Méthodologie de conception de composants virtuels comportementaux pour une chaîne de traitement du signal embarquée

Guillaume Savaton

Thèse

pour obtenir le titre de

Docteur de l'Université de Bretagne Sud

Mention : Sciences et Sciences de l'Ingénieur

présentée et soutenue publiquement par

Guillaume Savaton

le 10 décembre 2002

Méthodologie de Conception de Composants Virtuels Comportementaux pour une Chaîne de Traitement du Signal Embarquée

---

Directeur de thèse : Eric MARTIN

Jury composé de :

MM.	E. CASSEAU	Maître de conférences à l'Université de Bretagne Sud	Examinateur
	C. LAMBERT-NEBOUT	Docteur-Ingénieur de recherche au CNES	Examinateur
	E. MARTIN	Professeur à l'Université de Bretagne Sud	Examinateur
	M. PAINDAVOINE	Professeur à l'Université de Bourgogne	Rapporteur
	H. PELON	Ingénieur recherche et développement, Astrium	Invité
	P. QUINTON	Professeur à l'Université de Rennes I	Rapporteur
	M. ROBERT	Professeur à l'Université de Montpellier II	Examinateur

Laboratoire d'Electronique des Systèmes Temps Réel (LESTER)
Université de Bretagne Sud, Lorient

Remerciements

Les travaux présentés dans ce mémoire ont été effectués au *Laboratoire d'Electronique des Systèmes Temps Réel* (*LESTER*, Université de Bretagne Sud), en collaboration avec le *Centre National d'Etudes Spatiales* (*CNES*, Toulouse) et la société *Astrium* (Vélizy).

Je tiens tout d'abord à exprimer ma reconnaissance envers Monsieur Eric MARTIN, Professeur des Universités à l'Université de Bretagne Sud et directeur du *LESTER*, pour m'avoir accueilli dans son équipe de recherche, m'avoir proposé ce sujet de thèse et avoir accepté d'être mon directeur de thèse. Ses conseils et critiques constructives m'ont été une aide précieuse tout au long de ce travail.

Je tiens également à remercier Monsieur Emmanuel CASSEAU, Maître de Conférences à l'Université de Bretagne Sud, pour avoir co-encadré ce travail, en toute simplicité mais non sans efficacité.

Je remercie Madame Catherine LAMBERT-NEBOUT et Monsieur Bruno SABA, ingénieurs de recherche au *CNES*, Messieurs Hubert PELON, Marc SOUYRI et Jean-Luc POUPAT, ingénieurs recherche et développement au sein de la société *Astrium*, pour leur participation à l'avancement de ces travaux et pour leur pragmatisme dans l'évaluation des résultats.

Je remercie Messieurs Michel PAINDAVOINE, Patrice QUINTON et Michel ROBERT, qui me font l'honneur de participer à mon jury de thèse.

Merci à tous les membres du *LESTER*, enseignants-chercheurs, ingénieurs de recherche, doctorants, secrétaire, pour leur disponibilité et leur bonne humeur.

Ma reconnaissance va également à Monsieur Alain LE DUFF, enseignant-chercheur à l'*Ecole Supérieure d'Electronique de l'Ouest* (*ESEO*, Angers) , qui m'a encouragé dans mon cheminement vers la thèse, et m'a fait connaître le *LESTER*.

Enfin, ma profonde gratitude pour Muriel, qui a surmonté avec patience les longs mois de rédaction de ce mémoire.

Résumé

L'offre de services en imagerie spatiale tend à faire converger les techniques de traitement et de codage des données à bord des satellites avec celles utilisées dans les applications multimedia grand public. Les futures générations de satellites d'observation de la Terre doivent notamment faire face à des besoins croissants en résolution, précision et qualité des images. Le coût du stockage de tels volumes de données à bord des satellites, et la bande passante limitée des canaux de transmission imposent de recourir à de nouvelles techniques de compression des images, parmi lesquelles le standard *JPEG2000* est un candidat prometteur. Ces techniques se caractérisent par des algorithmes de forte complexité qui exigent un effort accru de la part des concepteurs de circuits. Les systèmes embarqués en milieu spatial doivent en effet satisfaire des contraintes fortes de faible encombrement, faible consommation, tolérance aux radiations, et de traitement des informations en temps réel.

Les nouvelles technologies intégrées de type *ASIC* ou *FPGA* autorisent aujourd'hui l'implantation d'un système complet, de complexité matérielle équivalant à plusieurs dizaines de millions de transistors, sur le même substrat. La notion de système sur puce, ou *SoC* pour *System-on-a-Chip*, repose ainsi sur l'intégration dans un même circuit d'une variété de fonctionnalités combinant des parties numériques et analogiques, matérielles et logicielles, munies d'une infrastructure de communication complexe allant du bus intégré aux réseaux sur puce (*NoC* pour *Network-on-Chip*).

Face à la complexité croissante des applications et des technologies, les outils et méthodologies de conception et de vérification classiques apparaissent inadaptés à la réalisation des systèmes embarqués dans des délais raisonnables. Les nouvelles approches envisagées reposent tout d'abord sur une élévation du niveau d'abstraction de la spécification d'un système permettant, par le biais de modèles formels et de langages de haut niveau, d'orthogonaliser les choix d'implantation relatifs à chaque sous-système (implantation en matériel ou en logiciel, choix des interfaces de communication, *etc.*) et d'accélérer la vérification d'un système à différents niveaux d'abstraction. Afin d'accélérer le raffinement de la spécification abstraite d'un système en une architecture synthétisable, l'intégration des différents blocs fonctionnels qui le constituent privilégie la réutilisation de composants pré-définis et pré-vérifiés – dénommés *composants virtuels*, ou blocs *IP* (*Intellectual Property*) – plutôt que leur re-conception.

Dans cette thèse, nous nous intéressons à la conception de composants matériels réutilisables pour des applications intégrant des fonctions de traitement du signal et de l'image. Ces fonctions présentent plusieurs particularités que nous avons largement exploitées dans notre travail : tout d'abord, les algorithmes auxquels nous nous intéressons présentent une forte complexité calculatoire dont l'implantation matérielle

sous contrainte de performances n'est pas triviale. Ensuite, ces algorithmes sont fortement paramétrables en fonction des besoins liés à l'application cible : la spécification d'une architecture implémentant un tel algorithme doit ainsi faire apparaître un haut degré de flexibilité, en termes de fonctionnalité d'une part, et de performances d'autre part.

Ce haut degré de flexibilité est incompatible avec les méthodes de conception et de synthèse de matériel classiques au niveau transfert de registres (*RTL*). Notre approche consiste à élever le niveau d'abstraction de la spécification d'un composant virtuel et à bénéficier des outils de synthèse de haut niveau pour générer automatiquement une variété d'architectures, respectant une variété de contraintes de performances, à partir d'une unique description paramétrable du comportement d'un composant.

Notre travail a ainsi consisté à définir une méthodologie de conception de composants virtuels décrits au niveau comportemental et orientés vers les outils de synthèse de haut niveau. La nécessité d'une telle méthodologie est motivée en particulier par la variété des outils de synthèse de haut niveau – tant universitaires que commerciaux – disponibles et par l'absence de consensus quant à la syntaxe et à la sémantique des langages de description comportementale supportés. Cette diversité pose des limites fortes sur l'interopérabilité des flots de conception et de réutilisation d'un composant virtuel et sur la prédictibilité de ses performances.

Notre effort méthodologique a ainsi porté sur la définition de modèles formels de représentation d'une description comportementale à différents stades du flot de conception, permettant de consolider le lien entre conception au niveau système et synthèse de haut niveau. Ces modèles permettent tout d'abord d'attacher une sémantique précise à la notion de description comportementale indépendamment des outils de synthèse existants. Ils servent également de support à l'expression de contraintes de performances et répondent ainsi au besoin de prédictibilité des performances. Afin de résoudre la question de l'interopérabilité, nous proposons des techniques de transformation et de traduction de notre modèle de représentation vers le modèle, le langage et le style d'écriture requis par un outil de synthèse donné.

Nous avons expérimenté notre méthodologie sur l'implantation sous forme d'un composant virtuel comportemental d'un algorithme de transformation en ondelettes bidimensionnelle pour la compression d'images au format *JPEG2000*. La représentation du comportement à l'aide d'un modèle formel de type graphe de dépendances polyédral nous a permis d'explorer différentes alternatives de répartition des calculs au cours du temps et de mettre en évidence les paramètres pertinents pour la personnalisation du comportement et des performances du composant. La traduction de ce modèle en une description *VHDL* paramétrable de haut niveau compatible avec les règles d'écriture supportées par l'outil *Monet*, de *Mentor Graphics*, nous a permis d'explorer une variété de solutions architecturales présentant différentes performances en temps/surface et répondant à différents jeux de paramètres fonctionnels.

Table des matières

Remerciements iii

Résumé v

Introduction 1

I Méthodologie de Conception de Composants Virtuels Réutilisables de Haut Niveau 7

1 Nouvelles Approches en Conception de Systèmes Intégrés 9
1.1 Maîtriser la complexité : la conception au niveau système 10
1.1.1 Domaines de conception et niveaux d'abstraction 10
1.1.2 Propriétés d'un système 14
1.1.3 Modèles formels 16
1.1.4 Langages de conception de systèmes 19
1.2 Accélérer le flot de conception : la réutilisation de composants virtuels 20
1.2.1 Définition et problématique 22
1.2.2 Acteurs de la conception et de la réutilisation 22
1.2.3 Propriétés d'un composant virtuel et "réutilisabilité" 23
1.2.4 Interopérabilité des flots de conception et d'intégration 25
1.2.5 Protection de la propriété intellectuelle 27
1.2.6 Outils d'aide à la spécification et à la réutilisation de composants virtuels 28
1.3 Explorer l'espace des solutions architecturales : la synthèse de haut niveau 29
1.3.1 De la synthèse *RTL* à la synthèse de haut niveau 29
1.3.2 Décrire au niveau comportemental 30
1.3.3 Le flot de synthèse de haut niveau 32
1.3.4 Synthèse de haut niveau et synthèse système 37

2 Conception de Composants Virtuels Flexibles de Haut Niveau 39
2.1 Problématique 39
2.1.1 Synthèse de haut niveau et réutilisation 39
2.1.2 Vers une méthodologie de conception de composants virtuels comportementaux 42
2.1.3 Travaux connexes 43
2.2 Définitions préliminaires 45

2.2.1 Attributs d'un composant virtuel comportemental 45
2.2.2 Niveaux d'abstraction et hiérarchie des spécifications 48
2.3 Spécification et raffinement d'un composant virtuel comportemental . 53
2.3.1 Modélisation au niveau fonctionnel 53
2.3.2 Sélection d'algorithmes 54
2.3.3 Raffinement des formats de données 57
2.3.4 Extraction d'un motif répétitif de calcul 57
2.3.5 Insertion des contraintes de temps et d'entrée/sortie 61
2.3.6 Décomposition structurelle d'un motif 63
2.3.7 Résumé 63

3 Synthèse de Composants Virtuels Comportementaux 67
3.1 Exemple introductif 68
3.1.1 Déroulage des boucles 70
3.1.2 Non-déroulage des boucles 72
3.2 Modèle de représentation "universel" 78
3.2.1 Primitives de synthèse de haut niveau 78
3.2.2 Modélisation des primitives de synthèse de haut niveau 83
3.3 Des primitives de description comportementales aux primitives de synthèse de haut niveau 90
3.3.1 Flot de pré-synthèse de haut niveau 90
3.3.2 Primitives de description comportementale 91
3.3.3 Séparation contrôle/traitement 92

Bilan 99

II Conception et Synthèse d'un Composant Virtuel Comportemental pour la Transformation en Ondelettes Discrète 2D 105

4 Transformations en Ondelettes pour la Compression d'Images Embarquée 107
4.1 Contexte de l'étude : traitement bord et compression d'images spatiales 108
4.1.1 Spécificités des applications spatiales 108
4.1.2 Compression d'images spatiales et standards 109
4.2 Compression d'images et ondelettes : le nouveau standard JPEG2000 . 110
4.2.1 Principes de la compression d'images 110
4.2.2 Analyse multirésolution et transformation en ondelettes discrète 111
4.2.3 De JPEG à JPEG2000 113
4.3 Implantation de la transformation en ondelettes discrète 114
4.3.1 Algorithmes de transformation en ondelettes 114
4.3.2 Architectures VLSI de transformation en ondelettes 127

5 Conception d'un Composant Virtuel Comportemental pour la Transformation en Ondelettes Discrète 2D 137
5.1 Transformation en ondelettes pour JPEG2000 137
5.2 Transformation en ondelettes lifting 2D 138

5.2.1 Algorithme . . . 138
5.2.2 Choix des formats de données . . . 142
5.3 Transformation en ondelettes 2D au fil de l'eau . . . 144
5.3.1 Choix des axes temporels et causalité . . . 146
5.3.2 Choix de motifs de calcul . . . 149
5.3.3 Séparation du contrôle et du traitement dans le motif de transformation en ondelettes 2D . . . 151
5.3.4 Taille de motif, contraintes de temps et parallélisme de calcul . 152
5.3.5 Réduction de la complexité en vue de la synthèse de haut niveau 155
5.3.6 Contraintes d'entrée/sortie . . . 161
5.3.7 Résumé . . . 162
5.4 Synthèse du composant virtuel comportemental . . . 163
5.4.1 Instanciation du composant virtuel . . . 163
5.4.2 Résultats de synthèse . . . 165
5.4.3 Discussion . . . 167

Conclusion et Perspectives **169**

Bibliographie **173**

III Annexes **183**

A Etat de l'Art des Outils de Synthèse de Haut Niveau **185**
A.1 L'outil *GAUT* du LESTER/LASTI . . . 185
A.2 Behavioral Compiler et CoCentric SystemC Compiler de Synopsys . . 190
A.3 L'outil Monet de Mentor Graphics . . . 193
A.4 L'outil A|RT Designer d'Adelante Technologies . . . 195
A.5 L'outil Architectural Compiler de *Get2Chip* . . . 198
A.6 Résumé . . . 200

B Exploration des Graphes de Dépendances **203**
B.1 Définition . . . 203
B.2 Graphe de dépendances et ordre de parcours des données . . . 204
B.2.1 Ordre partiel des données d'entrée/sortie . . . 204
B.2.2 Ordre partiel des données intermédiaires . . . 205
B.3 Périodicité du graphe et motifs répétitifs de calcul . . . 206
B.3.1 Motif de calcul : définition . . . 207
B.3.2 Propriétés d'un motif . . . 207

C Machines à Etat Fini avec Chemin de Données **209**
C.1 FSM . . . 209
C.2 FSMD . . . 209
C.2.1 Définition . . . 209
C.2.2 Application . . . 210
C.3 SFSMD . . . 211

D Analyse Multirésolution et Transformation en Ondelettes Discrète 213
D.1 Analyse multirésolution 213
D.1.1 Approximations d'une fonction à différentes résolutions 213
D.1.2 Extraction des détails d'une fonction à différentes résolutions . 214
D.2 Transformation en ondelettes et son inverse 215
D.2.1 Transformation en ondelettes et décomposition en sous-bandes 215
D.2.2 La transformation en ondelettes bidimensionnelle 216
D.3 Transformation en ondelettes et compression 216

Table des figures

1.1 Le diagramme en Y de Gajski et Kuhn 11
1.2 Transitions dans le diagramme en Y 12
1.3 La taxonomie RASSP/VSIA 13
1.4 Spécification comportementale détaillée dans la taxonomie RASSP/VSIA 14
1.5 Interprétation d'une description comportementale par les outils de synthèse RTL et de haut niveau 31
1.6 Flot général de synthèse de haut niveau 33
1.7 Deux modèles de représentation d'une description comportementale . . 35
1.8 Différentes stratégies d'ordonnancement en synthèse de haut niveau . . 36

2.1 Influence des paramètres d'un composant virtuel sur l'architecture RTL 41
2.2 Synthèse de haut niveau et flexibilité 42
2.3 Attributs d'un composant virtuel comportemental 46
2.4 La synthèse de haut niveau dans le diagramme en Y 48
2.5 Spécification fonctionnelle 50
2.6 Spécification algorithmique en précision infinie 51
2.7 Spécification algorithmique en précision fixée 51
2.8 Spécification comportementale au niveau transaction 51
2.9 Spécification comportementale au cycle près 52
2.10 Spécification au niveau transfert de registres 52
2.11 Spécification structurelle au niveau transfert de registres 52
2.12 Raffinement d'un composant virtuel comportemental 55
2.13 Extrait du graphe de dépendances de l'algorithme d'interpolation . . . 60
2.14 Graphe de dépendances causal de l'algorithme d'interpolation 60
2.15 Différents motifs de traitement pour l'algorithme d'interpolation 62

3.1 Le flot de synthèse de haut niveau universel 68
3.2 Calcul de distance de Manhattan : réécriture de la description comportementale et CDFG correspondant 74
3.3 Calcul de distance de Manhattan : FSM générées par les outils Monet et BC 75
3.4 Calcul de distance de Manhattan : CDFG après optimisation manuelle de la description 77
3.5 Modélisation d'une séquence de primitives de traitement : langage C/-notation symbolique/DFG 81
3.6 Calcul de distance de Manhattan : SFSMD de Moore et de Mealy . . . 84

3.7 Calcul de distance de Manhattan : primitives de synthèse de haut niveau et modèle HSFSMD 86
3.8 Calcul de distance de Manhattan : relation entre HSFSMD et SFSMD 87
3.9 Fusion de transitions d'une HSFSMD à l'intérieur d'un même niveau de boucle 88
3.10 Fusion de transitions d'une HSFSMD à travers un niveau de boucle . . 89
3.11 Calcul de distance de Manhattan : HSFSMD optimisée 89
3.12 Modélisation des contraintes de temps et d'entrée/sortie par annotation des primitives de synthèse de haut niveau 90
3.13 Flot en V pour la synthèse de haut niveau 91
3.14 Des primitives de description comportementales aux primitives de synthèse de haut niveau : transformation des instructions d'affectation . . 97
3.15 Des primitives de description comportementales aux primitives de synthèse de haut niveau : transformation des instructions conditionnelles . 98
3.16 Des primitives de description comportementales aux primitives de synthèse de haut niveau : transformation des boucles 98

4.1 La chaîne image SPOT 109
4.2 Principe de la décomposition en sous-bandes 112
4.3 Décomposition en ondelettes 2D sur trois niveaux 113
4.4 Structure d'un codeur JPEG2000 114
4.5 Banc de filtres 1D pour la transformation en ondelettes sur trois niveaux 115
4.6 Banc de filtres 2D pour la transformation en ondelettes sur un niveau . 116
4.7 Banc de filtres 1D pour la transformation en ondelettes inverse sur trois niveaux 116
4.8 Banc de filtre 1D et sa structure lifting équivalente 118
4.9 Structure lifting d'un banc de filtres 1D de reconstruction 118
4.10 Différentes modes d'extension d'une image pour la transformation en ondelettes discrète 2D 121
4.11 Ordre des passes de filtrage pour l'algorithme en pyramide 123
4.12 Ordre des passes de filtrage pour l'algorithme en pyramide récursif . . 124
4.13 Parcours multirésolution d'une image à l'aide de la courbe fractale de MORTON 126
4.14 Implantation série d'un filtre FIR 128
4.15 Implantation parallèle d'un filtre FIR 128
4.16 Implantation parallèle d'un filtre FIR symétrique 129
4.17 Implantation d'un filtre FIR avec décomposition polyphase 130
4.18 Implantation directe du banc de filtres lifting 9/7 130
4.19 Implantation du banc de filtres lifting 9/7 avec partage d'opérateurs . 131
4.20 Implantation du banc de filtres lifting 5/3 avec lecture des échantillons deux par deux 131
4.21 Architecture réduite pour la transformation 1D multi-niveaux 133
4.22 Architecture rebouclée pour la transformation 1D multi-niveaux 134
4.23 Comparaison des besoins en mémoire des transformations horizontale et verticale pour un parcours de type raster 135

5.1 Graphe de dépendances de la DWT 2D Lifting sur trois niveaux pour l'ondelette 9/7 140
5.2 Anatomie d'un nœud lifting ou dual lifting 142
5.3 Nombre de bits des données dans un nœud de lifting ou dual lifting . . 143
5.4 Image test pour le calcul de PSNR : Lena 145
5.5 Mesure de PSNR pour différentes tailles de données intermédiaires . . 145
5.6 Graphe de dépendances causal de la DWT 2D Lifting pour l'ondelette 9/7 148
5.7 Motif de calcul pour l'ondelette 9/7 sur trois niveaux 150
5.8 Modèle HSFSMD du motif de transformation en ondelettes 2D 152
5.9 Modèle HSFSMD optimisé du motif de transformation en ondelettes 2D 153
5.10 Répartition de la charge de calcul au cours du temps pour différents degrés de parallélisme 154
5.11 Utilisation de boucles non déroulées dans le motif de décomposition en ondelettes 9/7 sur trois niveaux 157
5.12 Utilisation de composants combinatoires dédiés pour la décomposition en ondelettes 9/7 160
5.13 Insertion des contraintes d'entrée/sortie dans le motif de transformation en ondelettes 2D 162
5.14 Espace des solutions architecturales sous contraintes de vitesse de traitement et de surface 165
5.15 Vitesse de traitement maximale autorisée en fonction du parallélisme d'entrée/sortie 166
5.16 Surface de registres en fonction du parallélisme d'entrée/sortie 167
5.17 Nombre d'opérateurs arithmétiques en fonction de la vitesse de traitement 168

A.1 Le modèle d'architecture de l'outil GAUT 189
A.2 Le modèle d'architecture de l'outil Behavioral Compiler 193
A.3 Le flot de synthèse sous A|RT Designer 199

Liste des tableaux

1.1 Langages, styles de spécification, classes de systèmes et flot de conception 21
1.2 Langages, propriétés et modèles formels 21
1.3 Liste des groupes de travail (DWGs) de l'organisation VSIA 24

2.1 Classes de spécifications d'un composant virtuel de niveau comportemental 49
2.2 Algorithme d'interpolation : propriétés des trois motifs présentés . . . 61
2.3 Paramètres et contraintes relatifs à chaque niveau de spécification . . . 64
2.4 Propriétés relatives à chaque niveau de spécification 65

3.1 Déroulage forcé d'une boucle *for* 71
3.2 Deux exemples de réécriture des boucles non déroulées 73
3.3 Calcul de distance de Manhattan : caractéristiques des FSM produites par Monet et BC 75
3.4 Calcul de distance de Manhattan : propriétés des FSM générées par Monet et BC après optimisation manuelle de la description 76
3.5 Déroulage forcé des boucles non déroulables 93
3.6 Réécriture des boucles à ne pas dérouler 95
3.7 Distributivité des instructions conditionnelles 95
3.8 Choix d'écriture des instructions conditionnelles 96
3.9 Instruction de branchement dans une instruction conditionnelle 96
3.10 Boucles non déroulées dans les instructions conditionnelles 97
3.11 Modèles sémantiques associés à chaque niveau de spécification 99
3.12 Délivrables d'un composant virtuel comportemental 101

4.1 Cœfficients des bancs de filtres recommandés par la norme JPEG2000 117
4.2 Complexité des bancs de filtres recommandés par la norme JPEG2000 pour différents algorithmes de filtrage 120
4.3 Quantité de mémoire requise par les algorithmes de décomposition monodimensionnelle multi-niveaux 124
4.4 Quantité de mémoire requise par les algorithmes de décomposition bidimensionnelle multi-niveaux 127
4.5 Complexité architecturale de différentes implantations du banc de filtres 9/7 132
4.6 Performances en vitesse de différentes implantations du banc de filtres 9/7 132

5.1 Nombre d'opérations par pixel pour la transformation en ondelettes lifting 2D 141
5.2 Estimation de la durée minimale de calcul d'une donnée de sortie de la transformation en ondelettes lifting 2D 142
5.3 Propriétés du motif de transformation en ondelettes lifting 2D 149
5.4 Durée de vie des données et contraintes de mémorisation pour le motif de transformation en ondelettes lifting 2D 150
5.5 Parallélisme et rendement des ressources de calcul en fonction de la contrainte de temps 155
5.6 Nombre d'opérations à ordonnancer dans le corps des boucles non déroulées 156
5.7 Composants combinatoires dédiés et optimisations logiques 160
5.8 Nombre d'opérations à ordonnancer en fonction de la granularité des fonctions 161
5.9 Récapitulatif des paramètres du composant virtuel DWT 2D 164

A.1 Outils de synthèse de haut niveau commerciaux et académiques 186
A.2 Comparaison de trois outils de synthèse de haut niveau 201
A.3 Outils de synthèse de haut niveau universitaires et commerciaux : comparaison des styles de spécification 202

Introduction

Le marché des systèmes électroniques intégrés est aujourd'hui fortement orienté vers les applications grand public (*consumer electronics*), qu'elles soient destinées à l'activité professionnelle ou au loisir [110]. Là où le consommateur se voyait il y a quelques années proposer un ensemble de dispositifs répondant chacun à une classe bien identifiée de besoins – téléphonie, bureautique, télévision, vidéo, hi-fi, jeux, *etc.* –, les tendances actuelles s'orientent vers un regroupement des services dans des systèmes de plus en plus compacts [104].

Ces tendances se traduisent en particulier par une très forte orientation des applications vers les télécommunications [110] et les réseaux domestiques sans fil [104]. Après le succès d'Internet – combinant les capacités d'un réseau à l'échelle planétaire avec la richesse des applications multimedia – et de la téléphonie mobile, les systèmes électroniques mobiles de demain, dits de troisième génération (*3G*), combinent les fonctions d'un téléphone, d'un navigateur web, d'un agenda/organiseur personnel (*PDA* pour *Personal Digital Assistant*), d'un lecteur audio et vidéo, etc [106]. Le revenu mondial engendré par le marché des télécommunications mobiles de troisième génération est estimé à 320 milliards de dollars *US* à l'horizon 2010 [126].

Ces nouvelles applications étant destinées à faire interagir des terminaux portatifs sans fil avec des stations implantées soit au sol, soit à bord de satellites, la conception des systèmes doit faire face à des exigences de plus en plus contraignantes en termes d'implantation d'une part – miniaturisation des architectures matérielles, faible consommation, traitement temps réel – et de coût d'autre part. Ces contraintes sont d'autant plus difficiles à tenir que la complexité des applications augmente rapidement : l'*International Technology Roadmap for Semiconductors*, édition 2001 [18], prend pour exemple d'application un organiseur personnel (*PDA*) avec fonctionnalités multimedia, dont les performances, évaluées à 0,3 *GOPS* (giga-opérations par seconde) en 2001, sont estimées à 103 *GOPS* à l'horizon 2010, soit un accroissement moyen de 90% par an.

En parallèle à cette évolution, la croissance de la capacité et de la vitesse des technologies intégrées va dans le sens d'une concentration des fonctions sur des surfaces de silicium de plus en plus réduites. Conformément à la loi de Moore [19], qui stipule que le nombre de portes par unité de surface de silicium double tous les dix-huit mois, il est aujourd'hui possible d'intégrer sur un même substrat plusieurs dizaines de millions de portes, contre quelques centaines de milliers il y a cinq ans [18]. Il est donc aujourd'hui permis d'envisager l'intégration d'un système complet sur un seul *ASIC* ou *FPGA*, donnant naissance à la notion de système sur puce, ou *System-on-a-Chip* (*SoC*). Une extension de la loi de Moore [17] indique que, du fait de l'augmentation

conjointe de la fréquence d'horloge et du parallélisme de traitement, la puissance de calcul, en nombre d'instructions par seconde, double elle aussi tous les dix-huit mois. Parallèlement à cette évolution technologique, le coût moyen d'implantation par fonction n'a cessé de décroître avec une réduction d'environ 25 à 30% par an [18].

En revanche, les méthodes et outils de conception et de vérification couramment utilisés ne sont pas adaptés à gérer la complexité croissante des systèmes et de leur support physique. D'une part la capacité d'intégration réelle des concepteurs ne croît que de 32% par an [18], de telle sorte que le fossé entre capacité d'intégration théorique et pratique ne cesse de se creuser. L'augmentation du nombre d'ingénieurs par projet ne permettrait de réduire cet écart que de manière très limitée, tout en augmentant de manière significative les coûts de développement. D'autre part, la finesse de gravure des technologies sub-microniques profondes pose de nouveaux problèmes physiques qui ont un impact sur la complexité du processus de fabrication et sur la fiabilité des circuits [18]. Les coûts de génération des masques (coûts non récurrents ou *NRE* pour *Non-Recurrent Engineering costs*) et de fabrication d'un nouvel *ASIC* ont été multipliés par dix au cours de la dernière décennie [15, 18] il est plus que jamais indispensable de fiabiliser le flot de conception de manière à fournir à la fabrication une spécification exempte d'erreur.

L'industrie des semiconducteurs se trouve ainsi dans une véritable situation de crise [10, 15] qui cumule de nouvelles contraintes applicatives, architecturales et physiques, face auxquelles les flots de conception classiques trouvent leurs limites, avec une pression économique accrue résultant de l'évolution rapide des besoins qui impose des délais de mise sur le marché d'un nouveau produit de plus en plus brefs [15, 71].

De nouvelles méthodologies de conception sont nécessaires. Celles-ci doivent permettre de répartir efficacement la complexité des systèmes à concevoir sur les ressources humaines et matérielles par le biais d'une automatisation des tâches de conception les plus coûteuses [15, 53, 56, 57]. Une accélération du flot de conception est indispensable afin de conserver des délais raisonnables de mise sur le marché d'un produit et d'exploiter au mieux les nouvelles technologies au moment où elles sont disponibles [71]. Les techniques de vérification doivent être adaptées à la complexité croissante des systèmes de manière à fiabiliser le flot de conception et aboutir le plus tôt possible à une spécification correcte [10].

Les nouvelles approches envisagées reposent essentiellement sur une élévation du niveau d'abstraction de la spécification des systèmes, permettant de hiérarchiser les tâches de conception et de réutiliser de manière intensive des composants matériels et logiciels [15].

La conception au niveau système repose sur une démarche incrémentale de raffinement d'une spécification initiale exempte de tout détail d'implantation. Cette approche permet en particulier d'orthogonaliser les problèmes [15] : d'un point de vue fonctionnel en décomposant hiérarchiquement un système en sous-systèmes qui pourront être modélisés indépendamment à différents niveaux d'abstraction ; d'un point de vue comportemental en traitant indépendamment les aspects liés à la fonctionnalité d'un sous-système et ceux liés à la communication avec son environnement ; d'un point de vue architectural en permettant l'exploration de différentes solutions d'implantation reposant sur différentes stratégies de partitionnement matériel/logiciel. Les enjeux de la vérification de systèmes de forte complexité nécessitent la mise en place

de langages de modélisation [35–37] et d'outils de vérification capables de confronter les informations fournies par la spécification d'un système et de ses sous-systèmes à différents niveaux d'abstraction [15].

Dans la continuité de cette approche, une voie prometteuse pour l'accélération des tâches de conception consiste à réutiliser autant que faire se peut des blocs matériels ou logiciels déjà conçus et vérifiés [74]. L'apparition d'un marché des *composants virtuels* [112, 127], ou *Intellectual Property cores* (*IP*), entend répondre au besoin des concepteurs de concentrer leurs efforts sur la partie réellement innovante d'un système plutôt que sur la re-conception de composants déjà existants. En effet, bien qu'ils soient de plus en plus complexes, une part importante de ces composants – cœurs de processeurs, contrôleurs de bus, fonctions de traitement du signal, *etc.* – font désormais partie des briques de base de tout *SoC* [9]. La réutilisation de blocs *IP* matériels s'inscrit par conséquent dans le prolongement des développements qui ont permis l'élévation du niveau d'abstraction des primitives de description des architectures, d'abord du niveau électrique (transistors) au niveau portes logiques, puis au niveau transfert de registres. Une architecture à l'échelle du système est aujourd'hui représentée comme l'interconnexion de blocs fonctionnels réalisant des tâches complexes et communiquant entre eux par l'intermédiaire de protocoles élaborés [66] allant du simple *handshake* aux bus intégrés, avec en perspective la notion de réseau sur silicium (*NoC*) [67].

Aujourd'hui, la conception de ces composants *IP* repose cependant toujours sur les méthodes de conception et de synthèse au niveau transfert de registres. Pour l'implantation de fonctions de calcul intensif – de traitement du signal et de l'image, par exemple – cette approche apparaît de moins en moins adaptée à la complexité croissante des algorithmes d'une part, et d'autre part au degré élevé de flexibilité requis pour permettre leur réutilisation dans des contextes variés de besoins applicatifs, d'environnements systèmes, de contraintes de fonctionnement et d'implantation [77]. Répondre efficacement à cette nouvelle exigence de flexibilité nécessite l'adoption de techniques de génération automatiques d'architectures à partir de spécifications paramétrables de haut niveau. De telles techniques, dites de synthèse de haut niveau – autrement nommée synthèse comportementale ou synthèse d'architecture –ont fait l'objet de développements et d'expérimentations dans les laboratoires académiques depuis bientôt vingt ans [56, 59, 61]. Elles sont à présent implantées dans des outils de synthèse commerciaux distribués par des fournisseurs de *CAO* de premier plan tels *Mentor Graphics* [53] et *Synopsys* [57] mais leur adoption par les concepteurs d'*ASIC* reste très limitée [52]. Les réticences majeures sont essentiellement dues à un manque de méthodologie [2, 8] pour guider l'utilisation de ces nouveaux outils [52] : si la sémantique des spécifications au niveau transfert de registre et les règles d'écriture associées dans les langages de description de matériel *VHDL* et *Verilog* ont fait l'objet d'une standardisation [42, 43], la notion de *spécification comportementale* reste sujette à une variété de définitions – couvrant différents niveaux d'abstraction intermédiaires entre le niveau algorithmique et le niveau transfert de registres – selon les outils et les domaines d'applications considérées. Les concepteurs habitués à raisonner sur des modèles *RTL* perdent la maîtrise d'une partie des décisions d'implantation et doivent s'adapter aux nouveaux leviers fournis par les outils – certains explicites comme les contraintes de synthèse, d'autres implicites comme le choix du style d'écriture – sans pleinement mesurer l'impact de leurs actions sur la topologie, la complexité et la

vitesse de l'architecture générée [2].

Le travail que nous présentons dans ce mémoire a permis de définir une infrastructure méthodologique pour faciliter l'utilisation des outils de synthèse de haut niveau dans un contexte de conception pour la réutilisation. Nous avons en particulier étudié les étapes à introduire en amont de la synthèse de haut niveau afin qu'elle vienne s'insérer harmonieusement dans le flot de conception d'un système. Ce nouveau flot de conception repose sur une définition précise de ce que représente le niveau comportemental en termes de sémantique de simulation, de sémantique d'implantation et de restrictions syntaxiques à apporter aux langages de conception de matériel et/ou de systèmes afin de cibler efficacement les outils de synthèse de haut niveau en garantissant la prédictibilité des performances des architectures obtenues.

Cette démarche méthodologique s'inscrit dans le cadre du projet *RNRT MILPAT* (Méthodologie et Développement pour les Intellectual Properties pour Applications Télécom) [119], intégrant la notion de composant virtuel dans une perspective d'adéquation algorithme/architecture au moyen d'outils commerciaux de synthèse de haut niveau [77]. Elle est orientée vers les applications intégrant des fonctions de calcul intensif avec une prédilection pour les fonctions de traitement du signal et de l'image [1]. Conjointement, nos travaux en collaboration avec le *Centre National d'Etudes Spatiales* (*CNES*) et la société *Astrium* nous ont amené à étudier l'implantation de composants virtuels comportementaux pour la compression d'image embarquée à bord des satellites. A ce titre nous avons expérimenté notre démarche méthodologique sur la spécification d'un composant de transformation en ondelettes discrète pour la compression d'image selon le standard *JPEG2000* [3, 6, 7]. L'algorithme de transformation en ondelettes recommandé par le standard présente une forte complexité calculatoire et supporte un degré élevé de flexibilité, représenté par différentes options de la chaîne de compression. L'implantation d'un composant réutilisable supportant toutes ces options est une tâche non triviale qui en fait une cible privilégiée pour l'évaluation de notre méthodologie.

Organisation du mémoire

Dans la première partie de ce document, nous explorons les nouvelles tendances en conception des systèmes intégrés.

Le chapitre 1 nous permettra de dresser un tour d'horizon des méthodes, langages et outils qui permettent aujourd'hui d'élever le niveau d'abstraction pour la modélisation, la synthèse et la vérification des systèmes intégrés. Ce premier chapitre nous permettra de situer notre méthodologie dans un flot général de conception d'un système.

Dans le chapitre 2, nous poserons les bases de notre démarches méthodologique en exposant tout d'abord la problématique de l'insertion de la synthèse de haut niveau dans un flot de conception fondé sur la réutilisation de composants. Nous définirons ensuite les niveaux d'abstraction des vues successives d'un composant virtuel et les étapes-clés de la conception et de l'intégration d'un composant virtuel de niveau comportemental.

Le chapitre 3 s'attachera à formaliser un modèle de description au niveau comportemental permettant de rendre le flot de synthèse de haut niveau prédictible, indépen-

damment de l'outil de synthèse utilisé, en contraignant le modèle d'implantation cible.

La seconde partie de ce mémoire sera consacrée à l'expérience de conception et de synthèse d'un composant virtuel pour la transformation en ondelettes bidimensionnelle conforme au standard *JPEG2000*.

Le chapitre 4 introduira la problématique de la compression d'images embarquée, en mettant l'accent sur le contexte de l'étude, étroitement lié au domaine de l'imagerie spatiale. Nous décrirons dans ses grandes lignes la chaîne de compression proposée par le standard *JPEG2000* et nous nous focaliserons ensuite sur la transformation en ondelettes, d'un point de vue algorithmique en décrivant les méthodes usuelles de calcul de la transformée en ondelettes discrète et d'un point de vue architectural en discutant les stratégies d'implantation recensées dans la littérature pour la conception d'architectures *VLSI* de transformation en ondelettes.

Dans le chapitre 5, nous détaillerons l'application de notre méthodologie à la conception d'un composant virtuel flexible de niveau comportemental pour la transformation en ondelettes discrète bidimensionnelle. Nous présenterons des résultats de synthèse du composant avec l'outil de synthèse de haut niveau *Monet* de *Mentor Graphics*.

Première partie

Méthodologie de Conception de Composants Virtuels Réutilisables de Haut Niveau

Chapitre 1

Nouvelles Approches en Conception de Systèmes Intégrés

Par le passé, l'accroissement de la complexité des applications et de leur densité d'intégration a régulièrement nécessité de recourir à des modes de représentation de plus en plus abstraits et de plus en plus hiérarchisés des systèmes électroniques [9, 13, 15]. Dans un flot de conception descendant (*top-down* [71]), abstraction et hiérarchie autorisent le raffinement progressif de la spécification d'un système et la mise en œuvre des tâches de spécification et synthèse sur des blocs architecturaux dont la complexité reste accessible à une équipe d'ingénieurs de taille réduite [56]. La démarche complémentaire (*bottom-up* [71]) permet de vérifier un système de manière incrémentale en confrontant les propriétés de la spécification détaillée du système avec les besoins exprimés par les spécifications abstraites de plus haut niveau.

Aujourd'hui, la forte complexité des applications se traduit par un élargissement de l'espace des solutions architecturales [15, 23]. Parmi ces solutions, les possibilités de répartition des fonctions sur des ressources matérielles et logicielles sont nombreuses [11] et ont une forte incidence sur les performances du système. Afin d'obtenir dans les plus brefs délais une spécification fonctionnellement correcte et satisfaisant les contraintes de l'application, il est aujourd'hui indispensable de recourir à une démarche de conception dans laquelle les composants matériels et logiciels ne sont plus spécifiés, raffinés et vérifiés séparément, mais conjointement [9, 11, 23].

Cette nouvelle démarche a nécessité la mise en place de nouvelles méthodologies de conception s'intéressant tant au problème du partitionnement matériel/logiciel qu'à celui de la co-conception, co-synthèse et co-vérification des parties matérielles et logicielles d'un système [11]. Ces méthodologies reposent sur de nouveaux langages de spécification associés à des modèles formels permettant de faire interagir dans une même description exécutable d'un système des composants matériels et logiciels – voire numériques et analogiques – modélisés à différents niveaux d'abstraction [31, 35, 37, 49]. Les principes de la conception au niveau système, les modèles formels et les langages de conception sont présentés en section 1.1.

D'autre part, l'exploration architecturale et la spécification détaillée de l'architecture matérielle/logicielle d'un système font de plus en plus appel à des modèles prédéfinis de composants [71, 74]. Les composants matériels et logiciels réutilisables

– composants dits *virtuels*, ou blocs *IP* pour *Intellectual Property* – font aujourd'hui l'objet d'une importante activité commerciale [112,127] alors même que les méthodologies et standards [127, 128] sous-jacents n'ont pas encore atteint leur pleine maturité. Nous présentons en section 1.2 la problématique de la réutilisation de composants virtuels.

En complément de ces approches, la conception de composants matériels bénéficie de nouvelles techniques de synthèse tendant à l'automatisation des tâches de spécification détaillée d'une architecture *RTL* [53, 56, 57, 59, 61]. Les outils de synthèse de haut niveau – également dénommée synthèse *comportementale* ou synthèse d'architecture – commencent à voir le jour dans le domaine industriel [53, 57]. Le principe de la synthèse de haut niveau est présenté en section 1.3.

1.1 Maîtriser la complexité : la conception au niveau système

Au fil des étapes d'un flot de conception descendant, la description d'un système s'enrichit de manière incrémentale afin de refléter les décisions d'implantation successives prises par les concepteurs. Afin d'assurer un lien consistant entre les étapes successives d'un flot de conception et de vérification, il est nécessaire de définir formellement les différentes représentations possibles d'un système, tout d'abord en identifiant les niveaux d'abstraction sous lesquels il peut se présenter, puis en définissant les classes de propriétés et de détails d'implantation relevant de chaque niveau [12,13,56].

L'automatisation totale ou partielle des tâches de conception – telles le partitionnement matériel / logiciel, le raffinement, la vérification et la synthèse d'une architecture – nécessite la mise en place de représentations formelles du fonctionnement, de la structure et des contraintes d'implantation d'un système [15, 23, 27].

1.1.1 Domaines de conception et niveaux d'abstraction

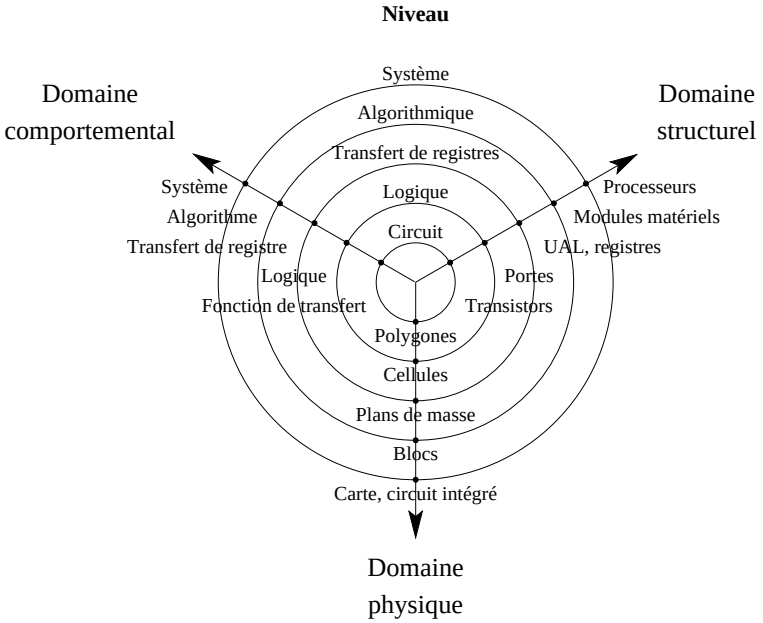
Une méthodologie de conception de systèmes doit tout d'abord s'appuyer sur une terminologie commune aux différents acteurs de la conception pour les différentes vues et niveaux d'abstraction sous lesquels un système peut se présenter [14, 16]. La définition d'une telle terminologie obéit au besoin d'assurer une interopérabilité au sens large des différentes étapes de la conception en facilitant la communication entre des ingénieurs relevant de différents domaines de compétences et pouvant être amenés à travailler ensemble sur des projets de grande envergure [14].

Le diagramme en *Y* [12] et la taxonomie *RASSP/VSIA* [14,16] que nous présentons ci-après répondent à ce souci de classifier les représentations d'un système et de fournir un support conceptuel pour l'élaboration de nouvelles méthodologies de conception.

Le diagramme en Y

Le diagramme en *Y* (Y-Chart), introduit par GAJSKI et KUHN [12], repose sur l'observation que la spécification d'un système combine des informations relatives à trois principaux domaines de représentation :

FIG. 1.1 – Le diagramme en Y de Gajski et Kuhn



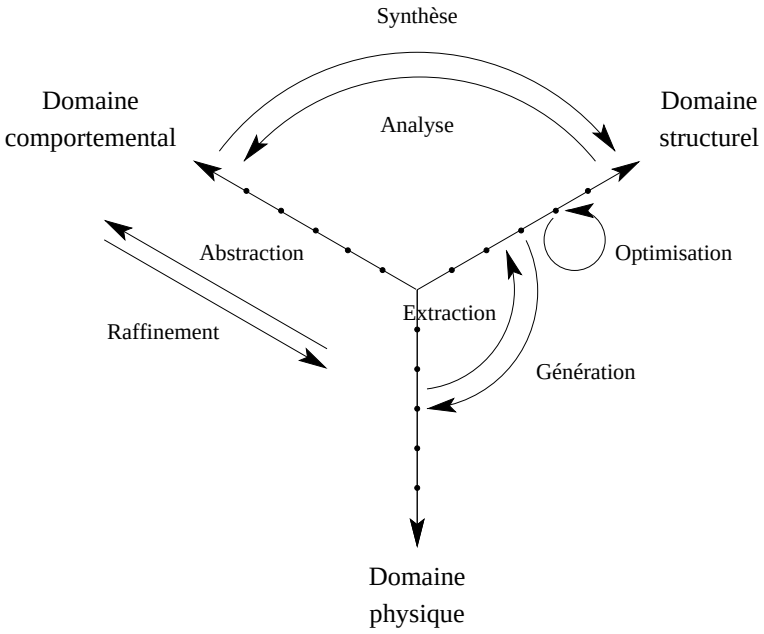
- Dans le domaine *comportemental*, un concepteur s'attache à décrire le fonctionnement d'un système ou de ses constituants, indépendamment des ressources matérielles que le constituent.
- Dans le domaine *structurel*, un système est décomposé en sous-systèmes, ou composants, interconnectés qui réalisent chacun une tâche élémentaire.
- Dans le domaine *physique*, l'implantation d'un système sur un support physique est exprimée en termes géométriques.

Dans chacun de ces domaines, un système peut être décrit avec une plus ou moins grande richesse de détails. Par exemple, lors de la spécification initiale d'une application, une fonction mathématique ou un algorithme suffit à décrire le comportement souhaité du système. En revanche, l'élaboration du produit fini nécessitera de modéliser précisément le comportement électrique du circuit afin de prendre en compte les contraintes de consommation. Le diagramme en Y propose ainsi une de représenter chaque domaine par un axe gradué en cinq niveaux d'abstraction (figure 1.1) : (1) niveau système ; (2) niveau algorithmique ; (3) niveau transfert de registres ; (4) niveau logique ; (5) niveau circuit.

Chaque étape de la conception d'un système se traduit par un changement de son mode de représentation. Ce changement peut se modéliser par des transitions d'un domaine à un autre ou d'un niveau d'abstraction à un autre dans le diagramme en Y. Les transitions typiques dans le diagramme en Y sont les suivantes 1.2 :

Le long d'un axe donné, le *raffinement* consiste à enrichir les détails propres à un domaine de représentation du circuit en descendant dans les niveaux d'abstraction.

FIG. 1.2 – Transitions dans le diagramme en Y



L'*abstraction* consiste au contraire à condenser ces détails de manière à en reconstituer la signification abstraite.

Les transitions de l'axe *comportement* vers l'axe *structure* correspondent à la projection de la description du fonctionnement du circuit sur une architecture, ce qui correspond à une opération de *synthèse*. La transition inverse correspond à une *analyse* de l'architecture en vue d'en reconstituer le fonctionnement à des fins de vérification formelle.

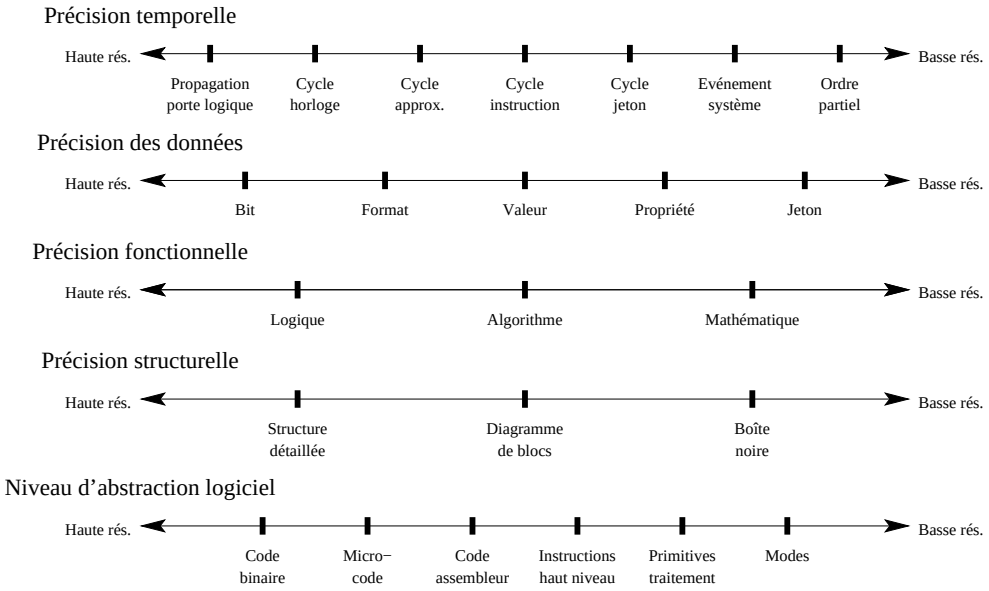
A un niveau d'abstraction donné, il est possible de fournir plusieurs représentations fonctionnellement équivalentes d'un système. L'*optimisation* consiste à transformer progressivement une spécification de manière à satisfaire une contrainte.

La *génération* correspond à une *synthèse physique* d'une architecture, ce qui revient à fournir pour une représentation structurelle donnée une représentation géométrique de son implantation sur une cible technologique.

La taxonomie RASSP/VSIA

Les membres du groupe de travail *RTWG* (*RASSP Terminology Working Group*) ont observé que les trois axes du diagramme en Y restent limités aux aspects matériels d'un système et sont donc insuffisants dans le cadre d'une démarche de co-design matériel/logiciel [14]. La taxonomie qu'ils proposent repose sur cinq axes dont les quatre premiers sont dupliqués afin de représenter séparément la vue *interne* et la vue *externe* (interface) d'un système ou d'un composant. Chaque axe est gradué selon une échelle de résolution, ou niveaux d'abstraction (figure 1.3). Les hautes résolutions (à gauche) fournissent une caractérisation fine des détails d'implantation et correspondent par

FIG. 1.3 – La taxonomie RASSP/VSIA

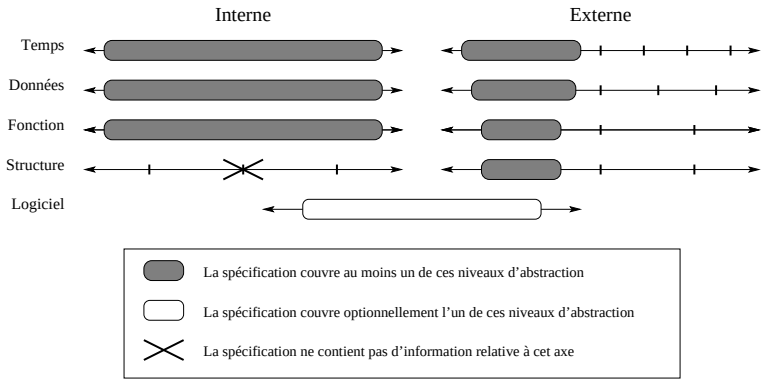


conséquent aux plus bas niveaux d'abstraction :

- L'axe *précision temporelle* représente le degré de précision avec laquelle sont exprimés les propriétés temporelles des événements se produisant au cours du fonctionnement du système (sept graduations allant de la relation d'ordre partiel entre événements jusqu'au temps de propagation d'une porte logique).
- L'axe *précision des données* représente le niveau d'abstraction des données manipulées par le système ou échangées avec son environnement (cinq graduations, d'une représentation à base de *jetons* abstraits à une représentation au bit près).
- L'axe *précision fonctionnelle* représentent le niveau d'abstraction selon lequel les fonctions et opérations réalisées par le système et ses sous-systèmes sont modélisées (trois graduations, du niveau mathématique au niveau logique).
- L'axe *précision structurelle* représente le degré de profondeur selon lequel le circuit est décomposé hiérarchiquement en sous-blocs (trois graduations, d'une vue de type *boîte noire* à la description détaillée de la structure).
- L'axe *niveau d'abstraction logiciel* concerne la finesse des détails avec lesquels le code exécuté par un composant programmable est décrit (six graduations, d'une description en termes de *modes* de fonctionnement à la représentation binaire du code exécutable).

A titre d'exemple, la figure 1.4 représente les attributs d'une spécification comportementale détaillée d'un composant matériel [16] : les détails d'implantation interne peuvent couvrir tous les niveaux d'abstraction suivant les axes *précision temporelle*, *précision des données* et *précision fonctionnelle*, mais ne fournissent aucune information structurelle ; la vue externe est spécifiée à un niveau d'abstraction relativement bas qui permet l'interfaçage du composant avec un environnement décrit au niveau

FIG. 1.4 – Spécification comportementale détaillée dans la taxonomie RASSP/VSIA



transfert de registres ; le composant peut éventuellement être programmable à un niveau d'abstraction arbitraire.

Le groupe de travail *System-Level Design* (*SLD*) de l'organisation *VSIA* (*Virtual Socket Interface Alliance*) – dont nous aurons l'occasion de reparler dans la section 1.2 – a retenu la taxonomie *RASSP*, moyennant quelques adaptations, pour l'appliquer au domaine de la conception de systèmes sur silicium à partir de composants virtuels réutilisables [16].

1.1.2 Propriétés d'un système

Selon le niveau d'abstraction et le domaine de conception (comportemental/structurel) auquel on le considère, un système sera susceptible d'être raffiné de manière hiérarchique en sous-systèmes ou tâches concurrentes nécessitant la mise en place de mécanismes de synchronisation et de communication [13,56]. Ainsi, une représentation d'un système devra reposer sur des mécanismes bien définis modélisant les propriétés suivantes [80] :

- la classe d'informations manipulées par le système ;
- la *réactivité* du système ou de ses constituants vis-à-vis de leur environnement ;
- la *hiérarchie* des composants ou tâches constituant un système ;
- la *concurrence* du fonctionnement des sous-systèmes ;
- la *synchronisation* et la *communication* entre sous-systèmes ;
- la précision avec laquelle la notion de *temps* est modélisée.

Classe d'informations traitées – Un système peut être dominé par les données ou par le contrôle [23] : un système dominé par les données sera décrit en termes de fonctions réalisant un traitement de type calculatoire sur des ensembles de valeurs. Dans un système dominé par le contrôle, c'est l'évolution de l'état du système au cours du temps qui est décrite en fonction de l'apparition d'événements ponctuels.

La spécification abstraite d'un système dominé par les données s'attachera ainsi à décrire la *relation mathématique* entre les valeurs des données d'entrée et de sortie.

Un système dominé par le contrôle sera décrit comme une *relation temporelle* entre événements d'entrée et de sortie.

Une part importante des systèmes nécessite une modélisation mixte où des fonctions orientées traitement de données – traitement de signaux audio ou vidéo, par exemple – doivent interagir en temps réel avec leur environnement.

Réactivité – La réactivité caractérise le mode d'interaction d'un système avec son environnement. On distingue en général trois types de systèmes [32] :

- Un système *transformationnel* réalise une tâche dont le début et la fin sont nettement identifiés. Un tel système consomme un ensemble de données d'entrée, traite ces données, produit un ensemble de données de sortie, et s'arrête. Aucune information n'est mémorisée entre deux exécutions successives.
- Les systèmes *interactifs* sont typiquement conçus pour intervenir dans des applications client/serveur. Ils reçoivent des demandes de services de leur environnement et traitent ces demandes selon la disponibilité des ressources. Les interactions entre un tel système et son environnement sont contraintes par la vitesse propre du système.
- Un système *réactif* [20, 23] est constamment à l'écoute de son environnement. Il réagit aux stimuli émis par l'environnement en émettant d'autres stimuli. L'activité d'un tel système est contrainte par la vitesse de l'environnement. Ce type de contrainte se rencontre principalement dans les applications temps-réel.

Hiérarchie – La hiérarchie permet de gérer la complexité d'un système en le décomposant en sous-systèmes présentant chacun une unité de fonctionnalité. La hiérarchie peut se présenter sous deux formes [13, 45] : (1) la hiérarchie *comportementale* – *i.e.* relative au *domaine* comportemental du diagramme en Y – permet de représenter une tâche comme la composition d'un ensemble de sous-tâches s'exécutant en séquence ou en parallèle ; (2) la hiérarchie *structurelle* a pour principe la représentation d'un système sous forme de composants interconnectés, chaque composant pouvant à son tour être décomposé structurellement.

Concurrence – Cette propriété exprime la faculté à décomposer la spécification en plusieurs tâches s'exécutant en parallèle. La concurrence peut être considérée à différents niveaux [13] : (1) au niveau tâche ou processus, (2) au niveau instruction, (3) au niveau opération.

Communication – Les échanges de données entre tâches concurrentes nécessitent la définition d'un modèle de communication afin de lever toute ambiguïté concernant les ordres relatifs entre données et traitements. Des processus concurrents peuvent échanger des données par deux méthodes [13, 45] : (1) par diffusion (*broadcast*) à l'ensemble des processus par l'intermédiaire d'une ressource partagée ; (2) par passage de messages à travers des canaux abstraits.

La diffusion nécessite la mise en place de mécanismes de gestion des accès aux ressources en écriture. La communication par messages autorise une plus grande variété de modèles de communication : un canal se caractérise par son caractère uni- ou

bidirectionnel ; par le nombre de connexions qu'il autorise (point à point, multivoie) ; par ses contraintes d'accès en lecture et en écriture (bloquant, non bloquant).

Synchronisation – La synchronisation est requise dans deux cas : (1) lorsque des informations sont échangées entre différents processus, (2) lorsque l'exécution de deux tâches doit être déclenchée simultanément ou en séquence.

Dans les systèmes dominés par le contrôle, les systèmes réactifs [20] font largement intervenir la notion d'*événement* : un événement peut correspondre, par exemple, à l'émission/réception d'un jeton à un instant donné, ou au changement de la valeur d'une donnée. Dans les systèmes dominés par les données, la synchronisation peut être imposée par le mode d'accès bloquant/non bloquant à un canal de communication [13].

La définition des ordres relatifs de déclenchement de deux tâches peut être explicite, c'est-à-dire modélisée par des émissions et attentes d'événements correspondant au début ou à la fin de chaque processus, soit implicite au moyen de primitives de contrôle de type *fork* – décomposition du flot d'exécution en plusieurs sous-tâches concurrentes – et *join* – attente de la terminaison d'un ensemble de tâches concurrentes [13].

Granularité temporelle – L'expression de la durée d'exécution d'une tâche est étroitement liée à son implantation finale. Elle peut en particulier servir de contrainte pour le flot de synthèse. L'expression du temps peut être envisagée à différents niveaux d'abstraction [45] : (1) temps *physique* : en nanosecondes ; (2) temps *logique* : en périodes d'horloges ; (3) temps *comportemental* : exprimé comme le nombre d'itérations d'un processus ; (4) temps *système* : relation d'ordre partiel entre événements.

Les applications de traitement du signal ou d'automatique intègrent des processus de traitement se répétant à l'infini et parfois récursifs. La notion de délai unitaire (opérateur z^{-1}) est une contrainte importante à prendre en compte dans les systèmes temps réel (au sens de la cadence des traitements).

1.1.3 Modèles formels

Un modèle sémantique formalise un ensemble de notions permettant de modéliser le comportement et/ou la structure d'un système en exploitant un certain nombre de propriétés [23,80]. En général, un système concurrent est vu comme une interconnexion d'*acteurs* échangeant des informations sous forme de données ou d'événements [27]. Chaque modèle repose sur une définition spécifique de la notion de temps [13]. Dans le domaine des systèmes numériques, nous retiendrons les suivants :

Les modèles à temps discret (DT) – Comme leur nom l'indique, ils reposent sur une discrétisation de l'axe temporel. Ces modèles sont particulièrement adaptés à la description de fonctions de traitement de données – traitement numérique du signal –, où les entrées/sorties sont lues et mises à jour à des instants définis par une loi d'échantillonnage.

De telles fonctions sont décrites sous la forme d'équations aux différences et peuvent être représentées graphiquement sous la forme de graphes flots de signaux (*SFG*). L'unité de temps, ou délai unitaire, correspond à l'intervalle séparant deux points d'échantillonnage successifs des données (temps logique).

Les modèles à événements discrets (DE) – Dans un modèle à événements discrets, la synchronisation des acteurs est réalisée au moyen d'événements apparaissant de manière ponctuelle le long d'un axe temporel généralement continu [21, 27].

Un acteur qui détecte un événement réagit en changeant d'état et/ou en émettant d'autres événements [23]. Les événements sont traités dans l'ordre chronologique.

Les modèles synchrones/réactifs (SR) – Les modèles synchrones/réactifs assurent deux propriétés [20, 23, 27] : (1) *réactivité* : un acteur réagit à la détection d'événements à ses entrées en changeant d'état et/ou en émettant d'autres événements. (2) *synchronisme* : la réaction est instantanée, ainsi l'échantillonnage de l'axe temporel est entièrement déterminé par la date des événements d'entrée du système.

Les processus séquentiels communicants (CSP) – Le modèle *CSP* [21, 26] repose sur des processus interconnectés au moyen de canaux unidirectionnels à lecture et écriture bloquantes [23]. Ce type de de canaux impose une synchronisation par *rendez-vous*, c'est-à-dire qu'un échange de données ne peut s'effectuer que lorsqu'émetteur et récepteur sont tous les deux dans un état propice à la communication.

Les réseaux de processus de Kahn (PN) – Dans un réseau de processus de KAHN [21], les acteurs sont des processus interconnectés communiquant par l'intermédiaire de canaux à lecture bloquante et écriture non bloquante. Ces canaux peuvent être modélisés par des *FIFO* de taille infinie qui imposent aux événements ou données produits par un processus de respecter la même chronologie que les événements ou données dont ils sont la réaction.

Les flots de données – Les modèles de type flot de données sont un cas particulier des réseau de processus de KAHN. Dans ces modèles, les acteurs réalisent des tâches calculatoires sur des séquences de données et produisent en réaction d'autres séquences de données [28]. Au cours du temps, un processus ne peut être déclenché que lorsqu'une quantité suffisante de données est présente à ses entrées. Les données nécessaires au traitement à l'instant considéré sont alors consommées et transformées en une séquence de données de sortie. Les règles de déclenchement (*firing*) du modèle définissent à chaque instant la quantité de données nécessaires à un processus et la quantité de données produites lors d'une passe élémentaire de traitement.

Par exemple, dans le modèle *SDF* (flot de données synchrones), chaque acteur consomme et produit des données par paquets de taille constante [27]. Dans le modèle *CSDF* (flot de données cyclo-statique), la quantité de données consommées et produites à chaque déclenchement d'un acteur varie de manière périodique au cours du temps.

Un système obéissant à un modèle flot de données peut être schématisé par un graphe flot de donnée (*DFG*) qui exprime chaque tâche ou opération sous forme d'un nœud, et matérialise les dépendances de données entre tâches par des arcs entre ces nœuds [23]. Ainsi, un *DFG* permet de mettre en évidence, par analyse des dépendances, les tâches pouvant être exécutées en parallèle.

Les *DFG* sont particulièrement bien adaptés à l'optimisation d'une spécification en termes de vitesse ou de quantité de ressources, sur la base de différentes stratégies d'ordonnancement. La définition de *DFG* hiérarchiques, reposant sur la possibilité de décomposer chaque nœud en un sous-*DFG*, a été exploitée dans [29] en tant que format intermédiaire de représentation d'algorithmes de calcul en vue de l'exploration architecturale.

Graphes flot de contrôle et de données (CDFG) – Un graphe flot de données et de contrôle est la combinaison d'un graphe flot de contrôle (*CFG*) avec un ensemble de *DFG* [56]. Le *CFG* modélise les dépendances de contrôle entre les différents *DFG* au moyen de nœuds spécifiques dont la sémantique est analogue aux structures de contrôles rencontrées dans les langages de programmation impératifs (mise en séquence, exécution conditionnelle, boucles).

Les méthodologies d'exploration architecturale à partir de spécifications algorithmiques, rédigées dans des langages impératifs comme le langage *C*, font un usage intensif de modèles de type *CDFG*. Une telle modélisation autorise l'ordonnancement des tâches en fonction de leurs dépendances de données et de contrôle.

Machines à état fini (FSM) – Une *FSM* modélise un comportement séquentiel en termes d'états que peut prendre le système et de transitions entre ces états. Ce caractère séquentiel limite l'aptitude des *FSM* à modéliser des systèmes complexes [23]. Des extensions du modèle ont été proposées afin d'y inclure les propriétés de hiérarchie et de concurrence : (1) les *CFSM* (*Codesign Finite State Machines* [22]) sont utilisées pour modéliser des systèmes mixtes matériel/logiciel sous la forme d'un ensemble de *FSM* fonctionnant en concurrence ; (2) le formalisme *StateCharts* [25] de Harel inclut également la notion de hiérarchie en permettant la décomposition d'un état d'une *FSM* en sous-états "et" – *i.e.* concurrents – et en sous-états "ou", raffinant un état sous la forme d'une *FSM* de plus bas niveau hiérarchique.

L'introduction de la concurrence nécessite la mise en place d'un modèle de communication et de synchronisation entre les *FSM* fonctionnant en parallèle. Dans les modèles *CFSM* et *StateCharts*, la communication s'effectue par diffusion d'événements et réaction instantanée.

Machines à état fini avec chemin de données (FSMD) – A l'instar des *CDFG*, une *FSMD* permet d'associer les propriétés d'un modèle orienté contrôle (*FSM*) à celles d'un modèle orienté données (*DFG*). Ainsi, une *FSMD* est construite à partir d'une *FSM* classique associée à un ensemble de variables du chemin de données et d'expressions mathématiques modélisant l'évolution de la valeur de ces variables en fonction de l'état courant et des entrées [58]. La définition formelle des *FSMD* est détaillée en annexe C.

Le modèle *FSMD* est particulièrement bien adapté à la représentation d'architectures matérielles au niveau transfert de registres, vues comme l'interconnexion d'une *FSM* de contrôle avec un chemin de données [30]. Les modèles de type *SFSMD* (*Superstate FSMD*) autorisent une modélisation des architectures matérielles à un niveau d'abstraction plus élevé où la notion d'état est indépendante de la notion d'horloge et

où les calculs associés à chaque état ou transition peuvent être arbitrairement complexes. L'utilisation des *SFSMD* est étudiée dans [58] pour la synthèse de haut niveau.

Résumé

Chacun des modèles que nous venons de présenter autorise la prise en compte d'un ensemble limité de propriétés d'un système. L'évolution des applications vers des systèmes intégrés fortement hétérogènes nécessite ainsi de pouvoir combiner dans une même spécification une variété de modèles de représentation, permettant de représenter les interactions entre des composants numériques et analogiques, matériels et logiciels, dominés par les données et par le contrôle. Ces préoccupations sont à la base du projet *Ptolemy* de l'université de Berkeley et de l'élaboration du langage *Rosetta* [31].

1.1.4 Langages de conception de systèmes

La spécification d'un système nécessite le choix d'un format ou langage de description. La syntaxe d'un tel langage doit être interprétable tant par les concepteurs humains que par les outils de *CAO*. Sa sémantique doit se plier à un modèle formel en adéquation avec les propriétés du système que le concepteur souhaite exprimer.

Dans la pratique courante, les langages utilisés lors de la conception d'un système sont de deux types : (1) les langages de programmation (*C/C++*) sont utilisés pour spécifier les parties logicielles d'une application et pour élaborer des modèles algorithmiques de référence pour les composants matériels ; (2) les langages de description de matériel (*VHDL*, *Verilog*) sont utilisés pour élaborer des modèles de simulation au cycle près, puis des modèles synthétisables au niveau transfert de registres des composants matériels.

Si la syntaxe des langages de conception de matériel est largement inspirée de celle des langages de programmation, les deux types de formalismes ont cependant évolué de manières divergentes afin de prendre en compte les caractéristiques spécifiques de leurs cibles technologiques. Ainsi, les langages de programmation comme les langages de description de matériel ne supportent qu'un sous-ensemble restreint des propriétés nécessaires à la modélisation d'un système complet [27, 45]. Le développement récent des langages de spécification de systèmes s'est efforcé d'unifier les deux types de représentation et de fournir un support pour la modélisation à un plus haut niveau d'abstraction, c'est-à-dire indépendamment des choix d'implantation relatifs aux différents sous-systèmes et à leur infrastructure de communication [27, 37].

Une des principales exigences soulevées quant à la définition des langages de description de systèmes est l'aptitude à orthogonaliser les différentes aspects de la conception d'un système [15, 27], de manière à permettre l'exploration efficace d'une variété d'alternatives d'implantation : (1) la *fonctionnalité* (ce que fait le système), et l'*architecture* (comment il le fait) ; (2) les *calculs*, et la *communication* ; (3) les *besoins/contraintes* et les *choix d'implantation*.

Fonctionnalité et architecture – Les langages de conception de système comme *SystemC* [47], *SpecC* [35] et *Superlog* [34] permettent de spécifier différentes versions interchangeables d'un même composant à différents niveaux d'abstraction.

Au plus haut niveau, la fonction réalisée par un composant peut être décrite sous la forme d'un algorithme – niveau *Untimed Functional (UTF)* de la méthodologie *SystemC* [38]. Au fil des étapes de raffinement d'un composant, celui-ci peut être décomposé hiérarchiquement en processus ou composants interconnectés jusqu'à atteindre, dans le cas d'une implantation matérielle, une *netlist* au niveau portes logiques.

Calculs et communication – Le langage *SystemC*, à partir de la version 2.0 [47], et le langage *SpecC* [35] autorisent la spécification séparée d'interfaces et de canaux de communication entre composants.

Les communications peuvent être modélisées à différents niveaux d'abstraction, d'une description à base de *FIFO* de taille infinie – communications au niveau transaction ou *TLM* (*Transaction-Level Model*) [44] – à la spécification détaillée d'un protocole de communication – niveaux *Bus Cycle-Accurate (BCA)* et *Cycle-Accurate* [46].

Besoins/contraintes et choix d'implantation – Une spécification formelle des besoins et contraintes indépendamment des aspects architecturaux est nécessaire à la vérification de systèmes complexes. Les langages *Superlog* [34], *Sugar* [33] et *OpenVera* [48] autorisent l'écriture d'assertions modélisant des relations fonctionnelles et temporelles entre les signaux d'une architecture. Le langage *Rosetta* [31] travaille à un niveau d'abstraction plus élevé et permet de spécifier des besoins et contraintes relatifs à une variété d'autres domaines : électrique, thermique, mécanique, chimique, *etc.*

Bilan : Langages, modèles formels et flot de conception

Les tableaux 1.1 et 1.2 résument les caractéristiques des principaux langages utilisés pour la spécification, la synthèse et la vérification de systèmes. Le tableau 1.1 s'intéresse plus particulièrement au style de spécification, aux classes de systèmes et aux étapes du flot de conception concernés par chaque langage. Le tableau 1.2 décrit plus en détail les propriétés sémantiques de ces langages et les relie aux modèles formels présentés en section 1.1.3.

1.2 Accélérer le flot de conception : la réutilisation de composants virtuels

Du fait du large éventail de fonctionnalités supportées par les nouvelles applications embarquées [110] et de la forte hétérogénéité des constituants d'un *SoC*, la conception d'un système intégré se trouve aujourd'hui à la jonction de domaines de compétences variés qu'il est difficile de réunir dans une équipe unique [71].

Face à la complexité croissante des applications, l'effort nécessaire à la conception détaillée, à l'optimisation et à la vérification de tous les composants d'un système ne permet plus d'atteindre des compromis temps/coûts de développement acceptables pour des délais de mise sur le marché toujours plus courts [71].

La démarche de rationalisation du flot de conception, vérification et synthèse engagée ces dernières années repose sur la constatation qu'un grand nombre de fonctionnalités sont communes à la plupart des applications et ont déjà été implantés de

TAB. 1.1 – Langages, styles de spécification, classes de systèmes et flot de conception

Langage	Syntaxe	Style	Aspects dominants	Cible	Flot
C	Textuelle	Impératif	Contrôle+Données	Logiciel	Synthèse
C++	Textuelle	Impératif Objet	Contrôle +Données	Logiciel	Synthèse
UML	Graphique	Objet	Contrôle	Logiciel	Spécification
VHDL	Textuelle	Impératif	Contrôle+Données	Matériel	Synthèse
Verilog	Textuelle	Impératif	Contrôle+Données	Matériel	Synthèse
SpecC	Textuelle	Impératif	Contrôle+Données	Système	Spécification
SystemC	Textuelle	Impératif Objet	Contrôle +Données	Système	Spécification
Superlog	Textuelle	Impératif	Contrôle+Données	Système	Spécification
StateCharts	Graphique	–	Contrôle	Système	Spécification
Esterel	Textuelle	Impératif	Contrôle	Système	Spécification
ECL	Textuelle	Impératif	Contrôle+Données	Système	Spécification
SDL	Graph./Text.	Impératif	Contrôle	Système	Spécification
Rosetta	Textuelle	Déclaratif	Contrôle+Données	Système	Vérification
Sugar	Textuelle	Déclaratif	Contrôle	Système	Vérification

TAB. 1.2 – Langages, propriétés et modèles formels

Langage	Hiérarchie	Concurrence	Comm.	Synch.	Modèles
C / C++	Comportementale	Processus	Diffusion Messages	OS	PN, CSP, CDFG
VHDL / Verilog	Comportementale Structurelle	Instruction Processus	Diffusion	Evénements	DE, FSMD
SpecC / SystemC	Comportementale Structurelle	Processus	Diffusion Messages	Evénements Contrôle	DE, CSP, PN CDFG, FSMD
Superlog	Comportementale Structurelle	Instruction Processus	Diffusion Messages	Evénements Contrôle	DE, CSP, PN CDFG, FSMD
StateCharts	Comportementale	Etat	Diffusion	Evénements Contrôle	HCFSM
Esterel	Comportementale Structurelle	Instruction Processus	Diffusion	Evénements Contrôle	SR, FSM
ECL	Comportementale Structurelle	Instruction Processus	Diffusion	Evénements Contrôle	SR, FSM, CDFG
SDL	Comportementale Structurelle	Instruction Processus	Messages	Accès Contrôle	CSP, FSM

nombreuses fois en matériel ou en logiciel [74]. Le flot de conception d'un système intégré peut ainsi être accéléré de manière significative en réutilisant des blocs matériels ou logiciels déjà conçus et vérifiés plutôt que de les re-concevoir. Un tel composant réutilisable est appelé "composant virtuel" [74], ou plus communément "bloc *IP*" pour *Intellectual Property*, la notion de *propriété intellectuelle* étant liée au fait que les concepteurs d'un tel composant sont *a priori* distincts de l'équipe chargée de l'intégration. L'objectif de la notion d'*IP* consiste à appliquer au matériel les principes de conception que le génie logiciel met en œuvre avec succès depuis des années sous forme de *composants logiciels* réutilisables.

1.2.1 Définition et problématique

Ainsi, nous voyons se dessiner une première définition en quatre points de la notion de composant virtuel réutilisable [74] :

- Un composant virtuel contient la description d'une architecture matérielle ou logicielle, projetable directement sur une cible technologique au moyen d'outils de synthèse/compilation automatiques.
- Un composant virtuel réalise une fonction non triviale, mais à usage suffisamment général pour être réinvestie dans un grand nombre d'applications.
- Un composant virtuel doit pouvoir être intégré à un système sans la présence du concepteur de ce composant. Il doit par conséquent être fiable et aisément réplicable à l'aide des outils d'intégration du commerce.
- Le coût de la réutilisation d'un composant virtuel doit être nettement inférieur à celui de sa re-conception.

Loin de se limiter à un fichier de description synthétisable ou compilable, un composant virtuel se définit ainsi comme l'ensemble des informations qui permettront la réutilisation d'un bloc matériel ou logiciel pré-conçu et pré-vérifié [71,74]. La réutilisation de composant virtuels s'inscrit simultanément dans deux perspectives complémentaires d'accélération du flot de conception d'un système d'une part, et de fiabilisation de ce flot d'autre part.

1.2.2 Acteurs de la conception et de la réutilisation

Un composant virtuel fait intervenir quatre acteurs principaux [69] : (1) le *créateur*, ou *concepteur*, du composant est l'*auteur* des différents fichiers – délivrables – qui le constituent ; (2) le *fournisseur* est une entreprise ayant un lien direct ou indirect avec le créateur, qui met le composant virtuel à son catalogue et se charge de sa commercialisation ; (3) l'*outilleur* fournit les logiciels de *CAO* nécessaires à la conception, à la vérification, à la documentation, à la sélection et à l'intégration de composants virtuels ; (4) l'*utilisateur*, ou *intégrateur* d'un composant virtuel est le client final qui sélectionne les composants correspondants à ses besoins, en fait l'acquisition et les insère dans le système en cours de conception.

Concepteur et fournisseur d'un composant s'engagent à mettre sur le marché une description fonctionnellement correcte, synthétisable – éprouvée sur au moins une cible technologique à l'aide des outils les plus répandus – et munie de toutes les fichiers et documents nécessaires à son implantation dans un système.

Les outilleurs garantissent une propriété d'interopérabilité entre les flots de conception et de réutilisation. Ils s'attachent à respecter des formats de données et des langages de spécification standards [42, 43], laissant à l'utilisateur le choix des outils et des cibles technologiques (*ASIC*, *FPGA*).

1.2.3 Propriétés d'un composant virtuel et "réutilisabilité"

La conception et la réutilisation de composants virtuels posent un certain nombre de difficultés d'ordre méthodologique, le premier problème étant de définir clairement ce que l'on entend par "réutilisable" [69]. L'aptitude à intégrer un composant virtuel dans un système repose sur trois facteurs : (1) l'*interopérabilité* des formats de représentation des informations qui le constituent ; (2) sa *flexibilité*, autrement dit l'aptitude de l'utilisateur à adapter la fonctionnalité et les performances d'un composant à ses besoins propres ; (3) sa *prédictibilité*, c'est-à-dire la précision avec laquelle l'utilisateur peut estimer les performances d'un composant avant qu'il n'ait été intégré au système cible.

L'interopérabilité est étroitement liée aux standards régissant les niveaux d'abstraction des descriptions synthétisables et simulables d'un composant, les formats de données et langages de description, la liste minimale des délivrables pour garantir l'aptitude à réutiliser un composant en l'absence du concepteur. Le principal acteur en matière de recommandation et de standardisation concernant les composants virtuel est l'organisation *VSIA* (*Virtual Socket Interface Alliance*) [128], formée en 1996 dans le but de promouvoir la conception de systèmes intégrés par la réutilisation de composants provenant de sources variées. *VSIA* réunit essentiellement des entreprises industrielles, concepteurs de circuits et vendeurs d'outils de *CAO*, et se décompose en onze groupes de travail dont la liste est donnée dans le tableau 1.3.

Les missions de ces groupes couvrent les objectifs suivants :

- Définir un vocabulaire commun pour les niveaux d'abstractions, les modèles de représentation et les délivrables d'un composants virtuel.
- Standardiser des méthodologies de conception et d'intégration de composants de différentes natures (numériques, analogiques, logiciels).
- Standardiser des techniques de vérification des composants virtuels, seuls et à l'intérieur d'un système, à différents niveaux d'abstraction et après fabrication.
- Définir des standards pour la protection de la propriété intellectuelle.

Loin de s'orienter vers la définition systématique de nouvelles techniques et standard, l'effort de standardisation de ces groupes porte de préférence sur la sélection de techniques existantes et la recommandation de pratiques éprouvées industriellement [74].

Niveaux d'abstraction d'un composant virtuel

La spécification synthétisable d'un composant virtuel peut être délivrée sous différentes formes, en fonction de l'étape du flot de conception à laquelle elle a été extraite (*e.g.* avant ou après synthèse *RTL*, après placement/routage), du format ou langage utilisé par la description, et de la cible technologique considérée (bibliothèques *standard-cell* ou *full-custom*, familles de circuit programmables).

VSIA définit trois classes de composants virtuels matériels en fonction du niveau

TAB. 1.3 – Liste des groupes de travail (DWGs) de l'organisation VSIA

AMS	*Analog/Mixed-Signal*
HdS	*Hardware-Dependent Software*
FV	*Functional Verification*
IV	*Implementation Verification*
IPP	*Intellectual Property Protection*
MrT	*Manufacturing-Related Test*
OCB	*On-Chip Bus*
PBD	*Platform-Based Design Study Group*
SLD	*System-Level Design*
VCT	*Virtual Component Transfer*
VCQ	*Virtual Component Quality*

d'abstraction de leur description synthétisable [74] : (1) un composant virtuel *hard* se présente sous la forme de masques prêts à fondre; (2) un composant virtuel *firm* se présente comme une *netlist* optimisée au niveau logique, et munie éventuellement de directives de placement; (3) un composant virtuel *soft* se présente comme une description synthétisable en *VHDL* ou *Verilog* au niveau transfert de registres.

Ces trois classes de composants virtuels se distinguent par des degrés différents de prédictibilité et de flexibilité :

- les composants *hard* sont très prédictibles, mais très peu flexibles, car leur placement/routage interne est déjà optimisé pour des contraintes et une bibliothèque technologique fixées;
- les composants *soft*, au contraire, sont peu prédictibles, car l'optimisation au niveau logique n'a pas encore eu lieu, et supportent un degré de flexibilité qui autorise la personnalisation non seulement de la technologie cible et des contraintes d'optimisation logique, mais aussi de paramètres architecturaux.

Dans la pratique, les intégrateurs de composants virtuels privilégient les *IP soft* pour leur flexibilité. Les créateurs, en revanche, apparaissent réticents à délivrer du code *RTL* synthétisable, plus difficile à protéger contre les modifications et réutilisations abusives.

Dans un grand nombre de cas, un composant virtuel *soft* pour son créateur est en réalité délivré sous la forme d'un composant *firm*, la synthèse logique étant réalisée chez le créateur en fonction des directives de l'utilisateur.

Eléments constitutifs d'un composant virtuel

VSIA a établi une liste conséquente des délivrables associés à chaque classe de composants virtuels [75]. Ces délivrables comprennent notamment un manuel utilisateur, un ou plusieurs modèles simulables du composant et un modèle synthétisable.

Le *manuel utilisateur* décrit tout d'abord la fonctionnalité, l'interface, les éventuels paramètres et les performances du composant. Il détaille les procédures à suivre pour simuler le composant – seul et une fois intégré au système –, le personnaliser et le synthétiser.

Les *modèles simulables* fournis avec un composant virtuel répondent à trois objectifs : (1) évaluer avant achat la conformité du produit, en termes de fonctionnalité et de performances temporelles, avec les besoins de l'utilisateur ; (2) vérifier après achat que le modèle synthétisable est correct ; (3) accélérer la simulation fonctionnelle et temporelle du système en cours de conception en y insérant un modèle comportemental exact au cycle près du composant.

Le *modèle synthétisable*, enfin, est fortement dépendant de la classe à laquelle appartient le composant virtuel. Les formats de description standard *de facto* sont les langages *VHDL* et *Verilog* pour les description *RTL*, *EDIF* pour les netlists et *GDSII* pour les masques. Un modèle synthétisable s'accompagne de scripts de synthèse, modifiables par l'utilisateur, pour les principaux outils du commerce.

Commerce de composants virtuels

Le caractère *virtuel* des blocs *IP* privilégie un mode d'échange de type commerce électronique par Internet. Un utilisateur doit ainsi pouvoir rapatrier rapidement dans une base de données locale l'ensemble des délivrables constituant les composants qu'il souhaite intégrer dans le système en cours de conception.

Sélectionner le bon composant à implanter dans un système pour réaliser une fonction donnée nécessite de pouvoir confronter les propriétés des blocs *IP* disponibles sur le marché avec les besoins et contraintes de l'utilisateur. Celui-ci doit ainsi pouvoir accéder aux informations pertinentes relatives à chaque bloc *IP* par le biais de catalogues [112]. Les catalogues existants sont soit propres aux sociétés productrices de composants virtuels, soit indépendants comme le catalogue de la société *Design&Reuse* [112] qui permet la comparaison rapide des offres de différents fournisseurs.

La mise à disposition des composants virtuels à la communauté des concepteurs de systèmes nécessite enfin la mise en place de bibliothèques de composants et d'une infrastructure sécurisée pour la transmission des fichiers constitutifs d'un bloc *IP* [127].

1.2.4 Interopérabilité des flots de conception et d'intégration

La notion d'interopérabilité [71] impose des exigences aux quatre acteurs – créateur, fournisseur, outilleur et utilisateur – intervenant dans la conception et la réutilisation d'un composant virtuel :

- Le créateur doit assurer la conformité de sa spécification avec des règles de conception et des formats standard.
- Le fournisseur doit assurer la communication sans ambiguïté de l'information nécessaire à la sélection et à l'évaluation d'un composant par l'utilisateur avant achat.
- L'outilleur doit mettre à la disposition du créateur et de l'utilisateur des outils respectant les standards de conception et de synthèse.
- L'utilisateur doit sélectionner les outils adaptés à la synthèse et à la vérification du composant dans le format sous lequel il sera délivré et vers le type de technologie ciblé.

Règles de conception

La qualité d'un composant virtuel tient en premier lieu à l'application de bonnes pratiques de conception qui garantiront l'aptitude de l'utilisateur à interfacer le composant avec le reste du système et à obtenir une architecture fonctionnellement correcte après synthèse.

L'ouvrage de référence en matière de conception de composants virtuels est le *Reuse Methodology Manual* (*RMM*) [71], issu de la collaboration entre *Mentor Graphics* et *Synopsis*, qui détaille les règles de spécification au niveau transfert de registres, les stratégies de synthèse pour le développement de composants *soft* et *hard*, les méthodes de vérification d'un composant virtuel intégré dans un système de forte complexité.

Les règles de style pour la conception *RTL* ont pour objectif d'en faciliter le débogage, la maintenance et la portabilité d'un outil de synthèse ou d'une technologie vers un(e) autre. Un certain nombre de standards permettent d'atteindre cet objectif :

- le standard *IEEE 1164* (*Standard Multivalue Logic System for VHDL Model Interoperability* [39]) pour la portabilité des types de données au niveau logique ;
- le standard *VITAL* (*VHDL Initiative Towards ASIC Libraries – IEEE 1076.4* [41]) pour la modélisation fonctionnelle et temporelle au niveau portes des bibliothèques de composants *VHDL* ;
- les standards *IEEE 1076.6* [42] et *IEEE 1364.1* [43] décrivent les sous-ensembles syntaxiques *VHDL* et *Verilog* supportés en synthèse *RTL*.

L'avènement des langages de description de systèmes inspirés de *C/C++* et leur utilisation en synthèse *RTL* automatique nécessitent la définition de standards analogues. Cette tâche est menée à bien par le comité *ALC* (*Architectural Language Committee*) de l'organisation *Accellera*, par l'entremise du groupe de travail *CWG* (*C/C++ Working Group*) [30].

Interfaces génériques et platform-based design

Les protocoles de communication envisageables entre composants d'un système sont nombreux et de complexités variées. Ils vont de la simple liaison point à point aux réseaux sur silicium (*NoC* pour *Network-on-Chip*) en passant par les bus intégrés. Cette diversité limite fortement l'aptitude à faire communiquer entre eux des composants virtuels provenant de sources variées et dont les interfaces peuvent être fortement incompatibles.

Concevoir des adaptateurs de protocoles (*wrappers*) entre différents composants peut représenter un temps de développement significatif et engendrer des pertes de performance du système. Deux solutions sont actuellement envisagées :

La première repose sur la standardisation par le groupe *OCB* (*On-Chip-Bus*) de l'organisation *VSIA* [128] d'un modèle générique d'interface (*VCI* pour *Virtual Component Interface*) modélisant une connexion point à point mais pouvant être également connectée à un bus par l'intermédiaire d'un adaptateur de faible complexité [68].

La seconde solution, largement adoptée par les industriels, repose sur la notion de plate-forme système [9, 68], partant de la constatation que la majorité des systèmes intégrés est construite selon une architecture type intégrant un cœur de processeur

à usage général (*ARM*, *MIPS*, *etc.*) ou un *DSP* qui dialogue avec des périphériques (*DMA*, *UART*, *etc.*) par l'intermédiaire d'un bus. Les fournisseurs d'*IP* proposent ainsi sous l'appellation *plate-forme* une structure générique de système accompagnée d'un choix de composants virtuels matériels et logiciels (*RTOS*) pouvant y prendre place, ainsi que des outils d'aide à la personnalisation et à la vérification de cette architecture. L'effort de conception de l'utilisateur se réduit alors à l'implantation des composants virtuels provenant d'autres sources et qu'il souhaite ajouter au système.

La généralisation de cette pratique a conduit à la nécessité de formaliser et de standardiser les aspects liés à la conception par plate-formes (*Platform-based design*) [9, 15, 27]. *VSIA* a récemment mis en place un groupe d'étude (stade préliminaire avant la création effective d'un groupe de travail) chargé d'examiner ces questions. *VSIA* définit une plate-forme *SoC* comme "une bibliothèque de composants virtuels et une infrastructure architecturale composées d'un ensemble de composants virtuels matériels et logiciels intégrés et pré-qualifiés, de modèles, d'outils de conception matérielle et logicielle, de bibliothèques et de méthodologies pour permettre le développement rapide de produits par l'exploration architecturale, l'intégration et la réutilisation" [128].

1.2.5 Protection de la propriété intellectuelle

A l'inverse des composants en boîtier dont il faut acheter un exemplaire pour chaque produit, les composants virtuels sont distribués sous forme de fichiers qui peuvent être réutilisés à volonté. Construire une copie conforme d'un composant discret nécessiterait un effort significatif de *reverse engineering*, les détails d'implantation étant masqués à l'utilisateur. Un composant virtuel, au contraire, peut être dupliqué et re-vendu sans qu'une compréhension de son architecture soit nécessaire [76].

D'autre part, un composant virtuel *soft*, dont le modèle synthétisable est décrit en *VHDL* ou en *Verilog* est suffisamment lisible pour permettre à un utilisateur d'en extraire et d'en modifier des parties qu'il souhaite réutiliser dans d'autres projets.

Ainsi, la protection de la propriété intellectuelle est un problème majeur dans le système d'échange de composants virtuels. La standardisation de ces aspects est prise en charge par le groupe de travail *IPP* (*Intellectual Property Protection* de l'organisation *VSIA* [76] et a déjà donné lieu à une première spécification de recommandations – *Virtual Component Identification and Physical Tagging* (*VCID* 1.1) – concernant le marquage des composants virtuels *hard*.

La protection de la propriété intellectuelle se décline ainsi sous deux axes [76] : (1) prévenir les réutilisations hors licence d'un composant ; (2) prévenir l'appropriation et la modification par les utilisateurs de tout ou partie d'une description.

Le premier axe met en jeu des techniques de marquage (*watermarking*) des composants, permettant d'en identifier le concepteur et l'utilisateur officiel [70]. L'efficacité d'une technique de marquage repose sur quatre critères [72] :

- Une marque doit être difficile à retirer.
- Il doit être difficile d'ajouter une marque à un composant une fois qu'il a été délivré.
- Une marque doit être indétectable par l'utilisateur.

- Elle doit préserver la fonctionnalité du composant et son influence sur les performance doit être négligeable.

Le second axe nécessite la mise en œuvre de techniques de cryptage des descriptions afin de rendre leur contenu incompréhensible aux utilisateurs. Avant l'acquisition définitive d'un composant par un utilisateur, le cryptage sert en particulier à en interdire la synthèse tout en autorisant sa simulation. Nous évoquerons ce point dans la présentation des outils de génération de modèles simulables.

1.2.6 Outils d'aide à la spécification et à la réutilisation de composants virtuels

En plus des outils utilisés en routine pour la spécification, la synthèse et la vérification d'architectures, la notion de composant virtuel réutilisable a donné lieu à l'élaboration de nouvelles classes d'outils :

- Les outils de qualification de composants virtuels (*QuickUse IP Qualification System* de *Mentor Graphics* [118]).
- Les outils de génération de modèles simulables (*IP Model Packager* de *Cadence* [108] ; *CMC*, *VhMC*, *VMC* de *Synopsys* [125] ; *Visual IP* de *Summit Design* [122]).
- Les outils de gestion de catalogues et de bibliothèques de composants (*IP E-Design Manager* de *Design&Reuse* [112], *IP Gear* de *Synchronicity* [123]).
- Les outils d'aide à l'intégration (*VCC* de *Cadence* [109], *CoWare N2C* [111]).

Dans ce paragraphe, nous nous intéresserons plus particulièrement aux outils de génération de modèles simulables et aux outils d'aide à l'intégration.

Les outils d'aide à la génération de modèles simulables

Ces outils permettent de délivrer aux intégrateurs des modèles d'évaluation protégés de composants virtuels. Ces modèles sont obtenus tout d'abord au moyen de techniques de compilation et de cryptage de la description *VHDL* ou *Verilog* du composant, afin de masquer à l'utilisateur les détails de l'architecture interne.

Le concepteur d'un composant virtuel sélectionne les signaux qu'il souhaite rendre accessibles à l'utilisateur. La description compilée du composant est alors encapsulée dans un modèle d'interface exécutable (*BFM* pour *Bus Functional Model*) permettant la simulation au cycle près au moyen de logiciels du commerce. La portabilité vers les logiciels de simulation est assurée au moyen d'interfaces logicielles standard, telles *OMI* (*Open Model Interface, IEEE 1499*), et *PLI* (*Procedural Language Interface*) qui permettent l'instanciation dans des descriptions *VHDL* et *Verilog* de modules compilés spécifiés en langage *C*.

Parmi les outils disponibles sur le marché, on retiendra les suivants :

- *IP Model Packager* de *Cadence* propose une compilation du modèle de l'*IP* en code natif, c'est-à-dire directement exécutable par le processeur de la plate-forme de simulation de l'utilisateur [108].
- Les outils *CMC*, *VMC* et *VhMC* (*C, Verilog, VHDL Model Compiler*) de *Synopsys* permettent la compilation de modules hiérarchiques multi-langages (*C*,

VHDL, *Verilog*) [125].
- L'outil *Visual IP* de *Summit Design* offre, en plus des fonctionnalités supportées par les précédents, la possibilité d'intégrer un banc de test, des vecteurs de test et de la documentation au modèle protégé d'un composant selon un procédé de création interactive de *datasheet* [122].

Les outils de co-conception matériel/logiciel

Les outils d'aide à la réutilisation de composants virtuels s'inscrivent dans une perspective générale de conception système [78]. Il permettent dans un premier temps l'entrée de la spécification initiale sous forme mixte structurelle/comportementale, en orthogonalisant la fonctionnalité de chaque composant et les aspects liés à la communication avec son environnement. Ces outils permettent dans un deuxième temps la projection de ces modèles comportementaux et des modèles de communication sur des composants virtuels en bibliothèque et des modèles de protocoles [109, 111].

En parallèle à ces deux tâches de spécification et de raffinement, l'utilisateur peut effectuer la vérification du système à différents niveaux d'abstraction par co-simulation matériel/logiciel. Cette co-simulation est mise en œuvre au moyen d'un moteur de simulation interne qui pilote d'une part les outils de simulations matérielle disponibles dans l'environnement de l'utilisateur (par exemple *ModelSim* de *Mentor Graphics*), et d'autre part des simulateurs de jeux d'instructions (*ISS*) pour l'exécution des parties logicielles. Le moteur de simulation s'accompagne de techniques de profilage permettant à un concepteur d'estimer l'impact de certains choix architecturaux sur le résultat de synthèse finale.

Les principaux outils disponibles relevant de ce domaine sont *VCC* de *Cadence* [109] et *CoWare N2C* [111].

1.3 Explorer l'espace des solutions architecturales : la synthèse de haut niveau

1.3.1 De la synthèse *RTL* à la synthèse de haut niveau

L'avènement des outils de synthèse *RTL* a permis de combler en partie le fossé entre la *modélisation* d'une architecture, en utilisant des langages de description de matériel comme *VHDL* et *Verilog*, et la *réalisation* proprement dite d'un circuit *VLSI*. La spécification au niveau transfert de registres en vue de la synthèse s'appuie sur un ensemble bien défini, et standardisé [42, 43], de constructions syntaxiques pour la description hiérarchique d'architectures à base de composants combinatoires et séquentiels. Aujourd'hui, les outils de synthèse *RTL* automatique ont atteint un degré de maturité tel que leur utilisation fait désormais partie de la pratique courante dans le domaine de la conception de circuits *VLSI* numériques.

Face à la complexité croissante des applications et des algorithmes à implanter, le recours à une élévation du niveau d'abstraction pour la spécification et la vérification d'un système (voir section 1.1) ouvre la voie à un vaste champ de recherche visant à automatiser l'exploration architecturale. Si les nouveaux langages de spécification de haut niveau d'une part, et la réutilisation de composants virtuels d'autre part, permettent en effet à un concepteur de systèmes de rehausser le niveau d'abstraction

des briques de base de sa description, ces approches ne font cependant que déplacer le problème de la spécification détaillée et de la synthèse de chaque composant [78, 79].

La synthèse de haut niveau (*HLS* pour *High-Level Synthesis*) – également appelée synthèse *comportementale* ou synthèse d'architecture – s'inscrit ainsi dans une perspective complémentaire aux méthodologies de conception au niveau système en automatisant une partie des tâches de raffinement d'un composant matériel jusqu'au niveau micro-architectural (*RTL*) [56]. Si les outils commerciaux de synthèse de haut niveau souffrent actuellement d'un manque de maturité, le besoin en nouvelles techniques de synthèse permettant de passer plus rapidement du niveau système au silicium – besoins exprimés dans l'*International Technology Roadmap for Semiconductors* 2001 [18] – devrait provoquer un regain d'intérêt pour ces outils et favoriser leur développement.

1.3.2 Décrire au niveau comportemental

La spécification d'entrée d'un outil de synthèse de haut niveau se veut aussi proche que possible du niveau algorithmique de manière à fournir, d'une part, un style de description plus intuitif, lisible et concis, et d'autre part à reporter les décisions d'implantation sur les étapes d'un flot de raffinement automatique ou interactif sous contraintes. Dans la plupart des outils de synthèse de haut niveau, de la spécification d'entrée est dite "de niveau *comportemental*", c'est-à-dire qu'elle s'attache à décrire l'algorithme à implanter sous la forme d'un processus répétitif en utilisant des constructions syntaxiques proches de celles utilisées dans les langages de programmation.

La synthèse de haut niveau peut ainsi être comparée à la compilation dans le domaine du logiciel : il s'agit dans les deux cas de passer d'un modèle décrit à un haut niveau d'abstraction – en utilisant un style d'écriture aussi proche que possible de la structure de l'algorithme à implanter et aussi indépendant que possible de la cible chargée de l'exécuter – à une description détaillée (code binaire exécutable par un processeur, modèle *RTL*).

Une définition ?

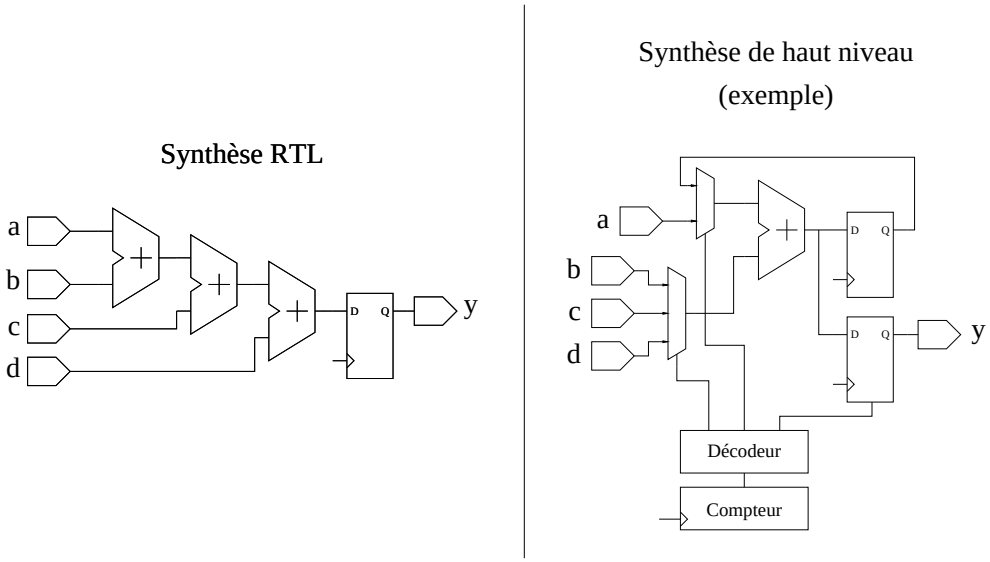
En synthèse de haut niveau, le *niveau comportemental* est considéré comme un niveau d'abstraction à part entière [79]. Dans le diagramme en *Y*, une description orientée vers la synthèse de haut niveau sera située sur l'axe correspondant au *domaine comportemental*, à mi-chemin entre le *niveau algorithmique* et le *niveau transferts de registres*.

En effet, une telle description est généralement le résultat du *raffinement temporel* d'une spécification algorithmique, au sens où les entrées/sorties et les traitements sont distribués sur les itérations successives d'un ou de plusieurs processus concurrents. Ainsi le niveau comportemental se caractérise par une notion de temps plus ou moins fine d'un point de vue *externe* – selon les outils de synthèse, l'interface peut être spécifiée au niveau transaction ou au cycle près – et une absence de temps d'un point de vue *interne* – l'ordonnancement des calculs au cycle près n'est pas spécifié.

Du point de vue du style d'écriture – et ceci se vérifie plus particulièrement lorsque le langage utilisé est un langage de description de matériel comme *VHDL* ou *Verilog* –

FIG. 1.5 – Interprétation d'une description comportementale par les outils de synthèse RTL et de haut niveau

```
process
begin
  wait until clk'event and clk='1';
  y <= a + b + c + d;
end process;
```



la distinction entre une description *RTL* et une description comportementale peut ne pas apparaître de manière flagrante. Au point que les deux types d'outils de synthèse peuvent traiter sans erreur une même description, mais avec des résultats souvent très différents.

L'exemple de la figure 1.5 montre comment un processus *VHDL* contenant une évaluation d'expression arithmétique synchronisée sur un signal d'horloge est interprétée par les outils de synthèse *RTL* et les outils de synthèse de haut niveau. Dans le premier cas, l'expression est traitée comme un bloc combinatoire dont la sortie est mise en registre. Dans le deuxième cas, l'expression est ordonnancée sur plusieurs cycles d'horloge et le nombre de ressources matérielles est optimisé en fonction du compromis vitesse/surface souhaité par le concepteur.

A titre de définition, nous retiendrons pour le moment qu'une description relève de la synthèse de haut niveau lorsque les conditions suivantes sont réunies :

1. Le composant à synthétiser est décrit en termes de comportement et non de structure.
2. La description du comportement ne contient pas d'information temporelle à l'exception des aspects externes (entrées/sorties) du composant.
3. Les contraintes de temps autorisent un temps de latence du comportement supérieur à une période d'horloge.

En synthèse *RTL*, des constructions syntaxiques bien identifiées peuvent être projetées directement sur des structures matérielles combinatoires ou séquentielles. En synthèse de haut niveau, la traduction *ligne à ligne* d'une description en une architecture n'est généralement pas possible : une description comportementale est tout d'abord projetée sur un modèle de représentation intermédiaire – généralement une graphe flot de données (*DFG*) ou un graphe flot de données et de contrôle (*CDFG*). Les techniques de synthèse employées sont fondées sur une certaine *interprétation* de ce modèle en termes de comportement temporel et de structure matérielle. Cette interprétation introduit une notion de *sémantique d'implantation* d'une description comportementale et peut être fortement dépendante de l'outil de synthèse utilisé.

Les avantages

Comme une description comportementale ne contient idéalement aucun détail d'implantation au niveau micro-architectural, l'effort d'écriture qu'elle nécessite est nettement inférieur à celui exigé par une description *RTL*. Par exemple, dans l'application présentée dans [63] pour le contrôle de trafic *ATM*, la description *RTL* nécessite un nombre de lignes de code *VHDL* trois fois supérieur à la description comportementale. Dans la majorité des cas, l'écriture d'un modèle comportemental est rendue d'autant plus aisée qu'un modèle de référence en *C* ou *C++* est généralement rédigé préalablement à la conception de l'architecture.

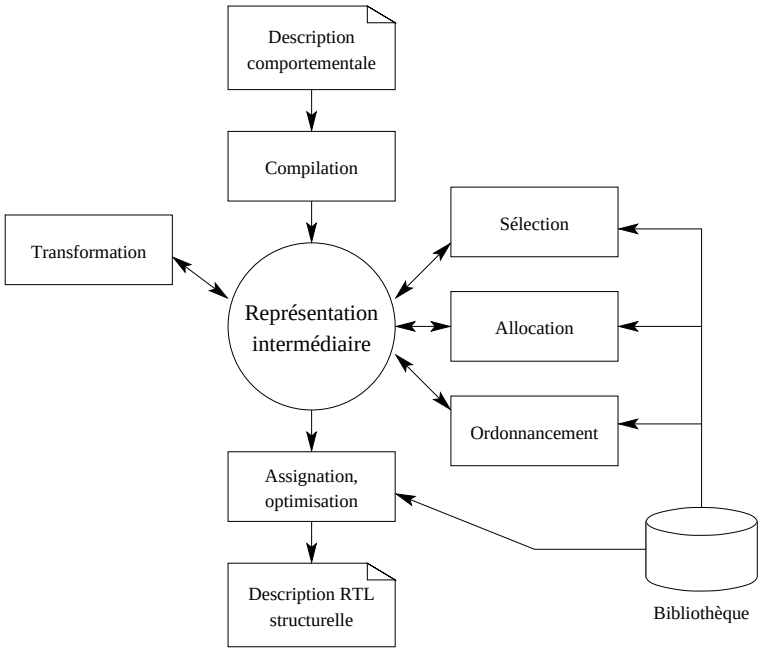
Du point de vue de la vérification, une description comportementale est tout d'abord plus facile à déboguer, car plus concise et plus lisible, qu'une description *RTL*. Dans cette dernière, les informations relatives à la fonctionnalité et au déroulement de l'algorithme sont en effet noyées dans la multitude de détails structurels de l'architecture. Une description comportementale est également plus rapide à simuler : la simulation *RTL* doit en effet gérer un grand nombre de signaux décomposés bit par bit et produisant chacun, éventuellement de manière asynchrone, un nombre significatif d'événements au cours du temps. Une description comportementale, au contraire, manipule des données de types *abstraits* (entiers, booléens, nombre en virgule fixe) ; l'évaluation d'une expression arithmétique ne nécessite pas de synchronisation entre opérateurs ; les échanges de données entre un processus et son environnement se font sur des points de synchronisation bien définis.

Enfin, le niveau comportemental s'insère naturellement dans un flot de synthèse système reposant sur une stratégie d'exploration architecturale. Dans la continuité de l'exploration au niveau système des différentes solutions de partitionnement matériel/-logiciel, le fait qu'un modèle comportemental contienne le moins possible de détails d'implantation matérielle autorise le concepteur à évaluer une variété de solutions *RTL* répondant à différents compromis de performances.

1.3.3 Le flot de synthèse de haut niveau

Un flot typique de synthèse de haut niveau procède par transformations successives d'une représentation intermédiaire de la description comportementale. Ce format intermédiaire encapsule les éléments pertinents de la description comportementale –

FIG. 1.6 – Flot général de synthèse de haut niveau



c'est-à-dire débarrassés de leur enveloppe syntaxique – et intègre de manière incrémentale les décisions d'implantation prises à chaque étape de synthèse.

Un flot générique de synthèse de haut niveau se décompose en cinq grandes étapes représentées sur la figure 1.6.

L'étape de compilation – Cette première étape réalise la vérification syntaxique et sémantique de la description source et la traduit dans le format intermédiaire supporté par l'outil. Cette traduction s'accompagne généralement de transformations de code visant d'une part à faire disparaître les éléments inutiles ou redondants d'une description (élimination de code mort, pré-évaluation et propagation des expressions constantes, mise en évidence des sous-expressions communes, *etc.*) et d'autre part à mettre en évidence des propriétés spécifiques au modèle de représentation intermédiaire. Deux types de modèles sont couramment employés (figure 1.7).

Les représentations de type graphe flot de données (*DFG*) (voir paragraphe 1.1.3) permettent d'identifier les dépendances de données entre les opérations afin de mettre en évidence les possibilités de parallélisation des calculs. Si une description comportementale fais usage d'instructions structurées telles des boucles ou des appels de sous-programme, une "mise à plat" de ces instructions sera nécessaire à l'élaboration d'un *DFG*. Les manipulations couramment effectuées sont le déroulage des boucles – dans la mesure où elles possèdent un nombre d'itération déterminé – et la mise en ligne des appels de sous-programmes.

Une des principales limitations des *DFG* réside dans leur incapacité à exprimer les comportements non déterministes, où le nombre d'itérations d'une boucle pourrait

par exemple varier en fonction des données fournies au composant. Une modélisation de type graphe flot de contrôle et de données (*CDFG*) permet de traiter une plus large classe de descriptions comportementales et laisse le choix d'un ensemble plus vaste de solutions architecturales.

L'étape de sélection – Elle consiste à choisir la nature des ressources matérielles qui réaliseront chaque opération (par la suite, nous distinguerons les notions d'*opération*, désignant une fonction élémentaire, et d'*opérateur*, désignant un composant matériel réalisant cette fonction). Ces composants sont choisis dans une bibliothèque en fonction de critères tels que leur surface combinatoire, leur vitesse, leur consommation. Les outils possèdent généralement une variété de bibliothèques d'opérateurs caractérisés pour différentes technologies *ASIC* ou *FPGA*.

Ces bibliothèques peuvent être enrichies par l'utilisateur, soit en proposant de nouvelles structures des opérateurs de base, soit en créant des opérateurs réalisant de nouvelles fonctions.

L'étape d'allocation – Cette étape détermine pour chaque type d'opérateur sélectionné le nombre d'instances à implanter dans l'architecture finale. Le nombre d'opérateurs alloué est choisi en fonction du parallélisme exploitable et du compromis surface/vitesse souhaité par l'utilisateur.

L'étape d'ordonnancement – Au cours de cette étape, une date d'exécution est affectée à chacune des opérations en tenant compte des dépendances de données d'une part, et des contraintes de performances d'autre part. Selon la valeur de la période d'horloge et les caractéristiques des composants sélectionnés, différentes stratégies d'ordonnancement peuvent être envisagées (figure 1.8).

La plus simple repose sur la sélection d'opérateurs dont le temps de traversée n'excède pas la période d'horloge (figure 1.8a). Il est alors possible d'associer à chaque opération un cycle d'exécution. Au niveau architectural, cette technique impose la mise en registre de la sortie de chaque opérateur.

Le *chaînage* consiste à faire se succéder dans un même cycle une séquence d'opérations présentant une chaîne de dépendances (figure 1.8b). Au niveau architectural, les opérateurs concernés seront reliés par des connexions directes, ce qui autorise des optimisations combinatoires lors de la synthèse logique.

L'utilisation d'opérateurs multi-cycles, au contraire, permet l'utilisation d'opérateurs dont le temps de traversée est supérieur à la période d'horloge. La figure 1.8c montre qu'en réduisant la période d'horloge, nous donnons plus de liberté à l'outil de synthèse pour ordonnancer les opérations, ce qui conduit ici à une solution plus rapide, qui comble les temps morts entre opérations.

Dans les exemples des figures 1.8a à 1.8c, l'allocation d'un unique multiplieur impose l'exécution des multiplications sur des intervalles de temps non recouvrants. Cette limitation peut être levée à l'aide d'opérateurs pipeline (figure 1.8d), dont chacun des étages peut être considéré comme un opérateur indépendant.

Enfin la durée du chemin critique peut être incompatible avec une contrainte de cadence d'entrée/sortie des données. Il peut alors être nécessaire de débuter une nouvelle itération du processus avant que l'itération en cours ne soit terminée. Sur la figure

FIG. 1.7 – Deux modèles de représentation d'une description comportementale

```
tmp := x(0)*h(0);
for i in 1 to 4 loop
   tmp := tmp + x(i)*h(i);
end loop;
y <= tmp;
```

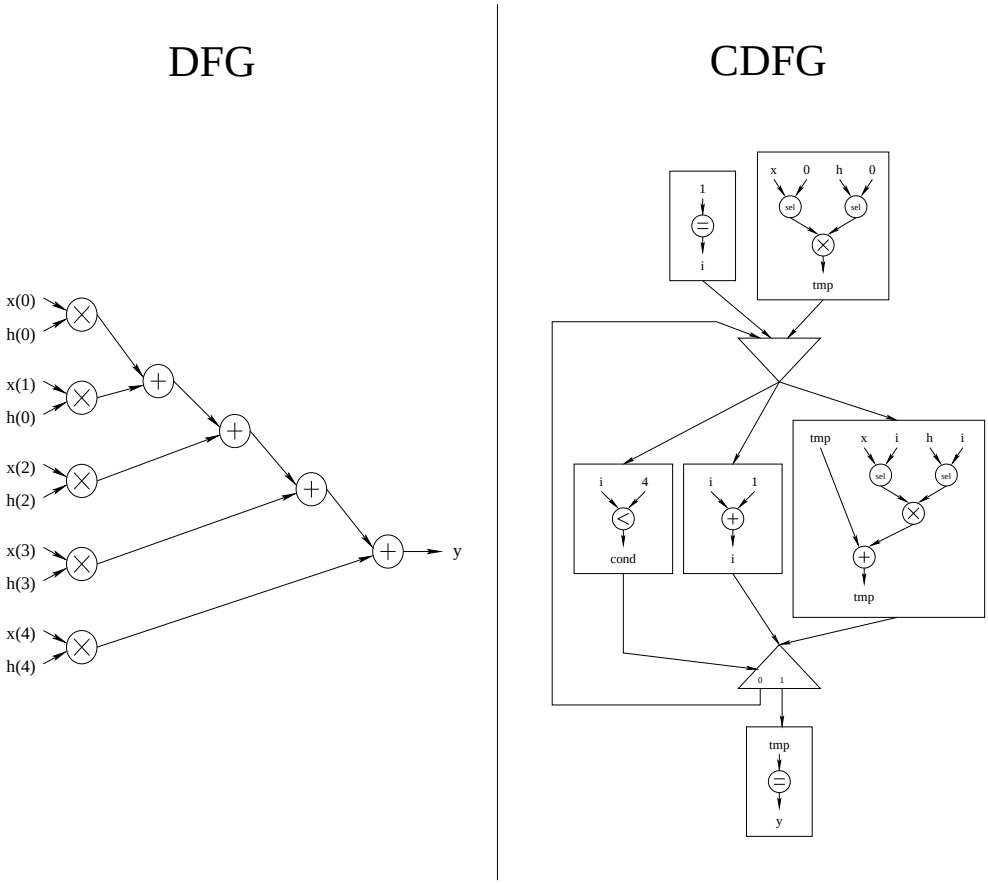
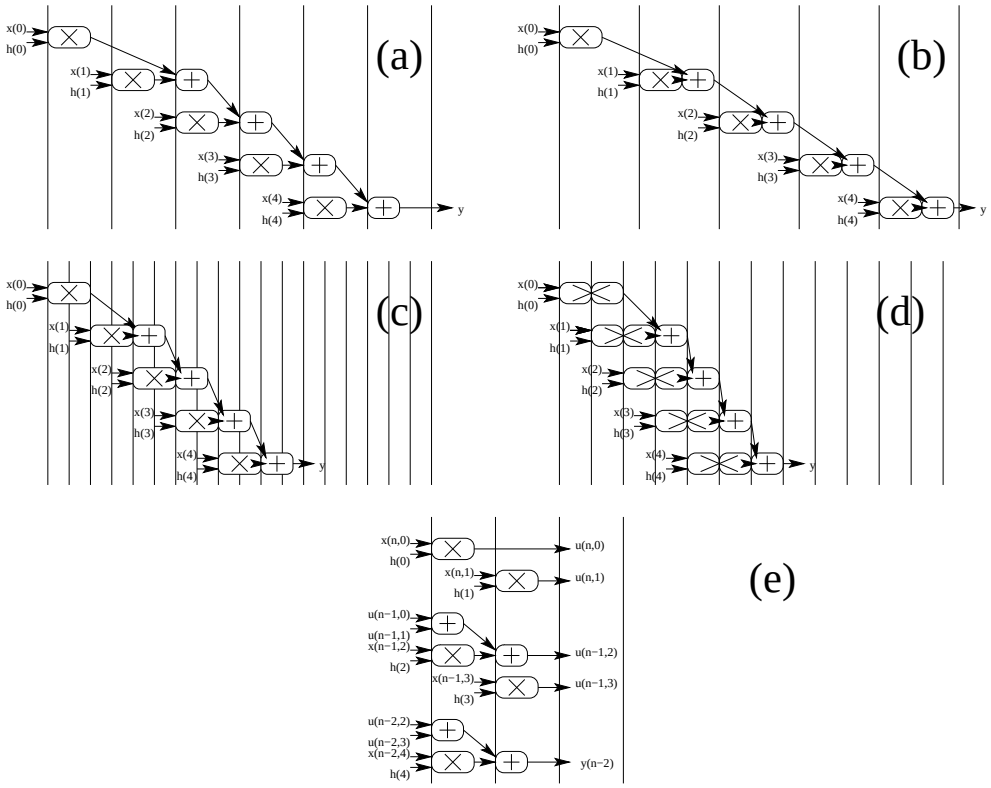


FIG. 1.8 – Différentes stratégies d'ordonnancement en synthèse de haut niveau : (a) une opération par cycle ; (b) chaînage ; (c) avec opérateurs multicycles ; (d) avec opérateurs pipeline ; (e) chemin de données pipeline



1.8e, l'ordonnancement de la figure 1.8a a été décomposé en trois étages de pipeline, où la sortie y_{n-2} est calculée avec un retard de deux itérations par rapport aux entrées les plus récentes $x_n(0)$ et $x_n(1)$. En sortie de chaque étage, des données intermédiaires $u_i(k)$ sont mémorisées afin d'être lue par l'étage suivant lors de l'itération suivante. L'ordonnancement obtenu présente une latence égale à la version non pipelinée (ici 6 cycles s'écoulent entre l'entrée $x_n(0)$ et la sortie y_n) pour une cadence multipliée par trois (un nouveau jeu de données d'entrée est fourni tous les deux cycles).

L'ordre de succession des étapes d'allocation et d'ordonnancement peut varier selon les outils et les contraintes de synthèse. De manière schématique, si le critère principal d'optimisation est la surface, il sera préférable de commencer par allouer des ressources dans les limites fixées par la contrainte de surface, puis d'ordonnancer les opérations en limitant le parallélisme en fonction du nombre de ressources allouées. Si au contraire les contraintes de temps sont prioritaires, il sera préférable de procéder d'abord à l'ordonnancement, puis d'allouer les ressources en fonction du parallélisme maximal exploité.

En réalité, les deux étapes sont intimement liées. Dans le cas où l'allocation est

effectuée en premier, elle est en général précédée d'une étape de pré-ordonnancement permettant d'estimer le parallélisme exploitable [61].

L'étape d'assignation – Elle finalise les choix effectués lors des étapes précédentes en associant chacune des opérations ordonnancées à un des opérateurs alloués. Au terme de cette étape, une architecture structurelle est obtenue, qui interconnecte un ensemble d'opérateurs, un ensemble d'éléments de mémorisation (registres), un contrôleur de type machine à état fini qui pilote la circulation des données entre opérateurs et registres par l'intermédiaire d'un réseau combinatoire (portes logiques, multiplexeurs).

Lors de cette étape, la quantité de matériel est optimisée en fusionnant les utilisations de registres et d'opérateurs qui ont été ordonnancées dans des intervalles de temps disjoints, et en insérant la logique nécessaire au partage des composants concernés. Cette étape d'optimisation, qui serait particulièrement lourde à mettre en œuvre manuellement, permet de répartir au mieux la charge de calcul sur les ressources matérielles allouées.

1.3.4 Synthèse de haut niveau et synthèse système

L'intégration de la synthèse de haut niveau dans un flot de synthèse système doit satisfaire une contrainte d'interopérabilité entre les flots de spécification en amont et de synthèse en aval : en amont, des passerelles bien formalisées doivent exister entre les niveaux d'abstraction et langages de spécification au niveau système d'une part, et les langages et styles d'écriture supportés par les outils de synthèse de haut niveau d'autre part. En aval, les descriptions *RTL* de sortie doivent être compatibles avec les outils de synthèse *RTL* utilisés en routine. Elles doivent également être accompagnées de scripts de synthèse adaptés à ces outils.

L'état de l'art des outils de synthèse de haut niveau universitaires et commerciaux montre que les langages supportés pour la description au niveau comportemental sont variés. Ils recouvrent :

- des sous-ensembles de haut niveau de langages de description de matériel comme *VHDL* (pour *Monet* de *Mentor Graphics* [53, 62] et *Behavioral Compiler* de *Synopsys* [57, 64], *GAUT* des laboratoires *LESTER* et *LASTI* [61], *AMICAL* du *TIMA* [63]), ou *Verilog* (*Behavioral Compiler* [57] et *Get2Chip Architectural Compiler* [113]) ;
- des sous-ensembles ou sur-ensembles de langages de programmation comme le langage *C* (*A/RT Designer* de la société *Adelante Technologies* [107], *Bach* des laboratoires *Sharp* [120]) ;
- des sous-ensembles de langages de description de systèmes comme *SystemC 1.0* (*CoCentric SystemC Compiler* de *Synopsys* [124]) ;
- des langages dédiés comme le *Silage* (*Hyper*, *Cathedral*), *Alpha* (*MMAlpha* de l'*IRISA*) [114] et *HardwareC* (*Olympus*).

A l'exception de *CoCentric SystemC Compiler*, le raffinement d'un modèle système d'un composant au moyen de la synthèse de haut niveau nécessite une étape de traduction qui, si elle n'est pas automatisée, peut induire des erreurs. Dans tous les cas, une adaptation du style d'écriture peut être nécessaire afin de rendre une description compatible avec les règles de conception propres à un outil donné.

De même, les outils de synthèse associent à ces différents langages une variété de modèles formels supportant différents niveaux de spécification de la synchronisation, de la communication et des propriétés temporelles d'un composant. On trouvera notamment :

- des modèles à événements discrets : c'est le cas de l'outil *AMICAL* où le comportement est synchronisé sur des événements [52, 63] ;
- des modèles à temps discret : c'est le cas de l'outil *GAUT* où un processus *VHDL* est associé à une durée d'itération fixe [61] ;
- des modèles où seul l'ordre des entrées/sorties est précisé : c'est le cas des outils *Monet* [53, 62], *Behavioral Compiler* [57, 64] et *CoCentric SystemC Compiler* en mode *superstate-fixed* [124] ;
- des modèles avec entrées/sorties au cycle près : c'est le cas des outils *Monet*, *Behavioral Compiler* et *CoCentric SystemC Compiler* en mode *cycle-fixed* et de l'outil *Get2Chip Architectural Compiler* [113] ;
- des modèles sans synchronisation explicite : c'est le cas de l'outil *A/RT Designer* [107].

En aval, les outils commerciaux comme *Monet* et *A/RT Designer* produisent des descriptions *VHDL* ou *Verilog* compatibles avec les outils de synthèse *RTL*. Les outils de la chaîne *Synopsys* comme *Behavioral Compiler* et *CoCentric SystemC Compiler* s'interfacent directement avec *Design Compiler* pour la synthèse *RTL*.

Le style de description au niveau comportemental et le mode d'expression des contraintes peuvent être déroutants pour un grand nombre de concepteurs *RTL* qui ne disposent plus des leviers habituels pour optimiser une architecture. Les réticences des ingénieurs quant à l'utilisation de ces nouveaux outils tiennent en particulier au fait qu'il devient difficile d'attacher une sémantique d'implantation précise aux constructions syntaxiques autorisées, d'où un manque de prédictibilité du flot de synthèse [2, 4, 8]. Nos expériences sur les outils *Monet* et *Behavioral Compiler* montrent que, bien qu'une description comportementale ne contienne pas de détail d'implantation d'ordre structurel, le choix d'un style d'écriture peut avoir une influence déterminante sur le résultat de la synthèse [2, 4]. L'adoption de la synthèse de haut niveau dans le milieu industriel nécessite la mise en place de règles de conception précises permettant à un concepteur de mesurer l'impact de ses décisions, en termes de choix de style d'écriture, au niveau comportemental sur les performances des architectures après synthèse de haut niveau [80].

Un état de l'art des principaux outils de synthèse de haut niveau commerciaux est présenté en annexe A.

Chapitre 2

Conception de Composants Virtuels Flexibles de Haut Niveau

2.1 Problématique

2.1.1 Synthèse de haut niveau et réutilisation

Afin de définir une méthodologie de conception et d'intégration de composants virtuels de niveau comportemental, il nous faut tout d'abord confronter les caractéristiques des outils de synthèse de haut niveau existants – en termes de langages de spécification et modèles sémantiques – avec les besoins et exigences liées à la réutilisation.

Comme nous l'avons mentionné au paragraphe 1.2, ces besoins et exigences portent sur :

- la flexibilité d'un composant, qui doit pouvoir s'intégrer dans une variété d'applications soumises à une variété de contraintes ;
- l'interopérabilité de la spécification d'entrée et de sortie d'un outils de synthèse de haut niveau avec les outils de conception *front-end* (spécification et raffinement système) et *back-end* (synthèse *RTL*) ;
- la prédictibilité des performances ;
- la protection de la propriété intellectuelle.

Synthèse et flexibilité architecturale

Dans le contexte de cette étude, nous définissons la notion de flexibilité d'un composant virtuel comme l'aptitude à adapter statiquement – c'est-à-dire avant synthèse – l'architecture du composant en fonction de paramètres fonctionnels, temporels, et de contraintes de performances.

Concevoir des spécifications d'architectures flexibles au niveau *RTL* est une tâche relativement difficile. Une description *VHDL RTL* peut faire usage des clauses *generic*, permettant de spécifier des composants paramétrables, et d'instructions *generate* autorisant l'instanciation conditionnelle ou itérative de composants [40]. Ces mécanismes nécessitent cependant que les paramètres de l'algorithme à implanter puissent être mis directement en correspondance avec les propriétés structurelles de l'architecture. Par

exemple le nombre de cœfficients d'un filtre *FIR* conditionne directement le nombre de cellules *MAC* (multiplication/accumulation) à instancier.

Le créateur d'un composant virtuel *soft* personnalisable est ainsi responsable du choix d'un modèle d'architecture extensible facilement paramétrable. Ce modèle d'architecture étant figé au moment de l'intégration du composant, l'utilisateur ne dispose d'aucun levier pour adapter les performances à ses besoins.

Dans un outil de synthèse de haut niveau, le choix de la structure détaillée du circuit n'est plus du ressort du concepteur. Ainsi, la relation entre les paramètres d'une description, les contraintes de synthèse et la structure de l'architecture est indirecte : l'influence de la valeur des paramètres et des contraintes est reportée sur le modèle intermédiaire de représentation du comportement (*DFG* ou *CDFG*) comme schématisé par la figure 2.1.

Ainsi, l'ensemble des valeurs de paramètres autorisées, conjugué à l'ensemble des contraintes de synthèse exprimables, permet d'élargir considérablement l'espace des solutions architecturales (figure 2.2).

Interopérabilité des flots de conception

La question de l'interopérabilité des flots de conception est étroitement liée aux formalismes de représentation :

- au niveau *syntaxique*, se pose le problème de la diversité des langages de description au niveau comportemental (voir paragraphe 1.3.2) ;
- au niveau *sémantique*, il est nécessaire de disposer d'une définition commune de la notion de *description comportementale* et de modèles formels supportant cette définition.

Si la synthèse de haut niveau a fait l'objet de nombreux travaux de recherche et donné lieu ces vingt dernières années au développement de nombreux outils expérimentaux dans les laboratoires universitaires (voir tableaux A.1 et A.3 en annexe A), l'émergence des outils de synthèse de haut niveau commerciaux reste récente et n'a pas à l'heure actuelle fait l'objet d'une utilisation industrielle suffisante pour que la nécessité de standards – analogues des standards *IEEE 1076.6* [42] et *1364.1* [43] pour la synthèse *RTL* à partir des langages *VHDL* et *Verilog* – se soit fait sentir.

L'interopérabilité entre les outils de synthèse de haut niveau concerne également le jeu de contraintes supportées et la manière dont ces contraintes sont exprimées – *e.g.* sous forme de scripts ou par l'intermédiaire de directives de synthèse incluses dans la description comportementale.

Prédictibilité des performances

La différence majeure entre synthèse *RTL* et synthèse de haut niveau tient au fait que dans la première, des constructions syntaxiques *VHDL* ou *Verilog* bien définies permettent d'inférer de manière immédiate et non ambiguë des structures matérielles à base de composants combinatoires et séquentiels de faible complexité – registres, multiplexeurs, *etc.* En synthèse de haut niveau, au contraire, les choix d'implantation structurelle ne sont plus exprimés dans la description elle-même, mais externalisés dans

FIG. 2.1 – Influence des paramètres d'un composant virtuel sur l'architecture RTL

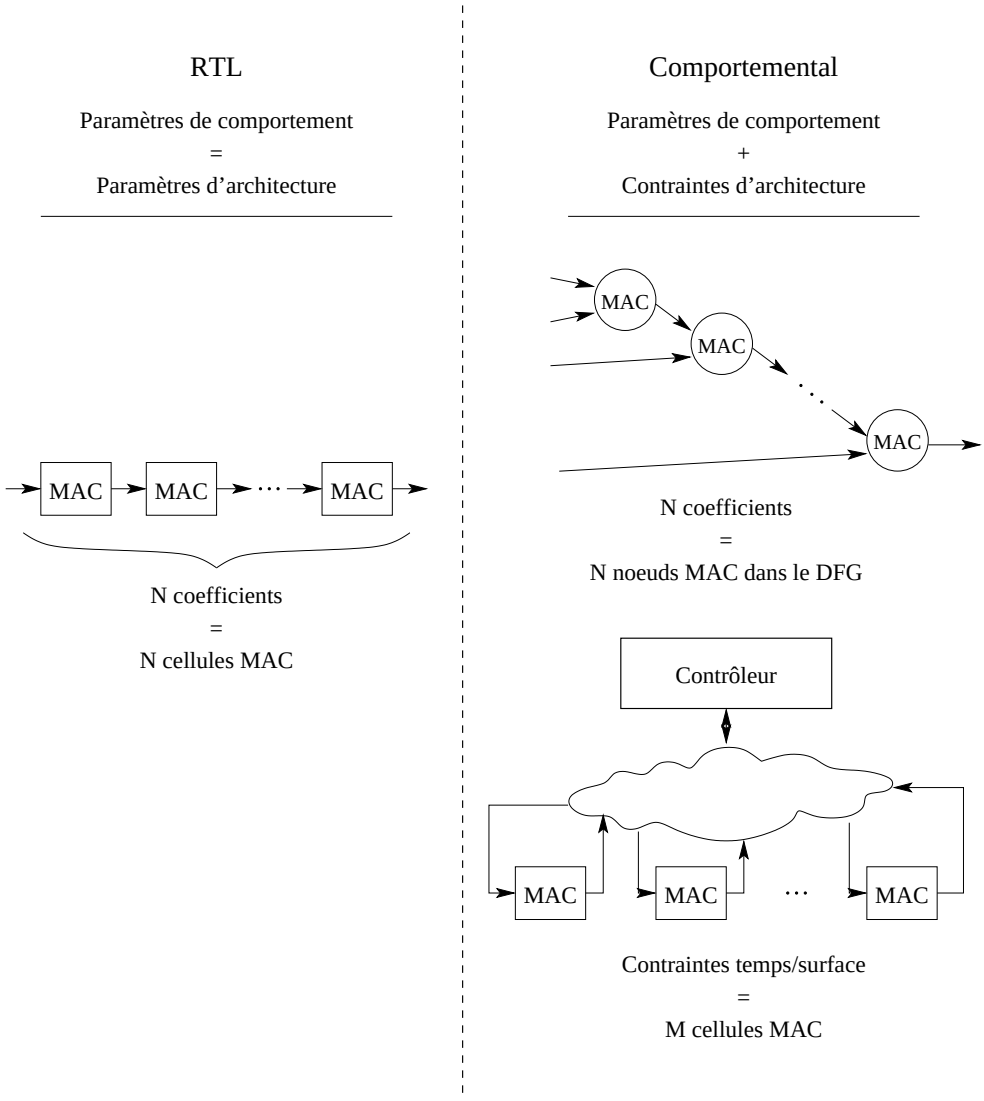
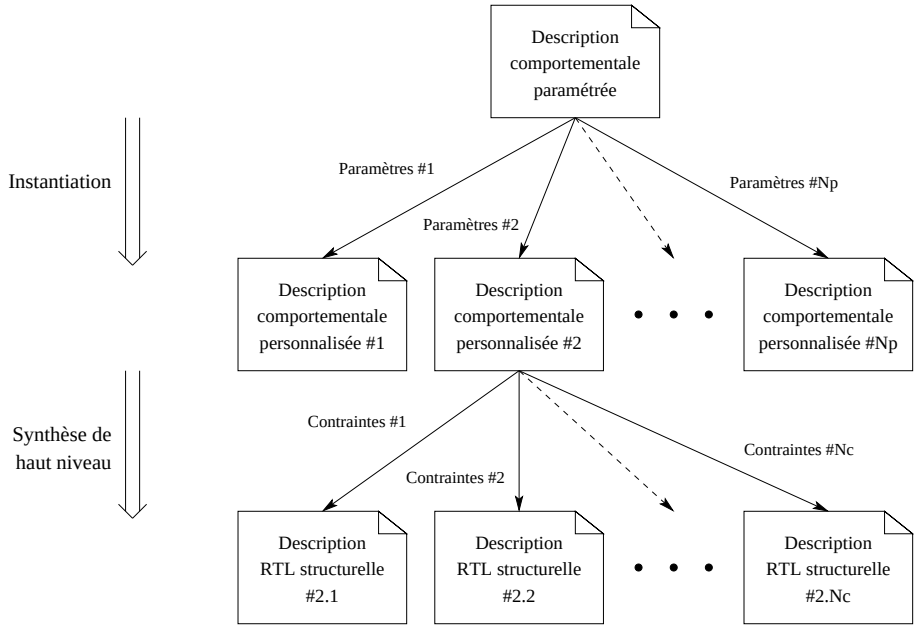


FIG. 2.2 – Synthèse de haut niveau et flexibilité



le modèle de représentation intermédiaire (*DFG*, *CDFG*) du comportement et dans les contraintes de synthèse (figure 2.1). Une traduction "ligne à ligne" de la description en une représentation structurelle n'est plus possible, si bien qu'il est relativement difficile à un concepteur d'estimer *a priori* – c'est-à-dire sans connaître le modèle de représentation propre à l'outil qu'il utilise – l'impact des contraintes de synthèse sur les performances de l'architecture [2].

La question de la prédictibilité des performances au niveau comportemental rejoint ainsi la question de la flexibilité évoquée précédemment : la synthèse de haut niveau se distingue de la synthèse *RTL* par le fait que paramètres et contraintes n'affectent plus l'architecture de manière directe, mais par l'intermédiaire d'un modèle de représentation du comportement.

2.1.2 Vers une méthodologie de conception de composants virtuels comportementaux

Comme nous l'avons montré, les outils de synthèse de haut niveau existants ne sont pas adaptés à la spécification de composants réutilisables [2, 8, 80].

Le travail méthodologique que nous présentons dans ce document propose des solutions visant à résoudre les difficultés liées à la flexibilité, à l'interopérabilité et à la prédictibilité d'un composant virtuel décrit au niveau comportemental.

Dans les paragraphes qui précèdent, nous avons observé une convergence de ces trois problématiques vers la question du modèle de représentation sous-tendant une

description comportementale. Le cœur de notre démarche méthodologique consistera ainsi à définir formellement des modèles pour les différentes vues – algorithmique, système, comportementale, *RTL* – d'un composant virtuel spécifié au niveau comportemental. Les différents modèles retenus servent de support à la prise de décision en permettant à un concepteur de répartir efficacement les contraintes de performances sur les éléments de sa description et d'en mesurer l'impact [8].

Dans ce chapitre nous nous attacherons à définir un flot général de conception et d'intégration d'un composant virtuel de niveau comportemental. La question de l'interopérabilité des outils sur le plan syntaxique sera traitée dans le chapitre 3 en formalisant une syntaxe abstraite – présentée dans sous la forme d'un méta-langage symbolique – et un modèle formel de représentation qui délimiteront d'une part le domaine des instructions autorisées pour la description d'un comportement en vue de la synthèse de haut niveau, et serviront d'autre part de support à la traduction ligne à ligne d'une description comportementale d'un langage vers un autre. Un flot de "pré-synthèse" de haut niveau sera proposé, permettant de cibler une variété d'outils de synthèse à partir d'une description "universelle" du comportement d'un composant.

2.1.3 Travaux connexes

Les travaux menés en synthèse de haut niveau depuis une vingtaine d'année se sont essentiellement attachés à prospecter les algorithmes de sélection, allocation, ordonnancement et assignation en laissant de côté la question d'une formulation plus précise de la sémantique d'implantation des descriptions comportementales. La définition d'une méthodologie de réutilisation de composants virtuels par synthèse de haut niveau apparaît ainsi comme une problématique résolument nouvelle. Parmi les travaux ayant un lien avec cette problématique, on trouvera essentiellement trois types de démarches :

- Utiliser les outils de synthèse de haut niveau pour concevoir rapidement des composants virtuels qui seront distribués sous forme de blocs *soft*, *firm* ou *hard*.
- Etendre la notion de synthèse de haut niveau à la synthèse système en automatisant la sélection et l'intégration de composants virtuels matériels.
- Formaliser des modèles de représentation des descriptions comportementales pour consolider le lien avec la spécification au niveau système.

La synthèse de haut niveau pour la conception d'IP soft

Dans [73], une méthodologie de conception d'*IP soft* par synthèse de haut niveau est présentée. Cette méthodologie est fortement axée sur les fonctionnalités offertes par l'outil *Behavioral Compiler* de *Synopsys*. Essentiellement destinée aux applications orientées contrôle, elle se concentre sur le polymorphisme des interfaces de communication et ne s'intéresse pas à la personnalisation des aspects fonctionnels d'un composant.

Le point clé de cette approche est la séparation explicite du comportement *interne* (description de la fonctionnalité) et du comportement *externe* (interface) d'un composant décrit en *VHDL*. Cette séparation est obtenue en décrivant le protocole de communication sous forme de sous-programmes destinés à être appelés par le processus décrivant le comportement interne d'un composant. Ainsi, il est possible de fournir dif-

férents protocoles en rédigeant différentes bibliothèques de sous-programmes, et sans avoir à modifier la description du comportement interne.

Cette méthode se heurte cependant aux restrictions de style d'écriture imposées par *Behavioral Compiler* : l'interaction entre les sous-programmes de communication et la partie calculatoire du comportement nécessite l'utilisation d'un style d'écriture d'assez bas niveau.

La synthèse de haut niveau pour automatiser la réutilisation

Les travaux présentés dans [54] introduisent la réutilisation de composants virtuels dans le flot de synthèse de haut niveau proprement dit. Le principe est d'automatiser la sélection et l'allocation de composants fonctionnels complexes pour la synthèse de la partie matérielle d'un système.

A la différence des outils présentés dans la section 1.3, l'étape de sélection des ressources matérielles ne travaille donc plus sur des bibliothèques de composants de faible granularité – opérateurs arithmétiques – mais sur des bibliothèques de composants virtuels pré-définis. De tels composants ne sont plus purement combinatoires, mais possèdent chacun son propre protocole de communication. La synthèse d'une architecture à ce niveau d'abstraction s'accompagne par conséquent d'une synthèse automatique des interfaces entre composants complexes.

Ces techniques de synthèse sont implantées d'une part dans des outils universitaires fondés sur la méthodologie *SpecC* développée dans l'équipe de Daniel D. GAJSKI (UC Irvine), et d'autre part dans un outil commercial distribué par la société *Y Explorations Inc.* (*YXI*).

Modèles de représentation des descriptions comportementales

La définition d'un modèle de représentation commun et permettant de rendre accessibles au concepteur les informations nécessaires au choix des contraintes de synthèse est un préalable indispensable à une diffusion et une acceptation plus vaste des techniques de synthèse de haut niveau dans le milieu industriel.

Les approches classiques reposent sur des modèles de type *CDFG* qui n'établissent pas de lien direct entre le comportement décrit et le modèle *RTL* (*FSMD*) de l'architecture cible. Dans ce paragraphe, nous présentons deux approches qui consistent à unifier les modèles de représentation au niveau comportemental et au niveau transfert de registres sur la base de modèles *FSMD*.

Dans [52], les auteurs mettent en avant le fait qu'une partie des tâches de synthèse de haut niveau comme la sélection/allocation des ressources et l'assignation font partie des fonctionnalités offertes par les dernières générations d'outils de synthèse *RTL*. A ce titre, seule l'étape d'ordonnancement apparaît spécifique à la synthèse de haut niveau et une méthodologie d'utilisation rationnelle de ces outils doit ainsi se concentrer sur l'expression des contraintes de communication et de synchronisation d'un composant avec son environnement.

Le modèle de représentation intermédiaire au niveau comportemental proposé par l'équipe *System-Level Synthesis* du *TIMA* (Grenoble) est une machine à état fini de niveau système dont les états modélisent des points de synchronisation (instructions

wait en *VHDL*), et dont les transitions correspondent aux instructions à exécuter entre deux points de synchronisation.

L'ordonnancement d'une description comportementale revient ainsi à raffiner ce modèle en une *FSM* décrite au cycle près. Le modèle ordonnancé associe à chaque transition un ensemble d'opérations devant être exécutées en parallèle pendant un cycle d'horloge donné. Ce modèle contient suffisamment d'information pour être traité par un outil de synthèse *RTL*, ce dernier prenant en charge l'allocation des ressources matérielles et l'assignation des opérations aux ressources allouées.

Cette approche a été expérimentée dans l'environnement de synthèse de haut niveau *AMICAL* développé au *TIMA*.

Dans [58], le modèle de description comportementale envisagé est de type *SFSMD* (*Superstate Finite State Machine with Datapath*). Une *SFSMD* est formellement équivalente à une *FSMD*, mais autorise un plus haut niveau d'abstraction en associant à chaque état ou transition non plus des opérations s'exécutant en parallèle, mais des expressions arithmétiques de plus grande complexité dont l'ordonnancement n'est pas défini *a priori*.

Un état d'une *SFSMD* représente ainsi un *super-état* dont la durée peut éventuellement couvrir plusieurs périodes d'horloge. Lors de la synthèse de haut niveau, une *SFSMD* est raffinée en décomposant chaque *super-état* en autant d'états que de cycles d'horloges nécessaires à l'ordonnancement des expressions arithmétiques qui lui sont associées.

A la différence du modèle développé au *TIMA* [52], le style de description comportementale présenté dans [58] ne fait pas apparaître les contraintes d'entrée/sortie : la répartition des instructions du comportement sur les *super-états* n'est donc plus liée à des points de synchronisation mais à des choix de partitionnement de la description qui sont entièrement du ressort du concepteur.

Ces deux approches sont peu adaptées à la modélisation des descriptions supportées par les outils de synthèse de haut niveau commerciaux comme *Monet*, *Behavioral Compiler*, *CoCentric SystemC Compiler* et *Get2Chip Architectural Compiler* : ces outils nécessitent en effet un modèle permettant d'une part de prendre en compte les structures de contrôle – par exemple les boucles non déroulées – dans la répartition des contraintes de temps, et d'autre part de représenter un comportement avec entrées/sorties au cycle près sans pour autant spécifier le détail de l'ordonnancement des calculs. Dans le chapitre 3, nous proposerons un modèle de type *SFSMD* hiérarchique répondant à ces exigences.

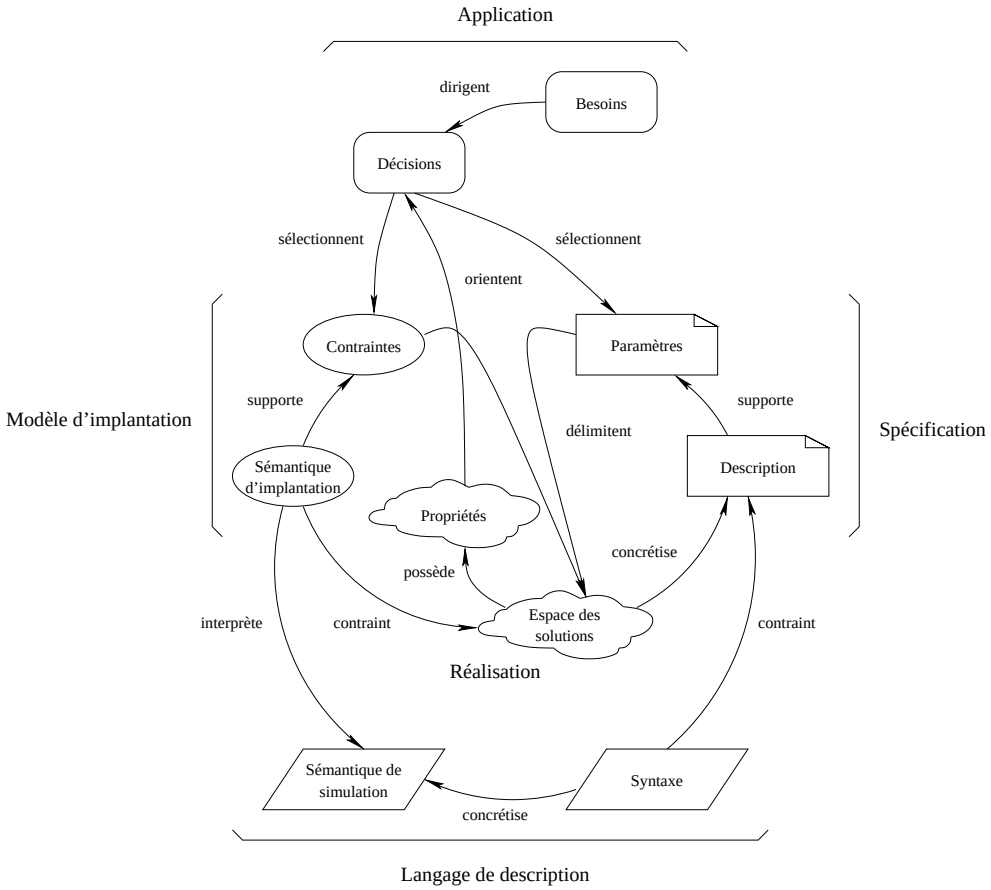
2.2 Définitions préliminaires

2.2.1 Attributs d'un composant virtuel comportemental

Un composant virtuel combine des informations provenant de quatre sources principales :

- les *applications cibles*, représentées par différentes classes de besoins ;
- le *modèle d'implantation*, conditionnant l'interprétation d'un comportement en termes matériels ;

FIG. 2.3 – Attributs d'un composant virtuel comportemental



- le *langage de description*, conditionnant le format de la spécification ;
- la *spécification* proprement dite, qui décrit le composant à un niveau d'abstraction plus ou moins élevé.

L'ensemble de ces informations définit un espace de solutions architecturales d'autant plus étroit que les informations fournies sont détaillées. La figure 2.3 représente les relations entre ces différents éléments.

Application – Le domaine applicatif ciblé par un composant virtuel se caractérise par un ensemble de besoins relevant de différents niveaux d'abstraction. Un créateur de composant s'attachera à répondre aux besoins *potentiels* des utilisateurs en définissant : (1) l'ensemble des *paramètres* permettant de personnaliser le comportement d'un composant ; (2) l'ensemble des *contraintes* permettant de personnaliser les performances de l'architecture ; (3) les *domaines de valeurs* autorisées pour les paramètres et contraintes. La définition des paramètres et contraintes est un processus incrémental qui n'est finalisé que lorsque le composant virtuel est prêt à être délivré.

Un intégrateur de composant spécifiera les valeurs de paramètres et contraintes de telle sorte que les *propriétés* des solutions architecturales correspondantes satisfassent ses besoins propres.

Spécification – La spécification d'un composant recouvre deux types d'information : (1) elle contient tout d'abord la *description* du composant à un niveau d'abstraction donné ; (2) elle contient également une spécification des *paramètres* permettant de personnaliser le composant.

La description peut être un simple modèle *simulable* de haut niveau à raffiner manuellement, ou un modèle *synthétisable* destiné à un outil de synthèse automatique. L'ensemble des paramètres et leurs plages de valeurs dépendent des besoins potentiels auquel le composant est susceptible de répondre et du niveau d'abstraction de la description.

Langage de description – La description du composant à un niveau d'abstraction donné est rédigée dans un langage. Un tel langage se caractérise par un *modèle sémantique* définissant formellement les propriétés exprimables – réactivité, hiérarchie, concurrence, synchronisation, communication. La *syntaxe* d'un langage associe une forme concrète – lisible par un être humain et/ou interprétable par un outil de *CAO* – à son modèle sémantique.

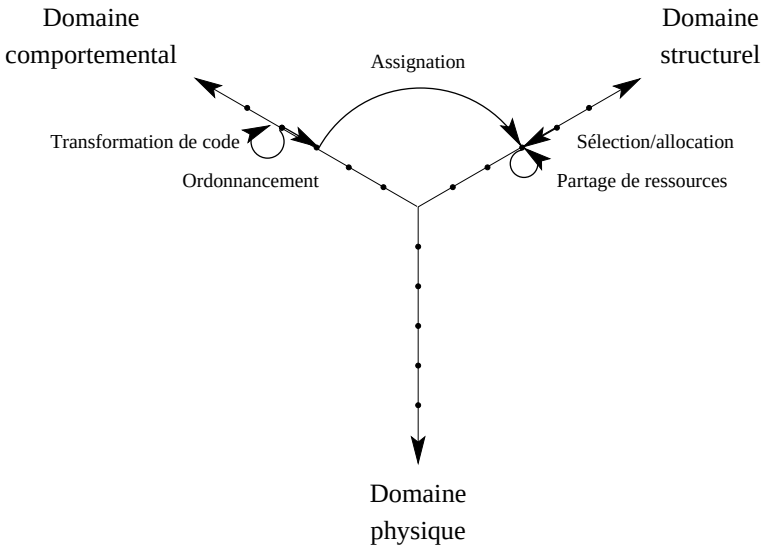
Modèle d'implantation – La sémantique de simulation du langage utilisé n'est pas toujours suffisante pour supporter les décisions d'implantation. Comme nous l'avons montré plus haut, l'interopérabilité, la flexibilité et la prédictibilité d'un composant virtuel de niveau comportemental nécessitent la mise en place d'un modèle de représentation intermédiaire qui permette d'associer par raffinements successifs des équivalents matériels au comportement décrit.

Réalisation – Du point de vue du créateur d'un composant, une réalisation est un espace de solutions architecturales satisfaisant un modèle d'implantation donné et exhibant différentes propriétés utiles à la prise de décision. Cet espace de solutions est délimité par l'ensemble des valeurs de paramètres et contraintes autorisées.

Du point de vue de l'utilisateur d'un composant, une réalisation est un ensemble de solutions dont les propriétés satisfont ses besoins propres.

Propriétés et paramètres – Les propriétés et paramètres d'un composant virtuel peuvent être classés d'une part en fonction du niveau d'abstraction auquel ils interviennent, et d'autre part en fonction du type d'influence qu'ils exercent sur la spécification. On distinguera quatre grandes classes de propriétés et de paramètres : (1) ceux liés à la *fonctionnalité* du composant ; (2) ceux liés à la *complexité* du composant ou de son modèle de représentation ; (3) ceux liés à la *topologie* du composant ou de son modèle de représentation ; (4) ceux liés aux *aspects temporels* du comportement interne et externe du composant.

FIG. 2.4 – La synthèse de haut niveau dans le diagramme en Y



2.2.2 Niveaux d'abstraction et hiérarchie des spécifications

Dans le diagramme en *Y*, la synthèse de haut niveau sera vue comme le passage d'une représentation *algorithmique* du *comportement* d'un composant à une représentation *structurelle* au niveau *transfert de registres*. Un flot typique de synthèse de haut niveau combine ainsi les étapes suivantes :

- *Optimisation* dans le domaine *comportemental* au niveau *algorithmique* : élimination de code mort, propagation des constantes, extraction des sous-expressions communes, déroulage des boucles, mise en ligne des appels de sous-programmes, *etc.*
- *Raffinement* dans le domaine *structurel* : c'est l'étape de sélection/allocation des ressources matériel du chemin de données.
- *Raffinement* dans le domaine *comportemental* : c'est l'étape d'ordonnancement où le comportement décrit sous forme algorithmique est réparti sur des pas de contrôle.
- *Synthèse* : c'est l'étape d'assignation au cours de laquelle le comportement est projeté sur les ressources matérielles allouées.
- *Optimisation* dans le domaine *structurel* au niveau *transfert de registres* : le nombre de ressources matérielles est minimisé en partageant des opérateurs et les registres.

La représentation proposée par le diagramme en *Y* ne permet cependant pas d'exprimer une certain nombre de propriétés d'une description dédiée à la synthèse de haut niveau. Notamment, dans un flot de synthèse de haut niveau, il peut être judicieux d'opérer une distinction claire entre les notions de description *algorithmique* et de description *comportementale* :

- Une description *algorithmique* propose une vision *transformationnelle* de la fonc-

TAB. 2.1 – Classes de spécifications d'un composant virtuel de niveau comportemental

Niveau de spécification	Informations apportées	Question
Applicatif	Utilité du composant	"pourquoi faire ?"
Fonctionnel	Fonction réalisée	"quoi ?"
Algorithmique	Détail des calculs	"comment ?"
Comportemental	Vue temporelle externe	"quand ?"
Transfert de registres	Vue temporelle interne	"quand ?"
Structurel	Ressources matérielles	"qui ?"

tion réalisée par un circuit : la consommation des données d'entrée, le traitement de ces données et la production des données de sortie ne font pas intervenir de notion de temps.
- Une description *comportementale* ajoute une propriété de *réactivité* à l'algorithme en modélisant le temps, la synchronisation et la communication du composant avec son environnement avec une plus ou moins grande finesse de détail.

Dans les paragraphes qui suivent, nous dressons un inventaire des types de spécification rédigées par le concepteur ou générées automatiquement par les outils tout au long du flot de conception et d'intégration d'un composant virtuel comportemental. Nous schématisons la résolution des détails d'implantation portés par chaque spécification à l'aide de la taxonomie *RASSP/VSIA* présentée en figure 1.3, page 13.

La présentation des différentes spécifications suit un flot descendant dans lequel chaque nouvelle spécification hérite des informations – ou détails d'implantation – portées par la précédente, et lui ajoute de nouvelles informations. La nature de ces nouvelles informations peut être caractérisées par une question relative au composant, chaque étape du flot et chaque décision d'implantation apportant une partie de la réponse. Le tableau 2.1 résume les classes de spécifications, du niveau applicatif jusqu'au niveau structurel.

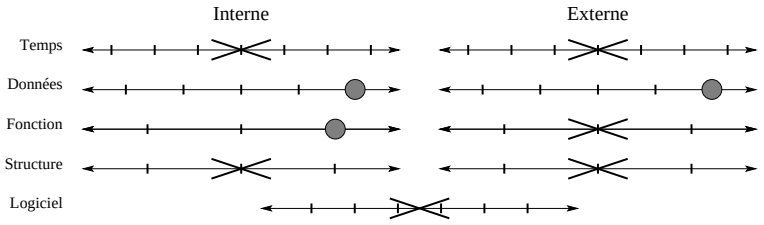
Spécification au niveau applicatif – Une spécification au niveau *applicatif* s'intéresse à la classe d'applications adressée par un composant et aux rôles qu'il est susceptible de jouer dans ces applications.

Ainsi, une telle spécification fournit une vue essentiellement qualitative du composant et de ses contextes d'utilisation. Les informations relatives à un composant à ce niveau précisent son utilité (question "pourquoi faire ?") indépendamment de son fonctionnement.

Spécification fonctionnelle – Une spécification *fonctionnelle* décrit la fonction – au sens mathématique du terme – effectivement réalisée par le composant (question "quoi ?")

Une fonction dédiée au traitement de données sera représentée comme une transformation d'un vecteur de données d'entrée en un vecteur de données de sortie. Le choix de la fonction et l'ajustement de ses éventuels paramètres sont conditionnés

FIG. 2.5 – Spécification fonctionnelle



par des propriétés quantitatives liées aux besoins fonctionnels des applications cibles. Dans le cadre d'une chaîne de compression d'images, ce sera par exemple le compromis taux de compression/qualité de reconstruction qui servira de référence aux choix d'un décorrélateur et d'un quantificateur.

Le type des données manipulées est exprimé avec une précision dépendant de la fonctionnalité :

- Une fonction de traitement du signal comme la transformation de Fourier pourra être décrite en nombre réels ou en arithmétique complexe.
- Une fonction de codage pourra manipuler des données d'entrée de type "symbole" et produire des données de sortie binaires, les données intermédiaires (fréquence d'occurrence des symboles, par exemple) pouvant être des nombre rationnels.

Spécification algorithmique – Une spécification *algorithmique* s'intéresse à la manière (question "comment ?") dont les calculs décrits mathématiquement seront effectivement réalisés.

Il est en effet fréquent qu'une même fonction puisse être calculée en appliquant une variété d'algorithmes plus ou moins complexes. Le cas typique est celui de la transformation de Fourier discrète, pour laquelle différents algorithmes de calcul rapide ont été proposés (avec décimation en temps ou décimation en fréquence, *radix*-2 ou *radix*-4, *etc.*).

Le choix d'un algorithme sera ici conditionné par la comparaison de propriétés quantitatives mesurant entre autres leur complexité calculatoire, leur parallélisme potentiel, leur régularité.

Une spécification algorithmique *exacte au bit près* est obtenue par sélection de nouveaux types de données, de format bien défini, en remplacement des types employés dans la spécification algorithmique.

Spécification comportementale – Une spécification *comportementale* introduit une notion de temps (question "quand ?") et de réactivité à la représentation algorithmique d'un composant.

Sans aller jusqu'à préciser la date exacte à laquelle chaque opération est exécutée, il est possible à ce stade de décrire au cycle près le modèle de comportement aux entrées/sorties du composant. Le niveau comportemental se décline sur deux sous-niveaux :

FIG. 2.6 – Spécification algorithmique en précision infinie

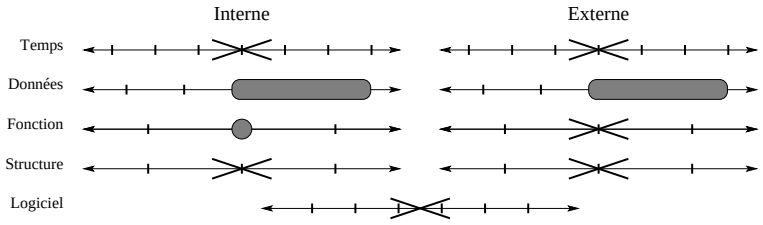
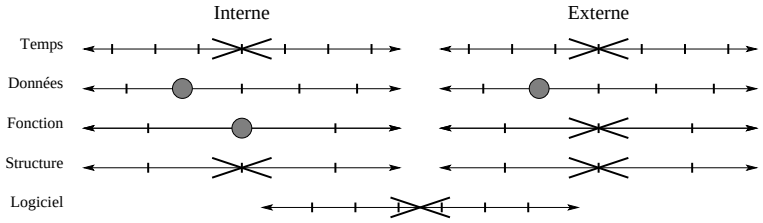


FIG. 2.7 – Spécification algorithmique en précision fixée



- Le niveau "transaction" fait communiquer le composant avec son environnement par le biais de primitives de communication abstraites.
- Le niveau "cycle" implique la spécification exacte au cycle près de la date de lecture/écriture de chaque donnée consommée ou produite pendant une itération du comportement.

Spécification au niveau transfert de registres – Une spécification au niveau transfert de registre décrit au cycle près la répartition temporelle du comportement, entrées/sorties et calculs.

En synthèse de haut niveau, une telle spécification est obtenue après l'étape d'ordonnancement. Le détail des ressources matérielles – opérateurs, registres – réalisant chaque opération ne sera défini que lors de l'étape d'assignation.

FIG. 2.8 – Spécification comportementale au niveau transaction

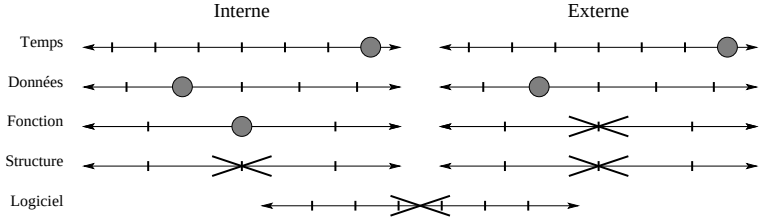


FIG. 2.9 – Spécification comportementale au cycle près

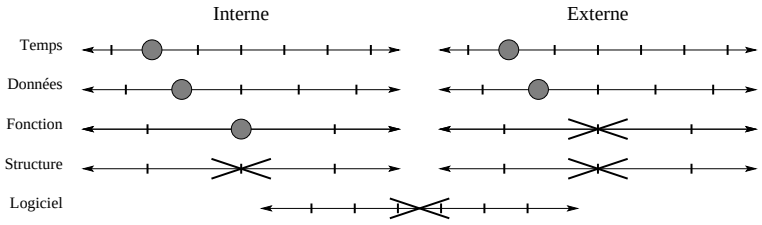
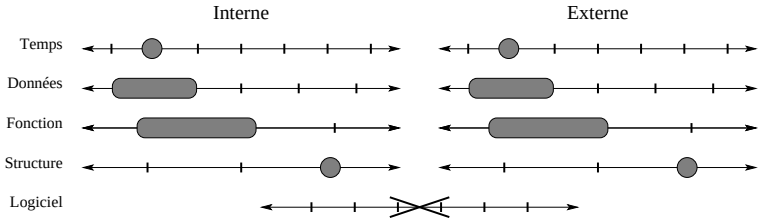


FIG. 2.10 – Spécification au niveau transfert de registres

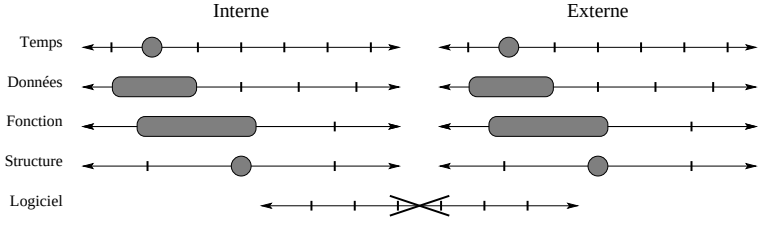


Une spécification *structurelle* au niveau transfert de registre fournit une vue structurelle des éléments matériels implémentant le comportement du composant. Elle se compose généralement d'un chemin de donnée piloté par un contrôleur.

Le chemin de données regroupe les ressources matérielles (question "qui ?") réalisant le traitement des données manipulées par l'algorithme. On y trouvera : (1) des opérateurs arithmétiques mono-fonction (additionneurs, soustracteurs, multiplieurs, comparateurs) ou multi-fonctions (unités arithmétiques et logiques) ; (2) des éléments de mémorisation (registres, *RAM*) ; (3) des éléments d'aiguillage des données entre opérateurs et registres (multiplexeurs, démultiplexeurs, *tri-states*).

Le contrôleur est généralement décrit comme une machine à état fini qui active au moment opportun les éléments du chemin de donnée.

FIG. 2.11 – Spécification structurelle au niveau transfert de registres



2.3 Spécification et raffinement d'un composant virtuel comportemental

Dans cette section, nous détaillons les étapes du flot de conception d'un composant virtuel comportemental. Ce flot repose sur une démarche descendante au cours de laquelle chaque étape fait passer la spécification d'un niveau d'abstraction au niveau inférieur. Au cours du raffinement de la spécification d'un composant, chaque étape enrichit l'ensemble des paramètres et contraintes permettant de personnaliser les propriétés du composant à différents niveaux d'abstraction [80–82].

2.3.1 Modélisation au niveau fonctionnel

Au niveau fonctionnel, un composant est décrit sous la forme d'une relation mathématique entre un ensemble de données d'entrée et un ensemble de données de sortie. Une telle relation peut être modélisée dans un environnement comme *Matlab*, fournissant un modèle de référence pour la vérification des spécifications ultérieures.

La modélisation au niveau fonctionnel nécessite notamment de définir la structure des données d'entrée/sortie. Typiquement, ces données seront organisées sous forme de vecteurs ou de tableaux multidimensionnels. Le choix du nombre de dimensions et de la répartition des données est lié à la nature des informations traitées – échantillons d'un signal audio, pixels d'une image, *etc.*

Selon les applications, les coordonnées relatives à chaque dimension pourront être bornées – c'est typiquement le cas des images qui présentent généralement des frontières bien définies – ou non bornées suivant l'une des dimensions lorsque la fonction à implanter réalise un traitement sur des données acquises de manière ininterrompue au cours du temps.

Propriétés et paramètres de niveau fonctionnel – Si la nature qualitative des données traitées (signal audio, image fixe, vidéo, scène *3D*) relève des propriétés de niveau applicatif, la manière dont ces données sont structurées est bien une propriété fonctionnelle. Nous retiendrons essentiellement deux propriétés conditionnant la *topologie* des données : (1) le *nombre de dimensions* des tableaux contenant les données ; (2) le caractère *borné/non borné* des coordonnées relatives à chaque dimension.

Les paramètres de complexité au niveau fonctionnel concernent la quantité de données traitées, c'est-à-dire le nombre d'échantillons contenus dans un tableau de données d'entrée/sortie. Ce seront par exemple la largeur et la hauteur, en pixels, d'une image.

Les autres paramètres au niveau fonctionnel seront spécifiques à chaque fonction et seront classés *a priori* parmi les paramètres liés à la fonctionnalité du composant. Dans le cas d'une fonction de filtrage, par exemple, ce pourront être sa/ses fréquence(s) de coupure et sa sélectivité.

Exemple

Dans ce chapitre, nous illustrerons les différents niveaux de spécification à l'aide d'une fonction de traitement de signaux monodimensionnels. Cette fonction réalise

un sur-échantillonnage d'un signal avec un facteur K, suivi d'une interpolation polynomiale d'ordre $N-1$ (N pair). Le vecteur e contenant les échantillons d'entrée est borné à gauche, et non borné à droite.

Dans le domaine continu, une telle interpolation peut être réalisée au moyen des polynômes de Lagrange calculés sur N points de coordonnées x_0 à x_{N-1} du signal e :

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{N-1} \frac{x - x_j}{x_i - x_j} \tag{2.1}$$

$$y(x) = \sum_{i=0}^{N-1} L_i(x) e(x_i) \tag{2.2}$$

Dans le domaine discret, nous définissons tout d'abord le signal sur-échantillonné e_s :

$$e_s(n) = \begin{cases} e(\frac{n}{K}) & \text{si} \quad n \equiv 0[K] \\ 0 & \text{sinon} \end{cases} \tag{2.3}$$

Pour un échantillon d'indice n de e_s, nous définissons les coordonnées x_i des N points de référence situés de part et d'autre de n :

$$x_i = \left(\lfloor \frac{n}{K} \rfloor - \frac{N}{2} + i + 1 \right) \times K \tag{2.4}$$

L'interpolation au point n peut ainsi être calculée à l'aide des polynômes de Lagrange de la manière suivante :

$$y(n) = \sum_{i=0}^{N-1} L_i(n) e_s(x_i) \tag{2.5}$$

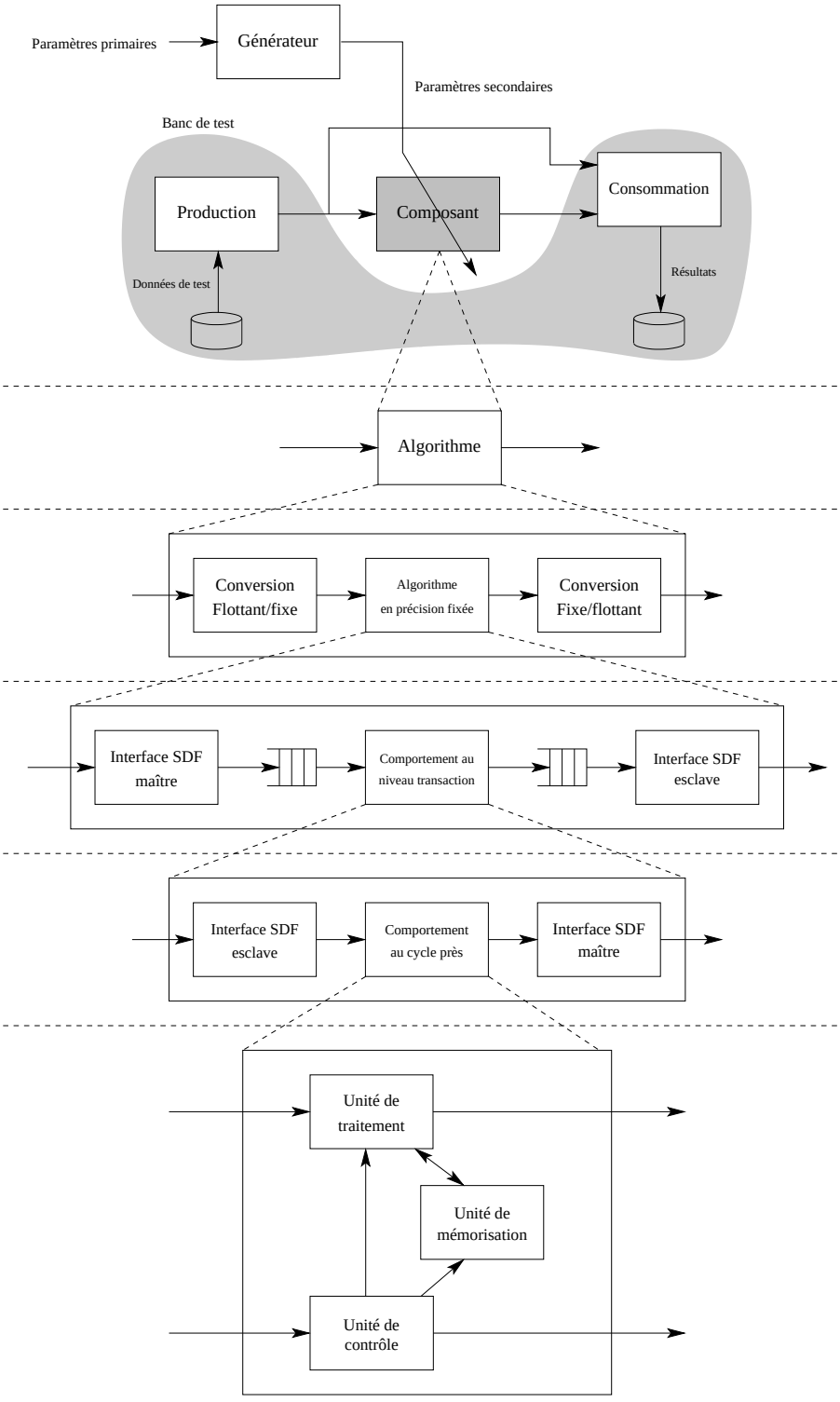
2.3.2 Sélection d'algorithmes

Une même fonction mathématique peut être implantée à l'aide d'une variété d'algorithmes présentant différents degrés de complexité et de parallélisme exploitable. Un algorithme peut être spécifié dans un langage de programmation impératif tel le langage *C*, sous la forme d'un simple sous-programme prenant en entrée un tableau de données à traiter et fournissant en sortie un tableau de données résultats.

Dans un langage de description de systèmes comme *SpecC* ou *SystemC*, il est recommandé de décrire séparément la partie algorithme et la partie banc de test sous forme de *behaviors* ou *modules* interconnectés (figure 2.12). Une telle approche permet notamment de réutiliser le même banc de test tout au long du flot de conception en rendant interchangeables les descriptions successives du composant obtenues au fil des étapes de raffinement.

Paramètres de niveau algorithmique – On distinguera deux types de paramètres de niveau algorithmique : (1) les paramètres permettant de choisir parmi un ensemble d'algorithmes fonctionnellement équivalents ; (2) les paramètres spécifiques à chaque algorithme. Dans le cas d'une fonction de filtrage spécifiée au niveau fonctionnel par un gabarit idéal, il sera par exemple possible d'envisager une implantation

FIG. 2.12 – Raffinement d'un composant virtuel comportemental



à base de filtres à réponse impulsionnelle finie (*FIR*) ou infinie (*IIR*). Chacune de ces deux implantations nécessite de spécifier le nombre de cœfficients du filtre et la valeur de ces cœfficients.

Un lien fort peut exister entre les paramètres de niveau fonctionnel et les paramètres algorithmiques. Dans l'exemple de la fonction de filtrage, le choix de la fréquence de coupure et de l'algorithme de filtrage peut imposer le nombre et la valeurs des cœfficients. A ce stade, nous devons ainsi distinguer deux types de paramètres : (1) les paramètres *primaires* seront directement accessibles à l'utilisateur du composant virtuel ; (2) les paramètres *secondaires* seront déduits des précédents.

Cette distinction s'appliquera à tous les niveaux d'abstraction. La génération automatique des valeurs de paramètres secondaires permet à l'utilisateur de travailler avec un jeu de paramètres dont la signification lui est directement accessible sans qu'une connaissance des détails d'implantation soit requise.

Propriétés de niveau algorithmique – La notion de *complexité* d'un algorithme prend un sens différent selon que le type d'implantation ciblé est matériel ou logiciel.

Dans une implantation logicielle, les notions de quantité de calcul et de temps de calcul sont étroitement liées du fait que le parallélisme d'un processeur est limité. Au contraire, dans une implantation matérielle, le parallélisme exploitable est *a priori* illimité et le temps de calcul ne dépend que des contraintes d'implantation. Nous mesurerons ainsi la complexité d'un algorithme dominé par le calcul comme le nombre moyen d'opérations arithmétiques par donnée d'entrée ou de sortie. Les propriétés temporelles seront mesurées en termes de *latence* minimale entre une donnée d'entrée et les données de sortie qui en dépendent, cette latence pouvant être exprimée en nombre d'opérations traversées.

Dans le domaine des propriétés de topologie, nous nous intéresserons à la *régularité* d'un algorithme, c'est-à-dire le degré selon lequel l'ensemble de l'algorithme peut être décomposé en itérations successives d'un motif de calcul plus ou moins complexe. Cette notion est étroitement liée à la structure de boucles imbriquées de l'algorithme, qui reflète les répétitions multiples d'une même séquence d'instruction.

Modèle de représentation – Un algorithme décrit un calcul de manière purement transformationnelle. D'un point de vue externe, il ne fait ainsi intervenir aucune notion de temps, c'est-à-dire que les données qu'il utilise sont supposées être toutes disponibles au moment où le calcul commence. D'un point de vue interne, la description de l'algorithme implique une relation d'ordre partiel entre les calculs.

Afin de permettre l'extraction des propriétés de complexité et de topologie, le détail des opérations et de leurs dépendances doit être représenté. Nous privilégions ainsi un modèle de type graphe flot de données (*DFG*) ou graphe flot de données et de contrôle (*CDFG*).

Exemple

L'algorithme d'interpolation repose sur l'observation que les cœfficients $L_i(n)$ présentent une périodicité de K échantillons et peuvent donc être pré-calculés indépendamment du nombre d'échantillons à traiter.

Ici, l'algorithme possédera deux paramètres primaires : le facteur d'échantillonnage (K) et l'ordre du polynôme interpolant ($N-1$). Les paramètres secondaires sont les cœfficients $L_i(n)$, calculés à partir de K et N.

Un exemple de description est donné ci-après dans le langage *SystemC*. Le module *Interpolation* décrit possède un port d'entrée *e_in* et un port de sortie *y_out* de types vecteur. Le constructeur du module reçoit les paramètres N et K et les affecte à des variables internes avant de les vérifier à l'aide de la méthode *check* et de calculer les paramètres secondaires $L_i(n)$ à l'aide de la méthode *generate*. La méthode *run* décrit l'algorithme d'interpolation proprement dit. Le symbole *INFINITY* modélise le fait que le nombre d'échantillons à traiter, *a priori* infini, doit cependant être spécifié pour fixer un temps de simulation fini.

2.3.3 Raffinement des formats de données

Le choix des format des données aura généralement un impact sur des propriétés de niveau fonctionnel et de niveau *RTL* structurel : au niveau fonctionnel, l'arithmétique en virgule fixe peut introduire un bruit de calcul dont le niveau devra être réglable par l'utilisateur [60]. Au niveau *RTL* structurel, les formats de données influencent la surface et le temps de traversée des opérateurs arithmétiques ainsi que la quantité de mémoire nécessaire au traitement.

Une spécification algorithmique en précision fixée permet de mesurer tôt dans le flot de conception l'impact du choix des formats de données sur la qualité du résultat au niveau fonctionnel. En fonction du niveau de bruit de calcul toléré, il sera possible de fixer le nombre minimum de bits autorisé pour les données. Ce nombre de bits étant choisi, il tient lieu de contrainte de synthèse lors de l'étape de sélection/allocation du flot de synthèse de haut niveau.

Dans la spécification algorithmique, les types de données flottants doivent être remplacés par des types entiers ou virgule fixe dont les caractéristiques peuvent être soit figées, soit paramétrables. A la différence des langages de description de matériel et des langages de programmation, les langages de description de systèmes comme *SystemC* fournissent des types de données en virgule fixe. Dans une description du type de celle présentée dans le paragraphe précédent, la réutilisation du banc de test mis en place pour l'algorithme en virgule flottante nécessite l'insertion d'étages d'adaptation des formats de données en entrée et en sortie de l'algorithme en précision fixée.

L'analyse de l'algorithme permet d'estimer la propagation des erreurs de calcul et de dimensionner chaque donnée afin d'éviter les débordements en sortie des opérateurs arithmétiques [60]. Il existe des outils comme *CoCentric Fixed Point Designer* de *Synopsys* qui guident le concepteur dans le choix des formats de données [124].

2.3.4 Extraction d'un motif répétitif de calcul

Une spécification comportementale au niveau transaction modélise l'algorithme sous la forme d'un motif répétitif de calcul qui communique avec son environnement en échangeant des données par blocs d'échantillons de taille constante. Plusieurs étapes d'analyse et de transformation de la description sont nécessaires à l'extraction d'un

Listing 2.1 – Algorithme d'interpolation polynomiale en SystemC

```
SC_MODULE(Interpolation) {
      sc_port<VectorRead_if> e_in;
      sc_port<VectorWrite_if> y_out;

      int N, K;
      float**L;

      Interpolation(int K_par, int N_par) {
            N = N_par;
            K = K_par;

            check();
            generate();
            SC_METHOD(run);
      }

      void check() {
        if(N%2 != 0)
          throw(...);       // Erreur !
      }

      void generate() {
        // Calcul des L[i][n]
        ...
      }

      void run() {
        for(n=0;n<INFINITY;n++)
          if(n%K == 0)
            y_out->write(n,e_in->read(n/K));
          else {
            tmp = 0;
            for(i=0;i<N;i++) {
              x = n/K-N/2+i+1;
              if((x>=0)&&(x<INFINITY))
                tmp += L[i][n%K] * e_in->read(x);
            }
            y_out->write(n,tmp);
          }
      }
    }
}
```

tel motif. Ces étapes sont formalisées en annexe B. Nous les présentons ici dans leurs grandes lignes :

Algorithme avec affectation unique – L'écriture d'un algorithme avec affectation unique consiste à assurer l'identité entre chaque variable ou élément de tableau figurant dans la description et chaque donnée manipulée, ce qui permet une analyse plus aisée des dépendances de données.

Un algorithme présentant des affectations multiples d'une même variable peut être réécrit en procédant à un renommage des variables concernées, ou à une indexation si une même variable est affectée à chaque itération d'une boucle. Une telle indexation est illustrée dans le cas de l'algorithme d'interpolation par le listing 2.2 pour la variable *tmp*. Les techniques d'indexation de variables scalaires, et de réindexation de variables tableaux, permettant d'assurer la propriété d'affectation unique sont présentées dans [55].

Graphe de dépendances – L'indexation des données traitées au cœur d'un hiérarchie de boucles imbriquées permet de projeter les dépendances de données dans un espace multidimensionnel où la topologie de l'algorithme peut être visualisée. Les graphes de dépendances polyédraux (*PDG*) [24] formalisent ce type de représentation et sont utilisés notamment en synthèse d'architectures régulières avec l'outil *MMAlpha* [50, 51].

La figure 2.13 représente un extrait du graphe de dépendances de l'algorithme d'interpolation pour $K = 3$ et $N = 4$. Les points noirs représentent les données et les bulles blanches représentent des opérations ou groupes d'opérations travaillant sur ces donnés.

Listing 2.2 – Algorithme d'interpolation polynomiale avec affectation unique

```
void run() {
  for(n=0;n<INFINITY;n++)
    if(n%K == 0)
      y_out->write(n,e_in->read(n/K));
    else {
      tmp[0][n] = 0;
      for(i=0;i<N;i++) {
        x = n/K-N/2+i+1;
        if((x>=0)&&(x<INFINITY))
          tmp[i+1][n] = tmp[i][n]+L[i][n%K]*e_in->read(x);
      }
      y_out->write(n,tmp[N][n]);
    }
}
```

Modélisation du temps et causalité – La notion de temps est introduite en faisant l'hypothèse qu'il existe une relation d'ordre partiel sur les ensembles de données d'entrée/sortie. Les données sont éventuellement réindexées de manière à faire apparaître cette relation d'ordre.

FIG. 2.13 – Extrait du graphe de dépendances de l'algorithme d'interpolation

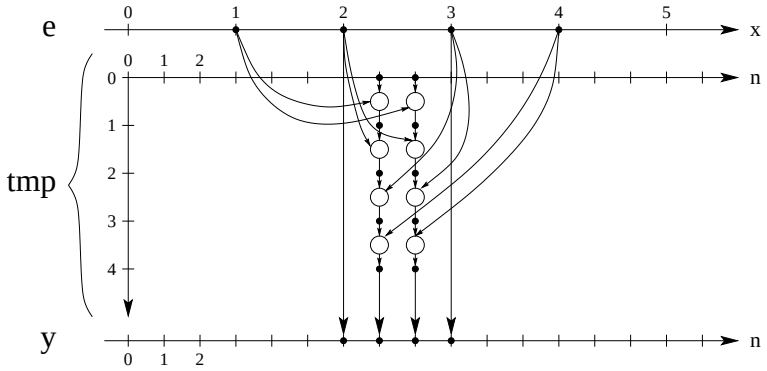
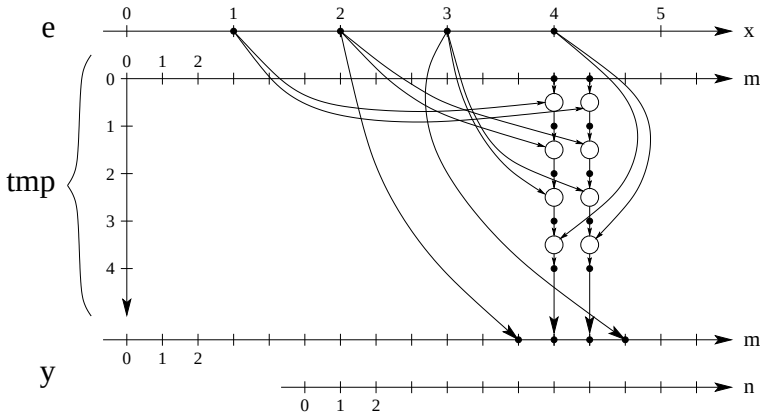


FIG. 2.14 – Graphe de dépendances causal de l'algorithme d'interpolation



A ce stade, le concepteur contraint les ordres de parcours possibles des données sans spécifier l'intervalle de temps – supposé positif ou nul – qui sépare deux données successives. Le respect de la causalité entre données et opérations peut nécessiter une réindexation des opérations elles-mêmes. Par exemple, si nous supposons que les données d'entrée/sortie de l'algorithme d'interpolation sont respectivement parcourues dans le sens croissant des indices x et n – sans préjuger de la durée positive ou nulle s'écoulant entre deux données d'indices successifs –, l'indice n n'est pas approprié à modéliser l'écoulement du temps en respectant la causalité. La figure 2.14 présente une version causale de ce graphe, reposant sur la transformation d'indice suivante : $m \leftarrow n + \frac{NK}{2} - 1$.

Motif répétitif de calcul – Un motif de calcul sera défini comme un sous-graphe dont la répétition périodique le long des "axes temporels" engendre tout ou partie du graphe de dépendances.

Cette périodicité définit une succession d'intervalles de temps disjoints pendant

TAB. 2.2 – Algorithme d'interpolation : propriétés des trois motifs présentés

Motif	Nombre d'opérations	Parallélisme maximum	Transferts			Durée de vie moy.	
			e	tmp	y	e	tmp
(a)	8	2	4	0	3	3	0
(b)	16	4	5	0	6	1,5	0
(c)	8	8	4	6	3	3	1

lesquels le processus de traitement itère la même séquence de calculs. Le choix du sous-graphe doit respecter la causalité des calculs, c'est-à-dire que les données intermédiaires nécessaires au calcul de l'itération courante doivent avoir été calculées lors d'itérations précédentes du motif.

Modèle de représentation – Chaque nouvelle itération du processus consomme une quantité constante de données d'entrée et produit une quantité constante de données de sortie. A ce stade de la conception, le modèle de communication du composant avec son environnement est de type flot de données synchrone *SDF*, c'est-à-dire que le déclenchement d'une itération du processus est conditionné par la présence d'une quantité minimale, constante, de données à ses entrées.

Une implantation possible dans un langage de description de systèmes fera usage de canaux abstraits modélisés comme des *FIFO* de taille infinie (lecture bloquante, écriture non bloquante).

Propriétés et paramètres – Le choix du motif permet d'ajuster des propriétés de complexité et de temps :

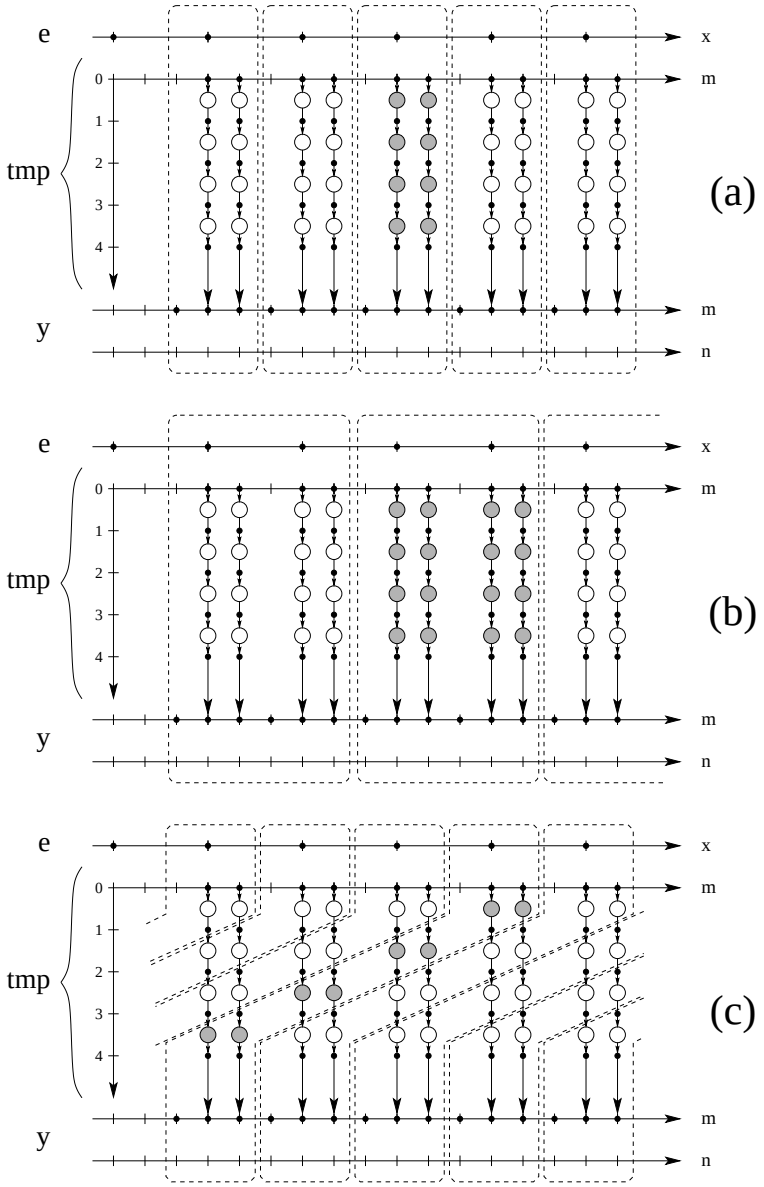
- *nombre d'opérations par itération* : ce nombre est proportionnel à la *taille* du motif, *i.e.* au *nombre de périodes* du graphe de dépendances y figurant.
- *parallélisme* exploitable au niveau opération, à l'intérieur d'une itération du motif.
- nombre de *transferts* de données entre l'itération courante, l'environnement et les itérations précédentes.
- *durée de vie* des données, c'est-à-dire le nombre d'itérations du motif s'écoulant entre la production et la dernière utilisation de chaque donnée.

Plusieurs exemples de motifs sont représentés en figure 2.15 et leurs propriétés sont présentées dans le tableau 2.2 pour $K = 3$ et $N = 4$. L'analyse de la quantité de transferts de données par itération et de la durée de vie de ces données permet de choisir le type de ressources de mémorisation : typiquement, les données à durée de vie courte seront mises en registres tandis que les données à durée de vie longue seront stockées en *RAM*.

2.3.5 Insertion des contraintes de temps et d'entrée/sortie

La synthèse d'une architecture *RTL* à partir d'un modèle de comportement au niveau transaction nécessite de spécifier des contraintes de temps et d'entrée / sortie

FIG. 2.15 – Différents motifs de traitement pour l'algorithme d'interpolation



afin de permettre aux outils de synthèse de haut niveau d'ordonnancer les opérations en fonction de la disponibilité des données [81].

Dans les outils de synthèse de haut niveau supportant l'expression de contraintes de temps et d'entrée/sortie au cycle près, ces contraintes peuvent être soit spécifiées directement dans le code comportemental (*Monet*, *Behavioral Compiler* et *CoCentric SystemC Compiler* en mode *cycle-fixed* ; *Get2Chip Architectural Compiler*), soit exprimées séparément dans les directives de synthèse (version de *GAUT* [61] en cours de développement au *LESTER* [116], projet "*Tsunami*" de *Mentor Graphics* [65]).

Dans le premier cas, nous aurons affaire à une spécification comportementale dite *exécutable* au cycle près. Une telle spécification est particulièrement utile pour la vérification du respect des contraintes de temps au niveau *RTL* lors de l'intégration d'un composant virtuel dans un système.

Paramètres et propriétés – A ce niveau de spécification, les paramètres permettent de personnaliser la date au cycle près des entrées/sorties et la durée d'exécution d'une itération du motif de calcul.

Ces paramètres se traduiront par un degré plus ou moins élevé de parallélisme d'entrée/sortie, fixant une taille minimale des ports d'entrée/sortie.

2.3.6 Décomposition structurelle d'un motif

Lors de la synthèse de haut niveau, le modèle au cycle près d'un composant virtuel comportemental sera décomposé en trois unités [81] : (1) l'unité de *traitement* décrit la partie purement calculatoire du motif de calcul ; (2) l'unité de *mémorisation* modélise la structure de stockage des données intermédiaires à durée de vie longue ; (3) l'unité de *contrôle* modélise le déroulement temporel du calcul sous la forme d'une machine à état fini.

L'action de l'unité de contrôle se situe à deux niveaux. Au niveau *cycle*, elle pilote les transferts de données entre opérateurs arithmétiques de l'unité de traitement au cours d'une itération du motif. Au niveau *itération*, elle permet de modéliser les exécutions successives du motif comme autant de *macro-états* dont peuvent éventuellement dépendre les calculs à réaliser.

Un cas typique illustrant ce deuxième point est celui où un traitement spécifique – s'étalant éventuellement sur plusieurs itérations du motif – doit être appliqué au voisinage des bords du domaine des données d'entrée/sortie. Ce domaine peut alors être délimité en régions correspondant chacune à un *état* du motif.

2.3.7 Résumé

Les tableaux 2.3 et 2.4 résument les paramètres d'un composant virtuel comportemental et les propriétés associées aux différents niveaux de spécification.

TAB. 2.3 – Paramètres et contraintes relatifs à chaque niveau de spécification

<table>
<tr><th>Spécification</th><th>Fonctionnalité</th><th>Complexité</th><th>Topologie</th><th>Temps</th></tr>
<tr><td>Fonctionnelle</td><td>Paramètres primaires</td><td>Nombre d'échantillons</td><td>–</td><td>–</td></tr>
<tr><td rowspan="3">Algorithmique en précision infinie</td><td rowspan="3">–</td><td colspan="3">Choix de l'algorithme</td></tr>
<tr><td colspan="3">Autres Paramètres primaires</td></tr>
<tr><td colspan="3">Paramètres secondaires</td></tr>
<tr><td>Algorithmique en précision fixée</td><td colspan="2">Formats de données</td><td>–</td><td>–</td></tr>
<tr><td>Comportementale au niveau transaction</td><td>–</td><td colspan="3">Choix du motif de calcul</td></tr>
<tr><td rowspan="2">Comportementale au cycle près</td><td rowspan="2">–</td><td rowspan="2">–</td><td rowspan="2">–</td><td>Date des E/S au cycle près</td></tr>
<tr><td>Cadence de traitement</td></tr>
</table>

TAB. 2.4 – Propriétés relatives à chaque niveau de spécification

<table>
<tr><th>Spécification</th><th>Fonctionnalité</th><th>Complexité</th><th>Topologie</th><th>Temps</th></tr>
<tr><td rowspan="2">Fonctionnelle</td><td rowspan="2">–</td><td rowspan="2">–</td><td>Nombre de dimensions des données d'E/S</td><td rowspan="2">–</td></tr>
<tr><td>Coordonnées bornées/ non bornées</td></tr>
<tr><td rowspan="2">Algorithmique en précision infinie</td><td rowspan="2">–</td><td rowspan="2">Nombre moyen d'opérations par E/S</td><td>Régularité</td><td>Ordre partiel des calculs</td></tr>
<tr><td colspan="2">Longueur du chemin critique</td></tr>
<tr><td>Algorithmique en précision fixée</td><td>Bruit de calcul</td><td>Volume de données manipulées</td><td>–</td><td>–</td></tr>
<tr><td rowspan="2">Comportementale au niveau transaction</td><td rowspan="2">–</td><td>Nombre d'opérations par itération</td><td colspan="2" rowspan="2">Parallélisme exploitable au niveau opération</td></tr>
<tr><td>Nombre de transferts par itération</td></tr>
<tr><td>Comportementale au cycle près</td><td>–</td><td colspan="2">Taille des ports d'E/S</td><td>Débit d'E/S</td></tr>
<tr><td>RTL</td><td>–</td><td>–</td><td>–</td><td>Parallélisme effectif au niveau opération</td></tr>
<tr><td>RTL structurelle</td><td>–</td><td>Nombre de ressources matérielles</td><td>–</td><td>–</td></tr>
</table>

Chapitre 3

Synthèse de Composants Virtuels Comportementaux

Dans le chapitre 2, nous avons présenté les étapes générales d'un flot de conception de composants virtuels de niveau comportemental en nous concentrant plus particulièrement sur le raffinement d'une description algorithmique en une description comportementale. A ce stade, nous avons laissée ouverte la question du style d'écriture et du modèle de représentation interne du comportement permettant de supporter l'expression des contraintes de temps et d'entrées/sorties.

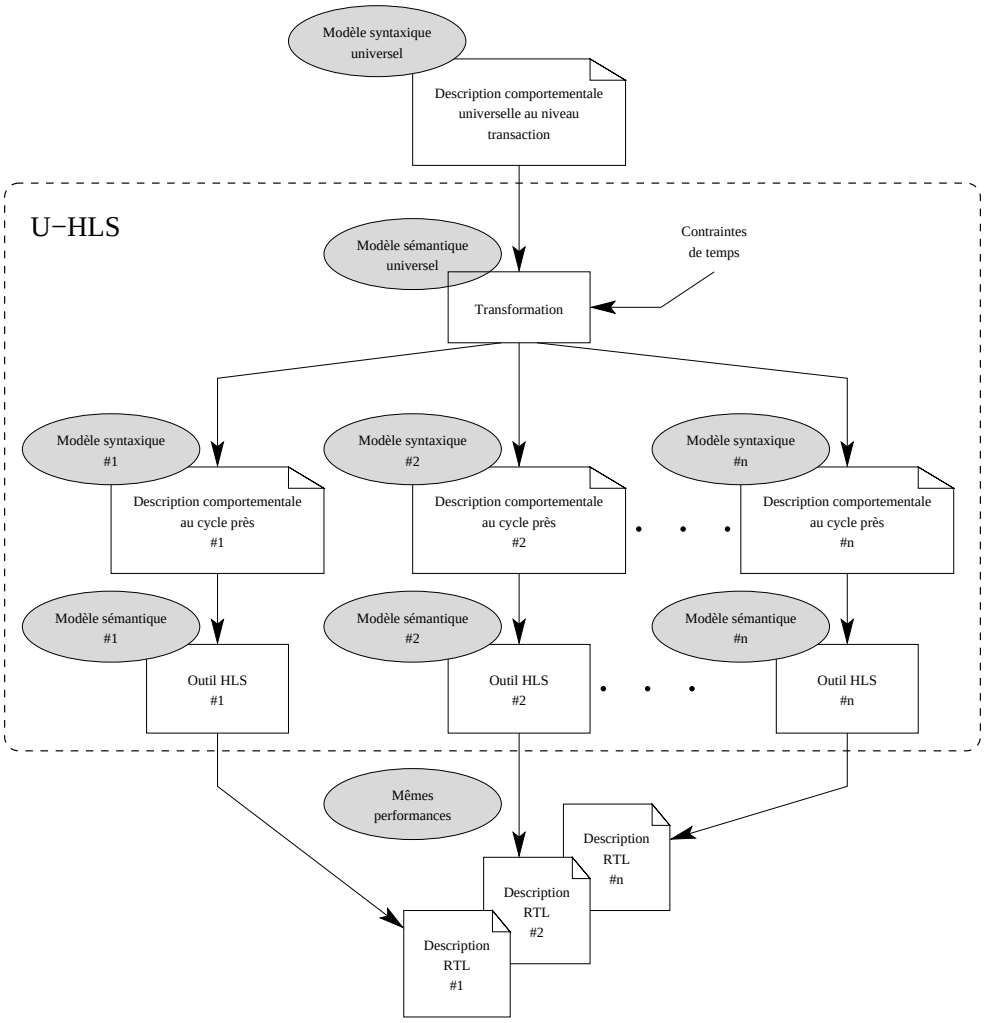
Afin de remplir les exigences d'interopérabilité et de prédictibilité nécessaires à la réutilisation d'un composant virtuel, comportemental, nous formalisons dans ce chapitre le style d'écriture et la sémantique d'implantation d'une description de niveau comportemental. A la différence du standard *IEEE 1076.6*, qui établit la correspondance entre *constructions syntaxiques VHDL* et *structures matérielles* combinatoires ou séquentielles au niveau transfert de registres, l'élévation du niveau d'abstraction inhérent à la synthèse de haut niveau reporte la question de la sémantique d'implantation sur la définition d'un *modèle de représentation* intermédiaire interprétable sans ambiguïté par les outils de synthèse de haut niveau.

Face à la diversité des langages et modèles supportés par les outils, notre démarche nous amène tout d'abord à considérer les problèmes suivants :

1. la définition d'un style d'écriture *universel* au niveau comportemental, adaptable à une variété de langages de description ;
2. la définition d'un modèle de représentation *universel* pour les descriptions comportementales ;
3. la définition des contraintes de temps et d'entrées/sorties exprimables à travers ce modèle.

Ces trois points formalisent un modèle d'outil de synthèse de haut niveau universel (*U-HLS*) [1, 2, 8] capable d'associer une sémantique d'implantation bien définie à une description comportementale avec entrées/sorties au niveau transaction. A partir de ce modèle d'outil, nous construisons un flot de synthèse de haut niveau fondé sur les outils existants et satisfaisant les exigences d'interopérabilité et de prédictibilité [80, 82]. Pour cela, nous insérons en amont des outils de synthèse de haut niveau une étape

FIG. 3.1 – Le flot de synthèse de haut niveau universel



de transformation (figure 3.1) consistant à adapter le langage et le style d'écriture de la description de telle sorte que la représentation interne, propre à chaque outil, du comportement soit conforme au cycle près à notre modèle universel [1, 2, 8].

3.1 Exemple introductif

Le listing 3.1 décrit en langage *C* le comportement d'un composant de calcul de distorsion pour l'estimation de mouvement par block-matching destiné à être intégré, par exemple, à un codeur *MPEG4*. L'étude de l'implantation de cet algorithme sous la forme d'un composant virtuel comportemental a été réalisée dans le cadre du projet *RNRT MILPAT* [83].

La distorsion (SAD) est définie comme la distance de Manhattan entre un macro-

bloc $MBref$ de pixels d'une image de référence et un macro-bloc $MBcurr$ de pixels de l'image courante.

$$SAD = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} |MBcurr(i,j) - MBref(i,j)| \tag{3.1}$$

La taille des blocs peut ici être modifiée dynamiquement : elle peut être de 8×8, 8×16 ou 16×16 pixels.

Dans un premier temps, nous proposons une description comportementale sans contrainte d'entrée/sortie (niveau transaction) : l'entrée *typeMB* est de type symbolique et indique la taille du macro-bloc à traiter, les entrées *mbX* et *mbY* fournissent les pixels des deux macro-blocs à comparer. Ces pixels sont disposés sous forme d'un tableau tridimensionnel pouvant contenir de 1 à 4 sous-blocs de 8×8 pixels. Afin d'éviter des calculs inutiles, le traitement s'interrompt dès que la distance calculée dépasse un seuil lu sur l'entrée *seuil*. Le test d'échappement est effectué à la fin du traitement de chaque ligne d'un sous-bloc de 8×8 pixels. La sortie booléenne *depassement* indique si le seuil a été dépassé. La distance calculée est émise sur la sortie *distance*.

Listing 3.1 – Algorithme de calcul de distance de Manhattan en C

```
switch(typeMB) {
  case MB_8x8   : nb_blocs = 1; break;
  case MB_16x8  : nb_blocs = 2; break;
  case MB_16x16 : nb_blocs = 4;
}

distance = 0;
depassement = false;

for(b=0;b<nb_blocs;b++) {      // Boucle "MacroBlocLoop"
  for(l=0;l<8;l++) {           // Boucle "LigneLoop"
    for(k=0;k<8;k++) {         // Boucle "ColonneLoop"
      if(mbX[b][l][k] >= mbY[b][l][k])
        distance += mbX[b][l][k] - mbY[b][l][k];
      else
        distance += mbY[b][l][k] - mbX[b][l][k];
    }
    depassement = (distance >= seuil);
    if(depassement) break;
  }
  if(depassement) break;
}
```

Dans cette section, nous allons porter notre attention sur la synthèse des boucles et de l'instruction de branchement *break*. Nous montrons que la seule lecture de la description comportementale proposée ne permet pas d'inférer un unique modèle de représentation intermédiaire sur lequel le concepteur peut insérer sans ambiguïté les contraintes d'entrée/sortie nécessaires à l'élaboration d'un modèle comportemental exact au cycle près. La sémantique d'implantation des boucles, par exemple, est ambiguë et leur traitement dépend d'une part de directives de la part du concepteur

(dérouler ou ne pas dérouler une boucle) et des règles de projection du niveau comportemental vers le modèle de représentation.

Nous montrons également que les variations de la sémantique d'implantation peuvent être résolues en effectuant une réécriture de la description de manière à ce que les constructions syntaxiques qu'elles contient soient interprétables sans ambiguïté.

3.1.1 Déroulage des boucles

Le déroulage des boucles permet de mettre en évidence le parallélisme exploitable entre itérations successives. Parmi les trois boucles imbriquées figurant dans le listing 3.1, seule la boucle *MacroBlocLoop* ne peut pas être déroulée complètement car son nombre d'itération peut varier dynamiquement en fonction de la valeur de l'entrée *typeMB*.

Certains outils de synthèse comme *GAUT* ne supportent pas le non-déroulage des boucles. Cette description ne serait par conséquent pas synthétisable par un tel outil. Cependant, la connaissance des cas de fonctionnement du composant permet de constater que le nombre d'itérations de la boucle *MacroBlocLoop* ne dépassera pas 4. Un utilisateur souhaitant à tout prix dérouler cette boucle peut ainsi transformer sa description de la manière présentée dans le tableau 3.1a. Cette transformation consiste à remplacer la borne d'itération finale par sa valeur limite et à insérer une instruction de sortie conditionnelle (*break*) permettant d'interrompre la boucle lorsque la dernière itération a été effectuée. Le nombre d'itérations étant à présent déterminé statiquement, le déroulage de cette boucle donne une description du type de celle présentée dans le tableau 3.1b.

Comme nous le constatons, l'utilisation de l'instruction *break* se traduit par une cascade de structures *if* imbriquées. Cette imbrication se traduit par des *dépendances de contrôle* qui limitent le parallélisme exploitable [53, 57] : la quatrième itération du corps de boucle ne peut en effet être exécutée que lorsque les quatre comparaisons ($b == nb\_blocs$) ont été calculées. Si l'utilisation de l'instruction *break* est naturelle dans le domaine du logiciel où les instructions sont naturellement exécutées en séquences, nous constatons qu'elle peut être pénalisante dans le domaine de la description de matériel. Le tableau 3.1c décrit une autre méthode qui peut sembler intuitivement moins efficace, mais donne en réalité plus de liberté à l'outil de synthèse pour ordonnancer et paralléliser les opérations – dans la mesure où les dépendances de données entre itérations successives le permettent. Le tableau 3.1d montre en effet que les quatre versions du corps de boucle après déroulage sont à présent indépendantes.

Ce premier exemple nous montre que le style de description adopté initialement ne permet pas de cibler un type d'architecture exploitant du parallélisme entre itérations de la boucle *MacroBlocLoop*. Les directives de synthèse supportées par les outils existants ne permettent pas de rendre la boucle déroulable et des stratégies de contournement doivent être envisagées.

Enfin, nous avons vu que le style d'écriture a une influence directe sur les performances du matériel [2]. Le choix du style le plus adapté dans un contexte de synthèse d'architecture n'est pas nécessairement le plus intuitif ni le plus lisible.

TAB. 3.1 – Déroulage forcé d'une boucle *for*

(a)

```
for(b=0;b<4;b++) {
  if(b == nb_blocs)
    break;
  {...}
}
```

(b)

```
if(0 == nb_blocs)
  ; // Sortie de boucle
else {
  // Corps de boucle (b=0)
  {...}
  if(1 == nb_blocs)
    ; // Sortie de boucle
  else {
    // Corps de boucle (b=1)
    {...}
    if(2 == nb_blocs)
      ; // Sortie de boucle
    else {
      // Corps de boucle (b=2)
      {...}
      if(3 == nb_blocs)
        ; // Sortie de boucle
      else {
        // Corps de boucle (b=3)
        {...}
      }
    }
  }
}
```

(c)

```
for(b=0;b<4;b++) {
  if(b < nb_blocs) {
    {...}
  }
}
```

(d)

```
if(0 < nb_blocs) {
  // Corps de boucle (b=0)
  {...}
}
if(1 < nb_blocs) {
  // Corps de boucle (b=1)
  {...}
}
if(2 < nb_blocs) {
  // Corps de boucle (b=2)
  {...}
}
if(3 < nb_blocs) {
  // Corps de boucle (b=3)
  {...}
}
```

3.1.2 Non-déroulage des boucles

Lorsque toutes les boucles sont déroulées, un motif de calcul peut être entièrement représenté sous forme d'un graphe flot de données auquel sera alloué un budget de temps fixe. Dans la description du calcul de distance de *Manhattan* au contraire, l'effet escompté des instructions *break* est l'interruption immédiate du calcul afin de rendre le composant disponible pour le traitement d'un nouveau macro-bloc. Ce type de comportement ne peut pas être modélisé à l'aide de boucles déroulées.

Le non-déroulage d'une boucle permet aux instructions de branchement *break* et *continue* du langage *C* (*exit* et *next* en *VHDL*) d'avoir une influence effective sur le déroulement temporel du comportement [53,57]. Au niveau transfert de registres, elles se traduiront par des transitions de la *FSM* de contrôle de l'architecture, rompant la succession linéaire des états. Dans ce paragraphe, nous considérons la situation où les boucles *MacroBlocLoop* et *LigneLoop* ne sont pas déroulées.

Boucles non déroulées et contraintes de temps – Lorsqu'une description contient des boucles non déroulées dont le nombre d'itérations peut varier dynamiquement, il n'est plus possible au concepteur de spécifier une contrainte de temps unique correspondant à la durée totale d'une itération du motif. Si l'on représente la description sous forme de *CDFG*, la durée d'exécution du motif correspondra à la durée cumulée des *DFG* traversés. Le parcours dans le *CDFG* étant *a priori* indéterminé, les contraintes de temps doivent être précisées indépendamment pour chaque *DFG*.

La principale difficulté quant à l'utilisation des outils de synthèse de haut niveau réside dans le fait qu'ils masquent leur modèle de représentation interne, et que ce modèle n'est pas nécessairement identique à la représentation intuitive sur laquelle le concepteur raisonne.

Le tableau 3.2 présente deux visions du déroulement du calcul de la distance de *Manhattan*. Les deux listings sont obtenus en séparant explicitement les instructions qui relèvent du contrôle (boucles et branchement) de celles qui exécutent des calculs (affectation d'expressions arithmétiques et logiques). La colonne de gauche correspond à une vision plutôt "logicielle", c'est-à-dire conforme à la séquence d'instructions qu'exécuterait un processeur ; la colonne de droite correspond au modèle de représentation interne de l'outil *Monet*. Les deux descriptions obtenues peuvent être représentées schématiquement sous forme de *CDFG* (figure 3.2).

Nous constatons que le *CDFG* vu par le concepteur et celui vu par un outil de synthèse peuvent présenter des répartitions différentes du flot de données sur le flot de contrôle. La connaissance du modèle de représentation propre à l'outil de synthèse utilisé est nécessaire à la répartition des contraintes de temps sur les différentes sections de la description.

Exemples de synthèse – Nous avons traduit directement en *VHDL* la description du calcul de distance de *Manhattan* en suivant les règles d'écriture figurant dans les manuels d'utilisation des outils *Monet* [62] et *Behavioral Compiler* [64]. Si nous spécifions comme contrainte de temps que le corps de la boucle *LigneLoop* doit s'exécuter en N cycles d'horloge par itération, nous obtenons deux solutions architecturales dont

TAB. 3.2 – Deux exemples de réécriture des boucles non déroulées

Concepteur

```
switch(typeMB){...}
distance = 0;
depassement = false;
b = 0;
for(;;) {
  tmp1 = (b>=nb_blocs);
  if(tmp1) break;
  l = 0;
  for(;;) {
    tmp2 = (l==8);
    if(tmp2) break;
    for(k=...){...}
    depassement = (distance>=seuil);
    if(depassement) break;
    ++l;
  }
  if(depassement) break;
  ++b;
}
```

Monet

```
switch(typeMB){...}
distance = 0;
depassement = false;
next_b = 0;
tmp1 = (next_b<nb_blocs);
if(tmp1) {
  for(;;) {
    b = next_b;
    ++next_b;
    tmp1 = (next_b==nb_blocs);
    next_l = 0;
    for(;;) {
      l = next_l;
      ++next_l;
      tmp2 = (next_l==8);
      for(k=...){...}
      depassement = (distance>=seuil);
      if(depassement) break;
      if(tmp2) break;
    }
    if(depassement) break;
    if(tmp1) break;
  }
}
```

FIG. 3.2 – Calcul de distance de Manhattan : réécriture de la description comportementale et CDFG correspondant

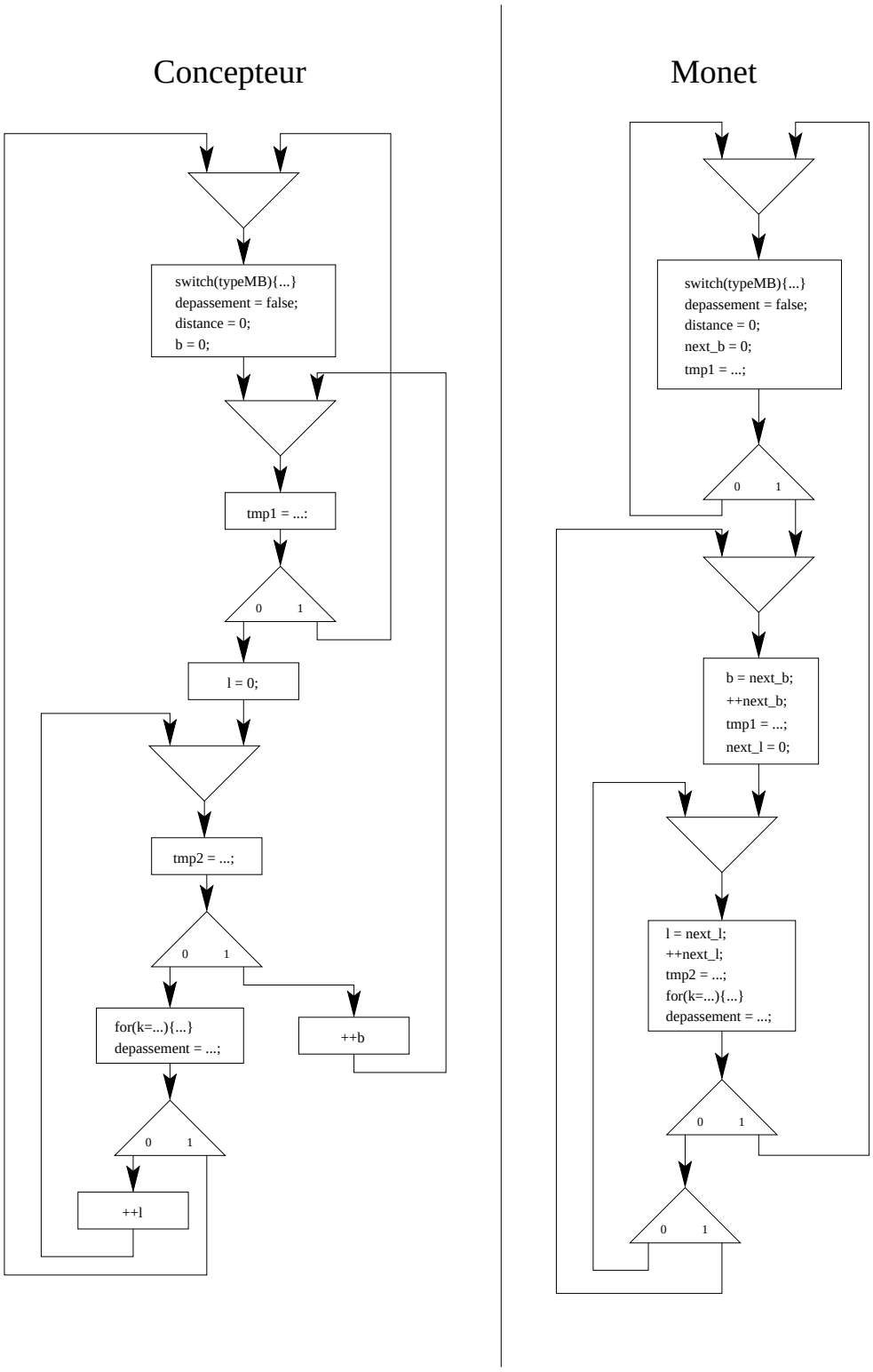
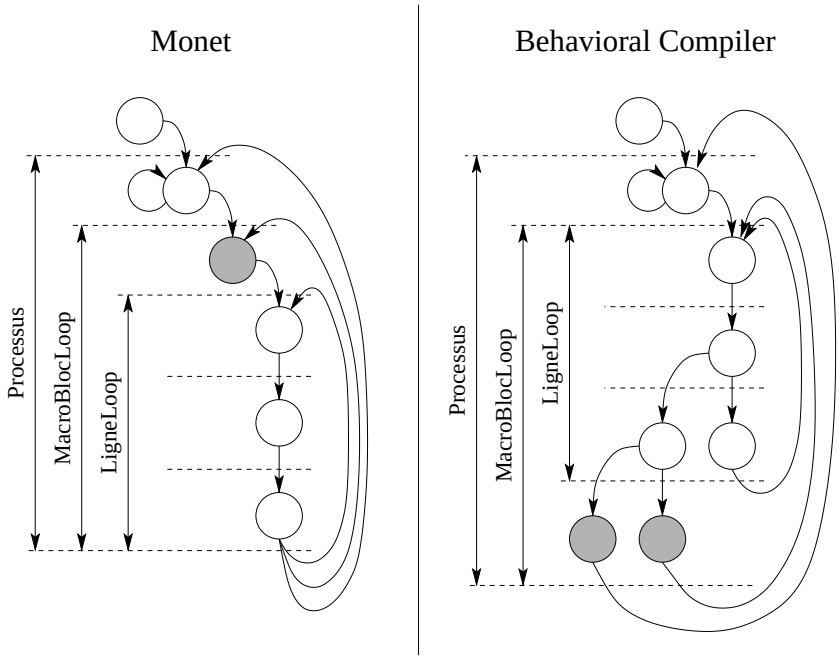


FIG. 3.3 – Calcul de distance de Manhattan : FSM générées par les outils Monet et BC



TAB. 3.3 – Calcul de distance de Manhattan : caractéristiques des FSM produites par Monet et BC

Outil	Etats	Transitions	Cycles/macro-bloc	Date de lecture des pixels
Monet	$2+N$	$4+N$	$1+nb\_blocs \times (1+8N)$	$\mathbf{2}+b \times (1+8N)+l \times N$
BC	$4+N$	$8+N$	$1+nb\_blocs \times (1+8N)$	$\mathbf{1}+b \times (1+8N)+l \times N$

le contrôleur possédera bien N états dédiés à l'exécution du corps de cette boucle (voir figure 3.3 pour $N = 3$).

Lors de l'ordonnancement de la boucle *MacroBlocLoop*, un cycle est réservé aux opérations liées à l'évolution du compteur de boucle b. *Monet* place ce cycle en début de boucle [53] tandis que *BC* le situe en fin de boucle [57]. Les états correspondants figurent en grisé sur la figure 3.3. Si nous supposons que les 16 valeurs de pixels traitées pendant une itération de la boucle *LigneLoop* sont lues simultanément au début de chaque itération, nous observerons alors un décalage d'un cycle entre les dates de lecture des données des deux solutions architecturales (tableau 3.3).

Ainsi, bien que le style d'écriture utilisé au niveau comportemental soit *a priori* dénué de tout détail d'implantation, le choix des instructions de description d'une part, et le modèle de représentation propre à chaque outil d'autre part, imposent des restrictions quant aux comportements temporels exprimables : dans la situation qui nous intéresse, le choix d'une écriture à base de deux boucles *for* imbriquées impose systématiquement la réservation d'un cycle au début ou à la fin de chaque itération

TAB. 3.4 – Calcul de distance de Manhattan : propriétés des FSM générées par Monet et BC après optimisation manuelle de la description

Outil	Etats	Transitions	Cycles/macro-bloc	Date de lecture des pixels
Monet	$1+N$	$3+N$	$1+nb\_blocs \times 8N$	$1+b \times 8N+l \times N$
BC	$1+N$	$3+N$	$1+nb\_blocs \times 8N$	$1+b \times 8N+l \times N$

de la boucle la plus externe. La rigidité des modèles de représentation de ces outils se traduit ainsi par un manque de *flexibilité* du comportement aux entrées/sorties. Ce manque de flexibilité s'accompagne d'une *imprédictibilité* du comportement temporel, lequel dépend fortement de l'outil utilisé.

Synthèse sous contrainte de modèle – Résoudre ces difficultés impose de recourir à un style de description de plus bas niveau, notamment en réécrivant les boucles *for* à l'aide de boucles infinies et en plaçant manuellement les instructions d'incrémentation du compteur de boucle et de sortie conditionnelle à des endroits bien choisis [8].

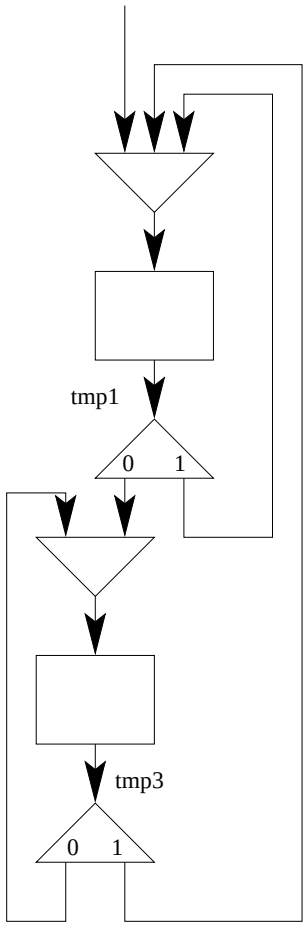
Le listing 3.2 donne un exemple de description modifiée. Le *CDFG* correspondant est représenté en figure 3.4. Les deux boucles imbriquées de la description initiale ont été fusionnées de manière à faire disparaître l'état inséré par les outils en début ou fin de la boucle *MacroBlocLoop*. Après traduction en *VHDL* et synthèse sous *Monet* et *BC*, les deux solutions obtenues présentent le même comportement aux entrées/sorties. Les caractéristiques de leurs *FSM* de contrôle sont présentées dans le tableau 3.4.

Listing 3.2 – Calcul de distance de Manhattan : optimisation manuelle de la description

```
switch(typeMB){...}
distance = 0;
depassement = false;
b = 0;
l = 0;
tmp1 = (b>=nb_blocs);
if(!tmp1) {
  for(;;) {
    for(k=...){...}
    depassement = (distance>=seuil);
    tmp2 = (l==7);
    if(tmp2) {
      l = 0;
      ++b;
      tmp1 = (b==nb_blocs);
    }
    else
      ++l;
    tmp3 = (depassement||tmp1);
    if(tmp3) break;
  }
}
```

Une telle réécriture n'est pas triviale et se traduira généralement par une perte de lisibilité de la description. **L'un des objectifs du formalisme proposé dans ce**

FIG. 3.4 – Calcul de distance de Manhattan : CDFG après optimisation manuelle de la description



chapitre est l'automatisation des choix de style d'écriture en fonction de l'outil ciblé et des contraintes de temps et d'entrées/sorties.

3.2 Modèle de représentation "universel"

L'exemple que nous venons de présenter a mis en lumière un certain nombre de problèmes inhérents à l'utilisation de la synthèse de haut niveau. Nous avons en particulier montré que certaines instructions comme la boucle *for* n'ont pas une sémantique d'implantation unique, mais supportent une variété de représentations dépendant tant des directives de synthèse (déroulage/non-déroulage) que du modèle de représentation de l'outil ciblé [80]. Contraindre des outils de synthèse différents à respecter un modèle de représentation commun et conforme aux souhaits du concepteur nécessite de recourir à des styles d'écriture de plus bas niveau séparant explicitement les instructions de contrôle des instructions de traitement.

Il existe ainsi deux niveaux de style d'écriture d'une description comportementale :

- le premier niveau fait usage de *primitives de description comportementale*, c'est-à-dire d'instructions reconnues valides en synthèse de haut niveau, mais auxquelles une variété de modèles de représentation peuvent être associés ;
- le second niveau se fonde sur un ensemble de *primitives de synthèse de haut niveau*, sous-ensemble des primitives de description comportementales, mais dont le modèle de représentation est défini de manière unique.

Dans la suite de ce chapitre, nous formalisons tout d'abord l'ensemble des primitives de synthèse de haut niveau et nous définissons un modèle de représentation "universel" – c'est-à-dire indépendant des outils existants – supportant l'optimisation des propriétés temporelles d'une description, l'expression de contraintes de temps et d'entrée/sortie. Ensuite, nous étendons le domaine de description autorisé en définissant l'ensemble des primitives de description comportementale et les règles de transformation vers les primitives de synthèse de haut niveau.

Il est important de noter que la question des langages de spécification est secondaire dans cette approche. Plutôt que de formaliser un langage de description unique, nous proposons de contraindre seulement le style d'écriture d'une description.

3.2.1 Primitives de synthèse de haut niveau

Les primitives de description comportementale et les primitives de synthèse de haut niveau seront présentées à l'aide d'un langage symbolique aisément adaptable à une variété de langages de description. La grammaire de ce langage symbolique fournit un modèle standard d'*arbre syntaxique* pouvant servir de base à la définition de *parsers* pour les langages de description usuels. Les nœuds non terminaux de cette structure arborescente seront présentés à l'aide de la notation *EBNF* suivante :

$$TypeNoeud ::= \textbf{mot-clé}(\text{nom\_champ} \rightarrow TypeNoeud, \ldots)$$

Le mot-clé indique la sémantique d'un nœud tandis que la liste de ses champs renvoie à ses nœuds-fils.

Les instructions relevant des primitives de synthèse de haut niveau ($Prim_{HLS}$) sont celles dont la sémantique peut être identifiée sans ambiguïté comme relevant soit du traitement de données ($Prim_{Data}$), soit du contrôle ($Prim_{Ctrl}$).

$$Prim_{HLS} ::= Prim_{Data} | Prim_{Ctrl}$$

Dans un souci de clarté, les séquences d'instructions seront notées en les séparant par des point-virgules :

$$Seq_{HLS} ::= Prim_{HLS}\{\ ;\ Prim_{HLS}\}$$

Primitives de traitement

Les instructions de traitement concernent toutes les instructions pouvant être projetées sur un *DFG* [8]. Cela inclut :
- les instructions d'affectation ($Assign$) ;
- les instructions conditionnelles ne contenant que des instructions de traitement (If_{Data}, $Case_{Data}$) ;
- les instructions de boucles à dérouler ne contenant que des instructions de traitement (For_{Data}) ;
- les appels de sous-programmes ($PCall$).

Nous définissons également la notion de séquence de primitives de traitement (Seq_{Data}) : dans le modèle *HSFSMD*, deux séquences de primitives de traitement disjointes, c'est-à-dire séparées par au moins une primitive de contrôle, sont associées à deux transitions distinctes.

$$\begin{aligned} Prim_{Data} &::= Assign | If_{Data} | Case_{Data} | For_{Data} | PCall \\ Seq_{Data} &::= Prim_{Data}\{\ ;\ Prim_{Data}\} \end{aligned}$$

Une instruction d'affectation ($Assign$) stocke le résultat d'une expression dans une variable interne ou dans une sortie. Ces variables peuvent avoir un type scalaire ou structuré (tableau ou *record*). Ici, nous considérerons qu'un *record* est un cas particulier de tableau dont l'indice est symbolique au lien d'être numérique. Une expression ($Expr$) est constituée soit d'une valeur littérale ($Litt$), d'un identifiant (Id) représentant une variable, d'un élément de tableau ou *record* ($Cell$), ou enfin de l'évaluation d'une opération (Op) sur un ensemble d'expressions opérandes. Les appels de fonctions sont considérées comme des opérations.

$$\begin{aligned} Assign &::= \textbf{set}(\text{left} \rightarrow Obj, \text{right} \rightarrow Expr) \\ Obj &::= Id | Cell \\ Cell &::= \textbf{select}(\text{name} \rightarrow Id, \text{indices} \rightarrow ExprList) \\ Expr &::= Litt | Obj | Eval \\ Eval &::= \textbf{eval}(\text{operation} \rightarrow Op, \text{operands} \rightarrow ExprList) \\ ExprList &::= \text{“<”} Expr\{, Expr\} \text{“>”} \end{aligned}$$

La modélisation d'une expression sous forme d'un *DFG* suppose que la notation est entièrement parenthésée. Le choix du positionnement des parenthèses peut avoir

une influence sur les possibilités de parallélisation des sous-expressions [2].

Les instructions conditionnelles ne contenant que des instructions de traitement sont classées parmi les instructions de traitement car elles sont généralement modélisées sous la forme d'opérations logiques de sélection [53, 57]. Dans le chemin de données des architectures obtenues en synthèse de haut niveau elles apparaissent sous la forme de multiplexeurs permettant la circulation conditionnelle des données. Les instructions conditionnelles supportées sont l'alternative "*si ...alors ...sinon ...*" (*If*) et le choix parmi un certain nombre de cas (*Case*) :

$$
\begin{aligned}
If_{Data} &::= \textbf{if}(\text{test} \rightarrow Expr, \text{then} \rightarrow Seq_{Data}, \text{else} \rightarrow Seq_{Data}) \\
Case_{Data} &::= \textbf{case}(\text{test} \rightarrow Expr, \text{choices} \rightarrow ChoiceList_{Data}, \text{default} \rightarrow Seq_{Data}) \\
ChoiceList_{Data} &::= \text{“<”}Choice_{Data}, \{Choice_{Data}\}\text{“>”} \\
Choice_{Data} &::= \textbf{when}(\text{values} \rightarrow ExprList, \text{body} \rightarrow Seq_{Data})
\end{aligned}
$$

La figure 3.5 donne un exemple de modélisation d'une instruction conditionnelle extraite du calcul de distance de *Manhattan*.

Les instructions de boucles à dérouler (*For*) seront des boucles à compteur (*ctr*) dont les bornes d'itération (*first*, *last*) et l'incrément (*step*) seront des expressions statiquement déterminées.

$$
\begin{aligned}
For_{Data} ::= \textbf{for}(\text{name} &\rightarrow Label, \text{ctr} \rightarrow Id, \\
\text{first} &\rightarrow Expr, \text{last} \rightarrow Expr, \text{step} \rightarrow Expr \\
\text{body} &\rightarrow Seq_{Data})
\end{aligned}
$$

Primitives de contrôle

Afin de séparer explicitement le contrôle du traitement, nous restreignons les instructions de contrôle aux trois suivantes [8] :

- boucles infinies (instruction *loop* en *VHDL*, ou *for(;;)* en langage *C*) ;
- poursuite d'une boucle (instructions *next* et *continue*) ;
- sortie d'une boucle (instructions *exit* et *break*).

$$Prim_{Ctrl} ::= Loop_{Ctrl} | Next_{Ctrl} | Exit_{Ctrl}$$

Une boucle sera systématiquement associée à un *label* permettant aux instructions de branchement de spécifier explicitement la boucle à laquelle elles s'adressent. L'entrée dans une boucle peut être conditionnelle. La condition d'entrée est matérialisée par un nom de variable booléenne affectée dans le chemin de données.

$$Loop_{Ctrl} ::= \textbf{loop}(\text{name} \rightarrow Label, \text{condition} \rightarrow Id, \text{body} \rightarrow Seq_{HLS})$$

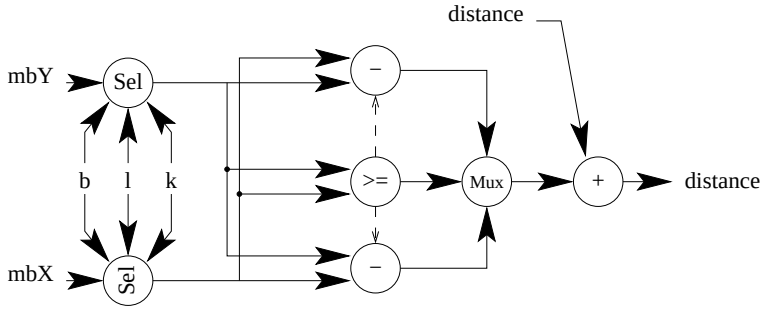
Les instructions de poursuite ($Next_{Ctrl}$) et de sortie ($Exit_{Ctrl}$) d'une boucle peuvent être conditionnelles. La condition de sortie ou de poursuite est matérialisée par un nom de variable booléenne affectée dans le chemin de données.

$$
\begin{aligned}
Next_{Ctrl} &::= \textbf{next}(\text{loopname} \rightarrow Label, \text{condition} \rightarrow Id) \\
Exit_{Ctrl} &::= \textbf{exit}(\text{loopname} \rightarrow Label, \text{condition} \rightarrow Id)
\end{aligned}
$$

FIG. 3.5 – Modélisation d'une séquence de primitives de traitement : langage C/notation symbolique/DFG

```
if(mbX[b][l][k] >= mbY[b][l][k])
  distance += mbX[b][l][k] - mbY[b][l][k];
else
  distance += mbY[b][l][k] - mbX[b][l][k];
```

```
if(test→eval(operation→"≥",
             operands→≺select(name→ mbX,
                              indices→≺b,l,k>),
                      select(name→ mbY,
                              indices→≺b,l,k>)>),
   then→set(left→distance,
            right→eval(
                     operation→"+",
                     operands→≺
                       distance,
                       eval(operation→"−",
                            operands→≺
                              select(name→ mbX,
                                     indices→≺b,l,k>),
                              select(name→ mbY,
                                     indices→≺b,l,k>)>)>)),
   else→set(left→distance,
            right→eval(
                     operation→"+",
                     operands→≺
                       distance,
                       eval(operation→"−",
                            operands→≺
                              select(name→ mbY,
                                     indices→≺b,l,k>),
                              select(name→ mbX,
                                     indices→≺b,l,k>)>)>)),
```



Le listing 3.3 donne une traduction possible du calcul de distance de Manhattan à l'aide des primitives de synthèse de haut niveau. Cette traduction correspond à la vue *concepteur* présentée dans le tableau 3.2.

Listing 3.3 – Traduction du calcul de distance de Manhattan (vue "concepteur" du tableau 3.2) à l'aide de notre langage symbolique

```
case(test → typeMB, choices → ...) ;
set(left → distance, right → 0) ;
set(left → depassement, right → false) ;
set(left → b, right → 0) ;
loop(label → MacroBlocLoop,
     body →
       set(left → tmp1, right → ...) ;
       exit(loopname → MacroBlocLoop, condition → tmp1) ;
       set(left → l, right → 0) ;
       loop(label → LigneLoop,
            body →
              set(left → tmp2, right → ...) ;
              exit(loopname → LigneLoop, condition → tmp2) ;
              for(label → ColonneLoop, ctr → k, ...) ;
              set(left → depassement, right → ...) ;
              exit(loopname → LigneLoop, condition → depassement) ;
              set(left → l, right → ...)
           ) ;
       exit(loopname → MacroBlocLoop, condition → depassement) ;
       set(left → b, right → ...)
   )
```

Sous-programmes

Les sous-programmes sont particulièrement utiles pour hiérarchiser une description. Au sens de l'effort de conception, ils permettent d'éliminer des redondances de code, d'aboutir à une description plus lisible et donc plus aisée à maintenir et à déboguer. En synthèse de haut niveau, les appels de fonctions et procédures se traduisent généralement par la mise en ligne du corps de ces sous-programmes. En fonction des directives de synthèse, les sous-programmes permettent également d'isoler explicitement des fragments de code que le concepteur souhaite synthétiser séparément ou projeter sur des composants existant en bibliothèque.

Les contraintes liées à l'utilisation de sous-programmes en synthèse sont les suivantes : lorsqu'un appel de sous-programme est destiné à être projeté sur un composant en bibliothèque, nous devons nous assurer que ce sous-programme ne réalise pas d'effet de bord, c'est-à-dire n'accède pas à des ressources partagées – par exemple une variable globale – qui représenteraient des entrées/sorties cachées. Lorsqu'un appel de sous-programme est destiné à être mis en ligne, les éventuels appels récursifs doivent pouvoir être résolus statiquement.

Fonctions et procédures sont identifiées par un nom, une liste de paramètres formels "données", et dans le cas de la procédure une liste de paramètres formels "résultats". Le corps d'un sous-programme est une séquence d'instructions.

Nous limitons le domaines des sous-programmes autorisés pour la mise en ligne en leur imposant de ne contenir que des primitives de traitement :

$$
\begin{aligned}
Func_{Data} &::= \textbf{func}(\text{name} \rightarrow Id, \text{data} \rightarrow IdList, \text{body} \rightarrow Seq_{Data}) \\
Proc_{Data} &::= \textbf{proc}(\text{name} \rightarrow Id, \text{data} \rightarrow IdList, \text{results} \rightarrow IdList, \text{body} \rightarrow Seq_{Data}) \\
IdList &::= \text{“<”}Id\{, Id\}\text{“>”}
\end{aligned}
$$

Une procédure est appelée au moyen de son nom, d'une liste d'arguments "données", qui sont des expressions arithmétiques ou logiques et d'une liste d'arguments "résultats" qui sont des noms de variables. Un appel de procédure sera considéré comme une primitive de traitement.

$$PCall ::= \textbf{pcall}(\text{procname} \rightarrow Id, \text{data} \rightarrow ExprList, \text{results} \rightarrow IdList)$$

3.2.2 Modélisation des primitives de synthèse de haut niveau

Les méthodologies de synthèse de haut niveau font usage de deux types de modèles de représentation : (1) les *CDFG* représentent un comportement en termes de dépendances de données et de contrôle ; (2) les *SFSMD* utilisent un formalisme à base d'états et de transitions.

Le second modèle présente l'avantage d'être plus proche du modèle *FSMD* sur lequel reposent les spécifications au niveau transfert de registre. La synthèse d'une *SFSMD* revient à décomposer chaque super-état en autant d'états que de cycles d'horloge nécessaires à l'exécution des instructions qu'il contient, et à répartir les opérations sur chacun des états obtenus.

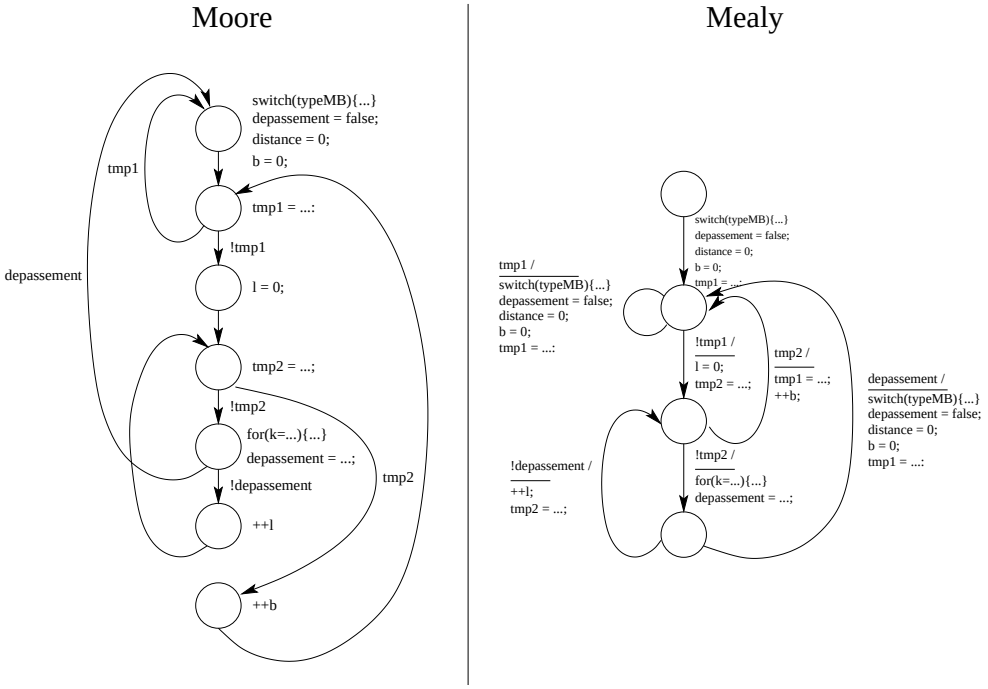
La figure 3.6 représente deux *SFSMD* obtenues à partir de la vue "concepteur" du calcul de distance de Manhattan (voir tableau 3.2). La première est une *SFSMD* de Moore, obtenue en associant chaque groupe d'instructions de traitement situé entre deux instructions de contrôle à un super-état. Le résultat présente une structure très proche du *CDFG* de la figure 3.2. La seconde est une *SFSMD* de Mealy, qui associe à chaque transition l'ensemble des calculs effectués entre deux instructions de branchement conditionnel.

Les deux modèles présentent des différences significatives en termes de *compacité* – le modèle de Mealy présente nettement moins d'états que le modèle de Moore – et de *redondance* – une expression associée à un unique état de la machine de Moore peut être associée à plusieurs transitions de la *SFSMD* de Mealy. La compacité du modèle de Mealy est ici due à une meilleure répartition des instructions dans le temps : des instructions qui figuraient dans des états distincts du modèle de Moore sont ici regroupées sur une même transition et deviennent parallélisables ou chaînables. En revanche, du fait de sa redondance, le modèle de Mealy fournit un support moins intuitif à l'expression des contraintes de temps. Enfin, aucun des deux modèles ne fait apparaître de manière explicite la hiérarchie des boucles non déroulées.

SFSMD hiérarchiques et primitives de synthèse de haut niveau

Le modèle que nous proposons est une extension des *SFSMD* de Mealy, munie d'une propriété de hiérarchie permettant de visualiser clairement l'imbrication des boucles non déroulées d'une description [8]. Ce modèle peut être directement déduit

FIG. 3.6 – Calcul de distance de Manhattan : SFSMD de Moore et de Mealy



d'une description à base de primitives de synthèse de haut niveau. Il fournit un support visuel pour les transformations de code et la spécification des contraintes de temps.

Le modèle *HSFSMD* se compose tout d'abord d'états simples et d'états hiérarchiques (c'est-à-dire pouvant être chacun décomposé en une *HSFSMD*). Un état simple représente un *point de contrôle* dans le flot d'exécution, c'est-à-dire le point de départ et/ou de destination d'une ou plusieurs instructions de branchement. Une séquence d'instructions de branchements ($Next_{Ctrl}$, $Exit_{Ctrl}$) sera modélisée par un unique état simple.

Un état hiérarchique représente une boucle infinie (*Loop*) dont les sous-états modélisent les points de contrôle contenus dans le corps de boucle. Un tel état hiérarchique possède toujours une transition inconditionnelle de rebouclage sur lui-même.

Les transitions sortant d'un état simple peuvent être de deux types : (1) soit elles indiquent la destination d'un branchement, et ne sont associées à aucun calcul ; (2) soit elles correspondent aux instructions de traitement (*Assign*, *If*, *Case*, *For*) exécutées entre deux points de contrôles successifs.

Une instruction de boucle décrivant un comportement itératif à l'intérieur d'un flot d'exécution séquentiel, elle possède un unique point d'entrée (la première instruction du corps de boucle) et un unique point de sortie (l'instruction qui suit immédiatement la boucle). Afin de retraduire cette propriété dans notre modèle, nous introduisons trois états simples nécessairement présents à l'intérieur de tout état hiérarchique :

– un *état d'entrée* (*entry*), où se rejoignent les transitions entrant dans l'état hié-

rarchique ;
- un *état fictif de sortie* (*exit*), où se rejoignent les transitions sortant de l'état hiérarchique ;
- un *état fictif de poursuite* (*next*), où se rejoignent les transitions qui rebouclent sur le point d'entrée.

Ces deux derniers états sont présentés comme *"fictifs"* car ils sont appelés à disparaître lors du raffinement d'une *HSFSMD* en une *SFSMD* propre à la synthèse de haut niveau (voir figures 3.7 et 3.8). Les transitions modélisant les instructions de branchement sont toujours dirigées vers les états fictifs de sortie et de poursuite.

La représentation de l'algorithme de calcul de distance de Manhattan sous forme de *HSFSMD* est présentée en figure 3.7. Les symboles *(S1)* à *(S7)* renvoient aux séquences de primitives de traitement que l'on peut extraire du listing 3.3. Les états représentés en grisé sont les états fictifs.

Relation entre HSFSMD et SFSMD

A partir d'une *HSFSMD* conforme à notre modèle, il est possible d'extraire une *SFSMD* équivalente en mettant à plat la structure hiérarchique (figure 3.8a) et en éliminant les états fictifs (figure 3.8b). Dans ce processus d'élimination, toute transition entre deux états non fictifs d'une *HSFSMD*, ou toute chaîne de transitions ne passant que par des états fictifs sera remplacée par une transition directe sur laquelle seront reportées les conditions de transitions et les calculs traversés.

Chaque transition d'une *SFSMD* de Mealy possède son propre intervalle de temps pour l'exécution des instructions qui lui sont attachées. Des instructions associées à deux transitions distinctes ne sont par conséquent pas parallélisables lors de la synthèse de haut niveau. Optimiser les performances temporelles d'une *HSFSMD* signifie effectuer un réarrangement des instructions autorisant des suppressions d'états, de manière à réduire le nombre de transitions dans la *SFSMD* équivalente.

HSFSMD pour l'optimisation temporelle d'une description

Les propriétés caractéristiques d'une *HSFSMD* conforme à notre modèle autorisent un certain nombre de transformations, notamment [8] :
- la fusion de séquences de calculs associés à des transitions différentes afin de paralléliser ou de chaîner leurs opérations ;
- l'élimination d'états lorsque les transitions qui s'y rattachent ne sont associées à aucun calcul ;
- la fusion de boucles imbriquées.

Les figures 3.9 et 3.10 décrivent deux règles de transformation permettant la suppression d'un état, soit en fusionnant des calculs portés par deux transitions successives (figure 3.9), soit en extrayant un calcul effectué en début de boucle et en le reportant à la fois sur la transition d'entrée dans la boucle et sur les transitions de poursuite de cette boucle (figure 3.10). L'optimisation complète du modèle *HSFSMD* du calcul de distance de Manhattan est présentée en figure 3.11.

FIG. 3.7 – Calcul de distance de Manhattan : primitives de synthèse de haut niveau et modèle HSFSMD

```
(S1) ;
loop(label → MacroBlocLoop,
     body →
       (S2) ;
       exit(loopname → MacroBlocLoop, condition → tmp1) ;
       (S3) ;
       loop(label → LigneLoop,
            body →
              (S4) ;
              exit(loopname → LigneLoop, condition → tmp2) ;
              (S5) ;
              exit(loopname → LigneLoop, condition → depassement) ;
              (S6) ;
            ) ;
       exit(loopname → MacroBlocLoop, condition → depassement) ;
       (S7)
   )
```

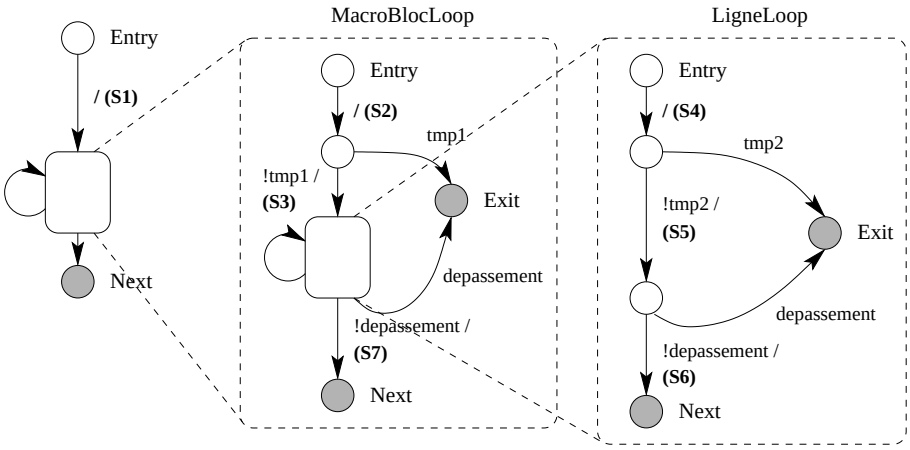
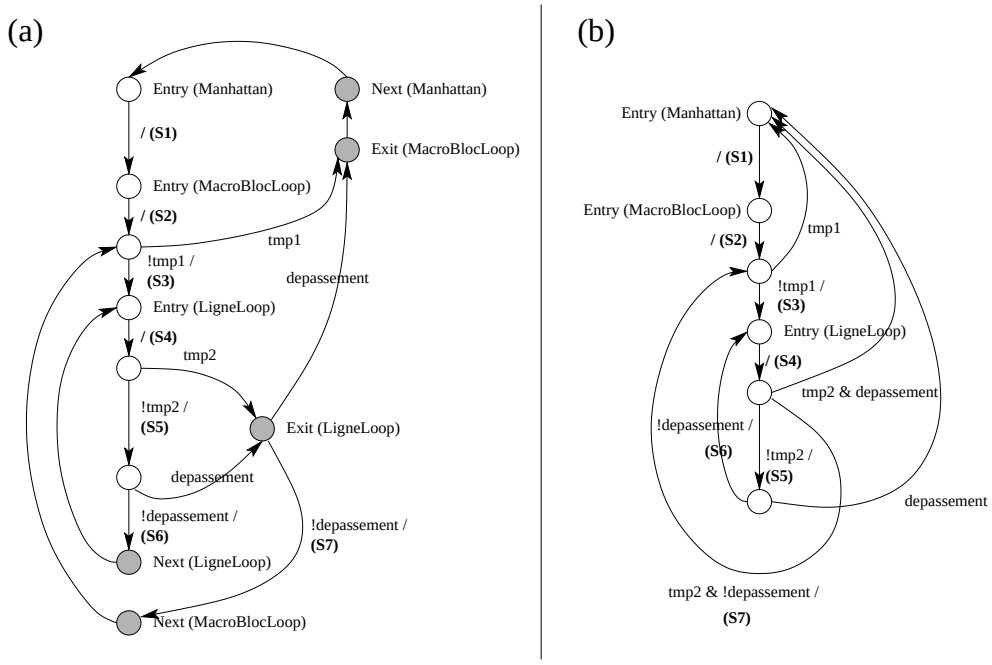


FIG. 3.8 – Calcul de distance de Manhattan : relation entre HSFSMD et SFSMD



HSFSMD pour l’expression des contraintes de temps

Après avoir optimisé le modèle en réduisant le nombre d’états et en fusionnant les boucles, les contraintes de temps peuvent être spécifiées en associant un nombre de périodes d’horloge à chaque transition sur laquelle des calculs sont effectués [8]. Afin d’assurer la flexibilité du comportement temporel d’un composant virtuel comportemental, le concepteur pourra définir un ensemble de paramètres de personnalisation des contraintes de temps et exprimer la durée de chaque transition en fonction de ces paramètres.

Dans la notation symbolique des primitives de synthèse de haut niveau, nous insérons une primitive $Time$ qui associe à chaque séquence de primitives de traitement une durée ($delay$) sous la forme d’une expression statique à valeur entière :

$$Time ::= \mathbf{time}(\text{delay} \rightarrow Expr, \text{body} \rightarrow Seq_{Data})$$

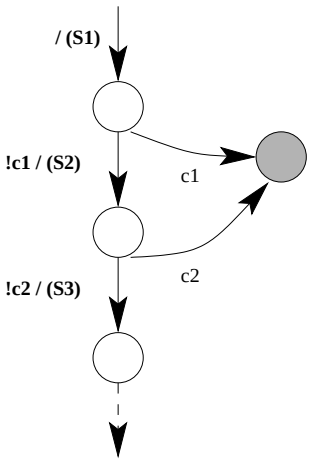
Ensuite, il convient de définir, pour chaque transition dont les calculs consomment des données d’entrée et produisent des données de sortie, la date d’arrivée de chaque donnée et la date de validité de chaque sortie [81]. Ces dates sont exprimées en périodes d’horloge relativement à la date de début de la transition courante. Les primitives $Read$ et $Write$ sont introduites afin de modéliser les lectures et écritures de données. Chacune associe à une entrée/sortie (identifiant simple ou élément de tableau) une date sous la forme d’une expression statique à valeur entière :

$$\begin{aligned} Read &::= \mathbf{read}(\text{input} \rightarrow Obj, \text{date} \rightarrow Expr) \\ Write &::= \mathbf{write}(\text{output} \rightarrow Obj, \text{date} \rightarrow Expr) \end{aligned}$$

FIG. 3.9 – Fusion de transitions d'une HSFSMD à l'intérieur d'un même niveau de boucle

Version originale

```
(S1) ;
exit(loopname→ L,
     condition→ c1) ;
(S2) ;
exit(loopname→ L,
     condition→ c2) ;
(S3) ;
```



Version optimisée

```
(S1) ;
if(test→
     eval(operation→ "not",
          operands→ <c1>),
   then→ (S2)) ;
exit(loopname→ L,
     condition→
       eval(operation→ "or",
            operands→ <c1, c2>)) ;
(S3) ;
```

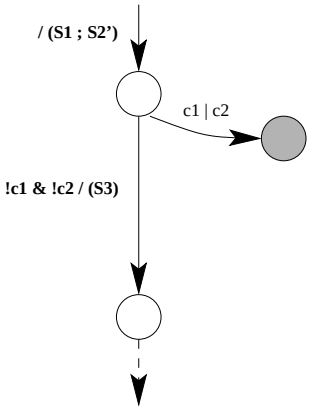


FIG. 3.10 – Fusion de transitions d'une HSFSMD à travers un niveau de boucle

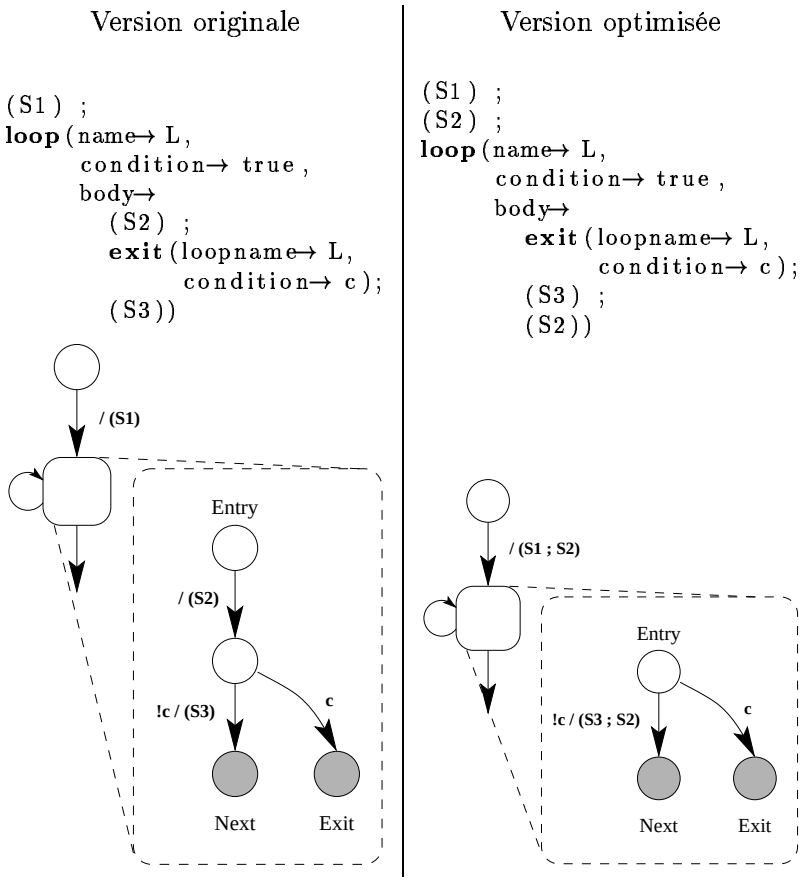


FIG. 3.11 – Calcul de distance de Manhattan : HSFSMD optimisée

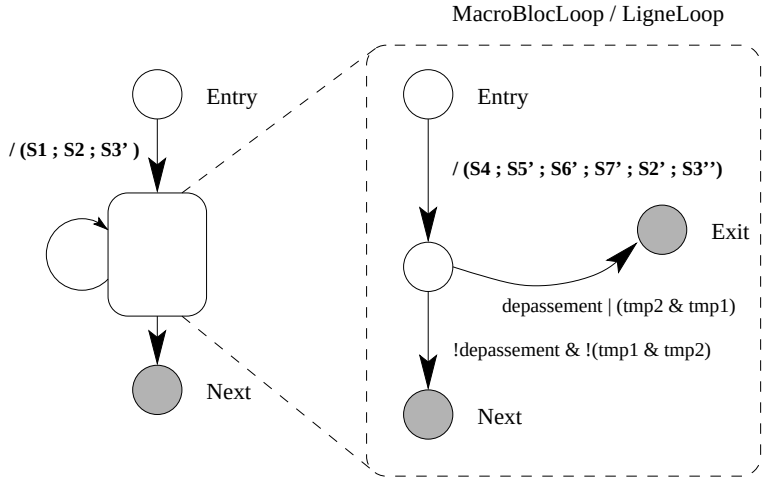
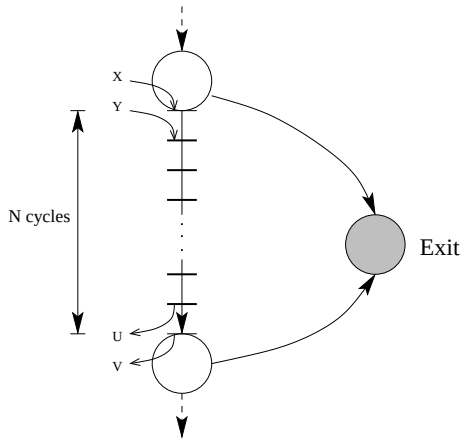


FIG. 3.12 – Modélisation des contraintes de temps et d'entrée/sortie par annotation des primitives de synthèse de haut niveau

```
exit(loopname→ L, condition→ c1) ;
time(delay→ N,
     body→
          read(input→ x, date→ 0) ;
          read(input→ y, date→ 1) ;
          (S) ;
          write(output→ u, date→ N−1) ;
          write(output→ v, date→ N)) ;
exit(loopname→ L, condition→ c2) ;
```



Un exemple de transition avec contraintes de temps et d'entrée/sortie est présenté en figure 3.12.

3.3 Des primitives de description comportementales aux primitives de synthèse de haut niveau

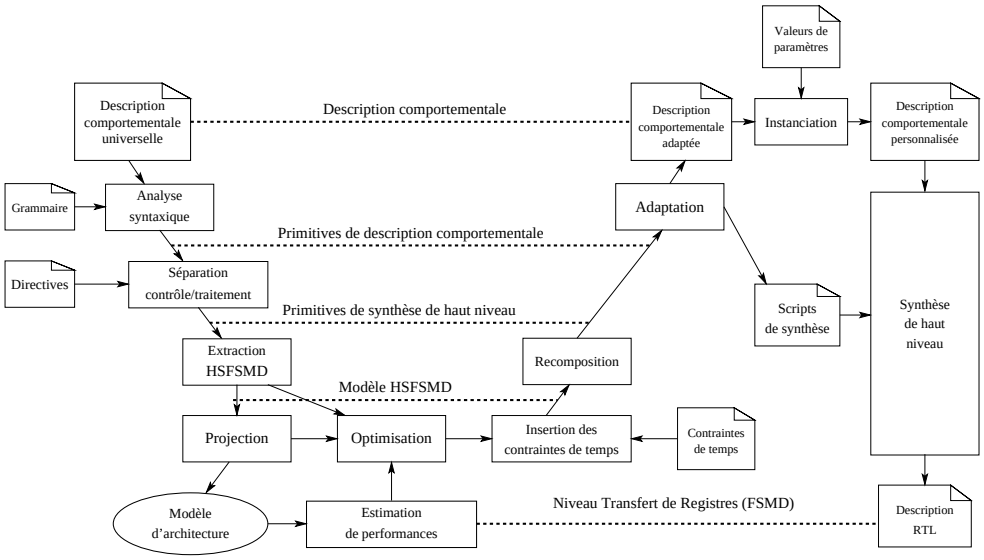
3.3.1 Flot de pré-synthèse de haut niveau

La spécification à base de primitives de synthèse de haut niveau impose au concepteur de séparer explicitement les aspects du comportement liés au contrôle et ceux liés au traitement. Ce style d'écriture revient ainsi à insérer dans la description des informations de bas niveau sur le déroulement temporel des calculs au sein de l'architecture finale.

Dans le flot de conception de composants virtuels comportementaux tels que nous l'avons présenté dans le chapitre précédent, le raffinement d'une description comportementale avec entrées/sorties au niveau transaction en une description avec entrées/sorties au cycle près se décompose ainsi en quatre étapes :

1. séparation du contrôle et du traitement et représentation sous forme de *HSFSMD* ;

FIG. 3.13 – Flot en V pour la synthèse de haut niveau



2. optimisation temporelle à l'aide du modèle *HSFSMD* ;
3. répartition des contraintes de temps sur les transitions du modèle ;
4. traduction de la description dans le langage et le style d'écriture supportés par un outil de synthèse de haut niveau.

Nous pouvons ainsi représenter la succession des étapes de raffinement d'une description comportementale à l'aide du flot en V représenté en figure 3.13. La première branche du V consiste à insérer progressivement des détails d'implantation dans une description universelle du comportement. La seconde branche permet de faire remonter ces détails d'implantation dans une description adaptée à un outil.

A partir d'une même description comportementale, l'étape de séparation du contrôle et du traitement autorise une variété de choix qui auront des incidences plus ou moins fortes sur les performances de l'architecture.

3.3.2 Primitives de description comportementale

Les primitives de description comportementale sont un sur-ensemble des primitives de synthèse de haut niveau présentées dans la section précédente. Nous y retrouverons les mêmes instructions, mais munies d'une syntaxe moins restrictive de manière à augmenter la lisibilité des descriptions :

$$\begin{aligned} Prim_{Beh} &::= Assign | If_{Beh} | Case_{Beh} | Loop_{Beh} | For_{Beh} | Next | Exit | PCall \\ Seq_{Beh} &::= Prim_{Beh} \{ \; ; \; Prim_{Beh} \} \end{aligned}$$

La définition de l'instruction d'affectation ($Assign$) reste inchangée. Les branches des instructions conditionnelles (If_{Beh}, $Case_{Beh}$) et le corps des boucles ($Loop_{Beh}$,

For_{Beh}) peuvent à présent contenir des séquences d'instructions quelconques. Les bornes et le pas d'itération d'une boucle *for* ne sont plus tenus d'être des expressions statiques.

En plus des boucles infinies et des boucles à compteur, les langages proposent généralement deux autres types de boucles : les boucles *while*, avec test d'une condition de poursuite au début de chaque itération ; les boucles *do...while* (langage *C*) avec test en fin d'itération. La primitive $Loop_{Ctrl}$ définie plus haut est ainsi redéfinie en adjoignant à la boucle infinie ($Forever$) deux nouvelles primitives $While$ et $DoWhile$:

$$
\begin{aligned}
Loop_{Beh} &::= Forever|While|DoWhile \\
Forever &::= \textbf{loop}(\text{name} \rightarrow Label, \text{body} \rightarrow Seq_{Beh}) \\
While &::= \textbf{while}(\text{name} \rightarrow Label, \text{condition} \rightarrow Expr, \text{body} \rightarrow Seq_{Beh}) \\
DoWhile &::= \textbf{dowhile}(\text{name} \rightarrow Label, \text{condition} \rightarrow Expr, \text{body} \rightarrow Seq_{Beh})
\end{aligned}
$$

Les conditions portées par les instructions de branchement peuvent être des expressions booléennes quelconques :

$$
\begin{aligned}
Next_{Beh} &::= \textbf{next}(\text{loopname} \rightarrow Label, \text{condition} \rightarrow Expr) \\
Exit_{Beh} &::= \textbf{exit}(\text{loopname} \rightarrow Label, \text{condition} \rightarrow Expr)
\end{aligned}
$$

3.3.3 Séparation contrôle/traitement

La séparation du contrôle et du traitement consiste à choisir, pour chaque primitive ou combinaison de primitives de description comportementale une combinaison de primitives de synthèse de haut niveau de comportement équivalent.

Ce choix peut s'effectuer tout d'abord en fonction d'un objectif de performances (vitesse, surface), qui conduira par exemple à dérouler ou ne pas dérouler une boucle selon le degré de parallélisme que le concepteur souhaite exploiter. Ensuite, la connaissance des *cas de fonctionnement* du composant – par exemple le nombre maximum d'itérations d'une boucle non déroulable – permet d'exploiter des propriétés du comportement que la seule analyse de la description ne permet pas toujours de déterminer, et d'effectuer des transformations – par exemple le déroulage forcé d'une boucle – que les outils de synthèse savent rarement traiter.

Boucles et branchements – Le choix du déroulage ou du non-déroulage d'une boucle déroulable est du ressort du concepteur. Pour les boucles dont le nombre d'itérations n'est pas défini statiquement, il est possible de forcer le déroulage lorsque le nombre d'itérations est borné en valeur supérieure.

Ce forçage consiste à remplacer l'instruction de boucle par une boucle *for* dont le compteur évolue sur un intervalle $[1 \ldots N]$ – N étant le nombre maximal d'itérations susceptibles d'être effectuées – et à exécuter le corps de boucle conditionnellement jusqu'à ce que la condition d'arrêt soit vérifiée (tableau 3.5).

Les boucles à ne pas dérouler sont converties en boucles infinies avec insertion d'une instruction de sortie conditionnelle dans le corps de boucle. Pour chaque instruction de branchement relative à une boucle ne devant pas être déroulée, l'expression correspondant à la condition de branchement est isolée dans une variable booléenne

TAB. 3.5 – Déroulage forcé des boucles non déroulables

Primitives de description comportementale	Primitives de synthèse de haut niveau
while(name→ L, test→ e_L, body→ (S))	**set**(left→ tmp, right→ true) ; **for**(name→ L, ctr→ i, first→ 1, last→ N, step→ 1, body→ **if**(test→ e_L, then→ **set**(left→ tmp, right→ false)) ; **if**(test→ tmp, then→ (S)))
dowhile(name→ L, test→ e_L, body→ (S))	**set**(left→ tmp, right→ true) ; **for**(name→ L, ctr→ i, first→ 1, last→ N, step→ 1, body→ **if**(test→ tmp, then→ (S))) **if**(test→ e_L, then→ **set**(left→ tmp, right→ false)) ;
for(name→ L, ctr→ i, first→ e_1, last→ e_2, step→ e_3, body→ (S))	**set**(left→ i, right→ e_1) ; **for**(name→ L, ctr→ j, first→ 1, last→ N, step→ 1, body→ **if**(test→ **eval**(operation→ "≤", operands→ <i, e_2>), then→ (S)) ; **set**(left→ i, right→ **eval**(operation→ "+", operands→ <i, e_3>)))

intermédiaire. Le tableau 3.6 donne une traduction pour chaque type de boucle dans le cas le plus général.

Expressions et instructions d'affectations – L'analyse des dépendances de données n'est généralement pas suffisante pour établir un *DFG* de manière unique à partir d'une expression : la construction d'un arbre d'opérations suppose l'existence de règles régissant l'ordre d'évaluation des sous-expressions. Ces règles portent d'une part sur la priorité des opérateurs et d'autre part sur le choix arbitraire d'un ordre d'évaluation – par exemple de la gauche vers la droite – lorsque des opérateurs associatifs et/ou commutatifs de même priorité sont combinés.

La seule manière de produire de manière non ambiguë un arbre d'opérations à partir d'une séquence d'instructions d'affectation consiste à utiliser une notation complètement parenthésée des expressions arithmétiques. Le concepteur dispose alors de la possibilité de placer les parenthèses selon différents objectifs :

- pour une optimisation en vitesse, une expression doit être mise sous la forme d'un arbre équilibré d'opérations.
- pour une optimisation en surface, on privilégiera une structure mettant en évidence des motifs répétitifs de calcul.

Instructions conditionnelles – Les instructions conditionnelles peuvent être transformées en séquences d'instructions équivalentes en fonction du type de sous-séquences rencontrées dans les branches.

Nous reposons pour cela sur une propriété de distributivité des instructions conditionnelles exprimée dans le tableau 3.7 dans le cas de l'instruction If. En appliquant cette propriété, nous pouvons transformer une description de manière à restreindre les types d'instructions conditionnelles aux trois cas suivants : (1) celles dont les branches sont des séquences de primitives de traitement ; (2) celles dont les branches sont des séquences d'instructions de branchement ; (3) celles dont les branches contiennent chacune au plus une instruction de boucle à ne pas dérouler.

Lorsque seules des instructions d'affectation apparaissent dans une instruction conditionnelle, le concepteur a le choix entre deux styles d'écriture : (1) l'*évaluation conditionnelle* consiste à laisser telles quelles les instructions figurant dans les branches ; (2) l'*affectation conditionnelle* consiste à pré-évaluer les expressions situées dans les branches en les affectant à des variables intermédiaires, de telle sorte que les branches ne contiennent plus que des affectations de variable à variable (tableau 3.8). La première solution impose l'ordonnancement des expressions situées dans les branches *après* l'expression de test de l'instruction conditionnelle, ce qui limite le parallélisme exploitable mais autorise le partage des ressources en tenant compte du caractère mutuellement exclusif des différentes branches. La seconde solution offre une plus grande liberté aux outils de synthèse pour paralléliser les opérations et autorise de plus grandes performances en vitesse. Elle augmente cependant la quantité de calculs à effectuer et s'avère peu économique en surface. Elle n'est à recommander que lorsque la vitesse de traitement devient critique.

Lorsque seules des instructions de branchement apparaissent dans une instruction conditionnelle, l'expression de test de l'instruction conditionnelle est fusionnée aux

TAB. 3.6 – Réécriture des boucles à ne pas dérouler

Primitive de description comportementale	Primitive de synthèse de haut niveau
while(name→ L, condition→ e_L, body→ (S))	**set**(left→ tmp, right→ e_L) ; **loop**(name→ L, body→ **exit**(loopname→ L, condition→ tmp) ; (S) ; **set**(left→ tmp, right→ e_L))
dowhile(name→ L, condition→ e_L, body→ (S))	**loop**(name→ L, body→ (S) ; **set**(left→ tmp, right→ e_L) ; **exit**(loopname→ L, condition→ tmp))
for(name→ L, ctr→ i, first→ e_1, last→ e_2, step→ e_3, body→ (S))	**set**(left→ i, right→ first) ; **set**(left→ tmp, right→ **eval**(operation→ ">", operands→ <i, e_2>)) ; **loop**(name→ L, body→ **exit**(loopname→ L, condition→ tmp) ; (S) ; **set**(left→ i, right→ **eval**(operation→ "+", operands→ <i, e_3>)) ; **set**(left→ tmp, right→ **eval**(operation→ ">", operands→ <i, e_2>)))

TAB. 3.7 – Distributivité des instructions conditionnelles

Instructions conditionnelles	Distribution des sous-séquences
if(test→ c, then→ (S11 ; S12), else→ (S21 ; S22))	**if**(test→ c, then→ (S11), else→ (S21)) ; **if**(test→ c, then→ (S12), else→ (S22))

TAB. 3.8 – Choix d'écriture des instructions conditionnelles

Evaluation conditionnelle	Affectation Conditionnelle
if(test→ e_c , then→ **set**(left→ x, right→ e_1), else→ **set**(left→ x, right→ e_2))	**set**(left→ tmp0, right→ e_c) ; **set**(left→ tmp1, right→ e_1) ; **set**(left→ tmp2, right→ e_2) ; **if**(test→ tmp0, then→ **set**(left→ x, right→ tmp1), else→ **set**(left→ x, right→ tmp2))

TAB. 3.9 – Instruction de branchement dans une instruction conditionnelle

Primitives de description comportementale	Primitives de synthèse de haut niveau
if(test→ e_{if} , then→ **exit**(loopname→ L, condition→ e_{exit}))	**set**(left→ tmp, **eval**(operation→ "and", operands→ { e_{if} , e_{exit} })) ; **exit**(loopname→ L, condition→ tmp) ;

conditions de branchement (tableau 3.9).

Une instruction conditionnelle ne contenant que des boucles infinies prend le sens d'une instruction de contrôle modélisant l'entrée conditionnelle dans une boucle. L'expression de test est reportée sur le champ *condition d'entrée* de la boucle par l'intermédiaire d'une variable booléenne (tableau 3.10).

Résumé

Les figures 3.14, 3.16 et 3.15 décrivent les étapes de la transformation d'une spécification à base de primitives de description comportementale vers une spécification à base de primitives de synthèse de haut niveau. Le traitement complet d'une description peut nécessiter plusieurs passes de transformation.

La transformation d'une primitive prend en compte des propriétés syntaxiques qui conditionnent les transformations autorisées (par exemple, une boucle peut être déroulable ou non). Des propriétés dynamiques connues du concepteur (cas de fonctionnement) peuvent guider la simplification de certaines primitives.

TAB. 3.10 – Boucles non déroulées dans les instructions conditionnelles

Primitives de description comportementale	Primitives de synthèse de haut niveau
if(test→ e_{if}, then→ **loop**(name→ L, condition→ c, body→ ...))	**set**(left→ tmp, right→ **eval**(operation→ "and", operands→ {e_{if}, c})) ; **loop**(name→ L, condition→ tmp, body→ ...)

FIG. 3.14 – Des primitives de description comportementales aux primitives de synthèse de haut niveau : transformation des instructions d'affectation

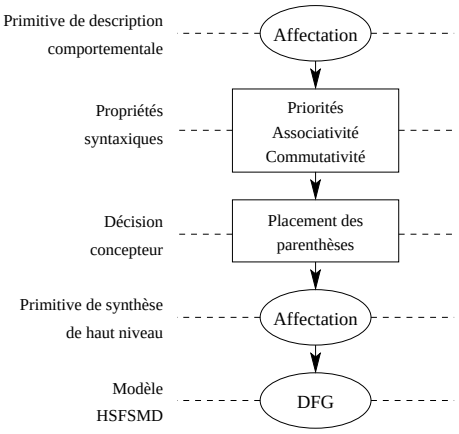


FIG. 3.15 – Des primitives de description comportementales aux primitives de synthèse de haut niveau : transformation des instructions conditionnelles

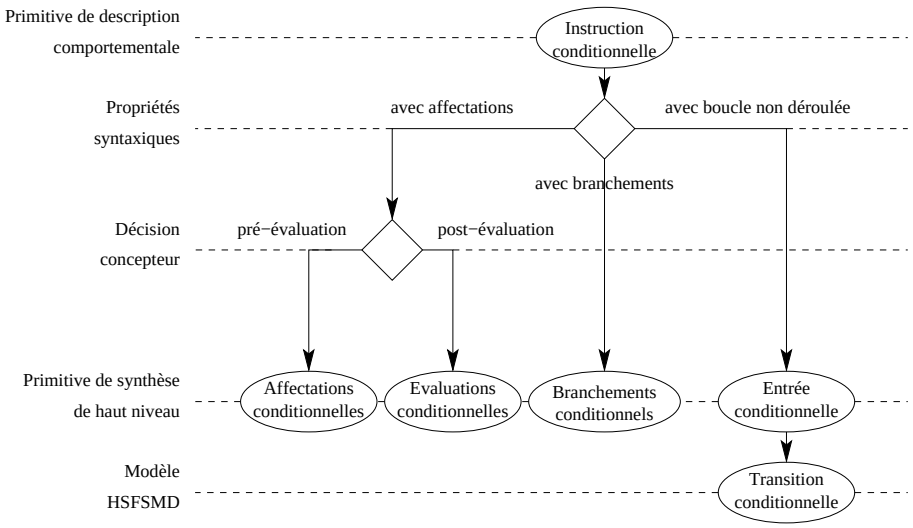
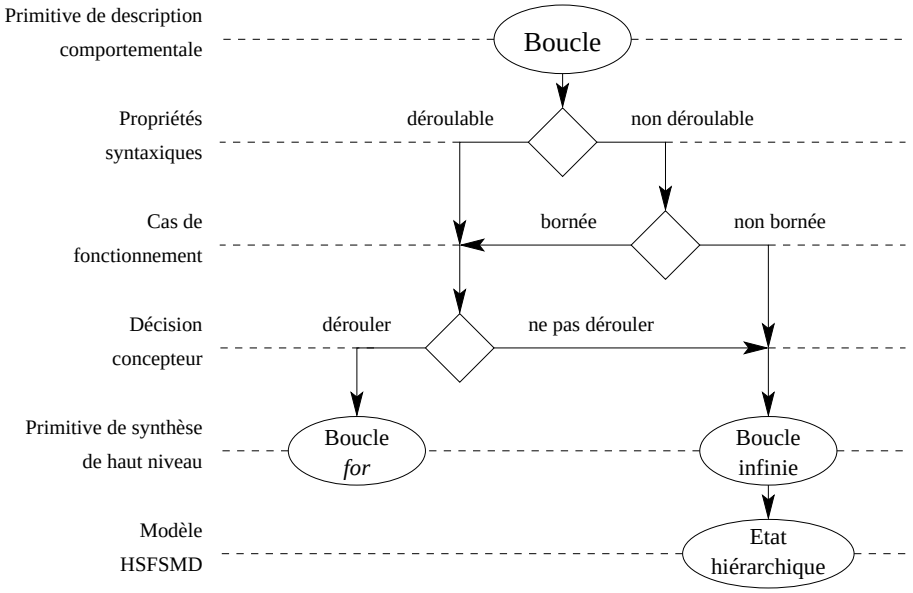


FIG. 3.16 – Des primitives de description comportementales aux primitives de synthèse de haut niveau : transformation des boucles



Bilan

Les chapitres 2 et 3 nous ont permis de mettre en place une démarche méthodologique pour la conception et la synthèse de composants virtuels comportementaux. Pour ce faire, nous avons proposé un flot de conception descendant dont le point de départ est une spécification fonctionnelle du composant à réaliser, et dont le point d'arrivée est une description comportementale avec entrées/sorties au cycle près destinée à être traitée par des outils de synthèse de haut niveau en respectant des contraintes de temps spécifiées de manière non ambiguë.

Les étapes successives du flot proposé s'attachent à raffiner les propriétés temporelles du comportement d'un composant en respectant une succession de modèles formels. La correspondance entre classes de spécifications et modèles de représentation est résumée par le tableau 3.11. Nous distinguons deux ensembles de modèles : ceux qui décrivent le comportement temporel *externe* du composant – c'est-à-dire les règles d'interfaçage du composant avec son environnement – et ceux qui décrivent son comportement fonctionnel et temporel *interne*, c'est-à-dire l'organisation relative des calculs et des entrées/sorties au cours du temps.

Ces modèles servent les trois exigences de *réutilisabilité* d'un composant :

1. *Flexibilité* : en fournissant tout d'abord un support sémantique pour l'expression de paramètres de personnalisation du composant et pour la définition des contraintes de performances ;
2. *Prédictibilité* : en systématisant l'extraction de propriétés de complexité, de topologie et de temps pour l'estimation de performances en fonction des valeurs

TAB. 3.11 – Modèles sémantiques associés à chaque niveau de spécification

Spécification	Externe	Interne
Fonctionnelle	Transformationnel	Mathématique
Algorithmique	Transformationnel	DFG/CDFG
Comportementale au niveau transaction	Flot de données synchrone	HSFSMD
Comportementale au cycle près	Evénements discrets	HSFSMD annotée
Transferts de registres	Evénements discrets	FSMD

de paramètres ;

3. *Interopérabilité* : en unifiant les modèles de représentation des différents outils de synthèse de haut niveau.

L'utilisation de modèles formels permet également de fiabiliser le flot de conception en fournissant un support pour la vérification du fonctionnement d'un composant à différents niveaux d'abstraction.

Délivrables d'un composant virtuel comportemental

La plupart des spécifications produites au cours de la conception d'un composant virtuel comportemental (schématisées en figure 2.12 page 55) ont un rôle à jouer dans le flot de réutilisation. En plus de la spécification comportementale synthétisable, les spécifications de plus haut niveau permettent notamment d'évaluer un composant avant achat, d'en vérifier le fonctionnement – seul ou intégré dans le système cible –, d'en estimer les performances en fonction des paramètres de l'application.

Le tableau 3.12 met en relation la liste des principaux délivrables avec les étapes du flot de réutilisation. Le format des différents délivrables est donné à titre d'exemple, les langages candidats pour les spécifications de niveau système étant encore susceptibles d'évoluer au cours des prochaines années.

Le *générateur* est un logiciel d'accompagnement dont le premier rôle est d'affecter une valeur à chacun des paramètres du composant et de vérifier la validité des valeurs fournies. Son second rôle est d'adapter le langage et le style d'écriture de la description comportementale synthétisable à l'outil de synthèse de haut niveau ciblé. Pour ce faire, il doit disposer d'une bibliothèque de règles de traduction spécifiques à chaque outil existant.

A cette liste, il convient d'ajouter les documents détaillant la fonctionnalité et l'interface du composant, ses paramètres, les procédures de personnalisation, vérification et synthèse.

Protection de la propriété intellectuelle

Dans la présentation de notre méthodologie, nous avons laissé de côté la question de la protection de la propriété intellectuelle. Les difficultés déjà mentionnées pour les *IP soft* sont ici amplifiées par le haut niveau d'abstraction de la description synthétisable. S'il est encore envisageable de crypter les descriptions afin d'en délivrer des modèles simulables pour évaluation avant achat, la livraison du modèle synthétisable nécessite de repenser la chaîne de distribution des composants virtuels. La méthode la plus simple et la plus sûre envisageable à l'heure actuelle est celle déjà appliquée par un grand nombre de fournisseurs d'*IP* : elle consiste à réaliser la synthèse de haut niveau chez le fournisseur et à ne délivrer au client final qu'une *netlist* optimisée tenant compte des paramètres et des contraintes propres à l'application cible.

Une méthode alternative consisterait à mettre en œuvre une infrastructure de distribution interactive permettant à l'utilisateur de réaliser la synthèse de haut niveau à distance. Dans ce modèle, l'utilisateur est autorisé à rapatrier directement l'ensemble des délivrables non synthétisables. Il peut effectuer différentes passes de synthèse de

TAB. 3.12 – Délivrables d'un composant virtuel comportemental

Délivrable	Format	Flot de réutilisation
Générateur	Logiciel d'accompagnement	Personnalisation, adaptation
Banc de test,	C/C++/SystemC VHDL/Verilog	Evaluation, vérification du composant seul
Vecteurs de test	–	
Spécification algorithmique en virgule flottante	C/C++/SystemC	Evaluation fonctionnelle
Spécification algorithmique en virgule fixe	C/C++/SystemC	Etude de précision
Spécification comportementale au niveau transaction	SystemC	Vérification fonctionnelle au niveau système
Interfaces *SDF*	SystemC	Raffinement des communications
Spécification comportementale au cycle près	SystemC VHDL/Verilog	Vérification temporelle au niveau système
Spécification comportementale détaillée	SystemC VHDL/Verilog (Primitives de synthèse de haut niveau)	Synthèse de haut niveau

haut niveau pour différents jeux de paramètres et contraintes. En fonction des rapports de synthèse obtenus, il sélectionne la solution *RTL* qu'il souhaite implanter et télécharge la description *VHDL* ou *Verilog* correspondante.

Dans le cadre d'une telle infrastructure, le générateur et la description comportementale synthétisable ne font plus partie des délivrables. Le générateur reste accessible de manière indirecte. Dans la liste présentée dans le tableau 3.12, ces deux éléments cèdent la place à la description synthétisable, *RTL* ou de plus bas niveau, obtenue après synthèse de haut niveau à distance.

La problématique de la distribution des composants virtuels comportementaux et de la protection de la propriété intellectuelle nécessiteront des travaux de recherche supplémentaires. Ces travaux représentent un complément indispensable à la méthodologie de conception développée dans le cadre de notre thèse.

Outils d'aide à la spécification et à la synthèse

Le flot de conception et de synthèse de composants virtuels comportementaux que nous avons présenté dans ce chapitre tend à combler le fossé entre le niveau d'abstraction recommandé pour la spécification au niveau système et le niveau dit *comportemental* supporté par les outils de synthèse de commerce.

Ce flot tend vers une approche de *synthèse système* reposant de préférence sur un flot de raffinement *interactif* plutôt que sur un flot entièrement *automatique*. L'idée est de proposer à un concepteur ou utilisateur de composants virtuels des outils d'*aide à la spécification* et d'*aide à la réutilisation* permettant à un opérateur humain d'accéder aux informations pertinentes relatives au composant en cours de conception, ou à l'instance du composant en cours d'intégration, et de mesurer l'influence de ses choix d'implantation sur les propriétés du composant.

La mise en œuvre de notre méthodologie sur des applications complexes nécessite ainsi le développement d'outils reposant sur les modèles formels présentés. Ces outils devront ainsi permettre : (1) l'analyse de graphes de dépendances et le choix de motifs de calcul, (2) l'élaboration et l'optimisation d'un modèle *HSFSMD* d'une description comportementale, (3) la traduction de ce modèle vers différents langages et styles d'écriture propres aux outils existants, (4) la vérification syntaxique et sémantique d'une description comportementale en vue de la synthèse.

Un premier pas vers l'outillage complet de la méthodologie est représenté par l'outil *U-HLS Editor*, développé au *LESTER*. Cet outil fournit les fonctionnalités d'un éditeur de texte orienté par la syntaxe et la sémantique [80, 117]. Il autorise l'écriture d'une description comportementale dans différents langages et associe à chaque langage et à chaque outil de synthèse cible un ensemble de règles syntaxiques et sémantiques. Ces règles permettant de vérifier automatiquement d'une part la conformité d'une description comportementale avec les primitives de synthèse de haut niveau et d'autre part la conformité du style d'écriture avec un outil de synthèse donné.

Dans le cadre de notre collaboration avec le *Centre National d'Etudes Spatiales* et la société *Astrium*, nous avons donné la priorié à la mise en application de la méthodologie sur un exemple concret plutôt qu'au développement d'outils. L'expérience

réalisée a consisté à concevoir et synthétiser un composant virtuel comportemental pour la compression d'images embarquée. Cette expérience fait l'objet de la deuxième partie de ce mémoire.

Deuxième partie

Conception et Synthèse d'un Composant Virtuel Comportemental pour la Transformation en Ondelettes Discrète 2D

Chapitre 4

Transformations en Ondelettes pour la Compression d'Images Embarquée

Le travail présenté dans cette partie a été mené en collaboration avec le *Centre National d'Etudes Spatiales* (*CNES*) et la société *Astrium*. Elle s'inscrit dans le cadre d'une étude visant à adapter les systèmes de compression d'images embarqués à bord des satellites d'observation de la Terre aux nouveaux besoins des applications qui utilisent ces images.

Que ce soit dans le domaine de l'imagerie spatiale ou dans les applications multimedia grand public, les systèmes d'acquisition et de traitement des images manipulent en effet des volumes de données croissants. Cette augmentation reflète l'évolution conjointe des technologies d'acquisition des images – notamment les capteurs *CCD* qui présentent des résolutions de plus en plus fines – et des besoins des applications exploitant ces technologies. Entre les deux générations de satellites d'observation de la Terre *SPOT4* et *SPOT5* la résolution maximale des images panchromatiques est passée de 10 mètres à 2,5 mètres par pixel (mode *Haute Résolution Géométrique* de *SPOT5*, dit *Supermode* [99]).

Lors de l'élaboration du nouveau standard de codage d'images *JPEG2000*, les besoins recensés dans les différents profils applicatifs [95] considéraient des images dont la taille maximale atteignait 4000×4000 pixels pour la photographie numérique, 10000×10000 pour l'imagerie médicale et $24000 \times \infty$ pour la télédétection optique. Le nombre de composantes d'une image (luminance, Rouge-Vert-Bleu, *etc.*) introduit un facteur multiplicatif dans ce volume de données. Ce nombre peut atteindre 500 dans le cas de l'imagerie multispectrale et hyperspectrale. Enfin la précision de la valeur d'intensité des pixels ne cesse d'augmenter, elle varie de 8 à 16 bits par composante en photographie numérique et peut atteindre 20 bits en télédétection optique.

L'accroissement des volumes de données se traduit par une augmentation des coûts de stockage des données images – sur des supports temporaires ou dans des bases d'images – et de transmission à travers les réseaux de communication. Cette dernière souffre en particulier de la largeur limitée des bandes passantes disponibles [98]. Dans

le domaine des applications grand public, la viabilité des réseaux terrestres et sans fil pour la transmission de données multimedia exige une optimisation des rendements de la communication, c'est-à-dire de la quantité d'information *pertinente* transmise par seconde.

Les techniques de compression d'images vont dans le sens d'une réduction significative des capacités nécessaire au stockage des données, et d'une adaptation de la quantité d'information à transmettre en fonction de la bande passante que l'on souhaite exploiter. Ce gain doit cependant être rapporté au surcoût résultant des techniques de compression elles-mêmes : d'une part la réduction du volume de données peut se traduire par des dégradations de la qualité des images, et d'autre part l'insertion d'un dispositif de compression dans un système embarqué peut altérer les performances en vitesse du système et augmenter de manière significative la complexité des architectures de traitement des données.

4.1 Contexte de l'étude : traitement bord et compression d'images spatiales

Comme nous l'avons indiqué plus haut, les images spatiales se caractérisent par leur grande taille [95,98]. Les satellites d'observation de la Terre sont un cas particulier de dispositifs d'imagerie où l'acquisition en continu des lignes de pixels se traduit par des images de hauteur infinie.

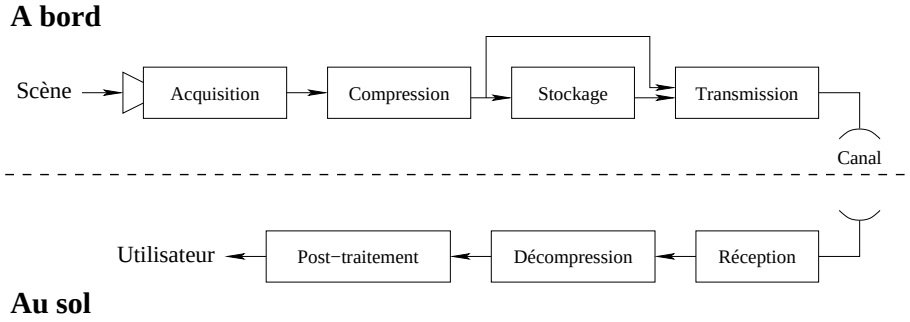
L'augmentation de la résolution des capteurs (de 10 mètres à 5 mètres par pixels entre les générations *SPOT4* et *SPOT5*), l'insertion du mode *Haute Résolution Géométrique* [99] (*HRG* ou *Supermode*, qui combine deux images échantillonnées à 5 mètres par pixel pour obtenir une nouvelle image à 2,5 mètres par pixels) pour une largeur de scène qui reste constante (60 kilomètres) aboutissent à des largeurs d'images pouvant atteindre 12000 pixels représentés chacun sur 8 à 10 bits. Les applications futures envisagent des images dont la largeur maximale sera de 24000 pixels avec une précision de 16 bits par pixel.

4.1.1 Spécificités des applications spatiales

Un schéma synoptique d'une chaîne image typique pour l'imagerie spatiale est représentée en figure 4.1 [98]. La compression des données images est rendue nécessaire en particulier par la nécessité de les stocker à bord pendant les périodes de non-visibilité du satellite depuis les stations de réception au sol. En effet, les contraintes spécifiques de l'implantation d'architectures à bord de véhicules spatiaux ont une influence significative sur le coût des mémoires embarquées. Deux points cruciaux sont la masse du matériel et sa consommation.

Tout d'abord, du fait des fortes contraintes mécaniques subies par les cartes électroniques, notamment lors du lancement, celles-ci doivent être rigidifiées à l'aide de structures métalliques. Une capacité mémoire de 200 Gbits nécessitera une masse de matériel embarqué de l'ordre de 30 kg [105]. Ensuite, du fait de l'absence de convection dans le vide spatial, la chaleur dissipée par les composants électroniques reste confinée dans la structure du satellite. Eviter les dysfonctionnement dus à l'échauffement des composants nécessite, outre la mise en place de dissipateurs thermiques, de limiter la

FIG. 4.1 – La chaîne image SPOT



puissance consommée par les circuits [105]. La puissance dissipée par l'électronique embarquée sur le satellite *SPOT5* est estimée à 90W contre 50W pour *SPOT4*.

Le coût lié à ces deux types de contraintes est estimé à 150 K€ par kilogramme de matériel embarqué, et à 60 K€ par watt, d'où l'importance d'optimiser les charges utiles tant en complexité, qu'en quantité de mémoire et en consommation [105].

Les contraintes de débit de transmission vers le sol jouent également un rôle dans le choix du taux de compression à appliquer. De *SPOT4* à *SPOT5*, le débit d'acquisition des pixels est multiplié par 4 pour un débit de transmission vers le sol qui n'est multiplié que par 2 [99]. Afin de conserver un bon compromis entre taux de compression et qualité des images, de nouveaux algorithmes de compression ont dû être mis en œuvre. Les taux de compression appliqués sont de 1,3 pour les générations de satellites *SPOT1* à *SPOT4*, 3 pour *SPOT5*, 4 à 12 pour la mission *Phobos*, 4 à 16 dans le cas de la mission d'observation lunaire *Clementine* [98, 99].

4.1.2 Compression d'images spatiales et standards

Aux contraintes fonctionnelles et matérielles liées à la compression d'images, s'ajoute un nouveau besoin lié à l'utilisation finale des images et au format de fichiers délivrés aux consommateurs. Le recours à des standards de codage d'images est une condition importante pour faciliter l'interfaçage entre les dispositifs de production des images et les applications cibles.

Actuellement, les données images *SPOT* sont majoritairement délivrées au format *GeoTIFF*, compatible avec la norme *TIFF* fondée sur des techniques de compression sans pertes [121]. Ce format n'est cependant pas le même que celui sous lequel les données binaires sont transmises du satellite vers le sol. Des algorithmes spécifiques de codage mis au point par le *CNES* sont employés [98, 99].

A partir des satellites *SPOT5*, *Phobos* et *Clementine*, les nouvelles contraintes de performance aboutissent au choix d'une chaîne de compression commune très proche du standard *JPEG* [99]. L'implication du *CNES* dans le comité de standardisation *JPEG2000* [95, 96] pour le codage des images fixes, et la prise en compte depuis mars 1999 des exigences du champ d'application télédétection optique dans le cahier des charges de ce nouveau standard [95], tendent vers une unification du format de codage

des images à bord des véhicules spatiaux avec le format de diffusion de ces images.

4.2 Compression d'images et ondelettes : le nouveau standard JPEG2000

4.2.1 Principes de la compression d'images

La compression de données en général se fixe pour objectif de fournir une représentation binaire aussi compacte que possible d'un ensemble d'informations [93]. Les performances d'une chaîne de compression s'expriment généralement en termes de *taux de compression* (TC) ou de *débit* (D) :

$$TC = \frac{\text{Nombre de bits des données source}}{\text{Nombre de bits des données compressées}}$$

$$D = \frac{\text{Nombre de bits des données compressées}}{\text{Nombre de symboles codés}} \text{ en bits par symbole}$$

Une chaîne de compression d'images se décompose typiquement en trois étapes successives nommées décorrélation, quantification et codage [93] :

Décorrélation – Les techniques de compression sont en général choisies en fonction de caractéristiques *a priori* de la structure des données à compresser. Dans le cas des images, l'hypothèse dominante considère que des pixels voisins sont fortement corrélés, ce qui signifie que la valeur d'intensité de chaque pixel n'apporte que peu d'information nouvelle par rapport à son voisinage [93]. Par conséquent, une chaîne de compression d'images typique débutera par une étape de décorrélation visant à extraire l'information pertinente de l'image en réduisant autant que faire se peut sa redondance spatiale.

La plupart des décorrélateurs reposent sur une transformation réversible de l'image, visant à obtenir une image transformée contenant majoritairement des valeurs nulles ou proches de zéro, l'information non redondante étant représentée sur un nombre réduit de valeurs significatives [93]. On distingue généralement trois types de méthodes :

1. les méthodes travaillant dans le domaine *spatial* comme le codage prédictif [93], ou *DPCM* (*Differential Pulse Coded Modulation*), employé avec succès dans les générations de satellites *SPOT1* à *SPOT4* [98], dont le principe consiste à exploiter les faibles variations d'intensité entre pixels voisins dans une image ;
2. les méthodes travaillant dans le domaine *fréquentiel*, reposant sur la transformation de FOURIER discrète, ou plus souvent la transformation en cosinus discrète (DCT) [92] et exploitant la concentration du contenu fréquentiel des images naturelles sur un nombre réduit de cœfficients de FOURIER dans les basses fréquences [93, 94] ;
3. les méthodes travaillant dans le domaine *spatio-fréquentiel*, permettant de tenir compte du caractère non-stationnaire du contenu fréquentiel d'une image : l'approche retenue dans le cas du standard *JPEG* consiste à décomposer l'image en blocs de 8×8 pixels et à appliquer une DCT indépendamment sur chaque bloc [94]. Ce type de décomposition a été mis en œuvre dans les satellites *SPOT5*,

Phobos, et *Clementine* [98,99]. Dans le standard *JPEG2000* [84,95,96], la transformation en ondelettes discrète (*DWT* pour *Discrete Wavelet Transform*) permet une analyse plus fine des détails d'une image en la décomposant en une série de *sous-bandes* [86, 100, 103], donnant chacune la répartition spatiale des détails selon une orientation donnée (horizontale, verticale, diagonale) et à une résolution donnée.

Codage – L'étape de décorrélation produit un ensemble ordonné de symboles, valeurs entières ou réelles, qui doit être codé sous forme binaire. D'après la théorie de l'information et du codage, la redondance d'information présente dans une séquence de symboles parfaitement décorrélée est de nature statistique [93]. Les techniques de codage *entropique* s'appliquent à réduire la redondance statistique d'information dans une image décorrélée de manière à produire une séquence binaire présentant un nombre de bits moyen par symbole aussi faible que possible.

Les techniques les plus largement employées consistent à associer à chaque symbole un mot-code dont le nombre de bits sera d'autant plus petit que le symbole à coder est fréquent [93]. C'est le principe du codage de HUFFMAN, utilisé notamment dans la norme *JPEG* [94], et qui permet d'approcher d'assez près la limite théorique [93].

Quantification – Afin d'atteindre des taux de compression élevés, une étape de quantification vient souvent s'insérer entre les étapes de décorrélation et de codage afin de réduire le nombre de symboles à coder. L'étape de quantification se traduit par une perte plus ou moins importante d'information, entraînant une dégradation plus ou moins significative de la qualité visuelle des images [93].

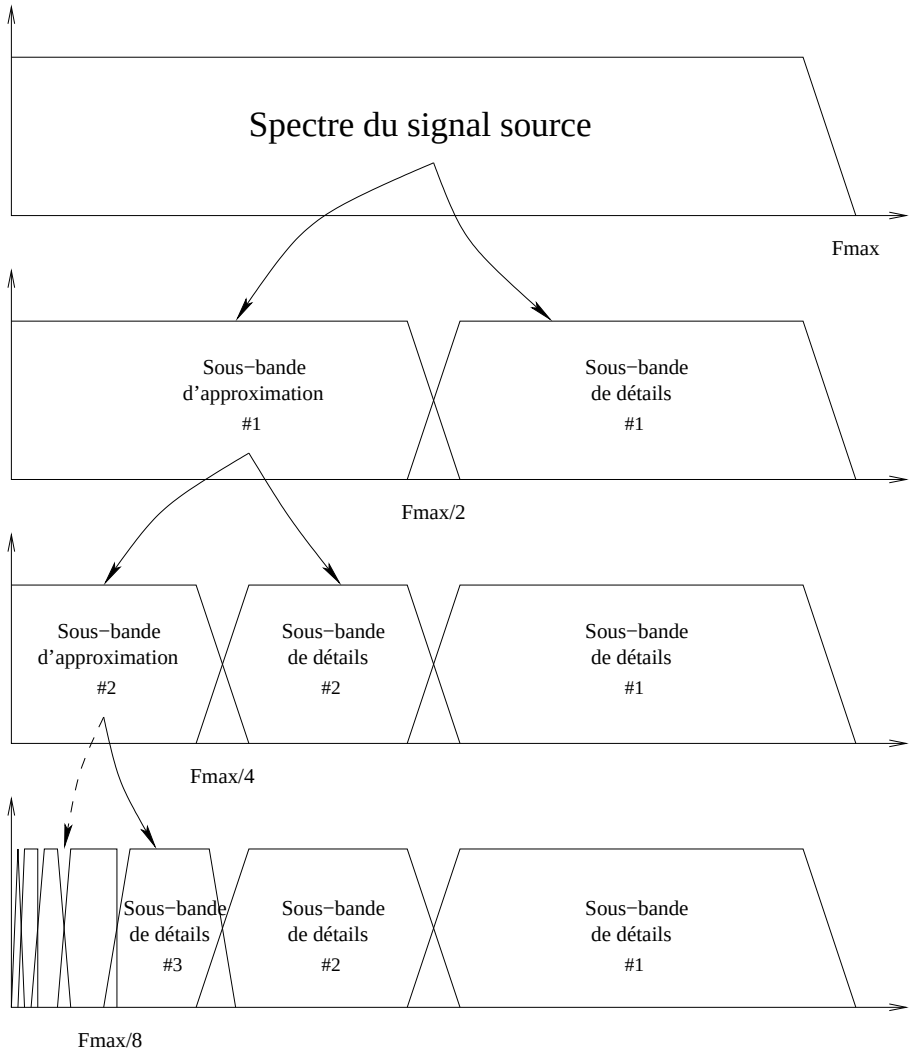
Cette étape est spécifique aux méthodes de compression dites *irréversibles*, ou méthodes *à pertes*, à la différence des méthodes *réversibles* où l'image d'origine peut être reconstruite fidèlement – c'est-à-dire au bit près – à partir des données compressées.

La donnée du taux de compression n'est par conséquent plus suffisante à caractériser les performances d'une chaîne de compression irréversible. Il convient d'y adjoindre une mesure de la distorsion introduite par le quantificateur, souvent exprimée comme l'erreur quadratique moyenne (ou puissance du bruit introduit par la compression) ou le rapport signal à bruit entre image source et image décompressée [93].

4.2.2 Analyse multirésolution et transformation en ondelettes discrète

La transformation en ondelettes discrète est issue de la théorie de l'analyse multirésolution, ou analyse espace/échelle par opposition à l'analyse temps/fréquence de FOURIER [100, 103]. A la différence de la transformée de FOURIER, qui représente uniquement l'information fréquentielle d'un signal, et de la transformée de FOURIER à fenêtre glissante [92], qui associe à chaque localisation spatiale un ensemble de composantes spectrales relatives à une fenêtre d'observation de taille fixe, l'analyse multirésolution propose une exploration progressive des détails constituant une image de la résolution la plus fine à la résolution la plus grossière [92, 100, 103]. Dans le principe, elle peut être comparée à une analyse temps/fréquence dans laquelle la largeur de la fenêtre d'observation ne serait plus fixe, mais adaptée à la résolution des détails que l'on cherche à analyser [92].

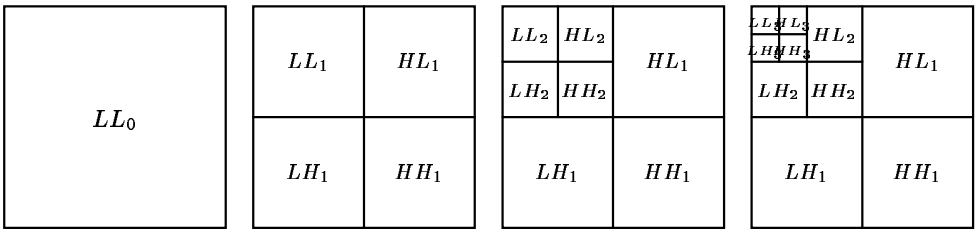
FIG. 4.2 – Principe de la décomposition en sous-bandes



La transformation en ondelettes discrète peut être rapprochée de la notion d'analyse en sous-bandes (figure 4.2) [92, 103] dans laquelle le spectre du signal à analyser est décomposé récursivement en une sous-bande de *détails* (hautes fréquences) et une sous-bande d'*approximation* (basses fréquences). Pour un signal à deux dimensions, ce ne sont pas deux, mais quatre sous-bandes qui sont extraites à chaque niveau de décomposition [86, 100, 103] : une sous-bande d'*approximation* (LL) contenant l'information basse fréquence selon les deux directions spatiales ; une sous-bande de *détails horizontaux* (LH) et une sous-bande de *détails verticaux* (HL) contenant respectivement l'information basse fréquence suivant les directions horizontale et verticale, et l'information haute fréquence suivant l'autre direction ; une sous-bande de *détails diagonaux* (HH) contenant les composantes haute fréquence suivant les deux directions.

La transformée en ondelettes d'une image représente chacune de ces sous-bandes dans le domaine spatial et fournit ainsi une cartographie des détails en termes de loca-

FIG. 4.3 – Décomposition en ondelettes 2D sur trois niveaux



lisation, résolution et direction. Le processus de décomposition récursive est représenté en figure 4.3 pour trois niveaux de résolution [100, 103].

La théorie de l'analyse multirésolution et les fondements mathématiques de la transformation en ondelettes discrète sont présentés plus en détail en annexe D.

4.2.3 De JPEG à JPEG2000

Le standard *JPEG* (*Joint Photographic Experts Group*) a connu une large adoption dans le domaine de la photographie numérique et de la diffusion d'images sur Internet [94] – on estime que 80% des images circulant sur le *web* sont au format *JPEG* [115].

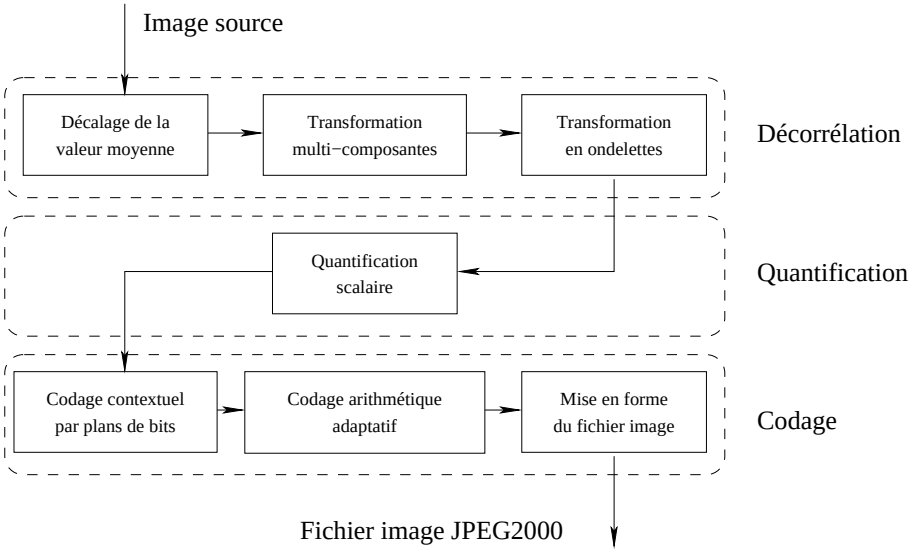
Cependant, les quantités de données croissantes manipulées par les applications d'imagerie numérique (§4.1) nécessitent de compresser à des taux plus élevés, auxquels le standard *JPEG* trouve ses limites : une image au format *JPEG* fortement compressée se caractérise par la présence d'artefacts aux frontières des blocs de 8×8 pixels sur lesquels s'effectuent la *DCT* et la quantification [93, 103].

L'élaboration du nouveau standard *JPEG2000* a été orientée d'une part vers une amélioration de la qualité des images à taux de compression égal, et d'autre part vers l'introduction de nouvelles fonctionnalités – telles que la prise en compte de régions d'intérêt – et d'une organisation des données plus flexible [95]. En toile de fond, une des préoccupations majeures du comité *JPEG2000* a été de prendre en compte dans un même standard les exigences d'une variété de domaines d'application. Ces exigences ont été regroupées dans un ensemble de *profils* qui ont servi de support à la sélection des techniques de codage. Les profils identifiés concernent les domaines suivants : Internet, la photographie numérique, l'imagerie médicale, la télédétection optique, la télécopie, les imprimantes, les scanners, les télécommunications mobiles, l'imagerie *SAR* [95].

Afin de pallier aux limitations de *JPEG* en termes de rapport taux de compression/distorsion, un décorrélateur à base de transformation en ondelettes discrète a été retenu (figure 4.4) [84, 96]. A la différence de la *DCT* qui projette une image ou un bloc de pixels sur une unique base de fonctions cosinus, la transformation en ondelettes peut être effectuée en utilisant un variété de bases d'ondelettes [86]. Le standard *JPEG2000* supporte deux bases d'ondelettes que nous décrirons plus loin : l'une est recommandée en compression réversible tandis que l'autre fournit un meilleur rapport taux de compression/distorsion en compression à pertes [84, 96].

Du fait de la variété des profils d'utilisation, des fonctionnalités proposées et des options supportées, le standard *JPEG2000* ne fournit pas un schéma de codage unique,

FIG. 4.4 – Structure d'un codeur JPEG2000



mais peut au contraire être dérivé en une variété de codeurs différents, chacun adapté à une classe donnée d'applications [96]. Ce degré élevé de flexibilité est un défi à la conception d'architectures matérielles réutilisables capables de répondre à tous les jeux de contraintes et de paramètres. Nous nous intéresserons plus particulièrement par la suite à l'implantation matérielle de la transformation en ondelettes discrète. Nous présenterons tout d'abord un état de l'art des algorithmes et architectures de transformation mono- et bidimensionnelles, puis nous détaillerons les étapes de la conception et de la synthèse d'un composant virtuel comportemental pour la *DWT 2D* conforme au standard *JPEG2000*.

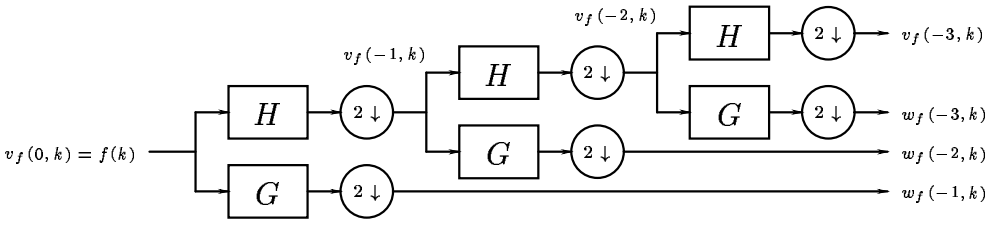
4.3 Implantation de la transformation en ondelettes discrète

4.3.1 Algorithmes de transformation en ondelettes

Comme nous le présentons en annexe D, la transformation en ondelettes discrète monodimensionnelle transforme une fonction f à variable réelle en une fonction d'approximation v_f et une fonction de détail w_f à variables discrètes. Nous désignerons respectivement par $v_f(j,k)$ et $w_f(j,k)$ le cœfficient d'approximation et le cœfficient de détail de la fonction f au niveau de résolution j et à la localisation k.

Cette transformation n'est pas sans rappeler le principe de la décomposition en série de FOURIER d'une fonction périodique : ici, n'importe quelle fonction de carré sommable peut être décomposée en série d'ondelettes [92]. En appliquant le principe de décomposition récursive présenté en figures 4.2 et 4.3, il suffit pour calculer la transformée en ondelette d'une fonction f de connaître son approximation v_f à une résolution j_0 aussi fine que souhaitée, et d'exprimer la relation de récurrence entre les cœfficients des sous-bandes d'approximation et de détail de niveau j et les cœfficients

FIG. 4.5 – Banc de filtres 1D pour la transformation en ondelettes sur trois niveaux



d'approximation de niveau $j+1$ [100, 103].

Dans le domaine du traitement des signaux numériques, il suffit de considérer que l'approximation au niveau de résolution $j_0 = 0$ d'un signal échantillonné $f(k)$ est égale à ce signal :

$$v_f(0,k) = f(k) \tag{4.1}$$

La relation entre sous-bandes de résolutions successives a été mise en évidence par Stéphane MALLAT [100] et a donné lieu à la méthode actuellement la plus employée pour calculer la transformation en ondelettes de signaux numériques : la méthode des bancs de filtres [100, 103].

Transformation en ondelettes et filtrage

La méthode des bancs de filtres repose sur un algorithme récursif dans lequel les cœfficients d'approximation $v_f(j,k)$ à un niveau j sont fournis simultanément à une paire de filtres numériques passe-bas (H) et passe-haut (G) produisant respectivement les cœfficients d'approximation et de détails au niveau $j-1$ (figure 4.5) [100, 103].

$$\begin{aligned} v_f(j-1,k) &= \sum_{n\in\mathbb{Z}} h(n) v_f(j,2k-n) \\ w_f(j-1,k) &= \sum_{n\in\mathbb{Z}} g(n) v_f(j,2k-n) \end{aligned} \tag{4.2}$$

L'utilisation d'une telle paire de filtres est conforme à la notion de décomposition en sous-bandes présentée brièvement au paragraphe 4.2.2 [100, 103]. Une opération de sous-échantillonnage avec un facteur 2 ($2\downarrow$) est effectuée en sortie de chaque filtre [100, 103], ce qui correspond à la décroissance de la résolution entre niveaux successifs (figure 4.5).

Dans la pratique, la décomposition à l'aide des bancs de filtres s'effectue sur un nombre fini N de niveaux – de l'ordre de 3 dans le domaine de la compression d'images – le jeu des sous-bandes de détails s'accompagne d'une sous-bande d'approximation résiduelle de basse résolution [86, 103]. Dans le cas bidimensionnel, à chaque niveau le même banc de filtres est appliqué successivement suivant les lignes et les colonnes d'une image [100, 103] (figure 4.6). Après le $N^{\text{ième}}$ étages, seule la dernière sous-bande d'approximation LL_N et les $3N$ sous-bandes de détails (LH_n, HL_n, HH_n pour $1 \leq n \leq N$) sont conservées et suffisent à reconstruire l'image d'origine [100, 103].

La transformation en ondelettes inverse s'effectue à l'aide d'une structure (H^*, G^*) duale de la précédente [100, 103]. Afin de garantir la reconstruction parfaite du

FIG. 4.6 – Banc de filtres 2D pour la transformation en ondelettes sur un niveau

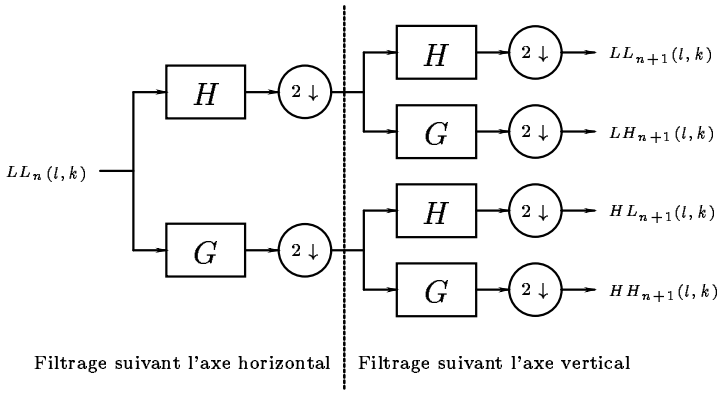
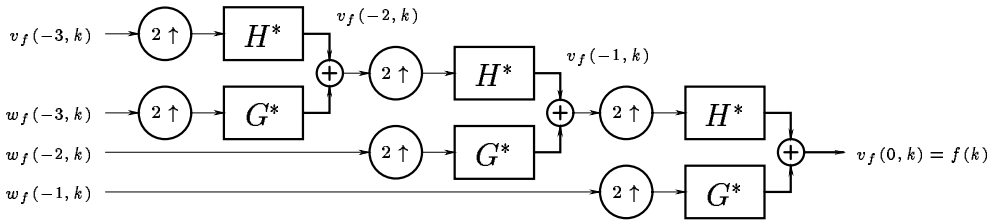


FIG. 4.7 – Banc de filtres 1D pour la transformation en ondelettes inverse sur trois niveaux



signal d'origine, les filtres de décomposition et de reconstruction doivent vérifier un certain nombre de conditions. Leur structure particulière doit notamment annuler le repliement de spectre pouvant résulter du sous-échantillonnage [100].

Les deux principaux types de bancs de filtres utilisés sont d'une part les *filtres miroirs en quadrature* (*QMF*) – où les filtres de reconstruction possèdent les mêmes cœfficients que les filtres de décomposition [100, 103], mais disposés dans l'ordre inverse – et d'autre part les bancs de filtres *biorthogonaux* [89, 103] pour lesquels les filtres de reconstructions ont un nombre et des valeurs de cœfficients différents des filtres de décomposition.

La norme *JPEG2000* [96] recommande l'utilisation des bancs de filtres biorthogonaux 9/7 et 5/3 de COHEN, DAUBECHIES et FEAUVEAU [89]. Les filtres de décomposition sont symétriques. Pour le banc 9/7, la réponse impulsionnelle du filtre passe-bas possède 9 cœfficients et celle du filtre passe-haut 7 cœfficients. Pour le banc 5/3, les filtres passe-bas et passe-haut possèdent respectivement 5 et 3 cœfficients. Les valeurs des cœfficients des filtres de décomposition et de reconstruction sont indiquées dans le tableau 4.1.

Le Lifting Scheme

L'implantation la plus courante des bancs de filtres repose sur un calcul de convolution entre le signal à transformer et la réponse impulsionnelle des filtres [100, 103].

TAB. 4.1 – Cœfficients des bancs de filtres recommandés par la norme JPEG2000

Filtres	Cœfficients				
	-4 4	-3 3	-2 2	-1 1	0
$H(9/7)$	0,0378	-0,0238	-0,1106	0,3774	0,8527
$G(9/7)$		0,0645	-0,0407	-0,4181	0,7885
$H^*(9/7)$		-0,03225	-0,02035	0,20905	0,39425
$G^*(9/7)$	0,0189	0,0119	-0,0553	-0,1887	0,42635
$H(5/3)$			-0,125	0,25	0,75
$G(5/3)$				-0,5	1
$H^*(5/3)$				0,5	1
$G^*(5/3)$			-0,125	-0,25	0,75

Un nouveau formalisme mis au point par Ingrid DAUBECHIES et Wim SWELDENS [90] consiste à représenter un banc de filtres *1D* sous la forme d'une structure en échelle (figure 4.8). A chaque niveau de décomposition, une transformation en ondelette triviale est tout d'abord appliquée, qui consiste à séparer les échantillons d'indice pair des échantillons d'indice impair. Ensuite, la structure en échelle fait alterner

- des pas de *dual lifting* – ou *prédiction* – au cours desquels les échantillons d'indice pair sont remplacés par la différence entre leur valeur et une valeur estimée calculée au moyen d'un filtre $P_i(z)$ en fonction des échantillons d'indices impairs du voisinage ;
- des pas de *lifting* au cours desquels les nouveaux échantillons pairs sont utilisés pour mettre à jour les échantillons impairs au moyen de filtres $U_i(z)$ (*Update*).

Le *dual lifting* a un effet décorrélateur qui s'apparente fortement aux techniques de codage prédictif (*DPCM*) [90, 93]. Le *lifting* a pour rôle de corriger l'effet de repliement de spectre introduit par le sous-échantillonnage [90]. Un pas de *scaling*, en bout de chaîne, applique un facteur correctif (K, $1/K$) à chacune des deux voies. Les valeurs corrigées correspondent aux cœfficients d'approximation et de détail au niveau courant.

Cette méthode connue sous le nom de *Lifting Scheme* [90] présente de sérieux avantages sur les algorithmes de convolution avec des filtres *FIR*. D'une part il présente une plus faible complexité calculatoire, et d'autre part il permet d'effectuer l'ensemble des calculs sur place : c'est-à-dire que les échantillons obtenus en sortie de chaque pas de *dual lifting*, *lifting* ou *scaling* viennent remplacer en mémoire les échantillons d'indices pairs ou impairs correspondant [90]. Ainsi la mémoire contenant les échantillons source suffit à elle seule à effectuer la totalité des étapes de la transformation multi-niveaux. La figure 4.9 montre que la transformation inverse s'effectue à l'aide des mêmes filtres P_i et U_i que la transformation directe [90].

Les fondements mathématiques du lifting scheme garantissent que tout banc de filtres biorthogonal possède une variété de structures en échelle équivalentes [90]. La norme *JPEG2000* recommande une implantation *lifting* du banc de filtres 9/7 dans laquelle la structure en échelle comprend deux pas de dual lifting, deux pas de lifting

FIG. 4.8 – Banc de filtre 1D et sa structure lifting équivalente

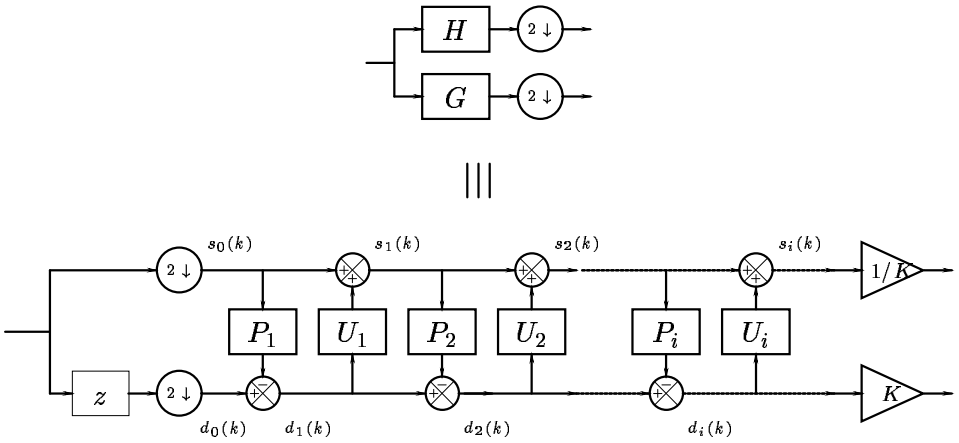
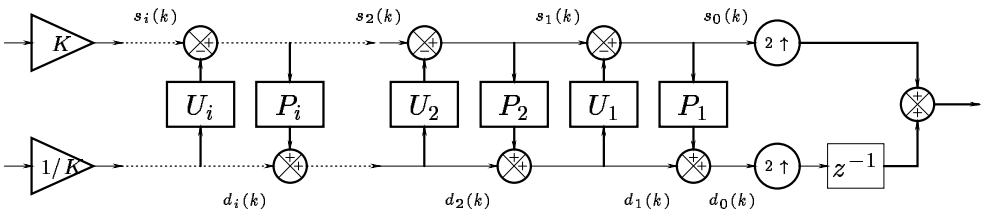


FIG. 4.9 – Structure lifting d'un banc de filtres 1D de reconstruction



et un pas de scaling [90, 96] :

$$\begin{array}{lrcl}
\text{Séparation pairs/impairs} & s_0(k) & = & v_f(j, 2k) \\
 & d_0(k) & = & v_f(j, 2k+1) \\
\text{Premier pas de dual lifting} & d_1(k) & = & d_0(k) + \alpha \times (s_0(k) + s_0(k+1)) \\
\text{Premier pas de lifting} & s_1(k) & = & s_0(k) + \beta \times (d_1(k-1) + d_1(k)) \\
\text{Deuxième pas de dual lifting} & d_2(k) & = & d_1(k) + \gamma \times (s_1(k) + s_1(k+1)) \\
\text{Deuxième pas de lifting} & s_2(k) & = & s_1(k) + \delta \times (d_2(k-1) + d_2(k)) \\
\text{Scaling} & v_f(j-1, k) & = & s_2(k)/K \\
 & w_f(j-1, k) & = & d_2(k) \times K
\end{array} \tag{4.3}$$

Les valeurs des cœfficients sont les suivantes [90, 96] :

$$\begin{aligned}
\alpha &= -1,586134342 \\
\beta &= -0,05298011854 \\
\gamma &= 0,8829110762 \\
\delta &= 0,4435068522 \\
K &= 1,149604398
\end{aligned}$$

La version *lifting* du banc de filtres 5/3 [90,96] comprend un pas de dual lifting, un pas de lifting et ne nécessite pas de facteur correctif sur les voies passe-bas et passe-haut (équations 4.4). Dans *JPEG2000*, ce banc de filtre réalise une transformation en ondelettes dite *entière* en tronquant la sortie de chaque filtre de prédiction P_i et de mise à jour U_i de telle sorte que les cœfficients d'approximation et de détail soient entiers [90, 96]. Cette propriété est très appréciée dans le domaine de la compression réversible car elle compense l'absence de quantification après décorrélation. Si ce banc de filtres présente des performances en décorrélation nettement inférieures au banc 9/7, son intérêt réside cependant dans sa faible complexité. Les divisions par 2 et par 4 peuvent en effet être efficacement matérialisées par des décalages de bits, moins coûteux que des multiplications ou des divisions [91].

$$\begin{aligned}
w_f(j-1, k) &= v_f(j, 2k+1) - \left\lfloor \frac{1}{2}(v_f(j, 2k) + v_f(j, 2k+2)) \right\rfloor \\
v_f(j-1, k) &= v_f(j, 2k) + \left\lfloor \frac{1}{4}(v_f(j, 2k-1) + v_f(j, 2k+1) + \frac{1}{2}) \right\rfloor
\end{aligned} \tag{4.4}$$

Le tableau 4.2 compare la complexité calculatoire des algorithmes de transformation par convolution d'une part et en utilisant le *Lifting Scheme* d'autre part. La mesure de complexité est donnée en nombre d'opérations arithmétiques nécessaires au calcul d'une paire de cœfficients passe-bas/passe-haut à un niveau de résolution donné. La dernière colonne du tableau indique le nombre moyen de données – échantillons source ou résultats intermédiaires de filtrages précédents – utilisées lors de ce calcul. Cette valeur est étroitement liée à la quantité de mémoire requise pour l'implantation du banc de filtres.

On observe que l'utilisation du *Lifting Scheme* permet de réduire de moitié le nombre d'opérations arithmétiques. Ceci est dû au fait que le *Lifting Scheme* réutilise de manière intensive des résultats intermédiaires de calculs produits par les passes de filtrage au voisinage du point considéré. Le *Lifting Scheme* réduit de 20 à 33% la

TAB. 4.2 – Complexité des bancs de filtres recommandés par la norme JPEG2000 pour différents algorithmes de filtrage

Banc de filtres	Algorithme	Additions / soustractions	Multiplications / décalages	Données
9/7	Convolution	14	16	9
	Lifting scheme	8	6	6
5/3	Convolution	6	7	5
	Lifting scheme	5	2	4

quantité moyenne de données nécessaires au calcul d'une paire d'échantillons transformés. Le choix du *Lifting Scheme* pour l'implantation matérielle de la transformation en ondelettes est ainsi doublement pertinent, tant du point de vue de la complexité calculatoire que du point de vue de la quantité de mémoire *a priori* nécessaire [90].

Filtrage et traitement au voisinage des bords

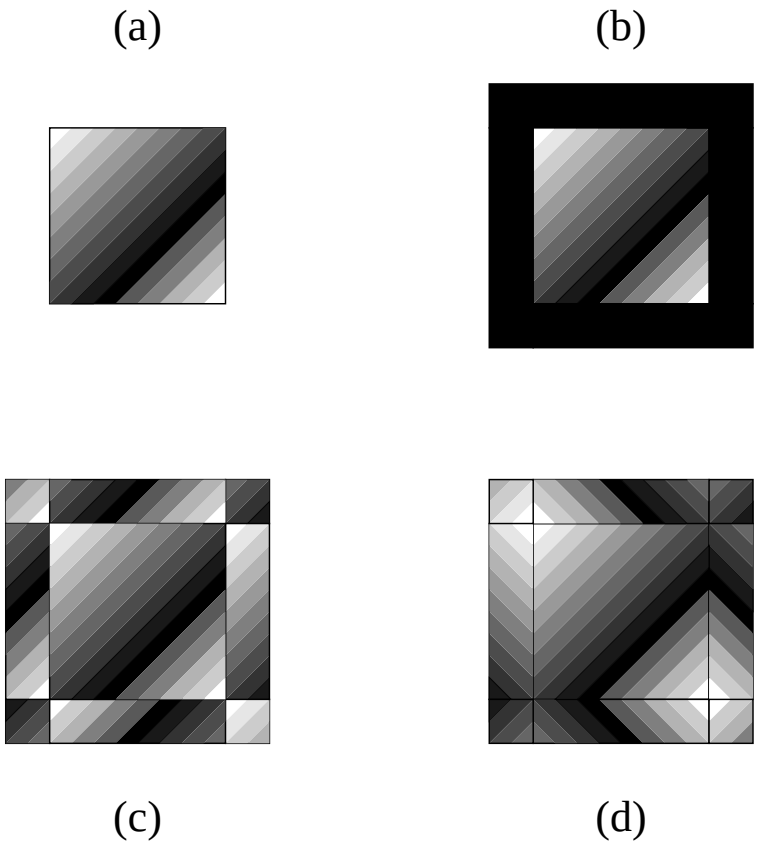
Dans la majorité des applications, les images numériques sont contenues dans une région rectangulaire de l'espace dont les dimensions sont bien définies (figure 4.10a). Au voisinage des frontières de l'image, l'algorithme de calcul de la convolution avec un filtre à réponse impulsionnelle finie mord en partie sur l'extérieur de l'image. Afin de mener à bien ce calcul, il convient par conséquent d'étendre l'image en remplissant son voisinage extérieur avec des pixels fictifs de valeurs bien choisies [103].

Plusieurs méthodes sont couramment envisagées : la première, représentée en figure 4.10b, consiste à étendre l'image avec des valeurs nulles (*zero-padding*) [103]. Cette méthodes présente deux inconvénients : d'une part, elle peut introduire des discontinuités de l'intensité des pixels – donc des détails de haute résolution – qui peuvent être préjudiciables aux performances d'une chaîne de compression. D'autre part, l'extension de l'image source se traduit par une image transformée élargie, donc présentant un plus grand nombre de cœfficients à coder que l'image source ne contenait de pixels.

Pour pallier au second inconvénient du *zero-padding*, une solution consiste à périodiser l'image avant de la filtrer [103] (figure 4.10c) : l'image décorrélée présentera alors la même périodicité et il suffira de n'en conserver qu'une période – *i.e.* un nombre de cœfficients égal au nombre de pixels sources – pour pouvoir reconstruire fidèlement l'image d'origine.

Une telle périodisation pouvant cependant introduire des discontinuités dans l'image source, on lui préfère généralement une extension par symétrie périodique (figure 4.10d), qui garantit la périodicité de l'image transformée tout en préservant la texture locale de l'image source au voisinage des bords [103]. C'est ce schéma d'extension qui a été sélectionné pour la chaîne de compression *JPEG2000* [96].

FIG. 4.10 – Différentes modes d'extension d'une image pour la transformation en ondelettes discrète 2D : (a) image non étendue ; (b) extension par zero-padding ; (c) extension périodique ; (d) extension par symétrie périodique



Ordonnancement des calculs pour la transformation en ondelettes multi-niveaux monodimensionnelle

Les algorithmes de filtrage que nous avons présentés sont destinés à s'insérer dans un schéma de décomposition multi-niveaux, mono- ou bidimensionnel. Alors que le filtrage est habituellement considéré comme un processus traitant séquentiellement les échantillons d'entrée, le caractère multirésolution de la transformation en ondelettes introduit une dimension supplémentaire dans le traitement et nécessite de définir dans quel ordre se succèdent les passes de filtrage suivant les différents niveaux. Deux algorithmes sont couramment utilisés : l'*algorithme en pyramide* (*PA*) et l'*algorithme en pyramide récursif* (*RPA*) [88, 91].

L'algorithme en pyramide consiste à effectuer l'intégralité des passes de filtrage à un niveau de résolution avant de passer au niveau suivant [88, 91]. Le listing 4.1 décrit la transformation sur N niveaux d'un vecteur f de largeur L. On suppose que le calcul d'une paire de cœfficients d'approximation et de détail nécessite au plus $2M+1$ échantillons de la sous-bande d'approximation du niveau précédent. M vaudra 4 dans le cas du banc de filtres 9/7 et 2 dans le cas du banc de filtres 5/3. L'ordre des passes de filtrage est illustré par la figure 4.11 dans le cas d'une transformation sur quatre niveaux. L'inconvénient de cette méthode est la durée de vie importante des cœfficients d'approximation intermédiaires [88]. L'algorithme en pyramide nécessite en effet le stockage de la totalité de la sous-bande d'approximation à un niveau donné en attendant que le calcul au niveau suivant commence.

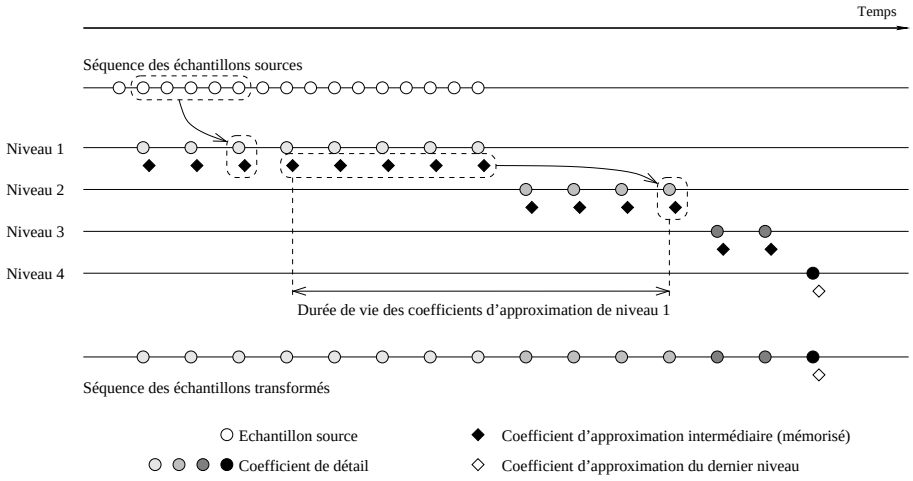
Listing 4.1 – L'algorithme en pyramide pour la transformation en ondelettes discrète monodimensionnelle

```
v(0,0..L-1) ← f
pour n de 1 à N faire  ' pour chaque étage de décomposition
  pour k de 0 à L/2^n - 1 faire  ' pour chaque échantillon
    v(n,k) ← approximation(v(n-1,2×k-M..2×k+M))
    w(n,k) ← détail(v(n-1,2×k-M..2×k+M))
  fin pour k
fin pour n
```

L'algorithme en pyramide récursif [88, 91] entrelace les passes de filtrage sur les différents niveaux (figure 4.12). Dès qu'une quantité suffisante de données à un niveau n a été calculée, une passe de filtrage au niveau suivant est effectuée. Ce procédé permet de minimiser la durée de vie des résultats intermédiaires de calcul, et par conséquent d'optimiser l'utilisation des ressources mémoire.

Le tableau 4.3 compare la quantité de mémoire nécessaire à chacun des deux algorithmes de transformation monodimensionnelle présentés. Dans le cas de l'algorithme en pyramide, cette quantité est égale à $L/2$ où L représente le nombre d'échantillons du vecteur source : les passes de filtrage sur les niveaux 2 à N peuvent en effet réutiliser les $L/2$ cellules mémoires nécessaires à la transformation au premier niveau. Dans le cas de l'algorithme récursif, la taille mémoire requise est multiple du nombre N de niveaux de décomposition et du nombre C de données nécessaires à l'obtention d'une paire de cœfficients passe-haut/passe-bas. La valeur de C dépend de la longueur des filtres et de l'algorithme de filtrage choisi (convolution ou *Lifting Scheme*). Un ordre

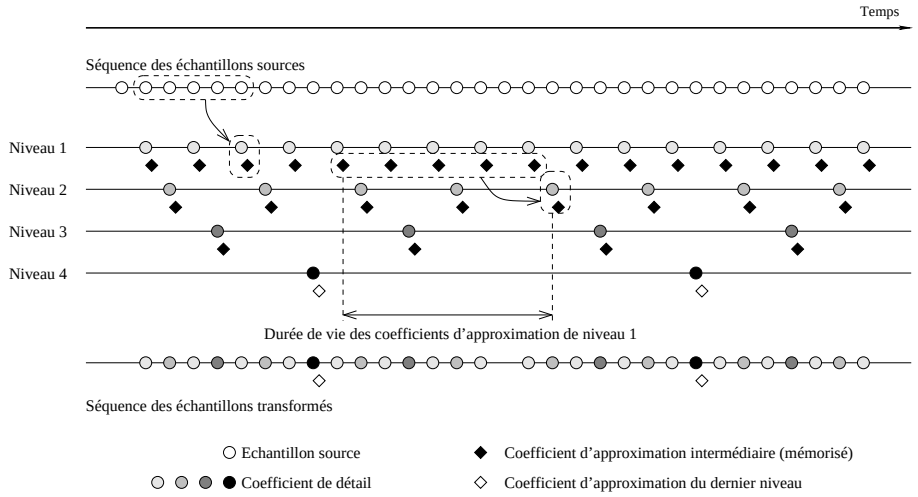
FIG. 4.11 – Ordre des passes de filtrage pour l'algorithme en pyramide



Listing 4.2 – L'algorithme en pyramide récursif pour la transformation en ondelettes discrète monodimensionnelle

```
v(0,0..L-1) ← f
pour k de 0 à L-1 faire
  RPA(1,k)
fin pour k

procédure RPA(n,k)
  si k est pair alors
    v(n,k/2) ← approximation(v(n-1,k-M..k+M))
    w(n,k/2) ← détail(v(n-1,k-M..k+M))
  sinon
    RPA(n+1,(k-1)/2-M)
  fin si
fin RPA
```

FIG. 4.12 – Ordre des passes de filtrage pour l'algorithme en pyramide récursif



TAB. 4.3 – Quantité de mémoire requise par les algorithmes de décomposition monodimensionnelle multi-niveaux

Algorithme	Taille mémoire au niveau n	Taille mémoire pour N niveaux
PA	$L/2^n$	$L/2$
RPA	C	$C \times N$

de grandeur de C est fourni dans la dernière colonne du tableau 4.2 pour les bancs 9/7 et 5/3.

Dans la plupart des cas, $C \times N$ est nettement inférieur à $L/2$. Par exemple, la transformation d'un vecteur de $L = 256$ échantillons sur $N = 3$ niveaux en utilisant la version *lifting* du banc de filtres 9/7 nécessite 128 cellules mémoire avec le *PA*, et seulement 18 cellules avec le *RPA*. Si le *RPA* minimise les ressources mémoire pour une complexité calculatoire équivalente, il se traduit cependant par une augmentation de la complexité du contrôle dû à l'entrelacement des passes de filtrage sur les différents niveaux.

Ordonnancement des calculs pour la transformation en ondelettes multi-niveaux bidimensionnelle

Les algorithmes en pyramide, non récursif et récursif, présentés dans le cas monodimensionnel ont leurs équivalents en deux dimensions. On désigne respectivement par L et H la largeur et la hauteur de l'image à transformer, en pixels.

L'algorithme en pyramide *2D* non récursif [87] est présenté sous forme de pseudo-code dans le listing 4.3. Les niveaux de décomposition sont effectués l'un après l'autre. La transformation verticale à un niveau donné débute dès que la totalité des passes de

filtrage horizontal ont été effectuées. La quantité de mémoire nécessaire au stockage des résultats intermédiaires de calcul est du même ordre de grandeur que le nombre total de pixels dans l'image. Afin de minimiser les accès à la mémoire, les pixels sont lus de préférence dans un ordre compatible avec la direction du filtrage – chaque banc de filtre étant supposé posséder sa propre file de registres : de gauche à droite sur chaque ligne pour la transformation horizontale, de haut en bas sur chaque colonne pour la transformation verticale.

Listing 4.3 – L'algorithme en pyramide pour la transformation en ondelettes discrète bidimensionnelle

```
v(0,0..H-1,0..L-1) ← f
pour n de 1 à N faire ' pour chaque étage de décomposition
  ' Transformation suivant les lignes au niveau n
  pour l de 0 à H/2^(n-1) - 1 faire ' pour chaque ligne
    pour k de 0 à L/2^n - 1 faire ' pour chaque pixel de la ligne l
      v(n,l,k) ← approximation(LL(n-1,l,2×k-M..2×k+M))
      w(n,l,k) ← détail(LL(n-1,l,2×k-M..2×k+M))
    fin pour k
  fin pour l

  ' Transformation suivant les colonnes au niveau n
  pour k de 0 à L/2^n - 1 faire ' pour chaque colonne
    pour l de 0 à H/2^n - 1 faire ' pour chaque pixel de la colonne k
      LL(n,l,k) ← approximation(v(n,2×l-M..2×l+M,k))
      LH(n,l,k) ← détail(v(n,2×l-M..2×l+M,k))
      HL(n,l,k) ← approximation(w(n,2×l-M..2×l+M,k))
      HH(n,l,k) ← détail(w(n,2×l-M..2×l+M,k))
    fin pour k
  fin pour l
fin pour n
```

L'algorithme récursif [87] calcule respectivement les passes de filtrage horizontal et vertical au niveau n sur les colonnes et lignes dont l'indice est multiple de 2^n. Dans un souci de clarté, le listing 4.4 repose sur un parcours de type *raster* de l'image source, c'est-à-dire que les lignes de pixels sont traitées l'une après l'autre de haut en bas, les pixels d'une ligne donnée étant parcourus de gauche à droite. Il a été cependant montré dans [97] que ce parcours n'est pas le plus adapté au traitement récursif de la transformation. Le parcours proposé dans [97] repose sur une courbe en Z fractal appelée courbe de MORTON qui tient compte du caractère multirésolution de la transformation. Cette courbe est représentée en figure 4.13. L'utilisation de ce parcours permet de minimiser la durée de vie des données intermédiaires et de lisser le débit d'accès à la mémoire.

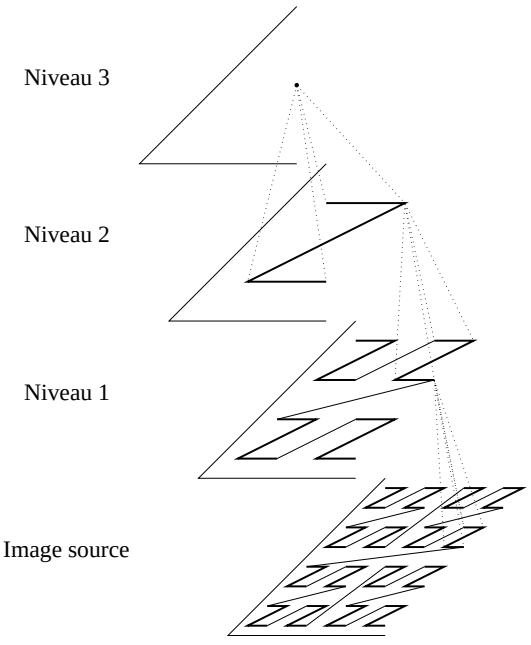
Le tableau 4.4 donne la quantité de mémoire pour le stockage des résultats intermédiaires de calcul à chaque niveau de décomposition ($1 \leq n \leq N$). La quantité totale de mémoire pour une transformation sur N niveaux est bornée en valeur supérieure. Elle varie selon une loi quadratique (*i.e.* multiple de $L \times H$) pour l'algorithme en pyramide, et selon une loi linéaire (*i.e.* multiple de $L+H$) pour l'algorithme en pyramide récursif.

Puisque la transformation en ondelettes bidimensionnelle est fondée sur l'applica-

Listing 4.4 – L'algorithme en pyramide récursif pour la transformation en ondelettes discrète bidimensionnelle

```
v(0,0..H-1,0..L-1) ← f
pour l de 0 à H-1 faire
  pour k de 0 à L-1 faire
    RPA(1,l,k)
  fin pour k
fin pour l

procédure RPA(n,l,k)
  si k est pair alors
    v(n,l,k/2) ← approximation(LL(n-1,l,k-M..k+M))
    w(n,l,k/2) ← détail(LL(n-1,l,k-M..k+M))

    si l est pair alors
      LL(n,l/2,k/2) ← approximation(v(n,l-M..l+M,k/2))
      LH(n,l/2,k/2) ← détail(v(n,l-M..l+M,k/2))
    fin si
  sinon
    si l est pair alors
      HL(n,l/2,(k-1)/2) ← approximation(w(n,l-M..l+M,(k-1)/2))
      HH(n,l/2,(k-1)/2) ← détail(w(n,l-M..l+M,(k-1)/2))
    sinon
      RPA(n+1,(l-1)/2-M,(k-1)/2-M)
    fin si
  fin si
fin RPA
```

FIG. 4.13 – Parcours multirésolution d'une image à l'aide de la courbe fractale de MORTON



TAB. 4.4 – Quantité de mémoire requise par les algorithmes de décomposition bidimensionnelle multi-niveaux

Algorithme	Taille mémoire au niveau n	Taille mémoire pour N niveaux
PA	$\frac{L \times H}{4^n}$	$\frac{L \times H}{4}$
RPA	$\frac{L+H}{2^n} \times C$	$< (L+H) \times C$

tion de transformations monodimensionnelles suivant les deux directions spatiales, la complexité calculatoire des algorithmes de transformation *2D* se déduit directement de celle des algorithmes *1D*. Du fait du sous-échantillonnage suivant les deux directions spatiales, la complexité des calculs relatifs à chaque niveau de décomposition est divisée par 4 entre un niveau et le suivant.

4.3.2 Architectures VLSI de transformation en ondelettes

Dans cette section, nous présentons différents modèles d'architecture rencontrés dans la littérature pour l'implantation de la transformation en ondelettes à l'aide de bancs de filtre biorthogonaux. Dans un souci de clarté, nous présentons tout d'abord un état de l'art des architectures de bancs de filtres monodimensionnels. Nous indiquons comment ces bancs de filtres peuvent être insérés dans une architecture de transformation multi-niveaux *1D*. Dans un troisième temps nous étendons au cas *2D* les observations faites pour la transformation *1D* et nous donnons quelques exemples d'architectures réalisées suivant ces modèles.

Modèles d'implantation d'un banc de filtres monodimensionnel

Nous avons vu qu'un banc de filtres peut être implanté en utilisant soit l'algorithme classique de convolution avec la réponse impulsionnelle de deux filtres *FIR*, soit le *lifting scheme*.

Le filtrage numérique par convolution peut s'effectuer avec différents modèles d'architectures exploitant plus ou moins les propriétés des filtres. Le modèle série (figure 4.14) repose sur la mise en cascade de cellules *MAC* (Multiplication/ACcumulation), tandis que le modèle parallèle 4.15 utilise un arbre équilibré d'additionneurs et de multiplieurs [88]. Les propriétés des filtres, tels que la symétrie ou l'antisymétrie des cœfficients peuvent donner lieu à des simplifications arithmétiques par factorisation de certains termes (4.16). Le nombre de multiplieurs peut alors être réduit de moitié [101].

Ces structures de filtres ne sont cependant pas spécifiques de la transformation en ondelettes : la plupart des applications utilisant des filtres symétriques peuvent mettre à profit ce type d'optimisations. En particulier, ces architectures n'exploitent pas le fait que seul un échantillon sur deux est conservé en sortie des filtres. Il en résulte donc que ces architectures passent la moitié du temps à effectuer des calculs inutiles. Une utilisation plus efficaces des opérateurs consiste à répartir aussi équitablement que possible la charge de calcul dans le temps, c'est-à-dire à effectuer la moitié des

FIG. 4.14 – Implantation série d'un filtre FIR

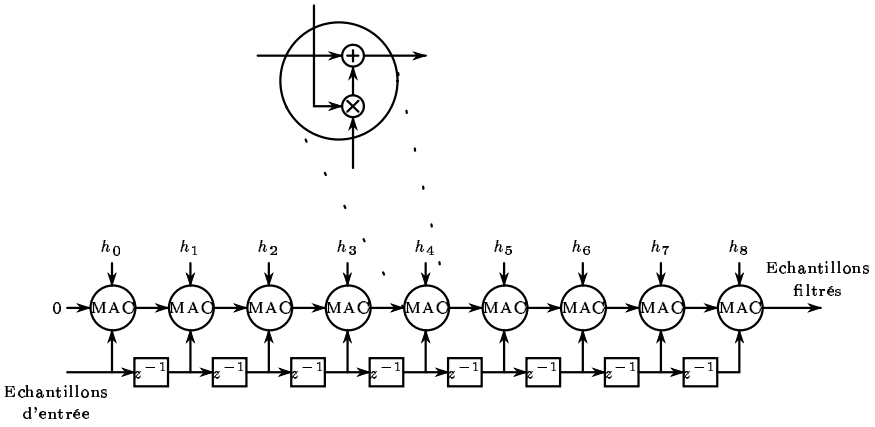


FIG. 4.15 – Implantation parallèle d'un filtre FIR

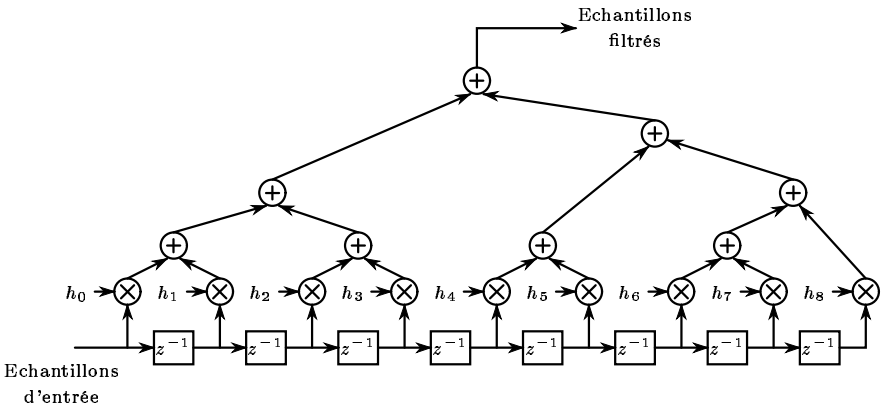
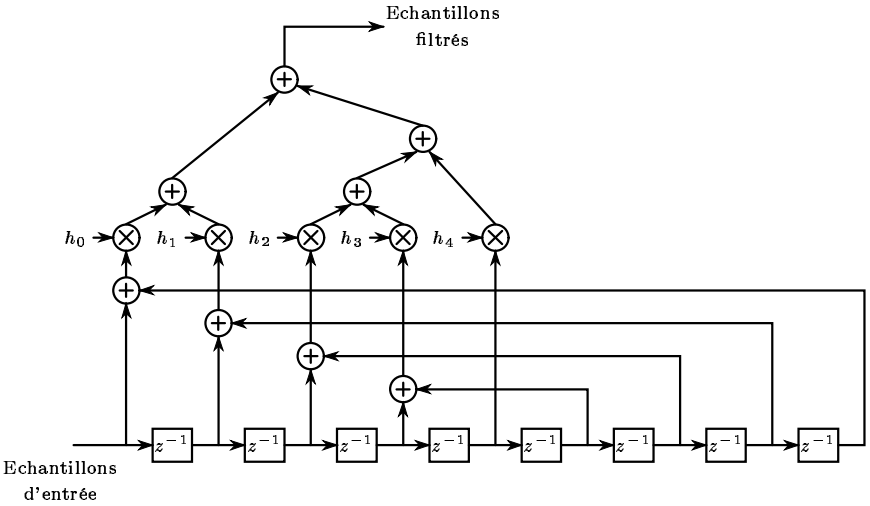


FIG. 4.16 – Implantation parallèle d'un filtre FIR symétrique



opérations sur les temps pairs et l'autre moitié sur les temps impairs, en réutilisant le même jeu d'opérateurs pour les deux phases. Une telle architecture est présentée en figure 4.17 [101]. Le formalisme mathématique sous-jacent est connu sous le nom de *décomposition polyphase* où les équations des filtres sont réécrites en séparant à l'entrée du banc de filtres les échantillons d'indices pairs et ceux d'indices impairs [90, 103].

Les architectures exploitant le *lifting scheme* peuvent bénéficier du même type d'optimisation. Une architecture pour le banc de filtres *lifting* 9/7 peut être proposée (figure 4.18) en rendant causales les équations 4.3 (page 119). Ces équations tiennent déjà compte de la symétrie des cœfficients. L'architecture proposée repose sur quatre opérateurs *lifting* constitués chacun de deux additionneurs et un multiplieur. A nouveau, les sorties de cette architectures ne doivent être prises en compte qu'un fois sur deux. La mise en œuvre du partage d'opérateurs entre temps pairs et impairs aboutit à une architecture ne comprenant que deux opérateurs lifting (figure 4.19). [85] présente une architecture de transformation bidimensionnelle utilisant ce principe pour différents bancs de filtres lifting parmi lesquels les bancs 9/7 et 5/3.

La démarche inverse consiste à considérer que les échantillons à transformer sont fournis deux par deux à chaque période d'échantillonnage : c'est alors le nombre de points de mémorisation qui est divisé par deux. Cette dernière approche a été présentée dans [91] pour le banc de filtres lifting 5/3. L'architecture correspondante est présentée en figure 4.20.

Le tableau 4.5 compare les ressources matérielles nécessaires à la décomposition en ondelettes monodimensionnelle sur un niveau. Les architectures lifting ont une complexité généralement inférieure aux architectures reposant sur la convolution. Pour les architectures parallèle avec décomposition polyphase, et lifting avec échantillons présentés deux par deux, le nombre d'opérateurs est – à un multiplieur près – le même, mais l'architecture lifting réduit de 60% la quantité de mémoire nécessaire. Si à l'échelle d'un banc de filtres cette différence n'est guère significative, elle prendra toute son importance lors de l'implantation de la transformation bidimensionnelle,

FIG. 4.17 – Implantation d'un filtre FIR avec décomposition polyphase

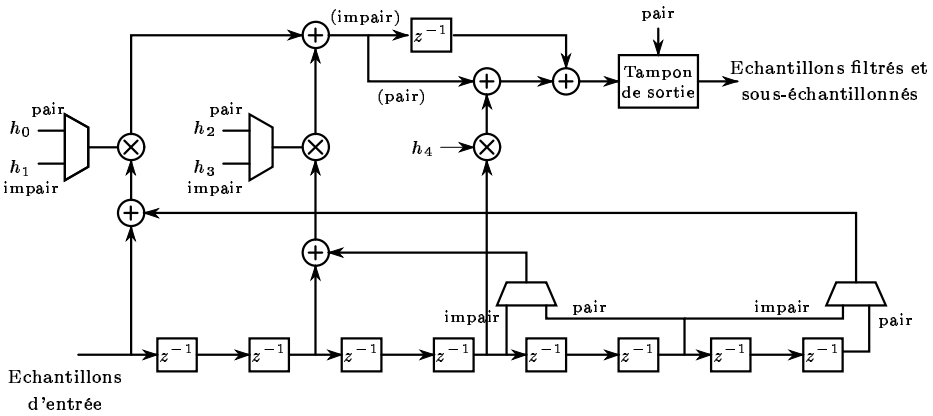


FIG. 4.18 – Implantation directe du banc de filtres lifting 9/7

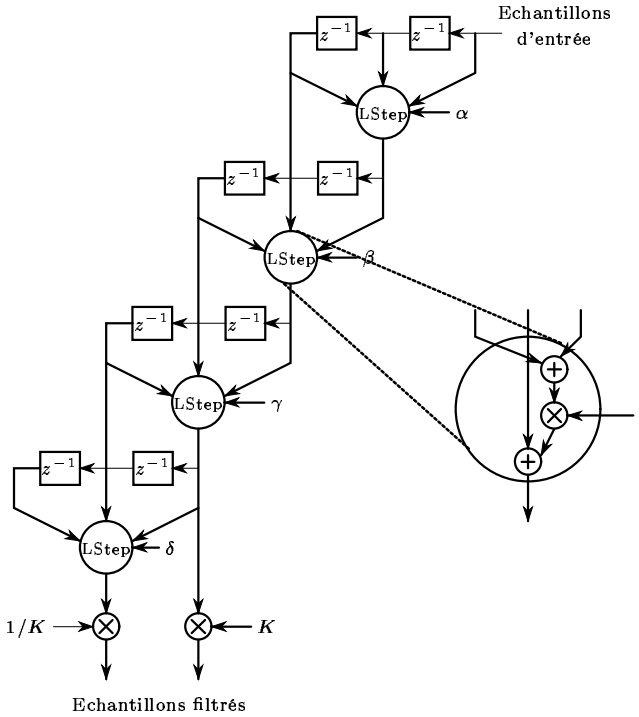


FIG. 4.19 – Implantation du banc de filtres lifting 9/7 avec partage d'opérateurs

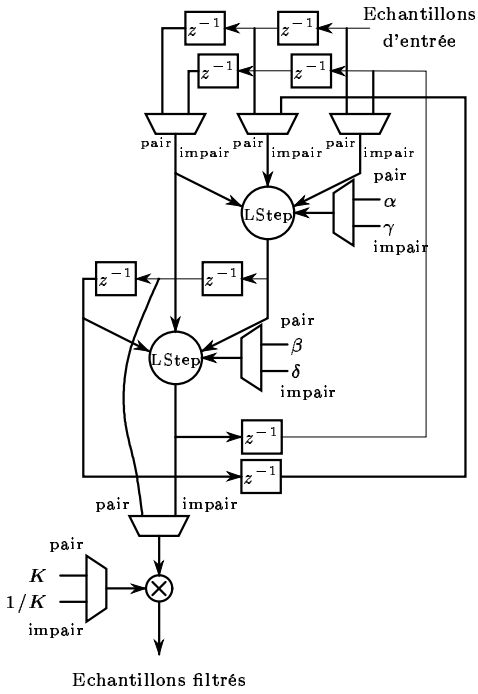
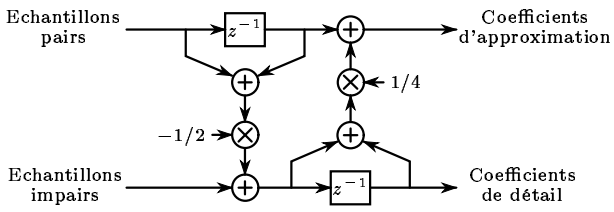


FIG. 4.20 – Implantation du banc de filtres lifting 5/3 avec lecture des échantillons deux par deux



TAB. 4.5 – Complexité architecturale de différentes implantations du banc de filtres 9/7

Modèle de filtre	Add	Mult	Mux	Regs
Série	16	16	0	8
Parallèle	14	16	0	8
Parallèle (symétrique)	14	9	0	8
Parallèle (polyphase)	8	5	10	10
Lifting	8	6	0	8
Lifting (2 par 2)	8	6	0	4
Lifting (polyphase)	4	3	7	8

TAB. 4.6 – Performances en vitesse de différentes implantations du banc de filtres 9/7

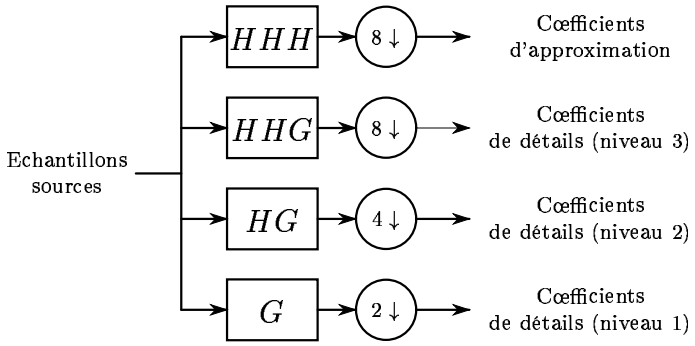
Modèle de filtre	Cadence (T_{emin})	Latence
Série	$T_{mult} + 9T_{add}$	$T_{mult} + 9T_{add}$
Parallèle	$T_{mult} + 4T_{add}$	$T_{mult} + 4T_{add}$
Parallèle (symétrique)	$T_{mult} + 4T_{add}$	$T_{mult} + 4T_{add}$
Parallèle (polyphase)	$T_{mult} + 4T_{add}$	$T_{mult} + 4T_{add}$
Lifting	$5T_{mult} + 8T_{add}$	$4T_{mult} + 8T_{add}$
Lifting (2 par 2)	$5T_{mult} + 8T_{add}$	$4T_{mult} + 8T_{add}$
Lifting (polyphase)	$3T_{mult} + 4T_{add}$	$T_e + 3T_{mult} + 4T_{add}$

où ce ne sont plus des échantillons individuels, mais des lignes entières de l'image en cours de transformation qu'il faudra mémoriser.

Le tableau 4.6 donne un aperçu des performances en vitesse pouvant être atteintes par ces filtres. La colonne cadence donne la période d'échantillonnage minimale T_{emin} en fonction des temps de traversée respectifs T_{mult} et T_{add} d'un additionneur et d'un multiplieur. T_{emin} est calculée comme le temps de traversée du chemin d'opérateurs arithmétiques le plus long parcouru par les données.

La colonne latence indique le temps séparant l'entrée d'un nouvel échantillon d'indice pair et la sortie du ou des échantillons transformés correspondants. A l'exception de l'architecture lifting (2 par 2), il faut ajouter T_e à cette valeur pour obtenir le temps séparant l'entrée d'un échantillon d'indice impair de la sortie des échantillons transformés correspondants. Si le *Lifting Scheme* autorise des implantations de faible complexité architecturale, elle possède cependant un degré de parallélisme exploitable plus faible que la convolution avec des filtres parallèles. Pour toutes ces architectures, la cadence de traitement peut être accélérée en recourant à des architectures pipeline. Le prix à payer est l'ajout d'éléments de mémorisation entre les étages de pipeline.

FIG. 4.21 – Architecture réduite pour la transformation 1D multi-niveaux



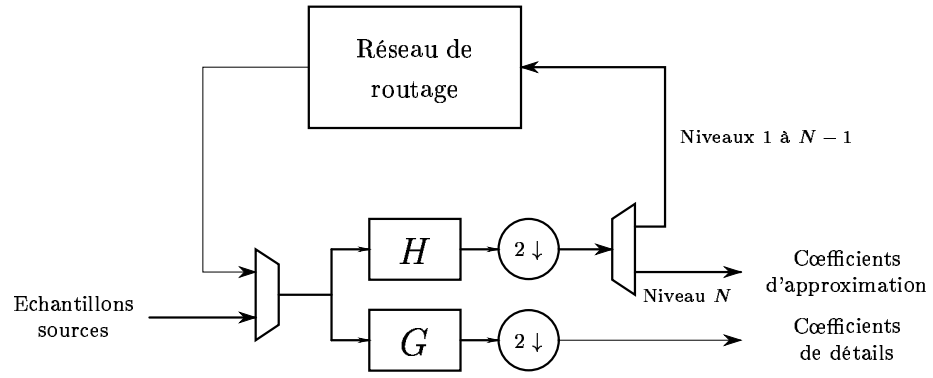
Modèles d'implantation de la transformation 1D multi-niveaux

On distingue généralement deux types d'architectures de transformation en ondelettes. Les architectures les plus proches de la vision intuitive que l'on a de la transformation à l'aide des bancs de filtres sont les architectures dites *mises à plat* – *unfolded* –, consistant à mettre en cascade autant d'étages de filtrage que de niveaux de décomposition à effectuer [88]. La figure 4.5 représente fidèlement la structure d'une telles architecture. Ce type d'architectures présente deux inconvénients liés d'une part à sa complexité architecturale, du fait de la réplication du matériel nécessaire au calcul, et d'autre part à l'inhomogénéité de la répartition de la charge de calcul au cours du temps, chaque étage travaillant à la moitié de la cadence de son prédécesseur.

Une alternative proposée dans [102] et représentée en figure 4.21 consiste à calculer pour chaque sous-bande le filtre équivalent à la cascade de filtres traversés. L'implantation des filtres obtenus se prête aux mêmes optimisations que celles présentées plus haut. Dans [102], chacun des filtres est implanté à l'aide d'une seule cellule MAC associée à un contrôleur et une ROM contenant les cœfficients des filtres.

Les architectures *rebouclées* – *folded* – effectuent au contraire tous les niveaux de décomposition à l'aide d'un unique banc de filtres [88]. Ce banc de filtres est couplé à un réseau de routage (figure 4.22) chargé d'aiguiller les données en fonction d'un ordonnancement pré-établi des passes de filtrage. Un tel réseau de routage doit prendre en charge deux types de tâches qui sont d'une part le contrôle du flot de données et d'autre part la mémorisation des données [88]. La complexité du réseau de routage dépend de l'algorithme d'ordonnancement appliqué. Au paragraphe précédent, nous en avons présenté deux : l'algorithme en pyramide et l'algorithme en pyramide récursif. Différents modèles d'architectures de routage sont décrits dans [88]. Ces modèles reposent soit sur une partie contrôle séparée de la partie mémorisation, les deux interagissant par l'intermédiaire de structures d'aiguillage (multiplexeurs si les données sont mémorisées dans des registres, décodeur d'adresse si les données sont stockées en RAM), soit sur une distribution du contrôle sur les cellules de mémorisation (réseaux semi-systoliques avec aiguillage par bus généraux, réseaux systoliques).

FIG. 4.22 – Architecture rebouclée pour la transformation 1D multi-niveaux



Extrapolation au cas bidimensionnel

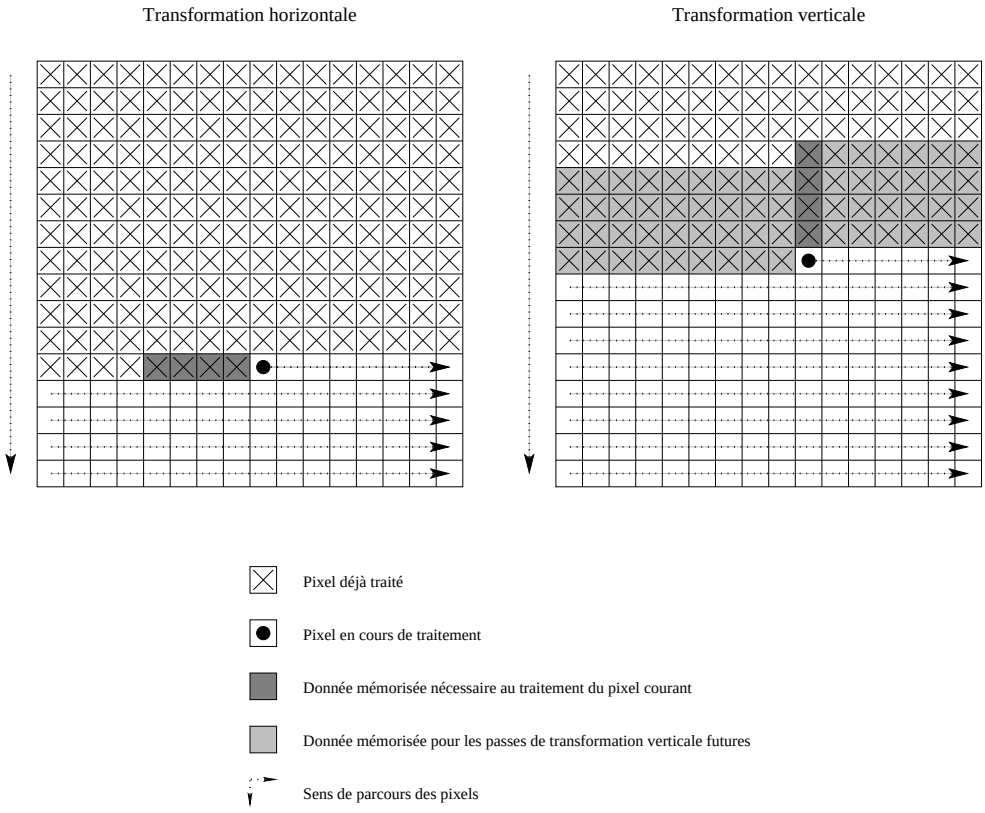
La transformation en ondelettes *2D* séparable permet d'utiliser la même architecture de banc de filtres pour chaque passe de décomposition horizontale ou verticale [100]. Si l'algorithme de transformation multi-niveaux est l'algorithme en pyramide, les architectures *1D* déjà présentées conviennent aux deux directions de transformation à condition d'adapter le sens de parcours des pixels en fonction de la direction du filtrage. Ces pixels sont lus et mis à jour dans une mémoire image globale, la séquence des accès étant gérée par un contrôleur.

L'algorithme en pyramide récursif enchaîne – une ligne sur deux et une colonne sur deux – une passe de filtrage horizontal et une passe de filtrage vertical. Les filtres verticaux traitant directement les données en sortie des filtres horizontaux au même niveau, l'ordre de parcours des pixels est le même pour les deux directions du filtrage. Si l'on se borne à un parcours de type *raster*, les structures de filtres *1D* que nous avons évoquées, avec une structure de mémorisation de type ligne à retard, sont bien adaptées au filtrage horizontal. En revanche, les filtres verticaux travaillent perpendiculairement à la direction du parcours (figure 4.23) et nécessitent une structure de mémorisation différente. La figure 4.23 montre que le temps séparant deux passes successives de filtrage suivant une colonne donnée de l'image correspond à la durée du parcours de deux lignes de pixels. Il s'ensuit que la taille mémoire nécessaire aux filtres verticaux est multiple de la largeur de l'image.

Dans [91], les filtres lifting horizontaux utilisent deux cellules mémoire (opérateurs z^{-1}). Les filtres verticaux sont conçus selon une architecture similaire où les éléments de mémorisation deviennent des lignes à retard (z^{-L} pour des images de largeur L). Les filtres verticaux travaillant une ligne sur deux, deux FIFOS supplémentaires sont insérées entre le banc de filtres horizontal et le banc de filtres vertical.

L'architecture lifting présentée dans [85] comprend un bloc mémoire entre les bancs de filtres horizontaux et verticaux. Ce bloc mémoire est partitionné en deux à quatre bancs mémoire selon le banc de filtres choisi. Il sert à stocker aussi bien les données issues du filtrage horizontal que des résultats partiels de filtrage vertical. Cette architecture est du type rebouclé (folded) et requiert un bloc mémoire supplémentaire

FIG. 4.23 – Comparaison des besoins en mémoire des transformations horizontale et verticale pour un parcours de type raster



en entrée du banc de filtres horizontal : ce bloc reçoit soit des pixels source, soit des cœfficients d'approximation en sortie des filtres verticaux au niveau précédent.

L'implantation de l'algorithme en pyramide récursif avec un parcours des pixels en Z fractal (voir figure 4.13 en page 126) est présentée dans [97]. L'architecture réalisée assure un débit régulier d'accès à la mémoire. La taille mémoire nécessaire est réduite de 75% comparée à l'implantation d'un parcours de type *raster*.

Flexibilité et "réutilisabilité" des architectures présentées

L'avènement de standards de codage utilisant la transformation en ondelettes discrète pour la compression de vidéo (*MPEG4*) et d'images fixes (*JPEG2000*) a conduit les concepteurs – chercheurs ou industriels – à concentrer leur attention sur les familles d'ondelettes proposées par ces standards. Parmi les articles déjà cités, [85], [91] et [101] se sont intéressés aux deux ondelettes sélectionnées par le comité *JPEG2000*.

La plupart des architectures que nous avons présentées supportent un degré restreint de flexibilité. L'architecture retenue pour le bloc *IP* présenté dans [101] repose sur une chaîne de cellules *MAC*. Une variété de filtres peuvent être produits à partir d'une telle structure : ce type d'architecture exploite une flexibilité structurelle qui consiste à instancier autant de cellules *MAC* que nécessaire et à leur associer les valeurs de cœfficients souhaitées pour implanter un filtre donné. La flexibilité structurelle peut également être employée pour la décomposition multi-niveaux, soit en recourant à des architectures non rebouclées dans lesquelles il suffit de cascader autant de bancs de filtres que souhaité, soit à l'aide d'architectures rebouclées utilisant un réseau de routage régulier de type systolique, où il suffit d'ajouter ou d'enlever des couches de cellules de mémorisation et de contrôle. La flexibilité structurelle est aisément obtenue à l'aide des flots de synthèse *RTL* classiques. Le langage *VHDL* permet par exemple de générer une architecture régulière paramétrable au moyen de clauses generic et de l'instruction generate.

Comme nous pouvons le constater, la flexibilité de ces architectures concerne uniquement des paramètres de niveau fonctionnel : nombre de niveaux de décomposition, réponse impulsionnelle des filtres d'ondelettes, taille des images. Pour un jeu donné de paramètres fonctionnels, nous obtenons pour chaque type d'implantation une architecture de complexité et de performances fixes, à l'exception de [97] où l'architecture est obtenue par synthèse de haut niveau en utilisant l'environnement *Cathedral III*.

D'autre part, toutes ces architectures reposent sur un ordre et une cadence fixe des entrées/sorties. L'absence de paramètres liés à la communication est un obstacle à la réutilisation : intégrer de telles architecture dans un système utilisant un ordre et une cadence d'entrées/sorties différents nécessite de concevoir des interfaces de communication qui peuvent être coûteuses en temps de développement et augmenter la complexité du système [66].

Dans le chapitre 5, nous montrons comment la méthodologie présentée en première partie de ce mémoire peut être appliquée afin de concevoir un composant virtuel de transformation en ondelettes hautement flexible. Le degré de flexibilité ciblé concerne tant la fonctionnalité du composant que ses performances – ajustées en générant automatiquement une architecture de chemin de données adaptée aux contraintes de l'utilisateur – et son comportement aux entrées/sorties.

Chapitre 5

Conception d'un Composant Virtuel Comportemental pour la Transformation en Ondelettes Discrète 2D

Dans ce chapitre, nous présentons les étapes de la conception et de la synthèse d'un composant virtuel comportemental réalisant la transformation en ondelettes bi-dimensionnelle multi-niveaux. Ce composant a été conçu à l'aide de la méthodologie présentée en première partie de ce mémoire.

Nous précisons en section 5.1 les aspects de la fonctionnalité du composant liés au standard *JPEG2000*. En section 5.2, nous détaillons la structure de l'algorithme, ses paramètres et ses propriétés exploitables dans le cadre d'une implantation *VLSI*. Nous nous intéressons ensuite à l'extraction d'un motif répétitif de calcul pour la transformation en ondelettes *au fil de l'eau* d'une image (section 5.3), avant de conclure cette étude par des résultats de synthèse de haut niveau obtenus à l'aide d'un outil du commerce (section 5.4).

5.1 Transformation en ondelettes pour JPEG2000

Comme nous l'avons vu dans le chapitre 4, deux ondelettes sont proposées par la norme *JPEG2000* : ce sont les ondelettes biorthogonales 9/7 et 5/3 [96]. La première n'est utilisée qu'en compression avec pertes. L'utilisation de la seconde est recommandée en compression réversible : dans ce contexte, elle est appliquée sous la forme d'une transformation en ondelettes entière [90].

Propriétés – Les propriétés au niveau fonctionnel sont liées à la topologie des données d'entrée/sortie : les données d'entrée sont des images, donc des signaux bidimensionnels. Dans les applications d'imagerie spatiale, et plus particulièrement dans le cas des satellites d'observation de la Terre, l'acquisition des lignes de pixels en continu se traduit par une hauteur d'image infinie [98, 99].

Paramètres – Notre implantation du composant virtuel impose des restrictions aux valeurs de paramètres supportées par le standard. Les paramètres de niveau fonctionnel sont les suivants :

- choix de l'ondelette : 9/7 ou 5/3 ;
- nombre de niveaux de décomposition : $N \geq 1$;
- largeur/hauteur d'une tuile, en pixels : L, H (H éventuellement infinie).

Pour une question de régularité de l'algorithme, nous imposons à L et H d'être multiples de 2^N. Nous assurons ainsi que chaque niveau de décomposition travaille sur une image de dimensions paires.

5.2 Transformation en ondelettes lifting 2D

5.2.1 Algorithme

Le document de référence qui a orienté le choix de l'algorithme de transformation en ondelettes est le *Final Committee Draft* de la norme de codage d'images *JPEG2000* [96]. L'algorithme préconisé consiste en une décomposition à base d'ondelettes biorthogonales calculée en appliquant la méthode du *Lifting Scheme*.

Nous présentons dans le listing 5.1 l'algorithme de transformation en ondelettes *2D* sur N niveaux sous forme de pseudo-code. Pour chaque niveau de résolution, une passe de filtrage *1D* est appliquée d'abord suivant chaque ligne, puis suivant chaque colonne. Chaque passe de filtrage *1D* se fait en $S + 1$ étapes, alternant tout d'abord $S/2$ pas de *dual lifting* et $S/2$ pas de *lifting*, et se terminant par un pas de *scaling*. Dans l'écriture de cet algorithme, nous avons tenu compte des propriétés spécifiques au ondelettes utilisées dans le standard *JPEG2000* [96] : (1) le nombre S de pas de lifting/dual lifting pour un étage de transformation *1D* est supposé pair ; (2) une passe de transformation *1D* commence toujours par un pas de dual lifting ; (3) chaque filtre de prédiction $P_i(z)$ ou de mise à jour $U_i(z)$ a une réponse de la forme : $\zeta(1 + z)$ ou $\zeta(1 + z^{-1})$, respectivement.

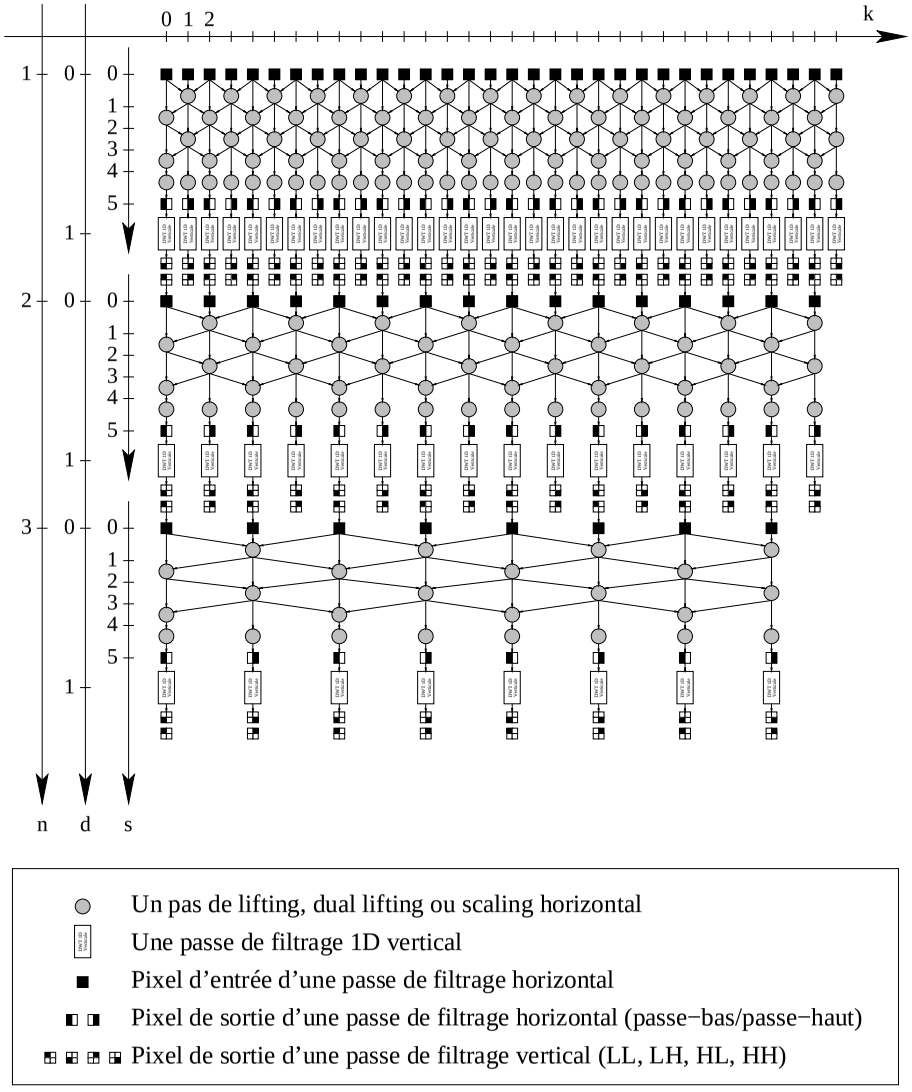
Dans la suite de ce chapitre, les exemples qui illustreront notre propos seront orientés vers un cas typique d'application utilisant l'ondelette 9/7 – préférée dans la majorité des applications pour son plus grand pouvoir de décorrélation [86] , mais présentant cependant une complexité supérieure à l'ondelette 5/3 – et réalisant une décomposition sur trois niveaux. Pour la compression des images naturelles, il est généralement reconnu qu'un nombre de niveaux supérieur à trois n'a en effet que peu d'influence sur les performances d'une chaîne de compression [86, 103].

Le graphe de dépendances de l'algorithme de transformation *2D* est présenté en figure 5.1 pour une décomposition sur 3 niveaux d'une image de 32 pixels de large avec l'ondelette 9/7 ($N = 3$ et $S = 4$). Chaque nœud est repéré par un quintuplet d'indices (l, k, n, d, s) où l et k désignent les coordonnées spatiales, n le niveau de décomposition courant, s l'indice du pas de *lifting* ou *dual lifting* courant et d la direction de la transformation (0 pour l'horizontale, 1 pour la verticale). La figure 5.1 donne une vue "en coupe" du graphe complet, dans laquelle seuls les nœuds relatifs aux passes de transformation horizontale suivant une ligne de l'image ont été représentés. Les nœuds en grisé représentent chacun, selon l'indice s des données qu'ils consomment, une opération de *dual lifting* (s pair strictement inférieur à S), de *lifting* (s impair) ou de *scaling* ($s = S$).

Listing 5.1 – Transformation en ondelettes lifting 2D pour JPEG2000

```
' pour chaque niveau de décomposition
pour n de 1 à N faire
  '------------------------------ Transformation horizontale
  ' pour chaque ligne de l'image
  pour l de 0 à H par 2^(n-1) faire
    ' Pour chaque pas de lifting/dual lifting
    pour s de 1 à S faire
      si s pair alors
        kmin ←2^n − 1
        kmax ←L − 2^n − 1
      sinon
        kmin ←0
        kmax ←L − 2^n
      fin si

      ' Pour chaque paire de pixels
      pour k de kmin à kmax par 2^n faire
        si k = 0 alors
          ' Symétrie au bord gauche
          x(l,k) ←x(l,k) + 2ζ(s) × x(l,k + 2^n − 1)
        sinon, si k = L − 2^n − 1 alors
          ' Symétrie au bord droit
          x(l,k) ←x(l,k) + 2ζ(s) × x(l,k − 2^n − 1)
        sinon
          x(l,k) ←x(l,k) + ζ(s) × (x(l,k − 2^n − 1) + x(l,k + 2^n − 1))
        fin si
      fin pour (k)
    fin pour (s)

    ' Scaling
    pour k de 0 à L − 2^n par 2^n faire
      x(k) ←x(k)/K  ' Passe−bas
      x(k + 2^n − 1) ←x(k + 2^n − 1) × K  ' Passe−haut
    fin pour (k)
  fin pour (l)

  '------------------------------ Transformation verticale
  ' pour chaque colonne de l'image
  pour k de 0 à L par 2^(n-1) faire
    ' Pour chaque pas de lifting/dual lifting
    pour s de 1 à S faire
      {...}
    fin pour (s)
    ' Scaling
    pour l de 0 à H − 2^n par 2^n faire
      {...}
    fin pour (l)
  fin pour (k)
fin pour (n)
```

FIG. 5.1 – Graphe de dépendances de la DWT 2D Lifting sur trois niveaux pour l'ondelette 9/7



TAB. 5.1 – Nombre d'opérations par pixel pour la transformation en ondelettes lifting 2D

	DWT 1D 1 niveau	*DWT 2D* N niveaux	*DWT 2D* $N = 3$, $S = 4$
(Dual) Lifting	$S/2$	$S\sum_{n=1}^{N} 1/4^{n-1} < 4S/3$	5,3
Scaling	1	$2\sum_{n=1}^{N} 1/4^{n-1} < 8/3$	2,6
Additions	S	$2S\sum_{n=1}^{N} 1/4^{n-1} < 8S/3$	10,5
Multiplications	$S/2+1$	$(S+2)\sum_{n=1}^{N} 1/4^{n-1} < (4S+8)/3$	7,9

Paramètres – Les paramètres de niveau algorithmique sont des paramètres secondaires, c'est-à-dire qu'ils se déduisent directement du choix de l'ondelette. Ce sont :
- le nombre de pas de lifting/dual lifting : S pair ;
- la valeur des cœfficients de lifting/dual lifting : $\zeta(s)$ pour $1 \leq s \leq S$;
- la valeur du cœfficient de scaling : K.

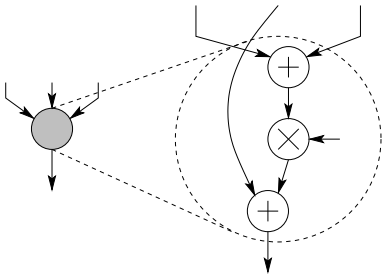
Propriétés – Le tableau 5.1 donne le nombre moyen d'opérations arithmétiques par pixel d'entrée ou de sortie. Ce nombre est exprimé à différents niveaux de granularité : (1) en pas de *lifting* ou *dual lifting* et multiplications de *scaling* d'une part, et (2) en additions et multiplications d'autre part. Il est donné tout d'abord dans le cas d'une transformation *1D* sur un niveau :
- pour une transformation *2D* sur 1 niveau, ce nombre doit être multiplié par deux afin de tenir compte des deux directions spatiales.
- pour N niveaux de décomposition *2D*, nous effectuons la somme des nombres moyens d'opérations par pixel pour chaque niveau de décomposition, la contribution d'un niveau d'indice n étant égale au quart de la contribution du niveau $n-1$ du fait du sous-échantillonnage par deux dans les deux directions spatiales.

On observe que l'augmentation du nombre de niveaux de décomposition n'a que peu d'influence sur le nombre moyen d'opérations par pixel, qui reste borné par une quantité ne dépendant que de la complexité des filtres *lifting* (complexité représentée par le nombre de pas de *lifting* et *dual lifting* S).

L'algorithme présente une structure très régulière qui consiste en la répétition d'une opération *lifting* ou *dual lifting* composée de deux additions et d'une multiplication (figure 5.2).

Au niveau algorithmique, la spécification du composant virtuel n'intègre aucune notion de temps. Il est cependant possible d'estimer la durée minimale de calcul d'une donnée de sortie en faisant l'hypothèse que toutes les données d'entrée dont elle dépend sont disponibles simultanément et que le calcul exploite le maximum de parallélisme entre opérations. Pour ce faire, nous dénombrons les opérations figurant sur la plus longue chaîne de dépendances entre une donnée de sortie et les données d'entrée dont elle dépend. Le tableau 5.2 indique les quantités obtenues pour les cœfficients de détail à chaque niveau n ($1 \leq n \leq N$) et pour les cœfficients d'approximation au niveau N. La durée minimale du calcul est exprimée en fonction de la durée d'une addition (T_{add}) et de la durée d'une multiplication (T_{mult}).

FIG. 5.2 – Anatomie d'un nœud lifting ou dual lifting



TAB. 5.2 – Estimation de la durée minimale de calcul d'une donnée de sortie de la transformation en ondelettes lifting 2D

	Détail (niveau n)	Approximation (niveau N)
Additions	$2(Sn-1)$	$2SN$
Multiplications	$(S+1)n-1$	$(S+1)N$
Latence pour $N=3$ et $S=4$	$(8n-2)T_{add}$ $+(5n-1)T_{mult}$	$24T_{add}$ $+15T_{mult}$

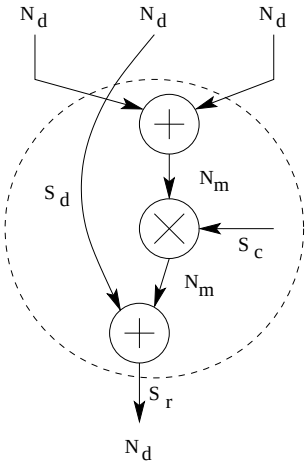
5.2.2 Choix des formats de données

Le composant virtuel *DWT 2D* a pour vocation de s'insérer en tant que décorrélateur dans une chaîne de compression d'images. Il doit en particulier pouvoir s'adapter à différentes contraintes de qualité de restitution de ces images. L'étude de précision, par analyse systématique de la dynamique des données intermédiaires produites au cours du calcul, a permis d'une part de dimensionner le chemin de données de manière à assurer un bruit de calcul minimal tout au long de la transformation, et d'autre part de permettre à l'utilisateur de choisir au moyen de paramètres le meilleur compromis entre complexité architecturale (taille des éléments de mémorisation et des opérateurs arithmétiques) et qualité de restitution des images.

Paramètres primaires – Les données d'entrée/sortie sont supposées être exprimées en virgule fixe cadrée à gauche. Les paramètres primaires régissant le format des données sont :

- le nombre de bits N_i des pixels d'entrées, le nombre de bits éventuels en partie fractionnaire F_i et le caractère signé/non signé des valeurs C_i ;
- le nombre de bits N_o des pixels de sortie et le nombre de bits F_o de leur partie fractionnaire ;
- le nombre de bits N_d des données intermédiaires en entrée/sortie d'un pas de *lifting*, *dual lifting* ou *scaling* ;
- le nombre maximum de bits N_m des données intermédiaires à l'intérieur d'un pas de *lifting* ou *dual lifting* (figure 5.3).

FIG. 5.3 – Nombre de bits des données dans un nœud de lifting ou dual lifting



Paramètres secondaires – Les paramètres primaires que nous avons définis imposent une taille de données fixe en entrée/sortie de chaque nœud *lifting* ou *dual lifting*.

A l'intérieur de chacun de ces nœuds, les additions et multiplications peuvent engendrer des débordements et nécessitent une adaptation de la dynamique des données. Cette adaptation se fait au moyen de décalages de bits qui peuvent être exprimés dans la description comportementale sous forme de multiplications ou divisions par des puissances de deux – sans induire pour autant un coût supplémentaire en ressources matérielles : les outils de synthèse de haut niveau savent généralement interpréter ces opérations en termes de manipulations de bits et n'instancient pas de composant multiplieur ou diviseur pour les réaliser [53, 57].

Pour chaque pas de *lifting* ou *dual lifting* d'indice s, trois paramètres déterminent les décalages à appliquer (figure 5.3) :

- $S_c(s)$ donne le décalage à appliquer aux cœfficients $\zeta(s)$ courant ;
- $S_d(s)$ correspond au décalage à appliquer à l'entrée centrale du nœud courant ;
- $S_r(s)$ correspond au décalage à appliquer à la sortie du nœud courant.

Ces paramètres sont calculés de manière automatique, en fonction des paramètres primaires, au moment de l'instanciation du composant virtuel.

Propriétés – Généralement, la mesure du bruit de calcul s'effectue en calculant la différence entre les pixels de sortie pour l'algorithme en précision infinie et les pixels de sortie pour l'algorithme en précision fixe.

Dans le cadre d'une application de compression d'image, la mesure du bruit de calcul est essentiellement destinée à caractériser la *réversibilité* de la transformation, c'est-à-dire l'aptitude à reconstruire fidèlement l'image d'origine à partir de l'image transformée. Il sera alors préférable de mesurer le bruit de calcul en comparant l'image

d'origine avec l'image ayant subi successivement une transformation directe en précision fixée et une transformation inverse en précision infinie.

Le bruit de calcul peut être exprimé comme l'erreur quadratique moyenne (EQM), l'erreur RMS, le rapport signal à bruit (SNR) ou encore le rapport signal à bruit crête ($PSNR$) entre l'image source $x(l,k)$ et l'image reconstruite $\hat{x}(l,k)$:

$$\begin{aligned} EQM &= \frac{1}{LH}\sum_{l=0}^{H-1}\sum_{k=0}^{L-1}\Big(\hat{x}(l,k)-x(l,k)\Big)^2 \\ RMS &= \sqrt{EQM} \\ SNR &= 10\log_{10}\frac{\sigma_x^2}{EQM} \\ PSNR &= 10\log_{10}\frac{max_{l,k}\{x(l,k)^2\}}{EQM} \end{aligned}$$

Exemple – La figure 5.5 donne les valeurs de $PSNR$ obtenues pour une transformation sur 3 niveaux avec l'ondelette 9/7 avec différentes tailles N_d de données en entrée/sortie des nœuds *lifting*, *dual lifting* et *scaling*. Ces valeurs ont été établies sur l'image *Lena* (figure 5.4) de taille 512×512 pixels, pour des valeurs de pixels d'entrée codés en nombres entiers non signés sur $N_i = 8$ bits, des pixels de sortie sur $N_o = 13$ bits dont $F_o = 2$ bits de partie fractionnaire et une taille maximale de données internes $N_m = 32$ bits.

Le $PSNR$ atteint son maximum (environ $90dB$) pour $N_d = 22$ bits. Pour $N_d > 22$, la qualité chute brutalement. Cette perte s'interprète par le fait que, pour une valeur constante de N_m, l'augmentation de précision sur les données intermédiaires s'accompagne d'une perte de précision sur les cœfficients de *lifting* et *dual lifting* – en figure 5.3, le nombre de bits pour ces cœfficients en entrée de la multiplication sera de l'ordre de $N_m - N_d$ afin d'assurer N_m bits pour le résultat de la multiplication. Pour $N_m = 32$, le point $N_d = 22$ réalise ainsi le meilleur compromis entre précision des données et précision des cœfficients.

Si l'on souhaite réduire la taille des données internes tout en conservant la même qualité d'image, il est également possible de jouer sur la taille de la partie fractionnaire des données de sortie :

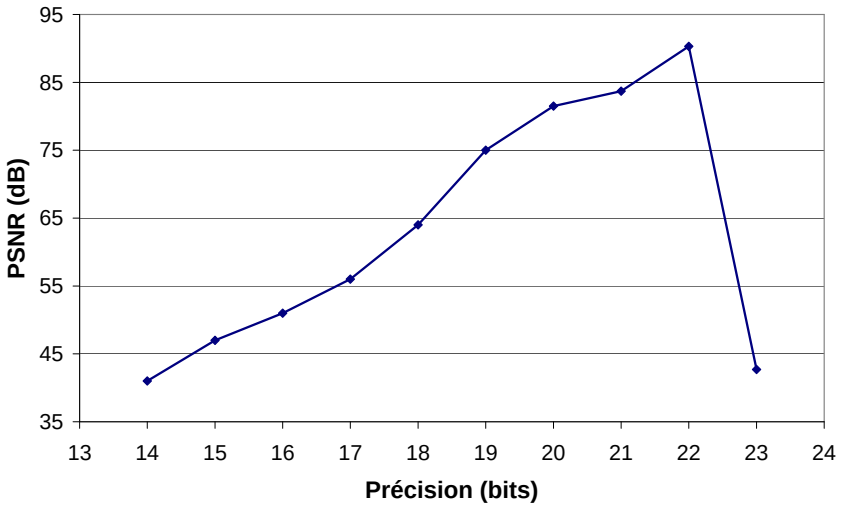
- pour $F_o = 3$ bits ($N_o = 14$ bits), la transformation devient complètement réversible ($PSNR$ infini) à partir de $N_d = 19$ bits ;
- pour $F_o = 5$ bits ($N_o = 16$ bits), la transformation est réversible à partir de $N_d = 18$ bits.

5.3 Transformation en ondelettes 2D au fil de l'eau

En fonction des contraintes de l'application et du système cible, deux types d'implantation d'un composant virtuel de transformation en ondelettes peuvent être envisagés : (1) le premier consiste à implanter l'algorithme de transformation présenté dans la section précédente en supposant que tous les pixels de l'image source sont disponibles en mémoire dès le début du calcul ; (2) le second suppose que les pixels sont fournis au composant dans un ordre déterminé au cours du temps. Dans le premier type d'implantation, le composant virtuel est maître des accès mémoire et traite

FIG. 5.4 – Image test pour le calcul de PSNR : Lena

FIG. 5.5 – Mesure de PSNR pour différentes tailles de données intermédiaires



les données à son propre rythme. Le second type est plus adapté aux applications temps réel dans lesquelles le traitement doit être effectué au rythme de l'acquisition des données.

Dans le contexte de la compression d'images spatiales, le coût du stockage des données à bord des satellites ne permet pas de conserver en mémoire la totalité des données image [98,99]. L'acquisition en continu de lignes de pixels impose le traitement en temps réel des données. Les techniques de compression d'image *au fil de l'eau* permettent de coder et de transmettre les données au fur et à mesure qu'elles sont acquises en minimisant la mémoire nécessaire au traitement : typiquement, à chaque fois qu'une ligne ou un groupe de lignes de pixels a été codé, ne sont conservées en mémoire que les données nécessaires au codage de la prochaine ligne ou du prochain groupe de ligne.

Cette problématique nous a conduit à rechercher dans le graphe de dépendances de la transformation *lifting* d'une image entière des motifs simples correspondant à différentes stratégies de transformation au fil de l'eau et répondant à différents compromis vitesse/complexité/quantité de mémoire [3,5–7]. Nous nous sommes efforcés de spécifier en une unique description comportementale générique un éventail de motifs pouvant répondre à différents jeux de contraintes.

5.3.1 Choix des axes temporels et causalité

Chaque donnée intermédiaire de la description algorithmique peut être repérée par un quintuplet d'indices (n, d, s, l, k) avec :

- n le niveau de décomposition courant : $1 \leq n \leq N$;
- d la direction de la passe de filtrage courante : $d = 0$ pour un filtrage horizontal ; $d = 1$ pour un filtrage vertical ;
- s l'indice du pas de calcul courant : $s = 0$ pour les données source ; s impair et $1 \leq s < S$ pour un pas de *dual lifting* ; s pair et $1 < s \leq S$ pour un pas de *lifting* ; $s = S + 1$ pour un pas de *scaling* ;
- l l'indice de la ligne de pixels courante dans l'image ;
- k l'indice de la colonne de pixels courante dans l'image.

La répartition des données et des opérations pour une ligne d'une image est représentée par le graphe de dépendances de la figure 5.1 pour une transformation sur trois niveaux avec l'ondelette 9/7. Nous rappelons que la notation $I_2 \succeq I_1$ (avec $I_1 = (n_1, d_1, s_1, l_1, k_1)$ et $I_2 = (n_2, d_2, s_2, l_2, k_2)$) exprime la relation "la donnée d'indices I_2 dépend de la donnée d'indices I_1" (voir annexe B). Le choix de l'opérateur "$\succeq$" implique une notion de succession des deux données dans le temps : en supposant que la durée de chaque opération est nulle, la donnée d'indices I_2 sera disponible au même instant que, ou après la donnée d'indices I_1. Les dépendances entre deux données consécutives sont de la forme suivante :

- pour les nœuds de *lifting* ou *dual lifting* horizontaux :

$$(n, 0, s, l, k) \succeq (n, 0, s-2, l, k) \text{ pour } 2 \leq s \leq S \tag{5.1}$$

$$(n, 0, 1, l, k) \succeq (n, 0, 1, l, k) \tag{5.2}$$

$$(n, 0, s, l, k) \succeq (n, 0, s-1, l, k \pm 2^{n-1}) \text{ pour } 1 \leq s \leq S \tag{5.3}$$

– pour les nœuds de *lifting* ou *dual lifting* verticaux :

$$(n,1,s,l,k) \succeq (n,1,s-2,l,k) \text{ pour } 2 \leq s \leq S \quad (5.4)$$
$$(n,1,1,l,k) \succeq (n,1,1,l,k) \quad (5.5)$$
$$(n,1,s,l,k) \succeq (n,1,s-1,l \pm 2^{n-1},k) \text{ pour } 1 \leq s \leq S \quad (5.6)$$

– pour les nœuds de *scaling* horizontaux et verticaux :

$$(n,0,S+1,l,k) \succeq (n,0,S,l,k) \quad (5.7)$$
$$(n,1,S+1,l,k) \succeq (n,1,S,l,k) \quad (5.8)$$

– entre étages de transformation horizontale et verticale :

$$(n,0,0,l,k) \succeq (n-1,1,S+1,l,k) \quad (5.9)$$
$$(n,1,0,l,k) \succeq (n,0,S+1,l,k) \quad (5.10)$$

Axes temporels – L'extraction d'un motif de transformation au fil de l'eau impose tout d'abord de définir une relation d'ordre partiel sur les pixels d'entrée et/ou de sortie (voir annexe B). Ici, nous prenons pour hypothèse que l'ordre de lecture des pixels de l'image d'origine suit l'ordre des indices l et k, c'est-à-dire qu'un pixel d'une ligne donnée sera lu avant, ou en même temps que ses voisins de droite, et qu'un pixel d'une colonne donnée sera lu avant, ou en même temps que ses voisins situés plus bas. Rappelons que $I_1 \preceq I_2$ signifie ici : "la donnée d'indices I_1 est disponible avant, ou en même temps que la donnée d'indices I_2" (voir annexe B).

$$\forall l,\ k_1 \leq k_2 \Rightarrow (0,0,0,l,k_1) \preceq (0,0,0,l,k_2)$$
$$\forall k,\ l_1 \leq l_2 \Rightarrow (0,0,0,l_1,k) \preceq (0,0,0,l_2,k)$$

Pour le moment, nous ne spécifions pas de priorité entre ces deux axes.

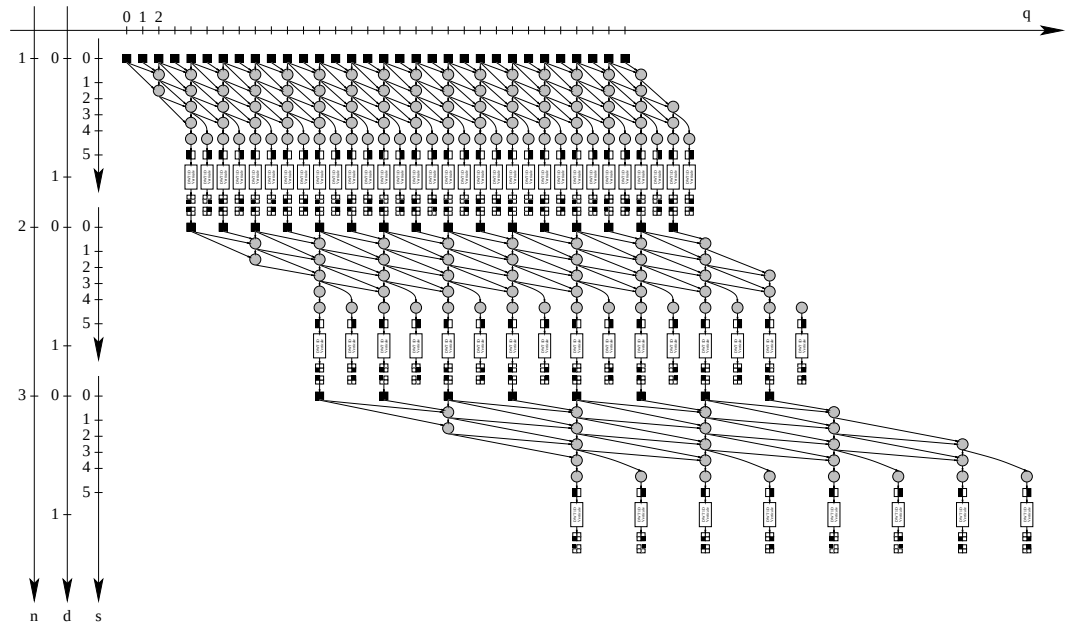
Résolution des dépendances non causales – Comme le montre la figure 5.1, si les indices l et k permettent de spécifier différents ordres de parcours des pixels d'entrée, ils ne permettent pas de représenter l'ordre relatif des données et des calculs en respectant la causalité. Chaque pas de *lifting* ou *dual lifting* horizontal (*resp.* vertical) dépend en effet de données situées à droite (*resp.* en-dessous) du pixel en cours de traitement. Les dépendances *non causales* relativement aux indices l et k entre données d'entrée/sortie d'un nœud *lifting* ou *dual lifting* sont représentées par les relations 5.3 et 5.6 dans lesquelles la donnée du membre de gauche dépend d'une donnée située à sa droite sur la même ligne, ou plus bas sur la même colonne de l'image.

Les relations 5.1 à 5.10 nous permettent de déduire l'étendue spatiale des dépendances non causales entre chaque donnée intermédiaire et les données de l'image source dont elle dépend. A l'aide d'un raisonnement par récurrence et après simplification des expressions, nous obtenons :

– pour un nœud de *lifting* ou *dual lifting* ($0 \leq s \leq S$) :

$$(n,0,s,l,k) \succeq (0,0,0,l+S\times(2^{n-1}-1),k+(S+s)\times 2^{n-1}-S) \quad (5.11)$$
$$(n,1,s,l,k) \succeq (0,0,0,l+(S+s)\times 2^{n-1}-S,k+S\times(2^n-1)) \quad (5.12)$$

FIG. 5.6 – Graphe de dépendances causal de la DWT 2D Lifting pour l'ondelette 9/7



– pour un nœud de *scaling* ($s = S + 1$) :

$$(n,0,s,l,k) \succeq (0,0,0,l+S\times(2^{n-1}-1),k+S\times(2^n-1)) \quad (5.13)$$

$$(n,1,s,l,k) \succeq (0,0,0,l+S\times(2^n-1),k+S\times(2^n-1)) \quad (5.14)$$

La résolution des dépendances non causales se fait à l'aide du changement d'indices $(l,k) \longrightarrow (r,q)$ défini comme suit :

– pour $d = 0$ et $0 \leq s \leq S$:

$$q = k + (S+s) \times 2^{n-1} - S \; ; \; r = l + S \times (2^{n-1} - 1)$$

– pour $d = 1$ et $0 \leq s \leq S$:

$$q = k + S \times (2^n - 1) \; ; \; r = l + (S+s) \times 2^{n-1} - S$$

– pour $d = 0$ et $s = S + 1$:

$$q = k + S \times (2^n - 1) \; ; \; r = l + S \times (2^{n-1} - 1)$$

– pour $d = 1$ et $s = S + 1$:

$$q = k + S \times (2^n - 1) \; ; \; r = l + S \times (2^n - 1)$$

Le changement d'indice selon la direction horizontale est représenté en figure 5.6 pour la transformation sur 3 niveaux avec l'ondelette 9/7 [6, 7].

TAB. 5.3 – Propriétés du motif de transformation en ondelettes lifting 2D

	Cas général	$w = h = 1$ $N = 3\,;\ S = 4$
Additions	$8S(4^N - 1)wh/3$	672
Multiplications	$(4S + 8)(4^N - 1)wh/3$	504
Pixels entrée	$4^N \times wh$	64
Pixels sortie	$4^N \times wh$	64
Contexte H. E/S	$2Sh(2^N - 1)$	56
Contexte V. E/S	$2Sw(2^N - 1)$	56
Total entrées	–	176
Total sorties	–	176

5.3.2 Choix de motifs de calcul

L'extraction d'un motif répétitif de calcul pour la transformation en ondelettes *lifting* repose sur l'observation que le graphe de dépendances – à l'exception des nœuds situés au voisinage des bords – présente une périodicité de 2^N, (N étant le nombre de niveaux de décomposition) suivant les indices q et r définis plus haut [6,7].

L'exploitation directe de cette périodicité permet d'extraire un motif de la forme schématisée en figure 5.7. Un tel motif a une largeur L_{motif} (selon l'axe des q) et une hauteur H_{motif} (selon l'axe des r) multiples de 2^N.

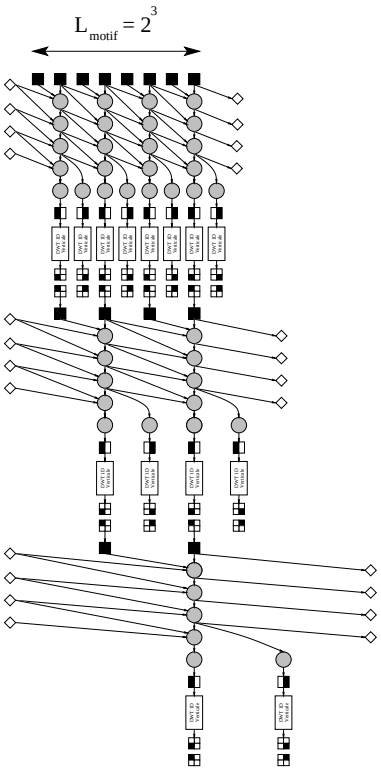
$$L_{motif} = w \times 2^N$$
$$H_{motif} = h \times 2^N$$

Chaque itération du motif consomme un bloc de $L_{motif} \times H_{motif}$ pixels de l'image d'origine et produit la même quantité de pixels de l'image transformée (cœfficients d'approximation au niveau N et de détail aux niveaux 1 à N). Le composant peut ainsi être inséré dans un environnement de communication conforme au modèle *flot de données synchrone* (*SDF*).

A chaque itération, ce motif consomme également des données intermédiaires (ou données de contexte) calculées lors d'itérations précédentes. Une partie des données intermédiaires produites lors de l'itération courante sera ainsi réutilisée lors d'itérations ultérieures. La durée de vie de ces données, exprimée comme le nombre d'itérations du motif entre leur production et leur dernière consommation, dépend de l'ordre de parcours des blocs de pixels dans l'image. Typiquement, dans un parcours de type *raster* – pour lequel les pixels sont fournis ligne de blocs par ligne de blocs de haut en bas, et bloc par bloc de gauche à droite – le contexte horizontal est mis à jour à chaque itération du motif tandis que les données de contexte vertical en sortie de l'itération courante ne seront réutilisées que lorsqu'une ligne de blocs complète aura été traitée.

Le tableau 5.3 donne les grandeurs caractéristiques de la complexité du motif, en nombre d'opérations arithmétiques d'une part, et en nombre de transferts de données par itération d'autre part.

FIG. 5.7 – Motif de calcul pour l'ondelette 9/7 sur trois niveaux



TAB. 5.4 – Durée de vie des données et contraintes de mémorisation pour le motif de transformation en ondelettes lifting 2D, avec un parcours raster par blocs des pixels de l'image source

	Cas général		$L = 256$; $w = h = 1$ $N = 3$; $S = 4$		
Données	Durée de vie	Qté de mémoire	Durée de vie	Qté de mémoire	Support
Contexte horizontal	1	$2Sh(2^N - 1)$	1	56	Registres RAM interne
Contexte vertical	L/L_{motif}	$2S(1 - 1/2^N)L$ $< 2SL$	32	1792	RAM externe

Le tableau 5.4 donne la durée de vie des données du contexte horizontal et vertical dans le cas d'un parcours *raster*. La quantité de mémoire est exprimée comme le nombre de valeurs produites par le motif sur un intervalle de temps égal à la durée de vie des données de contexte. La quantité de mémoire nécessaire au stockage du contexte horizontal ne dépend que de la *hauteur* $H_{motif} = h \times 2^N$ du motif, de la complexité S du banc de filtres *lifting* et du nombre N de niveaux de décomposition. Observons par ailleurs que la quantité de mémoire requise par le contexte vertical est bornée en valeur supérieure par une quantité ne dépendant que de la largeur de l'image et de la complexité de l'ondelette. Ces deux quantités ne dépendent pas de la largeur L_{motif} du motif : il est donc possible de choisir un motif arbitrairement large – *e.g.* afin d'ajuster le parallélisme de calcul exploitable – sans pour autant augmenter la quantité de mémoire nécessaire.

La quantité importante de données du contexte vertical devant être stockées et leur longue durée de vie privilégie un support de stockage de type *RAM* externe. Selon les performances en vitesse souhaitées, les données du contexte horizontal pourront être soit stockées dans une *RAM* interne au composant virtuel, au prix d'un débit d'accès plus limité, soit mises en registres au prix d'une plus grande complexité matérielle de l'architecture.

5.3.3 Séparation du contrôle et du traitement dans le motif de transformation en ondelettes 2D

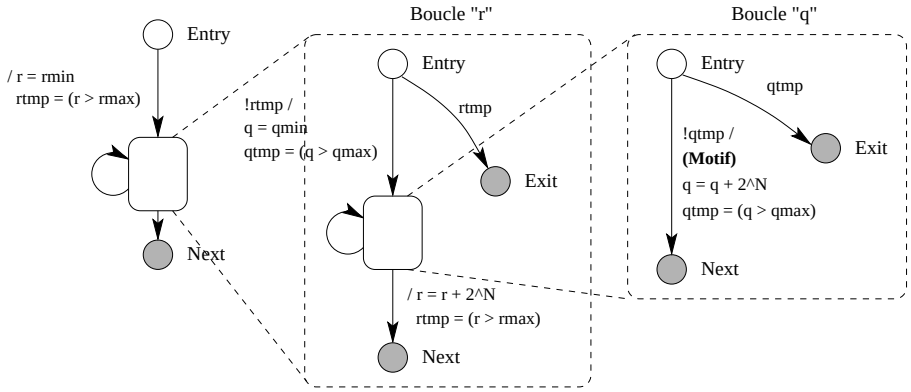
La mise en évidence du motif de calcul dans la description comportementale s'effectue en réarrangeant la hiérarchie des boucles de manière à placer au niveau le plus profond les instructions relatives à une itération du motif. Ce réarrangement est illustré par le listing 5.2 : les boucles d'indices r et q placées aux plus hauts niveaux donnent l'ordre de succession des itérations du motif. Cet ordre correspond également à un parcours par blocs de l'image à transformer, ici un parcours de type *raster* sur des blocs rectangulaires de dimensions H_{motif} et L_{motif}.

Listing 5.2 – Mise en évidence du motif de transformation en ondelettes lifting 2D

```
' pour chaque itération du motif
pour r de r_min à r_max par H_motif faire
  pour q de q_min à q_max par L_motif faire
    'Motif
    {...}
  fin pour (q)
fin pour (r)
```

En supposant que toutes les boucles à l'intérieur du motif sont déroulées, la description comportementale du composant présente deux niveaux de boucles non déroulées correspondant aux indices r et q. En mettant en évidence les primitives de synthèse de haut niveau, nous pouvons en déduire le modèle *HSFSMD* de la figure 5.8. La forte régularité de la transformation en ondelettes aboutit à un modèle *HSFSMD* comprenant peu d'états : l'absence d'instructions de branchement à l'intérieur des boucles se traduit en effet par une faible complexité de la partie contrôle de la description. Une version optimisée de cette *HSFSMD* est présentée en figure 5.9 : les initialisations des

FIG. 5.8 – Modèle HSFSMD du motif de transformation en ondelettes 2D



deux compteurs de boucles ont été regroupées sur une même transition et le motif s'inscrit dans un unique niveau de boucle non déroulée.

Paramètres statiques et paramètres dynamiques – Par construction, les bornes d'itérations des boucles internes au motif sont indépendantes des dimensions L et H de l'image. Ainsi, seules les boucles d'indices r et q ont un nombre d'itérations dépendant de la taille de l'image. Ces boucles ne devant pas être déroulées, nous pouvons ainsi envisager de faire figurer L et H non plus comme des paramètres *statiques* du composant virtuel – *i.e.* dont la valeur serait figée au moment de la synthèse du composant – mais comme des paramètres *dynamiques*, dont la valeur serait fournie sur des ports d'entrée spécifiques, autorisant ainsi une même instance du composant à traiter des images de différentes tailles.

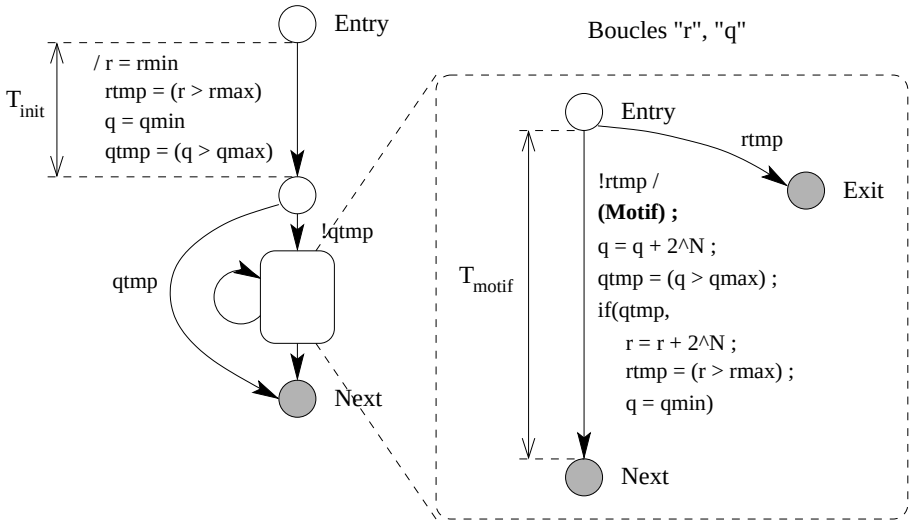
Si la largeur effective des images n'est plus connue au moment de la synthèse, le dimensionnement de la mémoire de contexte vertical nécessite alors de connaître la largeur *maximale* des images pouvant être traitées par le composant. Cette largeur maximale étant *a priori* dépendante de l'application, elle sera fournie sous la forme d'un paramètre statique L_{max}.

Insertion des contraintes de temps – A partir du modèle *HSFSMD* de la figure 5.9, la spécification des contraintes de temps se réduit à préciser les durées T_{init} et T_{motif} – en cycles d'horloge – des deux transitions sur lesquelles s'effectuent respectivement l'initialisation des compteurs de boucle et les calculs relatifs au motif.

5.3.4 Taille de motif, contraintes de temps et parallélisme de calcul

Le motif se caractérise par le fait que la quantité de calculs relative à chaque niveau de décomposition est divisée par quatre entre un niveau et le suivant. Lors de la synthèse de la description, l'étape d'ordonnancement devra s'appliquer à homogénéiser la charge de calcul au cours du temps de manière, d'une part, à limiter le nombre de ressources de calcul, et d'autre part à lisser la puissance électrique consommée au cours d'une itération du motif.

FIG. 5.9 – Modèle HSFSMD optimisé du motif de transformation en ondelettes 2D



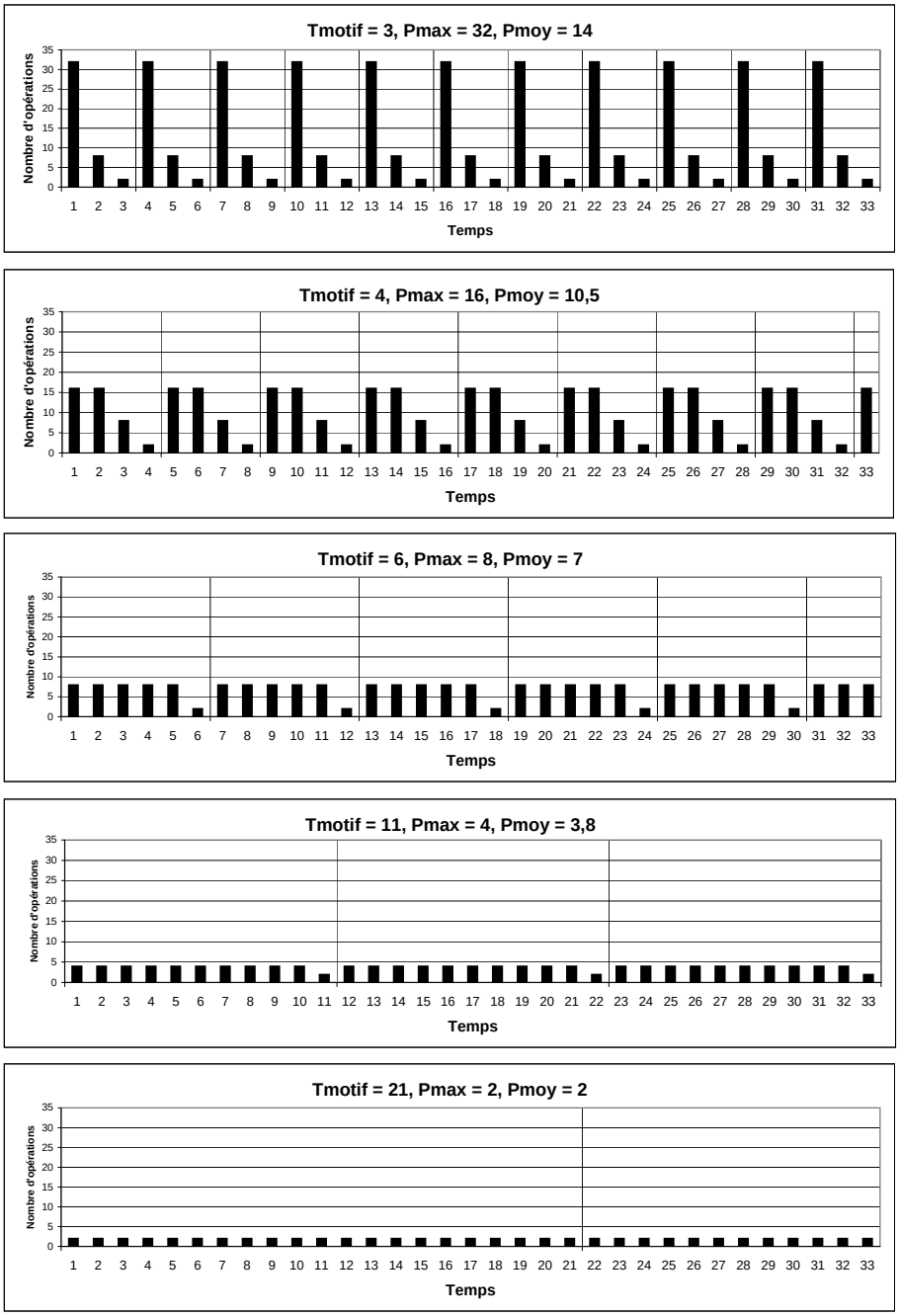
Comme le montre la figure 5.7, tous les nœuds *lifting*, tous les nœuds *dual lifting* et la moitié des nœuds *scaling* du motif appartiennent à un chemin critique. Deux approches sont envisageables pour lisser la répartition temporelle de la charge de calcul et maîtriser le parallélisme entre opérations : si la contrainte de temps est une contrainte de *latence* (applicable dans des outils comme *Monet* et *Behavioral Compiler* [53, 57]), il faudra spécifier une valeur de T_{motif} assez grande pour assurer une mobilité suffisante des nœuds du *DFG* ; si au contraire la contrainte de temps est une contrainte de *cadence* (applicable avec l'outil *GAUT* [61]), la parallélisme peut être lissé en appliquant des techniques d'ordonnancement *pipeline* (figure 1.8 page 36).

La figure 5.10 donne la répartition de la charge de calcul au cours du temps pour la transformation sur trois niveaux avec l'ondelette 9/7 et une taille de motif égale à 8×8. Pour le moment, nous prenons pour unité de temps la durée T_{niveau} d'une passe de transformation *2D* sur un niveau. Pendant chaque intervalle de temps, le nombre d'opérations est exprimé comme le nombre de nœuds *lifting* ou *dual lifting* exécutés en parallèle.

Différentes contraintes de temps sont étudiées, correspondant à différents degrés de parallélisme de calcul : P_{max} représente le nombre maximum d'opérations pouvant être effectuées simultanément et P_{moy} le nombre moyen d'opérations exécutées en parallèle sur une itération du motif. Sur la figure 5.10, nous observons que le fait de limiter le parallélisme permet de lisser la charge de calcul : pour le plus fort parallélisme de calcul ($P_{max} = 32$), le parallélisme moyen n'est en réalité que de $P_{moy} = 14$, ce qui signifie que le composant n'exploite en moyenne que 44% de la puissance de calcul disponible (tableau 5.5). Pour $P_{max} = 2$, en revanche, le parallélisme exploité est constant : les ressources sont utilisées à plein rendement. Comparée à la solution $P_{max} = 32$, la durée d'une itération du motif est cependant sept fois plus longue.

Si nous généralisons les résultats correspondant aux deux solutions présentées ci-

FIG. 5.10 – Répartition de la charge de calcul au cours du temps pour différents degrés de parallélisme



TAB. 5.5 – Parallélisme et rendement des ressources de calcul en fonction de la contrainte de temps ($S = 4$, $N = 3$, $w = h = 1$)

T_{motif}/T_{niveau}	3	4	6	11	21
P_{moy}	14	10,5	7	3,8	2
P_{max}	32	16	8	4	2
$\eta = P_{moy}/P_{max}$	44%	66%	88%	95%	100%

dessus, nous obtenons pour la solution exploitant le plus fort parallélisme :

$$\begin{aligned} T_{motif} &= N \times T_{niveau} \\ P_{max} &= \frac{L_{motif} \times H_{motif}}{2} = \frac{wh}{2} 4^N \\ P_{moy} &= \frac{P_{max}}{T_{motif}} \sum_{n=1}^{N} \frac{1}{4^{n-1}} = 2wh \frac{4^N - 1}{3N} \end{aligned}$$

et pour la solution *lissée* :

$$\begin{aligned} T_{motif} &= \frac{4^N - 1}{3} \times T_{niveau} \\ P_{max} &= P_{moy} = 2wh \end{aligned}$$

Nous constatons que la sélection du meilleur compromis *vitesse de traitement/rendement des ressources de calcul* peut être résolue d'une manière simple en choisissant une contrainte de temps T_{motif} suffisamment large et en jouant sur la taille du motif. Pour une durée d'itération T_{motif} fixée il existe alors une relation de proportionnalité entre la cadence de traitement, en pixels par seconde, et le parallélisme de calcul exploité.

5.3.5 Réduction de la complexité en vue de la synthèse de haut niveau

L'algorithme de transformation en ondelettes que nous avons choisi d'implanter présente une forte complexité calculatoire. Nous avons vu dans le tableau 5.3 que le nombre d'opérations arithmétiques par itération, pour le motif de calcul le plus petit dans le cadre d'une transformation sur trois niveaux avec l'ondelette 9/7, dépasse 670 additions et 500 multiplications. Les outils de synthèse du commerce sont généralement limités quant à la complexité calculatoire des descriptions comportementales : par exemple, dans le manuel d'utilisation de l'outil *Behavioral Compiler* de *Synopsys*, il est recommandé de ne pas dépasser 150 opérations par niveau de boucle non déroulée.

Dans notre situation, deux solutions peuvent être envisagées pour réduire le nombre d'opérations qu'un outil devra ordonnancer : la première consiste à ne pas dérouler certaines boucles internes au motif. La seconde consiste à identifier des sous-motifs simples utilisés fréquemment et à les synthétiser séparément sous forme de composants combinatoires [6, 7].

TAB. 5.6 – Nombre d'opérations à ordonnancer dans le corps des boucles non déroulées

Opérations	Quantité (cas général)	Cas $N = 3$, $S = 4$	
		$w = h = 1$	$w = h = 2$
Additions	$8Swh$	32	128
Multiplications	$(4S + 8)wh$	24	96
Total	$(12S + 8)wh$	56	224

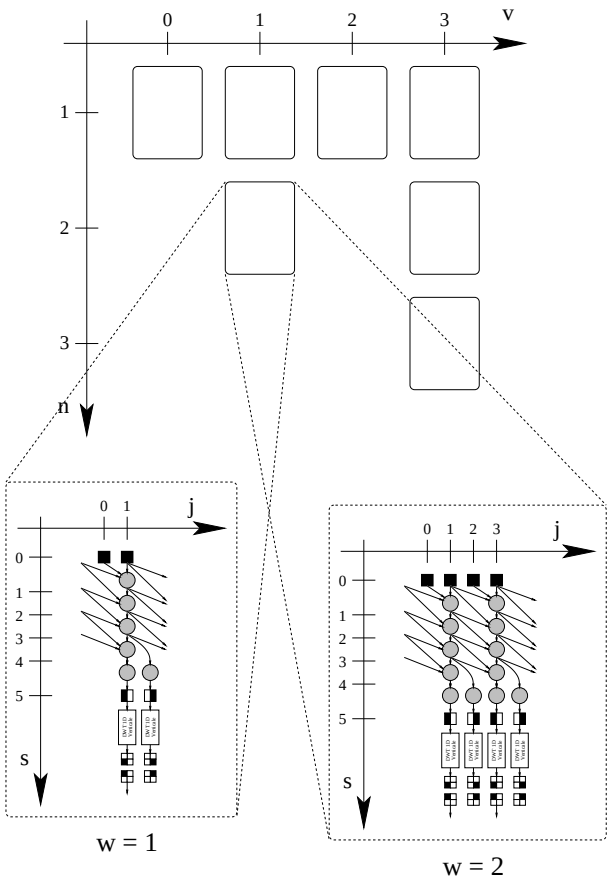
Utilisation de boucles non déroulées – A la différence des boucles déroulées, dans lesquelles l'ordre des calcul est imposé uniquement par les dépendances de données, les boucles non déroulées imposent l'ordre dans lequel des fragments du comportement seront exécutés. Le choix d'une description à base de boucles non déroulées revient ainsi à isoler dans le *DFG* du motif un sous-motif dont il faudra spécifier l'ordre d'exécution des itérations.

Un sous-motif de la forme présentée en figure 5.11 présente deux propriétés intéressantes en regard des considérations développées au paragraphe 5.3.4 : (1) la structure à base de boucles non déroulées ne permet plus de spécifier une contrainte de temps globale pour le motif, la durée d'exécution devenant multiple de la durée d'une itération du sous-motif ; (2) la structure du sous-motif limite le parallélisme maximum exploitable, le parallélisme moyen peut être réglé en jouant sur la taille du motif. Le tableau 5.6 indique le nombre d'opérations (additions et multiplications) figurant au niveau le plus profond de la hiérarchie des boucles non déroulées, dans le cas général et pour deux exemples de taille de motif. Dans le cas $w = h = 1$, le non-déroulage des boucles permet de diviser par 21 la quantité d'opérations *vues* par un outil de synthèse de haut niveau.

L'ordonnancement des itérations du sous-motif est directement imposé par la hiérarchie des boucles non déroulées. Au niveau architectural, le choix de l'algorithme d'ordonnancement aura une influence sur la quantité de mémoire nécessaire au stockage des données intermédiaires. Nous nous trouvons ainsi confrontés au choix d'un algorithme d'ordonnancement qui minimisera la durée de vie des données circulant entre itérations du sous-motif. Deux candidats, déjà présentés dans le chapitre 4, sont l'algorithme en pyramide (*PA*) et l'algorithme en pyramide récursif (*RPA*).

L'implantation de chacun de ces deux algorithmes dans la description comportementale nécessite un réarrangement de la hiérarchie des boucles. Les listings 5.3 et 5.4 présentent cette nouvelle hiérarchie pour l'algorithme en pyramide et l'algorithme en pyramide récursif respectivement. Dans l'élaboration du modèle *HSFSMD* proposé au paragraphe 5.3.3, nous avions fait l'hypothèse que toutes les boucles internes au motif étaient déroulées. L'implantation explicite du *PA* ou du *RPA* introduit trois niveaux supplémentaires de boucles non déroulées, qui se caractériseront par un modèle *HSFSMD* comprenant en tout cinq états hiérarchiques imbriqués. L'allure de la partie contrôle des deux modèles différera essentiellement par la présence, dans le cas du *RPA*, d'un état et d'une transition supplémentaires au niveau de boucle le plus profond (instruction "sortir" dans le listing 5.4).

FIG. 5.11 – Utilisation de boucles non déroulées dans le motif de décomposition en ondelettes 9/7 sur trois niveaux



Listing 5.3 – Parcours du motif avec l'algorithme en pyramide

```
pour n de 1 à N faire
  pour u de 2^(n-1) − 1 à 2^(N-1) − 1 par 2^(n-1) faire
    pour v de 2^(n-1) − 1 à 2^(N-1) − 1 par 2^(n-1) faire
      '————————————— Transformation horizontale
      pour i de 0 à 2h − 1 faire
        ' Lifting/dual lifting
        pour s de 1 à S faire
          pour j de 0 à 2w − 1 par 2 faire
            {...}
          fin pour (j)
        fin pour (s)
        ' Scaling
        pour j de 0 à 2w − 1 par 2 faire
          {...}
        fin pour (j)
      fin pour (i)
      '————————————— Transformation verticale
      pour j de 0 à 2w − 1 faire
        ' Lifting/dual lifting
        pour s de 1 à S faire
          pour i de 0 à 2h − 1 par 2 faire
            {...}
          fin pour (i)
        fin pour (s)
        ' Scaling
        pour i de 0 à 2h − 1 par 2 faire
          {...}
        fin pour (i)
      fin pour (j)
    fin pour (v)
  fin pour (u)
fin pour (n)
```

Listing 5.4 – Parcours du motif avec l'algorithme en pyramide récursif

```
pour u de 0 à 2^(N-1) − 1 faire
  pour v de 0 à 2^(N-1) − 1 faire
    pour n de 1 à N faire
      si u ≢ 2^(n-1) − 1 [2^(n-1)] ou v ≢ 2^(n-1) − 1 [2^(n-1)]
      alors sortir
      sinon
        '————————————— Transformation horizontale
        pour i de 0 à 2h − 1 faire
          {...}
        fin pour (i)
        '————————————— Transformation verticale
        pour j de 0 à 2w − 1 faire
          {...}
        fin pour (j)
      fin si
    fin pour (n)
  fin pour (v)
fin pour (u)
```

Pré-synthèse de composants combinatoires dédiés – La structure très régulière du *Lifting Scheme* nous permet de considérer différents niveaux de granularité des opérations [6,7]. La figure 5.12 présente trois possibilités de regroupement des opérations d'un sous-motif de transformation *1D*. Ce sous-motif est représenté en figure 5.12a à base d'opérations arithmétiques élémentaires (additions, multiplications).

Tout d'abord, il est possible de synthétiser chaque nœud *lifting* ou *dual lifting* sous la forme d'un composant combinatoire intégrant deux additionneurs et un multiplieur (figure 5.2 en page 142 et figure 5.12b). Cette possibilité permet notamment de prendre en compte dans le flot de synthèse de haut niveau les possibilités d'optimisation au niveau logique propres aux outils de synthèse *RTL* : là où un outil de synthèse de haut niveau cherchera à ordonnancer les trois opérations d'un nœud *lifting* – soit dans des cycles d'horloge séparés, soit en les chaînant à l'intérieur d'un même cycle – un outil de synthèse *RTL* produit un composant unique et optimisé dont les performances sont généralement meilleures que lorsque des opérateurs indépendants sont mis en cascade.

Un degré supplémentaire de regroupement est proposé en figure 5.12c, qui consiste à intégrer une chaîne de S nœuds *lifting* et *dual lifting* dans un même composant combinatoire. Ici, des optimisations logiques supplémentaires sont possibles du fait que tous les cœfficients de *lifting* sont câblés à l'intérieur de ce composant, ce qui permet de simplifier l'architecture des multiplieurs. Le tableau 5.7 montre que ces optimisations se traduisent par une réduction de 37% du nombre de portes, comparées à la mise en cascade de quatre composants *lifting* indépendants.

En figure 5.12d, les deux multiplieurs de *scaling* sont également insérés dans le composant combinatoire. Le tableau 5.8 indique le nombre d'opérations que l'outil de synthèse devra ordonnancer en fonction des regroupements effectués.

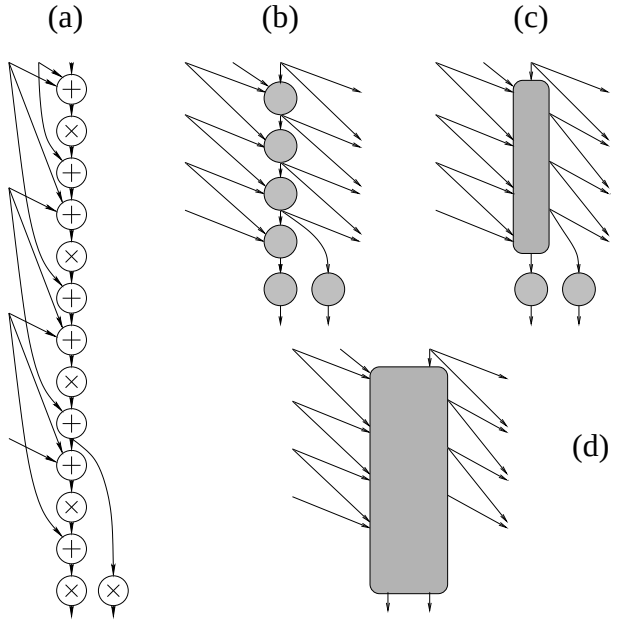
Dans une description comportementale, le mécanisme généralement utilisé pour permettre à un concepteur d'utiliser de tels composants dédiés consiste à inclure les opérations concernées dans un sous-programme et à indiquer, par le biais de directives destinées à l'outil de synthèse de haut niveau, que chaque appel à ce sous-programme doit être projeté sur un composant dédié présent dans une bibliothèque spécifique sous la forme d'une description *RTL* ou d'une *netlist* optimisée. Dans le contexte de la conception d'un composant virtuel, il est alors nécessaire que ladite bibliothèque figure parmi les délivrables du composant.

Comparaison des deux méthodes – Les deux approches présentées ont en commun de reposer sur l'extraction d'un sous-motif qui sera soit synthétisé séparément, soit inclus dans le corps d'une boucle non déroulée.

La principale différence réside dans la répartition des tâches de synthèse : dans la première méthode, nous réalisons manuellement une partie de l'ordonnancement des opérations, tâche normalement du ressort de l'outil de synthèse de haut niveau, et laissons à ce dernier le soin de synthétiser le sous-motif. Dans la deuxième méthode, nous avons au contraire spécifié et synthétisé le sous-motif avant de laisser l'outil de synthèse de haut niveau se charger de son ordonnancement.

La seconde méthode présente l'avantage sur la première de mieux répartir les tâches de synthèse en exploitant ce que chaque type d'outil sait le mieux faire : l'outil de synthèse *RTL* est particulièrement adapté à l'optimisation au niveau logique de

FIG. 5.12 – Utilisation de composants combinatoires dédiés pour la décomposition en ondelettes 9/7



TAB. 5.7 – Composants combinatoires dédiés et optimisations logiques

	Un composant "Lifting Step"	Quatre composants "Lifting Step" (non optimisés)	Quatre composants "Lifting Step" (optimisés)	Amélioration des performances
Temps de traversée (ns)	10,5	42	40	5%
Surface estimée (portes)	4030	16120	10200	37%

TAB. 5.8 – Nombre d'opérations à ordonnancer en fonction de la granularité des fonctions

Opérations	Quantité (cas général)	Cas $N = 3$, $S = 4$ $w = h = 1$	Total
Additions	$8S(4^N - 1)wh/3$	672	
Multiplications	$(4S + 8)(4^N - 1)wh/3$	504	1176
(Dual) Lifting	$4S(4^N - 1)wh/3$	336	
Mult. *scaling*	$8(4^N - 1)wh/3$	168	504
Chaînes *Lifting*	$4(4^N - 1)wh/3$	84	
Mult. *scaling*	$8(4^N - 1)wh/3$	168	252
Chaînes *DWT 1D*	$4(4^N - 1)wh/3$	84	84

fonctions arithmétiques simples tandis que les outils de synthèse de haut niveau savent gérer l'ordonnancement de fonctions complexes.

5.3.6 Contraintes d'entrée/sortie

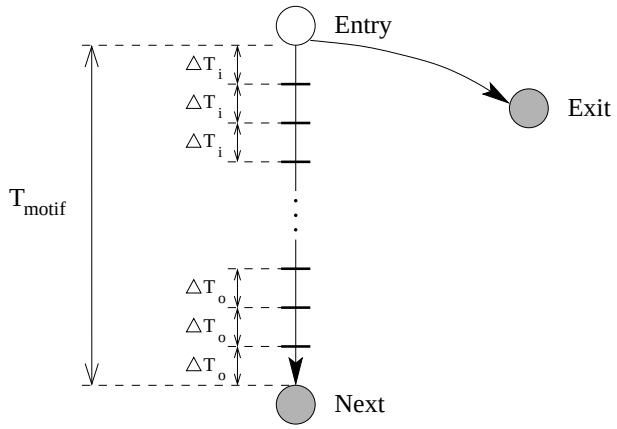
Selon le degré de parallélisme de calcul retenu, la quantité de données consommées et produites à chaque itération du motif de traitement pourra devenir conséquente. Il est nécessaire de fournir à l'intégrateur du composant virtuel un moyen de limiter le parallélisme d'accès aux données en fonction des contraintes d'implantation.

Jusqu'à présent, nous avons considéré que chaque itération du motif de calcul consomme et produit des blocs de pixels. Afin de limiter le parallélisme d'entrée/sortie, nous décomposons chaque bloc de pixels en *paquets* de taille fixe. Un paquet représente un ensemble de données lues ou écrites simultanément sur les ports du composant. Deux paquets distincts de données d'entrée (*resp.* de sortie) sont nécessairement lus (*resp.* écrits) sur des fronts d'horloge différents.

Cette décomposition nécessite la définition de plusieurs paramètres : (1) la quantité de données P_i et P_o contenues respectivement dans un paquets de pixels d'entrée ou de sortie ; (2) la répartition des pixels d'un bloc dans les différents paquets ; (3) la date de lecture/écriture des paquets à l'intérieur d'une itération du motif. Afin de ne pas alourdir le jeu de paramètres du composant, nous avons proposé une répartition des dates d'entrée/sortie fondée sur des intervalles de temps fixes ΔT_i et ΔT_o entre deux paquets successifs de pixels d'entrée et de sortie, respectivement. Le premier paquet de pixels d'entrée est lu sur le premier front d'horloge du motif et le dernier paquet de pixels de sortie est écrit sur le dernier front d'horloge (figure 5.13).

Contraintes d'entrée/sortie et contraintes de temps – La taille des paquets de données d'entrée/sortie est étroitement liée à la contrainte de temps fixée : le nombre de paquets constituant un bloc de pixels d'entrée (*resp.* de sortie) détermine le nombre minimum de périodes d'horloge nécessaires à la lecture (*resp.* l'écriture) du bloc. Ces durées incompressibles limitent la durée minimale autorisée pour l'exécution d'une itération du motif.

FIG. 5.13 – Insertion des contraintes d'entrée/sortie dans le motif de transformation en ondelettes 2D



La répartition des pixels d'entrée en paquets et l'ordre de présentation des paquets ont une influence sur la date d'exécution des opérations. Chaque nœud du motif ne peut en effet être exécuté que lorsque toutes ses données d'entrée sont disponibles. Dans le motif de transformation en ondelettes *2D* où tous les nœuds *lifting* d'entrée appartiennent à un chemin critique, le fait de retarder une donnée d'entrée entraîne la propagation de ce retard jusqu'aux nœuds relatifs au dernier niveau de décomposition.

Cette propagation des retards doit être prise en compte lors de la spécification des date de sortie des pixels transformés. Afin de garantir l'aptitude d'un outil de synthèse à ordonnancer le motif, il est ainsi nécessaire de prendre en compte les dépendances de données lors de la spécification des contraintes de temps et d'entrée/sortie.

Contraintes d'entrée/sortie et mémoire – Si les données en entrée d'un nœud *lifting* ou *dual lifting* du motif ne sont pas disponibles simultanément, cela signifie que certaines d'entre elles devront être mémorisées pendant une ou plusieurs périodes d'horloge. La propagation à travers le graphe des retards dus à l'ordonnancement des entrées va ainsi avoir des conséquences sur la quantité de mémoire nécessaire au stockage des données d'entrée et des données intermédiaires.

D'après [97], l'ordre de parcours le plus adapté à l'optimisation des échanges de données entre étages de transformation successifs est la courbe fractale de MORTON présentée en figure 4.13 en page 126. Ici, ce parcours n'est plus appliqué globalement à l'image entière, mais localement à l'intérieur de chaque bloc de pixels consommé par une itération du motif.

5.3.7 Résumé

Dans cette section, nous avons présenté les étapes successives de la conception d'un composant virtuel comportemental pour la transformation en ondelette *lifting* bidimensionnelle. A la différence de la description algorithmique qui nous a servi de point de départ (listing 5.1), la description comportementale obtenue intègre des

informations relatives à un ensemble de décisions d'implantation. Ces décisions portent sur : (1) le choix des formats de données ; (2) le choix du motif répétitif de calcul ; (3) le choix du déroulage des boucles ; (4) le choix de la granularité des opérateurs arithmétiques en bibliothèque ; (5) les contraintes de temps et d'entrée/sortie.

Le composant virtuel obtenu se caractérise par sa flexibilité, permettant à l'intégrateur d'adapter les propriétés fonctionnelles, temporelles et matérielles à ses besoins. Les paramètres supportés par le composant sont présentés dans le tableau 5.9.

En section 5.4, nous présentons des résultats de synthèse d'une variété de solutions architecturales obtenues en faisant varier certains de ces paramètres.

5.4 Synthèse du composant virtuel comportemental

5.4.1 Instanciation du composant virtuel

Nous nous plaçons dans un cas d'application typique utilisant l'ondelette 9/7 recommandée par le standard *JPEG2000* ($S = 4$) et réalisant la décomposition sur $N = 3$ niveaux. Dans les exemples de synthèse effectués, la largeur maximale d'une image est fixée à 512 pixels. La hauteur d'une image est illimitée.

Afin de limiter la complexité de la description, nous avons fixé la taille du motif à $L_{motif} = H_{motif} = 8$ ($w = h = 1$) et nous avons réduit la granularité des opérateurs en regroupant quatre nœuds *lifting* et *dual lifting* dans une même fonction que nous avons décrite au niveau transfert de registres et synthétisée séparément sous la forme d'un composant combinatoire. Les multiplications de *scaling* restent isolés afin de laisser à l'outil de synthèse une marge de manœuvre pour les ordonnancer.

Les pixels d'entrée sont représentés sur $N_i = 8$ bits non signés et les pixels de sortie sur $N_o = 16$ bits signés dont $F_o = 2$ bits de partie fractionnaire. Les résultats intermédiaires de calcul sont représentés sur $N_d = 21$ bits pour une dynamique maximale de $N_r = 32$ bits en sortie des opérateurs internes à un nœud *lifting*.

L'ordre de parcours des blocs de pixels dans l'image suit un schéma *raster*. La mémoire nécessaire au stockage des données *contexte horizontal* est implantée sous forme de registres dont l'utilisation est entièrement gérée par l'outil de synthèse de haut niveau. Les données *contexte vertical* sont mémorisées dans une *RAM* externe.

L'ordre d'entrée des pixels à l'intérieur d'un bloc de taille 8×8 suit la courbe fractale de MORTON afin de minimiser la durée de vie des données internes au motif. L'intervalle de temps entre deux paquets de pixels d'entrée/sortie est fixé à $\Delta T_i = \Delta T_o = 1$.

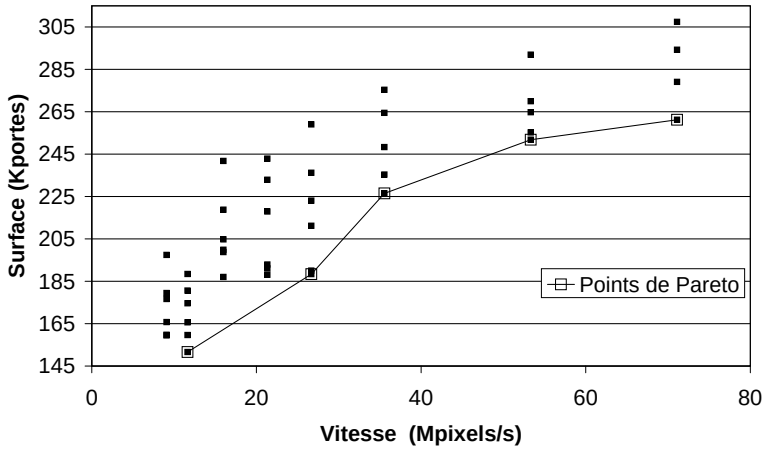
L'exploration architecturale a été réalisée en faisant varier la valeur des paramètres suivants :

- Parallélisme d'entrée/sortie : nous avons considéré un ensemble de cas dans lesquels le parallélisme d'accès aux données est le même en entrée et en sortie. Le nombre de bits de représentation des pixels transformés étant double de celui des pixels d'entrée, les solutions proposées définissent le nombre de pixels P_o par paquet de sortie comme égal à la moitié du nombre de pixels P_i par paquet d'entrée (à l'exception du cas $P_i = P_o = 64$ également traité).
  Les couples $(P_i; P_o)$ testés sont : $(64; 64)$, $(64; 32)$, $(32; 16)$, $(16; 8)$, $(8; 4)$ et $(4; 2)$.

TAB. 5.9 – Récapitulatif des paramètres primaires (1^{aire}), secondaires (2^{aire}) et dynamiques (Dyn) du composant virtuel DWT 2D

Notation	Type	Signification	Valeurs
W	1^{aire}	Ondelette	"9/7" ou "5/3"
S	2^{aire}	Nombre de pas de *lifting* ou *dual lifting*	2 ou 4
$\zeta(s)$	2^{aire}	Cœfficients de *lifting* et *dual lifting*	voir page 119
N	1^{aire}	Nombre de niveaux de décomposition	≥ 1
L_{max}	1^{aire}	Largeur maximale des images, en pixels	Multiple de 2^N
L, H	Dyn	Largeur, hauteur de l'image courante, en pixels	Multiples de 2^N $L \leq L_{max}$
C_i, N_i, F_i	1^{aire}	Format des pixels de l'image source	C_i = "signé" ou "non signé" $0 \leq F_i < N_i$
N_o, F_o	1^{aire}	Format des pixels de l'image transformée	$0 \leq F_o < N_o$
N_d, N_m	1^{aire}	Format des données intermédiaires	$0 < N_d < N_m$
S_d, S_c, S_r	2^{aire}	Décalages de bits après opérations arithmétiques	–
L_{motif}, H_{motif}	1^{aire}	Taille du motif de transformation	Multiples de 2^N
T_{motif}	1^{aire}	Nombre de périodes d'horloge par itération du motif	$> (4^N - 1)/3$
P_i, P_o	1^{aire}	Nombre de pixels par paquet en entrée/sortie du motif	Sous-multiples de $L_{motif} \times H_{motif}$
ΔT_i, ΔT_o	1^{aire}	Nombre de périodes d'horloge entre deux paquets successifs	$\Delta T_i / P_i \leq T_{motif} / (L_{motif} H_{motif})$ $\Delta T_o / P_o \leq T_{motif} / (L_{motif} H_{motif})$

FIG. 5.14 – Espace des solutions architecturales sous contraintes de vitesse de traitement et de surface



– Cadence de traitement : la durée d'exécution d'une itération du motif varie de 18 à 141 cycles, soit une cadence moyenne de traitement allant de 3,5 à 0,45 pixels par cycle.

5.4.2 Résultats de synthèse

La synthèse a été réalisée à l'aide de l'outil *Monet* de *Mentor Graphics* (version *v8.7_2.1, release R44*) pour la technologie *AMS* $0,35\mu m$. La période d'horloge utilisée est de $50ns$, ce qui correspond à la latence d'un composant *chaîne lifting* (estimée à $40ns$ après synthèse du composant combinatoire sous *Design Compiler* de *Synopsys*) augmentée de $10ns$ pour la prise en compte du délai des couches combinatoires et des interconnexions entre opérateurs et registres du chemin de données.

Notre description comportementale étant exacte au cycle près, l'ordonnancement sous *Monet* a été effectué en mode *cycle-fixed*.

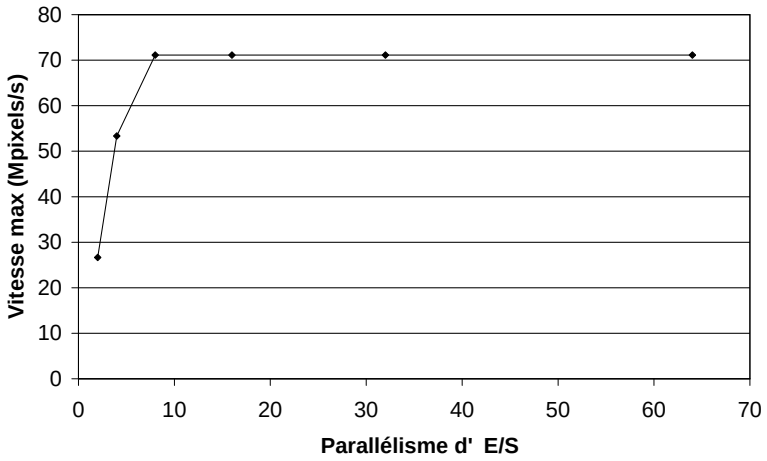
Espace des solution architecturales – Les paramètres présentés ci-dessus nous ont permis d'obtenir 48 solutions architecturales présentant différentes performances vitesse/surface. Ces espace de solutions est représenté en figure 5.14.

Pour une période d'horloge de $50ns$, les cadences de traitement varient de 9 Mpixels/s (0,45 pixels/cycle) à 71 Mpixels/s (3,5 pixels/cycle). La surface de l'architecture varie de 151000 à 307000 portes.

La courbe représentée en figure 5.14 contient les *points de Pareto* de l'espace des solutions proposé. Ces points correspondent aux solutions réalisant la meilleure performance en vitesse pour une surface donnée, et présentant la plus petite surface pour une vitesse donnée.

Les cinq points obtenus correspondent aux cinq solutions *pertinentes* si les seuls critères de sélection sont la vitesse et la surface. La moins coûteuse en surface (151

FIG. 5.15 – Vitesse de traitement maximale autorisée en fonction du parallélisme d'entrée/sortie



Kportes) correspond à une vitesse de 11,6 Mpixels/s tandis que la plus rapide (71 Mpixels/s) présente une surface de 261 Kportes.

Parallélisme d'entrée/sortie et vitesse de traitement – Le parallélisme d'entrée/sortie est exprimé comme le nombre P_o de pixels produits simultanément sur les ports de sortie du composant. Le nombre de données d'entrée P_i est généralement égal au double de P_o, à l'exception du cas $P_o = 64$.

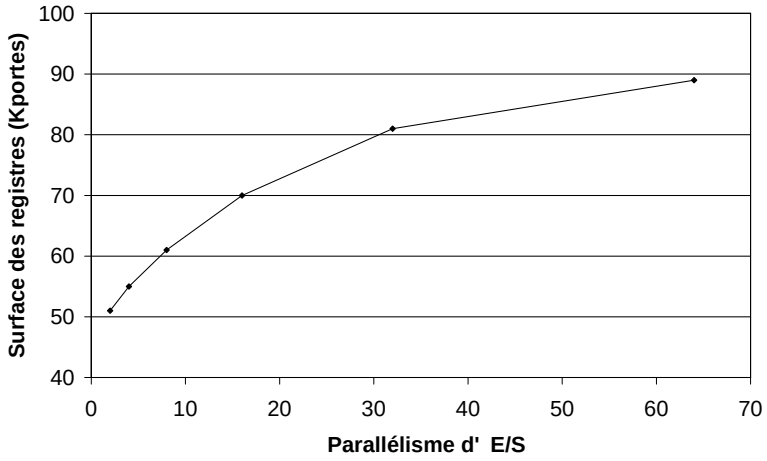
La figure 5.15 indique la vitesse de traitement maximale pouvant être atteinte par le composant en fonction du parallélisme d'entrée/sortie. Pour $P_o < 8$ ($P_i < 16$), nous observons que le nombre de cycles nécessaires à la lecture et à l'écriture des pixels d'entrée/sortie limite la vitesse de traitement. A partir de $P_o = 8$, la vitesse n'est plus limitée que par la longueur du chemin critique du motif.

Parallélisme d'entrée/sortie et allocation des registres – La figure 5.16 donne la surface de registres pour la mémorisation des données d'entrée/sortie, des données *contexte horizontal* et des données intermédiaires du motif.

Une partie de cette surface est occupée par les P_o registres placés systématiquement par l'outil de synthèse sur les P_o ports de sortie. Une autre partie de cette surface concerne les 56 registres de mémorisation du contexte horizontal, dont la quantité ne dépend pas du parallélisme d'entrée/sortie.

Comme nous l'avons souligné plus haut, un fort parallélisme d'entrée/sortie se traduit également par une quantité importante de données arrivant trop tôt dans le composant – c'est-à-dire à un instant où les opérateurs qui doivent les traiter ne sont pas disponibles – ou sortant trop tard, c'est-à-dire un ou plusieurs cycles après avoir été calculées. Limiter la taille des paquets de données permet à l'outil de synthèse d'ordonnancer les opérations au plus près des entrées/sorties qu'elles consomment ou produisent, dans la mesure où l'ordre de parcours des pixels a été bien choisi.

FIG. 5.16 – Surface de registres en fonction du parallélisme d'entrée/sortie



Vitesse de traitement et allocation des ressources de calcul – Le nombre d'opérateurs alloués est quasiment indépendant du parallélisme d'entrée/sortie. En revanche, la contrainte de cadence de traitement a une forte influence sur le parallélisme de calcul à mettre en œuvre. La figure 5.17 donne l'évolution du nombre d'opérateurs *chaîne lifting* et du nombre de multiplieurs de *scaling* en fonction de la vitesse. Nous nous plaçons ici dans le cas d'un parallélisme d'entrée/sortie maximal ($P_i = P_o = 64$) où l'ordonnancement des opérations n'est pas contraint par les entrées/sorties.

A la cadence de traitement la plus lente, nous observons que le nombre de composants *chaîne lifting* se réduit à 1, pour un nombre de multiplieurs égal à 3. Ces quantités atteignent respectivement 8 et 16 à la vitesse maximale. Dans cette dernière situation, ils représentent à eux seuls 48% de la surface totale de l'architecture.

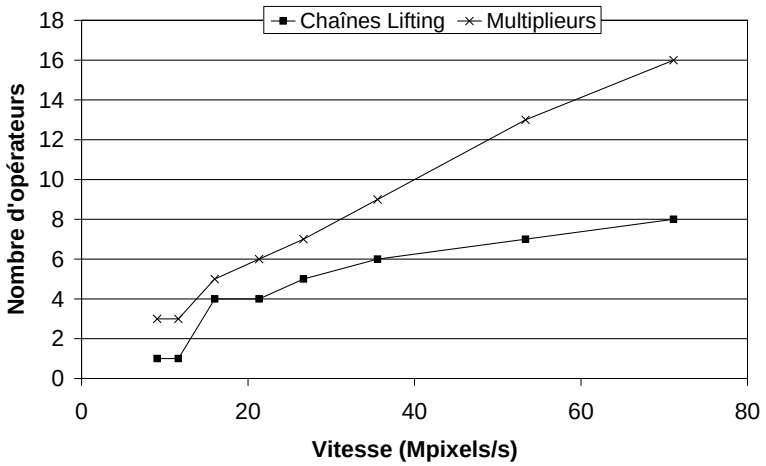
5.4.3 Discussion

Nous avons montré que l'utilisation de la synthèse de haut niveau autorise la génération d'une variété d'architectures *RTL* à partir d'une unique description comportementale paramétrée. L'éventail de paramètres disponibles et de performances pouvant être atteintes autorise la réutilisation de ce composant dans des applications soumises à des contraintes diverses.

A titre de comparaison des performances obtenues, une architecture de transformation *2D* de type *folded* utilisant des bancs de filtres *1D* de la forme décrite dans [101] nécessiterait au moins 12 bancs de filtres afin de réaliser en 18 cycles les 168 opérations de filtrage nécessaires à la transformation d'un bloc de 8×8 pixels sur trois niveaux. La surface estimée d'un tel banc de filtre est de 14000 portes [101], d'où une surface totale de 168000 portes pour les 12 bancs de filtres.

A ces 12 bancs de filtres, il convient d'adjoindre un réseau d'interconnexion et de mémorisation des données consommées et produites à chaque étape de la transformation. Les ressources de mémorisation couvrent une surface significative : dans les solutions que nous avons obtenues cette surface représente 50000 à 90000 portes.

FIG. 5.17 – Nombre d'opérateurs arithmétiques en fonction de la vitesse de traitement



La surface de l'architecture *folded* complète atteint ainsi aisément 200000 à 250000 portes, c'est-à-dire une surface du même ordre de grandeur que les solutions obtenues en synthèse de haut niveau.

Par ailleurs, notons que la majorité des architectures recensées dans la littérature ne traitent pas le problème de l'extension de l'image par symétrie périodique au voisinage des bords. Comme le montre le listing 5.1 en page 139, l'implantation d'un tel mécanisme dans une description destinée à la synthèse de haut niveau est très naturelle et l'effort de conception supplémentaire est négligeable.

Les résultats que nous présentons sont cependant loin d'épuiser les possibilités qu'offrent les paramètres supportés par notre composant virtuel de transformation en ondelettes *2D*.

Notamment, il aurait été intéressant de faire varier la taille du motif en même temps que la contrainte de temps afin de tester un jeu de solutions à parallélisme moyen constant. Nous nous heurtons ici aux limitations, en termes de complexité calculatoire des descriptions, de l'outil de synthèse de haut niveau *Monet*, limitations que nous croyons essentiellement dues au manque de maturité de l'outil.

Conclusion et Perspectives

Nous avons proposé dans ce travail de thèse une démarche méthodologique permettant d'insérer les outils de synthèse de haut niveau dans un flot de réutilisation de composants virtuels. Cette démarche définit la notion de *composant virtuel comportemental* comme un niveau d'abstraction supplémentaire – venant s'ajouter aux niveaux *hard*, *firm* et *soft* définis par l'organisation *VSIA* – à partir duquel un composant *IP* peut être délivré et synthétisé. Elle s'appuie sur des modèles formels contraignant la syntaxe et la sémantique des descriptions comportementales synthétisables afin d'assurer la *flexibilité*, la *prédictibilité* et la *portabilité* d'un composant virtuel décrit à ce niveau d'abstraction.

Nous avons tout d'abord défini les étapes du flot de conception d'un composant virtuel comportemental, du niveau fonctionnel vers le niveau comportemental avec entrées/sorties au cycle près. Face à la diversité des outils de synthèse de haut niveau, des langages qu'ils supportent et des modèles de représentation intermédiaires sur lesquels ils font reposer leur flot de synthèse, nous avons montré qu'il est possible de généraliser la notion de description comportementale en formalisant un outil de synthèse de haut niveau *universel* (*U-HLS*) reposant sur un modèle syntaxique et sémantique indépendant des outils existants.

La méthodologie et le flot de conception proposés établissent un lien fort entre les nouvelles approches de conception au niveau système et la synthèse de haut niveau. Ils permettent ainsi d'insérer les outils de synthèse de haut niveau dans un flot cohérent de *synthèse système* à base de composants réutilisables. Du fait du haut degré de flexibilité d'un composant virtuel comportemental, l'utilisateur d'un tel composant ne se voit plus délivrer une unique architecture, mais dispose au contraire d'un ensemble de solutions qu'il lui sera loisible d'explorer au moyen du jeu de paramètres associé au composant.

Cette méthodologie a pu être expérimentée sur deux applications. La première est l'application test retenue dans le cadre du projet *MILPAT*, consistant à réaliser un composant de calcul de distance de *Manhattan* pour l'estimation de mouvement par *block-matching* dans une chaîne de compression vidéo de type *MPEG4*. Cette application nous a permis de valider notre modèle de représentation *HSFSMD* pour la modélisation de comportements intégrant une partie contrôle à base de boucles et d'instructions de branchement. A partir d'une même modélisation et de contraintes de temps identiques, l'utilisation de ce modèle nous a permis de produire par un procédé de traduction systématique deux descriptions *VHDL* différentes adaptées aux outils de synthèse *Monet* et *Behavioral Compiler* et garantissant des performances temporelles identiques après synthèse.

Notre seconde application, développée de manière plus conséquente dans la deuxième partie de ce mémoire, a consisté à réaliser un composant de transformation en ondelettes discrète compatible avec la norme *JPEG2000* pour le codage des images fixes. Cette étude a été conduite en collaboration avec le *CNES* et la société *Astrium* dans la perspective d'applications d'imagerie spatiale. L'algorithme fondé sur le *lifting scheme* nous a permis de mettre en œuvre une exploration systématique des possibilités de répartition des calculs et des entrées/sorties au cours du temps, au moyen d'une modélisation à base de graphes de dépendances. Les résultats de synthèse de haut niveau présentés montrent le haut degré de flexibilité autorisé par la démarche de conception suivie.

La variété de solutions architecturales obtenues pour une application complexe comme la transformation en ondelettes bidimensionnelle illustre l'aptitude des outils de synthèse de haut niveau à exploiter des choix d'implantation originaux auxquels un concepteur humain ne se risquerait pas nécessairement.

Malgré le manque de maturité des outils commerciaux existants – qui d'une part sont pour la plupart encore dans leur phase de développement et dont la maîtrise exige d'autre part un savoir-faire significatif de la part du concepteur – les techniques de synthèse de haut niveau devraient jouer un rôle de premier plan dans les futures méthodologies de synthèse système.

Dans le prolongement des approches actuelles consistant à élever le niveau d'abstraction de la spécification d'un système et à réutiliser des composants virtuels préconçus et pré-vérifiés, la nécessité toujours plus pressante d'accélérer et de fiabiliser le flot de conception d'un système encourage en effet une automatisation de l'exploration architecturale, que ce soit au niveau plate-forme, par la sélection et l'instanciation automatiques de blocs *IP*, ou bien au niveau micro-architectural en automatisant la génération d'une architecture *RTL* à partir de la spécification d'un composant à un haut niveau d'abstraction.

Perspectives

Dans la méthodologie présentée, nous avons laissé ouverts deux problèmes importants dans le flot de réutilisation d'un composant virtuel comportemental. Le premier est celui de la protection de la propriété intellectuelle, pour lequel nous avons ébauché une piste de réflexion dans le bilan de la première partie de ce mémoire. Le second est celui de la synthèse des interfaces de communication entre le composant décrit au cycle près et son environnement – vu au niveau système à travers des interfaces *SDF* – et de la synthèse des mémoires.

Concernant le second point, il est possible de considérer que les interfaces de communication sont raffinées manuellement sous forme d'adaptateurs (*wrappers*) par l'utilisateur du composant virtuel en fonction du protocole de communication qu'il souhaite implanter. Cette solution peut imposer un effort de conception significatif à l'utilisateur et peut également altérer les performances du composant. Une meilleure solution serait de prendre en compte la synthèse des interfaces de communication dans le flot de synthèse de haut niveau proprement dit. Cependant, les outils de synthèse de haut niveau du commerce ne supportent pas encore une telle fonctionnalité.

L'automatisation de la génération d'une hiérarchie mémoire en fonction de la durée de vie des variables manipulées par un motif de calcul s'avère également problématique en regard des possibilités offertes par les outils de synthèse de haut niveau du commerce. La stratégie de contournement que nous avons retenue consiste à considérer les transferts de données à durée de vie longue comme des entrées/sorties du composant et à externaliser la gestion du stockage des données dans une unité de mémorisation à synthétiser séparément.

Face aux limitations évoquées, une évolution des outils de synthèse de haut niveau vers la synthèse des communications et la synthèse mémoire apparaît nécessaire à leur insertion efficace dans un flot complet de synthèse système. Les travaux en cours au *LESTER* consistent à intégrer de nouvelles fonctionnalités dans l'outil *GAUT* [61, 116] afin de traiter conjointement la synthèse de l'unité de traitement, de l'unité de mémorisation et de l'interface de communication d'un composant décrit au niveau comportemental.

Bibliographie

Bibliographie Personnelle

[1] S. PILLEMENT, O. SENTIEYS, D. CHILLET, E. CASSEAU, P. COUSSY, E. MARTIN, G. SAVATON, et S. ROUX. Design and Synthesis of Behavioral Level Virtual Components. Dans *Proc. International Conference on Very Large Scale Integration (VLSI-SoC)*, Décembre 2001.

[2] G. SAVATON, E. CASSEAU, et E. MARTIN. Behavioral VHDL Styles and High-Level Synthesis for IPs. Dans *Proc. Forum on Design Languages (FDL)*, Septembre 2000.

[3] G. SAVATON, E. CASSEAU, et E. MARTIN. High-Level Design and Synthesis of a Discrete Wavelet Transform Virtual Component for Image Compression. Dans *Proc. International Workshop on IP-based Synthesis and SoC design*, Décembre 2000.

[4] G. SAVATON, E. CASSEAU, et E. MARTIN. High-Level Synthesis and Behavioral VHDL Writing Style : Towards a Methodology for Behavioral IP Reuse. Dans *Proc. 13th IEEE ASIC/SoC Conference*, Septembre 2000.

[5] G. SAVATON, E. CASSEAU, et E. MARTIN. Design of a Flexible 2-D Discrete Wavelet Transform IP Core for JPEG2000 Image Coding in Embedded Imaging Systems. *Soumis à la revue Signal Processing*, 2002.

[6] G. SAVATON, E. CASSEAU, E. MARTIN, et C. LAMBERT-NEBOUT. Composants Virtuels Comportementaux pour Applications de Compression d'Images. Dans *Actes GRETSI*, Septembre 2001.

[7] G. SAVATON, E. CASSEAU, E. MARTIN, et C. LAMBERT-NEBOUT. Behavioral Virtual Components for Embedded Image Compression Systems. Dans *Proc. 17th Conference on Digital Circuits and Integrated Systems (DCIS)*, Novembre 2002.

[8] G. SAVATON, P. COUSSY, E. CASSEAU, et E. MARTIN. A Methodology for Behavioral Virtual Component Specification Targeting SoC Design with High-Level Synthesis Tools. Dans *Proc. Forum on Design Languages (FDL)*, Septembre 2001.

Conception de Systèmes Embarqués

[9] H. CHANG et al. *Surviving the SoC Revolution – A Practical Guide to Platform-Based Design.* Kluwer Academic Publishers, Novembre 1999.

[10] Mitchell DALE. Verification Crisis : Managing Complexity in SoC Design. *EE-Design*, Février 2001.

[11] Rolf ERNST. Codesign of Embedded Systems : Status and Trends. *IEEE Design and Test of Computers*, Avril 1998.

[12] D. D. GAJSKI et R. H. KUHN. Guest Editors Introduction - New VLSI Tools. *IEEE Computer*, 16(2) :11–14, 1983.

[13] D. D. GAJSKI, F. VAHID, S. NARAYAN, et J. GONG. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[14] C. HEIN, T. CARPENTER, P. KALUTKIEWICZ, et V. MADISETTI. RASSP VHDL Modeling Terminology and Taxonomy. Dans *Proc. 2nd Annual RASSP Conference*, Mai 2000. `http://www.eda.org/rassp`.

[15] K. KEUTZER, S. MALIK, A. R. NEWTON, J. RABAEY, et A. SANGIOVANNI-VINCENTELLI. System-Level Design : Orthogonalization of Concerns and Platform-Based Design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), Décembre 2000.

[16] VSI ALLIANCE/SLD DWG. Model Taxonomy – Version 2.1. Rapport technique, Juillet 2001.

Technologie des Circuits Intégrés

[17] P. P. GELSINGER et al. Microprocessors circa 2000. *IEEE Spectrum*, Octobre 1989.

[18] ITRS. International Technology Roadmap for Semiconductors. Rapport technique, 2001.

[19] Gordon E. MOORE. Cramming More Components onto Integrated Circuits. *Electronics*, Avril 1965.

Modèles Formels pour la Conception de Systèmes Embarqués

[20] A. BENVENISTE et G. BERRY. The Synchronous Approach to Reactive and Real Time Systems. *Proc. IEEE*, 79(9) :1270–1282, Septembre 1991.

[21] J. R. BURCH, R. PASSERONE, et A. L. SANGIOVANNI-VINCENTELLI. Modeling Techniques in Design-by-Refinement Methodologies. Dans *Proc. Integrated Design and Process Technology*, Juin 2002.

[22] M. CHIODO et al. A Formal Specification Model for Hardware/Software Codesign. Rapport technique, Dept. EECS, UC Brekeley, 1993. UCB/ERL M93/48.

[23] L. A. CORTÉS, P. ELES, et Z. PENG. SAVE Project Report – A Survey on Hardware/Software Codesign Representation Models. Rapport technique, Dept. of Computer and Information Science, Linköping University, Juin 1999.

[24] F. FRANSSEN, F. BALASA, M. van SWAAIJ, F. CATTHOOR, et H. De MAN. Modeling Multidimensional Data and Control Flow. *IEEE Transactions on VLSi Systems*, 1(3), Septembre 1993.

[25] D. HAREL. StateCharts : A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8 :231–274, Juin 1987.

[26] C. A. R. HOARE. *Communicating Sequential Processes.* Prentice Hall, 1985.

[27] E. A. LEE, S. NEUENDORFFER, et M. J. WIRTHLIN. Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits, Systems, and Computers*, Juillet 2002.

[28] E. A. LEE et T. PARKS. Dataflow Process Networks. *Proc. IEEE*, 83 :773–799, 1995.

[29] H. P. PEIXOTO et M. F. JACOME. Algorithm and Architecture-Level Design Space Exploration using Hierarchical Data Flows. Dans *Proc. 11th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 271–282, Juillet 1997.

Langages de Conception

[30] ACCELLERA/ALC/C C++ WORKING GROUP. RTL Semantics. Rapport technique, Février 2001. Draft specification 0.8.

[31] P. ALEXANDER, D. BARTON, et R. KARNATH. System Specification in Rosetta. Dans *Proc. IEEE Engineering of Computer Based Systems Symposium (ECBS)*, Avril 2000.

[32] G. BERRY. The Foundations of Esterel. Rapport technique, Ecole des Mines Paris/INRIA, 2000.

[33] C. EISNER et D. FISMAN. Sugar 2.0 – Proposal Presented to the Accellera Formal Verification Committee. Rapport technique, IBM Haifa Research laboratory, Mars 2002.

[34] P. L. FLAKE et S. J. DAVIDMANN. Evolving the Next Design Language. Dans *Proc. Forum on Design Languages (FDL)*, Septembre 2000.

[35] D. D. GAJSKI et al. *SpecC : Specification Language and Methodology.* Kluwer Academic Publishers, Mars 2000.

[36] D. D. GAJSKI et al. *System Design – A Practical Guide with SpecC.* Kluwer Academic Publishers, Mai 2001.

[37] J. GERLACH et W. ROSENSTIEL. System Level Design Using the SystemC Modeling Platform. Dans *Proc. 3rd Workshop on System Design Automation (SDA)*, Mars 2000.

[38] S. HOLLOWAY, D. LONG, et A. FITCH. From Algorithm to SoC with SystemC and CoCentric System Studio. Dans *Proc. Synopsys Users Group (SNUG)*, Mars 2002.

[39] IEEE/DASC/STD LOGIC 1164. Standard Multivalue Logic System for VHDL Model Interoperability. Rapport technique, 1993. IEEE P1164.

[40] IEEE/DASC/VASG. Draft IEEE Standard VHDL Language Reference Manual. Rapport technique, 2000. IEEE P1076.2000/D3.

[41] IEEE/DASC/VASG/TIMING WORKING GROUP. Draft Standard VITAL ASIC Modeling Specification. Rapport technique, Avril 1999. IEEE P1076.4.

[42] IEEE/DASC/VASG/VHDL Synthesis Interoperability Working Group. Draft Standard for VHDL Register Transfer Level Synthesis. Rapport technique, Mars 1998. IEEE P1076.6/D1.12.

[43] IEEE/DASC/Verilog Synthesis Subset Working Group. IEEE Standard P1364.1. `http://www.eda.org/vlog-synth/`.

[44] H. Keding. Transaction-Level Modeling (TLM) with SystemC. Dans *Proc. Synopsys Users Group (SNUG)*, Mars 2002.

[45] S. Narayan et D. Gajski. Features Supporting System-Level Specification in HDLs. Dans *Proc. EuroDAC/EuroVHDL*, pages 540–545, 1993.

[46] Open SystemC Initiative (OSCI). Functional Specification for SystemC 2.0. Rapport technique, Octobre 2001.

[47] Open SystemC Initiative (OSCI). SystemC Version 2.0 User's Guide. Rapport technique, 2001.

[48] V. Rajaraman, N. Vydianathan, et K. Hajikhani. A Scalable Verification Methodology using VERA. Dans *Synopsys Users' Group (SNUG)*, Mars 2002.

[49] D. Verkest, J. Kunkel, et F. Schirrmeister. System-Level Design using C++. Dans *Proc. Design Automation and Test in Europe (DATE)*, 2000.

[50] D. K. Wilde. The Alpha Language. Rapport technique, IRISA, Mai 1994. Publication interne 827.

[51] D. K. Wilde et Oumarou Sié. Regular Array Synthesis using Alpha. Rapport technique, IRISA, Mai 1994. Publication interne 829.

Synthèse de Haut Niveau

[52] W. O. Cesário, Z. Sugar, I. Moussa, et A. A. Jerraya. Efficient Integration of Behavioral Synthesis within Existing Design Flows. Dans *Proc. 13th International Symposium on System Synthesis (ISSS)*, pages 85–90, Septembre 2000.

[53] J. P. Elliott. *Understanding Behavioral Synthesis. A Practical Guide to High-Level Design.* Kluwer Academic Publishers, 2000.

[54] N. Fan, V. Chaiyakul, et D. Gajski. Usage-Based Characterization of Complex Functional Blocks for Reuse in Behavioral Synthesis. Dans *Proc. Asian and South Pacific Design Automation Conference (ASPDAC)*, Janvier 2000.

[55] P. Feautrier. Array Expansion. Dans *Proc. Second International Conference on Supercomputing*, 1988.

[56] D. Gajski et al. *High-Level Synthesis : Introduction to Chip and System Design.* Kluwer Academic Publishers, 1991.

[57] D. W. Knapp. *Behavioral Synthesis. Digital System Design Using the Synopsys Behavioral Compiler.* Prentice Hall, 1996.

[58] H. Lehr et D. D. Gajski. Modeling Custom Hardware in VHDL. Rapport technique, CECS, UC Irvine, Juillet 1999. ICS-TR-99-29.

[59] H. De Man, J. Rabaey, P. Six, et L. Claesen. Cathedral-II : A silicon compiler for digital signal processing. *IEEE Des. Test*, 3(6) :13–125, Décembre 1986.

[60] E. MARTIN, C. NOUET, et J.-M. TOUREILLES. Conception Optimisée d'Architectures en Précision Finie pour les Applications de Traitement du Signal. *Traitement du Signal*, 18(1), 2001.

[61] E. MARTIN, O. SENTIEYS, H. DUBOIS, et J.-L. PHILIPPE. GAUT : An Architectural Synthesis Tool for Dedicated Signal Processors. Dans *Proc. European Design Automation Conference (EuroDAC)*, pages 14–19, Septembre 1993.

[62] MENTOR GRAPHICS. Monet User's Reference Manual – Software Release R44. Rapport technique, Septembre 2000.

[63] I. MOUSSA et al. Comparing RTL and Behavioral Design Methodologies in the Case of a 2M Transistors ATM Shaper. Dans *Proc. Design Automation Conference (DAC)*, pages 598–603, Juin 1999.

[64] SYNOPSYS. Behavioral Compiler, User Guide – Version 1998.02. Rapport technique, 1998.

[65] A. TAKACH, P. GUTBERLET, et S. WATERS. Hardware Design using Algorithmic C. Dans *Proc. Forum on Design Languages (FDL)*, Septembre 2001.

Composants Virtuels

[66] P. COUSSY, A. BAGANNE, et E.MARTIN. A Design Methodology for IP Integration. Dans *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, Mai 2002.

[67] P. COUSSY, A. BAGANNE, et E.MARTIN. Analyse Fonctionnelle des Moyens de communication Proposés dans les Systèmes sur Silicium. Dans *Actes Journées Francophones de l'Adéquation Algorithme Architecture (JFAAA)*, Novembre 2002.

[68] G. CYR, G. BOIS, M. ABOULHAMID, et J. BAILLAIRGÉ. Synthesis of Communication Interfaces Using VSIA Recommendations. Dans *Proc. Design Automation and Test in Europe (DATE)*, 2001.

[69] D. D. GAJSKI et al. Essential Issues for IP Reuse. Dans *Proc. Asia and South Pacific Design Automation Conference (ASPDAC) – Embedded Tutorial*, pages 37–42, 2000.

[70] I. HONG et M. POTKONJAK. Behavioral Synthesis Techniques for Intellectual Property Protection. Dans *Proc. 36th ACM/IEEE Design Automation Conference*, pages 849–854, Juin 1999.

[71] M. KEATING et P. BRICAUD. *Reuse Methodology Manual for System-on-a-Chip Design*. Kluwer Academic Publishers, 1999.

[72] J. LACH, W. H. MANGIONE-SMITH, et M. POTKONJAK. Fingerprinting Techniques for Field-Programmable Gate Array Intellectual Property Protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(10) :1253–1261, Octobre 2001.

[73] A. MELIKIAN, D. ALTAS, et G. FAYAD. A High Level Synthesis Approach to Soft IP Reuse. Dans *Proc. Synopsys Users Group Forum (SNUG)*, 1999.

[74] VSI ALLIANCE. Architecture Document – Version 1.0. Rapport technique, 1997.

[75] VSI ALLIANCE. Deliverables Document – Version 2.3. Rapport technique, Octobre 2000.

[76] VSI ALLIANCE/IPP DWG. Intellectual Property Protection : Schemes, Alternatives and Discussion – Version 1.1. Rapport technique, Janvier 2001.

Projet MILPAT

[77] R. AIRIAU, A. CARER, E. CASSEAU, E. MARTIN, et O. SENTIEYS. Méthodologie de Conception de Composants Virtuels pour les Applications de TDSI. Dans *Actes Conférence Adéquation Algorithme Architecture (AAA)*, Janvier 2000.

[78] LESTER, LASTI, et FRANCE TELECOM R&D. Projet MILPAT – Rapport d'Avancement 1.1 – Niveau Comportemental : Outils HLS et IPs. Rapport technique, Réseau National de Recherche en Télécommunications (RNRT), 2000.

[79] LESTER, LASTI, et FRANCE TELECOM R&D. Projet MILPAT – Rapport d'Avancement 1.2 – Outils HLS Commerciaux et leurs Limitations. Rapport technique, Réseau National de Recherche en Télécommunications (RNRT), 2000.

[80] LESTER, LASTI, et FRANCE TELECOM R&D. Projet MILPAT – Rapport d'Avancement 1.3 – Définition de la sémantique d'entrée et du modèle U-HLS. Rapport technique, Réseau National de Recherche en Télécommunications (RNRT), 2000.

[81] LESTER, LASTI, et FRANCE TELECOM R&D. Projet MILPAT – Rapport d'Avancement 1.4 – Migration U-HLS vers HLS Commerciaux. Rapport technique, Réseau National de Recherche en Télécommunications (RNRT), 2000.

[82] LESTER, LASTI, et FRANCE TELECOM R&D. Projet MILPAT – Rapport d'Avancement 2.1 – Méthode de Spécification des Applications. Rapport technique, Réseau National de Recherche en Télécommunications (RNRT), 2000.

[83] LESTER, LASTI, et FRANCE TELECOM R&D. Projet MILPAT – Rapport d'Avancement 3.2 – Spécification et Expérimentation d'une Application Industrielle à base d'IPs. Rapport technique, Réseau National de Recherche en Télécommunications (RNRT), 2000.

Compression d'Image et Ondelettes

[84] M. D. ADAMS et R. WARD. Wavelet Transforms in the JPEG-2000 standard. Dans *Proc. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, volume 1, pages 160–163, Août 2001.

[85] K. ANDRA, C. CHAKRABARTI, et T. ACHARYA. A VLSI Architecture for Lifting-Based Wavelet Transform. Dans *Proc. IEEE Workshop on Signal Processing Systems (SiPS) Design and Implementation*, pages 70–79, Octobre 2000.

[86] M. ANTONINI, M. BARLAUD, P. MATHIEU, et I. DAUBECHIES. Image Coding using Wavelet Transform. *IEEE Transactions on Image Processing*, 1(2) :205–220, Avril 1992.

[87] C. CHAKRABARTI et C. MUMFORD. Efficient realizations of encoders and decoders based on the 2-D Discrete Wavelet Transform. Dans *International Conference on Supercomputing*, 1998.

[88] C. CHAKRABARTI, M. VISHWANATH, et R. M. OWENS. Architectures for Wavelet Transforms : A Survey. *Journal of VLSI Signal Processing, Image and Video Technology*, 14(2) :171–192, Novembre 1996.

[89] A. COHEN, I. DAUBECHIES, et J. FEAUVEAU. Bi-orthogonal Bases of Compactly Supported Wavelets. *Communications on Pure and Applied Mathematics*, 45 :485–560, 1992.

[90] I. DAUBECHIES et W. SWELDENS. Factoring Wavelet Transforms into Lifting Steps. *Journal of Fourier Analysis and Applications*, 4(3) :247–269, 1998.

[91] C. DIOU, L. TORRES, et M. ROBERT. VLSI implementation of Lifting Scheme Wavelet Transform. Dans *Proc. Design Automation and Test in Europe (DATE) conference, Designers' forum*, pages 67–72, Mars 2001.

[92] C. GASQUET et P. WITOMSKI. *Analyse de Fourier et Applications*. Masson, 1995.

[93] Jean-Paul GUILLOIS. *Techniques de Compression des Images*. Editions Hermès, collection Informatique, Juin 1996.

[94] ISO/IEC JTC1/SC29. Information technology – Digital compression and coding of continuous-tone still images : Requirements and guidelines. Rapport technique 10918-1 :1994, 1994.

[95] ISO/IEC JTC1/SC29. Information technology – JPEG 2000 Requirements and Profiles Version 6.3. Rapport technique, Juillet 2000.

[96] ISO/IEC JTC1/SC29. Information technology – JPEG 2000 Part I Final Committee Draft Version 1.0. Rapport technique FCD15444-1, Mars 2001.

[97] G. LAFRUIT, F. CATTHOOR, J. CORNELIS, et H. De MAN. An Efficient VLSI architecture for 2-D Wavelet Image Coding with Novel Image Scan. *IEEE Transactions on Very Large Scale Integration Systems*, 7(1) :56–68, Mars 1999.

[98] C. LAMBERT-NEBOUT, G. MOURY, et J.-E. BLAMONT. Status of Onboard Image Compression for CNES Space Missions. Dans *Proc. of SPIE'99*, volume 3808, pages 242–256, Octobre 1999.

[99] Ph. LIER, G. MOURY, C. LATRY, et F. CABOT. Selection of the SPOT5 Image Compression Algorithm. Dans *Proc. of SPIE'98*, volume 3439, pages 541–552, Octobre 1998.

[100] S. G. MALLAT. A Theory of Multi-Resolution Signal Decomposition : the Wavelet Representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7) :674–693, Juillet 1989.

[101] S. MASUD et J. V. MCCANNY. Design of Silicon IP Cores for Biorthogonal Wavelet Transforms. *Journal of VLSI Signal Processing*, 29 :179–196, 2001.

[102] C. SOUANI, M. ABID, K. TORKI, et R. TOURKI. VLSI Design of 1-D DWT Architecture with Parallel Filters. *Integration, the VLSI journal*, (29) :181–207, 2000.

[103] G. STRANG et T. NGUYEN. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, 1996.

Divers

[104] Steve BARD. Wireless Convergence of PC and Consumer Electronics in the e-Home. *Intel Technology Journal*, Mai 2001.

[105] CNES. *Techniques et Technologies des Véhicules Spatiaux*. Cépaduès, 1998.

Sites Internet

[106] 3G NEWSROOM. What is 3G ? `http://www.3gnewsroom.com/html/what_is_3g/index.shtml`, Novembre 2001.

[107] ADELANTE TECHNOLOGIES. A|RT Designer. `http://www.adelantetech.com/`.

[108] CADENCE. IP Model Packager. `http://www.cadence.com/datasheets/affirma_model_pkgr.html`.

[109] CADENCE. Virtual Component Codesign (VCC). `http://www.cadence.com/products/vcc.html`.

[110] CONSUMER ELECTRONICS ASSOCIATION. Digital America 2002 – The US Consumer Electronics Industry Today. `http://www.ce.org/publications/books_references/digital_america/default%.asp`, 2002.

[111] COWARE. N2C. `http://www.coware.com/cowareN2C.html`.

[112] DESIGN&REUSE. Site Internet. `http://www.design-reuse.com`.

[113] GET2CHIP. Architectural Compiler (G2C-AC). `http://www.get2chip.com/index/products/g2c-ac_ds.asp`.

[114] IRISA/API/COSI. Alpha Homepage. `http://www.irisa.fr/cosi/ALPHA/`.

[115] JOINT PHOTOGRAPHIC EXPERTS GROUP (JPEG). Site Internet. `http://www.jpeg.org`.

[116] LESTER. Outil GAUT. `http://lester.univ-ubs.fr/tools/gaut/tool.htm`.

[117] LESTER. Outil U-HLS Editor. `http://lester.univ-ubs.fr/tools/uhlseditor/tool.htm`.

[118] MENTOR GRAPHICS. QuickUse Development System (QDS). `http://www.mentor.com/quickuse/`.

[119] RNRT, LESTER, LASTI, et FRANCE TELECOM R&D. Site Internet du Projet MILPAT. `http://lester.univ-ubs.fr/milpat`.

[120] SHARP LABORATORIES OF EUROPE LTD. SLE Bach Technology. `http://www.sle.sharp.co.uk/research/scd/bach.htm`.

[121] SPOT IMAGE. Site Internet. `http://www.spotimage.fr`.

[122] SUMMIT DESIGN. Visual IP. `http://www.summit-design.com/VisualIP.htm`.

[123] SYNCHRONICITY. IP Gear. `http://www.synchronicity.com/products/publisher/publisher.htm`.

[124] SYNOPSYS. CoCentric SystemC Compiler. `http://www.synopsys.com/products/cocentric_systemC/cocentric_systemC.ht%ml`.

[125] SYNOPSYS. IP Modeling. http://www.synopsys.com/products/ipmodeling/ip.html.

[126] UMTS FORUM. The UMTS 3G Market Forecasts. http://www.3gnewsroom.com/html/whitepapers/year_2002.shtml, Mars 2002.

[127] VIRTUAL COMPONENT EXCHANGE (VCX). Site Internet. http://www.thevcx.com.

[128] VSI ALLIANCE. Site Internet. http://www.vsi.org.

Troisième partie

Annexes

Annexe A

Etat de l'Art des Outils de Synthèse de Haut Niveau

Un grand nombre d'outils de synthèse de haut niveau universitaires et commerciaux ont vu le jour depuis les années 1980 [79]. Le tableau A.1 en dresse un bref inventaire.

Parmi ceux-ci, il est intéressant de noter l'apparition d'un nombre significatif d'outils industriels dont la plupart ne sont pas encore pleinement opérationnels pour la production. Ce manque de maturité explique en partie le peu d'enthousiasme manifesté par les concepteurs pour la synthèse de haut niveau, et l'abandon du développement de certains outils faute d'un marché viable.

Parmi les outils mentionnés, l'abandon de certains se justifie par de toutes autres raisons : face à la perte de vitesse des langages de description de matériel comme VHDL et Verilog pour la spécification à un haut niveau d'abstraction, Monet et Behavioral Compiler cèdent la place à de nouveaux outils de synthèse de haut niveau dont le langage d'entrée est soit un sous-ensemble ANSI C/C++ dans le cas du projet "Tsunami" de Mentor Graphics, soit un sous-ensemble de SystemC pour CoCentric SystemC Compiler de Synopsys.

A.1 L'outil *GAUT* du LESTER/LASTI

L'outil *GAUT* [61] est dédié aux algorithmes de calcul intensif tels les algorithmes de traitement du signal et de l'image.

Une spécification d'entrée pour *GAUT* consiste en un couple entité/architecture écrit en *VHDL*. Le comportement à synthétiser est décrit par un unique processus. Dans sa version actuelle, l'outil *GAUT* supporte un sous-ensemble de la syntaxe *VHDL* limité aux instructions séquentielles : affectation d'expressions arithmétiques, structures conditionnelles, boucles *for* et *while*. Le modèle interne de représentation étant de type graphe flot de données, toutes les boucles doivent être déroulables, ce qui signifie que le nombre d'itérations d'une boucle doit pouvoir être déterminé statiquement au moment de la synthèse.

Le comportement temporel est conditionné par une contrainte de temps spécifiée explicitement dans le code *VHDL* comportemental. Cette contrainte exprimée en nanosecondes exprime la période d'échantillonnage des données d'entrée/sortie, et est

TAB. A.1 – Outils de synthèse de haut niveau commerciaux et académiques

Outil	Société	Début–†fin
Monet	Mentor Graphics	1997–†2000
"Tsunami"	Mentor Graphics	2001
Behavioral Compiler	Synopsys	1995–†2000
CoCentric SystemC Compiler	Synopsys	2000
Visual Architect	Cadence	1997–†1999
G2C-AC	Get2Chip	2000
A|RT Designer	Adelante Technologies	2000
Cynthesizer	Forte Design Systems	1999
System Compiler	C-Level Design	1998–†2001
Design Prototyper	Future Design Automation	2001
Bach	Sharp	1999
RapidPath	Dasys	1996–†1999

Outil	Laboratoire		Année
GAUT	LESTER	/Université de Bretagne Sud, Lorient	1992
	LASTI	/Université de Rennes I, Lannion	
DSS	DDEL	/University of Cincinnati	1993
Amical	TIMA	/Institut National Polytechnique de Grenoble	1992
Caddy	FZI	/Universität Karlsruhe	1992
Hyper	BWRC	/University of California, Berkeley	1992
Cathedral	IMEC		1986
MOODS	ECS	/University of Southampton	1997
MMAlpha	IRISA	/INRIA, CNRS,	1993
		/Université de Rennes I, INSA Rennes	
Olympus	CIS	/Stanford University	1990
DG2VHDL	CDSP	/Northeastern University, Boston	1998

indépendante de la période du signal d'horloge qui pilotera l'architecture *RTL*.

La principale contrainte du flot de synthèse étant la cadence d'entrée/sortie des données, l'ordonnancement du *DFG* s'applique à répartir les opérations sur un nombre suffisant d'étages de pipeline (voir figure 1.8e), chaque étage ayant une durée proche de la période d'échantillonnage souhaitée.

Les architectures synthétisées par *GAUT* reposent sur un modèle générique de processeur de traitement du signal qui se décompose en :

- une unité de traitement, contenant les ressources matérielles de calcul (opérateurs) de stockage temporaire (registres) et d'aiguillage des données (multiplexeurs, démultiplexeurs, *tri-states*) ;
- une unité de mémorisation, contenant les bancs mémoire et les générateurs d'adresse nécessaire au stockage des données d'entrée/sortie et des données temporaires à durée de vie longue ;
- une unité de contrôle constituée d'une machine à état fini qui produit des instructions relatives au déroulement des calculs. Un décodeur d'instructions se charge de générer les signaux de contrôle à destination des unités de traitement et de mémorisation.

Les éléments constitutifs de l'unité de traitement sont organisés sous forme de cellules de base comprenant un opérateur arithmétique, des registres pour chacune de ses entrées, des structures d'aiguillage pour la gestion du partage d'opérateur (*tri-states*) et du partage des registres (multiplexeurs/démultiplexeurs). L'unité de traitement communique avec l'unité de mémorisation par l'intermédiaire d'un bus de données dont la taille est déterminée en fonction du parallélisme d'entrée/sortie. Les opérations de lecture/écriture sur le bus sont orchestrées par l'unité de contrôle au moyen de *tri-states*.

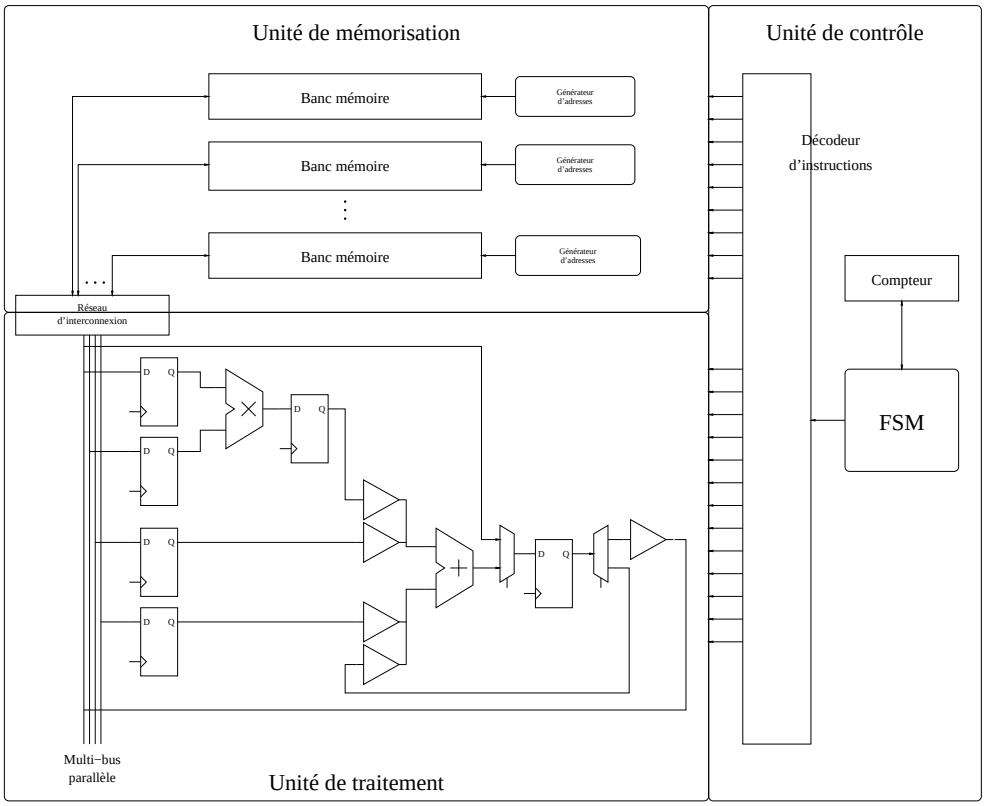
Le choix des données à mettre en mémoire, l'ordonnancement des entrées/sorties et des accès mémoire, le dimensionnement du bus interne, sont entièrement pris en charge par l'outil sans intervention du concepteur.

L'outil, bien adapté à la génération de processeurs dédiés de traitement du signal, est actuellement peu applicable dans un contexte de synthèse d'architectures réutilisables sous forme de composants virtuels. L'outil *GAUT* assume en effet une part importante des décisions d'implantation, dont certaines peuvent être en contradiction avec les contraintes d'intégration. C'est le cas, par exemple, du choix de la taille de bus et de l'ordonnancement des entrées/sorties. Des modifications profondes du fonctionnement de l'outil sont en cours au LESTER afin d'étendre le jeu de contraintes de synthèse et de permettre à un utilisateur d'avoir une plus grande maîtrise de l'interface des architectures produites. Cela passe en particulier par une spécification au cycle près de la date des entrées/sorties, fournie sous forme d'un fichier de contraintes indépendant de la description comportementale.

Listing A.1 – Description du produit matrice vecteur dans le langage VHDL pour GAUT

```
entity prodVect is
  constant N : integer ← 16;
  type Vecteur is array(0 to N-1) of integer;
  type Matrice is array(0 to N-1) of Vecteur;
  generic(
    T : time ← 100 ns
  );
  port(
    a : in Matrice;
    b : in Vecteur;
    c : out Vecteur;
  );
end prodVect;

architecture behavioral_gaut of prodVect is
begin
  process
    variable i,j : control;
    variable cvar : Vecteur;
  begin
    for i in 0 to N-1 loop
      cvar(i) ← 0;
      for j in 0 to N-1 loop
        cvar(i) ← cvar(i) + a(i)(j) × b(j);
      end loop;
      c(i) ≤ cvar(i);
    end loop;

    wait for T;
  end process;
end behavioral_gaut;
```

FIG. A.1 – Le modèle d'architecture de l'outil GAUT



A.2 Behavioral Compiler et CoCentric SystemC Compiler de Synopsys

Bien que n'ayant pas de domaine d'application privilégié, à l'inverse du précédent, l'outil Behavioral Compiler (BC) est fondé sur un flot de synthèse qui privilégie les algorithmes de calcul intensif.

Du fait de son étroite intégration avec l'outil de synthèse logique Design Compiler (DC) de Synopsys, BC se définit plus comme un ensemble de fonctionnalités avancées de synthèse pour DC que comme un outil indépendant. Ainsi, une même description VHDL ou Verilog pourra contenir des éléments relevant de différents niveaux d'abstraction, les instructions concurrentes et processus identifiés comme relevant de la synthèse RTL étant traités indépendamment par DC tandis que les processus nécessitant un ordonnancement seront pris en charge par BC.

Le listing A.2 donne le prototype d'un processus comportemental pour BC. L'ensemble du comportement est inséré dans une boucle infinie "resetLoop" dont le corps se décompose en une séquence de *reset* qui réalise l'initialisation des variables internes et des sorties lors de la remise à zéro du composant, et une seconde boucle infinie "mainLoop" qui décrit le comportement du composant lorsque le signal de *reset* est inactif.

La syntaxe supportée est assez riche pour prendre en compte la majorité des instructions séquentielles utiles à la description d'algorithmes. En particulier, BC supporte les boucles *for*, *while* et infinies ainsi que les instructions de branchement *next* et *exit* pour la sortie ou la poursuite conditionnelle d'une boucle.

Les boucles *while*, les boucles infinies et les boucles *for* dont les bornes ne sont pas définies statiquement ne sont pas déroulées par l'outil. Le corps de ces boucles est ordonnancé séparément et un mécanisme de rebouclage est implanté dans la partie contrôle de l'architecture. Les boucles *for* dont le nombre d'itérations est défini statiquement sont déroulées par défaut. L'utilisateur dispose de directives lui permettant de forcer le non-déroulage d'une telle boucle.

Une boucle non déroulée peut être ordonnancée selon un schéma pipeline. Cette possibilité permet de bénéficier des avantages des boucles non déroulées – instanciation du matériel uniquement nécessaire à une itération de la boucle, réduction du nombre d'états du contrôleur – tout en autorisant un degré plus ou moins élevé de parallélisme entre itérations successives à l'instar des boucles déroulées. A la différence de l'outil GAUT, qui traite le pipeline au niveau du processus tout entier, BC est capable d'insérer une ou plusieurs boucles pipeline à l'intérieur d'un processus non pipeline. Les directives de synthèse proposées par BC sont cependant peu maniables puisque l'utilisateur doit spécifier lui-même le nombre d'étage et le nombre de cycles d'initialisation de chaque boucle pipeline là où GAUT était capable de les déterminer seul en fonction de la contrainte de cadence.

Les contraintes d'entrée/sortie sont spécifiées directement dans le code source au moyen de l'instruction *wait*, toujours associée à la détection d'un front d'horloge. La sémantique d'implantation de cette instruction, en termes de comportement temporel de l'architecture, dépend d'une directive de l'utilisateur qui spécifie le mode d'ordon-

nancement des entrées/sorties. Trois modes sont supportés par BC.

1. Le mode *cycle-fixed* est le plus contraint. Les entrées/sorties y sont spécifiées au cycle près en tenant compte de la convention suivante : la lecture d'un signal d'entrée est effective sur le front d'horloge correspondant au *wait* qui la précède ; un signal de sortie est mis à jour sur le front d'horloge correspondant au *wait* qui suit son affectation.

   L'ordonnancement des calculs s'effectue dans les limites fixées par les dates des entrées/sorties. Des contraintes d'entrée/sortie trop serrées peuvent conduire à l'échec de l'ordonnancement.

2. Le mode *superstate-fixed* correspond à un relâchement de la contrainte de temps. Les instructions *wait* permettent de spécifier l'ordre des entrées/sorties, mais le nombre de cycles effectifs séparant deux *wait* successifs dépend de la contrainte de temps – spécifiée ici sous forme d'une directive de synthèse –, de surface, et de la quantité de calcul à ordonnancer.

3. Le mode *free-floating* n'associe aucune signification aux instructions *wait*. Dans ce mode, c'est l'ordonnancement des opérations sous contrainte de temps et de surface qui détermine l'ordre et la date d'entrée/sortie. L'outil est libre de ne pas respecter l'ordre spécifié par la description comportementale.

L'avantage du mode *cycle-fixed* est la conservation du comportement temporel aux entrées/sorties entre la description comportementale et le modèle RTL synthétisé, de telle sorte qu'un unique banc de test peut être utilisé pour les deux descriptions. En revanche, il exige de spécifier explicitement les dates d'entrée/sortie dans le code source. La modification des contraintes de temps pour obtenir différentes solutions RTL nécessite ainsi de réécrire partiellement la description.

Le mode *superstate-fixed* fournit un moyen de faire varier les contraintes de temps sans modifier la description et tout en préservant l'ordre de lecture/écriture des données. En revanche, l'ordonnancement précis des entrées/sorties n'est pas prédictible et peut nécessiter un effort d'adaptation de l'interface du composant avec son environnement.

Le mode *free-floating* donne le plus de liberté à l'outil pour optimiser l'architecture. L'imprédictibilité de l'ordonnancement des entrées/sorties est amplifiée par le fait que l'outil peut permuter l'ordre de lecture/écriture des données. Cette propriété est particulièrement difficile à gérer lorsque plusieurs lectures ou écritures successives doivent être effectuées sur un même port. L'intérêt de ce mode réside essentiellement dans son aptitude à donner rapidement à un concepteur des ordres de grandeurs de performances afin de l'aider à spécifier des contraintes de temps réalistes pour l'un des autres modes.

La figure A.2 représente le modèle des architectures RTL produites par BC. Chaque processus comportemental d'une description est projetée sur une architecture de chemin de données de type *mux-registre-mux-opérateur* possédant son propre contrôleur et ses propres ressources mémoire.

A l'inverse de l'outil *GAUT*, BC n'intervient pas sur le dimensionnement des ports d'entrée/sortie, qui restent conformes à la description comportementale quel que soit le parallélisme effectif d'accès aux ports.

La mise en mémoire des données est prise en charge directement par l'utilisateur, que ce soit pour la sélection et l'allocation des ressources mémoire, le choix des variables à mémoriser dans chaque ressource et le plan d'adressage.

Listing A.2 – Modèle de processus comportemental en VHDL pour Behavioral Compiler

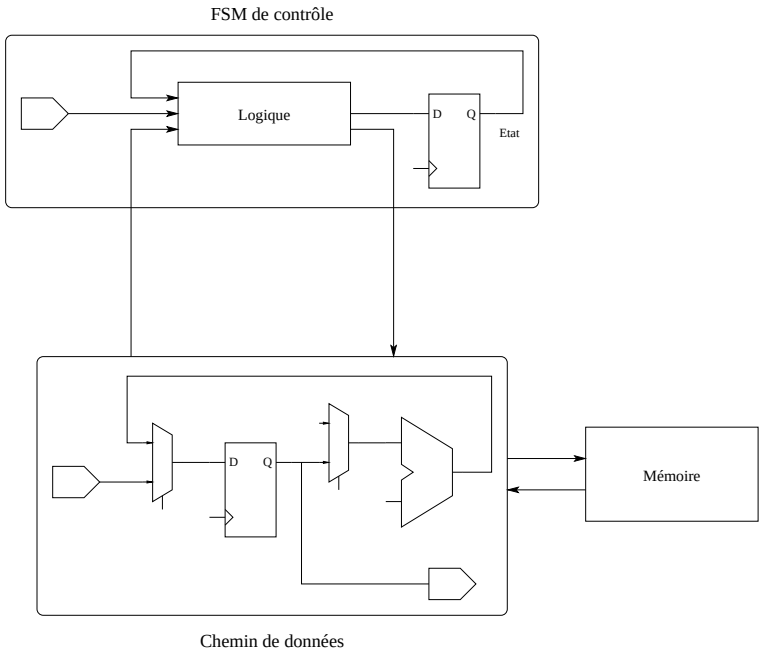
```
main : process
  -- Declaration des variables
  variable var : integer;
  ...
begin
  resetLoop : loop
    -- Etat de reset :
    -- initialisation des ports de sortie et des variables
    data_out ≤ 0;
    var ← 0;

    wait until clk'event and clk='1';
    exit resetLoop when rst='1';

    mainLoop : loop
      -- Boucle principale de traitement
      {...}

      wait until clk'event and clk='1';
      exit resetLoop when rst='1';
    end loop mainLoop;
  end loop resetLoop;
end process main;
```

L'outil CoCentric SystemC Compiler est l'un des piliers de la suite d'outils CoCentric de Synopsys, qui inclut également un outil pour l'aide à la spécification et la co-simulation (CoCentric System Studio) à partir de SystemC, ainsi qu'un outil d'aide au raffinement d'une spécification en virgule flottante vers une spécification en virgule fixe.

SystemC Compiler inclut dans un même environnement les fonctionnalités de Design Compiler et de Behavioral Compiler, mais le langage de description en entrée de ces outils est un sous-ensemble de SystemC 1.0. La syntaxe SystemC supportée en synthèse comportementale autorise des styles de description plus naturels et plus lisibles que ceux requis par BC. Il bénéficie notamment du type "signal d'horloge" proposé par SystemC, des processus synchronisés par une horloge (*SC_CTHREAD*) et de la clause *watching* permettant de rendre transparente l'action du signal de *reset*.

Les modes d'ordonnancement des entrées/sorties se réduisent aux modes *cycle-fixed* et *superstate-fixed*.

FIG. A.2 – Le modèle d'architecture de l'outil Behavioral Compiler



A.3 L'outil Monet de Mentor Graphics

L'outil Monet distribué par Mentor Graphics relève d'une philosophie très proche de celle adoptée par Synopsys pour Behavioral Compiler. La description *VHDL* d'entrée suit un style de spécification assez voisin, avec des restrictions syntaxiques plus légères qui rendent le code comportemental plus intuitif (listing A.3). La plupart des contraintes de synthèse, à l'exception des contraintes de temps en mode *cycle-fixed*, peuvent être fournies soit au moyen de scripts, soit en utilisant l'interface graphique de l'outil.

Le listing A.3 représente le prototype d'un processus comportemental pour Monet. Parmi les choix d'implantation proposés par l'outil, nous retrouverons la possibilité de dérouler ou non les boucles *for*, à la différence que Monet introduit la notion de déroulage partiel, autorisant la parallélisation de plusieurs itérations successives d'une boucle, qu'elle soit totalement déroulable ou non. Sous Monet, le déroulage partiel est un préalable à l'ordonnancement pipeline d'une boucle.

Les contraintes d'entrée/sortie sont exprimées d'une manière similaire à BC, c'est-à-dire au moyen d'instructions *wait* dont la sémantique dépend du mode d'ordonnancement des entrées/sorties. Les trois modes *cycle-fixed*, *superstate-fixed* et *free-floating* sont supportés par Monet avec les mêmes implications que sous BC. Le listing A.4 donne un exemple de description VHDL pour Monet en vue d'un ordonnancement en mode *superstate-fixed*. Les lectures et écritures multiples sur un même port sont séparées par des instructions *wait* de manière à préserver l'ordre de présentation des

données.

A la différence de BC, qui bénéficie du support de DC pour l'estimation des propriétés temps/surface de chaque ressource matérielle, Monet utilise des bibliothèques de composants pré-caractérisées en fonction de la cible choisie. Une telle bibliothèque comprend un ensemble de composants arithmétiques et logiques dont les formats d'entrées/sortie sont paramétrables de façon à les adapter aux formats de données utilisés dans la description. L'estimation des propriétés de chaque composant pour une technologie donnée est réalisée par synthèse RTL, par exemple avec DC, pour un ensemble donné de valeurs de paramètres. Pour les autres valeurs de paramètres possibles, l'estimation est réalisée par interpolation lors de l'étape de sélection/allocation.

Les techniques d'ordonnancement utilisées par Monet supportent le chaînage d'opérations et l'utilisation d'opérateurs multi-cycles. La dernière version de l'outil à ce jour ne permet pas l'utilisation d'opérateurs multi-fonctions et/ou pipelines, bien que de tels opérateurs figurent en bibliothèque.

L'ordonnancement est réalisé sur la seule base des opérations présentes dans le code comportemental, sans tenir compte du temps de traversées des couches combinatoires – multiplexeurs – ajoutées entre opérateurs et registres pour l'optimisation des ressources lors de l'assignation. L'outil prévoit une possible violation des contraintes de temps au niveau RTL en demandant à l'utilisateur de spécifier le pourcentage de la période d'horloge (*overhead*) réservé à la traversée de ces couches combinatoires. Ce pourcentage permet également de prendre en compte tôt dans le flot de conception les temps de propagation des interconnexions au niveau physique, qui deviennent non négligeables dans les technologies sub-microniques et sub-microniques profondes. Le choix de la valeur à affecter est cependant malaisé pour le concepteur qui doit attendre les rapports de synthèse finaux pour savoir si l'overhead a été sur-estimé ou sous-estimé.

L'engouement récent pour les langages de spécification basés sur C et C++ a conduit la société Mentor Graphics à revoir sa stratégie de développement en matière de synthèse de haut niveau. Le projet Monet a été abandonné depuis peu au profit d'un outil dit de *synthèse algorithmique*, orienté vers des cibles de type FPGA, avec un style de spécification plus proche du niveau système. Le choix du langage a porté sur un sous-ensemble du langage C++ indépendant des langages et bibliothèques existants tels SystemC. Le choix de ne pas ajouter à C++ de constructions spécifiques propres aux langages de description de matériel (par exemple l'instruction *wait*) implique que l'ensemble des informations de bas niveau comme les contraintes de temps, de dimensionnement des ports d'entrée/sortie, *etc.* soient fournis uniquement au moment de la synthèse par le biais de scripts ou de l'interface graphique.

Les listings A.5 et A.6 donnent une vue des deux styles de description supportés par ce nouvel outil. Le premier décrit le comportement à synthétiser au moyen d'une fonction C, pouvant aisément être invoquée dans une description de niveau système en SystemC ou SpecC. Le second style encapsule le comportement sous forme de méthode dans une classe modélisant le composant. Des méthodes spécifiques de lecture/écriture représentent les ports de ce composant.

Listing A.3 – Modèle de processus comportemental en VHDL pour Monet

```
main : process
  -- Declaration des variables
  variable var : integer;
  ...
begin
  -- Etat de reset :
  -- initialisation des ports de sortie et des variables
  data_out ≤ 0;
  var ← 0;

  mainLoop : loop
    -- Boucle principale de calcul
    wait until (clk'event and clk='1') or rst='1';
    exit mainLoop when rst='1';

    {...}

  end loop mainLoop;
end process main;
```

A.4 L'outil A|RT Designer d'Adelante Technologies

L'outil A|RT Designer distribué par la société Adelante Technologies repose sur un flot de synthèse où l'utilisateur est amené à prendre en charge de manière interactive une partie des tâches que les outils présentés précédemment effectuent automatiquement. Le modèle d'architecture cible est du type processeur de traitement du signal : il est centré sur un ensemble de ressources de calcul qui dialoguent avec des ressources de mémorisation par l'intermédiaire d'un bus, l'ensemble étant piloté par un contrôleur.

Le listing A.7 donne un exemple de description comportementale pour l'outil A|RT Designer. Le langage d'entrée est un sous-ensemble de C++ muni de types de données en virgule fixe. L'utilisateur a ici le choix entre les types définis dans les bibliothèques A|RT et les types propres au langage SystemC. Le comportement à synthétiser est décrit sous forme d'une fonction et ne contient aucune information temporelle.

Le flot de synthèse se déroule de la manière suivante (A.3) :

Après avoir rédigé et compilé la description comportementale, l'utilisateur est tout d'abord amené à sélectionner et allouer les ressources matérielles à implanter dans le chemin de données : c'est l'étape ici dénommée "création de l'architecture". Ces ressources comprennent des opérateurs arithmétiques paramétrés et des blocs de mémoire sélectionnés à partir des bibliothèques de l'outil. L'interconnexion des opérateurs et des mémoires n'est effectivement réalisée que dans les étapes suivantes de la synthèse. L'utilisateur n'est par conséquent pas tenu d'allouer les multiplexeurs et les registres du chemin de données.

L'étape de "projection" réalise l'assignation des opérations de la description comportementale aux ressources sélectionnées et allouées. Les données sont quant à elles projetées sur les ressources de mémorisation. Des multiplexeurs sont insérés dans l'ar-

Listing A.4 – Description du produit matrice vecteur dans le langage VHDL pour Monet

```
package prodVect_pkg is
  constant N : integer ← 16;
  type Vecteur is array(0 to N-1) of integer;
  type Matrice is array(0 to N-1) of Vecteur;
end prodVect_pkg;

entity prodVect is
  port( clk,rst   : in bit; start     : in boolean;
        data_in   : in integer; data_out : out integer );
end prodVect;

use work.prodVect_pkg.all;
architecture behavioral_monet of prodVect is begin
  process
    variable a : Matrice; variable b,c : Vecteur;
  begin
    data_out ≤ 0; ------------------------------- Reset --
    mainLoop : loop --================ Boucle principale --
      wait until (clk'event and clk='1') or rst='1';
      exit mainLoop when rst='1';
      next mainLoop when not start; -------- Attente start --
      for i in 0 to N-1 loop ----- Lecture de la matrice a --
        for j in 0 to N-1 loop
          if i ≠ 0 or j ≠ 0 then
            wait until (clk'event and clk='1') or rst='1';
            exit mainLoop when rst='1';
          end if;
          a(i)(j) ← data_in;
        end loop;
      end loop;
      for i in 0 to N-1 loop -------- Lecture du vecteur b --
        wait until (clk'event and clk='1') or rst='1';
        exit mainLoop when rst='1';
        b(i) ← data_in;
      end loop;
      for i in 0 to N-1 loop --- Calcul du produit c = a×b --
        c(i) ← 0;
        for j in 0 to N-1 loop
          c(i) ← c(i) + a(i)(j) × b(j);
        end loop;
      end loop;
      for i in 0 to N-1 loop ------- Ecriture du vecteur c --
        data_out ≤ c(i);
        if i ≠ N-1 then
          wait until (clk'event and clk='1') or rst='1';
          exit mainLoop when rst='1';
        end if;
      end loop;
    end loop mainLoop; --==========Fin boucle principale --
  end process;
end behavioral_monet;
```

Listing A.5 – Description d'un calcul de PGCD pour "Tsunami", version fonctionnelle

```
int gcd(int input1, int input2)
{
  if((input1 == 0)||(input2 == 0))
    return 0;

  while(input1 != input2)
  {
    if(input1 < input2)
      input2 -= input1;
    else
      input1 -= input2;
  }

  return input1;
}
```

Listing A.6 – Description d'un calcul de PGCD pour "Tsunami", version objet

```
class Gcd
{
public:
  void input1(int i){d_input1 = i;}
  void input2(int i){d_input2 = i;}
  int output(){return d_output;}

  void compute()
  {
    if((d_input1 == 0)||(d_input2 == 0))
      d_output = 0;
    else
    {
      while(d_input1 != d_input2)
      {
        if(d_input1 < d_input2)
          d_input2 -= d_input1;
        else
          d_input1 -= d_input2;
      }
      d_output = d_input1;
    }
  }

private:
  int d_input1;
  int d_input2;
  int d_output;
};
```

chitecture pour prendre en compte le partage des ressources.

L'étape d'ordonnancement définit la séquence des transferts de registres et des accès mémoire au cours du temps. Il est intéressant de noter ici que l'étape d'ordonnancement est effectuée après l'étape d'assignation, à la différence des outils présentés précédemment.

Listing A.7 – Description d'un filtre FIR en C++ pour A|RT Designer

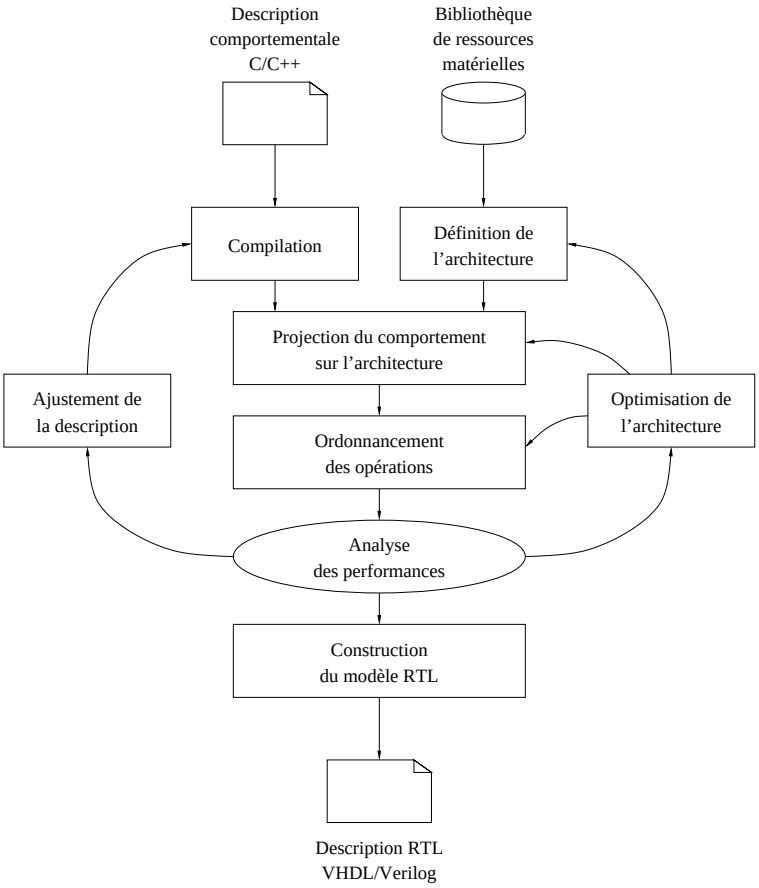
```
// Type entier sur 4 bits
typedef Int<4> DATA;

// Declaration et initialisation de la ligne à retard
DATA in_delay[3] = {0, 0, 0};

DATA filter(const DATA in, const bool shift)
{
  DATA A[3] = { 5, -3, 7 };
  DATA tmp = 0;
  DATA out;
  int i;

  // Lecture de l'echantillon courant
  in_delay[0] = in;

  // Multiplications/accumulations
  for (i = 0; i<3; ++i) {
       tmp += DATA(A[i]×in_delay[i]);
  }
  if (shift) {
    out = tmp >> DATA(1);
  } else {
    out = tmp;
  }
  // Vieillissement des donnees
  for (i=0; i<2; ++i) {
    in_delay[2-i] = in_delay[1-i];
  }
  return out;
}
```

A.5 L'outil Architectural Compiler de *Get2Chip*

L'outil G2C-AC (*Get2Chip* Architectural Compiler) développé par la société *Get2-Chip* s'interface directement avec l'outil de synthèse *RTL* G2C-RC (*Get2Chip RTL* Compiler) développé par la même société. Le langage de description au niveau comportemental est un sous-ensemble de *Verilog* : l'outil détermine seul par analyse du style d'écriture si la description qui lui est fournie relève du niveau comportemental ou d'un niveau inférieur.

L'outil supporte la plupart des techniques de synthèse déjà évoquées pour BC

FIG. A.3 – Le flot de synthèse sous A|RT Designer



et Monet : déroulage et non-déroulage des boucles, boucles pipeline, etc. Le style d'écriture au niveau comportemental est moins restrictif que celui requis par BC, notamment pour les règles d'écriture liées aux boucles non déroulées.

La spécification des contraintes d'entrée/sortie repose sur une syntaxe très proche de celle requise par Monet et BC en mode *cycle-fixed* : l'ordre et la date des entrées/sorties sont spécifiés directement dans le code source au moyen d'instruction en séparant les lectures et écriture par des instructions de synchronisation sur un front d'horloge *(posedge clk)*.

Cette approche permet de faire interagir dans une même description *Verilog* hiérarchique des blocs décrits à différents niveaux d'abstraction : netlists au niveau portes, composants *RTL*, blocs comportementaux dont les entrées/sorties sont décrites au cycle près. Dans l'environnement *Get2Chip*, ce style de spécification est identifié sous l'acronyme *POCA* (*Pins Out Cycle Accurate*).

Le modèle de description d'un processus comportemental en *Verilog* pour *G2C-AC* est présenté dans le listing A.8.

Listing A.8 – Modèle de processus comportemental en Verilog pour Get2Chip

```
always begin : main_loop
  // Etat de reset :
  // initialisation des ports de sortie et des variables
  data_out ≤ 0;
  var = 0;
  ...
  @(posedge clk or posedge rst);
  if(rst) disable main_loop;

  forever begin
    // Boucle pricipale de traitement

    {...}

    @(posedge clk or posedge rst);
    if(rst) disable main_loop;
  end
end
```

A.6 Résumé

Le tableau A.2 résume les caractéristiques principales des outils *GAUT* (LESTER/LASTI), Monet (Mentor Graphics) et BC (Behavioral Compiler, Synopsys). Nous observons que selon les outils, certaines tâches peuvent être réalisées soit automatiquement sous contrainte – c'est le cas de la mise en mémoire des variables, de l'ordonnancement des entrées/sorties et de l'optimisation du bus d'E/S sous *GAUT* –, soit manuellement au moyen de directives détaillées.

TAB. A.2 – Comparaison de trois outils de synthèse de haut niveau

	GAUT	Monet	BC
Déroulage des boucles	Toujours	Directive	Directive
Non-déroulage	–	Directive	Directive
Déroulage partiel	–	Directive	–
Contraintes de temps	Cadence d'E/S	Nombre d'états	Latence
Contraintes de surface		Minimiser	Borne supérieure
Bibliothèques	Pré-caractérisées	Scripts caractérisation	Caractérisation en synthèse
Chaînage	–	Oui	Oui
Composants multicycles	Oui	Oui	Oui
Composants pipeline	–	En projet	–
Ordonnancement pipeline	Processus	Boucle	Boucle
Optimisation bus	Oui	–	–
Contrainte d'ordre d'E/S	–	Mode superstate-fixed	
Contrainte de date d'E/S	–	Mode cycle-fixed	
Synthèse mémoire	Automatique	Directives	Directives

Le modèle de description sous-jacent autorise différents types d'ordonnancements. La prise en compte d'une contrainte de cadence de traitement par ordonnancement pipeline est réalisée par *GAUT* au niveau du processus tout entier alors que Monet et BC peuvent appliquer le même procédé à toute boucle non déroulée de la description.

Le tableau A.3 dresse une comparaison des styles de spécification et des jeux de contraintes supportés par quelques outils universitaires et commerciaux.

TAB. A.3 – Outils de synthèse de haut niveau universitaires et commerciaux : comparaison des styles de spécification

Outil	Langage de description	Granularité temporelle au niveau comportemental	Classes d'applications
Monet	VHDL	Free-floating Superstate-fixed Cycle-fixed	Données>contrôle
"Tsunami"	ANSI C/C++	Processus sans temps	Données>contrôle
Behavioral Compiler	VHDL/ Verilog	Free-floating Superstate-fixed Cycle-fixed	Données>contrôle
CoCentric SystemC Compiler	SystemC 1.0	Superstate-fixed Cycle-fixed	Données>contrôle
Visual Architect	VHDL	–	
G2C-AC	Verilog	POCA	Données>contrôle
A|RT Designer	C/SystemC A|RT Library	Processus sans temps	Données>contrôle
Cynthesizer	C++/Cynlib	–	
System Compiler	CycleC	–	
Design Prototyper	C/C++	E/S au cycle près	Données>contrôle
Bach	Bach-C	–	
RapidPath	VHDL/ Verilog	–	
GAUT	VHDL	Processus de période fixée	Données
DSS	VHDL	Synchro sur événements	Contrôle>données
Amical	VHDL	Synchro sur événements	Contrôle>données
Caddy	VHDL	–	
Hyper	Silage	Opérateur retard	Données
Cathedral	Silage	Opérateur retard	Données
MOODS	VHDL	–	
MMAlpha	Alpha	Processus sans temps	Données
Olympus/Hercules	HardwareC	Temps relatif aux d'E/S parallélisme implicite/explicite	Contrôle>données
DG2VHDL	DG	Algorithme	Données

Annexe B

Exploration des Graphes de Dépendances

B.1 Définition

Un graphe de dépendances sera défini comme un quintuplet $(\mathcal{I}, \mathcal{O}, \mathcal{T}, \mathcal{F}, \mathcal{D})$ où
- $\mathcal{I}$ est le domaine de définition des données d'entrée : $\mathcal{I} \subset \mathbb{Z}^{N_I}$;
- $\mathcal{O}$ est le domaine de définition des données de sortie : $\mathcal{O} \subset \mathbb{Z}^{N_O}$;
- $\mathcal{T}$ est le domaine de définition des données intermédiaires : $\mathcal{T} \subset \mathbb{Z}^{N_T}$.

Dans [24,50], ces domaines sont définis comme des polyèdres bornés suivant toutes les dimensions. Dans notre modèle, nous admettons des coordonnées non bornées suivant une seule des dimensions de chacun des domaines, et dans une seul sens.

Nous considérons tout d'abord que les données intermédiaires contiennent une copie des données d'entrée et de sortie, mais selon un arrangement plus adapté à la structure de l'algorithme. Nous définissons tout d'abord deux sous-domaines de $\mathcal{T}$:
- $\mathcal{T}_I \subset \mathcal{T}$ est le sous-domaine des données d'entrée ;
- $\mathcal{T}_O \subset \mathcal{T}$ est le sous-domaine des données de sortie.

Le réarrangement des données d'entrée vers les données intermédiaires et des données intermédiaires vers les données de sortie se fait au moyen de deux applications bijectives :

$$\begin{aligned} \mathcal{P}_I &: \mathcal{I} \longrightarrow \mathcal{T}_I \\ \mathcal{P}_O &: \mathcal{T}_O \longrightarrow \mathcal{O} \end{aligned} \tag{B.1}$$

$\mathcal{F}$ représente l'ensemble des fonctions, ou opérations, utilisées dans l'algorithme. Chaque opération $f \in \mathcal{F}$ se caractérise par
- le nombre de données qu'elle prend en entrée : $in(f) \in \mathbb{N}^*$
- le nombre de données qu'elle produit en sortie : $out(f) \in \mathbb{N}^*$

Les dépendances de données sont représentées par un ensemble $\mathcal{D}$ de triplets (f, T_{in}, T_{out}) où

- f est une opération : $f \in \mathcal{F}$;
- T_{in} est un n-uplet d'éléments de $\mathcal{T}$ représentant les données d'entrée de l'opération : $T_{in} \in \mathcal{T}^{in(f)}$;
- T_{out} est un n-uplet d'éléments de $\mathcal{T}$ représentant les sorties de l'opération considérée : $T_{out} \in \mathcal{T}^{out(f)}$.

B.2 Graphe de dépendances et ordre de parcours des données

B.2.1 Ordre partiel des données d'entrée/sortie

Lors du raffinement d'un comportement modélisé par un graphe de dépendances, le concepteur définit tout d'abord une relation d'ordre partiel sur les données d'entrée et/ou de sortie. Formellement, étant donnés deux points i_1 et i_2 du domaine $\mathcal{I}$, nous noterons $i_1 \preceq i_2$ la relation "i_1 est lue avant, ou en même temps que, i_2". De même pour o_1 et o_2 appartenant à $\mathcal{O}$, $o_1 \preceq o_2$ signifiera que la donnée de sortie repérée par o_1 est mise à jour avant, ou en même temps que celle repérée par o_2.

Exemple 1

Si les données d'entrée sont bidimensionnelles, nous pourrons par exemple spécifier une relation du type :

$$\begin{aligned} \forall i_1 = (x_1, y_1), i_2 = (x_2, y_2) \in \mathcal{I}, x_1 \leq x_2 \wedge y_1 = y_2 &\Rightarrow i_1 \preceq i_2 \\ y_1 \leq y_2 \wedge x_1 = x_2 &\Rightarrow i_1 \preceq i_2 \end{aligned} \tag{B.2}$$

Cette relation signifie que toute donnée est lue avant, ou en même temps que celles situées à sa "droite" sur la même ligne, et avant, ou en même temps que celles situées "plus bas" sur la même colonne. La relation ne définit aucune priorité entre la direction "horizontale" et la direction "verticale". Elle peut ainsi correspondre à une variété de parcours des échantillons.

Exemple 2

Un deuxième exemple est celui d'un parcours des données en diagonale :

$$\forall i_1 = (x_1, y_1), i_2 = (x_2, y_2) \in \mathcal{I}, x_1 + y_1 < x_2 + y_2 \Rightarrow i_1 \preceq i_2 \tag{B.3}$$

Dans cet exemple, l'ordre de succession des diagonales est connu, mais l'ordre de parcours le long d'une diagonale n'est pas spécifié.

Ordre partiel et axes temporels

Nous définissons une notion d'*axe temporel directionnel* en mettant en évidence des sous-relations d'ordre total dans la relation d'ordre partiel entre données d'entrée ou de sortie. Dans les relations B.2, nous observerons par exemple qu'il existe une relation d'ordre totale entre échantillons d'une même ligne ou d'une même colonne. Chaque échantillon intervient ainsi dans deux sous-relations d'ordre total, il est alors

possible de définir une notion de "temps vertical" et de "temps horizontal" consistant à repérer chaque donnée par une paire d'indices correspondant à son numéro d'ordre dans les deux relations totales où elle intervient.

Dans la pratique, ces relations d'ordre partiel seront spécifiées en réindexant les données de manière à faire apparaître différents ordres de parcours le long de différents axes. Les axes temporels seront communs aux données d'entrée, de sortie et intermédiaires. Nous définissons des fonctions de réindexation $\mathcal{R}_I$ et $\mathcal{R}_O$ qui associent respectivement aux indices des données d'entrée et de sortie un nouveau jeu de Q indices. La fonction $\mathcal{R}_I$ est définie comme suit :

$$\mathcal{R}_I : \mathbb{Z}^{N_I} \longrightarrow \mathbb{N}^Q \tag{B.4}$$

$$\forall i_1, i_2 \in \mathcal{I}, \mathcal{R}_I(i_1) = (n_1, n_2, \ldots, n_Q), \tag{B.5}$$

$$\mathcal{R}_I(i_2) = (m_1, m_2, \ldots, m_Q) \tag{B.6}$$

$$i_1 \preceq i_2 \wedge i_1 \neq i_2 \Rightarrow \exists! q \in [1..Q], n_q < m_q) \tag{B.7}$$

Après réindexation, la relation "i_1 est lue avant, ou en même temps que i_2" impliquera que tous les nouveaux indices de i_1 et i_2 sont identiques, à l'exception d'un seul, autrement dit la "droite" $(i_1 - i_2)$ est parallèle à l'un des axes temporels. La fonction de réindexation $\mathcal{R}_O$ suit le même type de définition.

Si $Q < N_I$, deux données différentes peuvent se voir affecter les mêmes indices temporels. Dans ce cas, il est nécessaire de compléter le jeu d'indices de manière à identifier chaque donnée par des coordonnées uniques. Nous noterons $\bar{\mathcal{R}}_I$ et $\bar{\mathcal{R}}_O$ les fonctions de calcul des indices complémentaires.

Dans le cas de la relation B.2, aucune réindexation n'est nécessaire : les indices de ligne et colonne expriment déjà la relation d'ordre partiel décrite. Dans le cas de la relation B.3, les fonctions de réindexation et sa fonction complémentaire peuvent être :

$$\forall i = (x, y) \in \mathcal{I}, \mathcal{R}_I(i) = x + y \tag{B.8}$$

$$\bar{\mathcal{R}}_I(i) = x \tag{B.9}$$

Nous verrons par la suite que la spécification séparée des axes temporels pour les données d'entrée et de sortie n'est pas nécessairement possible : les dépendances de données peuvent en effet restreindre tant le choix des axes que des valeurs d'indices associées à chaque donnée.

B.2.2 Ordre partiel des données intermédiaires

Les dépendances de données, de leur côté, imposent un ordre partiel entre opérations. Nous noterons $t_2 \succeq t_1$ la relation établissant que la donnée intermédiaire repérée par le n-uplet d'indices t_2 dépend de la donnée intermédiaire repérée par t_1 :

$$\exists d = (f, T_{in}, T_{out}) \in \mathcal{D} | t_1 \in T_{in}, t_2 \in T_{out} \Rightarrow t_2 \succeq t_1 \tag{B.10}$$

D'un point de vue temporel, nous supposons que chaque opération s'effectue en un temps nul, d'où le choix du symbole $\succeq$, indiquant que deux données dépendant l'une de l'autre peuvent être produites simultanément ou avec un certain retard. La notation $a \preceq b$ (a précède b dans le temps) ne doit pas être confondue avec la notation

$b \succeq a$ (b succède à a par dépendance). Ces deux notations sont liées par la relation suivante :

$$b \succeq a \Rightarrow a \preceq b \tag{B.11}$$

les deux relations d'ordre n'étant que partielles, la réciproque n'est pas nécessairement vraie.

Nous allons à présent chercher à propager les indices temporels des données d'entrée/sortie à travers les données intermédiaires en définissant une fonction de réindexation $\mathcal{R}_T$ selon les axes temporels, et une fonction complémentaire $\bar{\mathcal{R}}_T$. Tout d'abord, il est possible de faire correspondre directement ces indices aux ensembles de données intermédiaires qui sont des copies des données d'entrée/sortie :

$$\begin{array}{rcl} \forall i \in \mathcal{I}, t \in \mathcal{T}_I, t = \mathcal{P}_I(i) & \Rightarrow & \mathcal{R}_T(t) = \mathcal{R}_I(i) \wedge \bar{\mathcal{R}}_T(t) = \bar{\mathcal{R}}_I(i) \\ \forall o \in \mathcal{O}, t \in \mathcal{T}_O, o = \mathcal{P}_O(t) & \Rightarrow & \mathcal{R}_T(t) = \mathcal{R}_O(o) \wedge \bar{\mathcal{R}}_T(t) = \bar{\mathcal{R}}_o(o) \end{array} \tag{B.12}$$

Ensuite, la relation d'ordre partiel entre les données intermédiaires permet de propager ces indices dans les sens aval et amont en respectant la causalité :

$$\begin{array}{rcl} \forall t \in \mathcal{T} \backslash (\mathcal{T}_I \cup \mathcal{T}_O) & & \\ \forall j \in [1..Q], \mathcal{R}_T(t)_j & = & max\{R_T(u)_j | t \succeq u\} \\ & = & min\{R_T(v)_j | v \succeq t\} \end{array} \tag{B.13}$$

Comme nous le voyons, il existe une relation indirecte entre les indices temporels des données d'entrée et de sortie : les dépendances de données imposent elles-mêmes une relation d'ordre partiel entre entrées et sorties. Nous utiliserons la notation $o \succeq i$ pour désigner l'existence d'une chaîne de dépendances entre un donnée de sortie repérée par o et une donnée d'entrée repérée par i.

$$\forall o \in \mathcal{O}, i \in \mathcal{I}, \mathcal{P}_O^{-1}(o) \geq \mathcal{P}_I(i) \Rightarrow o \succeq i \tag{B.14}$$

Ainsi, il ne sera pas nécessairement possible de spécifier l'ordre des entrées et l'ordre des sorties séparément. Nous pouvons compléter les relations B.12 avec :

$$\begin{array}{rcl} \forall i \in \mathcal{I}, \mathcal{R}_I(i)_j & = & min\{\mathcal{R}_O(o)_j | o \succeq i\} \\ \forall o \in \mathcal{O}, \mathcal{R}_O(o)_j & = & max\{\mathcal{R}_I(i)_j | o \succeq i\} \end{array}$$

A ce stade, selon que l'algorithme produit moins de données qu'il n'en consomme, ou produit plus de données qu'il n'en consomme, nous aurons respectivement $\mathcal{R}_O(\mathcal{O}) \subset \mathcal{R}_I(\mathcal{I})$ ou $\mathcal{R}_I(\mathcal{I}) \subset \mathcal{R}_O(\mathcal{O})$. Dans la suite, nous ferons l'hypothèse que les indices dans les plus grand des ensembles $\mathcal{R}_I(\mathcal{I})$ et $\mathcal{R}_O(\mathcal{O})$ se succèdent avec un pas de 1.

B.3 Périodicité du graphe et motifs répétitifs de calcul

L'extraction d'un motif répétitif de traitement consiste à isoler un groupe de nœuds qui se répète périodiquement le long de chaque axe temporel. La périodicité sera définie comme l'existence d'une suite d'entiers strictement positifs $(N_q)_{1 \leq q \leq Q}$ tels qu'il existe

une application $\mathcal{F}$ qui laisse le graphe invariant par translation des données de N_q positions le long de l'axe temporel numéro q.

$$\begin{array}{rcl}
\mathcal{F} & : & \left\{ \begin{array}{rcl} (\mathcal{D}) & \longrightarrow & (\mathcal{D}) \\ d_1 = (f, T_{in,1}, T_{out,1}) & \longmapsto & d_2 = (f, T_{in,2}, T_{out,2}) \end{array} \right. \\
\forall q \in [1..Q] & , & \\
\forall j \in [1..in(f)] & , & \mathcal{R}_T(t_{in,2,j})_q = \mathcal{R}_T(t_{in,1,j})_q + N_q, \\
 & & \bar{\mathcal{R}}_T(t_{in,2,j})_q = \bar{\mathcal{R}}_T(t_{in,1,j})_q, \\
\forall j \in [1..out(f)] & , & \mathcal{R}_T(t_{out,2,j})_q = \mathcal{R}_T(t_{out,1,j})_q + N_q \\
 & & \bar{\mathcal{R}}_T(t_{out,2,j})_q = \bar{\mathcal{R}}_T(t_{out,1,j})_q,
\end{array} \tag{B.15}$$

Par la suite, nous noterons N_q les valeurs minimales satisfaisant cette propriété.

B.3.1 Motif de calcul : définition

Un motif de calcul est un graphe dont la réplication périodique le long des Q axes temporels engendre tout ou partie du graphe de dépendances d'un algorithme. Chacune des instances du motif à l'intérieur du graphe de dépendances représente une *itération* du processus modélisant le comportement du composant virtuel.

Les nœuds d'une itération d'un motif doivent vérifier la causalité des opérations : si nous définissons la *profondeur* $p(t)$ d'une donnée intermédiaire t comme la longueur de la plus longue chaîne de dépendances qui la relie aux données d'entrée, les indices temporels associés à t doivent être supérieurs ou égaux aux indices temporels des données de profondeur inférieure appartenant à la même itération du motif.

Chaque itération sera associée à un Q-uplet d'indices entiers naturels que nous dénommerons *indices d'état*. La projection temporelle du motif consiste à établir une relation d'ordre strict entre les itérations. Dans la pratique, cette relation sera définie à l'aide d'une fonction bijective $\mathcal{G}$ qui associe à chaque Q-uplet d'indices d'état un indice de date sous la forme d'un entier naturel.

Les indices de date doivent respecter la causalité. En d'autres termes, $\mathcal{G}$ doit être strictement croissante suivant chacun des directions temporelles.

B.3.2 Propriétés d'un motif

La *taille* S d'un motif sera définie comme un Q-uplet $(S_1, S_2, \ldots, S_Q)$ de valeurs entières, chaque valeur S_q étant multiple de N_q.

Etant donnée une itération du motif, nous nous intéresserons à caractériser les données en termes de nombre de transferts par itération et de durée de vie. Tout d'abord, chaque itération consomme une quantité constante de données d'entrée et produit une quantité constante de données de sortie. Ces quantités dépendent uniquement de la taille S du motif.

Cette propriété privilégie une représentation de la vue externe du comportement à l'aide d'un modèle flot de données synchrone *SDF*.

En fonction de la topologie du motif, une itération peut également consommer et produire des quantités constantes de données intermédiaires, respectivement calculées ou utilisées lors d'itérations précédentes ou ultérieures du motif. La durée de vie de ces données intermédiaires s'exprime comme le nombre d'itérations écoulées entre leur production et leur dernière consommation. Ce nombre s'obtient en appliquant la fonction $\mathcal{G}$ aux itérations concernées.

Le parallélisme exploitable au niveau opération est mesuré en extrayant des ensembles d'opérations appartenant à une même itération, mais n'ayant pas de dépendances entre elles.

Annexe C

Machines à Etat Fini avec Chemin de Données

C.1 FSM

Une machine à état fini (*FSM*) décrit le comportement séquentiel d'un système. Elle se définit mathématiquement comme un quintuplet (S, I, O, f, h) où

- S est l'ensemble des états que peut prendre le système ;
- I est l'ensemble des entrées possibles du système ;
- O est l'ensemble des sorties possibles du système ;
- f est la fonction de transition ;
- h est la fonction de sortie.

L'ensemble des transitions possibles entre deux états et les conditions sous lesquelles chaque transition peut être effectuée sont décrits par la fonction f. A un instant donné, la valeur du prochain état du système est calculée en fonction de l'état courant et de la valeur des entrées :

$$f : S \times I \longrightarrow S \tag{C.1}$$

Dans le cas général (machine de Mealy) la valeur des sorties à un instant donné dépend de l'état courant et de la valeur des entrées :

$$h : S \times I \longrightarrow O \tag{C.2}$$

L'initialisation d'une *FSM* nécessite la définition d'un état initial $s_0 \in S$. Les fonctions f et h peuvent être spécifiées sous forme de tableaux indiquant pour chaque valeur possible des entrées et de l'état à l'instant courant la valeur des sorties et de l'état suivant.

C.2 FSMD

C.2.1 Définition

Une machine à état fini avec chemin de données (*FSMD*) est définie comme une machine à état fini dont l'évolution détermine une séquence de calculs à réaliser, les

transitions pouvant elles-mêmes dépendre du résultat des calculs. Une *FSMD* repose sur les ensembles suivants :

- S : ensemble d'états ;
- I : ensemble d'entrées ;
- O : ensemble de sorties ;
- V : ensemble de variables du chemin de données ;
- OP : ensemble d'opérateurs ;
- EXP : ensemble d'expressions ;
- $STAT$: ensemble de signaux de *status*.

La valeur des variables V définit l'*état* du chemin de données. Les fonctions f et h définies dans le cas d'une *FSM* simple doivent être complétées afin de prendre en compte l'évolution conjointe de l'état de la partie *FSM* et de l'état de la partie chemin de données d'une *FSMD*.

Pour ce faire, nous définissons tout d'abord les expressions (éléments de EXP) modélisant les calculs effectués dans le chemin de données. Si nous représentons chaque donnée de I et V comme la concaténation, respectivement, de N et P valeurs ($I = I_1 \times I_2 \times \ldots \times I_N$ et $V = V_1 \times V_2 \times \ldots \times V_P$), le résultat d'une expression sera obtenu en appliquant une opération u à un extrait $(i_{j_1}, i_{j_2}, \ldots, v_{k_1}, v_{k_2}, \ldots)$ de la valeur des variables et des entrées :

$$EXP = \{u(x) | u \in OP \text{ et } x \in I_{j_1} \times I_{j_2} \times \ldots \times V_{k_1} \times V_{k_2} \times \ldots\} \tag{C.3}$$

Les signaux de *status* ($STAT$) représentent des relations logiques entre éléments de EXP. Elles fournissent un résultat booléen utilisé dans les transitions conditionnelles de la *FSM*.

$$STAT = \{Rel(a, b) | a, b \in EXP\} \tag{C.4}$$

La fonction de transition est décomposée en une fonction de calcul de l'état suivant de la *FSM* f_C – en fonction de l'état courant, des entrées et des signaux de *status* – et une fonction f_D de calcul de la valeur suivante des variables du chemin de données en fonction des entrées, des résultats d'expressions et des signaux de *status* :

$$f_C \quad : \quad S \times I \times STAT \longrightarrow S \tag{C.5}$$

$$f_D \quad : \quad S \times I \times STAT \times EXP \longrightarrow V \tag{C.6}$$

La fonction de calcul des sorties suit le même modèle que la fonction de mise à jour des variables :

$$h : S \times I \times STAT \times EXP \longrightarrow O \tag{C.7}$$

En plus d'un état initial s_0 pour la partie *FSM*, l'initialisation d'une *FSMD* nécessite la définition d'une valeur initial $v_0 \in V$ des variables.

C.2.2 Application

Le modèle *FSMD* est généralement utilisé pour modéliser le comportement des architectures matérielles au niveau transfert de registres. Les transitions s'effectuent au rythme de l'horloge pilotant l'architecture. L'ensemble des opérations associées à un état donné est censé s'exécuter en au plus une période d'horloge.

Le rapport [30] distingue cinq styles de modélisation de matériel au niveau *RTL* à partir d'une *FSMD* :

1. *Unmapped RTL*,
2. *Storage-mapped RTL*,
3. *Function-mapped RTL*,
4. *Connection-mapped RTL*,
5. *Exposed-control RTL*.

Dans le style *unmapped RTL*, les variables et opérations ne sont associées à aucune interprétation matérielle. Les styles 2 à 4 permettent d'associer progressivement des réalités matérielles aux éléments de la *FSMD*, tout d'abord en associant les variables à de simples connexions ou à des ressources de mémorisation (registres, *RAM*) en fonction de leur durée de vie (*storage-mapped RTL*), puis en projetant les opérations sur des unités fonctionnelles (*function-mapped RTL*) et en groupant les connexions portant des données de durées de vie disjointes sous forme de bus (*connection-mapped RTL*).

Le style *exposed-control RTL* sépare explicitement la partie traitement de l'architecture – représentée sous la forme d'une *netlist* d'éléments du chemin de données – de la partie contrôle, représentée comme une simple *FSM*.

Enfin, un système complexe intègre de multiples composants matériels, chacun modélisable sous la forme d'une *FSMD* séparée. Le rapport [30] spécifie la sémantique des *FSMD* concurrentes et communicantes.

C.3 SFSMD

Une *SFSMD* (*Superstate Finite State Machine*) suit un formalisme identique à celui des *FSMD*. La différence entre les deux modèles réside principalement dans l'usage qui en est fait : une *FSMD* permet de modéliser au cycle près le comportement d'une architecture *RTL* tandis que le nombre de cycles d'horloge s'écoulant entre deux transitions successives d'une *SFSMD* est indéterminé.

De même, la complexité des expressions modélisant le chemin de données n'est plus restreinte à une opération exécutable en au plus une période d'horloge comme c'était le cas dans les *FSMD*.

Le modèle *SFSMD* est particulièrement adapté à la synthèse de haut niveau dans le sens où une description comportementale peut être partitionnée en *super-états* – permettant notamment de fixer des points de synchronisation dans l'exécution d'un processus – sans préjuger du nombre de périodes d'horloges nécessaires à l'exécution de chaque super-état.

Une telle notion de *super-état* est employée en synthèse de haut niveau dans les outils *Monet*, *Behavioral Compiler* et *CoCentric SystemC Compiler*. Pour ces outils, chaque super-état correspond à un point de réactualisation de la valeur des signaux d'entrée/sortie. Ces valeurs sont supposés être stables pendant toute la durée d'un super-état.

L'étape d'ordonnancement dans un flot de synthèse de haut niveau peut ainsi être envisagée comme un raffinement d'un modèle *SFSMD* en un modèle *FSMD*, en dé-

composant chaque super-état en autant d'états que de cycles nécessaires à l'exécution des instructions qui lui sont attachées, et en répartissant les opérations sur chacun des états obtenus.

Annexe D

Analyse Multirésolution et Transformation en Ondelettes Discrète

La transformation en ondelettes se trouve à la jonction de diverses théories mathématiques. Parmi celles-ci, l'analyse multirésolution et la décomposition en sous-bandes nous paraissent les plus à même de justifier l'intérêt de la transformation en ondelettes dans le domaine de la compression d'images.

Nous présenterons dans un premier temps la transformation en ondelettes monodimensionnelle avant d'extrapoler au cas bidimensionnel.

D.1 Analyse multirésolution

D.1.1 Approximations d'une fonction à différentes résolutions

L'analyse multirésolution repose sur la décomposition de l'espace des fonctions de carré sommable – *i.e.* des signaux à énergie finie – $L^2(\mathbb{R})$ en un ensemble de sous-espaces imbriqués V_j appelés espaces d'approximation.

$$V_{-\infty} \subset \ldots \subset V_{-2} \subset V_{-1} \subset V_0 \subset V_1 \subset V_2 \subset \ldots \subset V_{+\infty} \tag{D.1}$$

Chacun des espaces V_j est caractérisé par la résolution maximale des fonctions qu'il contient. Les espaces de basse résolution contiennent des fonctions présentant des détails grossiers – *i.e.* des transitions douces – tandis que les espaces de haute résolution supportent des fonctions pouvant présenter des détails fins – correspondant à des variations rapides. L'évolution de la résolution d'un espace V_j à l'espace V_{j+1} se traduit par une réduction de la largeur des détails par un facteur 2. Autrement dit, il est possible de faire passer une fonction de V_j à V_{j+1} en contractant son support par 2 (équation D.2).

$$\forall j \in \mathbb{Z},\ v(t) \in V_j \iff v(2t) \in V_{j+1} \tag{D.2}$$

Etant donnée une suite d'espaces V_j il est possible d'approcher toute fonction f de $L^2(\mathbb{R})$ par une fonction $\hat{f}_j$ à une résolution j arbitrairement fine en projetant f sur V_j.

$$\hat{f}_j = \mathcal{P}_{V_j} f \tag{D.3}$$

Afin de mener à bien cette projection, on définit une base orthonormée $(\varphi_{j,k})_{k\in\mathbb{Z}}$ de V_j à partir d'une fonction φ de V_0 appelée fonction d'échelle. Chaque vecteur de cette base reproduit la fonction d'échelle après l'avoir dilatée ou contractée par un facteur 2^j et décalée de k.

$$\varphi_{j,k}(t) = 2^{j/2}\varphi(2^j t - k) \tag{D.4}$$

L'approximation $\hat{f}_j$ de f à la résolution j s'écrit comme une combinaison linéaire des vecteurs de base $\varphi_{j,k}$:

$$\hat{f}_j(t) = \sum_{k\in\mathbb{Z}} v_f(j,k)\varphi_{j,k}(t) \tag{D.5}$$

avec

$$v_f(j,k) = \langle f, \varphi_{j,k}\rangle = \int_{-\infty}^{+\infty} f(t)\bar{\varphi}_{j,k}(t)dt \tag{D.6}$$

Les cœfficients $v_f(j,k)$ sont appelés cœfficients d'approximation de f à la résolution j.

D.1.2 Extraction des détails d'une fonction à différentes résolutions

Les espaces d'approximation et leurs bases orthonormées dérivées de la fonction d'échelle fournissent un outil efficace pour approcher une fonction à n'importe quelle résolution. L'approche complémentaire va maintenant consister à extraire les détails d'une fonction à chaque niveau de résolution. Etant donnée une fonction f de V_{j+1}, c'est-à-dire comprenant des détails dont le niveau de résolution maximal est $j+1$, on peut considérer que projeter f sur V_j revient à éliminer de f les détails de niveau supérieur à j. A la fonction approchée $\hat{f}_j$ de f, il est ainsi possible d'associer une fonction $\hat{g}_j$ contenant les détails de f perdus lors de la projection :

$$f(t) = \hat{f}_j(t) + \hat{g}_j(t) \tag{D.7}$$

On complète la décomposition de $L^2(\mathbb{R})$ en définissant un ensemble d'espaces de détails W_j, chacun étant défini comme le complémentaire orthogonal de V_j dans V_{j+1} :

$$V_{j+1} = V_j \oplus W_j \tag{D.8}$$

On définit de même une base orthonormée $(\psi_{j,k})_{k\in\mathbb{Z}}$ de W_j à partir d'une fonction ψ de W_0 nommée ondelette-mère :

$$\psi_{j,k}(t) = 2^{j/2}\psi(2^j t - k) \tag{D.9}$$

Une ondelette est une fonction oscillante d'intégrale nulle et dont l'amplitude décroît assez rapidement. A l'instar de la base des fonctions d'échelle, une base d'ondelettes se construit par dilatation/contraction et translation de l'ondelette-mère (équation D.9).

La projection d'une fonction f sur les différents espaces W_j par l'intermédiaire de la base d'ondelettes permet ainsi d'en extraire les détails à différents niveaux de résolution :

$$\hat{g}_j(t) = \mathcal{P}_{W_j} f(t) = \sum_{k\in\mathbb{Z}} w_f(j,k)\psi_{j,k}(t) \tag{D.10}$$

les cœfficients de détails $w_f(j,k)$ étant définis par

$$w_f(j,k) = \langle f, \psi_{j,k} \rangle = \int_{-\infty}^{+\infty} f(t)\bar{\psi}_{j,k}(t)dt \tag{D.11}$$

On observera que le calcul d'un cœfficient de détail est proche d'un calcul d'intercorrélation entre $f(t)$ et $\psi(2^j t)$. Une interprétation possible de ce calcul consiste à considérer que les valeurs de $w_f(j,k)$ seront significatives aux points k où f présente le plus de ressemblance avec l'ondelette-mère dilatée ou contractée par 2^j. Une valeur significative de $w_f(j,k)$ dénote par conséquent la présence au point k d'un détail de niveau de résolution j. La forme de l'ondelette-mère, et plus particulièrement sa vitesse de décroissance, conditionne la structure des détails qui pourront être détectés de cette manière.

D.2 Transformation en ondelettes et son inverse

La fonction discrète w_f définie sur $\mathbb{Z}^2$ par la relation D.11 est la transformée en ondelettes discrète de f pour l'ondelette-mère ψ considérée.

En développant la relation de récurrence D.8, on observe qu'un espace d'approximation à un niveau de résolution donné se construit en sommant les espaces de détails des niveaux inférieurs (équation D.12). A la limite, l'ensemble $L^2(\mathbb{R})$ s'écrit comme la somme des espaces de détails à toutes les résolutions (équation D.13).

$$V_n = \bigoplus_{j=-\infty}^{n-1} W_j \tag{D.12}$$

$$L^2(\mathbb{R}) = \bigoplus_{j\in\mathbb{Z}} W_j \tag{D.13}$$

Les espaces W_j étant mutuellement orthogonaux, la famille d'ondelettes $(\psi_{j,k})_{j,k\in\mathbb{Z}}$ forme une base orthonormée de $L^2(\mathbb{R})$. Connaissant les cœfficients de détails $w_f(j,k)$ il est par conséquent possible de reconstruire fidèlement la fonction f. L'expression de la transformation en ondelettes inverse est alors la suivante :

$$f(t) = \sum_{j\in\mathbb{Z}} \sum_{k\in\mathbb{Z}} w_f(j,k)\psi_{j,k}(t) \tag{D.14}$$

D.2.1 Transformation en ondelettes et décomposition en sous-bandes

Un espace d'approximation à un niveau de résolution donnée peut être considéré comme l'ensemble des fonctions dont les composantes fréquentielles sont confinées dans un domaine de type passe-bas $B_{V_j} = [0; F_{max\,V_j}]$.

D'après la relation D.2, le passage d'un espace V_{j+1} à l'espace d'approximation de niveau immédiatement inférieur correspond à une contraction de la bande passante par un facteur 2 :

$$F_{max\,V_j} = \frac{F_{max\,V_{j+1}}}{2} \tag{D.15}$$

L'espace de détail correspondant sera quant à lui associé à un domaine de type passe-bande correspondant à la bande fréquentielle complémentaire :

$$\begin{aligned} F_{min\,W_j} &= \frac{F_{max\,V_{j+1}}}{2} \\ F_{max\,W_j} &= F_{max\,V_{j+1}} \end{aligned} \tag{D.16}$$

La décomposition de f en une somme de fonctions de détails aux différents niveaux de résolution s'apparente ainsi à isoler les composantes fréquentielles de f selon une partition de $L^2(\mathbb{R})$ en une suite de sous-bandes complémentaires.

D.2.2 La transformation en ondelettes bidimensionnelle

Nous nous intéressons ici au cas de la transformation en ondelettes bidimensionnelle séparable. Etant donnée une fonction f de deux variables x et y, une telle transformation revient à appliquer successivement la transformation en ondelettes monodimensionnelle (équation D.11) selon la variable x puis selon la variable y.

La fonction d'échelle et les ondelettes-mères bidimensionnelles sont obtenues en effectuant le produit tensoriel de leurs versions monodimensionnelles :

$$\begin{aligned} \begin{pmatrix} \varphi(y) \\ \psi(y) \end{pmatrix} \times \begin{pmatrix} \varphi(x) & \psi(x) \end{pmatrix} &= \begin{pmatrix} \varphi(x)\varphi(y) & \psi(x)\varphi(y) \\ \varphi(x)\psi(y) & \psi(x)\psi(y) \end{pmatrix} \\ &= \begin{pmatrix} \varphi(x,y) & \psi^{Vert}(x,y) \\ \psi^{Horiz}(x,y) & \psi^{Diag}(x,y) \end{pmatrix} \end{aligned} \tag{D.17}$$

Chaque espace d'approximation à un niveau donné se décompose par conséquent en quatre sous-espaces de niveaux inférieur, comprenant un espace d'approximation et trois espaces de détail – détails horizontaux avec $\psi^{Horiz}(x,y)$, verticaux avec $\psi^{Vert}(x,y)$ et diagonaux avec $\psi^{Diag}(x,y)$.

D.3 Transformation en ondelettes et compression

L'efficacité d'un décorrélateur sur une image – ou une classe d'images – donnée dépend essentiellement de l'adéquation entre l'image et le modèle implicite sur lequel se fonde le décorrélateur. Les images naturelles présentant une forte redondance spatiale, on s'attend généralement à ce qu'une image décorrélée contienne un nombre réduit de valeurs significatives.

La transformation en ondelette combine les avantages des méthodes de décorrélation dans le domaine spatial – par exemple le codage prédictif – et des méthodes de projection dans le domaine fréquentiel. Une base d'ondelettes tient compte de la variété de résolutions sur lesquelles se répartissent les détails d'une image. Le choix d'une ondelette-mère avec une décroissance suffisamment rapide permet de conserver une information précise sur la localisation spatiales des détails. Ainsi, à la différence de la *DCT* employée dans le standard *JPEG*, cette décomposition espace/résolution ne nécessite pas de partitionner l'image en blocs.

Munie de ces propriétés, la transformation en ondelettes a démontré dans le cas des images naturelles un pouvoir de décorrélation supérieur aux autres méthodes citées. Ces images présentant peu de détails fins, les cœfficients significatifs de leur transformée en ondelette seront concentrés dans les sous-bandes de basse résolution.

La représentation dans le domaine espace/résolution a été mise à profit dans JPEG2000 afin de contrôler plus finement les étapes de quantification et de codage. Par exemple, une loi de quantification plus forte peut être appliquée dans les sous-bandes de haute résolution, où l'œil humain est moins sensible aux dégradations. De plus, la connaissance de la localisation spatiale des détails permet d'adopter une loi de quantification différente pour les détails relatifs à une région d'intérêt. Enfin, différents choix d'ordre de codage des cœfficients d'ondelette permettent d'envisager différents modes de reconstruction progressive : région par région, résolution par résolution, *etc.*

Méthodologie de Conception de Composants Virtuels Comportementaux pour Une Chaîne de Traitement du Signal Embarquée

Résumé – Les futures générations de satellites d'observation de la Terre doivent concilier des besoins croissants en résolution, précision et qualité des images avec un coût élevé de stockage des données à bord et une bande passante limitée des canaux de transmission. Ces contraintes imposent de recourir à de nouvelles techniques de compression des images parmi lesquelles le standard *JPEG2000* est un candidat prometteur.

Face à la complexité croissante des applications et des technologies, et aux fortes contraintes d'intégration – faible encombrement, faible consommation, tolérance aux radiations, traitement des informations en temps réel – les outils et méthodologies de conception et de vérification classiques apparaissent inadaptés à la réalisation des systèmes embarqués dans des délais raisonnables. Les nouvelles approches envisagées reposent sur une élévation du niveau d'abstraction de la spécification d'un système et sur la réutilisation de composants matériels pré-définis et pré-vérifiés (*composants virtuels*, ou blocs *IP* pour *Intellectual Property*).

Dans cette thèse, nous nous intéressons à la conception de composants matériels réutilisables pour des applications intégrant des fonctions de traitement du signal et de l'image. Notre travail a ainsi consisté à définir une méthodologie de conception de composants virtuels hautement flexibles décrits au niveau comportemental et orientés vers les outils de synthèse de haut niveau. Nous avons expérimenté notre méthodologie sur l'implantation sous forme d'un composant virtuel comportemental d'un algorithme de transformation en ondelettes bidimensionnelle pour la compression d'images au format *JPEG2000*.

Mots-clés – Composant virtuel (IP) ; synthèse de haut niveau ; transformation en ondelette discrète ; JPEG2000.

Behavioral Virtual Component Design Methodology for Onboard Signal Processing

Abstract – Future satellites for earth observation will have to fulfill growing needs in image resolution, precision and quality. New image compression techniques are needed in order to overcome the heavy cost of storage device and the limited communication bandwidths : among the considered techniques, the *JPEG2000* standard offers a promising compression scheme.

The increasing complexity of applications and technologies, together with the heavy integration constraints – small dimensions, low power consumption, radiation tolerance, real-time data processing – sets a challenge to the design and verification methodologies for onboard embedded systems. Current trends consider raising the abstraction level of system specification and reusing pre-designed, pre-verified hardware components known as *virtual components*, or *Intellectual Property* (*IP*) cores.

We propose a new methodology for virtual component design targeting signal and image processing applications. Our approach aims at providing a framework for designing and synthesizing highly flexible *IP* cores through behavioral-level specification and High-Level Synthesis tools. Our methodology has been applied to the design of a Discrete Wavelet Transform virtual component for *JPEG2000* image compression.

Keywords – Virtual Component (IP core) ; High-Level Synthesis ; Discrete Wavelet Transform ; JPEG2000.