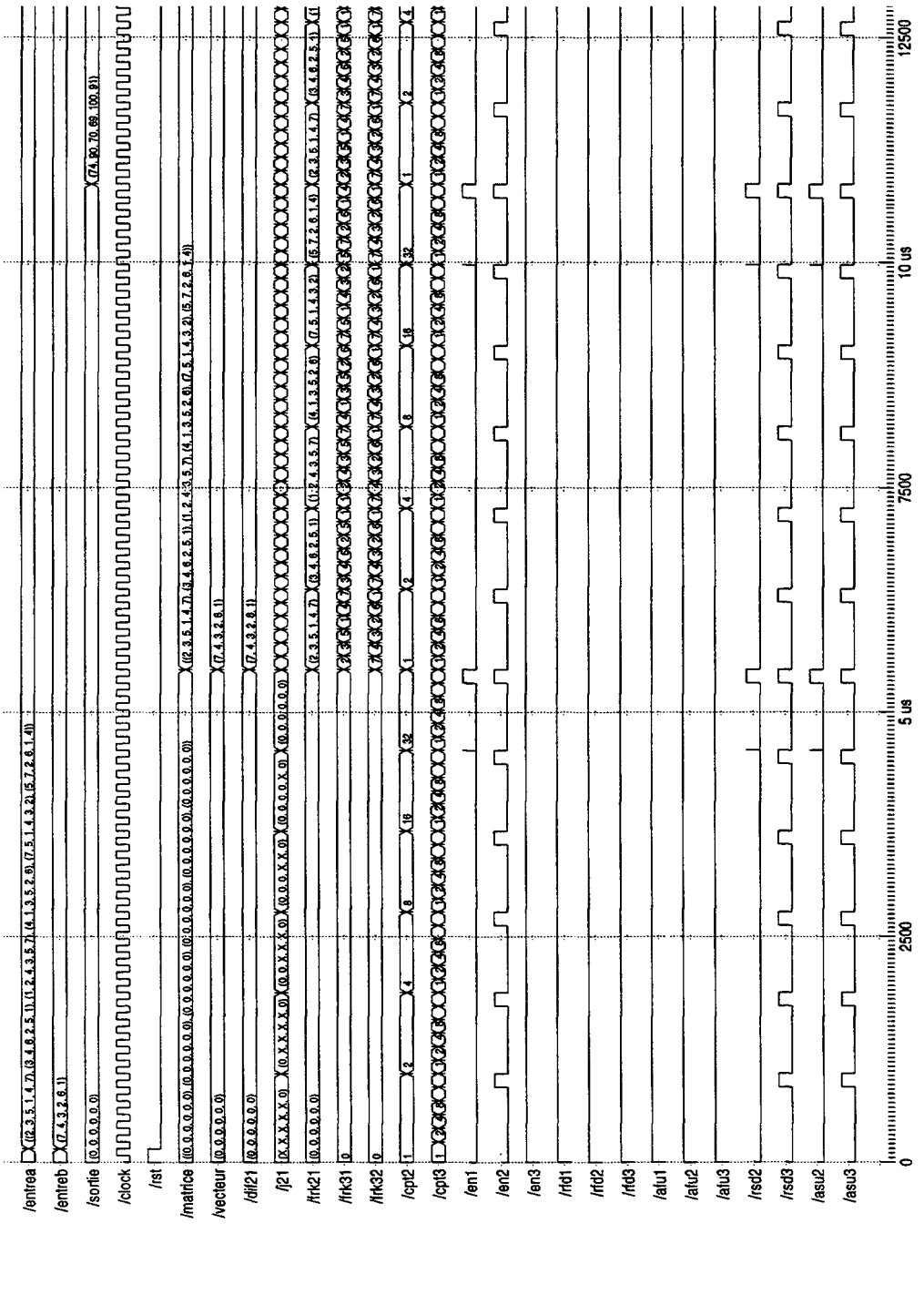


FIG. 3.38: Diagramme temporel du PMV factorisé

1. le point de départ est le graphe de relations de voisinage (figure 2.30) ;
   - FF_1 est productrice (côté "rapide") par rapport à FF_2 ;

- FF_2 est productrice (côté "lent") par rapport à FF_3 ;
- FF_3 est productrice (côté "rapide") par rapport à FF_4 ;
- FF_2 est consommatrice (côté "rapide") par rapport à FF_1 ;
- FF_3 est consommatrice (côté "lent") par rapport à FF_2 ;
- FF_4 est consommatrice (côté "rapide") par rapport à FF_3.

2. la frontière FF_1 est une frontière "infinie". C'est la frontière la plus en amont du graphe algorithmique. Pour cette frontière, il faut générer les signaux rfu_1, afd_1, rsu_1 et asd_1 :
   - productrice côté "rapide" : aucune (NU[1])

$$?\,rfu_1 = NU \tag{3.25}$$

   - consommatrice côté "rapide" : FF_2

$$?\,afd_1 =!\,afu_2 \tag{3.26}$$

   - producteur côté "lent" : l'environnement

$$?\,rsu_1 = 1 \tag{3.27}$$

   - consommatrice côté "lent" : aucune

$$?\,asd_1 = NU \tag{3.28}$$

3. la frontière FF_2 est une frontière "finie". Pour cette frontière, il faut générer les signaux d'entrée rfu_2, afd_2, rsu_2 et asd_2 :
   - productrice côté "rapide" : FF_1

$$?\,rfu_2 =!\,rfd_1 \tag{3.29}$$

   - consommatrice côté "rapide" : aucune

$$?\,afd_2 = NU \tag{3.30}$$

   - productrice côté "lent" : aucune

$$?\,rsu_2 = NU \tag{3.31}$$

[1] non-utilisé

- consommatrice côté "lent" : FF_3

$$?\,asd_2 =!\,asu_3 \tag{3.32}$$

4. la frontière FF_3 est une frontière "finie". Pour cette frontière, il faut générer les signaux d'entrée rfu_3, afd_3, rsu_3 et asd_3 :
   - productrice côté "rapide" : aucune

$$?\,rfu_3 = NU \tag{3.33}$$

   - consommatrice côté "rapide" : FF_4

$$?\,afd_3 =!\,afu_4 \tag{3.34}$$

   - productrice côté "lent" : FF_2

$$?\,rsu_3 =!\,rsd_2 \tag{3.35}$$

   - consommatrice côté "lent" : aucune

$$?\,asd_3 = NU \tag{3.36}$$

5. la frontière FF_4 est une frontière "infinie". C'est la frontière la plus en aval du graphe algorithmique. Pour cette frontière, il faut générer les signaux rfu_4, afd_4, rsu_4 et asd_4 :
   - productrice côté "rapide" : FF_3

$$?\,rfu_4 =!\,rfd_3 \tag{3.37}$$

   - consommatrice côté "rapide" : aucune

$$?\,afd_4 = NU \tag{3.38}$$

   - productrice côté "lent" : aucune

$$?\,rsu_4 = NU \tag{3.39}$$

   - consommatrice côté "lent" : l'environnement

$$?\,asd_4 = 1 \tag{3.40}$$

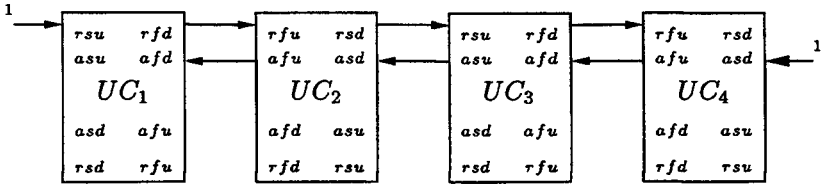


FIG. 3.39: Interconnexion des unités de contrôle pour le filtrage des lignes d'une image

Ainsi, à partir des éqs. 3.26 et 3.38, nous pouvons interconnecter les unités de contrôle UC_1, UC_2, UC_3 et UC_4, comme le montre la figure 3.39, qui correspond au graphe matériel détaillé du filtrage des lignes d'une image.

La figure 3.40 représente l'implantation matérielle du filtrage des lignes d'une image factorisé correspondant à la spécification algorithmique représentée par la figure 2.29. Le chemin de données est constitué par les opérateurs frontière de factorisation (F et J) et les opérateurs combinatoires délimités par les boîtes en pointillés : la boîte située entre FF_2 et FF_3 s'est composée de l'opérateur combinatoire $filt$, les autres boîtes se sont composées trivialement d'interconnexions entre les opérateurs frontière de factorisation. Le chemin de contrôle est constitué par les unités de contrôle UC_1, UC_2, UC_3 et UC_4, et par leurs signaux de contrôle r, a, cpt, en.

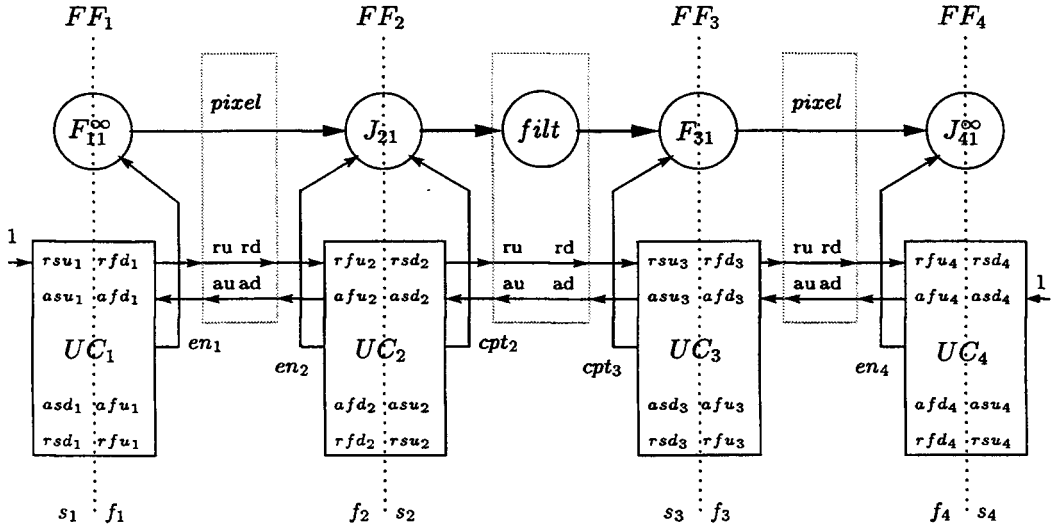


FIG. 3.40: Graphe matériel factorisé du filtrage des lignes d'une image

Le tableau 3.5 présente les résultats de l'implantation matérielle factorisée sur un FPGA *Xilinx* XC4020XL du filtrage des lignes (20 pixels en entrée, 18 pixels en sortie, masque de convolution de 3 éléments) d'une image dont les pixels sont codés sur 8 bits.

TAB. 3.5: Implantation matérielle du filtrage totalement factorisé

Implantation	*Nb. cycl.*	*Delay (ns)*	*Latence (ns)*	*Freq. (MHz)*	*Nb. FG*	*Nb. CLB*	*Nb. DFF*
Factorisée	38	58	2204	17,0	360	96	180

La figure 3.42 montre le diagramme temporel obtenu par simulation du code VHDL correspondant à l'implantation matérielle de la spécification factorisée du filtrage des lignes d'une image montré par la figure 3.41. Dans ce diagramme, les signaux *pxlin* et *pxlout* correspondent, respectivement, à l'entrée du capteur F_{11}^{∞} et à la sortie de l'actionneur J_{41}^{∞}. Les signaux *pixelin*, $j21$, $filt$ et $frk31$ correspondent, respectivement, aux sorties des opérateurs F_{11}^{∞}, J_{21}, $filt$ et F_{31}. Les signaux restants correspondent aux signaux de contrôle.

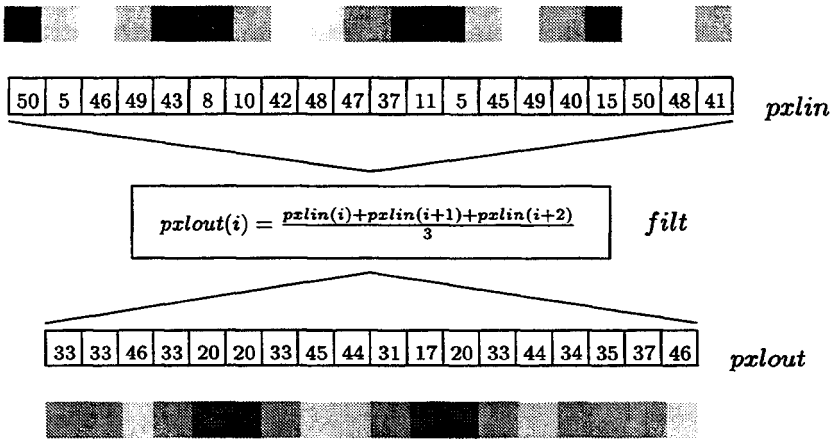


FIG. 3.41: Exemple de filtrage

3.11 Conclusion

Dans ce chapitre, nous avons défini les opérateurs qui permettent d'effectuer l'implantation matérielle correspondante à une spécification algorithmique donnée. Nous y avons décrit le mécanisme de contrôle des opérateurs frontières de factorisation et la synthèse du contrôle à partir des relations entre les frontières. En partant d'une spécification algorithmique basée sur un modèle de graphe de dépendances factorisé, nous avons montré qu'il est possible d'obtenir une implantation matérielle sous la forme d'un circuit, en utilisant des règles simples de synthèse des chemins de données et de contrôle.

Notre méthodologie, comme celle proposée par Cesário et al. [CSSJ99], explore la superposition entre la synthèse comportementale et la synthèse RTL, afin de permettre au concepteur d'explorer différents flots de synthèse en fonction de la nature de l'application et de ses contraintes, et de la qualité des outils de synthèse

utilisés en aval dans le cycle de conception. Le grand avantage de notre méthodologie par rapport aux travaux précédents réside dans le fait d'utiliser un même modèle de conception depuis la spécification algorithmique jusqu'à l'implantation matérielle. Ce processus de génération d'implantation matérielle pourra être automatisé pour générer un code VHDL structurel synthétisable pour les outils de CAO.

La logique de contrôle hiérarchique "délocalisée", présentée dans ce chapitre, permet aux outils de CAO utilisés pour la synthèse de placer les unités de contrôle plus proches des opérateurs à contrôler qu'une logique de contrôle "centralisée". Cela nous permet de réduire le chemin critique de façon importante, comme vérifié par Huang et Wolf [HW94]. La hiérarchisation des contrôleurs locaux réduit la surface occupée par le chemin de contrôle. Dans l'exemple de la figure 3.37, l'utilisation de notre stratégie "délocalisée" produit un contrôleur constitué de deux compteurs de trois bits et une vingtaine de portes logiques (*ET*, *OU* et *NON*). En revanche, l'implantation d'un contrôleur équivalent, en utilisant une stratégie "centralisée", exigerait une machine à état de 36 états, dont la surface est plus importante que la solution préconisée.

Comme la qualité des résultats de la synthèse ne dépend pas seulement de la qualité des outils utilisés, mais aussi du processus de conception, il faut offrir au concepteur la possibilité d'explorer l'espace de solutions possibles, afin d'obtenir une implantation optimisée. Dans le chapitre suivant, nous montrons comment effectuer l'optimisation de l'implantation matérielle. Nous présenterons le modèle de conception pour la génération de *netlist* pour architecture mono-FPGA, qui utilise un prédicteur de performances, afin de permettre l'optimisation par défactorisation, sans être obligé de passer par un cycle complet de conception.

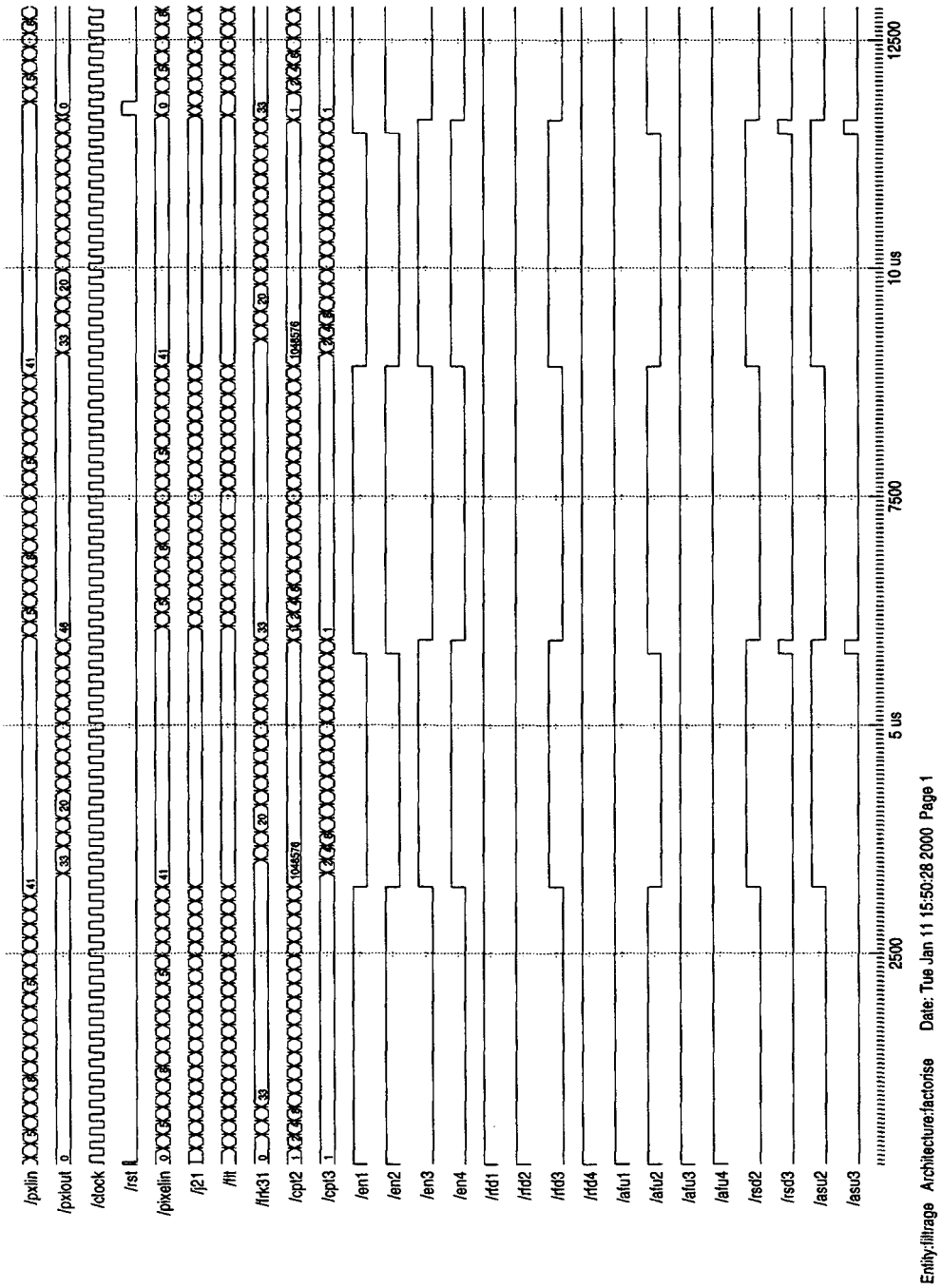


FIG. 3.42: Diagramme temporel du filtrage factorisé

Chapitre 4

Optimisation de l'implantation

L'automatisation partielle ou totale du processus de conception et la migration de cette automatisation vers les niveaux les plus élévés de la conception présentent plusieurs avantages [Lin97].

4.1 Introduction

L'automatisation assure un cycle de conception plus court ; elle permet aussi l'exploration de l'espace de solutions, puisque différentes implantations peuvent être générées et évaluées de manière rapide. Ensuite, les outils de synthèse peuvent générer des implantations plus efficaces qu'un concepteur moyen.

Notre méthodologie part d'une description comportementale (spécification algorithmique) qui représente les spécifications complètes de ce que le système doit accomplir et elle les transforment dans un schéma logique qui les implante (implantation matérielle respectant les contraintes temporelles). La recherche d'une transformation optimale, qui respecte les contraintes temporelles, tout en minimisant l'augmentation du nombre de ressources matérielles nécessaires à l'implantation matérielle, est un problème NP-complet [LS97] qui peut exiger l'investigation d'un ensemble très important de combinaisons.

Pour éviter cette explosion combinatoire, nous utilisons des méthodes heuristiques de recherche. Ces méthodes utilisent des informations heuristiques, qui dépendent de l'application pour réduire l'espace de solutions et, par conséquent, réduire la recherche. Certaines heuristiques peuvent réduire énormément l'effort de recherche, mais elles ne garantissent pas la découverte de la solution optimale.

L'exploration automatique de l'espace de solutions concerne quatre sous-problèmes interdépendants : le partitionnement matériel/logiciel, l'allocation de ressources matérielles, la transformation de la spécification et l'estimation des performances temporelles et du nombre de ressources matérielles nécessaires à l'im-

plantation [GV95]. Dans ce travail, nous ne nous intéressons qu'aux deux derniers sous-problèmes : la transformation de la spécification par l'intermédiaire d'une optimisation par défactorisation et la prédiction de la consommation de ressources matérielles et du temps de réponse de l'implantation optimisée, à partir du graphe algorithmique de l'application.

Les techniques d'estimation de consommation de ressources matérielles et de temps de réponse ne sont pas très précises, puisque la mise en correspondance entre la description comportementale et l'implantation matérielle n'est pas directe (un-pour-un) [GV95]. Les compilateurs et les outils de placement et routage effectuent des optimisations inconnues par l'estimateur. Ainsi, il est très difficile de prédire les réductions de surface ou de temps de réponse engendrées par ces outils. Nous nous contentons donc d'une prédiction qui, tout en n'étant pas grossière, est capable de guider les décisions des heuristiques d'optimisation. Dans le cas d'une implantation sur une architecture à base de FPGA (*Field Programmable Gate Array*), nous pouvons estimer le nombre total de CLB (*Configurable Logic Block* – bloc logique configurable) en additionnant les CLB utilisés par chaque opérateur du graphe matériel.

Dans ce chapitre, nous présentons l'état de l'art sur les méthodes d'optimisation utilisées par les systèmes de synthèse matérielle existants, afin de situer notre contribution par rapport aux travaux réalisés ailleurs. Comme notre problème d'allocation de ressources (recherche d'un minimum sous contraintes) est un problème NP-complet, nous nous intéressons plus particulièrement aux méthodes basées sur des heuristiques de recherche. Nous décrivons notre modèle de conception, orienté à la génération de *netlists* de configuration pour les architectures mono-FPGA, à partir d'un modèle de graphes factorisés de dépendances de données. Ensuite, nous discutons notre méthode de caractérisation matérielle des opérateurs et des unités de contrôle, en termes de surface et de latence, en fonction de l'architecture-cible. Cette méthode peut être appliquée à n'importe quel FPGA *Xilinx* de la série XC4000. Pour illustrer la méthode, nous avons choisi un FPGA de la série XC4000XL-3. Nous y présentons également notre méthode d'estimation du nombre de ressources matérielles nécessaires à l'implantation matérielle et de la latence totale de cette implantation. En dernier, nous décrivons notre méthode d'optimisation de l'implantation par défactorisation, basée sur une heuristique de recherche qui est guidée par des informations fournies par l'estimateur de surface et de latence.

4.1.1 Optimisation de la synthèse de haut niveau

Dans la section 3.1.1, nous avons dit que la synthèse de haut niveau doit être capable de prédire ou calculer l'effet des décisions prises aux niveaux système et comportemental (p.ex., l'ordonnancement et la distribution), afin d'obtenir une implantation matérielle efficace. Cet effet est mesuré en termes de coûts (surface, performances temporelles, consommation), qui doivent être utilisés par les algorithmes de HLS (*High-Level Synthesis* – synthèse de haut niveau) pour guider leurs décisions. Dans les systèmes de HLS existants, ces coûts sont estimés de façon grossière, donnant

une indication insuffisante de la complexité et de la performance des chemins de données et de contrôle de l'implantation matérielle. Ceci arrive parce qu'ils ignorent, presque complètement, quelques aspects fondamentaux, tels que la surface et le retard de la logique de contrôle, des multiplexeurs et des registres [Ber99]. Dans notre méthodologie, les opérateurs et les unités de contrôle sont d'abord caractérisés en fonction de l'architecture-cible. Après cette caractérisation, l'estimateur de surface et de latence de l'implantation effectue l'estimation des chemins de données et de contrôle, avec une précision moyenne de 15 % pour la surface et de 9 % pour la latence, comme nous montrent les tableaux A.15, A.16, A.17 et A.18, situés dans l'Annexe A.

4.1.2 Principes de l'optimisation de la synthèse de circuits programmables

Abouzeid et al. [ABCS93] ont proposé une technique de synthèse orientée aux FPGA *Xilinx* des séries XC3000 et XC4000, à partir d'un ensemble d'équations booléennes. Cette technique, basée sur des expressions lexicographiques et sur un partitionnement guidé par les entrées, s'est révélée très efficace en termes du temps de réponse et de la surface utilisée. D'abord, un critère d'optimisation cherche à minimiser le chemin critique pour, ensuite, minimiser le nombre total de blocs logiques configurables.

Prado Lopes Filho [Pra96] décrit une méthode d'optimisation globale pour la synthèse de circuits programmables. Le circuit à synthétiser est représenté par des MROBDD (*Multi-Reduced and Ordered Binary Decision Diagrams* – graphes de décision binaires à sorties multiples). En partant de la description comportementale en VHDL du circuit, la méthode permet d'optimiser l'implantation de l'architecture-cible. Le but de l'exploration de l'espace des solutions pendant l'optimisation est de trouver le "meilleur" ordonnancement des variables, du point de vue de la décomposition du Multi-ROBDD en un réseau de portes. La méthode s'applique à des architectures basées sur des FPGA *Xilinx* 3000, *Xilinx* 4000 et *Actel* Act1, à partir de deux caractéristiques de ces circuits : le nombre de blocs logiques utilisés et le nombre maximal de blocs logiques traversés. Le programme de synthèse utilisé est *Alligator*, développé à l'ex-laboratoire MASI (Méthodologie et Architecture des Systèmes Informatiques), actuel département d'Architecture des Systèmes Intégrés et Microélectronique (ASIM), à l'Université Pierre et Marie Curie, dans le cadre de la chaîne *Alliance* [PRD96].

Dans notre méthode d'optimisation par défactorisation, nous avons adopté une stratégie de synthèse similaire à celle proposé par Abouzeid et al. [ABCS93]. Ainsi, nous cherchons d'abord à satisfaire la contrainte temporelle, pour ensuite affiner l'optimisation, afin de réduire le nombre de ressources matérielles utilisées.

4.1.3 Méthodes heuristiques

Plusieurs problèmes concernant le partitionnement de graphes, l'ordonnancement de tâches et l'optimisation sont considérés comme des problèmes NP-complets, pour lesquels il n'existe pas d'algorithme optimal capable de les résoudre dans un temps polynomial. La solution optimale pour ces problèmes ne peut donc pas être obtenue, sans que l'on parcourt tout l'espace de solutions possibles.

Dans le cadre de ce travail, comme nous nous restreindrons aux architectures mono-FPGA, nous n'effectuerons pas le partitionnement de graphes. Toutefois, il sera important dans la suite des travaux quand cette notre méthodologie devra être étendue aux circuits multi-FPGA.

Partitionnement de graphes

Le partitionnement de graphes est un problème fondamental dans le domaine de la conception automatique de circuits. Son but est de séparer les sommets d'un graphe, dont les arcs sont étiquettés, en plusieurs sous-graphes soumis à des contraintes de taille ou d'équilibrage de tâches, tout en minimisant les interconnexions entre les sous-graphes. Le partitionnement de graphes sera d'importance extrême dans le prototypage rapide utilisant les architectures basées sur des multi-FPGA, puisque l'augmentation de la complexité et de la taille des circuits, engendrée par ces architectures exige des algorithmes plus efficaces et plus performants. Plusieurs heuristiques ont été conçues pour résoudre le problème de partitionnement de graphes [CSZ94]. Elles sont basées sur :

- hypergraphes × graphes : dans une heuristique basée sur un hypergraphe, le degré des arcs peut être supérieur à deux. Ainsi, les réseaux de circuits multi-broches peuvent être très bien représentés par ce modèle d'heuristique. Par contre, dans une heuristique basée sur un graphe, le degré des arcs est limité à deux ;
- spectrales × itératives : dans les techniques de partitionnement spectral, les vecteurs propres et les valeurs propres (spectre) d'un graphe sont calculés et une fonction de coût peut être minimisée par une fonction du spectre. Les heuristiques itératives explorent l'espace de solutions, par l'intermédiaire d'un grand nombre de mouvements (petits changements dans la solution), de façon aléatoire ou gloutonne, afin de trouver un minimum global [KL70, FM82] ;
- multi-mode (k-mode) × bi-mode : une heuristique de partitionnement k-mode divise l'hypergraphe en k sous-graphes de sommets disjoints [San89], alors que le bipartitionnement (bi-mode) divise un hypergraphe en deux sous-graphes [Kri84]. L'heuristique bi-mode peut être appliquée de façon récursive, afin de générer k partitions au détriment de la qualité de la solution trouvée.

Fonction-objectif et contraintes

Une fonction-objectif (ou fonction de coût) est une mesure que le système essaie de maximiser ou de minimiser, en fonction de certaines contraintes. Les contraintes sont des conditions qui doivent être satisfaites. En synthèse architecturale, les contraintes et les fonctions-objectif sont définies sur le nombre de ressources matérielles (surface) et le temps de traversée du circuit. L'ordonnancement peut essayer de minimiser le nombre d'étapes de contrôle, comme dans *Mimola* [Mar86]. *Hal* [PKG86], à son tour, minimise la surface, étant donnée une contrainte de temps. *Maha* [PPM86] et *Sehwa* [PP88] peuvent minimiser la surface avec des contraintes de temps ou minimiser le temps avec des contraintes de surface. BUD [McF86] utilise une fonction objectif qui est une combinaison de surface et de temps. Dans l'algorithme de partitionnement BCS (*Binary Constraint Search*) [VAH94], qui essaie de trouver une solution respectant les contraintes temporelles, tout en minimisant l'augmentation du matériel ajouté au processeur, la fonction de coût utilisée comporte deux termes : l'un correspond à la somme des violations des contraintes temporelles (différence entre le temps de réponse de la partition et la contrainte temporelle), l'autre correspond à la quantité de ressources matérielles utilisées :

$$Cout(Mat, Log, Cont) = k_t \cdot \sum_{i=1}^{m} viol(C_i) + k_s \cdot S_{Mat}$$

où

Mat : partition matérielle,
Log : partition logicielle,
$Cont$: contraintes temporelles,
k_t : coefficient temporelle,
k_s : coefficient de surface,
S_{mat} : ressources matérielles.

Si les coefficients k_t et k_s sont choisis de façon que $k_t >> k_s$, l'algorithme cherchera d'abord à respecter les contraintes temporelles, pour, ensuite minimiser les ressources matérielles nécessaires à son implantation.

Dans notre méthodologie, nous cherchons à optimiser l'implantation matérielle d'une spécification algorithmique factorisée par défactorisation, afin de trouver l'implantation qui respecte les contraintes temporelles et qui minimise l'augmentation des ressources matérielles nécessaires. La défactorisation n'est pas une transformation standard, comme le partitionnement de graphes ou l'ordonnancement de tâches, mais nous pouvons nous servir des principes utilisés pour résoudre ces problèmes, pour aboutir à une heuristique de défactorisation. Ainsi, nous utilisons une fonction objectif qui combine la latence (temps de réponse) et la surface (nombre de ressources matérielles) pour guider notre heuristique d'optimisation par défactorisation, comme nous verrons dans la section 4.5.

Problème de l'explosion combinatoire

En partant d'une spécification factorisée, il peut y avoir un nombre très grand de défactorisations possibles et capables de satisfaire plus ou moins les contraintes temporelles de l'application. Nous sommes face à un problème d'explosion combinatoire et la recherche de l'implantation optimisée est aussi un problème NP-complet. Nous faisons donc appel à des heuristiques, afin d'explorer l'espace de solutions et d'en trouver l'implantation optimisée, sans parcourir pour autant tout cet espace. Selon Gondran et Minoux [GM95], "les critères de choix dans les explorations sont fondamentaux. Les heuristiques faisant ces choix sont donc déterminantes. Les deux méta-heuristiques, *faire le choix le plus informant*[1] et *faire le choix le moins cher*, peuvent engendrer un très grand nombre d'heuristiques".

Nous nous sommes basés sur les caractéristiques de l'outil ALICE (*A Language for Intelligent Combinatorial Exploration*) [GM95] pour établir notre heuristique. ALICE est un outil d'exploration dont le fonctionnement se fait en trois phases :

1. recherche d'une solution réalisable ;
2. recherche d'une bonne solution (recherche d'une solution tenant compte de la fonction de coût) ;
3. recherche de la solution optimale.

La bonne utilisation de ces critères et le choix de l'ordre dans lequel ils sont utilisés permet de résoudre plusieurs problèmes combinatoires de natures très diverses et, parmi eux, des problèmes d'optimisation.

Évidemment, dans notre cas, nous tenons compte seulement des deux premières phases, puisque la recherche de la solution optimale exige le parcours de tout l'espace de solutions. Nous cherchons donc d'abord à satisfaire la contrainte temporelle pour ensuite minimiser la consommation de ressources matérielles.

4.2 Modèle de conception

Nous présentons ci-dessous une description de chacune des étapes du modèle de conception proposé pour la génération de *netlists* de configuration pour les architectures basées sur des circuits reconfigurables du type FPGA, à partir d'une spécification algorithmique sous la forme d'un GFDD. Le flot de conception représenté par la figure 4.1 a été conçu de façon à être intégré au logiciel *SynDEx* qui implante la méthode AAA développée à INRIA–Rocquencourt. Dans cette figure, les boîtes grises correspondent aux modules à réaliser pour étendre le logiciel *SynDEx* (caractérisation matérielle, traduction matérielle, estimation de surface et latence, evaluation de performances, l'optimisation par défactorisation et génération

[1] Le terme *informant* signifie 'qui contient plus d'information'.

de code VHDL). La *synthèse de circuit* est effectuée avec l'aide d'un outil de CAO, tel que *Leonardo*, *Cadence*, etc. Les boîtes *spécification algorithmique*, *spécification matérielle* et *contraintes temporelles* représentent les entrées du flot de conception, qui doivent être fournies par le concepteur. La boîte *netlists pour FPGA* correspond aux fichiers de configuration des FPGA générés par l'outil de CAO qui réalise la synthèse matérielle.

1. **Spécification algorithmique** : le concepteur décrit l'algorithme de l'application sous la forme d'un graphe factorisé de dépendances de données (GFDD), par l'intermédiaire de l'interface graphique de *SynDEx* ou à partir d'un fichier *.sdx* obtenu après traduction d'une spécification fait avec un des langages synchrones, comme décrit dans le chapitre 2 ;
2. **Spécification architecturale** : le concepteur modélise l'architecture-cible en définissant le nombre de FPGA, le type de FPGA et le type de connexion inter-FPGA, par l'intermédiaire de l'interface graphique de *SynDEx*. Comme dans ce travail nous nous restreidrons aux architectures mono-FPGA, la phase de spécification architecturale se résume à la sélection du type de FPGA utilisé, en termes du nombre de CLB et de E/S ;
3. **Caractérisation matérielle** : l'ensemble des opérations du graphe factorisé d'opérations est caractérisé en termes de surface, en nombre de générateurs de fonction F/G, de CLB et de bascules DFF, et en termes de latence, en fonction de l'architecture-cible, du codage des données et du nombre de répétitions des motifs factorisés, cf. décrit dans la section 4.3 ;
4. **Traduction matérielle** : un graphe matériel d'opérateurs peut être produit automatiquement à partir du GFDD, en remplaçant les sommets de ce dernier par les opérateurs correspondants (caractérisés en termes de surface et latence) et les transferts de données par les interconnexions des opérateurs. Le contrôleur est aussi caractérisé à partir de l'analyse du graphe des relations de voisinage entre les frontières, comme décrit dans le chapitre 3 ;
5. **Estimation de surface et de latence** : un estimateur de la consommation de ressources matérielles et des performances temporelles analyse le graphe matériel d'opérateurs et estime la surface correspondante au chemin de données et au chemin de contrôle (en nombre de F/G, CLB et DFF) et la latence de la traduction matérielle de la spécification algorithmique, cf. décrit dans la section 4.4 ;
6. **Évaluation des performances temporelles** : la latence calculée par l'estimateur des performances temporelles est comparée à la contrainte temps réel de l'application, afin d'obtenir les paramètres nécessaires à la fonction objectif (ou fonction de coût) qui va guider l'heuristique d'optimisation, cf. décrit dans la section 4.5 :

   - si la valeur de la latence estimée est inférieure à la contrainte temps réel et l'augmentation de la consommation de ressources matérielles a été minimisée, nous pouvons effectuer l'implantation matérielle détaillée de la spécification algorithmique,

- sinon, il faut réaliser l'optimisation par défactorisation de la spécification algorihmique ;

7. **Optimisation par défactorisation** : l'optimisation par défactorisation de la spécification algorithmique consiste à choisir une transformation à appliquer sur le GFDD, à l'aide de notre heuristique de défactorisation, en fonction des contraintes temporelles fournies par le concepteur, des performances temporelles (latence) et de la consommation de ressources matérielles (surface) calculées par l'estimateur pour cette spécification. La transformation choisie cherche à réduire la latence de l'implantation matérielle correspondante et à minimiser l'augmentation de la consommation de ressources matérielles provoquée par la défactorisation. Après cette étape, il faut répéter les étapes correspondantes à la *traduction matérielle*, à l'*estimation de surface et latence* et à l'*évaluation des performances* et, si nécessaire, la *défactorisation* de la spécification algorithmique, jusqu'à obtenir une implantation qui respecte les contraintes temporelles et minimise la surface, ou jusqu'à la défactorisation totale de la spécification algorithmique, cf. décrit dans la section 4.5 ;
8. **Implantation matérielle** : l'implantation matérielle détaillée produit un graphe matériel élargi, composé du graphe d'opérateurs généré par la *traduction matérielle* de la spécification algorithmique, plus les circuits nécessaires à la synchronisation des transferts de registres (unités de contrôle). Ces circuits, ajoutés au graphe matériel d'opérateurs, composent la "partie contrôle" (ou *control path*) de l'implantation matérielle, comme décrit dans le chapitre 3 ;
9. **Génération de code** : chaque opérateur du graphe correspondant à l'implantation matérielle doit être traduit sous la forme d'un composant d'une bibliothèque VHDL. La génération de code consiste à interconnecter ces composants pour obtenir un code VHDL structurel synthétisable, qui correspond à l'implantation optimisée de la spécification algorithmique définie par le concepteur, cf. décrit la section 4.5.4 ;
10. **Synthèse des circuits** : le code VHDL structurel synthétisable est utilisé comme entrée d'un outil de CAO (p.ex. *Leonardo*) pour effectuer la synthèse de l'architecture-cible. Dans ce travail, l'outil de synthèse génère les *netlists* nécessaires à la configuration des FPGA.

4.3 Caractérisation matérielle

Les systèmes de HLS traditionnels caractérisent leurs résultats de synthèse en se basant sur une estimation plutôt grossière des performances [Lin97]. La surface est estimée comme étant la somme des surfaces des unités fonctionnelles, de stockage et d'interconnexion. Le temps de réponse est estimé en supposant que le retard provoqué par le câblage est négligeable. La capacité à générer des chemins de données optimisés est une caractéristique fondamentale des méthodologies de conception modernes, puisque le chemin de données représente à peu près 80 % de la surface totale de l'implantation matérielle des applications [CKGJ97].

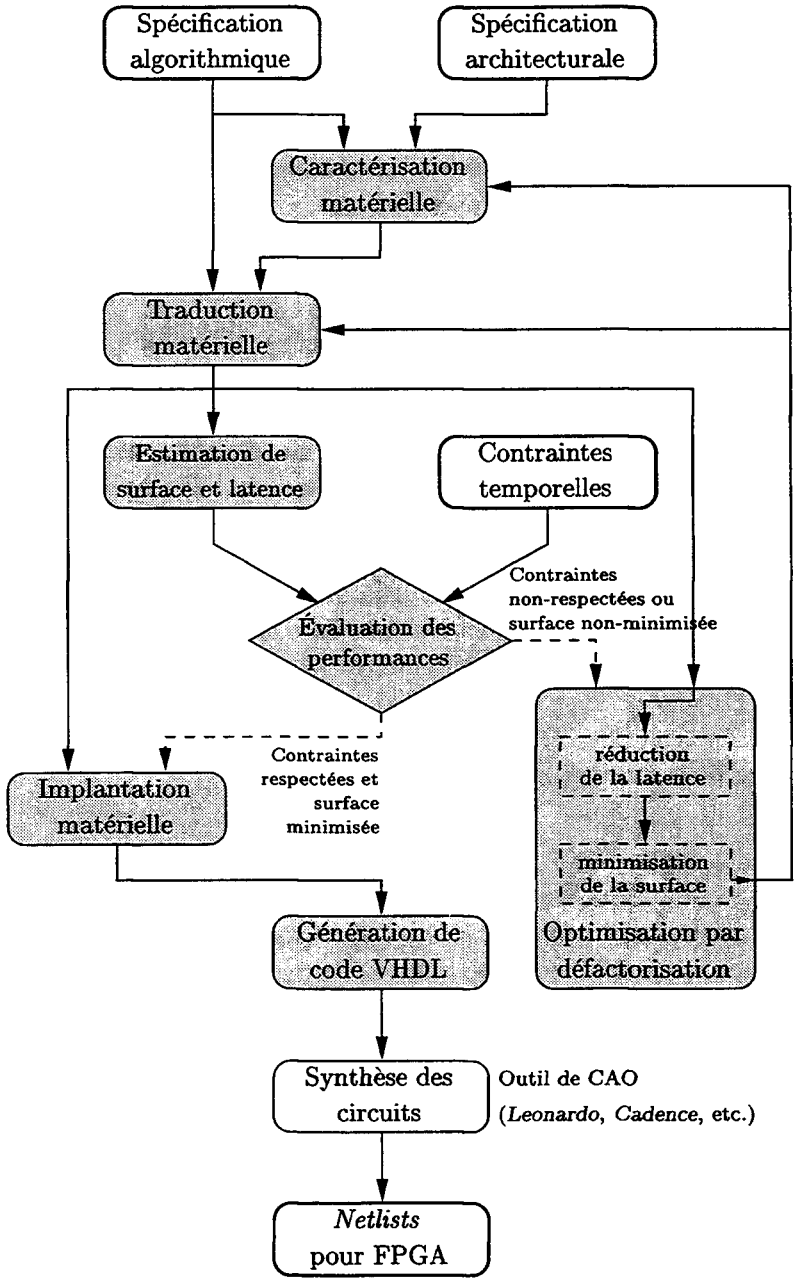


FIG. 4.1: Modèle de conception d'architectures mono-FPGA

Rim et Jain [RJ94] ont proposé un outil d'estimation qui, à partir d'un graphe flot de données représentant l'application, d'un ensemble de ressources matérielles, d'une liste de latences de ces ressources et d'une valeur de cycle d'horloge, calcule la borne inférieure du temps de réponse pour un problème d'ordonnancement non-pipeliné, sous contraintes de ressources. Chaudhuri et Walker [CW96] ont proposé un algorithme pour calculer les bornes inférieures du nombre d'unités fonctionnelles nécessaires pour ordonnancer un graphe flot de données en un nombre

donné d'étapes. Mecha et al. [M*96] ont proposé une méthode d'estimation de surface de haut niveau, orientée vers l'implantation de cellules standards. Cesário et al. [CKGJ97] ont présenté une étude comparative entre différents schémas d'interconnexion du chemin de données au niveau comportemental. Ils ont conclu que pour les implantations caractérisées par des grands chemins de données avec des petits contrôleurs, l'interconnexion basée sur des multiplexeurs est la plus efficace. Par contre, quand le contrôleur est caractérisé par un grand ensemble d'instructions, l'interconnexion basée sur des bus est la meilleure solution.

La caractérisation matérielle des sommets du graphe algorithmique nous permettra d'étiqueter ce graphe avec les valeurs de surface et de latence correspondant aux opérateurs qui les implantent. Cet étiquetage sera donc essentiel pendant l'étape d'estimation de surface et de latence de l'implantation de la spécification. Pour étiqueter les sommets d'un graphe algorithmique, il faut identifier chacun de ces sommets, analyser ses données en entrée et en sortie, et calculer la surface et la latence de l'opérateur équivalent à l'implantation de chaque sommet. Pour cela, il faut analyser l'implantation de chaque opérateur présenté dans le chapitre 2 et d'en déduire les règles d'estimation de surface et latence.

4.3.1 L'architecture-cible

Les architectures ciblées par l'extension de la méthodologie AAA aux circuits reconfigurables sont les architectures mono-FPGA basées sur les composants *Xilinx* de la série XC4000 [Xil94, Xil96, Xil99]. Ces composants sont implantés sous la forme d'une matrice régulière et programmable de CLB, interconnectés par des ressources de routage hiérarchiques et entourés par des IOB (*Input/Output Blocs* – blocs d'entrée/sortie) aussi programmables. Les CLB contiennent les éléments fonctionnels nécessaires à l'implantation de l'application ciblée. Les IOB assurent l'interface entre les broches d'E/S (entrée/sortie) et les lignes de signal internes au FPGA.

La structure simplifiée d'un CLB de la série XC4000 est montrée par la figure 4.2. Le CLB comprend deux fonctions logiques de quatre entrées (F et G) et une fonction logique de trois entrées (H). Le générateur de fonction H peut avoir comme entrée les sorties F' et/ou G' des générateurs F/G ou il peut utiliser des entrées externes. Un CLB peut donc implanter plusieurs fonctions logiques comprenant jusqu'à neuf variables. Chaque CLB possède deux éléments de stockage (bascules DFF), qui peuvent mémoriser les sorties des générateurs de fonction. Une donnée peut être stockée sans passer par les générateurs de fonction, en utilisant l'entrée directe D_{IN}. Les bascules peuvent être utilisées comme registres ou comme registre à décalage, sans pour autant empêcher les générateurs de fonction d'effectuer une tâche différente sans aucun rapport avec elles. Les tables de correspondance associées aux générateurs de fonction F et G peuvent être utilisées comme une matrice de cellules mémoire RAM. En fonction du mode sélectionné, un CLB peut être configuré comme une mémoire de 16×2, 32×1 ou 16×1 bits.

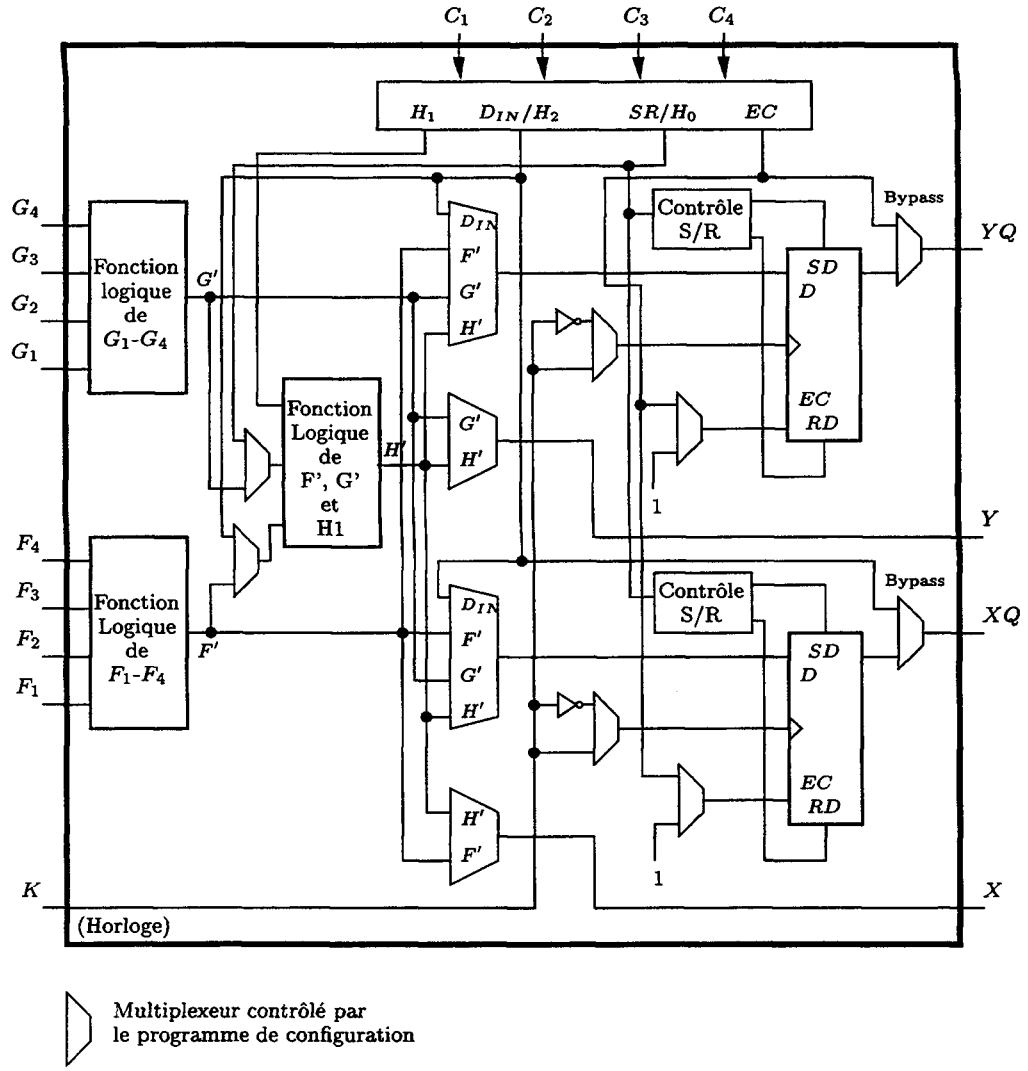


FIG. 4.2: Structure simplifiée d'un CLB de la série XC4000E/X [Xil99]

Dans notre méthode, nous caractérisons les opérateurs et les unités de contrôle et nous estimons la surface de l'implantation en termes du nombre de CLB, de générateurs de fonction F/G et/ou de bascules DFF.

4.3.2 Caractérisation des sommets

La surface correspondante au chemin de données est divisée en deux parties par les algorithmes traditionnels d'estimation [CKGJ97] : la surface active et la surface de câblage. La surface active correspond aux unités fonctionnelles, aux unités de stockage et aux unités de communication. La surface de câblage correspond à l'espace de routage. Cesário et al. [CKGJ97] ont montré que la surface de câblage est un facteur à peu près constant (autour de 70 %) de la surface totale de l'implantation.

La valeur de ce facteur dépend surtout de la technologie de l'architecture-cible et de l'outil de routage. L'estimation de la surface du contrôleur est un peu plus difficile, puisqu'il est fréquemment décrit sous la forme d'une FSM dont les états sont codés de façon symbolique, c'est-à-dire on ne connaît pas les détails de son implantation. Gajski et al. [GDWL92] présentent une technique d'estimation de la taille du contrôleur.

Ainsi, l'estimation de la surface occupée par un opérateur *op* peut être obtenue par l'équation suivante :

$$S_{op} = \alpha \cdot (S_{op_d} + S_{op_c}) \tag{4.1}$$

où

S_{op} : surface de l'opérateur,
α : facteur de surface en fonction du routage,
S_{op_d}: surface de la partie chemin de données,
S_{op_c}: surface de la partie chemin de contrôle.

Dans notre cas, pour effectuer le calcul de la latence des opérateurs, il faut les séparer en deux groupes : les opérateurs purement combinatoires (*CONSTANTE*, *RÉSULTAT*, *CALCUL*, *IMPLODE*, *EXPLODE*, *FORK*, *DIFFUSION* et *DIFFUSION*$^{\infty}$) et les opérateurs contenant des registres (*JOIN*, *ITERATE*, *FORK*$^{\infty}$, *JOIN*$^{\infty}$ et *ITERATE*$^{\infty}$). Le fonctionnement des opérateurs du premier groupe n'est pas synchronisé ou commandé par un signal d'horloge. Ainsi, nous définissons la latence de ces opérateurs comme le temps maximal entre la modification de ses entrées et la stabilisation de ses sorties. Le fonctionnement des opérateurs du deuxième groupe est déjà plus complexe, à cause de leurs registres internes, pilotés par un signal d'horloge.

L'estimation de la latence d'un opérateur *op* peut être obtenue par l'équation suivante :

$$L_{op} = L_{op_C} + L_{op_{wrR}} + L_{op_{rdR}} \tag{4.2}$$

où

L_{op} : latence de l'opérateur,
L_{op_C} : latence de la partie combinatoire,
$L_{op_{wrR}}$: latence de lecture sur registre,
$S_{op_{rdR}}$: latence d'écriture sur registre.

Opérateur *CONSTANTE*

L'implantation de l'opérateur *CONSTANTE* (K), qui produit en sortie une valeur constante T de dimension d, codée sur p bits, correspond à $(d \times p)$ lignes de communication. Ainsi, sa surface (S_K) , en nombre de CLB, F/G ou DFF, est nulle et sa latence (L_K) est négligeable.

$$S_K = 0$$

$$L_K \approx 0$$

Opérateur *RÉSULTAT*

L'implantation de l'opérateur *RÉSULTAT* (R), qui reçoit en entrée une valeur T de dimension d, codée sur p bits, correspond à $(d \times p)$ lignes de communication. Ainsi, sa surface (S_R), en nombre de CLB, F/G ou DFF, est nulle et sa latence (L_R) est négligeable.

$$S_R = 0$$

$$L_R \approx 0$$

Opérateur *CALCUL*

Pour chaque sommet du type *CALCUL* (Cal) du graphe algorithmique, il faut que le concepteur effectue lui-même une estimation de la surface et de la latence de l'opérateur équivalent, à partir des données en entrée et en sortie, et du type d'opération à être implantée. Dans la plupart des cas, la performance temporelle d'un opérateur ne peut pas être estimée avec une bonne précision avant son implantation matérielle, puisque la latence est affectée par des retards dûs au routage, qui sont inconnus avant l'implantation. Pourtant, l'estimation des performances temporelles d'un additionneur est possible, grâce à la logique dédiée au traitement des retenus existante dans les FPGA des série XC4000E/L/XL. Le chemin de retenu dans un additionneur va utiliser des interconnexions spécifiques entre les CLB. Ainsi, la latence d'un additionneur peut être facilement calculée à partir des spécifications du composant utilisé.

La surface occupée par un additionneur (S_{add}) de N bits, en nombre de FG, correspond à :

$$S_{add(FG)} = S_{add_{d(FG)}} = N + 1 \tag{4.3}$$

En nombre de CLB et de DFF, la surface de l'additionneur est nulle : $S_{add_{d(CLB)}} = 0$ et $S_{add_{d(DFF)}} = 0$. La surface de la partie chemin de contrôle est aussi nulle $S_{add_c} = 0$.

La latence d'un additionneur (L_{add}) de N bits [New96], implanté sur un FPGA de la série XC4000XL-3, peut être calculée par :

$$L_{add} = L_{add_C} = 0,13 \cdot N + 4,98\ ns \tag{4.4}$$

La surface occupée par un multiplicateur (S_{mul}) de N bits, en nombre de F/G, correspond à :

$$S_{mul(FG)} = N^2 + N + 1 \tag{4.5}$$

En nombre de DFF, la surface du multiplicateur est nulle : $S_{mul_{d(DFF)}} = 0$. En nombre de CLB, sa surface correspond à :

$$S_{mul_{d(CLB)}} = \left\lceil \frac{N^2 + N + 1}{2} \right\rceil + 1$$

où $\lceil (x) \rceil$ représente la partie entière de x.

La latence d'un multiplicateur (L_{mul}) de N bits, implanté sur un FPGA de la série XC4000XL-3, peut être calculée par :

$$L_{mul} = L_{mul_C} = 0,13 \cdot N^2 + 8,05 \cdot N + 6,58\ ns \tag{4.6}$$

Selon *Xilinx*, l'implantation d'un additionneur de 16 bits sur un FPGA de la famille *XC4000*, par exemple, exige une surface de 9 CLB et une latence de 20,5 *ns* (cf. [Xil94], p. 2-11). L'implantation d'un multiplicateur parallèle de 8 bits sur un FPGA de la famille *XC4000* exige une surface de 73 CLB et une latence de 37 *ns* (cf. [Xil96], p. 4-7).

Opérateur *IMPLODE*

L'implantation de l'opérateur *IMPLODE* (M), qui reçoit en entrée d_1 données T'_1 à T'_{d_1}, de dimension d_2, codées sur p bits et qui regroupe en sortie les d_1 bus unidirectionnels sous la forme $[1..d_1]T'$, correspond à $(d_1 \times d_2 \times p)$ lignes de communication.

Ainsi, sa surface (S_M), en nombre de CLB, F/G ou DFF, est nulle et sa latence (L_M) est négligeable.

$$S_M = 0$$

$$L_M \approx 0$$

Opérateur *EXPLODE*

L'implantation de l'opérateur *EXPLODE* (X), qui reçoit en entrée les données $[1..d_1]T'$ et les décompose en sortie sous la forme de d_1 données de dimension d_2, codées sur p bits, correspond à $(d_1 \times d_2 \times p)$ lignes de communication. Ainsi, sa surface (S_X), en nombre de CLB, F/G ou DFF, est nulle et sa latence (L_X) est négligeable.

$$S_X = 0$$

$$L_X \approx 0$$

Opérateur *FORK*

Comme nous avons vu dans la section 3.4, l'implantation de l'opérateur *FORK* (F), qui reçoit des données $[1..d_1]T'$ en entrée et énumère en sortie des données T'_i de dimension d_2, codées sur p bits, correspond à la combinaison d'un transcodeur générant $\log_2 d_1$ adresses de sélection (nous considérons que le transcodeur est neutre, c'est-à-dire l'ordre des adresses reste inchangé), d'un opérateur *EXPLODE* (X) et d'un multiplexeur $d_1 : 1$, formé par d_1 interrrupteurs (voir figure 3.8). La surface du *FORK* ($S_{F_{CLB}}$), en nombre de CLB, sera déterminée par la surface du multiplexeur, puisque les surfaces de l'*EXPLODE* et du transcodeur neutre sont nulles. Selon *Xilinx*, l'implantation d'un multiplexeur 16 : 1 sur un FPGA de la famille $XC4000$, par exemple, exige 5 CLB et une latence de 16 *ns* (cf. [Xil94], p. 2-11 et [Xil96], p. 4-10).

Chaque CLB peut implanter un multiplexeur à quatre entrées, en utilisant un des générateurs de fonction (F ou G) pour les données et l'autre pour l'adresse de sélection. Ainsi, la surface du *FORK*, en nombre de CLB, est obtenue par l'équation suivante :

$$S_{F(CLB)} = S_{F_d(CLB)} = \sum_{i=1}^{pr} int\left(\frac{d_1 \cdot d_2 \cdot p}{4^i}\right) + 1 \tag{4.7}$$

où

d_1 : dimension du vecteur de données en entrée,
d_2 : dimension des éléments du vecteur de données en entrée,
p : nombre de bits de codage des éléments du vecteur de données en entrée,
pr : profondeur de l'arbre formée par les CLB.

$$pr = \begin{cases} log_4(d_1 \cdot d_2 \cdot p + 1) - 1, & si\ \log_4(d_1 \cdot d_2 \cdot p + 1) \in N \\ \lceil log_4(d_1 \cdot d_2 \cdot p + 1) \rceil, & si\ \log_4(d_1 \cdot d_2 \cdot p + 1) \notin N \end{cases}$$

où $\lceil (x) \rceil$ représente la partie entière de x et

$$int(x) = \begin{cases} \lceil x \rceil, & si\ x \in N \\ \lceil x \rceil + 1, & si\ x \notin N \end{cases}$$

Le *FORK* est un opérateur combinatoire, n'utilisant donc aucun registre. Sa surface ($S_{F_{DFF}}$), en nombre de DFF, est nulle.

$$S_{F(DFF)} = 0$$

En termes de générateurs de fonction F/G, sa surface ($S_{F(FG)}$) correspond à :

$$S_{F(FG)} = S_{F_d(FG)} = 2 \cdot \left(\sum_{i=1}^{pr} int\left(\frac{d_1 \cdot d_2 \cdot p}{4^i}\right) + 1\right) \tag{4.8}$$

La latence de l'opérateur *FORK*, en nanosecondes, est une fonction du nombre de couches de CLB traversées, déterminé par la profondeur pr de l'arbre de CLB :

$$L_F = L_{F_G} = pr \cdot (T_{ITO} + T_{INT}) - T_{INT} \tag{4.9}$$

où T_{ITO} correspond à la latence entre les entrées des générateurs de fonction F et G et les sorties X et Y, à travers H' et T_{INT} correspond au retard moyen d'interconnexion

entre deux CLB. Dans notre cas, pour un FPGA *Xilinx* de la série XC4000XL(-3), $T_{ITO} = 2,7\ ns$ et $T_{INT} = 1,2\ ns$[2]. Ainsi,

$$L_F = 3,9 \cdot pr - 1,2 \tag{4.10}$$

Opérateur *JOIN*

Comme nous avons vu dans la section 3.4, l'implantation de l'opérateur *JOIN*, qui reçoit en entrée d_1 données T_i' de dimension d_2, codées sur p bits, et les collecte en sortie sous la forme d'un vecteur $[1..d_1]T'$, correspond à un transcodeur générant $\log_2 d_1$ adresses de sélection (nous considèrons que le transcodeur est neutre), à un démultiplexeur $1 : d_1$ et un registre de $(d_1 - 1)$ cases. La surface occupée par un *JOIN* (si le transcodeur est neutre) correspond donc à la surface du démultiplexeur (S_{DEMUX}) plus la surface du registre (S_{REG}) :

$$S_J = S_{DEMUX} + S_{REG} \tag{4.11}$$

Dans la série de FPGA *Xilinx XC4000*, il existe deux façons différentes d'effectuer le stockage de données : (a) un CLB peut tout simplement être configuré comme deux registres de 1 bit ou (b) un CLB peut être configuré comme des cellules-mémoire RAM de type 16×2 bits ou 32×1 bit (voir [Xil94], p. 2-13). En retenant, évidemment, l'alternative (b), configurée sous la forme 32×1 bit, la surface occupée par un registre ($S_{REG(CLB)}$), en nombre de CLB, est alors obtenue par :

$$S_{REG(CLB)} = S_{REG_d(CLB)} = int\left(\frac{p \cdot (d_1 - 1) \cdot d_2}{32}\right)$$

En termes de F/G, la surface du registre est nulle ($S_{REG(FG)} = 0$) et, en termes de DFF, sa surface correspond à :

$$S_{REG(DFF)} = p \cdot (d_1 - 1) \cdot d_2$$

Dans la série de FPGA *Xilinx XC4000*, un CLB peut implanter soit une fonction logique de cinq entrées, soit deux fonctions logiques de jusqu'à quatre entrées chacune. Les signaux de validation des bascules ($en_0, \ldots, en_{d_1-1}$), qui composent le registre de l'opérateur *JOIN*, sont générés par une logique composée de

[2]Les retards d'interconnexion ne sont pas fournis par le *data sheet*, variant entre 0,1 *ns* (entre deux CLB voisins connectés par un lien direct) et plus que 20 *ns* (si le signal doit traverser plusieurs points d'interconnexion intermédiaires) [Alf98]. Tout au long de ce travail, nous considérons la valeur de 1,2 *ns* comme le retard d'interconnexion typique [New96].

$(d_1 - 1)$ portes logiques *ET*. Le démultiplexeur est donc implanté par cette logique de sélection de bascules. La surface occupée par le démultiplexeur ($S_{DEMUX(FG)}$), en nombre de générateurs de fonction F/G, est obtenue par :

$$S_{DEMUX(FG)} = d_1 - 1$$

En termes de bascules DFF, la surface du démultiplexeur est nulle ($S_{DEMUX(DFF)} = 0$) et, en termes de CLB, sa surface correspond à :

$$S_{DEMUX(CLB)} = int\left(\frac{d_1 - 1}{2}\right)$$

Ainsi, en reprenant l'éq. 4.11, la surface de l'opérateur *JOIN* ($S_{J_{CLB}}$), en nombre de CLB, est obtenue par l'équation suivante :

$$S_{J(CLB)} = S_{J_d(CLB)} = max\left(int\left(\frac{p \cdot (d_1 - 1) \cdot d_2}{32}\right), int\left(\frac{d_1 - 1}{2}\right)\right) \qquad (4.12)$$

Pendant la phase de stockage des données sur l'opérateur *JOIN*, sa latence ($L_{J_{wr}}$) correspond à la latence de la logique de génération des adresses du démultiplexeur (T_{ITO}), plus le temps d'écriture des données sur le registre (T_{WOTS}), plus le temps d'interconnexion entre deux CLB (T_{INT}) :

$$L_{J_{wr}} = T_{ITO} + T_{WOTS} + T_{INT} \qquad (4.13)$$

où $T_{WOTS} = 8,1\ ns$, dans le cas d'une RAM 32 × 1 implantée sur un FPGA de la série XC4000XL(-3). Ainsi,

$$L_{J_{wr}} = 13,8\ ns \qquad (4.14)$$

Pendant la phase de lecture des données sur l'opérateur *JOIN*, sa latence ($L_{J_{rd}}$) correspond au temps de lecture des données sur le registre (T_{RCT}), qui est égal à 6,5 *ns* dans le cas d'une RAM 32 × 1 implantée sur un FPGA de la série XC4000XL-3.

$$L_{J_{rd}} = 6,5\ ns \qquad (4.15)$$

Opérateur *ITERATE*

Comme nous avons vu dans la section 3.4, l'implantation de l'opérateur *ITERATE* (I), en recevant en entrée une séquence de d_1 données de dimension d_2, codées sur p bits, correspond à un transcodeur (TC), qui reçoit $\log_2 d_1$ adresses en entrée et génère 2 adresses en sortie (où d_1 correspond au nombre de répétitions du motif répétitif factorisé par l'opérateur *ITERATE*) ; un multiplexeur 2 : 1 (MUX) et un registre d'une case (REG). Le transcodeur TC doit générer une adresse (adr) d'un bit pour le multiplexeur. Si $adr = 0$, le multiplexeur sélectionne l'entrée e_0 ; sinon, il sélectionne l'entrée e_i. Nous supposons, comme pour les opérateurs F et J, que le transcodeur est neutre. L'adresse adr correspond donc au bit le moins significatif de cpt.

Un CLB d'un FPGA *Xilinx* de la série XC4000 peut implanter un opérateur *ITERATE* de 2 bits, en utilisant une bascule DFF et un générateur de fonction F ou G pour traiter chaque bit. La surface ($S_{I(CLB)}$), en nombre de CLB, occupée par un *ITERATE*, peut être obtenue par :

$$S_{I(CLB)} = int\left(\frac{d_2 \cdot p}{2}\right) \tag{4.16}$$

En nombre de générateurs de fonctions F/G et de bascules DFF, la surface de l'opérateur I ($S_{I(FG)}$ et $S_{I(DFF)}$) correspond à :

$$S_{I(FG)} = S_{I(DFF)} = d_2 \cdot p$$

Il existe trois valeurs de latence pour l'opérateur *ITERATE* : celle concernant la valeur initiale e_0, nommée L_{I_0} ; celle concernant les valeurs intermédiaires $s_i \neq e_0$, nommée L_{I_i} ; et celle concernant la valeur finale s_f, nommée L_{I_f}. Ci-dessous nous caractérisons l'implantation de l'*ITERATE* sur un FPGA de la série XC4000XL(-3) :

$$L_{I_0} = T_{ITO} = 2,7\ ns \tag{4.17}$$

où T_{ITO} correspond à la latence entre les entrées des générateurs de fonction F et G et les sorties X et Y, à travers H'.

$$L_{I_i} = T_{CKO} + T_{ITO} = 4,8\ ns \tag{4.18}$$

où T_{CKO} correspond au temps de lecture de la bascule DFF, après un front montant de l'horloge ($T_{CKO} = 2,1\ ns$) et T_{ITO} correspond à la latence entre les entrées des générateurs de fonction F et G et les sorties X et Y, à travers H'.

$$L_{I_f} = 0\,ns \tag{4.19}$$

Opérateur *DIFFUSION*

L'implantation de l'opérateur *DIFFUSION* (D), qui effectue des connexions directes entre son entrée et sa sortie, exige une surface (S_D) nulle. Sa latence (L_D) correspond au temps de communication entre l'entrée et la sortie, qui est une valeur négligeable.

$$S_D = 0$$

$$L_D \approx 0$$

Opérateur $FORK^{\infty}$

Dans un système embarqué temps réel, l'opérateur $FORK^{\infty}$ (F^{∞}) correspond à un capteur, qui est externe au FPGA. Ainsi, la surface ($S_{F^{\infty}}$) occupée par cet opérateur est nulle et sa latence ($L_{F^{\infty}}$) correspond au temps de communication entre le *pad* d'entrée et le(s) opérateur(s) en aval du $FORK^{\infty}$ (T_{PLI} : temps de propagation entre un *pad* et I_1 ou I_2 à travers un *latch* = 3,7 *ns*).

Dans nos exemples, nous implantons le F^{∞} sur un FPGA de la série XC4000XL, sous la forme d'un registre qui fournit des données de dimension d_1, codées sur p bits. Ainsi, sa surface ($S_{F^{\infty}}$) correspond aux bascules DFF appartenant aux IOB :

$$S_{F^{\infty}} = d_1 \cdot p \tag{4.20}$$

Sa latence ($L_{F^{\infty}}$) correspond au temps de lecture de la bascule DFF (T_{IKRI}) :

$$L_{F^{\infty}} = T_{IKRI} = 1,7\,ns \tag{4.21}$$

Opérateur $JOIN^{\infty}$

Dans un système embarqué temps réel, l'opérateur $JOIN^{\infty}$ (J^{∞}) correspond à un actionneur, qui est, comme le F^{∞}, externe au FPGA. Ainsi, la surface occupée par

cet opérateur (S_{J^∞}) est nulle et sa latence (L_{J^∞}) correspond au temps de communication entre l'opérateur en amont du $JOIN^\infty$ et le *pad* de sortie (T_{OPF} : temps de propagation entre la sortie OUT et un pad = 4,1 ns).

Dans nos exemples, nous implantons le J^∞ sur un FPGA de la série XC4000XL, sous la forme d'un registre qui stocke des données de dimension d_1, codées sur p bits. Ainsi, sa surface (S_{J^∞}) correspond aux bascules DFF appartenant aux IOB :

$$S_{J^\infty} = d_1 \cdot p \tag{4.22}$$

Sa latence (L_{J^∞}) correspond au temps d'écriture sur la bascule DFF (T_{OOK}) = 0,5 ns).

$$L_{J^\infty} = T_{OOK} = 0,5\ ns \tag{4.23}$$

Opérateur *ITERATE*$^\infty$

L'opérateur *ITERATE*$^\infty$ (I^∞), correspond à un registre implanté à l'intérieur d'un FPGA, en utilisant les bascules DFF des IOB. Ainsi, la surface (S_{I^∞}), en nombre de CLB, occupée par l'*ITERATE*$^\infty$ est nulle. Elle est exprimée en termes du nombre de bascules DFF. Sa latence (L_{I^∞}) correspond au temps d'écriture sur ces bascules. Ci-dessous nous caractérisons l'implantation de l'*ITERATE*$^\infty$ sur un FPGA de la série XC4000XL.

$$S_{I^\infty} = d_1 \cdot p \tag{4.24}$$

$$L_{I^\infty} = T_{OOK} = 0,5\ ns \tag{4.25}$$

Opérateur *DIFFUSION*$^\infty$

L'implantation de l'opérateur *DIFFUSION*$^\infty$ est identique à celle de l'opérateur *DIFFUSION*, c'est-à-dire sa surface est nulle et sa latence est négligeable.

$$S_{D^\infty} = 0$$

$$L_{D^\infty} \approx 0$$

4.3.3 Caractérisation des unités de contrôle

Les unités de contrôle coordonnent le fonctionnement des opérateurs frontières de factorisation, ne faisant pas partie du chemin de données. Ainsi, les performances temporelles de ces unités ne sont pas considérées pendant l'estimation de la latence de l'implantation matérielle. Pourtant, il faut tenir compte de la surface qu'elles occupent, au moment du calcul de la surface équivalente de l'implantation matérielle.

Unité de contrôle finie

Comme nous avons vu précédemment (section 3.9.2), l'unité de contrôle (UC) associée à chaque frontière de factorisation finie est composée d'un composant *COMPTEUR* (décrit dans la section 3.6) et d'une logique supplémentaire. Le composant *COMPTEUR*, du type *one-hot encoding* (voir figure 3.18) aura autant de registres (bascules DFF) que le nombre de répétitions (d) du motif répétitif délimité par la frontière qu'il pilote. La logique supplémentaire (une quinzaine de portes logiques) peut être estimée à quatre générateurs de fonction F/G. Les unités de contrôle sont interconnectées pour former le chemin de contrôle, par l'intermédiaire de portes *ET*, qui réalisent la conjonction des signaux d'acquittement et de requête. Cette logique d'interconnexion, associée à chaque unité de contrôle, peut être estimée, en termes du nombre de générateurs de fonction F/G, à (n *div* $4 + 1$), où n correspond au nombre d'entrées des portes *ET* et *div* fournit le résultat de la division entière. Ainsi, la surface exigée par chaque unité de contrôle finie ($S_{UCF_{CLB}}$), en termes de CLB, peut être estimée à :

$$S_{UCF(CLB)} = S_{REG} + S_{LS} + S_{LI} \tag{4.26}$$

où

$$S_{REG} = d \; div \; 2 + 1$$

$$S_{LS} = 2$$

$$S_{LI} = (n_{rsu} + n_{rfu} + n_{afd} + n_{asd}) \; div \; 8 + 1$$

n_{rsu} : nombre de frontières productrices du côté "lent",
n_{rfu} : nombre de frontières productrices du côté "rapide",
n_{afd} : nombre de frontières consommatrices du côté "rapide",
n_{asd} : nombre de frontières consommatrices du côté "lent".

En termes du nombre de registres (bascules DFF), la surface d'une unité de contrôle finie correspond au nombre de répétitions du motif répétitif :

$$S_{UCF(DFF)} = d$$

et, en termes du nombre de générateurs de fonction F/G, elle correspond à :

$$S_{UCF(FG)} = (n_{rsu} + n_{rfu} + n_{afd} + n_{asd}) \; div \; 4 + 1.$$

Unité de contrôle infinie

L'implantation d'une unité de contrôle infinie (voir figure 3.34) n'exige aucun registre. Sa surface correspondra donc aux portes *ET* qui réalisent la conjonction des signaux d'acquittement (afd) et de requête (rfu) du côté "rapide" de la frontière. En termes du nombre de CLB, la surface ($S_{UCI(CLB)}$) est :

$$S_{UCI(CLB)} = (n_{rfu} + n_{afd}) \; div \; 8 + 1 \tag{4.27}$$

En nombre de bascules DFF, sa surface est nulle. En termes du nombre de générateurs de fonction F/G, sa surface correspond à :

$$S_{UCI(FG)} = (n_{rfu} + n_{afd}) \; div \; 4 + 1.$$

4.4 Estimation des ressources et des performances

La qualité de l'estimation des ressources matérielles nécessaires à l'implantation matérielle sous la forme d'un circuit d'une spécification algorithmique et de ses performances temporelles dépend du compromis entre la précison envisagée et le temps de calcul consacré à ces estimations. Madsen et al. [Mad*97] recommandent que, pendant la phase d'exploration de l'espace des solutions, l'estimation soit plus rapide que dans la phase d'implantation finale, par conséquent, moins précise. Ainsi, les méthodes d'estimation plus précises et plus lentes sont utilisées avant l'implantation finale, après avoir choisi les transformations à appliquer sur la spécification. Évidemment, les mesures les plus précises sont obtenues après la synthèse de l'implantation et son exécution sur l'architecture-cible. Cette méthode est pourtant inapplicable, puisque, pour être efficace, elle exigerait son utilisation pour toutes les solutions possibles, impliquant un temps de calcul très important. En conclusion, les valeurs fournies par les estimateurs pendant la phase d'exploration doivent renseigner le concepteur sur la qualité des solutions envisagées, afin de prédire les résultats

de l'implantation matérielle finale, sans pour autant qu'il soit obligé d'aller jusqu'à l'exécution de l'application sur l'architecture-cible. Nous nous intéressons à une estimation rapide de l'architecture [RBI95].

Alves de Barros [Alv94] a proposé un modèle d'évaluation de coût en surface d'une architecture dédiée sur FPGA, qui utilise comme référence de mesure un ensemble M de composants architecturaux élémentaires. Ces composants doivent être caractérisés *a priori* en termes de surface et vitesse, constituant ainsi une bibliothèque de base pour le modèle d'évaluation. Les composants élémentaires vont varier en fonction de l'application-cible, pouvant être des additionneurs, des soustracteurs, des multiplicateurs, des comparateurs, etc. Comme nous avons vu dans la section 4.3.2, nous avons utilisé dans ce travail une stratégie similaire, caractérisant tous les opérateurs de base ou de factorisation obtenus après traduction matérielle du graphe algorithmique correspondant à la spécification.

Notre méthode d'estimation des performances de l'implantation matérielle d'une spécification algorithmique consiste à déterminer la surface (en nombre de CLB, de générateurs de fonction F/G ou de bascules DFF) et la latence de cette implantation sur des FPGA *Xilinx* de la série XC4000XL-3. L'estimation des performances est effectuée à partir du graphe algorithmique de l'application qui doit avoir ses sommets et ses arcs étiquetés respectivement avec les valeurs de surface et de latence des opérateurs correspondant aux sommets et avec le temps de communication correspondant aux transferts de données représentés par les arcs.

La performance temporelle maximale d'un système, en termes de latence et/ou de cadence, est déterminée par la fréquence maximale de l'horloge principale. Celle-ci, à son tour, est une fonction de la longueur du chemin critique, déterminée par le plus grand chemin combinatoire entre deux registres d'un circuit synchrone[3]. L'estimation des performances temporelles avant l'implantation matérielle peut être très difficile à cause de l'influence des outils de placement et routage sur le chemin critique. Les ressources d'interconnexion programmables, utilisées pour connecter les E/S et les blocs logiques, introduisent des retards, parfois difficilement prévisibles dans les chemins du circuit [Xil97].

4.4.1 Modèles temporel et de surface

Une bonne estimation des performances temporelles et de la surface d'une implantation matérielle est essentielle pour guider, de façon efficace, l'exploration de l'espace de solutions par les algorithmes d'optimisation de l'implantation. La méthode d'estimation doit se baser, de préférence, sur des modèles simples, capables de représenter tous les types de composants présents dans un graphe matériel.

Le modèle temporel que nous avons développé est basé sur une technique de modélisation automatique de circuits logiques hiérarchiques, applicable aux circuits

[3]Circuit contenant des registres internes pilotés par un signal d'horloge.

synchrones, ne comprenant pas les opérateurs multiphases[4]. Comme ce modèle est très générique, il peut être appliqué à n'importe quel bloc logique généré par n'importe quel outil de CAO. Il est donc particulièrement intéressant pour la modélisation des IP (*Intellectual Property*) synthétisés par différents outils de CAO. Le modèle temporel repose sur un modèle de graphes appelé *graphe temporel*, qui représente l'ensemble des chemins critiques existant au sein du graphe algorithmique. Notre méthode d'estimation est basée sur la réduction de la taille du graphe temporel, gardant uniquement les arcs qui font partie des chemins critiques de ce graphe.

Afin de présenter notre modèle temporel, nous prenons en exemple le PMV dont sa sortie C est retardée d'un cycle avant d'être combinée avec son entrée B, comme le montre la figure 4.3, où A et B sont les entrées, C est la sortie, PMV correspond à un produit matrice-vecteur et I^{∞} correspond à un retard. Le chemin critique dans un circuit synchrone est le plus long chemin entre : deux registres, une entrée du circuit et un registre, une entrée et une sortie du circuit ou un registre et une sortie du circuit. La figure 4.4 montre le graphe algorithmique complet du PMV plus "retard". Cet exemple illustre les différents types de chemin présents dans un graphe algorithmique. Nous commençons par la création d'un graphe représentant la structure de l'implantation.

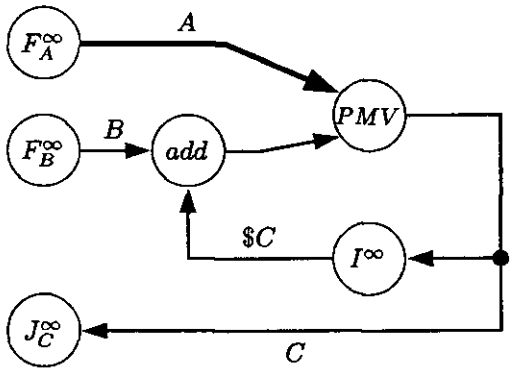


FIG. 4.3: PMV plus "retard"

Graphe de la structure de l'implantation

À partir d'un graphe matériel contenant des composants combinatoires[5] et séquentiels, nous pouvons construire un graphe correspondant à sa structure, où : les sommets E_i représentent ses entrées, les sommets S_i représentent ses sorties, chaque composant combinatoire correspond à un sommet C_i de m entrées et n sorties, chaque connection entre deux composants combinatoires (C_i et C_j) correspond à un arc entre deux sommets ($A(C_i, C_j)$), tous les sommets E_i sont connectés à un sommet *in* et tous les sommets S_i sont connectés à un sommet *out*. Le circuit de la

[4]Opérateurs synchrones dont les registres internes sont pilotés par différents signaux d'horloge.
[5]Composants qui ne contiennent pas de registres internes.

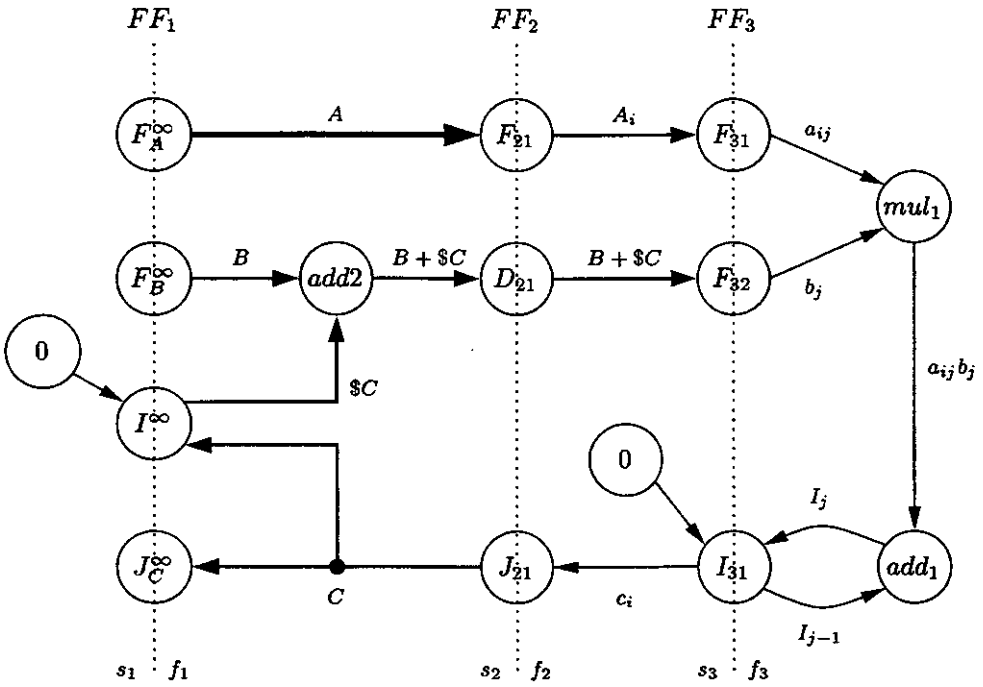


FIG. 4.4: Graphe algorithmique du PMV plus "retard"

figure 4.4 peut donc être modélisé par le graphe structurel[6] de la figure 4.5. Afin de réduire le temps de calcul nécessaire à la recherche du chemin critique de ce graphe, nous verrons par la suite qu'il faut réduire la taille de ce graphe.

Graphe temporel

Le graphe temporel d'un circuit est généré à partir de son graphe structurel, tenant compte des paramètres suivants (voir figure 4.6) :

- $T_{crit}(i, N)$: temps nécessaire pour que les données soient disponibles pour l'$i$ème entrée du sommet N ;
- $T_{IN}(i, N)$: temps qui doit être ajouté à $T_{crit}(N)$ afin d'attendre un registre du sommet N, à partir de sa $i$ème entrée ;
- $T_{OUT}(i, N)$: temps nécessaire pour que les résultats soient disponibles pour l'$i$ème sortie du sommet N. Cette valeur sera ajoutée à $T_{crit}(N)$, s'il existe un chemin combinatoire entre les sommets in et N ;
- $T_{Min}(clk)$: valeur minimale de la période de l'horloge. Cette valeur est fondamentale pour les opérateurs pipelinés.

[6] Le graphe structurel est une variante du graphe matériel, qui ne contient pas la partie chemin de données.

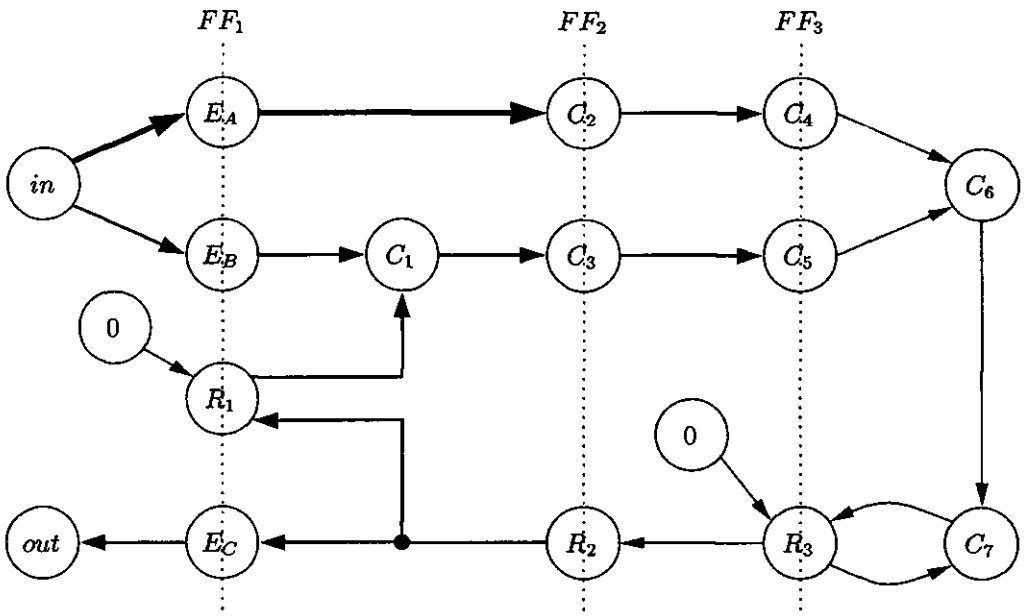


FIG. 4.5: Graphe structurel du PMV plus "retard"

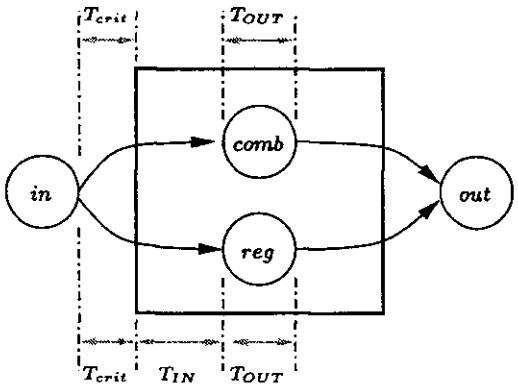


FIG. 4.6: Paramètres du graphe temporel

Les paramètres ci-dessus varient en fonction du type de sommet (combinatoire ou séquentiel). Ainsi, pour un sommet combinatoire, nous avons : $T_{IN}(i, Comb) = 0$, puisqu'il ne contient pas de registre; et $T_{OUT}(i, comb) = Max(T_{crit}(i, comb)) + T_{Lat_{comb}}$; $T_{Min}(clk) = 0$, puisque *comb* n'est pas pipeliné. Pour un sommet séquentiel, nous avons : $T_{IN}(i, Reg) = T_{setup}(reg)$, où $T_{setup}(reg)$ est le temps de *setup* du registre; $T_{IN}(clk, Reg) = 0$, pour l'entrée clk; $T_{Min}(clk) = 0$, puisque reg n'est pas pipeliné; et $T_{OUT}(i, reg) = T_{hold}(reg) + T_{crit}(clk, reg)$, où $T_{hold}(reg)$ correspond au temps de *hold* du registre.

Réduction du graphe temporel

Le but du graphe temporel est de stocker seulement les différents chemins critiques du graphe. Cette réduction, réalisée par l'intermédiaire d'algorithmes de recherche du chemin minimal, est guidée par les règles suivantes :

- deux chemins combinatoires acycliques peuvent être combinés (additionnées) entre eux ;
- un registre placé entre deux chemins combinatoires interrompt le flot combinatoire de la façon suivante : le premier chemin finit à l'entrée du registre et le deuxième commence à la sortie du registre.

L'Algorithme 3 décrit la transformation du graphe temporel, selon les règles mentionnées ci-dessus. La figure 4.7 montre les résultat de la transformation du graphe de la figure 4.5.

Algorithme 3 Transformation du graphe temporel

Requiert : Graphe temporel (G_T)
Calcule : Graphe temporel transformé (G'_T)
begin
Parcourrir G_T du sommet *in* jusqu'au sommet *out* ;

Pour tous les sommets contenant des registres ($T_{IN}(i, N) \geq 0$), créér un arc entre le registre N et un sommet $CREG$;

Pour tous les chemins qui partent d'un registre (c'est-à-dire $T_{OUT}(i, N)$ dépend du $T_{crit}(clk, n)$), créér un arc entre $CREG$ et le sommet connecté à la sortie i.

Éliminer les sommets contenant des registres et connecter les sommets en amont des registres à $CREG$.
end

Après cette transformation, nous pouvons réduire le graphe temporel (voir figure 4.8) par l'intermédiaire des algorithmes de recherche du chemin minimal :

- Chemin (1) : $T_{IN}(i, CREG)$ est égal au plus long chemin entre les entrées du graphe et un registre. Cette valeur dependra de $T_{crit}(i, S_1)$;
- Chemin (2) : $T_{OUT}(i, S_C)$ est égal au plus long chemin entre $CREG$ et la sortie de S_C ;
- Chemin (3) : $T_{Min}(CREG)$ est égal au plus long chemin entre deux registres $CREG$ (cette valeur correspond au temps de retard maximum des stages du pipeline).

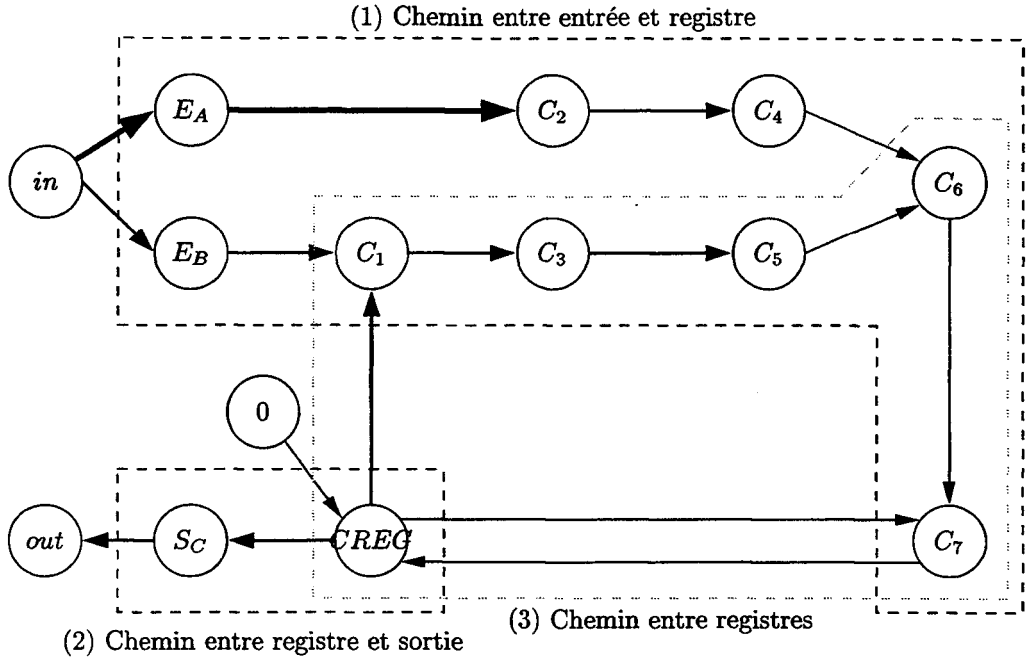


FIG. 4.7: Graphe temporel transformé du PMV plus "retard"

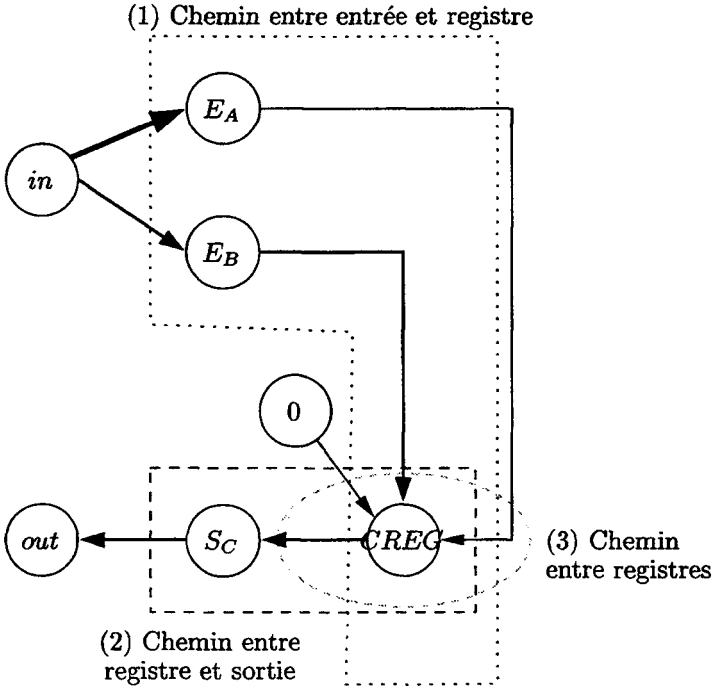


FIG. 4.8: Graphe temporel réduit du PMV plus "retard"

Calcul du chemin critique

Nous considérons le graphe matériel de l'application comme un ensemble de sous-graphes qui effectuent le traitement du flot de données et qui se communiquent

les résultats de ce traitement par l'intermédiaire de registres. Ainsi, les données en entrée d'un sous-graphe proviennent forcément d'un registre et les données en sortie d'un sous-graphe doivent également être stockées dans un registre. La durée du cycle d'horloge d'une implantation matérielle doit donc être égale ou supérieure à la durée du plus long chemin combinatoire entre deux registres quelconques. Ce chemin correspondant au plus long chemin combinatoire dans un graphe matériel est le chemin critique du graphe.

Le calcul du chemin critique de l'implantation matérielle, à partir d'un graphe d'opérateurs correspondant à la traduction matérielle du graphe algorithmique de l'application, permet de fournir au concepteur de l'architecture la prédiction de la fréquence maximale de l'horloge de l'architecture à synthétiser. Ce calcul peut être effectué de façon automatique. À chaque opérateur de ce graphe matériel, nous associons trois étiquettes temporelles, comme le montre la figure 4.9 :

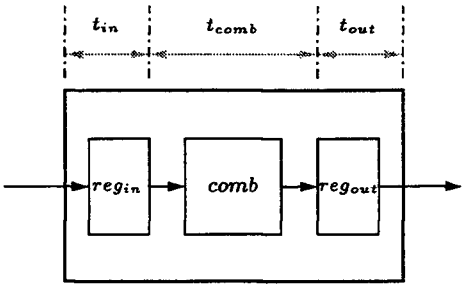


FIG. 4.9: Étiquettes temporelles d'un opérateur

- t_{in} : temps d'entrée, correspond au temps de *setup* de son registre interne. t_{in} est nul si l'opérateur est combinatoire ;
- t_{comb} : temps de latence, correspond à la latence des parties combinatoires de l'opérateur (multiplexeur, calcul, decodeur, etc.) ;
- t_{out} : temps de sortie, correspond au temps de *hold* de son registre interne. t_{out} est nul si l'opérateur est combinatoire.

Selon Aichouchi et al. [AKJ96], le chemin critique (T_{cc}) d'un cycle de base peut être calculé par l'équation suivante :

$$T_{cc} = t_1 + t_2 + t_3 \tag{4.28}$$

où

t_1 : temps de calcul pour l'état suivant et des signaux de contrôle pour le partie chemin de données,

t_2 : temps de calcul des opérations dans le chemin de données,
t_3 : temps de communication entre le chemin de données et le contrôleur (négligeable).

Afin d'adapter l'éq. 4.28 à notre modèle d'implantation matérielle (voir figure 4.10), nous y ajoutons les termes t_{in}, qui correspond au temps de *setup* des registres en amont du chemin de données (registres appartenant aux capteurs dans un système réactif, ou registres formés par les bascules DFF des IOB dans une implantation limitée au FPGA), et t_{out}, qui correspond au temps de *hold* des registres en aval du chemin de données (registres appartenant aux actionneurs dans un système réactif, ou registres formés par les bascules DFF des IOB dans une implantation limitée au FPGA).

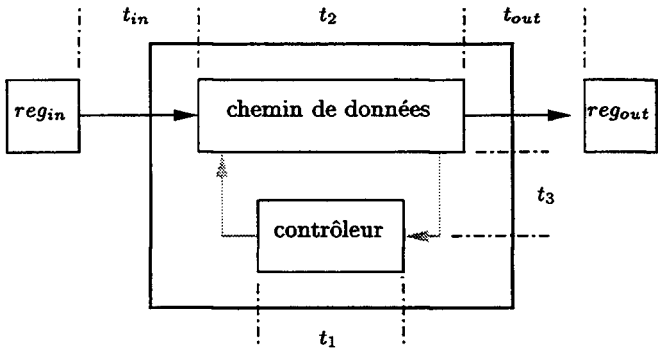


FIG. 4.10: Chemin critique d'un cycle de base

Les registres reg_{in} et reg_{out} intègrent des frontières infinies. Dans le chemin de données, nous trouvons les opérateurs de base combinatoires et les opérateurs de factorisation de motifs de graphe répétitifs infinis. Parmi ces derniers, les opérateurs *FORK* et *DIFFUSION* sont purement combinatoires, mais les opérateurs *JOIN* et *ITERATE* contiennent des registres qui pourraient modifier le chemin critique. Pourtant, ceci n'est pas le cas puisque, pour la dernière itération des frontières contenant les opérateurs *JOIN* et *ITERATE*, les données traversent ces opérateurs sans être stockées dans leurs registres respectifs, comme nous pouvons le constater dans les figure 3.9 et 3.10. Le chemin critique T_{cc} correspondra donc à :

$$T_{cc} = t_{in} + t_2 + t_{out} \tag{4.29}$$

Dans notre méthode, nous pouvons ignorer le temps de calcul pour l'état suivant et des signaux de contrôle pour le chemin de données (t_1), puisque notre contrôleur se compose d'un compteur à décalage (*one hot encoding*) et d'une logique supplémentaire très simple, dont le temps d'exécution est négligeable. Le temps de communication entre le chemin de données et le contrôleur (t_3) peut être également ignoré, car notre stratégie de contrôle délocalisé assure la proximité entre les sous-parties opératives et leurs contrôleurs respectifs. Ainsi, le calcul du chemin critique

d'un graphe matériel consiste à déterminer le plus long chemin entre ses entrées et ses sorties, cf. Algorithme 4. La longueur du chemin critique T_{cc} détermine la valeur minimale de la période de l'horloge globale du circuit (T_H). Ainsi,

$$T_H \geq Tcc.$$

Algorithme 4 Calcul du chemin critique

Requiert : Graphe matériel étiquetté ($G_M = (S_M, A_M)$)
Calcule : Longueur du chemin critique (T_{cc})
begin
Parcourir l'ensemble de sommets de ($G_M = (S_M, A_M)$) et identifier les sommets qui appartiennent à des frontières infinies ($S_I = s_1, \ldots, s_n$) ;

Identifier, dans l'ensemble S_I, les sommets d'entrée (F^∞ et I^∞) et les sommets de sortie (J^∞ et I^∞) ;

Calculer la longueur des chemins entre chaque sommet d'entrée et chaque sommet de sortie ;

Identifier la valeur maximale parmi toutes les longueurs de chemin. Celle-là correspond donc au chemin critique de G_M (T_{cc}).
end

Calcul du nombre de cycles

La connaissance du chemin critique d'un graphe matériel factorisé ne suffit pas pour calculer la latence ou le temps d'exécution de l'implantation correspondante. Il faut aussi connaître le nombre de répétitions de chaque motif répétitif, afin de calculer le nombre de cycles d'horloge nécessaires à l'exécution du graphe matériel. Pour calculer le nombre de cycles d'un graphe matériel factorisé (cf. Algorithme 5), nous analysons les relations entre les différentes frontières de factorisation de ce graphe, représentées sous la forme d'un graphe de relations entre frontières (voir section 2.4.2). Nous distinguons trois types de relations de voisinage entre les frontières, comme le montre la figure 4.11 :

- type $\times$: deux frontières voisines (FF_1 et FF_2) sont en relation du type $\times$, quand il existe au moins une dépendance de données entre le côté "rapide" de FF_1 (f_1) et le côté "lent" de FF_2 (s_2) et/ou entre le côté "lent" de FF_1 (s_1) et le côté "rapide" de FF_2 (f_2). Ainsi, le côté "lent" d'une frontière est adjacent au côté "rapide" de l'autre ou vice-versa. Dans le sous-graphe de la figure 4.11(a), à chaque fois que la frontière FF_1 exécute un cycle, FF_2 est exécutée d_2 fois. FF_1 et FF_2 sont à la fois productrices et consommatrices, l'une par rapport à l'autre. Le nombre de cycles équivalent à l'exécution des deux frontières ($NC_\times$) correspond au produit entre le nombre de répétitions du motif délimité par FF_1 (d_1) et celui délimité par FF_2 (d_2).

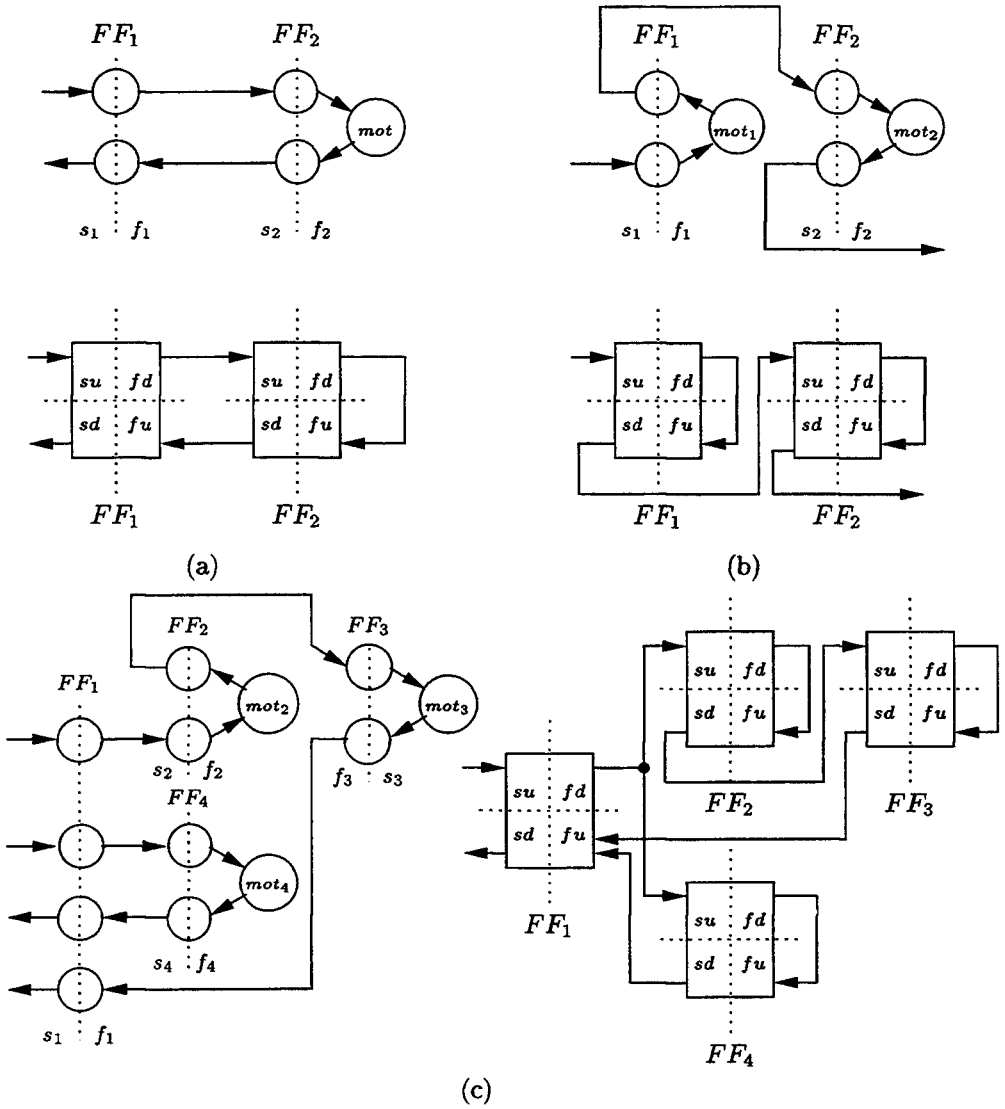


FIG. 4.11: Différents cas de relations de voisinage entre frontières : (a) type $\times$; (b) type $+$; (c) type *max*.

$$NC_{\times} = d_1 \times d_2 \tag{4.30}$$

La généralisation de l'éq. 4.30 pour le cas de n frontières en relation du type $\times$ correspond à :

$$NC_{\times} = d_1 \times d_2 \times \ldots d_n \tag{4.31}$$

- type $+$: deux frontières voisines (FF_1 et FF_2) sont en relation de type $+$, quand il existe au moins une dépendance de données entre le côté "lent" de

FF_1 (s_1) et le côté "lent" de FF_2 (s_2) ou entre le côté "rapide" de FF_1 (f_1) et le côté "rapide" de FF_2 (f_2). Ainsi, le côté "lent" d'une frontière est adjacent au côté "lent" de l'autre ou le côté "rapide" d'une frontière est adjacent au côté "rapide" de l'autre. Une frontière est productrice et l'autre est consommatrice par rapport à la première. Dans ce cas, le nombre de cycles équivalent à l'exécution des deux frontières (NC_+) correspond à la somme du nombre de répétitions du motif délimité par FF_1 (d_1) à celui délimité par FF_2 (d_2), car le flot de données traverse successivement et indépendamment chacune des frontières. La figure 4.11(b) montre le sous-graphe algorithmique de deux frontières de voisinage type +.

$$NC_+ = d_1 + d_2 \tag{4.32}$$

Dans le cas de deux frontières en relation de type + par l'intermédiaire d'un opérateur *ITERATE* (sommets I cascadés), la frontière productrice (FF_1) exécutera un cycle de moins, puisque le résultat de la dernière itération n'est pas stocké dans le registre d'*ITERATE*. Cette situation peut se produire également pour les autres cas où avons une frontière productrice (FF_1) dont le côté "lent" est connecté au côté "lent" de la frontière consommatrice (FF_2), c'est-à-dire *ITERATE* ou *JOIN* suivi de *FORK* ou *DIFFUSION*. Pourtant, ceci est valable si et seulement si toutes les productrices en amont et en aval de FF_2 ont fini de produire au moins un cycle avant la fin du cycle de FF_1. Ainsi, l'éq. 4.32 devient :

$$NC_+ = (d_1 - 1) + d_2 \tag{4.33}$$

Les généralisations des éq. 4.30 et 4.33 pour le cas de n frontières en relation du type + correspondent respectivement à :

$$NC_+ = d_1 + d_2 + \cdots + d_n \tag{4.34}$$

$$NC_+ = (d_1 - 1) + (d_2 - 1) + \cdots + d_n \tag{4.35}$$

- type MAX : N frontières sont en relation de type max, quand elles n'ont pas de dépendances de données directes entre elles, mais elles sont en dépendance de contrôle par l'intermédiaire d'une autre frontière. Une frontière FF_1 peut être en relation de production et de consommation avec plusieurs frontières (par exemple, FF_2, FF_3 et FF_4, comme le montre la figure 4.11). Dans ce cas, il peut avoir plusieurs chemins parallèles entre les entrées et les sorties de FF_1 (FF_2–FF_3 et FF_4, dans notre exemple). Il est donc nécessaire que tous les traitements parallèles aient terminés pour que le flot de données puisse progresser en aval. Ainsi, pour déterminer le nombre de cycles équivalent (NC_{max}) à l'exécution du sous-graphe localisé entre les entrées et les sorties de FF_1, nous

calculons la fonction MAX du nombre de cycles des chemins parallèles. Pour le graphe de la figure 4.11, NC_{max} correspond à :

$$NC_{max} = d_1 \times Max(d_2 + d_3, d_4) \tag{4.36}$$

Algorithme 5 Calcul du nombre de cycles

Requiert : Graphe de relation entre frontières ($G_V = (S_V, A_V)$)
Calcule : Nombre de cycles de l'implantation (NC_{eq})
begin
Identifier les sommets-source dans le graphe G_V ;

Parcourir le graphe G_V et calculer le nombre de cycles entre chaque sommet-source et chaque sommet-puits, en tenant compte des types de relations entre les frontières ;

Identifier la valeur maximale du nombre de cycles. Celle-là correspondra au nombre de cycles de l'implantation.
end

Calcul de la latence

La latence ou temps de exécution (T_{lat}) de l'implantation matérielle est le produit de son nombre de cycles (NC_{eq}) par la période de l'horloge (T_H), exprimée par l'équation suivante :

$$T_{lat} = NC_{eq} \cdot T_H \tag{4.37}$$

où

NC_{eq} : nombre de cycles équivalent,
T_H : période de l'horloge.

Calcul de la surface

Nous utilisons trois paramètres pour calculer la surface d'une implantation matérielle : le nombre de générateurs de fonction F/G, le nombre de CLB et le nombre de bascules DFF. La surface équivalente S_{eq} correspond à la somme des surfaces des opérateurs (S_{op}) et des unités de contrôle (S_{uc}), calculées comme décrit dans la section 4.3.2. Ainsi,

$$S_{eq} = \sum S_{op} + \sum S_{uc} \tag{4.38}$$

L'Algorithme 6 décrit la procédure triviale de calcul de la surface équivalente de l'implantation matérielle d'une application, à partir de ses graphes matériel et de relations entre frontières dûment étiquetés. Le concepteur doit, avant de démarrer la phase d'optimisation, construire une bibliothèque d'opérateurs étiquetés en termes de surface et de latence.

Algorithme 6 Calcul de la surface

Requiert : Graphe matériel étiquetté ($G'_M = (S'_M, A'_M)$), graphe de relations entre frontières étiqueté ($G'_V = (S'_V, A'_V)$)
Calcule : Surface équivalente de l'implantation du graphe matériel sur l'architecture-cible (S_{eq})
begin
Identifier les sommets de G'_M et de G'_V ;

Additionner les étiquettes des sommets de G'_M et de G'_V, en termes du nombre de générateurs de fonction F/G, de CLB et de bascules DFF.
end

4.4.2 Exemple d'estimation de surface et de latence

En partant du graphe de la spécification algorithmique factorisée du PMV entre une matrice de (6×6) éléments et un vecteur de 6 éléments codés sur 3 bits, représenté par la figure 2.26, nous identifions :

- un sommet $FORK^{\infty}$ (F_A^{∞}), de $6 \times 6 \times 3$ bits,
- un sommet $FORK^{\infty}$ (F_B^{∞}), de 6×3 bits,
- un sommet $JOIN^{\infty}$ (J_C^{∞}), de 6×7 bits,
- un sommet *FORK* (F_{21}), de $6 \times 6 \times 3$ bits,
- un sommet *DIFFUSION* (D_{21}), de 6×3 bits,
- un sommet *JOIN* (J_{21}), de 6×7 bits,
- un sommet *DONNEE* (0), de 6 bits,
- deux sommets *FORK* (F_{31} et F_{32}), de 6×3 bits,
- un sommet *ITERATE* (I_{31}), de 6 bits,
- un sommet *CALCUL* : multiplicateur (*mul*), de 3 bits,
- un sommet *CALCUL* : additionneur (*add*), de 6 bits,
- une frontière "infinie" de factorisation (FF_1),
- une frontière "finie" de factorisation (FF_2), réalisant 6 itérations,
- une frontière "finie" de factorisation (FF_3), réalisant 6 itérations.

Pour calculer la surface nécessaire à l'implantation matérielle de ce graphe algorithmique sur un FPGA *Xilinx* de la série XC4000XL-3, il faut caractériser chacun des opérateurs utilisés pour implanter les sommets du graphe algorithmique

comme décrit dans la section 4.3.2. Il faut également caractériser chaque unité de contrôle associée aux frontières de factorisation, à partir du graphe de relations entre frontières, comme décrit dans la section 4.3.3. Le résultat de cette caractérisation est représenté par le tableau 4.1. L'application de l'éq. 4.38 nous donne la surface correspondante à l'implantation matérielle du PMV.

TAB. 4.1: Caractérisation de la spécification totalement factorisée du PMV

SOMMET	SURFACE			LAT.
	Nb. F/G	Nb. CLB	Nb. DFF	(*ns*)
F_A^∞	0	0	108	1,7
F_B^∞	0	0	18	1,7
J_C^∞	0	0	42	0,5
F_{21}	74	37	0	10,5
D_{21}	0	0	0	0,0
J_{21}	5	3	35	13,8
0	0	0	0	0,0
F_{31}	16	8	0	6,6
F_{32}	16	8	0	6,6
I_{31}	6	3	6	4,8
mul	13	7	0	18,8
add	7	0	0	5,8
UC_1	1	0	0	-
UC_2	1	3	6	-
UC_3	1	3	6	-

Pour calculer la latence correspondante à cette implantation, il faut tout d'abord calculer le chemin critique du graphe matériel étiqueté, comme décrit par l'Algorithme 4. Le graphe possède deux sommets d'entrées (F_A^∞ et F_B^∞) et un sommet de sortie (J_C^∞). Ainsi, il existe deux chemins possibles entre les entrées et les sorties : un entre F_A^∞ et J_C^∞, l'autre entre F_B^∞ et J_C^∞ (voir figure 4.12), dont les longueurs[7] correspondent respectivement à 70,9 et 59,2 *ns*. La valeur la plus grande déterminera la période minimale de l'horloge : $T_H \geq 71$ *ns*.

Pour calculer le nombre de cycles de l'implantation (NC_{eq}), nous partons du graphe de relations entre frontières (figure 2.28) et nous appliquons l'Algorithme 5. Les frontières FF_2 et FF_3 sont en relation du type $\times$. NC_{eq} correspond donc au produit entre le nombre de répétitions du motif associé à FF_2 ($d_2 = 6$) et celui associé à FF_3 ($d_3 = 6$) : $NC_{eq} = 36$ cycles. La latence totale de l'implantation du PMV factorisé est obtenue par l'aplication de l'éq. 4.37 : $T_{lat} = 2556$ *ns*.

Le tableau 4.2 nous permet de comparer les valeurs obtenues par notre méthode d'estimation de surface avec les valeurs produites par le logiciel *Leonardo Spectrum*, version 1999.1i, d'*Exemplar Logic*. Nous y vérifions une erreur inférieure à

[7]La valeur de T_{INT} est de 1,2 *ns*, cf. note 2, page 183.

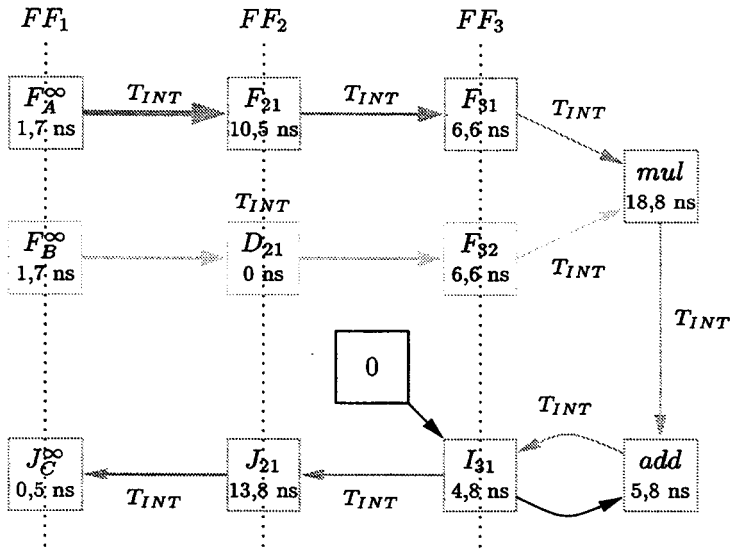


FIG. 4.12: Chemin critique du PMV totalement factorisé

10 % pour la surface et une erreur de 19 % pour la latence. Cette différence est toute à fait acceptable, puisque que les outils de placement et routage effectuent d'autres optimisations, dont les effets vont encore répercuter sur le nombre de ressources utilisées et sur la latence [Xil99].

TAB. 4.2: Surface et latence du PMV

MÉTHODE	SURFACE			LAT. (*ns*)
	Nb. F/G	Nb. CLB	Nb. DFF	
Notre estimateur	140	72	215	71
Leonardo Spectrum, v. 1999.1i	155	76	217	81
ERREUR	-10%	-5%	-1%	-12%

4.5 Optimisation par défactorisations

4.5.1 Définition

Si l'implantation matérielle d'une spécification algorithmique sur l'architecture cible, en tenant compte les contraintes technologiques, ne respecte pas les contraintes temps réel de l'application, il faut transformer le graphe correspondant à la spécification. Cette transformation, appelée *défactorisation*, est la transformation inverse de la factorisation et elle ne change pas la sémantique opératoire d'un GFDD.

4.5.2 Règles de défactorisation

La défactorisation du GFDD cherche à obtenir une implantation plus parallèle, améliorant les performances temporelles au prix de ressources matérielles supplémentaires. Une spécification factorisée peut avoir plusieurs défactorisations partielles : si elle possède n frontières finies de factorisation, il y a au minimum 2^n implantations défactorisées de la spécification. Chaque frontière peut être défactorisée partiellement : une factorisation de d répétitions d'un motif peut se décomposer en f factorisations de d/f répétitions du motif [LS97].

Toutes les défactorisations ne produisent pas forcément des implantations plus parallèles. Dans l'exemple du PMV, la première décomposition (voir figure 2.10) permet une exécution parallèle des produits scalaires, mais la seconde décomposition (voir figure 2.11) n'apporte de parallèlisme supplémentaire qu'entre les produits; les additions sont toutes interdépendantes. La présence d'un sommet *ITERATE* dans une frontière de factorisation indique un parallélisme potentiel réduit par les dépendances inter-motifs (voir figure 2.24).

Les sommets frontières de factorisation de motifs répétitifs sont utilisés pour implanter une spécification factorisée d'un motif répétitif. Une spécification non factorisée correspond à une implantation purement combinatoire qui utilise seulement des opérateurs de base (voir section 3.3). À l'opposé, une spécification factorisée permet, soit une implantation totalement défactorisée sans registres, soit une implantation factorisée identique à la spécification ou partiellement défactorisée avec des registres, multiplexeurs et/ou démultiplexeurs.

La méthode de défactorisation qui nous présentons dans ce chapitre est basée sur les hypothèses suivantes :

- nous partons d'une spécification algorithmique donnée factorisée;
- le graphe algorithmique correspondant à cette spécification contient M frontières de factorisation finies et N frontières de factorisation infinies;
- les frontières infinies (si $N \neq 0$) ne sont pas défactorisables;
- chaque frontière finie est défactorisable par un facteur de défactorisation k, dont les valeurs entières se situent entre 2 et le nombre de répétitions du motif délimité par la frontière.

Contraintes temporelles

L'utilisateur définit la contrainte temporelle (ctr) de l'application en termes de latence. Ensuite, en connaissant la durée de la période d'horloge (T_{clk}) correspondant à l'implantation directe de la spécification factorisée (cette valeur correspond à la longueur du chemin critique du graphe matériel équivalent à la spécification factorisée), il définit la contrainte temporelle en termes du nombre de cycles d'horloge (NC_{tr}).

$$NC_{tr} = \frac{ctr}{T_{clk}} \tag{4.39}$$

Frontières contenant des sommets *ITERATE*

Lorsqu'une frontière de factorisation possède un sommet de type *ITERATE*, la défactorisation partielle de cette frontière n'est pas très intéressante, puisque seule la partie amont de l'*ITERATE* apporte du parallélisme et les dépendances inter-motifs à travers des I ne nous permettent pas de réduire le nombre de cycles (et, par conséquant, de réduire la latence) nécessaires à son exécution. Pour extraire du parallélisme de tels algorithmes, il est donc nécessaire de réduire les effets des dépendances inter-motifs. Si la dépendance est dûe à un opération commutative tel que l'addition, la multiplication, l'union ou l'intersection, il est possible de transformer le graphe factorisé, afin de réduire la dépendance inter-motifs au prix de calculs supplémentaires.

Nombre de ressources matérielles nécessaires

Tout algorithme peut faire apparaître une partie parallélisable et une partie non parallélisable. Nous pouvons représenter le temps d'exécution (latence) d'un algorithme en fonction du nombre de ressources matérielles, comme le montre la figure 4.13, où A correspond à la durée de la partie parallélisable, n correspond au nombre de cycles, ctr correspond à la contrainte temps réel, F correspond au facteur de défactorisation, rm_{tf} correspond au nombre de ressources matérielles de l'implantation totalement factorisée et lat_{tf} correspond à la latence de l'implantation matérielle totalement factorisée.

Nous pouvons déduire la borne inférieure (avant la prise en compte des communications) du facteur de défactorisation (F') qui sera appliqué au graphe factorisé :

$$F' = \frac{n \cdot A}{ctr - lat_{tf} + n \cdot A} \tag{4.40}$$

Relations de voisinage entre frontières

Notre heuristique de défactorisation se base sur les trois cas de relations de voisinage entre frontières de factorisation représentés par la figure 4.11.

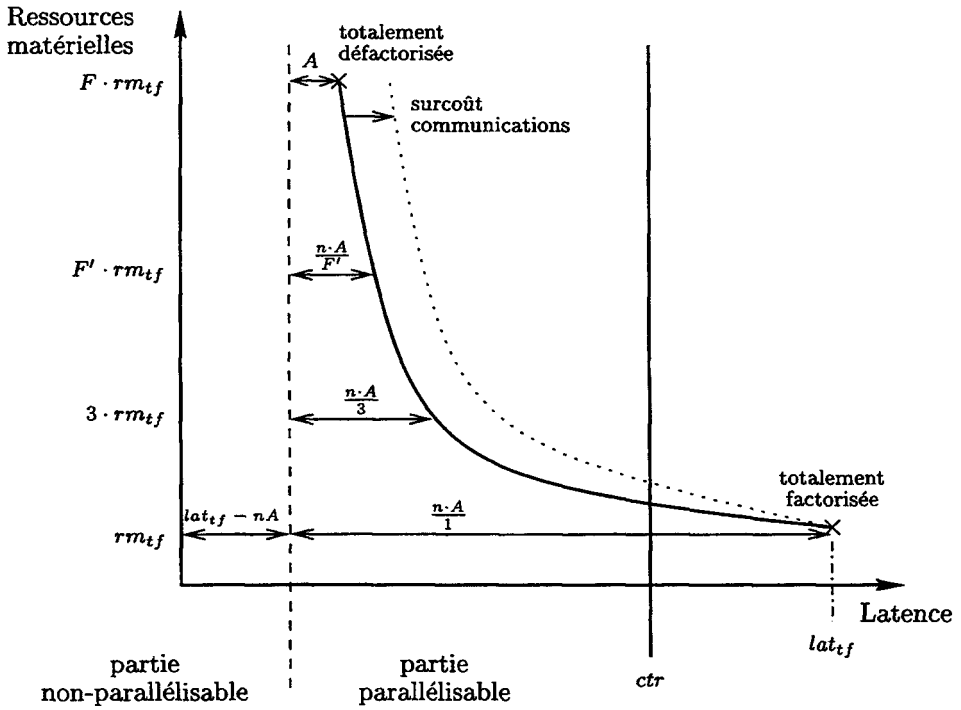


FIG. 4.13: Nombre de ressources matérielles nécessaires

Ordre de défactorisation

Parmi plusieurs frontières en relation de type $\times$, nous cherchons à défactoriser à partir de la frontière la plus profonde du sous-graphe cible. Cela assure la minimisation de la surface parce que la défactorisation des frontières moins profondes provoque une augmentation de la surface associée aux unités de contrôle et aux opérateurs de factorisation. Ainsi, les directives de défactorisation sont les suivantes :

- choisir les défactorisations qui font apparaître plus de parallélisme,
- choisir les défactorisations qui introduisent des durées d'attente moins importantes, c'est-à-dire les frontières dont la défactorisation permet de réduire le temps d'inactivité des frontières en relation de type + situées en aval,
- lorsqu'une frontière fait apparaître plus de répétitions qu'une autre, elle permet une meilleure distribution puisqu'elle possède plus de liberté pour remplir les "trous d'inactivité" des ressources matérielles allouées.

Stratégies de défactorisation

À partir des trois cas de relations de voisinage entre frontières de factorisation présentés ci-dessus, nous avons défini trois stratégies de défactorisation d'une implantation matérielle, comme nous verrons par la suite. Ainsi, pour défactoriser un

graphe matériel, il faut tout d'abord identifier les relations de voisinage entre ses frontières. Ensuite, il faut appliquer les différentes stratégies de défactorisation en fonction des relations de voisinage entre frontières. Ces stratégies sont guidées par un facteur de défactorisation, dont le calcul est présenté ci-dessous.

Facteur de défactorisation

Le facteur de défactorisation (k) définit le degré de défactorisation qui sera appliqué au graphe correspondant à l'implantation matérielle de la spécification factorisée pour obtenir une implantation qui respecte les contraintes temporelles de l'application. Le calcul de ce facteur prend en compte le nombre de cycles d'horloge de l'implantation de la spécification factorisée (NC_{fac}) et la contrainte temporelle en termes du nombre de cycles (NC_{tr}).

$$k = \begin{cases} div\left(\frac{NC_{fac}}{NC_{tr}}\right), & si\ mod\left(\frac{NC_{fac}}{NC_{tr}}\right) = 0 \\ div\left(\frac{NC_{fac}}{NC_{tr}}\right) + 1, & si\ mod\left(\frac{NC_{fac}}{NC_{tr}}\right) \neq 0 \end{cases} \tag{4.41}$$

où

div fournit le résultat de la division entière,
mod fournit le modulo de la division entière,
NC_{fac} correspond au nombre de cycles de l'implantation factorisée,
NC_{tr} correspond à la contrainte temporelle en termes du nombre de cycles.

Défactorisation de frontières en relation de type "×"

- Calculer le rapport entre le nombre de répétitions du motif de la frontière la plus profonde (FF_n) et le facteur de défactorisation k :

$$R_{d_n k} = div\left(\frac{d_n}{k}\right) \tag{4.42}$$

  où d_n correspond au nombre de répétitions du motif de la frontière FF_n.
- Si $R_{d_n k} \geq 1$, défactoriser la frontière FF_n par k.
- Sinon, défactoriser totalement la frontière FF_n par d_n et essayer de défactoriser la frontière adjacente (FF_{n-1}) par un facteur de défactorisation k' :

$$k' = div\left(\frac{NC_{tr}}{d_n}\right) + 1 \tag{4.43}$$

- Calculer le rapport entre le nombre de répétitions du motif de la frontière FF_{n-1} et le facteur de défactorisation k' :

$$R_{d_{n-1}k'} = div\left(\frac{d_{n-1}}{k'}\right) \tag{4.44}$$

où d_{n-1} correspond au nombre de répétitions du motif de la frontière FF_{n-1}.
- Si $R_{d_{n-1}k'} \geq 1$, défactoriser la frontière FF_{n-1} par k'.
- Sinon, défactoriser totalement FF_{n-1} et appliquer la même procédure successivement aux frontières adjacentes (FF_{n-2}, FF_{n-3}, ...) jusqu'à la satisfaction de la contrainte temporelle ou jusqu'à la défactorisation totale des frontières en relation de voisinage de type $\times$.

Défactorisation de frontières en relation de type "+"

- Calculer le rapport entre le nombre de répétitions du motif de chacune des n frontières en relation de type "+" (FF_1, ..., FF_n) et le facteur de défactorisation k :

$$R_{d_ik} = div\left(\frac{d_i}{k}\right) \tag{4.45}$$

- Si $R_{d_ik} \geq 1$, défactoriser la frontière FF_i par k.
- Sinon, défactoriser totalement la frontière FF_i par d_i.

Défactorisation de frontières en relation de type "*Max*"

- Identifier chacun des différents chemins parallèles dans le graphe matériel.
- Calculer le nombre de cycles de chacun de ces chemins parallèles (NC_{ch_i}).
- Éliminer tous les chemins dont le nombre de cycles est inférieur ou égal au nombre de cycles de la contrainte temporelle ($NC_{ch_i} \leq NC_tr$).
- Calculer le facteur de défactorisation à être appliqué aux chemins qui restent :

$$k_i = \begin{cases} div\left(\frac{NC_{ch_i}}{NC_{tr}}\right), & si\ mod\left(\frac{NC_{ch_i}}{NC_{tr}}\right) = 0 \\ div\left(\frac{NC_{ch_i}}{NC_{tr}}\right) + 1, & si\ mod\left(\frac{NC_{ch_i}}{NC_{tr}}\right) \neq 0 \end{cases} \tag{4.46}$$

où
div fournit le résultat de la division entière,
mod fournit le modulo de la division entière,
NC_{ch_i} correspond au nombre de cycles du chemin parallèle i,
NC_{tr} correspond à la contrainte temporelle en termes du nombre de cycles.
- Défactoriser les chemins qui restent par le facteur k' correspondant.

4.5.3 Heuristique proposée

Dans notre problème d'optimisation (la recherche d'une implantation qui respecte la contrainte temporelle, tout en minimisant l'augmentation de la surface exigée), nous identifions deux types de contraintes, comme le montre la figure 4.14 :

- temporelles : latence, temps de réponse, cadence, nombre de cycles ;
- spatiales : surface en nombre de générateurs de fonction F/G, de CLB ou de bascules DFF.

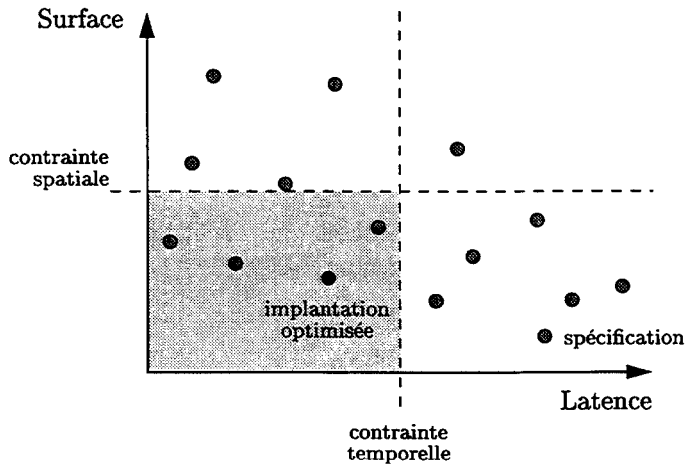


FIG. 4.14: Espace de solutions

À partir des deux types de contraintes, notre heuristique peut envisager deux phases d'optimisation :

- temporelle : consiste à sélectionner la(les) frontière(s) à défactoriser ;
- spatiale : consiste à sélectionner le degré de défactorisation qui sera appliquée à cette(ces) frontière(s).

Nous décrivons ci-dessous (Algorithme 7) notre heuristique de défactorisation d'une spécification algorithmique factorisée. Il faut souligner que, si la contrainte temporelle (*ctr*) est supérieure à la durée du chemin critique de l'implantation totalement factorisée, il n'existe pas de solution par défactorisation. Il faut donc modifier soit la spécification de l'application, soit le graphe matériel correspondant par pipeline, *retiming* ou par le choix d'un FPGA plus rapide.

Les résultats obtenus par l'application de notre heuristique de défactorisation sur la spécification algorithmique du produit matrice-vecteur (figure 2.26), sont très satisfaisants. Ces résultats assurent le respect des contraintes temporelle, tout en préservant un écart type de 25 % de l'augmentation de la surface entre la solution optimale et la solution optimisée préconisée par l'heuristique (voir Annexe A).

Algorithme 7 Heuristique de défactorisation

begin

À partir du graphe algorithmique correspondant à une spécification factorisée de l'application, identifier les frontières de factorisation présentes dans ce graphe ;

Grouper ces frontières par rangs ;

Calculer la longueur de tous les chemins du graphe entre les entrées et les sorties et le nombre de cycles nécessaire à son exécution ;

Identifier les chemins dont la latence est supérieure à la contrainte temporelle ;

Calculer, pour chacun de ces chemins, un facteur minimum de défactorisation K_i (fonction de coût ou fonction-objectif), en fonction de la contrainte temporelle (ctr) et de la latence du chemin (L_i), $k_i > div(L_i, ctr) + 1$, si $div(L_i, ctr) \geq 0,5$ ou $k_i > div(L_i, ctr)$, si $div(L_i, ctr) < 0,5$, où $k_i \in Z^+$, div et mod fournissent respectivement le résultat et le reste de la division entière entre L_i et ctr ;

Identifier, pour chacun de ces chemins, les frontières qui n'ont pas de dépendances de données inter-motifs à travers des sommets *ITERATE* et les marquer comme *défactorisables* ;

Identifier, pour chacun des chemins, parmi les frontières défactorisables, celles dont le nombre de répétitions du motif est plus grand ou égal au facteur de défactorisation k_i calculé pour son chemin respectif ;

Effectuer, pour chacun des chemins, une défactorisation totale de la frontière défactorisable appartenant au rang le plus élévé, dont le nombre de répétitions est égal ou supérieure à son respectif k_i et calculer sa nouvelle valeur de latence (L'_i) ;

Si dans un chemin, aucune frontière, prise isolément, ne satisfait pas à la condition précédente, grouper les frontières, en partant du rang de plus haut degré, jusqu'à trouver un produit du nombre de répétitions du motif, qui soit égal ou supérieure au facteur k_i du chemin respectif. Appliquer, pour chacun de ces chemins, une défactorisation totale aux frontières groupées et calculer sa nouvelle valeur de latence (L'_i) ;

Si les latences de tous les chemins sont inférieures à la contrainte temporelle, continuer. Sinon, il faut encore défactoriser les chemins qui ne respectent pas la contrainte temporelle ;

Les frontières à défactoriser ont été identifiées. Il faut donc affiner le critère de défactorisation, en essayant de minimiser la surface (réduire le degré de défactorisation par l'intermédiaire de défactorisations partielles). Calculer le rapport entre la nouvelle latence de chaque chemin (L'_i) et la contrainte temporelle (ctr) ;

Si $\frac{L'_i}{ctr} \leq 2$, effectuer la défactorisation partielle de la (des) frontière(s) identifiée(s) par un facteur égal à $div(L'_i, ctr)$. Si plusieurs frontières ont été identifiées sur un même chemin, défactoriser plus fortement les frontières appartenant aux rangs les plus élévés ;

Recalculer la latence de chacun des chemins (L''_i). Si toutes les latences sont inférieures à la contrainte temporelle, la défactorisation la plus adaptée a été trouvée. Sinon, il faut défactoriser les frontières par $div(L'_i, ctr) + div(ctr, L''_i) + 1$. Si nécessaire, répéter cette procédure jusqu'à la défactorisation totale des frontières.

end

4.5.4 Génération de code VHDL

VHDL est un langage conçu avec une sémantique de simulation, s'imposant comme le point d'entrée incontournable des outils de synthèse logique et architecturale. Son usage dans le domaine de la synthèse exige, pourtant, que le concepteur utilise un sous-ensemble de VHDL structurel dit *synthétisable*, qui correspond à des constructions matérielles identifiables [ACO92]. Nous énumérons ci-dessous quelques restrictions de ce sous-ensemble : tous les types de données codables sont admis ; les opérateurs logiques basés sur les types logiques standartisés, les opérateurs arithmétiques utilisant les types *INTEGER* (ou ses dérivés) ou les types vecteurs de bit hérités de paquetages normalisés, les opérateurs de comparaison pour les nombres à représentation binaire et les opérateurs de décalage arithmétiques et logiques sont également acceptés ; tout le jeu d'instruction de VHDL est applicable à la synthèse, sauf l'assertion (*assert*) ; une seule horloge est permise par processus et toutes les constructions VHDL sont acceptables.

La qualité de la spécification d'une application en VHDL dépend énormément de l'architecture-cible. Une légère modification dans la description en VHDL peut avoir un impact très important sur l'efficacité de l'implantation. Cela est encore plus vérifiable quand l'architecture-cible est formée par des circuits reconfigurables du type FPGA, dont les ressources matérielles (nombre de cellules logiques) sont fixées *a priori*. Gschwind et Salapura [GS95] ont présenté une méthode pour optimiser les descriptions VHDL, afin de permettre une meilleure utilisation des ressources des FPGA. Cette méthode consiste à écrire le code VHDL, en tenant compte, de façon rigoureuse, de la structure interne des FPGA.

À partir d'une spécification algorithmique au niveau comportemental sous la forme d'un GFDD, nous devons générer une description structurelle de l'application au niveau RTL, qui soit lisible, modifiable et modulaire, de façon à permettre son implantation intégrale ou partielle par différents concepteurs. Nous cherchons surtout à permettre la génération automatique de code VHDL, à partir d'une spécification de haut niveau, plus particulièrement à partir de notre modèle de GFDD et donc du graphe matériel (graphe d'opérateurs interconnectés) correspondant. Évidemment, ce code produit doit être compatible avec les outils de CAO existants (p.ex., *Leonardo*, *Cadence*, etc.). De plus, la description doit être simulable afin de permettre sa vérification fonctionnelle avant l'implantation finale.

Dans le cadre de ce travail, le code VHDL a été généré à la main, à partir d'une bibliothèque de composants VHDL (voir Annexe B). Cette bibliothèque a été écrite d'après la description des opérateurs de base, des opérateurs de factorisation et des unités de contrôle présentées dans le chapitre 3. La génération de code VHDL peut être réalisée trivialement, en remplaçant les sommets du graphe matériel détaillé par les respectifs composants VHDL. Les arcs de ce graphe, à leur tour, sont remplacés par les signaux d'interconnexion des composants. Les unités et les signaux de contrôle sont également remplacés par les respectifs composants et ses signaux.

4.6 Conclusion

Dans ce chapitre, nous avons décrit notre modèle de conception pour la génération de *netlists* de configuration pour les architectures mono-FPGA, à partir d'un modèle de graphes factorisés de dépendances de données. Nous avons montré comment ce modèle peut être implanté sous la forme d'une extension du logiciel *SynDEx*. Nous avons présenté notre méthode de caractérisation matérielle des opérateurs de base, des opérateurs de factorisation de motifs de graphes répétitifs et des unités de contrôle, en termes de surface (nombre de générateurs de fonction F/G, de CLB et de bascules DFF) et de latence, en fonction de l'architecture-cible (nous avons choisi les FPGA *Xilinx* de la série XC4000XL-3). Nous avons exposé également notre méthode d'estimation du nombre de ressources matérielles nécessaires à l'implantation matérielle (nombre de CLB ou de générateurs de fonction F/G) et de la latence totale de cette implantation. En dernier, nous avons montré notre méthode d'optimisation de l'implantation par défactorisation, basée sur une heuristique de recherche, guidée par des informations fournies par l'estimateur de surface et de latence.

Les résultats obtenus par notre estimateur de ressources matérielles et de latence (voir tableau A.15) sont très acceptables, par rapport aux valeurs obtenues par l'intermédiaire du logiciel *Leonardo Spectrum*, version 1999.1i, d'*Exemplar Logic* (16% de erreur moyenne pour le nombre de générateurs de fonction F/G, 10% pour le nombre de CLB, 4% pour le nombre de bascules DFF et 9% pour la latence). Il faut souligner l'impossibilité de réaliser une estimation précise de la latence, par exemple, à cause des optimisations effectuées par les outils de placement et routage qui intègrent le logiciel de CAO. La différence essentielle entre notre méthode d'estimation et celle des outils de CAO, tels que *Leonardo*, est que nous réalisons l'estimation à partir d'une description comportementale (spécification algorithmique). Ces outils, par contre, réalisent l'estimation à partir d'une description RTL (code VHDL synthétisable). Nous effectuons donc notre estimation dans un niveau d'abstraction plus élevé, ce qui nous permet d'explorer l'espace de solutions plus facilement avant de passer à la synthèse RTL ou physique.

L'application, à la main, de notre heuristique d'optimisation par défactorisation, à la spécification factorisée du PMV, a produit des résultats très satisfaisants. Nous assurons le respect des contraintes temporelles, tout en préservant une déviation maximale de 25 % de l'augmentation de la surface par rapport à la solution optimale. La stratégie consistant, d'abord à respecter la contrainte temporelle, pour ensuite minimiser les ressources matérielles, a démontré son efficacité, comme nous pouvons le constater dans l'Annexe A.

Le code VHDL correspondant à l'implantation matérielle optimisée a été lui aussi généré à la main, à partir du graphe matériel détaillé. Les sommets de ce graphe ont été remplacés par des composants (*component*) VHDL définis dans une bibliothèque. Cette bibliothèque a été construite à partir de la description des opérateurs de base, des opérateurs de factorisation et des unités de contrôle présentées dans le chapitre 3. Les arcs du graphe matériel ont été remplacés par les signaux d'interconnexions entre les composants VHDL.

Conclusions et perspectives

Le manque d'outils de conception permettant l'implantation optimisée d'algorithmes bas niveau de traitement du signal et des images (TSI) sur des architectures reconfigurables, intégrant la synthèse du chemin de données et du chemin de contrôle à partir d'un modèle unifié de la spécification algorithmique jusqu'à l'implantation matérielle a été le moteur de ce travail. De plus, nous cherchions à développer une méthodologie de conception orienté circuits qui, une fois intégrée au logiciel *SynDEx*, nous permettrait par la suite de réaliser la conception conjointe matériel/logiciel.

Nous avons entrepris une étude bibliographique sur la conception conjointe matériel/logiciel qui nous a permis de conclure que, parmi les différents outils universitaires et commerciaux de conception conjointe existants, aucun ne satisfait pas à tous les critères d'évaluation des outils de conception conjointe présentés dans la section 1.1.4 et aucun ne résoud de façon satisfaisante et simultanée les problèmes concernant la spécification indépendante de l'implantation et la synthèse automatique de systèmes réactifs temps réel. Cela nous a encouragé à développer une extension de la méthodologie "Adéquation Algorithme Architecture" (AAA) supportée par *SynDEx* aux circuits reconfigurables.

Nous avons réalisé un état de l'art sur la spécification, la modélisation, les langages de spécification et la validation de systèmes réactifs temps réel. Nous avons pu constater que le meilleur modèle est celui qui traduit le mieux les caractéristiques qu'il modélise. Nous avons pu constater également que les langages synchrones, à cause de leur sémantique et de leurs mécanismes de validation, sont les plus appropriés pour la spécification de tels systèmes. Les algorithmes bas niveau de TSI sont caractérisés par une grande régularité, traduite sous la forme de répétitions d'un motif. Cela nous a motivé pour les spécifier à l'aide d'un modèle basé sur des graphes factorisés de dépendances de données (GFDD), capables de mettre en évidence cette régularité. *SynDEx* nous apparaît encore une fois comme un bon choix pour héberger notre extension de la méthodologie AAA aux circuits reconfigurables, car il offre au concepteur la possibilité d'effectuer des vérifications formelles sur la spécification de l'algorithme à l'aide des langages synchrones. Ces langages fournissent, en utilisant le format commun DC, cette spécification à *SynDEx* pour l'implanter.

Afin de pouvoir situer notre approche par rapport aux outils et méthodes existantes, nous avons présenté un état de l'art sur la synthèse de circuits, plus particulièrement sur la synthèse aux niveaux comportemental et transfert de registres. Dans notre méthodologie, nous partons d'une description comportementale sous la forme d'un GFDD (graphe algorithmique) pour générer une description au niveau RTL sous la forme d'un schéma logique (graphe matériel). Ce graphe matériel est obtenu par traduction directe du graphe algorithmique : chaque sommet algorithmique (opération) est remplacé par un sommet matériel (opérateur). La synchronisation des transferts de données entre les registres des opérateurs synchrones est générée à partir de l'analyse des relations entre les sommets qui effectuent la factorisation des dépendances de données du graphe algorithmique. L'analyse de ces relations étant simple, la génération des synchronisations pourra être facilement automatisée. Nous avons ainsi montré que nous pouvons synthétiser simultanément les chemins de données et de contrôle d'une implantation matérielle à partir de quelques transformations appliquées à la spécification algorithmique.

Notre méthode d'implantation, comme celle proposée par Cesário et al. [CSSJ99], explore la superposition entre la synthèse comportementale et la synthèse RTL, afin de permettre au concepteur d'explorer différents flots de synthèse en fonction de la nature de l'application et de ses contraintes, et de la qualité des outils de synthèse utilisés en aval dans le cycle de conception. Un des grands avantages de notre méthode par rapport aux travaux précédents réside dans le fait d'utiliser un même modèle de conception, depuis la spécification algorithmique jusqu'à l'implantation matérielle.

Notre logique de contrôle hiérarchique "délocalisée" permet aux outils de CAO utilisés pour la synthèse logique de placer les unités de contrôle plus proches des opérateurs à contrôler, qu'une logique de contrôle "centralisée". Cela nous permet de réduire le chemin critique de façon importante, comme vérifié par Huang et Wolf [HW94]. La hiérarchisation des contrôleurs locaux réduit la surface occupée par le chemin de contrôle.

À partir d'une étude sur quelques méthodes d'optimisation de la synthèse de haut niveau, de la synthèse de circuits programmables et sur l'utilisation de méthodes heuristiques pour résoudre des problèmes NP-complets, tels que le partitionnement de graphes, l'ordonnancement de tâches et l'optimisation, nous avons pu définir notre stratégie d'optimisation. Nous cherchions à trouver une implantation matérielle qui respecte les contraintes temporelles de l'application, tout en minimisant l'augmentation des ressources matérielles. En partant d'une spécification algorithmique factorisée, nous pouvons obtenir plusieurs implantations matérielles différentes plus ou moins défactorisées. Comme la qualité des résultats de la synthèse ne dépend pas seulement de la qualité des outils utilisés, mais aussi du processus de conception, notre problème consistait à trouver l'implantation optimisée sans pour autant être obligé de parcourir tout l'espace de solutions et sans effectuer un cycle complet de conception (spécification, codage, implantation, simulation, estimation) pour chaque implantation possible.

Notre méthode d'optimisation, basée sur une heuristique de défactorisation, effectue l'estimation de la surface et de la latence correspondant à l'implantation de la spécification à partir d'une caractérisation des sommets du graphe algorithmique en fonction de l'architecture-cible. Les valeurs de surface et de latence ainsi calculées sont utilisées par une fonction de coût pour guider l'heuristique. Notre estimateur, qui opère au niveau comportemental, offre une précision moyenne supérieure à 80 % pour l'estimation de surface et supérieure à 90 % pour l'estimation de latence, des valeurs tout à fait acceptables.

L'originalité de cette approche d'implantation optimisée d'algorithmes bas niveau de TSI sur des circuits reconfigurables vis à vis des outils existants réside d'une part dans le fait qu'il n'y a pas de rupture entre la spécification comportementale sous la forme d'un GFDD et son implantation au niveau RTL sous la forme d'un schéma logique, et d'autre part dans le fait de permettre l'unification du traitement du chemin de données et du chemin de contrôle tout au long du processus de conception.

Les grands avantages de cette approche par rapport aux outils existants sont la simplification de l'optimisation de la synthèse au niveau comportementale, l'unification vis à vis de l'implantation optimisée des algorithmes moyen et haut niveau de TSI avec conditionnement, par l'intermédiaire de *SynDEx*, et la possibilité de réaliser la conception conjointe pour les architectures multi-composant.

Perspectives

Premièrement, toutes les étapes de notre méthodologie (traduction matérielle, optimisation par défactorisation, implantation optimisée et génération du code VHDL structurel synthétisable) pourraient être automatisées et intégrées au logiciel *SynDEx*.

Dans ce travail, nous nous sommes restreint aux architectures mono-FPGA. Ainsi, un des premiers développements envisageables à partir de nos résultats, est son extension aux architectures multi-FPGA. Dans ce cas, il faut prendre en compte le partitionnement du graphe algorithmique sur plusieurs FPGA.

La méthode d'optimisation de l'implantation peut être améliorée si on y ajoute, en plus de la transformation par défactorisation, d'autres transformations tels que le *retiming* qui permettrait de réduire le chemin critique du graphe matériel et, par conséquent, de réduire la latence de l'implantation, par l'intermédiaire de l'ajout de registres supplémentaires sur le chemin critique.

L'intégration de notre méthode à *SynDEx* nous permettrait d'effectuer l'implantation optimisée des algorithmes bas, moyen et haut niveau de traitement du signal et des images sur des architectures multi-composant hétérogènes, c'est-à-dire de réaliser la conception conjointe matériel/logiciel. Pour que ce processus de conception conjointe soit vraiment efficace, il faut développer une méthode de partitionnement matériel/logiciel automatique optimisée et l'intégrer à *SynDEx*. Elle sera fondée sur l'unification des différentes heuristiques utilisées par *SynDEx*, celles pour les multi-composant et celles pour les FPGA, afin de traiter de façon unique l'optimisation de l'implantation des algorithmes bas, moyen et haut niveau de TSI sur des architectures multi-composant hétérogènes.

Bibliographie

[ABCS93] ABOUZEID, Pierre, BABBA, Belgacem, CRASTES DE PAULET, Michel, SAUCIER, Gabrielle. Input-driven partitioning methods and application to synthesis on table-lookup-based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 12, n. 7, July 1993, p.913-925.

[AD97] ACOCK, S. J. B., DIMOND, K. R. Automatic mapping of algorithms on to multiple FPGA-SRAM modules. In : FPL'97–INTERNATIONAL WORKSHOP ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, 7, London, 1997. *Proceedings...* Wayne Luk, Peter Y. K. Cheung (Eds.) Berlin : Springer, 1997. p.255-264.

[AT96] ADAMS, Jay K., THOMAS, Donald E. The design of mixed hardware/-software systems. In : DESIGN AUTOMATION CONFERENCE, 33, Las Vegas, 1996. *Proceedings...*

[ASU89] AHO, Alfred V., SETHI, Ravi, ULLMAN, Jeffrey D. *Compilateurs* : principes, techniques et outils. s.l. : InterEditions, 1989.

[AKJ96] AICHOUCHI, Mohamed, KISSION, Polen, AMINE, Ahmed. Lien entre la synthèse architecturale et les outils de conception au niveau transfert de registres. *Technique et Science Informatiques*, v.15, n.2, 1996, p.179-199.

[ALSV96] AIGLON, Caroline, LAVARENNE, Christophe, SOREL, Yves, VICARD, Annie. Utilisation de SynDEx pour le traitement d'images *temps réel*. Rocquencourt : Institut National de Recherche en Informatique et en Automatique, 1996. 79p. (Rapport de Recherche No. 2968)

[ACO92] AIRIAU, Roland, CLOSSE, Etienne, OLIVE, Vincent. Synthèse d'algorithmes à partir de VHDL. In : WORKSHOP ADÉQUATION ALGORITHMES ARCHITECTURES POUR TRAITEMENT DU SIGNAL ET DES IMAGES, 1, Lannion, Sept. 1992. *Actes...* 12p.

[ABOR90] AIRIAU, R., BERGÉ, J.-M., OLIVE, V., ROUILLARD, J. *VHDL du langage à la modélisation*. Lausanne : Presses Polytechniques et Universitaires Romandes, 1990. 553p.

[Alf98] ALFKE, Peter. How to evaluate the XC4000XL for your next application. *XCell Journal*, n. 28, article Q2, 1998.
http ://www.xilinx.com/xcell/xl28/xl28_30.pdf

[Alv94] ALVES DE BARROS, Marcelo. *Traitement bas niveau d'images en temps réel et circuits reconfigurables.* Orsay : Université de Paris-Sud, 1994. 200p. (Thèse de Doctorat)

[AK98] ANDRADE, Hugo A., KOVNER, Scott. Software synthesis from dataflow models for G and LabVIEW. In : ASILOMAR CONFERENCE ON SIGNALS, SYSTEMS, AND COMPUTERS, 32, Pacific Grove, 1998. *Proceedings...* 5p.

[ABFS94] ANTONIAZZI, S., BALBONI, FORNACIARI, W., SCIUTO, D. A methodology for control-dominated systems codesign. In : INTERNATIONAL WORKSHOP ON HARDWARE-SOFWARE CODESIGN, 1994. *Proceedings...* p.2-9.

[ABBR92] ARNOLD, André, BEAUQUIER, Joffroy, BÉRARD, Béatrice, ROZOY, Brigitte. *Programmes parallèles* : modèles et validation. Paris : Armand Colin, 1992. p.31-43, 93-107, 123-139.

[ADN91] ASHAR, Pranav, DEVADAS, Srinivas, NEWTON, Richard. Optimum and Heuristic Algorithms for a Problem of Finite State Machine Decomposition. *IEEE Transactions on Computer-Aided Design*, v.10, n.3, March 1991, p.296-310.

[ACS*92] AYLOR, James, CAMPOSANO, Raul, SCHUETTE, Michael, WOLF, Wayne, WOO, Nam. The future of embedded systems design. In : ICCD'92–IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN : VLSI in Computers & Processors, 1992. *Proceedings...* p.144-146.

[Bab99] *BABEL : a glossary of computer oriented abbreviations and acronyms.* v.99c. Baltimore : Irving & Richard Kind, 1999.
http ://www.access.digex.net/~ikind/babel.html

[BFS96] BALBONI, A., FORNACIARI, W., SCIUTO, D. Partitioning and exploration strategies in the Tosca co-design flow. In : INTERNATIONAL WORKSHOP ON HARDWARE-SOFWARE CO-DESIGN, 1996. *Proceedings...* p.62-69.

[Bea*98] BEAUVAIS, J.-R. et al. A translation of Statecharts into Signal. In : CSP'98–INTERNATIONAL CONFERENCE ON APPLICATION OF CONCURRENCY TO SYSTEM DESIGN, Aizu-Wakamatsu, Japan, 1998. *Proceedings...* IEEE Publications. p.52-62.

[Bea99] BEAUVAIS, Jean-René. *Modélisation de STATECHARTS en SIGNAL pour la conception de systèmes critiques temps-réel.* Rennes : Université de Rennes 1, 1998. p.39-48. (Thèse de Doctorat en Informatique)

[Bel94] BELHADJ, Mohammed. *Conception d'architectures en utilisant SIGNAL et VHDL.* Rennes : Université de Rennes I, 1994. 162p. (Thèse de Doctorat en Informatique)

[Bel*98] BELLIFEMINE, F. et al. Hardware/software co-design for image processing. In : INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING AND COMMUNICATIONS, 1998. *Proceedings...*

[Ben*98] BENMOHAMMED, Mohamed, KISSION, Polen, RAHMOUNI, Maher, LIEM, Clifford, JERRAYA, Ahmed Amine. Génération automatique de contrôleurs reprogrammables dans un environnement de synthèse de haut niveau. *Technique et science informatique*, v.17, n.10, 1998, p.1277-1297.

[BB91] BENVENISTE, Albert, BERRY, Gérard. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, v. 79, n. 9, p.1270-1782, 1991. (Rapport de Recherche INRIA, n. 1445, juin 1991)

[BLSS95] BENVENISTE, Albert, LE GUERNIC, Paul, SOREL, Yves, SORINE, Michel. A denotational theory of synchronous reactive systems. *Information and Computation*, v. 99, n. 2, 1992, p.192-230.

[Ben93] BENZAKKI, J. *OSYS : outil de synthèse et de spécification*. Paris : Conservatoire National des Arts et Métiers, 1993. (Thèse de Doctorat)

[BCP90] BERGAMASCHI, Reinaldo, CAMPOSANO, Raul, PAYER, Michael. *Data path synthesis using path analysis*. Yorktown Heights : IBM Research Division, 1990. 13p. (Research Report RC 16274).

[Ber99] BERGAMASCHI, Reinaldo. A behavioral network graph unifying the domains of high-level and logic synthesis. In : DESIGN AUTOMATION CONFERENCE, 36, New Orleans, 1999. *Proceedings...* p.213-218.

[Ber92] BERKONE, Bachir. *Vérification des systèmes matériels numériques séquentiels synchrones*. Grenoble : Institut National Polytechnique de Grenoble, 1992. (Thèse de Doctorat)

[Ber91] BERRY, Gérard. *A hardware implementation of pure ESTEREL*. s.l. : Institut National de Recherche en Informatique et en Automatique, 1994. (Technical Report n. 1479)

[Ber93] BERRY, Gérard, TOUATI, Hervé. Optimized controller synthesis using Esterel. In : INTERNATIONAL WORKSHOP ON LOGIC SYNTHESIS, Lake Tahoe, 1993. *Proceedings...*

[Ber98] BERRY, Gérard. The Foundations of Esterel. In : *Proof, Language and Interaction* : Essays in Honour of Robin Milner. PLOTKIN, G., STIRLING, C., TOFTE (Ed.) MIT Press, 1998. 31p.

[BML99] BHATTACHARYYA, Shuvra S., MURTHY, Praveen K., LEE, Edward A. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing*, v.21, 1999. p.151-166.

[BELP93] BILSEN, G., ENGELS, M., LAUWEREINS, R., PEPERSTRAETE, J. A. Survey of algorithms for static load balancing. In : IASTED - INTERNATIONAL CONFERENCE ON MODELLING AND SIMULATION, Pittsburgh, 1993. *Proceedings...* p.418-421.

[BC97] BOUGÉ, L., CACHERA, D. A logical framework to prove properties of Alpha programs. In : APPLICATION SPECIFIC ARRAY PROCESSORS, 1997. *Proceedings...* s.l. : IEEE Computer Society, July 1997.

[BLLMS94] BOURNAI, P., LAVARENNE, C., LE GUERNIC, P., MAFFEÏS, O., SOREL, Y. *Interface SIGNAL-SynDEx*. Rocquencourt : Institut National de Recherche en Informatique et en Automatique, 1994. 49 p. (Rapport de Recherche INRIA, n. 2206)

[BD91] BOUSSINOT, F., DE SIMONE, R. The ESTEREL language. *Proceedings of the IEEE*, v.79, n.9, Sept. 1991, p.1293-1304.

[B*88] BRAYTON, R. et al. *The Yorktown silicon compiler*. Reading, Massachussets : Addison-Wesley, 1988. p.204-311.

[BHS93] BRAYTON, K., HACHTEL, G., SANGIOVANNI-VINCENTELLI, A. Multi-level logic synthesis. *Proceedings of the IEEE*, Feb. 1993, p.264-300.

[BCDM86] BROWNE, M.C., CLARKE, E.M., DILL, D., MISKRA, B. Automatic verification of sequential circuits using temporal logic. *IEEE Transaction on Computers*, v. C-35, n. 12, Dec. 1986, p.1035-1044.

[Bry86] BRYANT, R.E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computer*, v. C-35, n. 8, Aug. 1986, p.677-691.

[Buc93] BUCHENRIEDER, K. Hardware-software codesign : codesign and concurrent engineering. *Computer* : "Hot Topics", Jan. 1993, p.85-87.

[BCM*90] BURCH, J.R., CLARKE, E.M., DILL, D., HWANG, L.J. Sequential circuit verification using symbolique model checking. In : DESIGN AUTOMATION CONFERENCE, 1990. *Proceedings...*

[BJ92] BURLESON, W., JUNG, B. Arrest : an interactive graphic analysis tool for VLSI arrays. In : APPLICATION SPECIFIC ARRAY PROCESSORS, 1992. *Proceedings...* s.l. : IEEE Computer Society Press, 1992.

[BW97] BURNS, Alan, WELLINGS, Andy. *Real-time systems and programming languages*. 2 ed. Reading : Addison-Wesley, 1997. 611p.

[CW??] CALLAHAN, Tim, WAWRZYNEK, John. *Datapath oriented FPGA mapping*. Berkeley : UCLA, s.d. 1p.

[Cam90] CAMPOSANO, R. From behavior to structure : high-level synthesis. *IEEE Design & Test of Computers*, Oct. 1990, p.8-19.

[C*91] CAMPOSANO, R. et al. The IBM high level synthesis system. In : CAMPOSANO, R., WOLF, W. (Ed.) *High level VLSI synthesis*. Norwell : Kluwer Academic, 1991.

[Cam96] CAMPOSANO, Raul. Behavioral synthesis. In : DESIGN AUTOMATION CONFERENCE, 33, Las Vegas, 1996. *Proceedings...*

[CP88] CAMURATI, P., PRIMETTO, P. Formal verification of hardware correctness : introduction and survey of current research. *IEEE Computer*, July 1988, p.8-19.

[CB96] CAPOROSSI, Dino, BARRETT, Beoff. Formal verification of a large design. *Integrated System Design Magazine*, Jan. 1996.
http ://www.isdmag.com/Editorial/1996/CoverStory9601.html

[CVRD92] CATTHOOR, F., VAN SWAAIJ, M., ROSSEEL, J., DE MAN, H. Array design methodologies for real-time signal processing in the Cathedral-IV synthesis environment. In : QUINTON, P., ROBERT, Y. *Algorithms and parallel VLSI architectures II*. Amsterdam : Elsevier, 1992. p.211-222.

[CH90] CAVIN, R., HILBERT, J. Design of integrated circuits : directions and challenges. *Proceedings of the IEEE*, v. 78, n. 2, Feb. 1993, p.418-435.

[Ceb95] CEBELIEU, Marie-Claude. *Utilisation de macro-blocs en synthèse VHDL*. Grenoble : Institut National Polytechnique de Grenoble, 1995. 226p. (Thèse de Doctorat en Microélectronique)

[CKGJ97] CESÁRIO, W. O., KISSION, P., GUILLAUME, P., JERRAYA, A. A. Unified evaluation model for interconnexion schemes used in behavioral synthesis. In : IWLAS'97 - INTERNATIONAL WORKSHOP ON LOGIC AND ARCHITECTURE SYNTHESIS, Grenoble, 1997. *Proceedings...*

[CSSJ99] CESÁRIO, Wander O., SUGAR, Zoltan, SUESCAN, Rudolphe, JERRAYA, Ahmed A. Overlap and frontiers between behavioral and RTL synthesis. In : DATE'99 - USER'S FORUM, Munich, 1999. *Proceedings.*

[CSZ94] CHAN, Pak K., SCHLAG, Martine D. F., ZIEN, Jason Y. Spectral k-way ratio-cut partitioning and clustering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v.13, n.9, Sept. 1994, p.1088-1096.

[CL96] CHARTRAND, G., LESNIAK, L. *Graphs and digraphs*. 3 ed. London : Chapman & Hall, 1996. 422p.

[CW96] CHAUDHURI, S., WALKER, R. A. Computing lower bounds on functional units before scheduling. *IEEE Transactions on Very Large Scale Integrated Systems*, v.4, June 1996, p.273-279.

[Chi99] CHINOOK. *The Chinook project*. Washington : University of Washington/Dept. of CS and E., 1999.
http ://www.cs.washington.edu/research/projects/lis/www/chinook/

[C*94] CHIODO, Massimiliano et al. Hardware-software codesign of embedded systems. *IEEE Micro*, Aug. 1994, p.26-36.

[COB95] CHOU, Pai H., ORTEGA, Ross B., BORRIELLO, Gaetano. The Chinook hardware/software co-synthesis system. In : INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, 8, Cannes, 1995. *Proceedings...* p.22-27.

[CB98a] CHOU, Pai, BORRIELLO, Gaetano. Modal processes : towards enhanced retargetability through control composition of distributed embedded systems. In : DESIGN AUTOMATION CONFERENCE, 35, 1998. *Proceedings...* 6p.

[CB98b] ____. *Functional encapsulation vs. state encapsulation in the specification of reactive systems*. Seattle : University of Washington, 1998. (Working paper)
http ://www.cs.washington.edu/~chou-lctes98.pdf

[COH*99] CHOU, Pai, ORTEGA, Ross, HINES, Ken, PARTDRIDGE, Kurt, BORRIELLO, Gaetano. ipChinook : an integrated IP-based design framework for distributed embedded systems. In : DAC'99. *Proceedings...* p.44-49.

[CPTR89] CHU, C. M., POTKONJAK, M., THALER, M., RABAEY, J. HYPER : an interactive synthesis environment for high performance real time applications. In : ICCD'89 – INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, 1989. *Proceedings...* p.432-435.

[Cie99] CIERTO. *Cierto Virtual Component Co-design.* Cadence, 1999. http ://www.cadence.com/~ciertovcc

[CBE*92] CLAESEN, L., BARRIONE, D., EVEKING, H., MILNE, G., PAILLET, J.L., PRIMETTO, P. CHARME : towards formal design and verification for provably correct VLSI hardware. In : P. Primetto, P. Camurati (Eds.) *Correct hardware design methodologies*, 1992. p.3-25.

[Cla88] CLARK, Robert G. *Comparative programming languages.* Avon : Addison-Wesley, 1988. p.1-13, 322-325, 330-331.

[C*99] CLOUTÉ et al. Hardware/software co-design of an avionics communication protocol interface system : an industrial case study. In : CODES'99 – INTERNATIONAL WORKSHOP ON HARDWARE/-SOFTWARE CODESIGN, 7, Rome, 1999. *Proceedings...* p.48-53.

[Cod99] CODESIGN. *CodeSign Project.* Zurich : Eidgenössiche Technische Hochschule, 1999. http ://www.tik.ee.ethz.ch/~codesign/

[Com92] COMPASS. *ASIC synthesizer for VHDL design* : V8R4.0 edition. s.l. : Compass design automation, Nov. 1992.

[CL95] COROYER, Chistophe, LIU, Zhen. *Effectiveness of heuristics and simulated annealing for the scheduling of concurrent tasks* : an empirical comparison. Sophia-Antipolis : Institut National de Recherche en Informatique et en Automatique, 1991. 28p. (Rapport de Recherche INRIA, n. 1379)

[Cos98] COSYMA. *COSYnthesis for eMbedded Architectures.* ERNST, Rolf (Dir.) Braunschweig : Tecnhical University of Braunschweig, 1998. http ://www.ida.ing.tu-bs.de

[CT??] COSNARD, Michel, TRYSTRAM, Denis. *Algorithmes et architectures parallèles.* s.l. : InterEditions, s.d. p.1-34, 200-263, 338-359.

[CHL*99] COSTE, P., HESSEL, F., LE MARREC, Ph., SUGAR, Z., ROMDHANI, M., SUESCUN, R., ZERGAINOH, N., JERRAYA, A.A. Multilanguage design of heterogeneous systems. In : CODES'99 – INTERNATIONAL WORKSHOP ON HARDWARE/SOFTWARE CODESIGN, 7, Rome, 1999. *Proceedings...* p.54-58.

[CBM89] COUDERT, O., BERTHET, MADRE, J.C. Verification of sequential machines using boolean function vectors. In : IFIP INTERNATIONAL WORKSHOP ON APPLIED FORMAL METHODS FOR CORRECT VLSI DESIGN, 1989. *Proceedings...* p.111-128.

[Cou91] COUDERT, Olivier. *SIAM : Une boîte à outils pour la preuve formelle de systèmes séquentiels.* Paris : École Nationale Supérieure des Télécommunications, 1991. 154p. (Thèse de Doctorat)

[Cou93] COURTOIS, Bernard. *CAD and testing of ICs and systems* : Where are we going ? Grenoble : IMAG, 1993. (Technical Report)

[Cow99] COWARE. *The Coware and Symphony projects.* Leuven : IMEC, 1999. http ://dmz4.imec.be/vsdm/projects/coware_symphony/

[Chr99] CHRYSALIS. *Design VERIFYer version 2.4* : Formal equivalence checking solution for complex IC designs. Chrysalis Symbolic Design, Inc., 1999.
http ://www.chrysalis.com/products/DV_datasheet_4-99.html

[Da*96] DAVEAU, Jean-Marc et al. VHDL generation from SDL specifications. In : IFIP, 1996. *Proceedings...* 20p.

[DW99] DEHON, André, WAWRZYNEK, John. Reconfigurable computing : What, why, and implications for design automation. In : DESIGN AUTOMATION CONFERENCE, 36, New Orleans, 1999. *Proceedings...* p.610-615.

[DeH98] DEHON, André. Comparing computing machines. In : SPIE – CONFIGURABLE COMPUTING : TECHNOLOGY AND APPLICATIONS. *Proceedings of SPIE*, v. 3256, Nov. 1998.

[DCG*90] DE MAN, H., CATTHOOR, F., GOOSSENS, G., VANHOOF, J., VAN MEERBN, J., NOTE, S., HUISKEN, J. Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms. *Proceedings of IEEE*, Feb. 1990, p.44-47.

[De*96] DE MAN, H. et al. Co-design of DSP systems. In : DE MICHELI, G., SAMI, M. (Ed.) *Hardware/software codesign.* NATO Advanced Study Institute (ASI) Series E, v.310, p.75-104, Kluwer Academic Publishers, 1996.

[Dem91] DEMASSIEUX, Nicolas. *Architectures VLSI pour le traitement d'images* : une contribution à l'étude du traitement matériel de l'information. Paris : École National Supérieure de Télécommunications, 1991. 249p. (Thèse de Doctorat en Électronique et Communications)

[DKMT90] DE MICHELI, G., KU, D., MAILHOT, F., TRUONG, T. The Olympus synthesis system for digital design. *IEEE Design & Test of Computers*, v.7, n.1, Oct. 1990, p.37-53.

[DR85] DENYER, P., RENSHAW, D. *VLSI processing* : a bit serial approach. s.l. : Addison-Wesley, 1985.

[Den98] DÉNOYER, Thierry. *Optimisation d'applications temps réel distribuées par exploitation de la régularité algorithmique.* Rocquencourt : Institut National de Recherche en Informatique et en Automatique, Université Paris XI Orsay, 1998. 74p. (Rapport DEA "Systèmes Électroniques pour le Traitement de l'Information)

[DHW93] DEPLETERRE, Ed. F., HELD, Peter, WIELAGE, Paul. Model and methods for regular array design. *International Journal of High Speed Electronics and Systems*, v. 46, n. 2, 1993, p.133-201.

[DN89] DEVADAS, Srinivas, NEWTON, Richard. Decomposition and factorization of sequential finite state machines. *IEEE Transactions on Computer-Aided Design*, v.8, n.11, November 1989, p.1206-1217.

[DD91] DEWILDE, P., DEPLETERRE, E. Architectural of large, nearly regular algorithms : design, trajectory and environment. *Annales des Télécommunications*, v. 46, n2-3, 1991, p.45-59.

[Dez93] DEZAN, Cathérine. *Génération automatique de circuits avec Alpha de Centaur*. Rennes : Université de Rennes I, 1989. (Thèse de Doctorat)

[DQ94] DEZAN, Catherine, QUINTON, Patrice. Verification of regular architectures using Alpha : a case study. In : APPLICATION SPECIFIC ARRAY PROCESSORS'94, 1994. *Proceedings...*

[DALS98] DIAS, Ailton F., AKIL, Mohamed, LAVARENNE, Christophe, SOREL, Yves. Adéquation algorithme-architecture appliquée aux circuits reconfigurables. In : JOURNÉES ADÉQUATION ALGORITHME ARCHITECTURE EN TRAITEMENT DU SIGNAL ET IMAGES, 4, Saclay, 1998. *Actes.* Saclay : CEA/LETI, 1998. p.35-42.

[DLAS98] DIAS, Ailton F., LAVARENNE, Christophe, AKIL, Mohamed, SOREL, Yves. Optimized implementation of real-time image processing algorithms on field programmable gate arrays. In : ICSP'98–INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING, 4, Beijing, 1998. *Proceedings.* v. II. Beijing : IEEE Press, Publishing House of Electronics Industry, 1998. p.1080-1083.

[DALS00] DIAS, Ailton F., AKIL, Mohamed, LAVARENNE, Christophe, SOREL, Yves. Vers la synthèse automatique de circuits à partir de graphes algorithmiques factorisés. In : JOURNÉES ADÉQUATION ALGORITHME ARCHITECTURE EN TRAITEMENT DU SIGNAL ET IMAGES, 5, Rocquencourt, 2000. *Actes...*

[DSP99] DSP STATION. *DSP Station.* Frontier Design : Danville, 1999. http ://www.frontierd.com

[DKL92] DURRIEU, G., KESSACI, K., LE MAÎTRE, M. Transe : an experimental transformation assistant for digital circuit design. In : IFIP WG 10.2/WG 10.5 WORKSHOP ON DESIGNING CORRECT CIRCUITS, 2, 1992. *Proceedings.* J. Staunstrup, R. Sharp (Ed.) s.l. : North-Holland, 1992.

[DD97] DUTRIEUX, Laurent, DEMIGNY, Didier. *Logique programmable* : architecture des FPGA et CPLD, méthodes de conception, le langage VHDL. Paris : Eyrolles, 1997. 234p.

[Eas98] EASYNET. *FOLDOC - Free On-Line Dictionary of Computing.* 1998. http ://www.easynet.de/resources/foldoc/index.html

[ELLS97] EDWARDS, S., LAVAGNO, L., LEE, E. A., SANGIOVANNI-VINCENTELLI, A. Design of embedded systems : formal models, validation, and synthesis. *Proceedings of the IEEE*, v.85, n.3, March 1997, p.366-390.

[ELS92] ENESSER, F., LAVARENNE, C., SOREL, Y. *Méthode chronométrique pour l'optimisation du temps de réponse des exécutifs SynDEx.* Rocquencourt : Institut National de Recherche en Iinformatique et en Automatique, 1992. 41p. (Rapport de Recherche INRIA, n. 1769)

[ET93] EPPLING, J., THOMAS, D. Multiple controller synthesis for reducing control path delays. In : SRC TECHCON'93, Atlanta, Sept. 1993. *Proceedings...*

[EHB93] ERNEST, Ralf, HENKEL, Jörg, BENNER, Thomas. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, v.10, n.4, Dec. 1993, p.64-75.

[Ess96] ESSER, Robert. *CodeSign version 1.0* : concepts and tutorial. Zurich : TIK/ETH, 1996.

[EL97a] EXEMPLAR LOGIC. *Galileo Extreme* : user's guide, release 4.1. Exemplar Logic, Inc., 1997.

[EL97b] ____. *Galileo Extreme* : synthesis and technology guide, release 4.1. Exemplar Logic, Inc., 1997.

[EL97c] ____. *HDL synthesis guide, release 4.1.* Exemplar Logic, Inc., 1997.

[Fai93] FAIRBAIRN, Doug. Beyond synthesis. *Computer Design*, May 1993, p.23-25.

[FOW87] FERRANTE, Jeanne, OTTENSTEIN, Karl J., WARREN, Joe D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, v. 9, n. 3, Jul. 1987, p. 319-349.

[FM82] FIDUCCIA, C. M., MATTHEYSES, R. A linear-time heuristic for improving network partitions. In : DESIGN AUTOMATION CONFERENCE, 17, 1982. *Proceedings...* p.175-181.

[Fin96] FINTA, Lucian. *Ordonnancement dans les systèmes multiprocesseurs.* Nice : Université de Nice-Sophia Antipolis, 1996. 167p. (Thèse de Doctorat en Informatique)

[Fox93] FOX, J. A higher level of synthesis. *IEEE Spectrum*, Mar. 1993, p.43-47.

[Fra95] FRANÇOIS, Marc. *Heuristiques d'ordonnancement et d'allocation de ressources dans un outil de synthèse d'architecture.* Évry : Université d'Évry, 1995. (Thèse de Doctorat en Informatique)

[Fur96] FURBER, S. B. Asynchronous logic. In : IBERCHIP'96, São Paulo, 1996. *Proceedings...* 8p.
http ://www.cs.man.ac.uk/amulet/publication/papers.html

[GK83] GAJSKI, D.D., KUHN, R.H. New VLSI tools. *IEEE Computer*, v.16, n.12, 1983, p.11-14.

[GL??] GAJSKI, Daniel D., LIS, Joseph S. VHDL modelling practices for synthesis. *SIGDA Newsletter*, v.18, n.3 & 4, p.76.

[GDWL92] GAJSKI, Daniel D., DUTT, Nikil D., WU, Allen C.-H., LIN, Steve Y.-L. *High-level synthesis* : introduction to chip and system design. Boston : Kluwer Academic Press, 1992.

[GVNG94] GAJSKI, Daniel D., VAHID, Frank, NARAYAN, Sanjiv, GONG, Jie. *Specification and design of embedded systems.* Englewood Cliffs : Prentice-Hall, 1994. p.1-14, 63-65, 88-93, 113, 145-149, 168-169.

[GV95] GAJSKI, Daniel D., VAHID, Frank. Specification and design of embedded hardware-software systems. *IEEE Design & Test of Computers*, v.12, n.1, Spring 1995, p.53-67.

[GVNG98a] GAJSKI, Daniel D., VAHID, Frank, NARAYAN, Sanjiv, GONG, Jie. System-level exploration with SpecSyn. In : DESIGN AUTOMATION CONFERENCE, 35, San Francisco, 1998. *Proceedings...* p.812-817.

[GVNG98b] ____. SpecSyn : an environnement supporting the specify-explore-refine paradigm for hardware/software system design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v.6, n.1, March 1998, p.84-100.

[Ghe88] GHEZAL, Nadia. *Quelques méthodes de répartition et d'ordonnancement de processus de traitement du signal sur un réseau de transputers.* Orsay : Université de Paris-Sud, 1988. 182p. (Thèse de Doctorat en Électronique)

[GMP*90] GHEZAL, N., MATIATOS, S., PIOVESAN, P., SOREL, Y., SORINE, M. *SynDEx, un environnement de programmation pour multiprocesseur de traitement du signal* : mécanismes de communication. Rocquencourt : Institut National de Recherche en Informatique et en Automatique, 1990. 47p. (Rapport de Recherche INRIA, n. 1236)

[God90] GODEFROID, P. Using partial orders to improve automatic verification methods. In : COMPUTER AIDED VERIFICATION WORKSHOP, 1990. *Proceedings...* CLARKE, E. M., KURSHAN, R. P. (Ed.) DIMACS Series in Discrete Mathematica and Theoretical Computer Sceince, 1991. p.321-340.

[Goe97a] GOERING, Richard. Abstract hardware names CEO, maps U.S. thrust. *EETIMES.COM* : the technology site for engineers and technical management, Mar. 1997.
http ://eetimes.com/news/97/951news/names.html

[Goe97b] ____. Formal tools heat up in U.S. *EETIMES.COM* : the technology site for engineers and technical management, Mar. 1997.
http ://eetimes.com/news/97/951news/formal.html

[GM95] GONDRAN, Michel, MINOUX, Michel. *Graphes et algorithmes.* 3 ed. rev. aug. Paris : Éd. Eyrolles, 1995. 588p.

[Gos95] GOSLIN, Gregory Ray. *A guide to using field programmable gate arrays (FPGAs) for application-specific digital signal processing performance.* San Jose : Xilinx, Inc., 1995.
http ://www.xilinx.com/appsweb.htm#DSP

[GLS99] GRANDPIERRE, T., LAVARENNE, C., SOREL,Y. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In : CODES'99–INTERNATIONAL WORKSHOP ON HARDWARE/SOFTWARE CODESIGN, 7, Rome, 1999. *Proceedings...* p.74-78.

[GS95] GSHWIND, Michael, SALAPURA, Valentina. A VHDL design methodology for FPGAs. In : FPL'95–INTERNATIONAL WORKSHOP ON

FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, 5, Oxford, 1995. *Proceedings.* Will Moore, Wayne Luk (Eds.) Berlin : Springer, 1995. p.208-217.

[Gup91] GUPTA, Aarti. *Formal hardware verification methods* : a survey. s.l. : Carnegie Mellon University, Oct. 1991. (Technical Report CMU-CS-91-193)

[GD92] GUPTA, Rajesh, DE MICHELI, Giovanni. *System synthesis via hardware-software co-design.* Stanford : Stanford University, Computer Systems Laboratory, 1992. (Technical Report CSL-TR-92-548)

[GD93] GUPTA, R.K., DE MICHELI, G. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, v.10, n.3, 1993, p.29-41.

[HCRP91] HALBWACHS, Nicolas, CASPI, Paul, RAYMOND, Paul, PILAUD, Daniel. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, v. 79, n. 9, Sept. 1991, p.1305-1321.

[HLR92] HALBWACHS, N., LAGNIER, F., RATEL, C. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering*, Special Issue on the Specification and Analysis of Real-Time Systems, Sept. 1992. 18p.

[Hal93] HALBWACHS, N. *Synchronous programming of reactive systems.* Kluwer Academic Publishers : Dordrecht Boston, 1993.

[Har87] HAREL, D. StateCharts : a visual formalism for complex systems. *Science of Programming*, v.8, n.3, June 1987, p.231-274.

[HE89] HAROUN, B., ELMASRY, M. Architectural synthesis for DSP silicon compiler. *IEEE Transactions on CAD*, v. 8, n. 4, Apr. 1989, p.431-447.

[Har95] HARRINGTON, Scott E. *FPGA synthesis tutorial.* Durham, USA : Department of Electrical & Computer Engineering, Duke University, 1995.
http ://www.ee.duke.edu/research/VHDL_tutorial/index.html

[HL89] HARTENSTEIN, R.W., LEMMERT, K. Sys3 : a CHDL-based systoliv synthesis system. In : IFIP'89–COMPUTER HARDWARE DESCRIPTION LANGUAGES AND THEIR APPLICATIONS,1989. *Proceedings...* DARRINGER, J.A., RAMMING, F.J. (Ed.), s.n.t.

[Hil85] HILFINGER, P. N. A high-level language and silicon compiler for digital signal processing. In : IEEE CUSTOM INTEGRATED CIRCUITS CONFERENCE, 1985. *Proceedings...* p.213-216.

[HB97] HINES, Ken, BORRIELLO, Gaetano. Dynamic communication models in embedded system co-simulation. In : DESIGN AUTOMATION CONFERENCE, 34, 1997. *Proceedings...* p.395-400.

[HR93] HOANG, Phu D., RABAEY, Jan M. Scheduling of DSP programs onto multiprocessors for maximum throughput. *IEEE Transactions on Signal Processing*, v. 41, n. 6, June 1993, p.2225-2235.

[Hpa99] HPADS. *HP Advanced Design System*. Hewlett-Packard, 1999. http ://www.tm.agilent.com/ãdsoview.html

[HW94] HUANG, S. C.-Y., WOLF, W. H. How datapath allocation affects controller delay. In : INTERNATIONAL SYMPOSIUM ON HIGH LEVEL SYNTHESIS, Niagara Falls, 1994. *Proceedings...* p.158-163.

[HOI94] HWANG, Ting-Ting, OWENS, Robert Michael, IRWIN, Mary Jane. Logic sysnthesis for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 13, n. 10, Oct. 1994, p.1280-1286.

[IEEE87] IEEE. *IEEE standard VHDL language reference manual*. s.l. : IEEE, 1987. (IEEE Std. 1076-1987)

[Ime96] IMEC. In : *Telecom, design methods, training & services*. Leuven : IMEC, 1996. p.102-112.

[Isr95] ISRAËL, Michel. *Vers un Atelier d'accueil générique pour la Synthèse ARchitecturale bâti autour de CENTAUR* : ASAR. Évry : Université d'Évry Val d'Essonne, 1995. 8 p.

[ID97] ISRAËL, Michel, DUPONT, Denis. De la spécification formelle au partitionnement matériel/logiciel. *Traitement du Signal*, v. 14, n. 6, spécial 1997, p.559-568.

[Jac94] JACQUEMIN, C. *Logique et mathématiques pour l'informatique et l'IA*. Paris : Masson, 1994. 238p.

[JJ??] JAMIER, R., JERRAYA, A. A. *Apollon, a datapath silicon compiler*. Grenoble : Institut National Polytechnique de Grenoble, s.d., 31p. (Rapport de Recherche INPG, n. 478)

[Jav00] JAVA. *The source for Java technology*. Sun Microsystems, Inc., 2000. http ://java.sun.com/

[JDKR97] JERRAYA, A. A., DING, H., KISSION, P., RAHMOUNI, M. *Behavioral synthesis and component reuse with VHDL*. s.l. : Kluwer Academic Publishers, 1997.

[JB93] JUNG, B., BURLESON, W. *A structured high-level description langage for VLSI array design*. Amherst : University of Massachussets, 1993. (Technical Report n. TR-93-CSE-7 - Dept. of Electrical and Computer Engineering - MA 01003)

[KL93] KALAVADE, Asawaree, LEE, Edward. A hardware-software codesign methodology for DSP applications. *IEEE Design & Test of Computers*, Sept. 1993, p.16-28.

[KL94] KALAVADE, Asawaree, LEE, Edward A. Manifestations of heterogeneity in hardware/software codesign. In : ACM/IEEE DESIGN AUTOMATION CONFERENCE, 31, June 1994. *Proceedings...* p.437-438.

[Kes92] KESSACI, Kamel. *Synthèse de circuits digitaux synchrones par transformations de programmes fonctionnels*. Toulouse : École National Supérieure de l'Aéronautique et de l'Espace, 1992. 180p. (Thèse de Doctorat en Informatique)

[KCBJ93] KISSION, P., CLOSSE, E., BERGHER, L., JERRAYA, A. Industrial experimentation of high-level synthesis environnement. In : EDAC'93, Paris, 1993. *Proceedings...*

[KDJ94] KISSION, P., DING, H., JERRAYA, A. A. Structured design methodology for high level design. In : DESIGN AUTOMATION CONFERENCE, 31, 1994.

[KS98] KOCIK, R., SOREL, Y. A methodology to design and prototype optimized embedded robotic systems. In : CESA'98–COMPUTATIONAL ENGINEERING IN SYSTEMS APPLICATIONS, Tunisia, 1998. *Proceedings...*

[KL70] KORNIGHAN, B., LIN, S. An efficient heuristic procedure for partitioning graphs. *Bell System Technology Journal*, v.49, 1970.

[KG94] KOULOHERIS, Jack L., GAMAL, Abbas El. *Effect of logic cell granularity on FPGA performance and density*. Yorktown Heights : IBM Research Division, 1994. 68p. (Research Report RC 19404)

[KVQZ??] KRALJIĆ, Ivan C., VERDIER, François S., QUÉNOT, Georges M., ZAVIDOVIQUE, Bertrand. *Multi-level prototyping for real-time image processing VLSI system synthesis*. Arcueil : ETCA/CREA/SP, s.d. 12p.

[Kri84] KRISHNAMURTHY, B. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, v. C-33, May 1984, p.438-446.

[KD90] KU, D., DE MICHELI, G. *HardwareC : a language for hardware design, v.2.0*. Stanford : Computer System Laboratory, Stanford University, 1990. (Technical Report CSL-TR-90-419)

[Kur97] KURSHAN, R. P. Formal verification in a commercial setting. In : DESIGN AUTOMATION CONFERENCE, 34, Anaheim, 1997. *Proceedings...* p.258-262.

[Lab00] LABVIEW. *LabVIEW*. National Instruments, 2000.
http ://www.ni.com/labview/

[L*90] LANNEER, D. et al. Architectural synthesis for medium and high throughput signal processing with the new CATHEDRAL environment. In : CAMPOSANO, R., WOLF, W. (Ed.) *High level VLSI synthesis*. Norwell : Kluwer Academic, 1990.

[Lar91] LAROUSSE. *Larousse Informatique* : english-french/français-anglais. Middlesex : Peter Colin Publishing/Larousse, 1991. 323p.

[LEAP94a] LAUWEREINS, Rudy, ENGELS, Marc, ADÉ, Marleen, PEPERSTRAETE, J.A. Rapid prototyping of digital signal processing systems with GRAPE-II. *DSP & Multimedia Technology*, Sept. 1994, p.22-31.

[LEAP94b] LAUWEREINS, Rudy, ENGELS, Marc, ADÉ, Marleen, PEPERSTRAETE, J.A. GRAPE-II : Graphical RApid Prototyping Environment for digital signal processing systems. In : ICSPAT'94, Dallas, 1994. *Proceedings...*, p.646-651.

[LS99] LAVAGNO, Luciano, SENTOVICH, Ellen. ECL : a specification environment for system-level design. In : DESIGN AUTOMATION CONFERENCE, 36, New Orleans, 1999. *Proceedings.* p.511-516.

[LSSS91a] LAVARENNE, C., SEGHROUCHNI, O., SOREL, Y., SORINE, M. The SynDEx software environment for real-time distributed systems design and implementation. In : EUROPEAN CONTROL CONFERENCE, July 1991. *Proceedings...* 7p.

[LSSS91b] LAVARENNE, C., SEGHROUCHNI, O., SOREL, Y., SORINE, M. SynDEx : un environnement de programmation pour applications de traitement du signal distribuées. In : COLLOQUE GRETSI, 13, 1991. *Actes...* 4p.

[LS92] LAVARENNE, Christophe, SOREL, Yves. Spécification, optimisation de performance et génération d'exécutif pour application temps-réel embarquée multi-processeur avec SynDEx. In : REAL-TIME EMBEDDED PROCESSING FOR SPACE APPLICATIONS, Les Saintes-Maries-de-la-Mer, 1992. *Proceedings...* 8p.

[LS93] LAVARENNE, Christophe, SOREL, Yves. Performance optimisation of multiprocessor real-time applications by graph transformations. In : PARCO'93–PARALLEL COMPUTING, Grenoble, 1993. *Proceedings...* 8p.

[LMPRS93] LAVARENNE, C., MILAN, C., PAINDAVOINE, M., RICHARD, G., SOREL, Y. Implantation d'algorithmes de traitement des images sur une architecture multi-DSP avec l'environnement d'aide à l'implantation SynDEx. In : COLLOQUE GRETSI, 14, Juan-les-Pins, Sept. 1993. *Actes...*

[LSMP94] LAVARENNE, C., SOREL, Y., MILAN, C., PAINDAVOINE, M. Implantation d'un algorithme de segmentation d'image sur une architecture multi-processeur avec l'environnement d'aide à l'implantation SynDEx. In : WORKSHOP ADÉQUATION ALGORITHMES ARCHITECTURES POUR TRAITEMENT DU SIGNAL ET DES IMAGES, 2, Grenoble, jan. 1994. *Actes...*

[LS94] LAVARENNE, Christophe, SOREL, Yves. Optimisation et génération d'exécutifs distribués temps réel pour algorithmes spécifiés avec langages synchrones. In : REAL-TIME SYSTEMS, Paris, Jan. 1994. *Actes...* 9p.

[LS96] LAVARENNE, Christophe, SOREL, Yves. Modèle unifié pour la conception conjointe logiciel-matériel. In : JOURNÉES ADÉQUATION ALGORITHME ARCHITECTURE EN TRAITEMENT DU SIGNAL ET IMAGES, 3, Toulouse, Jan. 1996. *Actes...* Toulouse : CNES, 1996.

[LS97] LAVARENNE, Christophe, SOREL, Yves. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, v. 14, n. 6, 1997, p.569-577.

[L*93] LEDEUX, S. et al. The Siemens high level synthesis system CALLAS. *IEEE Transactions on Very Large Scale Integrated Systems*, v.1, n.3, Sept. 1993, p.144-153.

[LM87] LEE, Edward, MESSERSCHIMITT, David. Synchronous data flow. *Proceedings of IEEE*, v. 79, n. 9, Sept. 1987.

[LM93] LEE, Edward, MESSERSCHIMITT, David. *An overview of the Ptolemy project*. Berkeley : University of California, Department of E.E. and C.S., Jan. 1993.

[LLGL91] LE GUERNIC, Paul, LE BORGNE, Michel, GAUTIER, Thierry, LE MAIRE, Claude. Programming real-time applications with SIGNAL. *Proceedings of IEEE*, n. 9, Sept. 1991. Rocquencourt : Institut National de Recherche en Informatique et en Automatique, 1991. 33p. (Rapport de Recherche INRIA, n. 1446)

[LG91] LE GUERNIC, Paul, GAUTIER, Thierry. Data-flow to von neumann : the SIGNAL approach, In : J.L. Gaudrot, L. Bic (Eds.) *Advanced topics in data-flow computing*. s.l. : Prentice Hall, 1991. p.413-438.

[LS83] LEISERSON, Charles E., SAXE, James B. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, v. 1, n. 1, 1983, p.41-67.

[LS91] LEISERSON, C. E., SAXE, J. B. Retiming synchronous circuitry. *Algorithmica*, v.6, 1991, p.5-35.

[LD96] LEMAÎTRE, Michel, DURRIEU, Guy. Synthèse architecturale par transformation de programmes synchrones purement fonctionnels. *Technique et Science Informatiques*, v.15, n.10, 1996, p.1345-1366.

[LeM97] LE MOËNNER, Patricia. *Contribution à la réalisation d'une chaîne complete pour la synthèse de circuits réguliers*. Rennes : Université de Rennes I, 1997. 179p. (Thèse de Doctorat en Informatique)

[LB91] LENGAUER, C., BARNET, M. The synthesis of sistolic programs. In : *Research directions in high-level parallel programming languages*. Rennes : IRISA-INRIA, Springer-Verlag, June 1991.

[LMQ91] LE VERGE, H., MAURAS, C., QUINTON, P. The Alpha language and its use for design of systolic arrays. *Journal of VLSI Signal Processing*, n. 3, 1991, p.173-182.

[LQ92] LE VERGE, H., QUINTON, P. Derivation of regular parallel algorithms with the ALPHA language. In : BANÂTRE, J.P., LE METAYER, D. (Ed.) *Research directions in high-level parallel programming languages*. s.l. : Springer-Verlag, 1992. p.298-308.

[LVM??] LIN, Bill, VERCAUTEREN, Steven, DE MAN, Hugo. *Embedded architecture co-synthesis and system integration*. Leuven : IMEC, s.d. 8p.

[Lin97] LIN, Youn-Long. Recent developments in high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, v.2, n.1, Jan. 1997, p.2-21.

[L*91] LIPPENS, P. et al. Phideo : a silicon compiler for high-speed algorithms. In : EDAC'91 – EUROPEAN CONFERENCE ON DESIGN AUTOMATION, 1991. *Proceedings*. s.l. : CS Press, 1991. p.436-441.

[LJ95] LOTUFO, Roberto, JORDÁN, Ramiro. WWW Khorosware on digital image processing. In : SIBGRAPI'95–SIMPÓSIO BRASILEIRO DE COMPUTAÇÃO GRÁFICA E PROCESSAMENTO DE IMAGENS, 8, 1995. *Proceedings...*

[Mad*97] MADSEN, J., GRODE, J., KNUDSEN, P. V., PETERSEN, M. E., HAXTHAUSEN. LYCOS : the Lyngby Co-Synthesis-System. *Design Automation for Embedded Systems*, v.2, n.2, 1997. 43p.

[Mal93] MALINIAK, L. ESDA boots CAE technology to higher levels. *Electronic Design*, Dec. 1993, p.61-72.

[Mal95] MALINIAK, Lisa. Can FPGA design be device independent ? *Electronic Design*, Jan. 23, 1995. p.41-52.

[Mar89] MARANINCHI, Florence. Argonaute, graphical description, semantics and verification of reactive systems by using a process algebra. In : WOKSHOP ON AUTOMATIC VERIFICATION METHODS FOR FINITE STATE SYSTEMS, Grenoble, 1989. *Proceedings...*

[Mar91] ——. The Argos language : graphical representation of Automata and description of reactive systems. In : IEEE WORKSHOP ON VISUAL LANGUAGES, Kobe, Japan, 1991. *Proceedings...*

[Mar99] MARCHAL, Pierre. Field programmable gate arrays. *Communications of the ACM*, v.42, n.4, April 1999. p.57-59.

[MSDP93] MARTIN, E., SENTIEYS, O., DUBOIS, H., PHILIPPE, J. GAUT : an architectural synthesis tool for dedicated signal processors. In : EUROPEAN DESIGN AUTOMATION CONFERENCE, 1993. *Proceedings...* p.14-19.

[Mar86] MARWEDEL, P. A new synthesis algorithm for the MIMOLA software system. In : DESIGN AUTOMATION CONFERENCE, 23, 1986. *Proceedings...* p.271-277.

[Mat00] MATLAB. *Matlab 5.3.1.* The MathWoks, Inc., 2000. http ://www.mathworks.com/products/matlab/

[Mau89] MAURAS, Christophe. *Alpha : un langage équationnel pour la conception et la programmation d'architectures systoliques.* Rennes : Université de Rennes I, 1989. (Thèse de Doctorat)

[McG??] MCGIBBON, John. *Review and critique of a problematic FPGA design.* s.l. : Memec Design Services, s.d. 4p.

[McF86] MCFARLAND, M.C. Using bottom-up design techniques in the systhesis of digital hardware from abstract behavioral descriptions. In : DESIGN AUTOMATION CONFERENCE, 23, 1986. *Proceedings.* s.l. : ACM/IEEE, 1986. p.474-480.

[MPC90] MCFARLAND, M. C., PARKER, A. C., CAMPOSANO, R. The high level synthesis of digital systems. *Proceedings of IEEE*, v.78, n.2, Feb. 1990, p.301-318.

[McM92] MCMILLON, K. *Symbolic model checking* : an approach to the state explosion problem. s.l. : Carnegie Mellon University, 1992. (PhD thesis)

[MC80] MEAD, C. A., CONWAY, L. A. *Introduction to VLSI systems.* s.l. : Ed. Addison-Wesley, 1980.

[M*96] MECHA, H. et al. A method for area estimation of data path in high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits Systems*, v.15, n.2, Feb. 1996, p.258-265.

[MG93] MENTOR GRAPHICS. *EDC - DSP Station.* s. l. : Mentor Graphics Corp., 1993. (Data Sheet)

[MG95] ____. *QuickVHDL user's and reference manual - software version 8.4_4.* s.l. : Mentor Graphics Corp., 1995.

[MRPSL93] MILAN, C., RICHARD, G., PAINDAVOINE, M., SOREL, Yves, LAVARENNE, Christophe. Implantation d'algorithmes de traitement d'images sur une architecture multi-DSP avec l'environnement d'aide à l'implantation SynDEx. In : COLLOQUE GRETSI, 15, Juan-les-Pins, 1993. *Actes...* 4p.

[Moh96] MOHAMMADI, S. *Techniques asynchrones pour la réalisation d'une machine massivement parallèle reconfigurable.* Orsay : Université de Paris-Sud, 1996. 166p. (Thése de Doctorat en Sciences)

[MPT85] MORISON, J.D., PEELING, N.E., THORP, T.L. The design rationale of ELLA, a hardware design and description language. In : COMPUTER HARDWARE DESCRIPTION LANGUAGES AND THEIR APPLICATIONS, 1985. *Proceedings.* s.l. : Elsevier Science Publishers, IFIP, 1985. p.303-320.

[Mos99] MOSES. *The Moses Project* : modelling, simulation, and evaluation of systems. Zurich : Eidgenössiche Technische Hochschule, 1999.
http ://www.tik.ee.ethz.ch/~moses/

[M*99] MOUSSA, I. et al. Comparing RTL ans behavioral design methodologies in the case of a 2M transistors ATM shaper. In : DESIGN AUTOMATION CONFERENCE, 36, New Orleans, 1999. *Proceedings...*

[NNRD94] NARASIMHAN, J., NAKAJIMA, K., RIM, C. S., DAHBURA, A. T. Yield enhancement of programmable ASIC arrays by reconfiguration of circuit placements. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 13, n. 8, Aug. 1994, p.976-986.

[New96] NEW, Bernie. *Estimating the performance of XC4000E adders and counters.* San Jose : Xilinx, Inc., 1996. (Application Note XAPP 018, v. 2.0)

[Nil88] NILSSON, Nils J. *Principes d'intelligence artificielle.* Toulouse : Cepadues-Éditions, 1988.

[OBr93] O'BRIAN, Kevin. *Compilation de silicium* : du circuit au système. Grenoble : Institut National Polytechnique de Grenoble, 1993. (Thèse de Doctorat)

[Oly95] OLYMPUS. *Stanford Olympus Synthesis System.* Stanford : Stanford University, 1995.
http ://akebono.stanford.edu/users/cad/synthesis/olympus.html

[Pag98] PAGE, Ian. Design of future systems. In : DESIGN, AUTOMATION & TEST IN EUROPE, Paris, 1998. *Proceedings...* p.343-349.

[PRD96] PAGET, M.-M., ROSSET-LOUERAT, M. M., DERIEUX, A. Apprendre avec Alliance. In : RENCONTRE FRANCOPHONE SUR LA DIDACTIQUE DE L'INFORMATIQUE, 5, Monastir, Tunisie, 1996. *Actes...* 19p.

[PA99] PAPACHRISTOU, Chris, ALZAZERI, Yusuf. A method of distributed controller design for RTL circuits. In : DESIGN, AUTOMATION & TEST IN EUROPE, Munich, 1999. *Proceedings...* p.774-775.

[PP88] PARK, A., PARKER, A. Sehwa : a software package for synthesis of pipelines from behavioral specifications. *IEEE Transactions on CAD*, v. 7, n. 3, Mar. 1988, p.356-370.

[PPM86] PARKER, A., PIZARRO, J., MLIMAR, A. MAHA : a program for datapath synthesis. In : DESIGN AUTOMATION CONFERENCE, 23, 1986. *Proceedings...* p.461-466.

[PKG86] PAULIN, P., KNIGHT, J., GIRCZYC, E. HAL : a multi-paradigm approach to automatic datapath synthesis. In : DESIGN AUTOMATION CONFERENCE, 23, 1986. *Proceedings...* p.263-270.

[Per93] PERRAUDEAU, Laurent. *MADMACS : un outil pour le dessin des masques de réseau réguliers intégrés.* Rennes : Université de Rennes I, 1993. (Thèse de Doctorat)

[Pie93] PIERRE, Laurence. VHDL description and formal verification of systolic multipliers. In : COMPUTER HARDWARE DESCRIPTION LANGUAGES AND THEIR APPLICATIONS, 1993. *Proceedings...* p.213-230.

[Pol98] POLIS. *Polis : a framework for hardware/software co-design of embedded systems.* Berkeley : University of California, 1998.
http ://www-cad.eecs.berkeley.edu/~polis/

[PR94] POTKONJAK, M., RABAEY, J. Exploring the algorithmic design space using high level synthesis. In : *VLSI design methodologies for DSP architectures.* s.l. : Kluwer Academic Publisher, 1994. p.131-167.

[Pra96] PRADO LOPES FILHO, Eudes. *Algorithmes de synthèse de circuits programmables basés sur des graphes de décision binaire.* Paris : Université de Paris VI, 1996. 86p. (Thèse de Doctorat)

[Prat86] PRATT, Vaughan. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, v. 15, n. 1, 1986, p.33-71.

[Pto99] PTOLEMY PROJECT. *Overview of the Ptolemy project.* LEE, Edward (Dir.) Berkeley : Dept. EECS/University of California, 1999. 18p. (ERL Technical Report UCB/ERL n. M99/37)

[Pto00] PTOLEMY PROJECT :*Heterogeneous modeling and design.* LEE, Edward (Dir.) Berkeley : UC Berkeley–EECS, 2000.
http ://ptolemy.eecs.berkeley.edu

[RP90] RABAEY, J., POTKONJAK, M. Ressource driven synthesis in the HYPER system. In : INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 1990. *Proceedings.* s.l. : ACM Press, 1990. p.2592-2595.

[RSS90] RAMAMRITHAM, Krithi, STANKOVIC, John A., SHIAH, Perng-Fei. Efficient scheduling algorithms for real-time multi-processor systems. *IEEE Transactions on Parallel and Distributed Systems*, v. 1, n. 2, Apr. 1990, p.184-194.

[RB94] RAMSTEIN, G., BAKOWSKI, P. A front-end environment for by-default specification, simulation and synthesis of array processors architectures. In : VHDL FORUM, 1994. *Proceedings...*

[RK94] RAO, D. S., KURDAHI, F. J. Controller and datapath trade-offs in hierarchical RT-level synthesis. In : INTERNATIONAL SYMPOSIUM ON HIGH LEVEL SYNTHESIS, Niagara Falls, 1994. *Proceedings...* p.152-157.

[Rat92] RATEL, C. *Définition et réalisation d'un outil de vérification formelle de programme LUSTRE* : le système LESAR. Grenoble : Université Joseph Fourier, 1992. (Thèse de Doctorat)

[Ren00] RENOIR. Wilsonville : Mentor Graphics, 2000.
http ://www.mentor.com/renoir/

[RRV97] RENAUDIN, Marc, ROBIN, Frédéric, VIVET, Pascal. AAAA : asynchronisme et adéquation algorithme architecture. *Traitement du Signal*, v. 14, n. 6, 1997. p.589-604.

[RSL93] REYNAUD, R., SOREL, Y., LAVARENNE, C. Spécification et validation à l'aide d'un langage synchrone d'un protocole d'appariement de données synchrones. In : COLLOQUE GRETSI, 14, Juan-les-Pins, 1993. *Actes...* 4p.

[RJ94] RIM, M., JAIN, R. Lower-bound performance estimation for high level synthesis scheduling problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v.13, n.4, Apr. 1994, p.451-458.

[Roc92] ROCHETAU, Frédéric. *Extension du langage LUSTRE et application à la conception de circuits* : le langage LUSTRE-IV et le système POLLUX. Grenoble : Institut National Polytechnique de Grenoble, 1992. (Thèse de Doctorat)

[RH92] ROCHETAU, F., HALBWACHS, N. Pollux : a Lustre based hardware design environment. In : QUINTON, P., ROBERT, Y. (Ed.) *Algorithms and parallels VLSI architectures II.* s.l. : Elsevier Science Publishers, 1992.

[RVBM96] ROMPAEY, Karl van, VERKEST, Diederik, BOLSENS, Ivo, DE MAN, Hugo. CoWare - a design environment for heterogeneous hardware/software systems. *Design Automation for Embedded Systems*, v.1, n.4, Oct. 1996, p.357-362.

[Ros93] ROSA DO NASCIMENTO, Fernando. *Méthodologie de conception d'architectures spécialisées* : une étude de cas. Rennes : Université de Rennes I, 1993. 155p. (Thèse de Doctorat en Informatique)

[Ros94] ROSSEEL, J. *Synthesis of application specific architectures for real-time regular algorithms.* Louvain, Belgique : IMEC, 1994. (PhD Thesis)

[RBI95] ROUSSEAU, F., BERGÉ, J-M., ISRAËL, M. Synthèse des méthodes et algorithmes de partitionnement logiciel/matériel. In : SYMPOSIUM SUR LES ARCHITECTURES NOUVELLES DE MACHINES, 3, Rennes, 1995. *Actes...* 12p.

[R*98] ROWSON, James E. et al. Hardware-software co-design : the next embedded system design challenge (Panel). In : DESIGN AUTOMATION CONFERENCE, 35, San Francisco, 1998. *Proceedings...* p.174-175.

[Rul00] RULEBASE. *RuleBase.* IBM Haifa Research Laboratory : 2000. www.haifa.il.ibm.com/projects/verification/RB_Homepage/

[Rus95] RUSHTON, Andrew. *VHDL for logic synthesis.* Berkshire : McGraw-Hill, 1995. 219p.

[Sab98] SABER. *Saber mixed-signal simulator.* Beaverton : Analogy, Inc., 1998. http ://www.analogy.com/Products/simulation/simulation.htm#Saber

[San89] SANCHIS, L. A. Multi-way network partitioning. *IEEE Transactions on Computers*, v. C-38, Jan. 1989, p.62-81.

[SK00] SCHIRRMEISTER, Frank, KROLIKOSKI, Stan. *The system-level HW/SW co-design challenge* : Cadence and partners craft state-of-the-art virtual component HW/SW co-design software. Cadence : 2000. http ://www.cadence.com/~vol5No1/

[SKC94] SCHLAG, Martine, KONG, Jackson, CHAN, Pak K. Routability-driven technology mapping for lookup table-based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 13, n. 1, Jan. 1994, p.13-26.

[Sch83] SCHOELLKOPF, J. P. *Lubrick : a silicon assembler and its application to data-path design for FISC.* Grenoble : Institut National Polytechnique de Grenoble, 1983. 21p. (Rapport de Recherche INPG, n. 363)

[Sed90] SEDGEWICK, Robert. *Algorithms in C.* Reading : Addison-Wesley, 1990. p.415-435, 471-483.

[SJ90] SHEERAN, M., JONES, G. Circuit design in Ruby. In : LECTURE NOTES ON RUBY FROM A SUMMER SCHOLL IN LYNGBY, Denmark, Sept. 1990.

[SJ91] SHEERAN, M., JONES, G. Deriving bit-serial circuits in Ruby. In : VLSI'91, IFIP TRANSACTIONS A-1, 1991. *Proceedings...* HALAAS, A., DENYER, P.B. (Ed.) s.l. : North-Holland, 1991.

[Sie95] SIÉ, Oumarou. *Génération automatique de dessins de masques de circuits réguliers intégrés.* Rennes : Université de Rennes I, 1995. (Thèse de Doctorat)

[Sim00] SIMULINK. *Simulink 3.0.1.* The MathWorks, Inc., 2000. http ://www-europe.mathworks.com/products/simulink/

[SHM96] SINGH, S., HOGG, J., MCAULEY, D. Expressing dynamic reconfiguration by partial evaluation. In : IEEE SYMPOSIUM ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1996. *Proceedings...*

[SS*97] SINGH, S., SHEERAN, M. et al. *Lava.* http ://www.dcs.gla-ac.uk/satnam/lava/mail.html, Apr. 1997.

[Smi??] SMITH, J. W. VHDL modelling for synthesis. *SIGDA Newsletter*, v. 18, n. 3 & 4, p.77-78.

[Sor92] SOREL, Yves. Langages synchrones et exécutifs distribués optimisés. In : WORKSHOP ADÉQUATION ALGORITHMES ARCHITECTURES POUR TRAITEMENT DU SIGNAL ET DES IMAGES, 1, Lannion, Sept. 1992. *Actes...* 11p.

[Sor94] SOREL, Yves. Massively parallel computing systems with real-time constraints : the "Algorithm Architecture Adequation" methodology. In : MASSIVELY PARALLEL COMPUTING SYSTEMS, Ischia, Italy, May 1994. *Proceedings...*

[Sor96] SOREL, Yves. Real-time embedded image processing application using the A^3 methodology. In : IEEE INTERNATIONAL CONFERENCE ON IMAGE PROCESSING, 1996. *Proceedings...*

[S*91] SHUNG, C. et al. An integrated CAD system for algorithm specific IC design. *IEEE Transactions on CAD*, Apr. 1991, p.447-482.

[Sta92] STANKOVIC, John A. Real-time computing. *Byte*, Aug. 1992, p.155-160.

[SV88] STOK, L., VAN DEN BERN, R. EASY : microprocessor architecture optimization. In : INTERNATIONAL WORKSHOP ON LOGIC AND ARCHITECTURE SYNTHESIS FOR SILICON COMPILERS, 1988. *Proceedings.* s.l. : Elsevier Science Publishers, 1988. p.313-328.

[Syn99] SYNDEX. *The AAA methodology and SynDEx.* Rocquencourt : INRIA, 1999. http ://www-rocq.inria.fr/syndex/

[Syn92] SYNOPSIS. *Synopsis VHDL compiler reference manual.* s.l. : Synopsis, 1992.

[Tex92] TEXAS. *Field programmable gate array* : application handbook. ASIC products edition. s.l. : Texas Instruments, 1992.

[Thi92] THIELE, L. Compiler techniques for massive parallel architectures. In : DEWILDE, P. (Ed.) *State of the art in computer science.* s.l. : Kluwer Academic Publisher, 1992.

[TM91] THOMAS, D.E., MORRBY, P. *The Verilog hardware description language.* s.l. : Kluwer Academic Publishers, 1991.

[TAS93] THOMAS, Donald, ADAMS, Jay, SCHMIT, Herman. A model and methodology for hardware-software codesign. *IEEE Design & Test of Computers*, Sept. 1993, p.6-15.

[Tor96] TORRES, Lionel. *Intégration de filtres numériques pour le traitement d'images* : du silicium au système reconfigurable. Montpellier : Université de Montpellier II, 1996. 177p. (Thèse de Doctorat en Électronique, Optronique et Systèmes)

[TS86] TSENG, Chia-Jeng, SIEWIOREK, Daniel P. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design*, v. CAD-5, n. 3, July 1986, p.379-395.

[Tuc95] TUCK, Barbara. Formal verification tools increase in number, improve in quality. *Computer Design's Electronic Systems* : Technology and Design, Oct. 1995.

[Tuc99] ——. New tools/methodologies increase verification productivity. *Computer Design's Electronic Systems* : Technology and Design. June, 1999.

[Tur36] TURING, A. M. On computable numbers, with an application to the Entscheindungs problem. In : LONDON MATHEMATICS SOCIETY, London, 1936. *Proceedings...*

[Uni92] UNIVERSITY OF CALIFORNIA. *VHDL synthesis system (VSS)* : user's manual. Version 5.0 ed. Irvine : University of California, 1992.

[VAH94] VAHID, F., GONG, J., GAJSKI, D. D. A binary-constraint search algorithm for minimizing hardware during hardware/software partitionning. In : Euro-DAC'94 – EUROPEAN DESIGN AUTOMATION CONFERENCE, 1994. *Proceedings...* p.214-219.

[VP90] VAN DAGEN, V., PETIT, M. *PRESAGE : a tool for the parallelisation of nested-loop programs.* s.l. : IFIP, North-Holland, 1990. p.341-360.

[VVB*93] VAN HOOF, J., VAN BELSEN, K., BOLSENS, I., GOOSSENS, G., DE MAN, H. *High-level synthesis for real-time digital signal processing.* s.l. : Kluwer Academic Publishers, 1993.

[VSZ??] VERDIER, François, SAFIR, Abdelhakim, ZAVIDOVIQUE, Bertrand. *A high level synthesis algorithm including control constraints.* Arcueil : ETCA–CREA–SP, s.d. 9p.

[VZ93] VERDIER, François, ZAVIDOVIQUE, Bertrand. A complete environment for global architecture synthesis. In : COMPUTER ARCHITECTURES FOR MACHINE PERPECTION WOKRSHOP, New Orleans, 1993. *Proceedings...*, p.77-81.

[VZ94] VERDIER, François S., ZAVIDOVIQUE, Bertrand. *A high level synthesis system for VLSI image processing applications.* Arcueil : ETCA/CREA/SP, 1994. 19p. (Submitted to International Journal of VLSI Design)

[Ver95] VERDIER, François. *Conception de logiciels d'optimisation sous contraintes d'architectures VLSI pour le traitement d'images* : le problème du contrôle. Orsay : Université de Paris-Sud 1995. 199p. (Thèse de Doctorat en Sciences)

[VZ97] VERDIER, François, ZAVIDOVIQUE, Bertrand. Des architectures intégrées pour la vision : synthèse automatique en trois exemples. *Traitement du Signal*, v.14, n.2, 1997, p.227-248.

[Vhd99] VHDL-AMS. *VHDL - Analog and Mixed Signal.* Beaverton : Analogy, Inc., 1999. http ://www.vhdlams.com/

[Vie00] VIEWLOGIC. *FPGA Express.* Viewlogic, 2000. http ://www.viewlogic.com/ftm/ep_designer.html

[V*96] VUILLEMIN, Jean E. et al. Programmable active memories : reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, v.4, n.1, Mar. 1996, p.59-69.

[Xil94] XILINX. *The programmable logic data book*. San Jose : Xilinx, Inc., 1994.

[Xil96] ____. *The programmable logic data book*. San Jose : Xilinx, Inc., 1996.

[Xil97] ____. *Speed metrics for high-performance FPGAs*. San Jose : Xilinx, Inc., 1997. (Application Brief XBRF015, v. 1.0)

[Xil99] ____. *XC4000E and XC4000X series field programmable gate arrays*. San Jose : Xilinx, Inc., 1999. p.6_5-6_72.
http ://www.xilinx.com/partinfo/databook.htm

[WM95] WEBER, J., MEAUDRE, M. *Circuits numériques et synthèse logique*; un outil : VHDL. Paris : Masson, 1995. p.161-200.

[Wil94] WILSON, R. et al. *The SUIF compiler system*. Stanford : Stanford University, 1994.

[WKWP99] WOLFF, Francis G., KNIESER, Michael J., WEYER, Dan J., PAPACHRISTOU, Chris. Using codesign techniques to support analog functionality. In : CODES'99–INTERNATIONAL WORKSHOP ON HARDWARE/SOFTWARE CODESIGN, 7, Rome, 1999. *Proceedings...* p.79-84.

[W*90] WOUDSMA, R. et al. Piramid : an architecture-driven silicon compiler for DSP applications. In : INTERNATIONAL SYNPOSIUM ON CIRCUITS AND SYSTEMS, 1990. *Proceedings...* p.2596-2600.

[WCC95] WU, Q., CHEN, C.Y.R., CARLSON, B.S. LILA : layout generation of iterative logic arrays. *IEEE Transaction of Computer-Aided Design of Integrated Circuits and Systems*, v. 14, n. 11, Nov. 1995, p. 1359-1369.

[ZG99] ZHU, Jianwen, GAJSKI, Daniel. A unified formal model of ISA and FSMD. In : CODES'99–INTERNATIONAL WORKSHOP ON HARDWARE/SOFTWARE CODESIGN, 7, Rome, 1999. *Proceedings...* p.121-125.

[Zim80] ZIMMERMANN, G. MDS : the Mimola design method. *Journal of Digital Systems*, v. 4, n.3, p.337-369.

BIBLIOGRAPHIE

Glossaire

Allocation : transformation appliquée à une spécification, qui consiste à assigner, à chaque opération, variable ou chemin de communication de cette spécification, la ressource matérielle (opérateurs, connexions) qui l'implante.

ALPHA : outil de synthèse algorithmique dédié aux circuits spécifiques [LeM97].

Anti-dépendance : relation de consommation entre deux frontières. Par exemple, une frontière FF_2 dépend d'une frontière FF_1, si FF_2 requiert que les opérations correspondant à F_1 soient exécutées avant celles délimitées par FF_2. FF_1 fournit donc ses données d'entrée à FF_2. Ainsi, FF_1 est une frontière anti-dépendante par rapport à FF_2 et FF_2 est une frontière dépendante par rapport à FF_1.

Architecture régulière : architecture constituée par un assemblage de réseaux de processeurs élémentaires répétés régulièrement.

Backtracking (retour en arrière) : schéma (ou algorithme) utilisé pour résoudre une série de sous-problèmes, où chacun peut avoir de multiples solutions possibles et où la solution retenue pour un sous-problème peut affecter les solutions possibles des sous-problèmes précédents. Cet algorithme peut être utilisé pour implanter du non-déterminisme. Il effectue d'abord une recherche en profondeur (*depth-first search*) dans l'espace de solutions [Eas98].

Boucle critique : chemin bouclé le plus long dans un graphe. Il détermine la cadence de l'implantation.

Cadence : intervalle de temps qui sépare la réception par le système de deux stimuli consécutifs.

Chemin critique : chemin sans boucle le plus long dans un graphe. Il détermine la période de l'horloge de l'application ou la latence s'il n'y pas de boucle.

Chemin de données : (1) chemin composé par les unités fonctionnelles et le bus de données internes à une CPU [Eas98] ou ; (2) chemin composé par les parties purement opératives et de trafic de données (logique combinatoire et bus) d'une architecture ou d'un circuit.

Chemin de contrôle : chemin composé par les parties non-combinatoires (contrôle de registres, multiplexeurs et démultiplexeurs) d'une architecture ou d'un circuit.

Circuit spécifique : circuit (ASIC) constitué d'une partie opérative réalisant des calculs dont la synchronisation est implantée dans la partie contrôle [LeM97].

Combinatoire : un opérateur (ou une ressource) combinatoire est un opérateur qui ne possède pas de registres internes.

Co-design : voir *Conception conjointe matériel/logiciel.*

Conception : acte de définir un système ou un sous-système en fonction de contraintes technologiques, physiques ou de l'application [Pto99].

Conception conjointe matériel/logiciel : désigne la conception de systèmes comportant une partie matérielle, constituée d'un ou plusieurs circuits spécifiques (ASIC, composants reconfigurables), et une partie logicielle qui s'exécute sur une architecture à base de processeurs standards et/ou spécifiques (microcontrôleurs, DSP). La conception conjointe peut être interprétée comme la spécification, la validation et l'exploration des différentes alternatives de conception d'un système mixte matériel/logiciel, dans le but d'optimiser des critères de coût et/ou de performances [Bel94].

Concurrence : propriété d'une application. La concurrence peut être *spatiale* (parallélisme), où les tâches peuvent être exécutées simultanément par plusieurs processeurs, ou *temporelle* (pipeline), où une chaîne de tâches est séparée en plusieurs stages, chacun traitant les résultats obtenus du stage précédent [HR93].

Control path : voir *Chemin de contrôle.*

Datapath : voir *Chemin de données.*

Deadlock (interblocage) : situation dans laquelle deux processus ou plus se trouvent incapables d'évoluer, chacun d'entre eux attend une action de la part de l'un des autres [Eas98].

Décomposition de sommets : transformation appliquée aux sommets d'un graphe, qui permet de passer d'une représentation de gros grain à une représentation de grain fin.

Dépendance de données : relation qui décrit le transfert d'une donnée d'une opération productrice à une opération consommatrice.

Dépendance de flot : relation qui décrit le transfert d'un flot de données d'une opération productrice à une opération consommatrice.

Description matérielle : résultat d'une traduction matérielle structurelle (*mapping*) appliquée à une spécification algorithmique.

Description synthétisable : description dont la réalisation matérielle fait l'objet d'un processus automatique, en faisant appel à des outils de synthèse logique ou de synthèse architecturale [ACO92].

Distribution : allocation spatiale des sommets du graphe logiciel aux sommets du graphe matériel et allocation spatiale des dépendances de données aux liens physiques.

Factorisation : transformation appliquée à une spécification, qui consiste à remplacer, dans un graphe, un motif d'opérations répétitif par un seul exemplaire du motif, délimité par des sommets spéciaux "frontières" : F, J, I et D. Cette délimitation est appelée *frontière de factorisation.*

FPGA : circuit composé d'un ensemble de blocs logiques disposés dans une matrice. La fonctionnalité de ces blocs et la façon dont ils sont interconnectés sont configurables. Les blocs logiques contiennent des ressources combinatoires et séquentielles de l'architecture [Pra96].

Frontière de factorisation : abstraction qui regroupe des opérateurs frontières de factorisation qui factorisent un même motif répétitif.

Granularité : propriété d'une spécification algorithmique, qui indique le niveau de détails représenté par les sommets : un sommet peut représenter un système complet ou un sous-système quelconque (gros grain), ou il peut représenter une porte ou une cellule logique (grain fin).

Graphe de dépendances : type de graphe orienté dont les arcs expriment les dépendances de données entre ses sommets.

Graphe factorisé de dépendances de données : type de graphe dont le motifs répétitifs ont été factorisés.

Graphe flot de données : voir *Graphe de dépendances.*

Graphe périodique : type de graphe qui est répété (ou itéré) plusieurs fois. Voir *Graphe factorisé de dépendances de données.*

Hardware/software co-design : voir *Conception conjointe matériel/logiciel.*

Heuristiques : voir *Méthodes heuristiques de recherche.*

Implantation matérielle : résultat de l'application de la synthèse RTL ou de la synthèse logique à une description matérielle.

Implantation optimisée : implantation qui respecte les contraintes temporelles (latence, temps de réponse), tout en minimisant la consommation de ressources matérielles (surface).

Informations heuristiques : informations qui dépendent du domaine de l'application, utilisées pour réduire la recherche des solutions [Nil88].

Latence : (1) intervalle de temps entre la réception de la donnée engendrée par un stimulus et l'émission de la donnée engendrée par une réaction à l'issue du traitement ; (2) longueur du chemin critique du graphe algorithmique, dont les sommets sont étiquetés par les durées d'exécution des opérateurs correspondants, y compris celles des communications inter-processeurs.

LCA : format utilisé dans l'outil *XACT* de *Xilinx* pour décrire les circuits après le placement et le routage [Pra96].

Logiciel embarqué : logiciel qui réside dans des dispositifs embarqués [Pto99].

Méthodes heuristiques de recherche : procédures de recherche basées sur des informations heuristiques [Nil88].

Modélisation : acte de représenter formellement un système ou un sous-système [Pto99].

Motif de factorisation : motif d'opérations répétitif, délimité par des sommets spéciaux "frontières".

Motif répétitif : voir *Motif de factorisation.*

Optimisation en délai : réalisation d'un circuit en minimisant le nombre de blocs logiques traversés par les signaux.

Optimisation en surface : réalisation d'un circuit avec le minimum de blocs logiques.

Ordonnancement : (1) du point de vue matériel, allocation temporelle des sommets du graphe matériel distribué aux sommets du graphe matériel et allocation temporelle des dépendances de données aux liens physiques. Cette transformation consiste à étudier les relations d'ordre de tous les sous-graphes engendrés par les opérations de calcul ou de communication contraintes à être exécutées sur des unités de calcul ou communication ; (2) du point de vue logiciel, consiste à déterminer une date d'exécution pour chaque opération, en essayant de minimiser le temps d'exécution de l'algorithme.

Parallélisme : voir *Concurrence*.

Parallélisme disponible : propriété d'une architecture qui exprime sa capacité d'exécuter des opérations en parallèle.

Parallélisme potentiel : propriété d'un algorithme qui exprime sa capacité d'être exécuté de façon plus ou moins parallèle.

Partitionnement : partage d'un système en une partie logicielle et une partie matérielle afin de minimiser une certaine fonction de coût [Bel94]. Voir aussi *Distribution* et *Allocation*.

Période d'échantillonnage : voir *Cadence*.

Pipeline : voir *Concurrence*.

Placement : action qui consiste à attribuer une position physique à un élément.

Processeur élémentaire : bloc de matériel dont les constituants sont connectés localement dans le but d'effectuer un traitement élémentaire sur les données du programme [LeM97].

Retiming : transformation appliquée à un graphe matériel, dont le but est de réduire son chemin critique par l'intermédiaire de l'insertion de registres entre les deux registres (entrée et sortie) du chemin critique.

Routage : (1) transformation appliquée à un graphe matériel dont le but est de définir les différentes routes, permettant de communiquer d'un composant matériel (processeur ou cellule logique) à l'autre, lorsque ceux-ci ne sont pas reliés ; (2) action qui consiste à générer les liens entre les éléments d'une liste d'interconnexions.

Séquentiel : un opérateur (ou une ressource) séquentiel est un opérateur qui contient des registres internes cadencés par un signal d'horloge.

Spécification algorithmique : description comportementale d'une application sous la forme algorithmique, en utilisant des langages procéduraux, des langages flot de données synchrones, des graphes, etc.

Synthèse : procédé automatisé permettant de faire passer la représentation d'une architecture du domaine comportemental au domaine structurel. La synthèse est la transformation d'une représentation de haut-niveau en une représentation de bas niveau, qui est plus proche de l'implantation [COH*99].

La synthèse désigne également les étapes de transformations internes à un domaine [LeM97].

Synthèse algorithmique : procédé réalisée directement par un synthétiseur de circuit ou par raffinements successifs des descriptions au sein du domaine comportemental.

Synthèse architecturale : procédé automatisé qui génère, à partir d'une spécification exprimée dans le domaine comportemental, la description d'un circuit en termes de blocs interconnectés. Elle vise une description structurelle, dont les éléments ont un équivalent matériel. Tous les éléments paramétrables y sont caractérisés et peuvent fournir une estimation de leurs propriétés temporelles, topologiques et électriques [ACO92, LeM97].

Synthèse comportementale : voir *Synthèse algorithmique.*

Synthèse haut niveau : type de synthèse qui englobe les synthèses architecturales partant des niveaux algorithmique ou système.

Synthèse physique : type de synthèse qui génére une implantation sous la forme d'une *netlist* de portes ou de modèles de *layout.*

Synthèse logicielle : voir *Synthèse de programmes.*

Synthèse logique : type de synthèse qui travaille à partir d'équations booléennes ou de machines à états finies. Son passage au domaine structurel produit une description en termes de portes logiques et de bascules. Elle vise une description purement structurelle, dont les éléments sont des cellules standards pour circuits intégrés ou une programmation particulière de composants de type FPGA ou EPLD [ACO92, LeM97].

Synthèse niveau transfert de registres : type de synthèse qui part d'une description des valeurs des états des éléments de mémorisation (registres) à l'aide de fonctions de transfert de valeurs entre ces registres.

Synthèse de programmes : type de synthèse où le système informatique est capable de lire une description de très haut niveau de ce qui le programme doit accomplir et de produire ce même programme [Nil88].

Synthèse système : type de synthèse dont le but est d'exprimer la spécification de l'architecture sous forme de processus communicants ; de s'assurer de sa pertinence ; d'évaluer certains critères, comme la performance de l'architecture ; et de partitionner la description entre diférentes entités physiques interconnectées, qui font l'objet ensuite d'une synthèse au niveau algorithmique [LeM97].

Système embarqué : classe de systèmes qui sont caractérisés par leur interaction avec l'environnement et par leur besoin de respecter des contraintes temps réel. Un système embarqué doit réagir à des événements externes dans un temps borné ou traiter les données dans une certaine cadence [Cod99].

Système réactif : voir *Système embarqué.*

Système temps réel : classe de systèmes réactifs ou activités de traitement de l'information qui doivent répondre a des *stimuli* externes, tout en respectant un délai spécifié et fini [BW97].

Temps de réponse : voir *Latence*.

Traduction structurelle : interprétation directe de la spécification algorithmique en description matérielle. Pour cela, une correspondance opération-opérateur est effectuée.

XNF : format de *netlist Xilinx* qui permet la description d'un réseau de portes. Il s'agit d'une connectique abstraite sans information concernant le placement [Pra96].

Annexe A

Étude du Produit Matrice-Vecteur

Dans cet annexe, nous présentons une analyse détaillée de l'implantation matérielle à partir de la spécification totalement factorisé du produit matrice-vecteur (PMV) entre une matrice de 6×6 éléments et un vecteur de 6 éléments, codés sur 3 bits, montré par la figure 2.26. Cette spécification est décrite par l'éq. 2.13, que nous réproduisons ci-dessous pour $m = n = 6$:

$$C = \left[\sum_{j=1}^{6} a_{ij} b_j \right]_{i=1}^{6} \tag{A.1}$$

A.1 Défactorisations possibles

Cette spécification factorisée du PMV peut avoir 36 défactorisations différentes, en fonction des facteurs de défactorisation k_2 et k_3 appliqués respectivement aux frontières FF_2 et FF_3, comme le montre le tableau A.1.

Défactorisation partielle de FF_3

La défactorisation partielle de la frontière FF_3 du graphe algorithmique du PMV factorisé (figure 2.26) par un facteur de défactorisation $k = 2$, peut être représentée par l'éq. A.2. Le graphe algorithmique correspondant à cette défactorisation partielle est montré par la figure A.1.

TAB. A.1: Défactorisations du PMV

Spécification	k_2	k_3	*Remarque*
$(\infty, 6/1, 6/1)$	1	1	totalement factorisée
$(\infty, 6/1, 6/2)$	1	2	défactor. partielle FF_3
$(\infty, 6/1, 6/3)$	1	3	défactor. partielle FF_3
$(\infty, 6/1, 6/4)$	1	4	défactor. partielle FF_3
$(\infty, 6/1, 6/5)$	1	5	défactor. partielle FF_3
$(\infty, 6/1, 6/6)$	1	6	défactor. totale de FF_3
$(\infty, 6/2, 6/1)$	2	1	défactor. partielle FF_2
$(\infty, 6/2, 6/2)$	2	2	défac. part. FF_2 et FF_3
$(\infty, 6/2, 6/3)$	2	3	défac. part. FF_2 et FF_3
$(\infty, 6/2, 6/4)$	2	4	défac. part. FF_2 et FF_3
$(\infty, 6/2, 6/5)$	2	5	défac. part. FF_2 et FF_3
$(\infty, 6/2, 6/6)$	2	6	défactor. totale de FF_3
$(\infty, 6/3, 6/1)$	3	1	défactor. partielle FF_2
$(\infty, 6/3, 6/2)$	3	2	défac. part. FF_2 et FF_3
$(\infty, 6/3, 6/3)$	3	3	défac. part. FF_2 et FF_3
$(\infty, 6/3, 6/4)$	3	4	défac. part. FF_2 et FF_3
$(\infty, 6/3, 6/5)$	3	5	défac. part. FF_2 et FF_3
$(\infty, 6/3, 6/6)$	3	6	défactor. totale de FF_3
$(\infty, 6/4, 6/1)$	4	1	défactor. partielle FF_2
$(\infty, 6/4, 6/2)$	4	2	défac. part. FF_2 et FF_3
$(\infty, 6/4, 6/3)$	4	3	défac. part. FF_2 et FF_3
$(\infty, 6/4, 6/4)$	4	4	défac. part. FF_2 et FF_3
$(\infty, 6/4, 6/5)$	4	5	défac. part. FF_2 et FF_3
$(\infty, 6/4, 6/6)$	4	6	défactor. totale de FF_3
$(\infty, 6/5, 6/1)$	5	1	défactor. partielle FF_2
$(\infty, 6/5, 6/2)$	5	2	défac. part. FF_2 et FF_3
$(\infty, 6/5, 6/3)$	5	3	défac. part. FF_2 et FF_3
$(\infty, 6/5, 6/4)$	5	4	défac. part. FF_2 et FF_3
$(\infty, 6/5, 6/5)$	5	5	défac. part. FF_2 et FF_3
$(\infty, 6/5, 6/6)$	5	6	défactor. totale de FF_3
$(\infty, 6/6, 6/1)$	6	1	défactor. totale de FF_2
$(\infty, 6/6, 6/2)$	6	2	défactor. totale de FF_2
$(\infty, 6/6, 6/3)$	6	3	défactor. totale de FF_2
$(\infty, 6/6, 6/4)$	6	4	défactor. totale de FF_2
$(\infty, 6/6, 6/5)$	6	5	défactor. totale de FF_2
$(\infty, 6/6, 6/6)$	6	6	totalement défactorisée

$$C = \left[\sum_{j=1}^{3} a_{ij} b_j + \sum_{j=4}^{6} a_{ij} b_j \right]_{i=1}^{6} \tag{A.2}$$

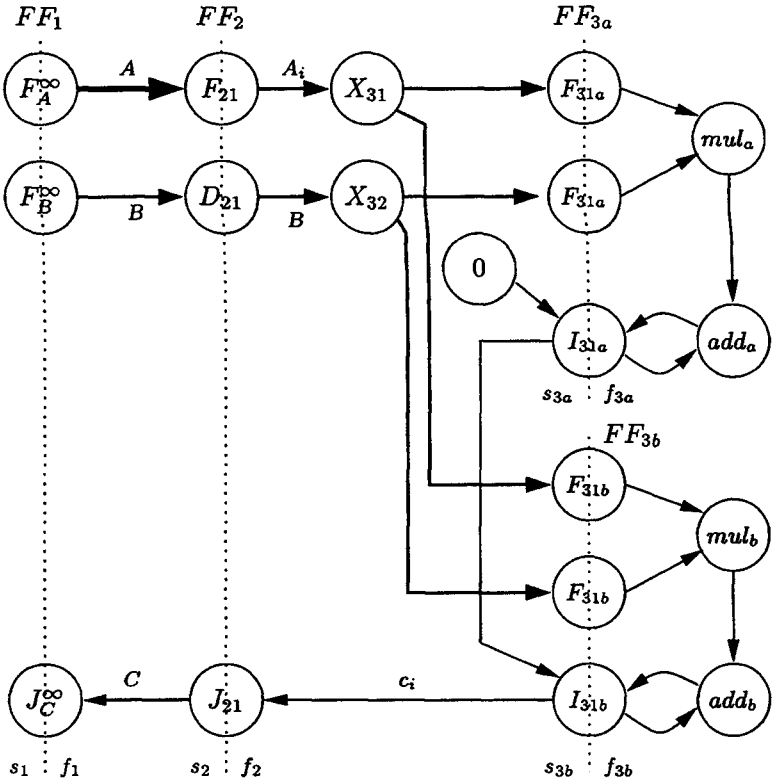


FIG. A.1: Graphe algorithmique défactorisé du PMV (∞,6/1,6/2)

À partir du graphe algorithmique (figure A.1), nous construisons le graphe de relations entre frontières de factorisation (figure A.2). L'interconnexion des signaux de requête et d'acquittement est effectuée par l'intermédiaire de l'analyse des relations de voisinage entre les frontières de factorisation FF_1, FF_2, FF_{3a} et FF_{3b} :

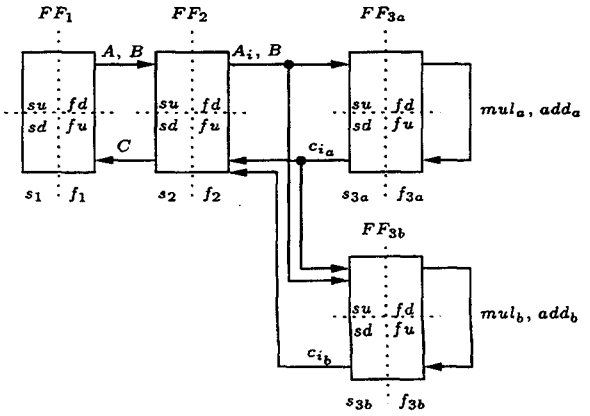


FIG. A.2: Relations de voisinage entre frontières du PMV défactorisé (∞,6/1,6/2)

1. le point de départ est le graphe de relations de voisinage (figure A.2) ;
   - FF_1 est productrice (côté "lent") par rapport à FF_2,

- FF_1 est consommatrice (côté "lent") par rapport à FF_2,
- FF_2 est consommatrice (côté "rapide") par rapport à FF_1,
- FF_2 est productrice (côté "rapide") par rapport à FF_1,
- FF_2 est productrice (côté "lent") par rapport à FF_{3a},
- FF_2 est consommatrice (côté "lent") par rapport à FF_{3a},
- FF_{3a} est productrice (côté "rapide") par rapport à FF_2,
- FF_{3a} est consommatrice (côté "rapide") par rapport à FF_2,
- FF_2 est productrice (côté "lent") par rapport à FF_{3b},
- FF_2 est consommatrice (côté "lent") par rapport à FF_{3b},
- FF_{3b} est productrice (côté "rapide") par rapport à FF_2,
- FF_{3b} est consommatrice (côté "rapide") par rapport à FF_2,
- FF_{3a} est productrice (côté "lent") par rapport à FF_{3b},
- FF_{3b} est consommatrice (côté "rapide") par rapport à FF_{3a},
- FF_{3a} est productrice (côté "rapide") par rapport à FF_{3a},
- FF_{3a} est consommatrice (côté "rapide") par rapport à FF_{3a},
- FF_{3b} est productrice (côté "rapide") par rapport à FF_{3b},
- FF_{3b} est consommatrice (côté "rapide") par rapport à FF_{3b},

2. la frontière FF_1 est une frontière "infinie", étant à la fois, la frontière la plus en amont et la plus en aval du graphe algorithmique. Pour cette frontière "infinie", il faut générer les signaux d'entrée rfu_1, afd_1, rsu_1 et asd_1 :
   - productrice côté "rapide" : FF_2

$$?\,rfu_1 = !\,rsd_2 \tag{A.3}$$

   - consommatrice côté "rapide" : FF_2

$$?\,afd_1 = !\,asu_2 \tag{A.4}$$

   - producteur côté "lent" : l'environnement

$$?\,rsu_1 = 1 \tag{A.5}$$

   - consommateur côté "lent" : l'environnement

$$?\,asd_1 = 1 \tag{A.6}$$

3. la frontière FF_2 est une frontière "finie". Pour cette frontière, il faut générer les signaux d'entrée rfu_2, afd_2, rsu_2 et asd_2 :

- productrices côté "rapide" : FF_{3a} et FF_{3b}

$$?\,rfu_2 =!\,rsd_{3a}\ \&\ !\,rsd_{3b} \tag{A.7}$$

- consommatrices côté "rapide" : FF_{3a} et FF_{3b}

$$?\,afd_2 =!\,asu_{3a}\ \&\ !\,asu_{3b} \tag{A.8}$$

- productrice côté "lent" : FF_1

$$?\,rsu_2 =!\,rfd_1 \tag{A.9}$$

- consommatrice côté "lent" : FF_1

$$?\,asd_2 =!\,afu_1 \tag{A.10}$$

4. la frontière FF_{3a} est une frontière "finie". Il faut générer les signaux d'entrée rfu_{3a}, afd_{3a}, rsu_{3a} et asd_{3a} :
   - productrice côté "rapide" : FF_{3a}

$$?\,rfu_{3a} =!\,rfd_{3a} \tag{A.11}$$

   - consommatrice côté "rapide" : FF_{3a}

$$?\,afd_{3a} =!\,afu_{3a} \tag{A.12}$$

   - productrice côté "lent" : FF_2

$$?\,rsu_{3a} =!\,rfd_2 \tag{A.13}$$

   - consommatrices côté "lent" : FF_1 et FF_{3b}

$$?\,asd_{3a} =!\,afu_2\ \&\ !\,asu_{3b} \tag{A.14}$$

5. la frontière FF_{3b} est une frontière "finie". Il faut générer les signaux d'entrée rfu_{3b}, afd_{3b}, rsu_{3b} et asd_{3b} :
   - productrice côté "rapide" : FF_{3b}

$$?\,rfu_{3b} =!\,rfd_{3b} \tag{A.15}$$

- consommatrice côté "rapide" : FF_{3b}

$$?afd_{3b} =! afu_{3b} \tag{A.16}$$

- productrices côté "lent" : FF_2 et FF_{3a}

$$?rsu_{3b} =! rfd_2 \,\&\, !rsd_{3a} \tag{A.17}$$

- consommatrice côté "lent" : FF_1

$$?asd_3 =! afu_2 \tag{A.18}$$

La figure A.3 représente l'implantation matérielle du PMV factorisé correspondant à la spécification algorithmique représentée par la figure A.1. Le chemin de données est constitué des opérateurs frontière de factorisation (F, D, J et I) et des opérateurs combinatoires (X et M) : les chemins de données à droite de FF_{3a} et FF_{3b} se sont composés des opérateurs combinatoires mul_a / add_a et mul_b / add_b, respectivement. Les autres chemins de données se sont composés trivialement des interconnexions entre les opérateurs frontière de factorisation. Le chemin de contrôle est constitué par les unités de contrôle UC_1, UC_2, UC_{3a} et UC_{3b}, et par leurs signaux de contrôle r, a, cpt, en.

La frontière FF_3 de la figure 2.26 peut encore être défactorisée par d'autres facteurs de défactorisation ($k = 3$, $k = 4$, $k = 5$ ou $k = 6$), selon les équations suivantes :

- $k = 3$

$$C = \left[\sum_{j=1}^{2} a_{ij}b_j + \sum_{j=3}^{4} a_{ij}b_j + \sum_{j=5}^{6} a_{ij}b_j\right]_{i=1}^{6} \tag{A.19}$$

- $k = 4$

$$C = \left[\sum_{j=1}^{2} a_{ij}b_j + \sum_{j=3}^{4} a_{ij}b_j + a_{i5}b_5 + a_{i6}b_6\right]_{i=1}^{6} \tag{A.20}$$

- $k = 5$

$$C = \left[\sum_{j=1}^{2} a_{ij}b_j + a_{i3}b_3 + a_{i4}b_4 + a_{i5}b_5 + a_{i6}b_6\right]_{i=1}^{6} \tag{A.21}$$

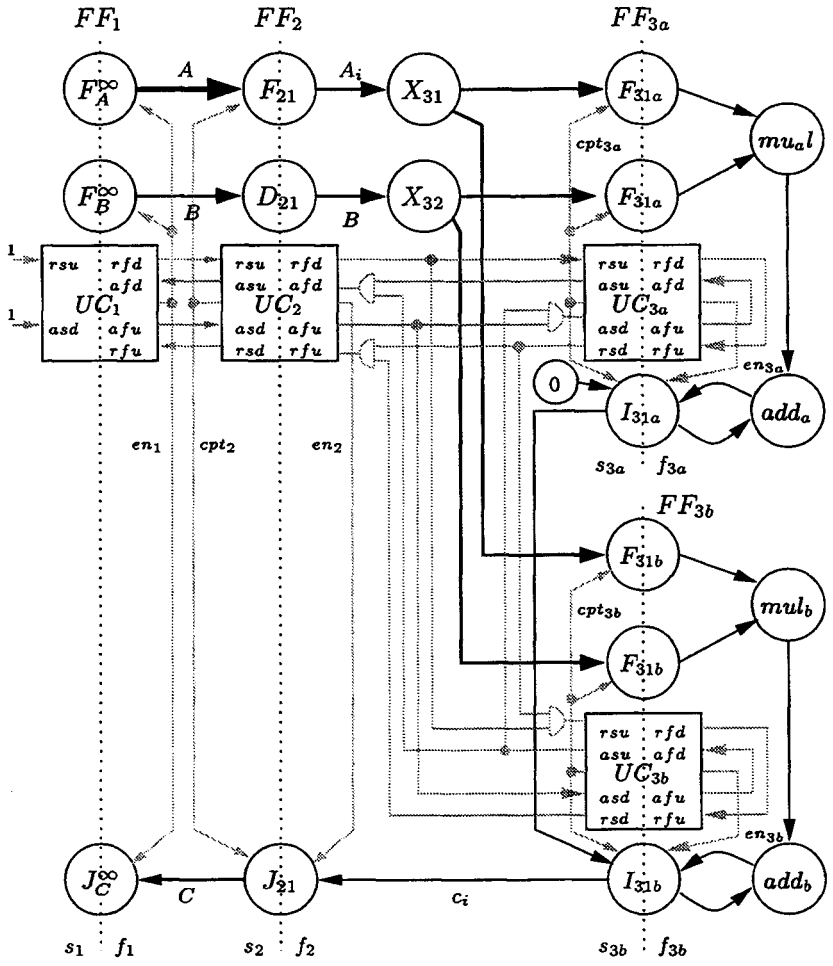


FIG. A.3: Graphe matériel défactorisé du PMV (∞,6/1,6/2)

- $k = 6$

$$C = \left[a_{i1}b_1 + a_{i2}b_2 + a_{i3}b_3 + a_{i4}b_4 + a_{i5}b_5 + a_{i6}b_6 \right]_{i=1}^{6} \tag{A.22}$$

La défactorisation de la frontière FF_3 par un facteur $k = 6$ implique sa défactorisation totale, où tous les opérateurs frontières de factorisation qui la délimitaient ont été remplacés par des opérateurs combinatoires, comme le montre la figure A.4.

À partir du graphe algorithmique (figure A.4), nous construisons le graphe de relations entre frontières de factorisation (figure A.5). L'interconnexion des signaux de requête et d'acquittement est effectuée par l'intermédiaire de l'analyse des relations de voisinage entre les frontières de factorisation FF_1 et FF_2 :

1. le point de départ est le graphe de relations de voisinage (figure A.5) ;

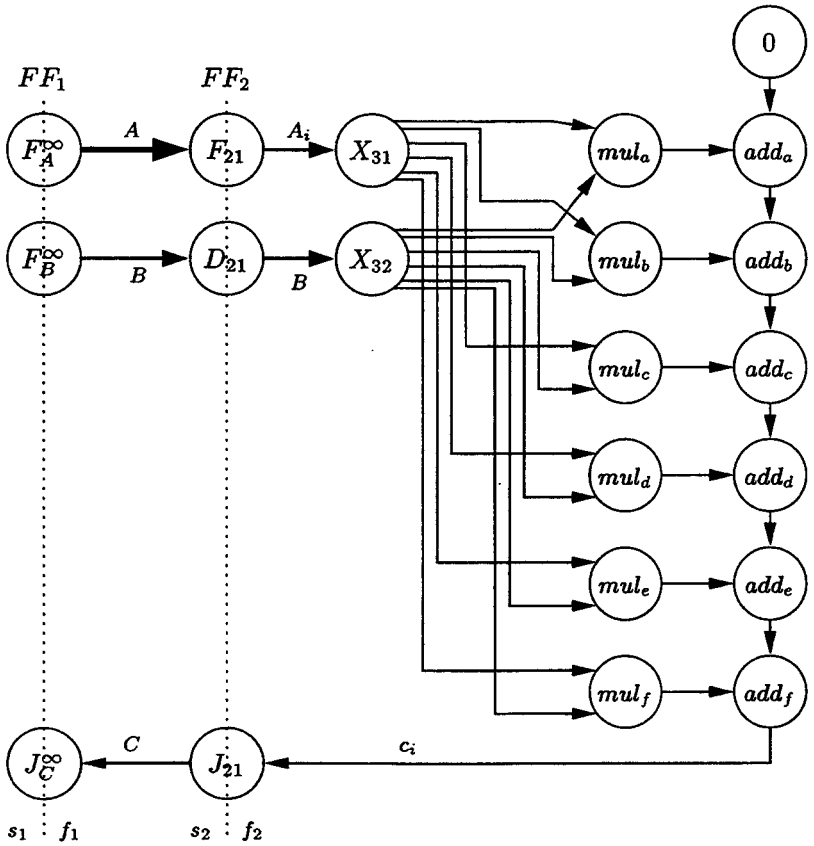


FIG. A.4: Graphe algorithmique défactorisé du PMV (∞,6/1,6/6)

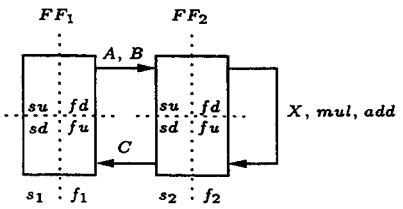


FIG. A.5: Relations de voisinage entre frontières du PMV défactorisé (∞,6/1,6/6)

- FF_1 est productrice (côté "lent") par rapport à FF_2,
- FF_1 est consommatrice (côté "lent") par rapport à FF_2,
- FF_2 est consommatrice (côté "rapide") par rapport à FF_1,
- FF_2 est productrice (côté "rapide") par rapport à FF_1,

2. la frontière FF_1 est une frontière "infinie", étant à la fois, la frontière la plus en amont et la plus en aval du graphe algorithmique. Pour cette frontière "infinie", il faut générer les signaux d'entrée rfu_1, afd_1, rsu_1 et asd_1 :
   - productrice côté "rapide" : FF_2

$$?\,rfu_1 =!\,rsd_2 \tag{A.23}$$

- consommatrice côté "rapide" : FF_2

$$?\,afd_1 =!\,asu_2 \tag{A.24}$$

- producteur côté "lent" : l'environnement

$$?\,rsu_1 = 1 \tag{A.25}$$

- consommateur côté "lent" : l'environnement

$$?\,asd_1 = 1 \tag{A.26}$$

3. la frontière FF_2 est une frontière "finie". Pour cette frontière, il faut générer les signaux d'entrée rfu_2, afd_2, rsu_2 et asd_2 :
   - productrice côté "rapide" : FF_2

$$?\,rfu_2 =!\,rfd_2 \tag{A.27}$$

   - consommatrice côté "rapide" : FF_2

$$?\,afd_2 =!\,afu_2 \tag{A.28}$$

   - productrice côté "lent" : FF_1

$$?\,rsu_2 =!\,rfd_1 \tag{A.29}$$

   - consommatrice côté "lent" : FF_1

$$?\,asd_2 =!\,afu_1 \tag{A.30}$$

La figure A.6 représente l'implantation matérielle du PMV factorisé correspondant à la spécification algorithmique représentée par la figure A.4. Le chemin de données est constitué des opérateurs frontière de factorisation (F, D, J et I) et des opérateurs combinatoires (X et M) : les chemins de données à droite de FF_2 se sont composés des opérateurs combinatoires X, *mul* et *add*, respectivement. Les autres chemins de données se sont composés trivialement des interconnexions entre les opérateurs frontière de factorisation. Le chemin de contrôle est constitué par les unités de contrôle UC_1 et UC_2, et par leurs signaux de contrôle r, a, *cpt*, *en*.

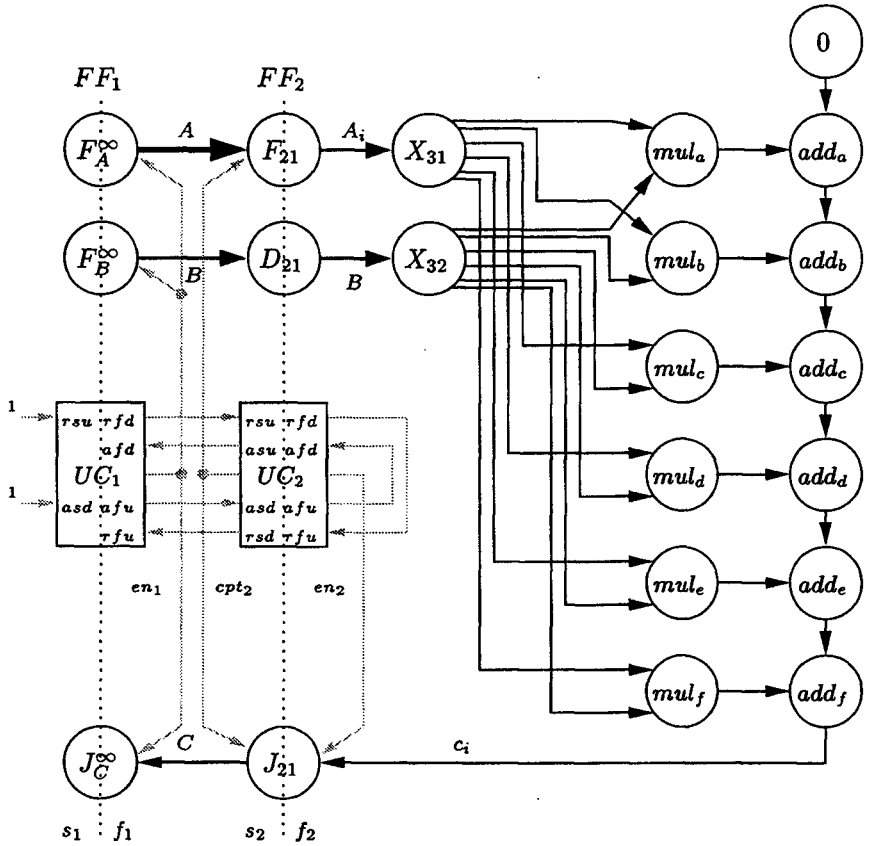


FIG. A.6: Graphe matériel défactorisé du PMV (∞,6/1,6/6)

Défactorisation partielle de FF_2

La défactorisation partielle de la frontière FF_2 du graphe algorithmique du PMV factorisé (figure 2.26) par un facteur de défactorisation $k = 2$, peut être représenté par l'éq. A.31. Le graphe algorithmique correspondant à cette défactorisation partielle est montré par la figure A.7.

$$C = \begin{bmatrix} \left[\sum_{j=1}^{6} a_{ij} \cdot b_j \right]_{i=1}^{3} \\ \left[\sum_{j=1}^{6} a_{ij} \cdot b_j \right]_{i=4}^{6} \end{bmatrix} \tag{A.31}$$

À partir du graphe algorithmique (figure A.7), nous construisons le graphe de relations entre frontières de factorisation (figure A.8). L'interconnexion des signaux de requête et d'acquittement est effectuée par l'intermédiaire de l'analyse des

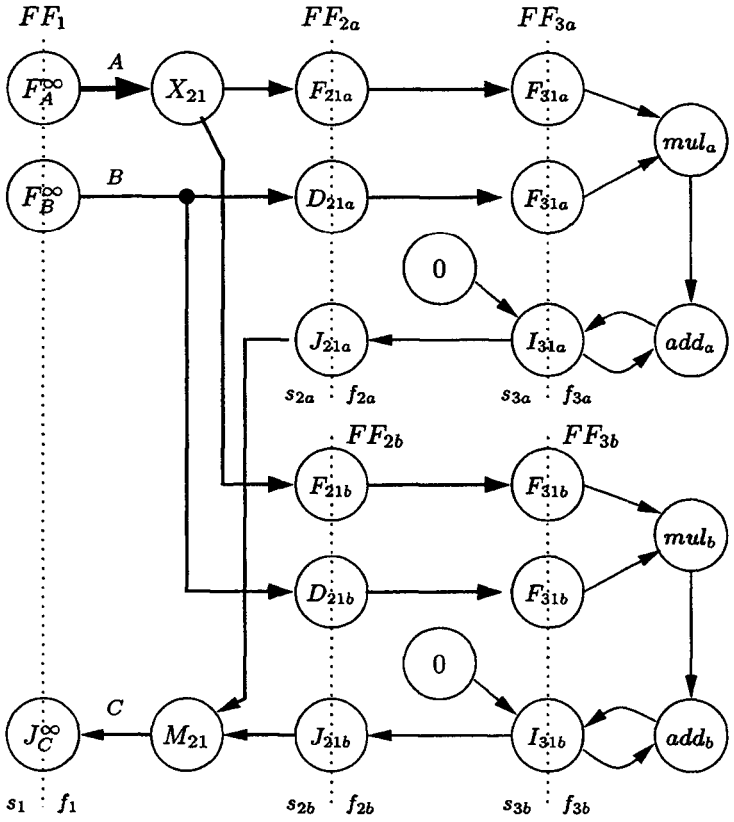


FIG. A.7: Graphe algorithmique défactorisé du PMV (∞,6/2,6/1)

relations de voisinage entre les frontières de factorisation FF_1, FF_{2a}, FF_{2b}, FF_{3a} et FF_{3b} :

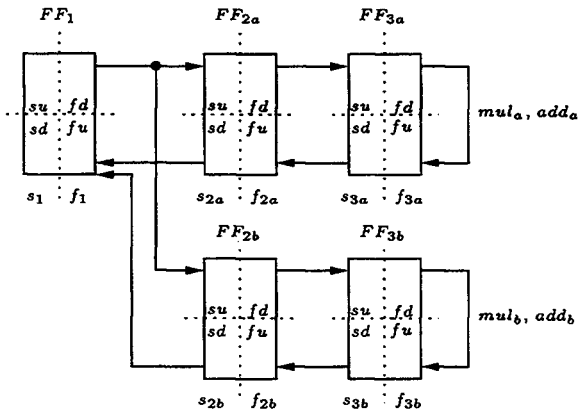


FIG. A.8: Relations de voisinage entre frontières du PMV défactorisé (∞,6/2,6/1)

1. le point de départ est le graphe de relations de voisinage (figure A.8) ;
   - FF_1 est productrice (côté "lent") par rapport à FF_{2a} et FF_{2b},
   - FF_1 est consommatrice (côté "lent") par rapport à FF_{2a} et FF_{2b},

- FF_{2a} et FF_{2b} sont consommatrices (côté "rapide") par rapport à FF_1,
- FF_{2a} et FF_{2b} sont productrices (côté "rapide") par rapport à FF_1,
- FF_{2a} est productrice (côté "lent") par rapport à FF_{3a},
- FF_{2a} est consommatrice (côté "lent") par rapport à FF_{3a},
- FF_{3a} est productrice (côté "rapide") par rapport à FF_{2a},
- FF_{3a} est consommatrice (côté "rapide") par rapport à FF_{2a}.
- FF_{2b} est productrice (côté "lent") par rapport à FF_{3b},
- FF_{2b} est consommatrice (côté "lent") par rapport à FF_{3b},
- FF_{3b} est productrice (côté "rapide") par rapport à FF_{2b},
- FF_{3b} est consommatrice (côté "rapide") par rapport à FF_{2b}.

2. la frontière FF_1 est une frontière "infinie", étant à la fois, la frontière la plus en amont et la plus en aval du graphe algorithmique. Pour cette frontière "infinie", il faut générer les signaux d'entrée rfu_1, afd_1, rsu_1 et asd_1 :
   - productrices côté "rapide" : FF_{2a} et FF_{2b}

$$?\,rfu_1 = !\,rsd_{2a}\ \&\ !\,rsd_{2b} \tag{A.32}$$

   - consommatrices côté "rapide" : FF_{2a} et FF_{2b}

$$?\,afd_1 = !\,asu_{2a}\ \&\ !\,asu_{2b} \tag{A.33}$$

   - producteur côté "lent" : l'environnement

$$?\,rsu_1 = 1 \tag{A.34}$$

   - consommateur côté "lent" : l'environnement

$$?\,asd_1 = 1 \tag{A.35}$$

3. la frontière FF_{2a} est une frontière "finie". Pour cette frontière, il faut générer les signaux d'entrée rfu_{2a}, afd_{2a}, rsu_{2a} et asd_{2a} :
   - productrice côté "rapide" : FF_{3a}

$$?\,rfu_{2a} = !\,rsd_{3a} \tag{A.36}$$

   - consommatrice côté "rapide" : FF_{3a}

$$?\,afd_{2a} = !\,asu_{3a} \tag{A.37}$$

- productrice côté "lent" : FF_1

$$?\,rsu_{2a} =!\,rfd_1 \tag{A.38}$$

- consommatrice côté "lent" : FF_1

$$?\,asd_{2a} =!\,afu_1 \tag{A.39}$$

4. la frontière FF_{3a} est une frontière "finie". Il faut générer les signaux d'entrée rfu_{3a}, afd_{3a}, rsu_{3a} et asd_{3a} :
   - productrice côté "rapide" : FF_{3a}

$$?\,rfu_{3a} =!\,rfd_{3a} \tag{A.40}$$

   - consommatrice côté "rapide" : FF_{3a}

$$?\,afd_{3a} =!\,afu_{3a} \tag{A.41}$$

   - productrice côté "lent" : FF_{2a}

$$?\,rsu_{3a} =!\,rfd_{2a} \tag{A.42}$$

   - consommatrice côté "lent" : FF_{2a}

$$?\,asd_{3a} =!\,afu_{2a} \tag{A.43}$$

5. la frontière FF_{2b} est une frontière "finie". Pour cette frontière, il faut générer les signaux d'entrée rfu_{2b}, afd_{2b}, rsu_{2b} et asd_{2b} :
   - productrice côté "rapide" : FF_{3b}

$$?\,rfu_{2b} =!\,rsd_{3b} \tag{A.44}$$

   - consommatrice côté "rapide" : FF_{3b}

$$?\,afd_{2b} =!\,asu_{3b} \tag{A.45}$$

   - productrice côté "lent" : FF_1

$$?\,rsu_{2b} =!\,rfd_1 \tag{A.46}$$

– consommatrice côté "lent" : FF_1

$$?\,asd_{2b} =!\,afu_1 \tag{A.47}$$

6. la frontière FF_{3b} est une frontière "finie". Il faut générer les signaux d'entrée rfu_{3b}, afd_{3b}, rsu_{3b} et asd_{3b} :

– productrice côté "rapide" : FF_{3b}

$$?\,rfu_{3b} =!\,rfd_{3b} \tag{A.48}$$

– consommatrice côté "rapide" : FF_{3b}

$$?\,afd_{3b} =!\,afu_{3b} \tag{A.49}$$

– productrice côté "lent" : FF_{2b}

$$?\,rsu_{3b} =!\,rfd_{2b} \tag{A.50}$$

– consommatrice côté "lent" : FF_{2b}

$$?\,asd_{3b} =!\,afu_{2b} \tag{A.51}$$

La figure A.9 représente l'implantation matérielle du PMV factorisé correspondant à la spécification algorithmique représentée par la figure A.7. Le chemin de données est constitué des opérateurs frontière de factorisation (F, D, J et I) et des opérateurs combinatoires (X et M) : les chemins de données à droite de FF_{3a} et FF_{3b} se sont composés des opérateurs combinatoires mul_a / add_a et mul_b / add_b, respectivement. Les autres chemins de données se sont composés trivialement des interconnexions entre les opérateurs frontière de factorisation. Le chemin de contrôle est constitué par les unités de contrôle UC_1, UC_{2a}, UC_{2b}, UC_{3a} et UC_{3b}, et par leurs signaux de contrôle r, a, cpt, en.

La frontière FF_2 de la figure 2.26 peut encore être défactorisée par d'autres facteurs de défactorisation ($k = 3$, $k = 4$, $k = 5$ ou $k = 6$), selon les équations suivantes :

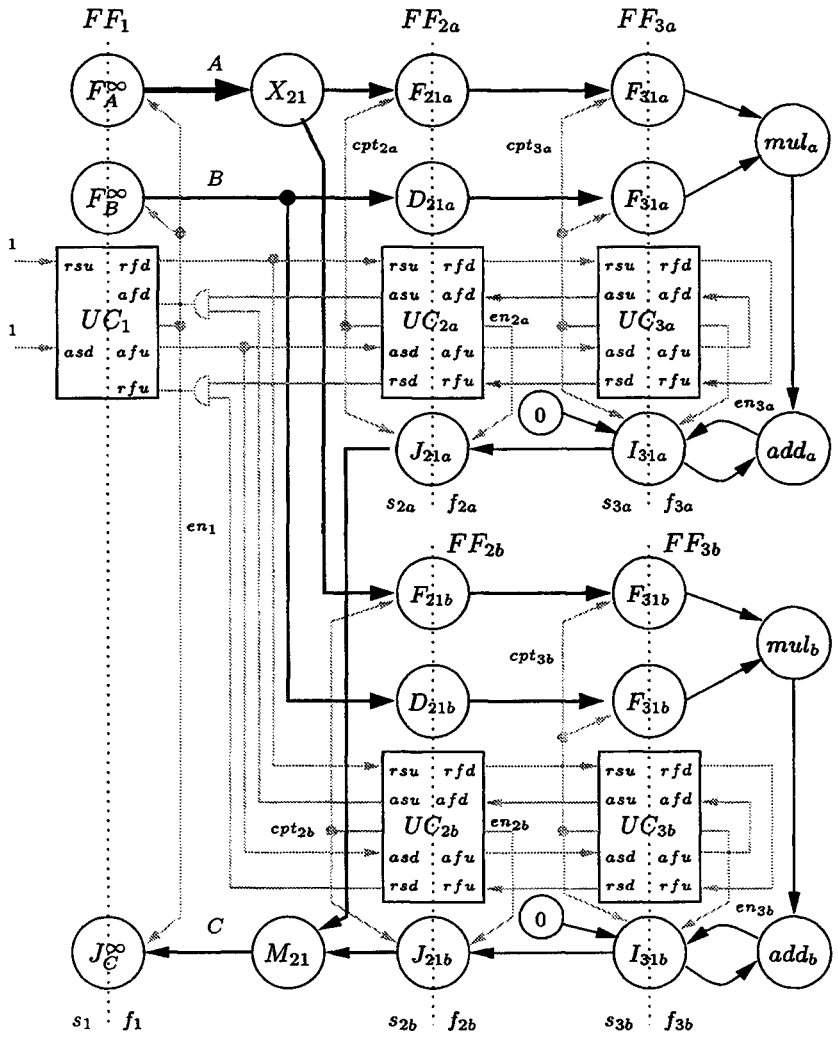


FIG. A.9: Graphe matériel défactorisé du PMV (∞,6/2,6/1)

- $k = 3$

$$C = \begin{bmatrix} \left[\sum_{j=1}^{6} a_{ij} \cdot b_j\right]_{i=1}^{2} \\ \left[\sum_{j=1}^{6} a_{ij} \cdot b_j\right]_{i=3}^{4} \\ \left[\sum_{j=1}^{6} a_{ij} \cdot b_j\right]_{i=5}^{6} \end{bmatrix} \tag{A.52}$$

- $k = 4$

$$C = \begin{bmatrix} \left[\sum_{j=1}^{6} a_{ij} \cdot b_j\right]_{i=1}^{2} \\ \left[\sum_{j=1}^{6} a_{ij} \cdot b_j\right]_{i=3}^{4} \\ \sum_{j=1}^{6} a_{5j} \cdot b_j \\ \sum_{j=1}^{6} a_{6j} \cdot b_j \end{bmatrix} \tag{A.53}$$

- $k = 5$

$$C = \begin{bmatrix} \left[\sum_{j=1}^{6} a_{ij} \cdot b_j\right]_{i=1}^{2} \\ \sum_{j=1}^{6} a_{3j} \cdot b_j \\ \sum_{j=1}^{6} a_{4j} \cdot b_j \\ \sum_{j=1}^{6} a_{5j} \cdot b_j \\ \sum_{j=1}^{6} a_{6j} \cdot b_j \end{bmatrix} \tag{A.54}$$

- $k = 6$

$$C = \begin{bmatrix} \sum_{j=1}^{6} a_{1j} \cdot b_j \\ \sum_{j=1}^{6} a_{2j} \cdot b_j \\ \sum_{j=1}^{6} a_{3j} \cdot b_j \\ \sum_{j=1}^{6} a_{4j} \cdot b_j \\ \sum_{j=1}^{6} a_{5j} \cdot b_j \\ \sum_{j=1}^{6} a_{6j} \cdot b_j \end{bmatrix} \tag{A.55}$$

La défactorisation de la frontière FF_2 par un facteur $k = 6$ implique sa défactorisation totale, où tous les opérateurs frontières de factorisation qui la délimitaient ont été remplacés par des opérateurs combinatoires, comme le montre la figure A.10.

À partir du graphe algorithmique (figure A.10), nous construisons le graphe de relations entre frontières de factorisation (figure A.11). L'interconnexion des signaux de requête et d'acquittement est effectuée par l'intermédiaire de l'analyse des relations de voisinage entre les frontières de factorisation FF_1, FF_{3a}, FF_{3b}, FF_{3c}, FF_{3d}, FF_{3e} et FF_{3f}.

L'interconnexion des signaux de requête et d'acquittement est effectuée par l'intermédiaire de l'analyse des relations de voisinage entre les frontières de factorisation FF_1, FF_{3a}, FF_{3b}, FF_{3c}, FF_{3d}, FF_{3e} et FF_{3f} :

1. le point de départ est le graphe de relations de voisinage (figure 2.28) ;
   - FF_1 est productrice (côté "lent") par rapport à FF_{3a}, FF_{3b}, FF_{3c}, FF_{3d}, FF_{3e} et FF_{3f},
   - FF_1 est consommatrice (côté "lent") par rapport à FF_{3a}, FF_{3b}, FF_{3c}, FF_{3d}, FF_{3e} et FF_{3f},
   - FF_{3a}, FF_{3b}, FF_{3c}, FF_{3d}, FF_{3e} et FF_{3f} sont consommatrices (côté "rapide") par rapport à FF_1,
   - FF_{3a}, FF_{3b}, FF_{3c}, FF_{3d}, FF_{3e} et FF_{3f} sont productrices (côté "rapide") par rapport à FF_1.
2. la frontière FF_1 est une frontière "infinie", étant à la fois, la frontière la plus en amont et la plus en aval du graphe algorithmique. Pour cette frontière "infinie", il faut générer les signaux d'entrée rfu_1, afd_1, rsu_1 et asd_1 :
   - productrices côté "rapide" : FF_{3a}, FF_{3b}, FF_{3c}, FF_{3d}, FF_{3e} et FF_{3f}

$$?\,rfu_1 = !\,rsd_{3a}\ \&\ !\,rsd_{3b}\ \&\ !\,rsd_{3c}\ \&\ !\,rsd_{3d}\ \&\ !\,rsd_{3e}\ \&\ !\,rsd_{3f} \quad \text{(A.56)}$$

   - consommatrices côté "rapide" : FF_{3a}, FF_{3b}, FF_{3c}, FF_{3d}, FF_{3e} et FF_{3f}

$$?\,afd_1 = !\,asu_{3a}\ \&\ !\,asu_{3b}\ \&\ !\,asu_{3c}\ \&\ !\,asu_{3d}\ \&\ !\,asu_{3e}\ \&\ !\,asu_{3f} \quad \text{(A.57)}$$

   - producteur côté "lent" : l'environnement

$$?\,rsu_1 = 1 \quad \text{(A.58)}$$

   - consommateur côté "lent" : l'environnement

$$?\,asd_1 = 1 \quad \text{(A.59)}$$

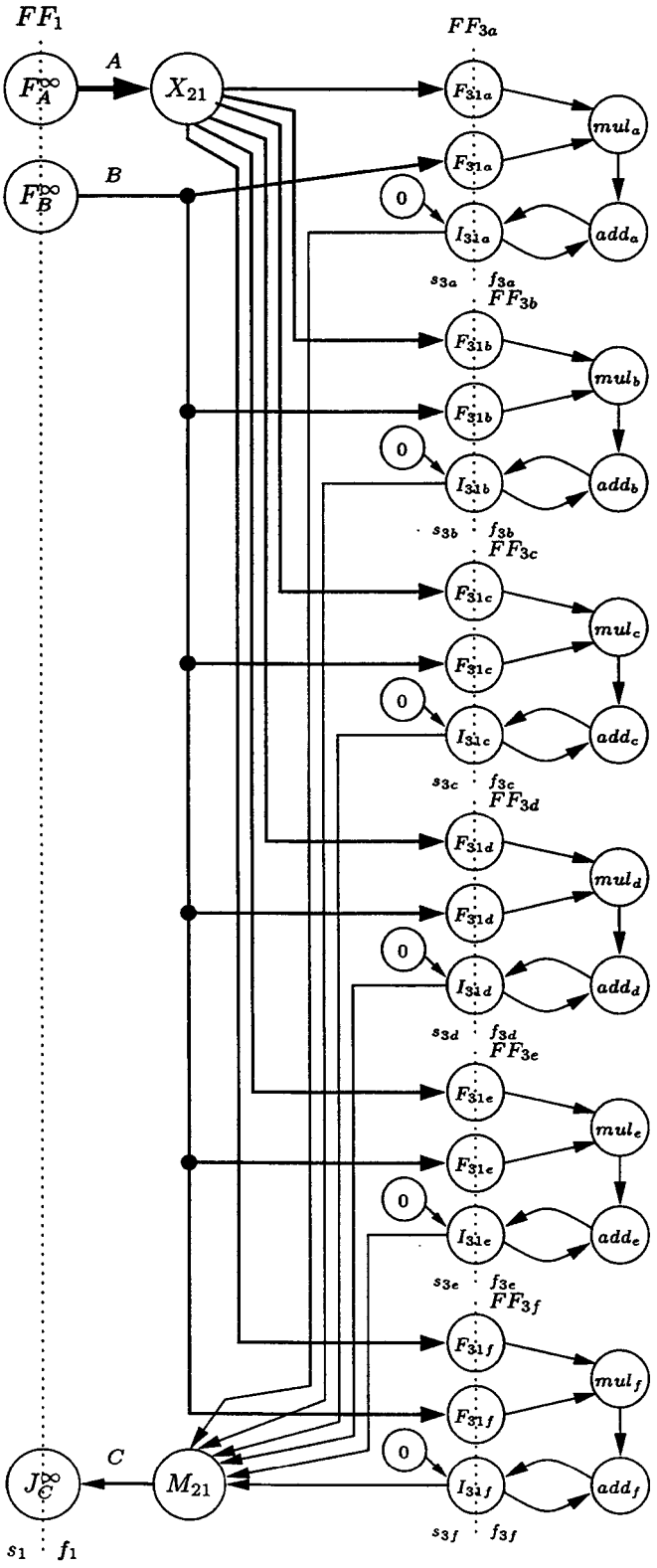


FIG. A.10: Graphe algorithmique défactorisé du PMV (∞,6/6,6/1)

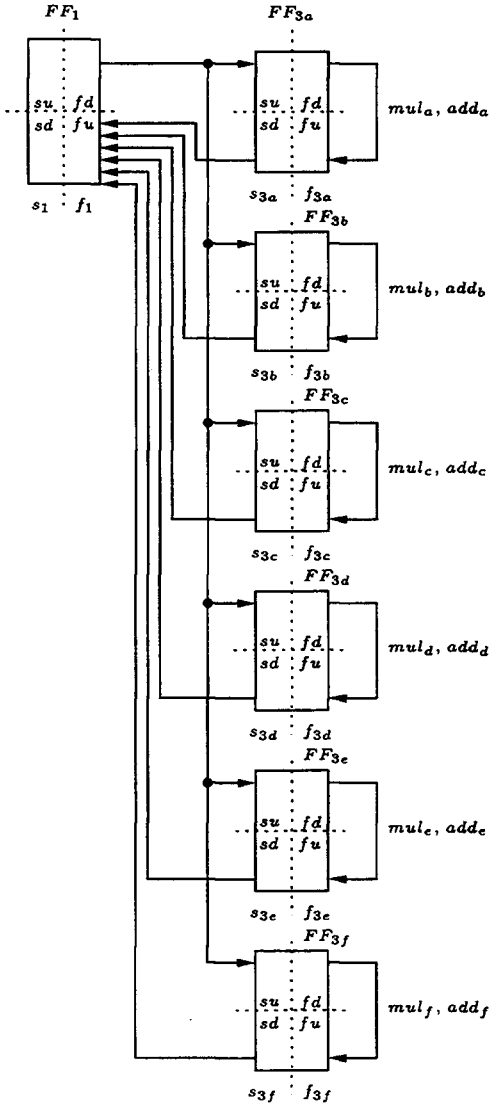


FIG. A.11: Relations de voisinage entre frontières du PMV défactorisé (∞,6/6,6/1)

3. la frontière FF_{3a} est une frontière "finie". Pour cette frontière, il faut générer les signaux d'entrée rfu_{3a}, afd_{3a}, rsu_{3a} et asd_{3a} :

   - productrice côté "rapide" : FF_{3a}

$$?\,rfu_{3a} =!\,rsd_{3a} \tag{A.60}$$

   - consommatrice côté "rapide" : FF_{3a}

$$?\,afd_{3a} =!\,asu_{3a} \tag{A.61}$$

   - productrice côté "lent" : FF_1

$$?\,rsu_{3a} =!\,rfd_1 \tag{A.62}$$

   - consommatrice côté "lent" : FF_1

$$?\,asd_{3a} =!\,afu_1 \tag{A.63}$$

4. la frontière FF_{3b} est une frontière "finie". Il faut générer les signaux d'entrée rfu_{3b}, afd_{3b}, rsu_{3b} et asd_{3b} :

   - productrice côté "rapide" : FF_{3b}

$$?\,rfu_{3b} =!\,rfd_{3b} \tag{A.64}$$

   - consommatrice côté "rapide" : FF_{3b}

$$?\,afd_{3b} =!\,afu_{3b} \tag{A.65}$$

   - productrice côté "lent" : FF_1

$$?\,rsu_{3b} =!\,rfd_1 \tag{A.66}$$

   - consommatrice côté "lent" : FF_1

$$?\,asd_{3b} =!\,afu_1 \tag{A.67}$$

5. la frontière FF_{3c} est une frontière "finie". Il faut générer les signaux d'entrée rfu_{3c}, afd_{3c}, rsu_{3c} et asd_{3c} :

- productrice côté "rapide" : FF_{3c}

$$?\,rfu_{3c} =!\,rfd_{3c} \tag{A.68}$$

- consommatrice côté "rapide" : FF_{3c}

$$?\,afd_{3c} =!\,afu_{3c} \tag{A.69}$$

- productrice côté "lent" : FF_1

$$?\,rsu_{3c} =!\,rfd_1 \tag{A.70}$$

- consommatrice côté "lent" : FF_1

$$?\,asd_{3c} =!\,afu_1 \tag{A.71}$$

6. la frontière FF_{3d} est une frontière "finie". Il faut générer les signaux d'entrée rfu_{3d}, afd_{3d}, rsu_{3d} et asd_{3d} :
   - productrice côté "rapide" : FF_{3d}

$$?\,rfu_{3d} =!\,rfd_{3d} \tag{A.72}$$

   - consommatrice côté "rapide" : FF_{3d}

$$?\,afd_{3d} =!\,afu_{3d} \tag{A.73}$$

   - productrice côté "lent" : FF_1

$$?\,rsu_{3d} =!\,rfd_1 \tag{A.74}$$

   - consommatrice côté "lent" : FF_1

$$?\,asd_{3d} =!\,afu_1 \tag{A.75}$$

7. la frontière FF_{3e} est une frontière "finie". Il faut générer les signaux d'entrée rfu_{3e}, afd_{3e}, rsu_{3e} et asd_{3e} :
   - productrice côté "rapide" : FF_{3e}

$$?\,rfu_{3e} =!\,rfd_{3e} \tag{A.76}$$

- consommatrice côté "rapide" : FF_{3e}

$$?\,afd_{3e} =!\,afu_{3e} \tag{A.77}$$

- productrice côté "lent" : FF_1

$$?\,rsu_{3e} =!\,rfd_1 \tag{A.78}$$

- consommatrice côté "lent" : FF_1

$$?\,asd_{3e} =!\,afu_1 \tag{A.79}$$

8. la frontière FF_{3f} est une frontière "finie". Il faut générer les signaux d'entrée rfu_{3f}, afd_{3f}, rsu_{3f} et asd_{3f} :

- productrice côté "rapide" : FF_{3f}

$$?\,rfu_{3f} =!\,rfd_{3f} \tag{A.80}$$

- consommatrice côté "rapide" : FF_{3f}

$$?\,afd_{3f} =!\,afu_{3f} \tag{A.81}$$

- productrice côté "lent" : FF_1

$$?\,rsu_{3f} =!\,rfd_1 \tag{A.82}$$

- consommatrice côté "lent" : FF_1

$$?\,asd_{3f} =!\,afu_1 \tag{A.83}$$

À partir des équations ci-dessus, du graphe algorithmique et du graphe des relations entre les frontières correspondant à la solution où la frontière FF_2 a été défactorisée par un facteur $k = 6$, nous pouvons construire le graphe matériel correspondant. Pourtant, nous n'allons pas le représenter ici à cause de ces dimensions importantes.

Défactorisation totale de FF_2 et FF_3

La défactorisation totale simultanée des frontières FF_2 et FF_3 du graphe algorithmique du PMV factorisé (figure 2.26) par un facteur de défactorisation $k = 6$, peut être représenté par l'éq. A.84.

$$C = \begin{bmatrix} a_{11} \cdot b_1 + a_{12} \cdot b_2 + a_{13} \cdot b_3 + a_{14} \cdot b_4 + a_{15} \cdot b_5 + a_{16} \cdot b_6 \\ a_{21} \cdot b_1 + a_{22} \cdot b_2 + a_{23} \cdot b_3 + a_{24} \cdot b_4 + a_{25} \cdot b_5 + a_{26} \cdot b_6 \\ a_{31} \cdot b_1 + a_{32} \cdot b_2 + a_{33} \cdot b_3 + a_{34} \cdot b_4 + a_{35} \cdot b_5 + a_{36} \cdot b_6 \\ a_{41} \cdot b_1 + a_{42} \cdot b_2 + a_{43} \cdot b_3 + a_{44} \cdot b_4 + a_{45} \cdot b_5 + a_{46} \cdot b_6 \\ a_{51} \cdot b_1 + a_{52} \cdot b_2 + a_{53} \cdot b_3 + a_{54} \cdot b_4 + a_{55} \cdot b_5 + a_{56} \cdot b_6 \\ a_{61} \cdot b_1 + a_{62} \cdot b_2 + a_{63} \cdot b_3 + a_{64} \cdot b_4 + a_{65} \cdot b_5 + a_{66} \cdot b_6 \end{bmatrix} \tag{A.84}$$

Évidemment, il existe d'autres défactorisations possibles qui sont les défactorisations partielles simultanées des frontières FF_2 et FF_3, comme, par exemple, celle représentée par l'éq. A.85, où les deux frontières ont été défactorisées par un facteur $k = 2$.

$$C = \begin{bmatrix} \left[\sum_{j=1}^{3} a_{ij} \cdot b_j + \sum_{j=4}^{6} a_{ij} \cdot b_j \right]_{i=1}^{3} \\ \left[\sum_{j=1}^{3} a_{ij} \cdot b_j + \sum_{j=4}^{6} a_{ij} \cdot b_j \right]_{i=4}^{6} \end{bmatrix} \tag{A.85}$$

A.2 Implantations du PMV

La spécification factorisée du PMV représentée par l'éq. A.1 peut avoir 36 implantations différentes, selon les défactorisations effectuées, comme le montre le tableau A.2. Dans ce tableau, nous résumons les résultats obtenus après simulation et synthèse de toutes ces défactorisations. Nous y présentons le nombre de cycles (*Nb. cycles*), la longueur du chemin critique (*Delay*), la latence, la fréquence de l'horloge (*Fréq.*) et la surface en termes du nombre de générateurs de fonction (*Nb. F/G*), de CLB (*Nb. CLB*) et de bascules DFF (*Nb. DFF*).

Nombre de F/G *versus* latence

Nous montrons dans la figure A.12 la relation entre le nombre de générateurs de fonction F/G et la latence, en fonction des différentes implantations. Nous constatons que, en partant de l'implantation totalement factorisée (∞, 6/1, 6/1) vers l'implantation totalement défactorisée (∞, 6/6, 6/6) le nombre de F/G augmente pendant que la latence diminue. Cela confirme notre expectative annoncée à l'Introduction de ce travail et illustrée par la figure 1 : la défactorisation permet de réduire la latence, tout en augmentant le nombre de ressources matérielles nécessaires à l'implantation. Ce comportement est observé soit quand nous défactorisons FF_2, soit quand nous défactorisons FF_3, soit quand nous défactorisons FF_2 et FF_3 simultanément.

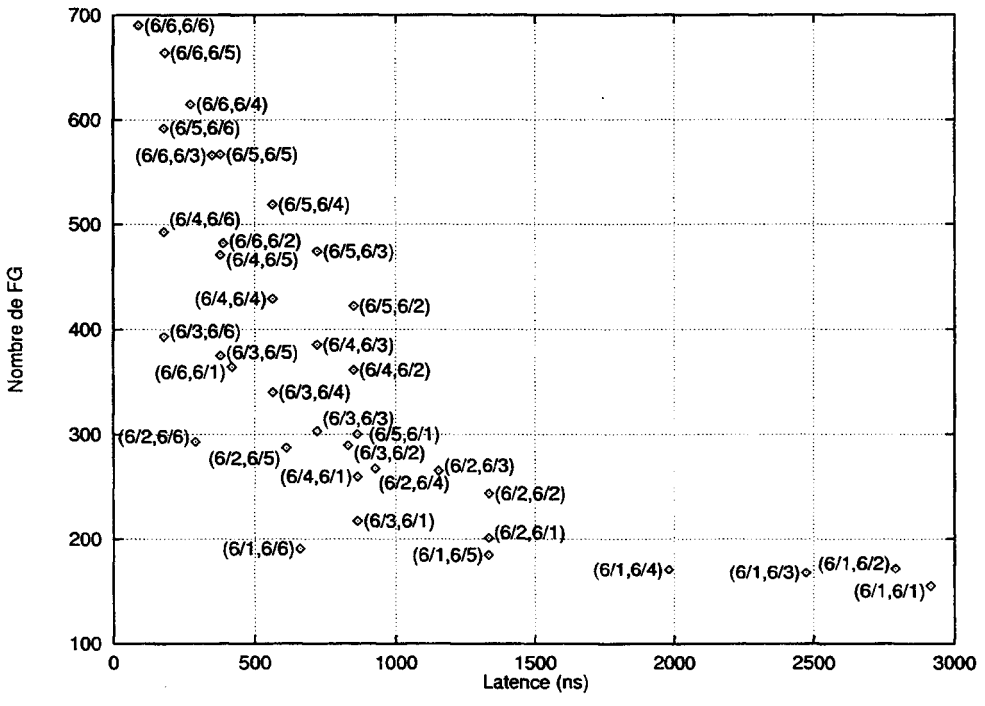


FIG. A.12: Implantations du PMV : F/G *versus* latence

TAB. A.2: Implantations matérielles du PMV

Implantation	*Nb. cycl.*	*Delay (ns)*	*Latence (ns)*	*Freq. (MHz)*	*Nb. F/G*	*Nb. CLB*	*Nb. DFF*
(∞, 6/1, 6/1)	36	81	2916	12,4	155	76	217
(∞, 6/1, 6/2)	30	93	2790	10,8	172	92	227
(∞, 6/1, 6/3)	24	103	2472	9,7	168	90	228
(∞, 6/1, 6/4)	18	110	1980	9,0	171	82	221
(∞, 6/1, 6/5)	12	111	1332	9,0	185	80	214
(∞, 6/1, 6/6)	6	110	660	9,0	191	79	207
(∞, 6/2, 6/1)	18	74	1332	13,5	201	99	224
(∞, 6/2, 6/2)	15	89	1335	11,1	243	130	244
(∞, 6/2, 6/3)	12	96	1152	10,4	265	142	246
(∞, 6/2, 6/4)	9	103	927	9,7	267	124	232
(∞, 6/2, 6/5)	6	102	612	9,8	287	122	218
(∞, 6/2, 6/6)	3	97	291	10,2	293	128	204
(∞, 6/3, 6/1)	12	72	864	13,8	217	101	222
(∞, 6/3, 6/2)	10	83	830	12,0	289	151	252
(∞, 6/3, 6/3)	8	90	720	11,1	303	164	255
(∞, 6/3, 6/4)	6	94	564	10,6	340	159	234
(∞, 6/3, 6/5)	4	95	380	10,5	375	149	213
(∞, 6/3, 6/6)	2	89	178	11,2	393	153	192
(∞, 6/4, 6/1)	12	72	864	13,8	259	126	224
(∞, 6/4, 6/2)	10	85	850	11,7	361	188	264
(∞, 6/4, 6/3)	8	90	720	11,1	385	207	268
(∞, 6/4, 6/4)	6	94	564	10,6	429	196	240
(∞, 6/4, 6/5)	4	95	380	10,5	471	178	212
(∞, 6/4, 6/6)	2	89	178	11,2	493	181	184
(∞, 6/5, 6/1)	12	72	864	13,8	300	147	226
(∞, 6/5, 6/2)	10	85	850	11,7	422	223	276
(∞, 6/5, 6/3)	8	90	720	11,1	474	246	281
(∞, 6/5, 6/4)	6	94	564	10,6	519	230	246
(∞, 6/5, 6/5)	4	95	380	10,5	567	208	211
(∞, 6/5, 6/6)	2	89	178	11,2	592	208	176
(∞, 6/6, 6/1)	6	70	420	14,3	364	168	228
(∞, 6/6, 6/2)	5	78	390	12,7	482	263	288
(∞, 6/6, 6/3)	4	88	352	11,4	566	301	294
(∞, 6/6, 6/4)	3	91	273	11,0	615	273	252
(∞, 6/6, 6/5)	2	91	182	10,9	664	240	210
(∞, 6/6, 6/6)	1	87	87	11,4	690	234	168

Nombre de CLB *versus* latence

Nous montrons dans la figure A.13 la relation entre le nombre de CLB et la latence, en fonction des différentes implantations. Nous constatons que, en partant de l'implantation totalement factorisée (∞, 6/1, 6/1) vers l'implantation totalement défactorisée (∞, 6/6, 6/6) il y a une tendance d'augmentation du nombre de CLB pendant que la latence diminue. Ce comportement n'est confirmé que quand nous défactorisons la frontière FF_2. La défactorisation totale de FF_3, par exemple, correspond à un nombre de CLB inférieur à celui de sa défactorisation par un facteur $k = 3$, même si la latence a été réduite de façon importante.

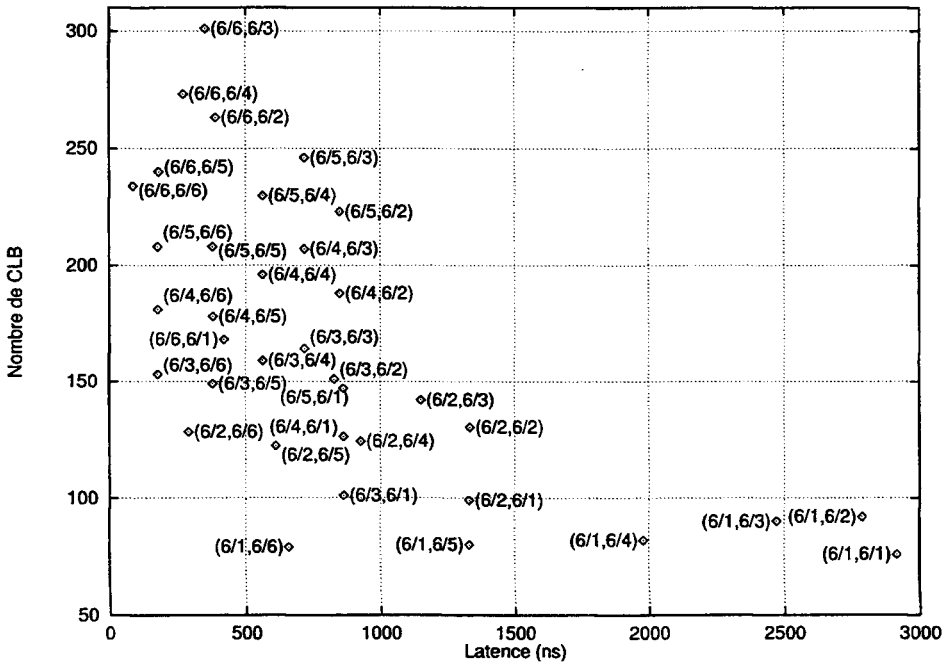


FIG. A.13: Implantations du PMV : CLB *versus* latence

Nombre de DFF *versus* latence

Nous montrons dans la figure A.14 la relation entre le nombre de registres DFF et la latence, en fonction des différentes implantations. Nous constatons que, en partant de l'implantation totalement factorisée (∞, 6/1, 6/1) vers l'implantation totalement défactorisée (∞, 6/6, 6/6), il n'y a pas de tendance d'augmentation du nombre de DFF pendant que la latence diminue. Le nombre de DFF est une fonction des opérateurs de factorisation qui contiennent des registres. Ainsi, l'implantation totalement défactorisé exigera évidemment moins de DFF que l'implantation totalement factorisée.

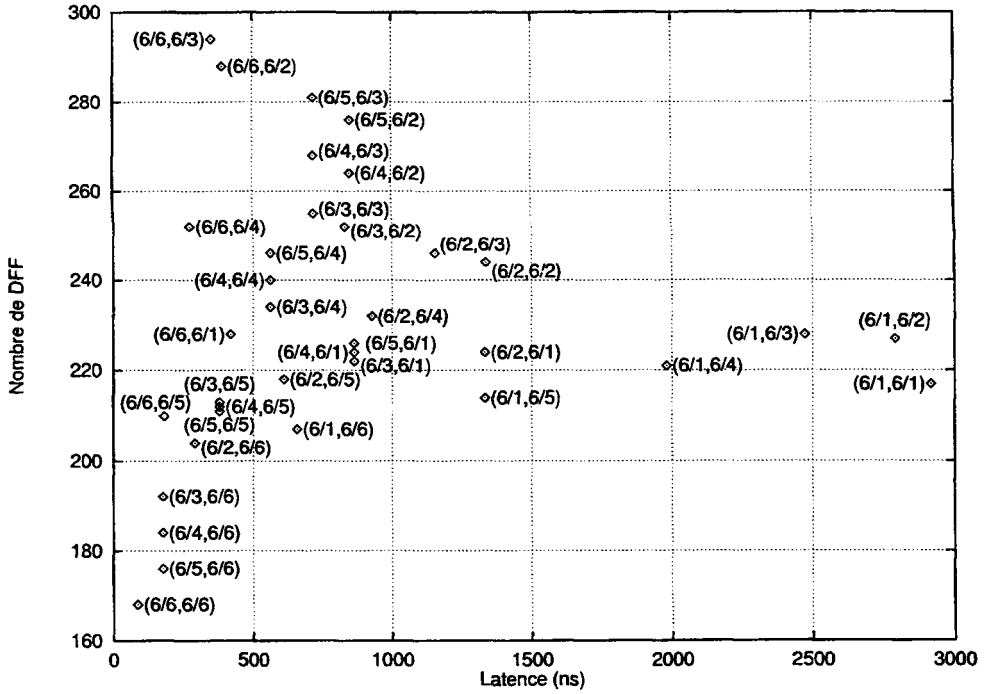


FIG. A.14: Implantations du PMV : DFF *versus* latence

Bilan

Toutes les mesures de surface (nombre de F/G, de CLB ou de DFF) apportent des informations très importantes concernant l'implantation. Pourtant, seuls le nombre de F/G et de CLB doivent être utilisés pour guider l'heuristique de défactorisation, puisque l'un (le nombre de F/G) s'approche le plus du comportement théorique et l'autre (le nombre de CLB) est la mesure la plus répandue parmi les concepteurs de circuits à base de FPGA.

A.3 Estimation de surface et de performances temporelles

Le tableau A.3 montre les résultats obtenus par l'application de la méthode de estimation de surface décrite dans le chapitre 4, à toutes les implantations possibles du PMV. Nous constatons que l'erreur moyenne de notre méthode d'estimation est de 16 % pour le nombre de générateurs de fonction F/G, de 10 % pour le nombre de CLB et de 4 % pour le nombre de registres DFF. Ces valeurs d'erreur moyenne, inférieures à 20 % nous autorisent à utiliser notre estimateur pour guider l'heuristique de défactorisation.

Le tableau A.4 montre les résultats obtenus par l'application de la méthode de estimation temporelle décrite dans le chapitre 4, à toutes les implantations possibles du PMV. L'erreur moyenne de cette estimation est de 9 %, une valeur bien inférieure aux 20 % préconisé comme acceptable par des outils de CAO tels que *Leonardo*.

TAB. A.3: Estimations de surface du PMV

IMPLANTATION	SURFACE								
	Nb. F/G			Nb. CLB			Nb. DFF		
	Cal	*Sim*	Δ%	*Cal*	*Sim*	Δ%	*Cal*	*Sim*	Δ%
(∞, 6/1, 6/1)	140	155	-10	72	76	-5	215	217	-1
(∞, 6/1, 6/2)	170	172	-1	82	92	-11	227	227	0
(∞, 6/1, 6/3)	202	168	+20	94	90	+4	233	228	+2
(∞, 6/1, 6/4)	202	171	+18	91	82	+11	225	221	+2
(∞, 6/1, 6/5)	200	185	+8	88	80	+10	217	214	+1
(∞, 6/1, 6/6)	201	191	+5	85	79	+8	209	207	+1
(∞, 6/2, 6/1)	202	201	0	101	99	+2	226	224	+1
(∞, 6/2, 6/2)	261	243	+7	121	130	-7	238	244	-2
(∞, 6/2, 6/3)	327	265	+23	145	142	+2	250	246	+2
(∞, 6/2, 6/4)	325	267	+22	139	124	+12	234	232	+1
(∞, 6/2, 6/5)	323	287	+13	133	122	+9	218	218	0
(∞, 6/2, 6/6)	325	293	+11	127	128	-1	202	204	-1
(∞, 6/3, 6/1)	247	217	+14	132	101	+31	231	222	+4
(∞, 6/3, 6/2)	351	289	+21	162	151	+7	249	252	-1
(∞, 6/3, 6/3)	450	303	+49	199	164	+21	267	255	+5
(∞, 6/3, 6/4)	447	340	+31	189	159	+19	273	234	+17
(∞, 6/3, 6/5)	444	375	+18	180	149	+21	199	213	-7
(∞, 6/3, 6/6)	447	393	+14	171	153	+12	195	192	+2
(∞, 6/4, 6/1)	295	259	+14	144	126	+14	230	224	+3
(∞, 6/4, 6/2)	409	361	+13	186	188	-1	258	264	-2
(∞, 6/4, 6/3)	543	385	+41	236	207	+14	282	268	+5
(∞, 6/4, 6/4)	539	429	+26	222	196	+13	250	240	+4
(∞, 6/4, 6/5)	535	471	+14	210	178	+18	238	212	+12
(∞, 6/4, 6/6)	539	493	+9	198	181	+9	186	184	+1
(∞, 6/5, 6/1)	327	300	+9	160	147	+9	237	226	+5
(∞, 6/5, 6/2)	472	422	+12	210	223	-6	267	276	-3
(∞, 6/5, 6/3)	621	474	+31	270	246	+10	297	281	+6
(∞, 6/5, 6/4)	633	519	+22	257	230	+12	257	246	+4
(∞, 6/5, 6/5)	621	567	+10	240	208	+15	217	211	+3
(∞, 6/5, 6/6)	629	592	+6	225	208	+8	177	176	+1
(∞, 6/6, 6/1)	359	364	-1	174	168	+4	240	228	+5
(∞, 6/6, 6/2)	533	482	+11	234	263	-11	276	228	-4
(∞, 6/6, 6/3)	731	566	+29	311	301	+3	312	294	+6
(∞, 6/6, 6/4)	725	615	+18	290	273	+6	264	252	+5
(∞, 6/6, 6/5)	719	664	+8	270	240	+13	216	210	+3
(∞, 6/6, 6/6)	721	690	+4	252	234	+8	168	168	0

TAB. A.4: Estimations temporelles du PMV

IMPLANTATION	PERFORMANCES TEMPORELLES								
	Nb. Cycles			T_H (*ns*)			Latence (*ns*)		
	Cal	*Sim*	Δ%	*Cal*	*Sim*	Δ%	*Cal*	*Sim*	Δ%
(∞, 6/1, 6/1)	36	36	0	71	81	-12	2556	2916	-12
(∞, 6/1, 6/2)	30	30	0	87	93	-6	2610	2790	-6
(∞, 6/1, 6/3)	24	24	0	106	103	+3	2544	2472	+3
(∞, 6/1, 6/4)	18	18	0	102	110	-7	1836	1980	-7
(∞, 6/1, 6/5)	12	12	0	96	111	-14	1152	1332	-14
(∞, 6/1, 6/6)	6	6	0	93	110	-15	558	660	-15
(∞, 6/2, 6/1)	18	18	0	69	74	-7	1242	1332	-7
(∞, 6/2, 6/2)	15	15	0	86	89	-3	1290	1335	-3
(∞, 6/2, 6/3)	12	12	0	105	96	+9	1260	1152	+9
(∞, 6/2, 6/4)	9	9	0	100	103	-3	900	927	-3
(∞, 6/2, 6/5)	6	6	0	95	102	-7	570	612	-7
(∞, 6/2, 6/6)	3	3	0	92	97	-5	276	291	-5
(∞, 6/3, 6/1)	12	12	0	69	72	-4	828	864	-4
(∞, 6/3, 6/2)	10	10	0	86	83	+4	860	830	+4
(∞, 6/3, 6/3)	8	8	0	105	90	+17	840	720	+17
(∞, 6/3, 6/4)	6	6	0	101	94	+7	606	564	+7
(∞, 6/3, 6/5)	4	4	0	95	95	0	380	380	0
(∞, 6/3, 6/6)	2	2	0	92	89	+3	184	178	+3
(∞, 6/4, 6/1)	12	12	0	69	72	-4	828	864	-4
(∞, 6/4, 6/2)	10	10	0	86	85	+1	860	850	+1
(∞, 6/4, 6/3)	8	8	0	105	90	+17	840	720	+17
(∞, 6/4, 6/4)	6	6	0	100	94	+6	600	564	+6
(∞, 6/4, 6/5)	4	4	0	95	95	0	380	380	0
(∞, 6/4, 6/6)	2	2	0	92	89	+3	184	178	+3
(∞, 6/5, 6/1)	12	12	0	69	72	-4	828	864	-4
(∞, 6/5, 6/2)	10	10	0	86	85	+1	860	850	+1
(∞, 6/5, 6/3)	8	8	0	103	90	+14	824	720	+14
(∞, 6/5, 6/4)	6	6	0	100	94	+6	600	564	+6
(∞, 6/5, 6/5)	4	4	0	95	95	0	380	380	0
(∞, 6/5, 6/6)	2	2	0	92	89	+3	184	178	+3
(∞, 6/6, 6/1)	6	6	0	47	70	-33	282	420	-33
(∞, 6/6, 6/2)	5	5	0	63	78	-19	315	390	-19
(∞, 6/6, 6/3)	4	4	0	82	88	-7	328	352	-7
(∞, 6/6, 6/4)	3	3	0	77	91	-15	231	273	-15
(∞, 6/6, 6/5)	2	2	0	72	91	-21	144	182	-21
(∞, 6/6, 6/6)	1	1	0	69	87	-21	69	87	-21

À partir des résultats présentés dans les tableaux A.3 et A.4, nous avons tracé les courbes montrées par les figures A.15, A.16, A.17 et A.18, qui représentent respectivement les modules des erreurs vérifiés entre notre méthode de estimation de surface et de performance temporelle des différentes implantations du PMV et les résultats fournis par l'outil de synthèse.

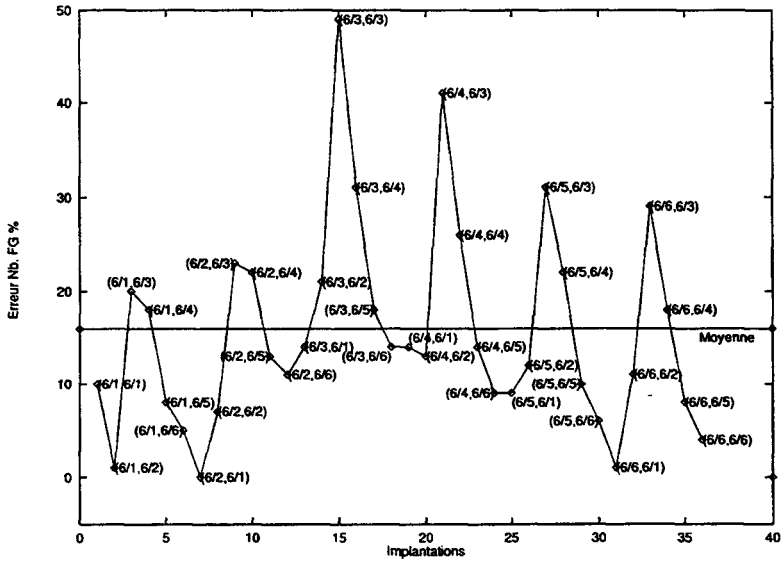


FIG. A.15: Implantations du PMV : erreur % (nombre de F/G)

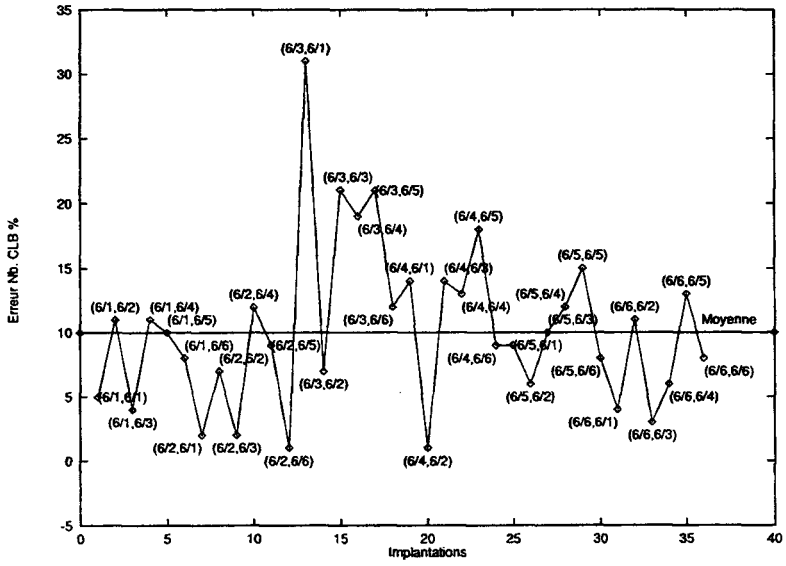


FIG. A.16: Implantations du PMV : erreur % (nombre de CLB)

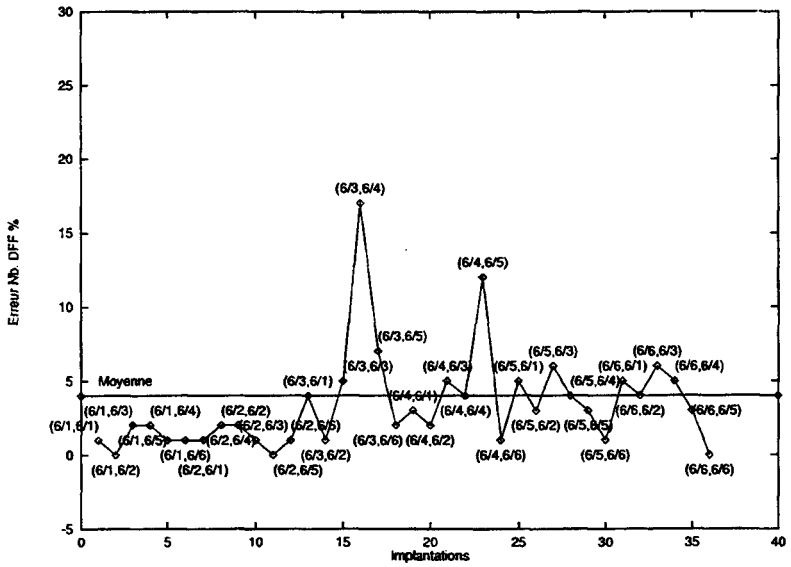


FIG. A.17: Implantations du PMV : erreur % (nombre de DFF)

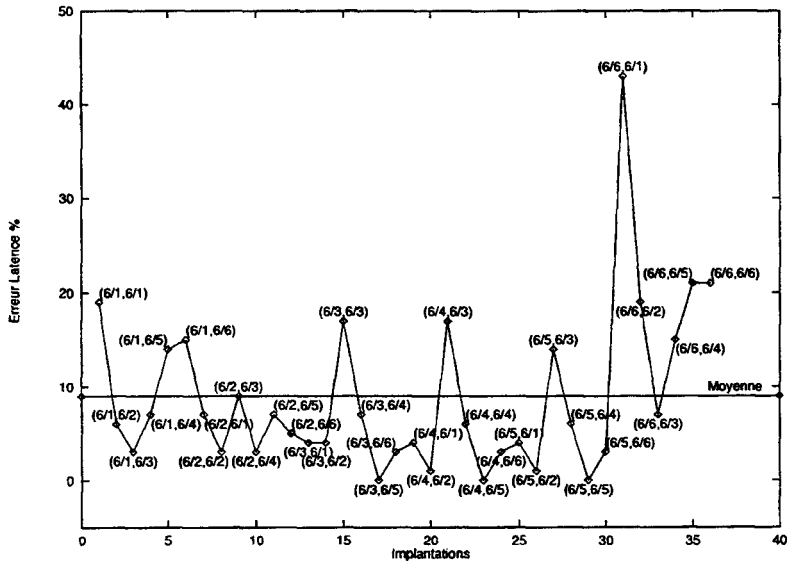


FIG. A.18: Implantations du PMV : erreur % (Latence)

A.4 Heuristique de défactorisation

Nous montrons dans le tableau A.5 quelques résultats obtenus par notre heuristique de défactorisation, en partant d'une spécification totalement factorisée. Nous avons supposé quelques valeurs de contraintes temporelles (3000, 2500, 2000, 1200 et 800 *ns*) et nous avons appliqué notre heuristique afin de déterminer la solution optimisée. La solution optimale est celle qui respecte la contrainte temporelle tout en minimisant l'augmentation des ressources matérielles.

TAB. A.5: Résultats de l'heuristique

Contrainte (ns)	*Solutions*		*Bilan*	
	Optimale	*Optimisée*	*F/G*	*Lat.*
3000	(∞, 6/1, 6/1)	(∞, 6/1, 6/1)	0%	0%
2500	(∞, 6/1, 6/3)	(∞, 6/2, 6/1)	+20%	-46%
2000	(∞, 6/1, 6/4)	(∞, 6/2, 6/1)	+18%	-33%
1200	(∞, 6/3, 6/1)	(∞, 6/3, 6/1)	0%	0%
800	(∞, 6/1, 6/6)	(∞, 6/1, 6/6)	0%	0%

Annexe B

Bibliothèque VHDL

Dans cet annexe, nous présentons la bibliothèque d'opérateurs VHDL utilisée pour l'implantation matérielle des opérateurs de base, des opérateurs de factorisation de motifs de graphes répétitifs et des unités de contrôle des frontières finies et infinies. La fonctionnalité des opérateurs de base et de factorisation a été décrite dans le chapitre 2. La structure interne des opérateurs et des unités de contrôle a été détaillée dans le chapitre 3. Nous y présentons également le code VHDL correspondant à l'implantation de la spécification totalement factorisée du PMV. Nous ne présenterons pas le code VHDL correspondant aux autres implantations possibles du PMV afin de ne pas augmenter encore plus la taille de cet annexe.

B.1 Définition des constantes, types et sous-types

Nous présentons ci-dessous la définition des différentes constantes, types et sous-types utilisés pour spécifier les opérateurs de base, de factorisation et les unités de contrôle des frontières de factorisation :

```
-- ******** DEFINITION DES CONSTANTES, TYPES, SOUS-TYPES ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

package DEFINITIONS is

   -- Definition des constantes
   constant NBITS : integer range 3 to 32 := 3; -- Numero de bits
   constant DNBITS : integer range 6 to 64 := NBITS * 2; -- Numero de bits x 2
   constant DIMENSION : integer range 2 to 300 := 6; -- Dimension (matrice carree)
   constant NLIG : integer range 2 to 300 := 6; -- No. lignes image
   constant NCOL : integer range 2 to 300 := 6; -- No. colonnes image
   constant MAXBITS : integer range 4 to 1032 := DNBITS+1; -- No. max bits
   constant FACTEUR : integer range 2 to DIMENSION := 2; -- Facteur defacrisation
   constant F2 : integer range 1 to DIMENSION := 6; -- Facteur 2
   constant F3 : integer range 1 to DIMENSION := 6; -- Facteur 3

   -- valeurs de 'setup'
   constant SetUpMul : time := 0 ns;
   constant SetUpAdd : time := 0 ns;
   constant SetUpImp : time := 0 ns;
   constant SetUpIt : time := 0 ns;
   constant SetUpCp : time := 0 ns;
   constant SetUpSt : time := 0 ns;
   constant SetUpJin : time := 0 ns;
   constant SetUpT : time := 0 ns;
   constant SetUpReg : time := 0 ns;
   constant SetUpDiff : time := 0 ns;
   constant SetUpExp : time := 0 ns;
   constant SetUpConv : time := 0 ns;

   -- Definition des types et soustypes
   subtype QSV is qsim_state_vector;- signal

   type VECT_SIG is array (natural RANGE <>) of QSV(MAXBITS-1 downto 0); - vecteur de signaux
   type MAT_SIG is array (natural RANGE <>) of VECT_SIG(0 to DIMENSION-1); - matrice de signaux
   type MMAT_SIG is array (natural RANGE <>) of MAT_SIG(0 to DIMENSION-1); - matrice de matrice
                                                                            - de signaux
end DEFINITIONS;
```

B.2 Opérateurs de base

Les opérateurs de base dont les respectives implantations VHDL nous montrons ci-dessus sont l'*IMPLODE*, l'*EXPLODE* et les opérateurs d'addition et de multiplication.

IMPLODE VECTEUR

Cet opérateur regroupe les éléments d'un vecteur de dimension *Dim* :

```
-- ******** IMPLODE VECTEUR ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity IMPLODEV is
   generic (wsize : integer := NBITS;
            SetUp : time := SetUpImp;
            Dim : integer := DIMENSION);
   port (entree : in VECT_SIG(0 to Dim-1);
         sortie : out VECT_SIG(0 to Dim-1));
end IMPLODEV;

architecture OPERATEUR of IMPLODEV is
begin
   sortie <= entree;
end OPERATEUR;
```

IMPLODE PARTIEL VECTEUR

Cet opérateur regroupe les éléments d'un vecteur de dimension *Modulo* partiellement séparé, en fonction de la valeur du facteur K :

```
-- ******** IMPLODE PARTIEL VECTEUR ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity IMPLODEPV is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpExp;
           Modulo : integer := DIMENSION;
           Modp : integer := DIMENSION/FACTEUR;
           K : integer := FACTEUR);
  port (Entree : in MAT_SIG(0 to K-1);
        sortie : out VECT_SIG(0 to Modulo-1));
end IMPLODEPV;

architecture OPERATEUR of IMPLODEPV is
begin
  IMPLO : process (Entree)
  begin
    for i in 0 to (K - 1) loop
      if i < (Modulo rem K) then
        for j in 0 to Modp loop
          sortie(i*(Modp+1)+j) <= Entree(i)(j);
        end loop; -- j
      else
        for j in 0 to Modp-1 loop
          sortie(i*Modp+j+Modulo rem K) <= Entree(i)(j);
        end loop; -- j
      end if;
    end loop; -- i
  end process IMPLO;
end OPERATEUR;
```

IMPLODE MATRICE

Cet opérateur regroupe les éléments d'une matrice de dimension *Dim* :

```
-- ******** IMPLODE MATRICE ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity IMPLODEM is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpImp;
           Dim : integer := DIMENSION);
  port (entree : in MAT_SIG(0 to Dim-1);
        sortie : out MAT_SIG(0 to Dim-1));
end IMPLODEM;

architecture OPERATEUR of IMPLODEM is
begin
  sortie <= entree;
end OPERATEUR;
```

EXPLODE VECTEUR

Cet opérateur décompose un vecteur de dimension *Dim* en ses éléments :

```
-- ******** EXPLODE VECTEUR ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity EXPLODEV is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpImp;
           Dim : integer := DIMENSION);
  port (Entree : in VECT_SIG(0 to Dim-1);
        sortie : out VECT_SIG(0 to Dim-1));
end EXPLODEV;

architecture OPERATEUR of EXPLODEV is
begin
  sortie <= Entree;
end OPERATEUR;
```

EXPLODE PARTIEL VECTEUR

Cet opérateur décompose partiellement un vecteur de dimension *Modulo*, en fonction du facteur K :

```
-- ******** EXPLODE PARTIEL VECTEUR ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity EXPLODEPV is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpExp;
           Modulo : integer := DIMENSION;
           Modp : integer := DIMENSION/FACTEUR;
           K : integer := FACTEUR);
  port (Entree : in VECT_SIG(0 to Modulo-1);
        sortie : out MAT_SIG(0 to K-1));
end EXPLODEPV;

architecture OPERATEUR of EXPLODEPV is
begin
  EXPLO : process (Entree)
  begin
    for i in 0 to (K - 1) loop
      if i < (Modulo rem K) then
        for j in 0 to Modp loop
          sortie(i)(j) <= Entree(i*(Modp+1)+j);
        end loop; -- j
      else
        for j in 0 to Modp-1 loop
          sortie(i)(j) <= Entree(i*Modp+j+Modulo rem K);
        end loop; -- j
      end if;
    end loop; -- i
  end process EXPLO;
end OPERATEUR;
```

EXPLODE MATRICE

Cet opérateur décompose une matrice de dimension *Dim* en ses éléments :

```
-- ******** EXPLODE MATRICE ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity EXPLODEM is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpImp;
           Dim : integer := DIMENSION);
  port (Entree : in MAT_SIG(0 to Dim-1);
        sortie : out MAT_SIG(0 to Dim-1));
end EXPLODEM;

architecture OPERATEUR of EXPLODEM is
begin
  sortie <= Entree;
end OPERATEUR;
```

EXPLODE PARTIEL MATRICE

Cet opérateur décompose partiellement une matrice de dimension *Modulo*, en fonction du facteur K :

```
-- ******** EXPLODE PARTIEL MATRICE ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity EXPLODEPM is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpExp;
           Modulo : integer := DIMENSION;
           Modp : integer := DIMENSION/FACTEUR;
           K : integer := FACTEUR);
  port (Entree : in MAT_SIG(0 to Modulo-1);
        sortie : out MMAT_SIG(0 to K-1));
end EXPLODEPM;

architecture OPERATEUR of EXPLODEPM is
begin
  EXPLO : process (Entree)
  begin
    for i in 0 to (K - 1) loop
      if i < (Modulo rem K) then
        for j in 0 to Modp loop
          sortie(i)(j) <= Entree(i*(Modp+1)+j);
        end loop; -- j
      else
        for j in 0 to Modp-1 loop
          sortie(i)(j) <= Entree(i*Modp+j+Modulo rem K);
        end loop; -- j
      end if;
    end loop; -- i
  end process EXPLO;
end OPERATEUR;
```

ADDITION

Cet opérateur effectue l'addition sans retenu entre deux signaux :

```
- ******** ADDITION ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity ADD is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpAdd);
  port (Entre1 : in QSV(wsize-1 downto 0);
        Entre2 : in QSV(wsize-1 downto 0);
        sortie : out QSV(wsize-1 downto 0));
end ADD;

architecture OPERATEUR of ADD is
begin
  sortie <= to_qsim_state(to_integer(Entre1) + to_integer(Entre2), wsize);
end OPERATEUR;
```

ADDITION AVEC 'CARRY'

Cet opérateur effectue l'addition avec retenu entre deux signaux :

```
- ******** ADDITION AVEC 'CARRY'********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity ADDITIONNE is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpAdd);
  port (Entre1 : in QSV(wsize-1 downto 0);
        Entre2 : in QSV(wsize-1 downto 0);
        sortie : out QSV(wsize downto 0));
end ADDITIONNE;

architecture OPERATEUR of ADDITIONNE is
begin
  sortie <= to_qsim_state(to_integer(Entre1) + to_integer(Entre2), wsize+1);
end OPERATEUR;
```

MULTIPLICATION

Cet opérateur effectue la multiplication entre deux signaux :

```
-- ******** MULTIPLICATION ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity MULTIPLIE is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpMul);
  port (Entre1 : in QSV(wsize-1 downto 0);
        Entre2 : in QSV(wsize-1 downto 0);
        sortie : out QSV(2*wsize-1 downto 0));
end MULTIPLIE;

architecture OPERATEUR of MULTIPLIE is
begin
  sortie <= to_qsim_state(to_integer(Entre1) * to_integer(Entre2), 2*wsize);
end OPERATEUR;
```

B.3 Opérateurs frontière de factorisation

Ci-dessous nous montrons le code VHDL correspondant à l'implantation des unités de contrôle des frontières infinie et finie, des opérateurs *FORK*, *JOIN*, *ITERATE* et *DIFFUSION*.

UNITÉ DE CONTRÔLE DE FRONTIÈRE INFINIE

Cet opérateur implante l'unité de contrôle associée à une frontière de factorisation infinie :

```
-- ******** UNITE DE CONTROLE DE FRONTIERE INFINIE ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity UCI is
  generic (Modulo : integer := DIMENSION);
  port (en : out qsim_state;
        rfd : out qsim_state;
        afu : out qsim_state;
        rsu : in qsim_state;
        afd : in qsim_state;
        asd : in qsim_state);
end UCI;

use work.all;

architecture OPERATEUR of UCI is
begin
  U : process (rsu, afd, asd)
  begin
    rfd <= rsu;
    afu <= asd;
    en <= afd;
  end process U;

end OPERATEUR;
```

UNITÉ DE CONTRÔLE DE FRONTIÈRE FINIE

Cet opérateur implante l'unité de contrôle associée à une frontière de factorisation finie :

```
-- ******** UNITE DE CONTROLE DE FRONTIERE FINIE ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity UC is
  generic (Modulo : integer := DIMENSION);
  port (en : out qsim_state;
        cpt : out qsim_state_vector(Modulo-1 downto 0);
        rfd : out qsim_state;
        afu : out qsim_state;
        rsd : out qsim_state;
        asu : out qsim_state;
        clk : in qsim_state;
        rst : in qsim_state;
        rsu : in qsim_state;
        rfu : in qsim_state;
        asd : in qsim_state;
        afd : in qsim_state);
end UC;

use work.all;

architecture OPERATEUR of UC is

component COMPTEUR
  generic (Modulo : integer := DIMENSION);
  port (Compte : out qsim_state_vector(Modulo-1 downto 0);
        Fin : out qsim_state;
        Clock : in qsim_state;
        Reset : in qsim_state;
        Init : in qsim_state;
        Enable : in qsim_state);
end component;

signal fin : qsim_state;          -- signal fin de comptage
signal ini : qsim_state;          -- signal d'initialisation
signal enl : qsim_state;          -- signal de validation

begin
  C : COMPTEUR generic map(Modulo)
    port map(cpt, fin, clk, rst, ini, enl);

  U : process (clk, fin, rst, rsu, rfu, asd, afd)
  begin
    ini <= rst or (fin and afd and asd);
    enl <= rsu and afd;
    rfd <= rsu;
    asu <= fin and afd;
    rsd <= fin and rfu;
    afu <= asd or (not fin and (rsu and afd));
    en <= afd and rsu and (not fin or asd);
  end process U;

end OPERATEUR;
```

FORK : VECTEUR – > SCALAIRE

Cet opérateur effectue la factorisation d'un vecteur, en énumérant des scalaires en sortie :

```
- ******** FORK : VECTEUR – > SCALAIRE ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity FORK is
  generic (wsize : integer := NBITS;
            SetUpTC : time := SetUpT;
            SetUpX : time := SetUpExp;
            Modulo : integer := DIMENSION);
  port (entree : in VECT_SIG(0 to Modulo-1);
        sortie : out QSV(wsize-1 downto 0);
        cpt : in qsim_state_vector(0 to Modulo-1));
end FORK;

use work.all;

architecture OPERATEUR of FORK is

  component EXPLODEV
    generic (wsize : integer := NBITS;
              SetUp : time := SetUpExp;
              Dim : integer := DIMENSION);
  port (Entree : in VECT_SIG(0 to Dim-1);
        sortie : out VECT_SIG(0 to Dim-1));
  end component;

  component TC
  generic (SetUpTC : time := SetUpT;
              Modulo : integer := DIMENSION);
  port (entree : in qsim_state_vector(Modulo-1 downto 0);
        sortie : out qsim_state_vector(Modulo-1 downto 0));
  end component;

  component CONVERT
  generic (Modulo : integer := DIMENSION;
              SetUpCv : time := SetUpConv);
  port (entree : in qsim_state_vector(Modulo-1 downto 0);
        sortie : out integer range 0 to Modulo-1);
  end component;

signal s_exp : VECT_SIG(0 to Modulo-1);
signal s_tc : qsim_state_vector(Modulo-1 downto 0);
signal index : integer range 0 to Modulo-1;

begin
  X : EXPLODEV generic map(wsize, SetUpX, Modulo)
    port map(entree, s_exp);

  T : TC generic map(SetUpT, Modulo)
    port map(cpt, s_tc);

  C : CONVERT generic map (Modulo, SetUpConv)
    port map (s_tc, index);

  S : process (index, s_exp)
  begin - process
    sortie <= s_exp(index)(wsize-1 downto 0);
  end process S;
end OPERATEUR;
```

FORK : MATRICE – > VECTEUR

Cet opérateur effectue la factorisation d'une matrice, en énumérant des vecteurs en sortie :

```
-- ******** FORK : MATRICE – > VECTEUR ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity FORKM is
  generic (wsize : integer := NBITS;
           SetUpTC : time := SetUpT;
           SetUpX : time := SetUpExp;
           dimL : integer := NLIG;
           dimC : integer := NCOL);
  port (entree : in MAT_SIG(0 to dimL-1);
        sortie : out VECT_SIG(0 to dimC-1);
        cpt : in qsim_state_vector(dimL-1 downto 0);
end FORKM;

use work.all;

architecture OPERATEUR of FORKM is

  component EXPLODEM
    generic (wsize : integer := NBITS;
             SetUp : time := SetUpExp;
             Dim : integer := dimL);
  port (Entree : in MAT_SIG(0 to Dim-1);
        sortie : out MAT_SIG(0 to Dim-1));
  end component;

  component TC
  generic (SetUpTC : time := SetUpT;
           Modulo : integer := dimL);
  port (entree : in qsim_state_vector(Modulo-1 downto 0);
        sortie : out qsim_state_vector(Modulo-1 downto 0));
  end component;

  component CONVERT
  generic (Modulo : integer := dimL;
           SetUpCv : time := SetUpConv);
  port (entree : in qsim_state_vector(Modulo-1 downto 0);
        sortie : out integer range 0 to Modulo-1);
  end component;

signal s_exp : MAT_SIG(0 to dimL-1);
signal s_tc : qsim_state_vector(dimL-1 downto 0);
signal index : integer range 0 to dimL-1;

begin
  X : EXPLODEM generic map(wsize, SetUpX, dimL)
    port map(entree, s_exp);

  T : TC generic map(SetUpT, dimL)
    port map(cpt, s_tc);

  C : CONVERT generic map (dimL, SetUpConv)
    port map (s_tc, index);

  S : process (s_exp, index)
  begin – process
    sortie <= s_exp(index);
  end process S;
end OPERATEUR;
```

JOIN : SCALAIRE – > VECTEUR

Cet opérateur effectue la factorisation d'un flot de scalaires, en les collectant sous la forme d'un vecteur en sortie :

```
-- ******** JOIN : SCALAIRE – > VECTEUR ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity JOIN is
  generic (wsize : integer := NBITS;
           SetUpTC : time := SetUpT;
           SetUpM : time := SetUpImp;
           Modulo : integer := DIMENSION);
  port (entree : in QSV(wsize-1 downto 0));
        sortie : out VECT_SIG(Modulo-1 downto 0);
        cpt : in qsim_state_vector(0 to Modulo-1);
        en : in qsim_state;
        clk : in qsim_state);
end JOIN;

use work.all;

architecture OPERATEUR of JOIN is

  component IMPLODEV
    generic (wsize : integer := NBITS;
             SetUp : time := SetUpImp;
             Dim : integer := DIMENSION);
  port (Entree : in VECT_SIG(0 to Dim-1);
        sortie : out VECT_SIG(0 to Dim-1));
  end component;

  component REGISTRE_D
    generic (wsize : integer := NBITS;
             TLH : time := SetUpReg;
             THL : time := SetUpReg);
  port (Q : out QSV(wsize-1 downto 0);
        D : in QSV(wsize-1 downto 0);
        clk : in qsim_state;
        en : in qsim_state);
  end component;

for all : REGISTRE_D use entity work.REGISTRE_D;

  component TC
  generic (SetUpTC : time := SetUpT;
           Modulo : integer := DIMENSION);
  port (entree : in qsim_state_vector(Modulo-1 downto 0);
        sortie : out qsim_state_vector(Modulo-1 downto 0));
  end component;

signal e_imp : VECT_SIG(0 to Modulo-1);
signal s_tc : qsim_state_vector(Modulo-1 downto 0);
signal enbl : qsim_state_vector(Modulo-2 downto 0);

begin
  T : TC generic map(SetUpT, Modulo)
    port map(cpt, s_tc);

  REGISTRES : for i in 0 to Modulo-2 generate
    R : REGISTRE_D generic map (wsize, SetUpReg, SetUpReg)
      port map (e_imp(i)(wsize-1 downto 0), entree, clk, enbl(i));
  end generate REGISTRES;

  L : process (entree, en, s_tc)
  begin
    for j in 0 to Modulo-2 loop
      enbl(j) <= en and s_tc(j);
    end loop; -j
    e_imp(Modulo-1)(wsize-1 downto 0) <= entree;
  end process L;

  X : IMPLODEV generic map(wsize, SetUpM, Modulo)
    port map(e_imp, sortie);

end OPERATEUR;
```

JOIN : VECTEUR − > MATRICE

Cet opérateur effectue la factorisation d'un flot de vecteurs, en les collectant sous la forme d'une matrice en sortie :

```
-- ******** JOIN : VECTEUR - > MATRICE ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity JOINM is
  generic (wsize : integer := NBITS;
           SetUpTC : time := SetUpT;
           SetUpM : time := SetUpImp;
           SetUpR : time := SetUpReg;
           Modulo : integer := DIMENSION);
  port (entree : in VECT_SIG(0 to Modulo-1);
        sortie : out MAT_SIG(0 to Modulo-1);
        cpt : in qsim_state_vector(Modulo-1 downto 0);
        en : in qsim_state;
        clk : in qsim_state);
end JOINM;

use work.all;

architecture OPERATEUR of JOINM is

  component IMPLODEM
    generic (wsize : integer := NBITS;
             SetUp : time := SetUpImp;
             Dim : integer := DIMENSION);
  port (entree : in MAT_SIG(0 to Dim-1);
        sortie : out MAT_SIG(0 to Dim-1));
  end component;

  component NREGISTRES_D
    generic (Modulo : integer := DIMENSION;
             TLH : time := SetUpReg;
             THL : time := SetUpReg);
  port (Q : out VECT_SIG(0 to Modulo-1);
        D : in VECT_SIG(0 to Modulo-1);
        clk : in qsim_state;
        en : in qsim_state);
  end component;

for all : REGISTRE_D use entity work.REGISTRE_D;

  component TC
  generic (SetUpTC : time := SetUpT;
           Modulo : integer := DIMENSION);
  port (entree : in qsim_state_vector(0 downto Modulo-1);
        sortie : out qsim_state_vector(0 to Modulo-1));
  end component;

signal e_imp : MAT_SIG(0 to Modulo-1);
signal s_tc : qsim_state_vector(0 to Modulo-1);
signal en1, en2 : qsim_state;

begin
  T : TC generic map(SetUpT, Modulo)
    port map(cpt, s_tc);

    R1 : NREGISTRES_D generic map (Modulo, SetUpReg, SetUpReg)
      port map (e_imp(Modulo-3), entree, clk, en1);
    R2 : NREGISTRES_D generic map (Modulo, SetUpReg, SetUpReg)
      port map (e_imp(Modulo-2), entree, clk, en2);

  L : process (entree, en, s_tc)
  begin
    en1 <= en and s_tc(Modulo-3);
    en2 <= en and s_tc(Modulo-2);
    e_imp(Modulo-1) <= entree;
  end process L;

  M : IMPLODEM generic map(wsize, SetUpM, Modulo)
    port map(e_imp, sortie);

end OPERATEUR;
```

SUB-ITERATE SCALAIRE

Cet opérateur est utilisé pour implanter l'opérateur *ITERATE* scalaire :

```
-- ******** SUB-ITERATE SCALAIRE ********
entity SUB_ITERATE is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpIt;
           Modulo : integer := DIMENSION);
  port (Init : in QSV(wsize-2 downto 0);
        Entree : in QSV(wsize-2 downto 0);
        partiel : out QSV(wsize-2 downto 0);
        Compte : in qsim_state_vector(Modulo-1 downto 0);
        Enable : in qsim_state;
        Clock : in qsim_state);
end SUB_ITERATE;

use work.all;

architecture OPERATEUR of SUB_ITERATE is

signal Retard : QSV(wsize-2 downto 0);

begin
  I : process (Clock, Compte, Entree, Init, Retard, Enable)
  begin
    if Clock'event and (Clock = '1') and (Clock'last_value = '0') then
      if Enable = '1' then
        Retard := Entree;
      end if;
    end if;
    if Compte(0) = '1' then
      partiel <= init(wsize-2 downto 0);
    else
      partiel <= Retard(wsize-2 downto 0);
    end if;
  end process I;

end OPERATEUR;
```

ITERATE SCALAIRE

Cet opérateur implante la factorisation des dépendances de données inter-motifs :

```
-- ******** ITERATE SCALAIRE ********
entity ITERATE is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpIt;
           Modulo : integer := DIMENSION);
  port (e_0 : in QSV(wsize-2 downto 0);
        e_i : in QSV(wsize-1 downto 0);
        s_i : out QSV(wsize-2 downto 0);
        s_f : out QSV(wsize-1 downto 0);
        cpt : in qsim_state_vector(Modulo-1 downto 0);
        en : in qsim_state;
        clk : in qsim_state);
end ITERATE;

use work.all;

architecture OPERATEUR of ITERATE is
begin
  subitr : SUB_ITERATE generic map(wsize, SetUp, Modulo)
    port map(e_0, e_i(wsize-2 downto 0), s_i, cpt, en, clk);
  I : process (e_i)
  begin
    s_f <= e_i;
  end process I;

end OPERATEUR;
```

DIFFUSION SCALAIRE

Cet opérateur implante la factorisation d'un flot de données (scalaire) qui traverse une frontière de factorisation :

```
-- ******** DIFFUSION SCALAIRE ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity DIFFUSION is
   generic (wsize : integer := NBITS;
             SetUp : time := SetUpDiff);
   port (entree : in QSV(wsize-2 downto 0);
         Sortie : out QSV(wsize-2 downto 0));
end DIFFUSION;

use work.all;

architecture OPERATEUR of DIFFUSION is
begin
   Sortie <= entree;
end OPERATEUR;
```

DIFFUSION VECTEUR

Cet opérateur implante la factorisation d'un flot de données (vecteur) qui traverse une frontière de factorisation :

```
-- ******** DIFFUSION VECTEUR ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity DIFFUSIONV is
   generic (wsize : integer := NBITS;
             SetUpD : time := SetUpDiff;
             dim : integer := DIMENSION);
   port (entree : in VECT_SIG(0 to dim-1);
         sortie : out VECT_SIG(0 to dim-1));
end DIFFUSIONV;

use work.all;

architecture OPERATEUR of DIFFUSIONV is
begin
   sortie <= entree;
end OPERATEUR;
```

FORK INFINI : VECTEURS

Cet opérateur correspond à un capteur qui fournit des données sous la forme d'un vecteur :

```
-- ******** FORK INFINI : VECTEURS ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity FORKIV is
  generic (wsize : integer := NBITS;
           dim : integer := DIMENSION;
           SetUp : time := SetUpSt);
  port (Entree : in VECT_SIG(0 to dim-1);
        sortie : out VECT_SIG(0 to dim-1)
        Enable : in qsim_state;
        Clock : in qsim_state;
        Reset : in qsim_state);
end FORKIV;

use work.all;

architecture OPERATEUR of FORKIV is
begin
  S : process (Reset, Clock, Enable, Entree)
  begin
    if Reset = '1' then
      for i in 0 to (dim-1) loop
        sortie(i) <= (others => '0');
      end loop;
    elsif (Clock'event) and (Clock = '1') and (Clock'last_value = '0') then
      if Enable = '1' then
        sortie <= Entree;
      end if;
    end if;
  end process S;
end OPERATEUR;
```

FORK INFINI : MATRICES

Cet opérateur correspond à un capteur qui fournit des données sous la forme d'une matrice :

```
-- ******** FORK INFINI : MATRICES ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity FORKIM is
  generic (wsize : integer := NBITS;
           dim : integer := DIMENSION;
           SetUp : time := SetUpSt);
  port (Entree : in MAT_SIG(0 to dim-1);
        sortie : out MAT_SIG(0 to dim-1)
        Enable : in qsim_state;
        Clock : in qsim_state;
        Reset : in qsim_state);
end FORKIM;

use work.all;

architecture OPERATEUR of FORKIM is
begin
  S : process (Reset, Clock, Enable, Entree)
  begin
    if Reset = '1' then
      for i in 0 to (dim-1) loop
        for j in 0 to (dim-1) loop
          sortie(i)(j) <= (others => '0');
        end loop;
      end loop;
    elsif (Clock'event) and (Clock = '1') and (Clock'last_value = '0') then
      if Enable = '1' then
        sortie <= Entree;
      end if;
    end if;
  end process S;
end OPERATEUR;
```

JOIN INFINI : SCALAIRES

Cet opérateur correspond à un actionneur qui reçoit des données sous la forme d'un scalaire :

```
-- ******** JOIN INFINI : SCALAIRES ********
library work;
use work.DEFINITIONS.all;

entity JOINI is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpSt);
  port (Entree : in QSV(wsize-1 downto 0);
        sortie : out QSV(wsize-1 downto 0)
        Enable : in qsim_state;
        Clock : in qsim_state;
        Reset : in qsim_state);
end JOINI;

architecture OPERATEUR of JOINI is
begin
  S : process (Reset, Clock, Entree, Enable)
  begin
    if Reset = '1' then
      sortie <= (others => '0');
    elsif (Clock'event) and (Clock = '1') and (Clock'last_value = '0') then
      if Enable = '1' then
        sortie <= Entree;
      end if;
    end if;
  end process S;
end OPERATEUR;
```

JOIN INFINI : VECTEURS

Cet opérateur correspond à un actionneur qui reçoit des données sous la forme d'un vecteur :

```
-- ******** JOIN INFINI : VECTEURS ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity JOINIV is
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpSt;
           dim : integer := DIMENSION);
  port (Entree : in VECT_SIG(0 to dim-1);
        sortie : out VECT_SIG(0 to dim-1);
        Enable : in qsim_state;
        Clock : in qsim_state;
        Reset : in qsim_state);
end JOINIV;

architecture OPERATEUR of JOINIV is
begin
  S : process (Reset, Clock, Entree, Enable)
  begin
    if Reset = '1' then
      for i in 0 to (dim-1) loop
        sortie(i) <= (others => '0');
      end loop;
    elsif (Clock'event) and (Clock = '1') and (Clock'last_value = '0') then
      if Enable = '1' then
        sortie <= Entree;
      end if;
    end if;
  end process S;
end OPERATEUR;
```

JOIN INFINI : MATRICES

Cet opérateur correspond à un actionneur qui reçoit des données sous la forme d'une matrice :

```
- ******** JOIN INFINI : MATRICES ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity JOINIM is
  generic (wsize : integer := NBITS;
           dim : integer := DIMENSION;
           SetUp : time := SetUpSt);
  port (Entree : in MAT_SIG(0 to dim-1);
        sortie : out MAT_SIG(0 to dim-1);
        Enable : in qsim_state;
        Clock : in qsim_state;
        Reset : in qsim_state);
end JOINIV;

architecture OPERATEUR of JOINIM is
begin
  S : process (Reset, Clock, Entree, Enable)
  begin
    if Reset = '1' then
      for i in 0 to (dim-1) loop
        for j in 0 to (dim-1) loop
          sortie(i)(j) <= (others => '0');
        end loop;
      end loop;
    elsif (Clock'event) and (Clock = '1') and (Clock'last_value = '0') then
      if Enable = '1' then
        sortie <= Entree;
      end if;
    end if;
  end process S;
end OPERATEUR;
```

B.4 PMV factorisé : (∞, 6/1, 6/1)

Nous présentons ci-dessous le code VHDL correspondant à l'implantation totalement factorisée du PMV :

```
- ******** PMV TOTALEMENT FACTORISE ********
library mgc_portable;
use mgc_portable.qsim_logic.all;

library work;
use work.DEFINITIONS.all;

entity PMV is
  generic (wsize : integer := NBITS;
           dimL : integer := NLIG;
           dimC : integer := NCOL);
  port (EntreA : in MAT_SIG(0 to dimL-1);
        EntreB : in VECT_SIG(0 to dimC-1);
        sortie : out VECT_SIG(0 to dimC-1);
        Clock : in qsim_state;
        Rst : in qsim_state;
        GN : in qsim_state);
end PMV;

library designs;
use designs.all;

library work;
use work.DEFINITIONS.all;
```

```
architecture FACTORISE of PMV is

  component UC
    generic (Modulo : integer := DIMENSION);
    port (en : out qsim_state;
          cpt : out qsim_state_vector(Modulo-1 downto 0);
          rfd : out qsim_state;
          afu : out qsim_state;
          rsd : out qsim_state;
          asu : out qsim_state;
          clk : in qsim_state;
          rst : in qsim_state;
          rsu : in qsim_state;
          rfu : in qsim_state;
          asd : in qsim_state;
          afd : in qsim_state);
  end component;

  component UCI
    port (en : out qsim_state;
          rfd : out qsim_state;
          afu : out qsim_state;
          rsu : in qsim_state;
          afd : in qsim_state;
          asd : in qsim_state);
  end component;

  component FORKIM
    generic (wsize : integer := NBITS;
             dim : integer := DIMENSION;
             SetUp : time := SetUpSt);
    port (Entree : in MAT_SIG(0 to dim-1);
          sortie : out MAT_SIG(0 to dim-1);
          Enable : in qsim_state;
          Clock : in qsim_state;
          Reset : in qsim_state);
  end component;

  component FORKIV
    generic (wsize : integer := NBITS;
             dim : integer := DIMENSION;
             SetUp : time := SetUpSt);
    port (Entree : in VECT_SIG(0 to dim-1);
          sortie : out VECT_SIG(0 to dim-1);
          Enable : in qsim_state;
          Clock : in qsim_state;
          Reset : in qsim_state);
  end component;

  component FORKM
    generic (wsize : integer := NBITS;
             SetUpTC : time := SetUpT;
             SetUpX : time := SetUpExp;
             dimL : integer := NLIG;
             dimC : integer := NCOL);
    port (entree : in MAT_SIG(0 to dimL-1);
          sortie : out VECT_SIG(0 to dimC-1);
          cpt : in qsim_state_vector(0 to dimL-1));
  end component;

  component DIFFUSIONV
    generic (wsize : integer := NBITS;
             SetUpD : time := SetUpDiff;
             dim : integer := dimC);
    port (entree : in VECT_SIG(0 to dimC-1);
          sortie : out VECT_SIG(0 to dimC-1));
  end component;

  component FORK
    generic (wsize : integer := NBITS;
             SetUpTC : time := SetUpT;
             SetUpX : time := SetUpExp;
             Modulo : integer := DIMENSION);
    port (entree : in VECT_SIG(0 to Modulo-1);
          sortie : out QSV(wsize-1 downto 0);
          cpt : in qsim_state_vector(0 to Modulo-1));
  end component;

  component ITERATE
    generic (wsize : integer := NBITS;
             SetUp : time := SetUpIt;
             Modulo : integer := DIMENSION);
    port (e_0 : in QSV(wsize-2 downto 0);
          e_i : in QSV(wsize-1 downto 0);
          s_i : out QSV(wsize-2 downto 0);
          s_f : out QSV(wsize-1 downto 0);
          cpt : in qsim_state_vector(Modulo-1 downto 0);
          en : in qsim_state;
          clk : in qsim_state);
  end component;
```

```
component MULTIPLIE
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpMul);
  port (Entre1 : in QSV(wsize-1 downto 0);
        Entre2 : in QSV(wsize-1 downto 0);
        sortie : out QSV(2*wsize-1 downto 0));
end component;

component ADDITIONNE
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpAdd);
  port (Entre1 : in QSV(wsize-1 downto 0);
        Entre2 : in QSV(wsize-1 downto 0);
        sortie : out QSV(wsize downto 0));
end component;

component JOIN      generic (wsize : integer := NBITS;
           SetUpTC : time := SetUpT;
           SetUpM : time := SetUpImp;
           SetUpR : time := SetUpReg;
           Modulo : integer := DIMENSION);
  port (entree : in QSV(wsize-1 downto 0);
        sortie : out VECT_SIG(Modulo-1 downto 0);
        cpt : in qsim_state_vector(Modulo-1 downto 0);
        en : in qsim_state;
        clk : in qsim_state);
end component;

component JOINIV
  generic (wsize : integer := NBITS;
           SetUp : time := SetUpSt;
           dim : integer := dimC);
  port (Entree : in VECT_SIG(0 to dim-1);
        sortie : out VECT_SIG(0 to dim-1);
        Enable : in qsim_state;
        Clock : in qsim_state;
        Reset : in qsim_state);
end component;

signal Matrice : MAT_SIG(0 to dimL-1);
signal Vecteur : VECT_SIG(0 to dimC-1);
signal dif21 : VECT_SIG(0 to dimC-1);
signal InitVl : QSV(2*wsize-1 downto 0);
signal j21, frk21 : VECT_SIG(0 to dimC-1);
signal frk31, frk32 : QSV(wsize-1 downto 0);
signal mlt : QSV(2*wsize-1 downto 0);
signal itrp : QSV(2*wsize-1 downto 0);
signal add, itrf : QSV(2*wsize downto 0);
signal cpt2 : qsim_state_vector(dimL-1 downto 0);
signal cpt3 : qsim_state_vector(dimC-1 downto 0);
signal en1, en2, en3 : qsim_state;
signal rfd1, rfd2, rfd3 : qsim_state;
signal afu1, afu2, afu3 : qsim_state;
signal rsd2, rsd3 : qsim_state;
signal asu2, asu3 : qsim_state;
signal s_un : qsim_state;

begin
  -****** UC FRONTIERE 1 : MODULO INFINI
  s_un <= not GN;
  uc1 : UCI
    port map(en => en1,
             rfd => rfd1,
             afu => afu1,
             rsu => s_un,
             afd => asu2,
             asd => s_un);

  -****** UC FRONTIERE 2 : MODULO NOMBRE DE LIGNES DE LA MATRICE
  uc2 : UC generic map(dimL)
    port map(en => en2,
             cpt => cpt2,
             rfd => rfd2,
             afu => afu2,
             rsd => rsd2,
             asu => asu2,
             clk => Clock,
             rst => Rst,
             rsu => rfd1,
             rfu => rsd3,
             asd => afu1,
             afd => asu3);
```

```
--****** UC FRONTIERE 3 : MODULO NOMBRE DE COLONNES DE LA MATRICE
uc3 : UC generic map(dimC)
  port map(en => en3,
           cpt => cpt3,
           rfd => rfd3,
           afu => afu3,
           rsd => rsd3,
           asu => asu3,
           clk => Clock,
           rst => Rst,
           rsu => rfd2,
           rfu => rfd3,
           asd => afu2,
           afd => afu3);

--****** FORK INFINI : CAPTEUR DES MATRICES
forkinf1 : FORKIM generic map(wsize, dimL, SetUpSt)
  port map(Entree => EntreA,
           sortie => Matrice(0 to dimL-1),
           Enable => en1,
           Clock => Clock,
           Reset => Rst);

--****** FORK INFINI : CAPTEUR DES VECTEURS
forkinf2 : FORKIV generic map(wsize, dimC, SetUpSt)
  port map(Entree => EntreB,
           sortie => Vecteur(0 to dimC-1),
           Enable => en1,
           Clock => Clock,
           Reset => Rst);

--****** FORK FINI : SEPARE LES LIGNES DE LA MATRICE
fork21 : FORKM generic map(wsize, SetUpT, SetUpExp, dimL, dimC)
  port map(Entree => Matrice(0 to dimL-1),
           sortie => frk21(0 to dimC-1),
           cpt => cpt2);

--****** DIFFUSION FINI : DIFFUSE LE VECTEUR
diff21 : DIFFUSIONV generic map(wsize, SetUpSt, dimC)
  port map (Entree => Vecteur(0 to dimC-1),
           sortie => dif21(0 to dimC-1));

--****** FORK FINI : SEPARE LES ELEMENTS DE LA MATRICE
fork31 : FORK generic map(wsize, SetUpT, SetUpExp, dimC)
  port map(entree => frk21(0 to dimC-1),
           sortie => frk31,
           cpt => cpt3);

--****** FORK FINI : SEPARE LES ELEMENTS DU VECTEUR
fork32 : FORK generic map(wsize, SetUpT, SetUpExp, dimC)
  port map(entree => dif21(0 to dimC-1),
           sortie => frk32,
           cpt => cpt3);

--################################################
--********** CALCUL **********
mult : MULTIPLIE generic map(wsize, SetUpMul)
  port map(Entre1 => frk31,
           Entre2 => frk32,
           sortie => mlt);

add1 : ADDITIONNE generic map(2*wsize, SetUpAdd)
  port map(Entre1 => mlt,
           Entre2 => itrp,
           sortie => add);
--################################################

InitVl <= (others => GN);
--****** ITERATE FINI : REPETITION DU CALCUL POUR CHAQUE ELEMENT
iter31 : ITERATE generic map(2*wsize+1, SetUpIt, dimC)
  port map(e_0 => InitVl,
           e_i => add,
           s_i => itrp,
           s_f => itrf,
           cpt => cpt3,
           en => en3,
           clk => Clock);

--****** JOIN FINI : REGROUPEMENT DES ELEMENTS SOUS LA FORME D'UN VECTEUR
join21 : JOIN generic map(2*wsize+1, SetUpT, SetUpImp, SetUpReg, dimC)
  port map(entree => itrf,
           sortie => j21(0 to dimC-1),
           cpt => cpt2,
           en => en2,
           clk => Clock);
```

```
  -****** JOIN INFINI : ACTIONNEUR DU VECTEUR
  joinf : JOINIV generic map(2*wsize+1, SetUpSt, dimC)
    port map(Entree => j21(0 to dimC-1),
             sortie => sortie(0 to dimC-1),
             Enable => en1,
             Clock => Clock,
             Reset => Rst);


end FACTORISE;
```